Rootkits for JavaScript Environments

Ben Adida
Harvard University
ben_adida@harvard.edu

Adam Barth
UC Berkeley
abarth@eecs.berkeley.edu

Collin Jackson
Stanford University
collinj@cs.stanford.edu

Abstract

A number of commercial cloud-based password managers use bookmarklets to automatically populate and submit login forms. Unfortunately, an attacker web site can maliciously alter the JavaScript environment and, when the login bookmarklet is invoked, steal the user's passwords. We describe general attack techniques for altering a bookmarklet's JavaScript environment and apply them to extracting passwords from six commercial password managers. Our proposed solution has been adopted by several of the commercial vendors.

1. Introduction

One promising direction for building engaging web experiences is to combine content and functionality from multiple sources, often called a "mashup." In a traditional mashup, an integrator combines gadgets (such as advertisements [1], maps [2], or contact lists [3]), but an increasingly popular mashup design involves the user adding a bookmarklet [4] (also known as a JavaScript bookmark or a favelet) to his or her bookmark bar. To activate the mashup, the user clicks the bookmarklet and the browser runs the JavaScript contained in the bookmarklet in the context of the current web page. This type of mashup is opportunistic: the bookmarklet's code runs within a web page it might not have encountered previously. Bookmarklets were popularized by the social bookmarking site Delicious, whose bookmarklet enables users to add the currently viewed site to their Delicious feed, URL and title included, with just one click. Bookmarklets offer advantages over browser plug-ins [5], [6]: they are easier to develop, easier to install, and work with every browser. Many other web sites have published bookmarklets, including Google, Yahoo!, Microsoft Windows Live, Facebook, Reddit, WordPress, and Ask.com.

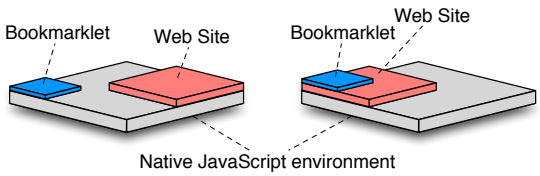

Figure 1. Developers assume the bookmarklet interacts with the native JavaScript environment directly (left). In fact, the bookmarklet's environment can be manipulated by the current web page (right).

If the user clicks a bookmarklet while visiting an untrusted web page, the bookmarklet's JavaScript is run in the context of the malicious page, potentially letting an attacker manipulate its execution by carefully crafting its JavaScript environment, essentially installing a "rootkit" in its own JavaScript environment (See Figure 1). Instead of interacting with the native JavaScript objects, the bookmarklet interacts with the attacker's objects. This attack vector is not much of a concern for Delicious' social bookmarking service, because the site's own interests are served by advertising its true location and title. However, these rootkits are a serious concern for more advanced bookmarklets, where the malicious web site might benefit from subverting the bookmarklet's functionality. In particular, these rootkits are a problematic for login bookmarklets.

By using bookmarklets, a server-based password manager can enable one-click sign-on at each of the user's favorite sites, synchronization of passwords across computers, and automatic generation of strong passwords, features that are not found in typical browser-built-in password managers. To automatically log the user into the current site, the login bookmarklet must make a critical security decision about *which* password to use. If the bookmarklet supplies the wrong password, an attacker might be able to trick the bookmarklet into revealing the user's online banking password.

A traditional rootkit [7], [8] modifies the user-program-accessible behavior of the operating system and escapes detection by interception of the operating system's reflection APIs, for example removing itself from the operating system's list of running processes. Analogously, a JavaScript rootkit modifies the bookmarklet-visible behavior of a JavaScript environment and escapes detection by overriding the native JavaScript objects. Because bookmarklets are invoked once a web page is already loaded, a malicious web page is effectively a rootkit to the bookmarklet's script.

We present attack techniques to corrupt a JavaScript environment in today's unmodified browsers. The simplest technique shadows a native JavaScript object, while the more complex uses explicit JavaScript features [9] to defeat an advanced bookmarklet's reflection-based attempts at detecting our rootkit, and even to extract the bookmarklet's full source code, which sometimes reveals inline secrets. We note that the cat-and-mouse game between a malicious page cloaking its interposed objects and a bookmarklet attempting to detect them unfortunately favors the malicious page, which is loaded first.

We examined six commercially available login bookmarklets. Using our JavaScript rootkit techniques, an attacker can fully compromise all six systems and extract all of the user's passwords if the user attempts to use his or her bookmarklet when visiting the attacker's web site. In four of the systems, the attacker can construct a rootkit that fools the bookmarklet into revealing all of the user's passwords with a single click. In the remaining two cases, the attacker's rootkit cannot extract the passwords directly (because the passwords are never present in the attacker's JavaScript environment), but the attacker can corrupt the bookmarklet's cross-site scripting filter, which does run in the attacker's JavaScript environment, eventually leading to the full compromise of the user's password store.

We propose a secure design for login bookmarklets that does not rely on the untrusted JavaScript environment for security. We avoid the cat-and-mouse game between the bookmarklet author and the attacker's rootkit. Our proposed solution is compatible with web browsers that comprise 99% of the market and, as a result of our disclosures, has been adopted by several of the login bookmarklet vendors.

2. Threat Model

The attacker's goal is to learn the user's password for an honest site, e.g., bank.com. We consider the web attacker threat model [10], with a properly enforced same-origin policy that isolates web pages that the browser retrieved from different origins [11]. In particular, we assume the attacker owns an untrusted domain name, e.g. attacker.com, operates a web server, and possesses a valid HTTPS certificate for attacker.com. We assume the user visits attacker.com where she invokes her login bookmarklet, but we do not assume that the user mistakes attacker.com for another web site. If the login bookmarklet process normally prompts the user for a master password, we assume the user enters it at the appropriate time in the login process. We argue this threat model is appropriate for password managers because their purpose is to help the user authenticate to multiple sites without revealing a common password to all the sites.

3. Attack Techniques

The browser's security model does not erect trust boundaries within a single JavaScript environment. Therefore, the attacker can employ a number of techniques to disrupt the integrity or confidentiality of trusted JavaScript running in an untrusted JavaScript environment. For example, the attacker can shadow the names of native objects and emulate their behavior, or the attacker can alter the semantics of built-in types by altering their prototype objects. By applying these techniques to the JavaScript's reflection API, the attacker can hide these modifications from a bookmarklet that attempts to use introspection to uncover the rootkit.

Typically, the attacker modifies the JavaScript environment by running malicious JavaScript while the web page is loading, before the honest JavaScript runs. None of the techniques presented in this section escape the JavaScript sandbox, disrupt JavaScript running in other security origins, or compromise the user's operating system. Not all of these techniques work in all browsers, but many of the techniques work in many of the major browsers.

Shadowing. An attacker can replace native JavaScript objects in the global scope by declaring global variables with the same name as the native objects. For example, suppose the bookmarklet used the following code to determine whether it was running on bank.com:

```
if (window.location.host == "bank.com")
  doLogin(password);
```

In Internet Explorer and Google Chrome, the attacker can disrupt the integrity of this computation by declaring a global variable named `window` whose value is an object with a fake `location` object:

```
var window = {
  location: { host: "bank.com" } };
```

When the bookmarklet tries to read the `host` property of the native `location` object, the bookmarklet instead reads the `host` property of the fake `location` object created by the attacker. Notice that the attacker cannot modify the native `location` object directly because assigning to the native `location` object navigates the current frame to that location.

Browsers let web pages override native objects to help with compatibility. For example, a pre-release version of Firefox introduced an immutable `JSON` object into the global scope, but this change broke a number of prominent web sites that implemented their own `JSON` object [12]. To avoid breaking these sites, Firefox changed the `JSON` object to be mutable (e.g., overwritable by attackers).

Emulation. By interacting with an object, the bookmarklet could hope to determine whether the object has been replaced by the attacker. For example, the native `location` object behaves differently than a shadowing object if the bookmarklet assigns to `window.location`. Consider the value of `window.location` after running the following:

```
window.location = "#";
```

If `window.location` is the native object, the value will be `"https://bank.com/login#"` after the assignment. However, if `window.location` is an object shadowing the real location object, the value will be `"#"`. Unfortunately, the attacker can emulate the behavior of native objects using getters and setters.

Most JavaScript implementations, including Internet Explorer 8, Firefox 3, Safari 3.1, Google Chrome, and Opera 9.5, support "getter" and "setter" properties. If an object `obj` has a getter for property `p`, the expression `obj.p` results in calling the getter function and using the return value of the function as the value of the expression. Similarly, if `obj` has a setter for `p`, the assignment `obj.p = x` calls the setter function with the value of `x` as an argument.

Browsers implement getters and setters to let authors of JavaScript libraries extend native objects to emulate features available in other browsers. For example, Internet Explorer supports the non-standard `document.all` property. To emulate this features in other browsers, some JavaScript libraries install a getter for the `all` property of `document`. Getters and setters are also used by JavaScript libraries to provide more convenient DOM-like APIs to web developers.

Using getters and setters, the attacker can emulate the behavior of native objects. For example, in Safari 3.1, Google Chrome, and Internet Explorer 8, the attacker can emulate the native `location` object:

```
window.__defineGetter__("location",
  function () {
    return "https://bank.com/login#";
  });
window.__defineSetter__("location",
  function (v) { });
```

Because the attacker can construct the malicious JavaScript after seeing the bookmarklet, the attacker need only emulate the native objects to the extent necessary to fool the bookmarklet.

Prototype poisoning. An attacker can also alter the behavior of native objects, such as strings, functions, and regular expressions, by manipulating the prototype of these objects. For example, suppose the bookmarklet attempts to ensure that a URL begins with either `http` or `https` using the following regular expression:

```
(/^https?:/i).exec(url)
```

The attacker can compromise the integrity of this test by modifying the prototype of regular expressions:

```
RegExp.prototype.exec =
  function () { return true; }
```

Instead of calling the native `exec` function, the bookmarklet will instead call the attacker's function, which erroneously reports that `javascript:doAttack()` is a URL that begins with `http` or `https`.

Reflection. The bookmarklet author could hope to detect emulation or prototype poisoning using JavaScript's reflection APIs. For example, most JavaScript implementations provide a native method called `__lookupGetter__` that can be used to determine whether a property has been replaced by a getter. Similarly, the bookmarklet author could hope to use a function's `toString` property to determine if a function has been replaced by the attacker. For example, the bookmarklet might test for the existence of a getter using the following code:

```
typeof obj.__lookupGetter__(
  propertyName) !== "undefined"
```

Notice that this code carefully uses `!==` instead of `!=` to avoid the implicit call to `valueOf` and uses `typeof` instead of comparing directly against `undefined` because the attacker can redefine `undefined` to a value of the attacker's choice [13].

Unfortunately, the attacker can defeat even this carefully crafted defense by emulating the reflection API itself. In particular, the attacker can replace `__lookupGetter__` by altering `Object.prototype`:

```
Object.prototype.__lookupGetter__ =
  function() { ... };
```

Similarly, the attacker can alter the `toString` method of every function by poisoning the `Function.prototype` object.

Global variable hijacking. Bookmarklets cannot rely on the confidentiality or integrity of global variables because those variables can be controlled by getters and setters. For example, suppose a bookmarklet contained the following statement: `var x = "#";`. An attacker can subvert the integrity of `x` by adding a getter and a setter for `x` in the `window` object:

```
window.__defineGetter__(
    "x", function () { ... });
window.__defineSetter__(
    "x", function (v) { ... });
```

Instead of acting like a normal variable, `x` now behaves according to the functions defined by the attacker. The bookmarklet author can defend against this attack by never using any variables or by wrapping the bookmarklet inside an anonymous function:

```
(function () { ... code ... })();
```

Notice that this function must be anonymous because otherwise the attacker could install a setter for the name of the function and read its source code by calling the function's `toString()` method.

Caller. Wrapping the bookmarklet code in an anonymous function can lead to another vulnerability. If the bookmarklet ever calls a function defined by the attacker, the attacker can easily obtain a pointer to the anonymous function by checking the call stack, and can then read the function's source code by calling its `toString()` method. For example, suppose the bookmarklet contains the following code:

```
(function () {
... if (obj == "bank.com") ... })();
```

Before comparing objects, the == operator implicitly calls the `valueOf` method of each object, a function which the attacker now controls. The attacker can then extract the bookmarklet's secrets in Internet Explorer, Firefox 3, Safari 3.1, and Chrome. (The == operator is analogous to `equals?` function in LISP, which can be overridden by the objects being compared. The === operator, analogous to the `eq?` function in LISP, is more predictable and (roughly) compares the identities of the objects.) If `obj` is an object defined by the attack, the attacker can read the entire source code of the bookmarklet, including any passwords or keys stored in the bookmarklet, by defining a `valueOf` method on `obj` as follows:

```
functon f() {
  ... f.caller.toString() ...}
var obj = { valueOf: f };
```

The bookmarklet author can defend against this technique by never calling an untrusted function, but that goal is difficult to achieve, especially if the browser supports getters and setters. In most browsers, every property read or write could potentially be a function call that leaks the bookmarklet's secrets.

4. Case Studies

To evaluate whether these attack techniques are sufficient to defeat real-world implementations of login bookmarklets, we examine six commercially available bookmarklet-based password managers. We find that all six systems are vulnerable to our attacks. Four of the systems rely on the integrity of native JavaScript objects to protect the user's site-specific passwords. The remaining two systems run cross-site scripting filters in the attacker's JavaScript environment, where they can be corrupted using our attack techniques.

The commercial systems we examine are significantly more complex than the prototype bookmarklets we constructed in our laboratory. For example, the systems often have a substantial server-side component that lets users manage their passwords via their browsers and several contain a master password so user's can log into their password manager from multiple machines. The bookmarklets themselves often retrieve additional JavaScript code from the server to implement encryption or site-specific form filling functions. However, even with all these additional layers of complexity, our attack techniques apply directly.

VeriSign. The first system we examine is VeriSign's One-Click Sign-In Bookmark [14]. To extract the user's Facebook password, for example, the attack need only shadow the global `location` object:

```
var location = {
  hostname: "www.facebook.com",
  href: "http://www.facebook.com/" };
```

Alternately, the attacker can define a getter for `window.location`:

```
window.__defineGetter__(
  "location", function() {
    return {
      hostname: "www.facebook.com",
      href: "http://www.facebook.com/"
    };});
```

A variant on this attack extracts all of the user's site-specific passwords at once, rather than one per click. We reported this vulnerability to VeriSign on 10 October 2008, and VeriSign implemented and deployed the secure design we suggest in Section 5.

MashedLife. MashedLife is a bookmarklet-based password manager. The attacker can extract the user's bank password by poisoning the `String.prototype` object:

```
String.prototype.toLowerCase =
  function() { return "bank.com"; };
```

This attack works because the bookmarklet converts the site's host name to lower case before determining which password to use. We reported this vulnerability to MashedLife on 10 October 2008 and suggested the secure design we detail in Section 5. MashedLife deployed our design on 2 November 2008.

Passpack. Passpack is a bookmarklet-based password manager. Passpack's bookmarklet calls the `toString` method on the current page's location, letting the attacker extract the user's Delicious password, for example, by poisoning the `String.prototype` object:

```
String.prototype.toString =
  function() {
    return "https://delicious.com"; };
```

By itself, this attack is not sufficient because Passpack also validates the HTTP `Referer` header. However, they implement lenient `Referer` validation [15] (they accept requests that lack a `Referer` header). By using one of a number of `Referer` suppression techniques, the attacker can extract the user's passwords.

We reported this vulnerability to Passpack on 10 October 2008, and Passpack implemented and deployed the secure design we suggest in Section 5 within 20 minutes. Passpack has also published a full description of the vulnerability and their patch [16].

1Password. 1Password, a password manager for Mac-OS-X-based browsers, includes a bookmarklet-based component designed for the iPhone. Unlike the three systems described above, the 1Password bookmarklet does not contact its server during the login process. Instead, 1Password stores all of the user's site-specific passwords within the bookmarklet itself. When the user clicks his or her bookmarklet, the encrypted passwords are stored in a global variable named `database`. Using the global variable technique, the attacker can extract all of the user's encrypted passwords at once:

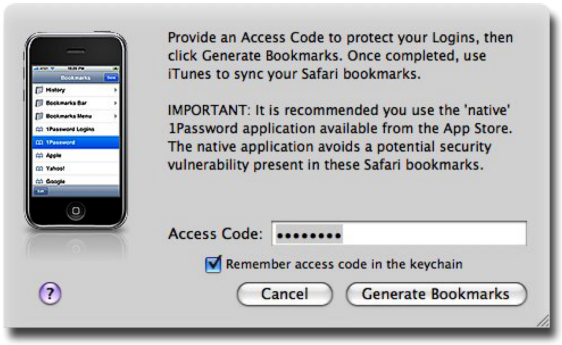

Figure 2. 1Password's iPhone bookmarklet leaks passwords to web sites, so the setup dialog recommends using the native application instead.

```
window.__defineSetter__("database",
  function (v) { ... });
```

Then, the 1Password bookmarklet prompts the user for a master password by displaying a form within the current page. This master password is used as a decryption key for the password database. The attacker can override the event handler that the bookmarklet calls after the user types their master password and decrypt every password in the database using the `decryptEntry` function defined by the bookmarklet.

We reported this vulnerability to 1Password on 12 October 2008. 1Password has acknowledged that their bookmarklet is exploitable as we describe, but has decided not to repair the vulnerability. A dialog displayed during setup recommends using the 1Password iPhone application instead (See Figure 2).

Clipperz. Unlike the above four bookmarklet-based password managers, Clipperz's bookmarklet does not fill out the site's password form. Instead, the bookmarklet helps the user register a new site with the Clipperz service. Upon being clicked, the bookmarklet serializes the contents of the current page's login form and displays the raw data in current page. The bookmarklet then instructs the user to copy and paste this information into the Clipperz web site. When the user wants to log in to the site using Clipperz, Clipperz uses JavaScript to submit a form with the recorded login credentials to the target site, essentially mounting a login cross-site request forgery attack [15] against the target site. Only sites that are vulnerable to login cross-site request forgery can be used with Clipperz.

Because Clipperz creates the cross-site login form on the clipperz.com domain, Clipperz sanitizes the form parameters to prevent cross-site scripting attacks. This sanitization is performed by the bookmarklet during the registration process, but the attacker can circumvent the filter by poisoning the `Array.prototype` object:

```
Array.prototype.join = function() {
  ... return evilString; }
```

We reported this vulnerability to Clipperz on 13 October 2008, recommending that they perform the sanitization in the clipperz.com security context. Clipperz patched this vulnerability on 17 October 2008.

MyVidoop. MyVidoop uses a design similar to Clipperz, but instead of relying on the user to paste the serialized form details into the password manager site, MyVidoop's bookmarklet sends them to myvidoop.com via an HTTP request. As with Clipperz, MyVidoop relies on the bookmarklet to run a cross-site scripting filter in the attacker's JavaScript environment. The attacker can evade the filter by poisoning the `RegExp.prototype` object:

```
RegExp.prototype.exec =
  function () { return true; }
```

To defend against this attack, MyVidoop should filter cross-site scripting attacks on the server. We reported this vulnerability to MyVidoop on 12 October 2008. MyVidoop patched the issue on 13 October 2008.

5. Defenses

The login bookmarklets we examine are vulnerable to attack because they rely on an untrusted JavaScript environment. To defend against these attacks, a bookmarklet requires confidential storage for secrets and a mechanism to authenticate the site before delivering the user's site-specific password.

Instead of storing the user's passwords in the bookmarklet, the password manager can store a short *master secret* in a `Secure` cookie for pwdmngr.com. The browser's security policy prevents the attacker's web site from reading the contents of these cookies. To read the master secret, the bookmarklet can initiate a network request to https://pwdmngr.com by adding a `<script>` tag to the current page. The password manager can then use the master secret as a key to decrypt the site-specific passwords stored on the server.

The bookmarklet must authenticate which web site receives the password. We recommend using the HTTP `Referer` header [17] for authentication. When the bookmarklet issues a network request to pwdmngr.com, the password manager server should disclose the user's bank.com password only if the `Referer` header begins with the string `https://bank.com/`. If the `Referer` header is absent or contains an unexpected value, the password manager should not disclose the user's site-specific password. Although some network operators suppress the header, the `Referer` header is present 99.9% of the time over HTTPS [15], and the password manager should be using HTTPS for transferring the user's passwords over the network.

We propose combining these browser security features to implement a secure login bookmarklet. In our design, the bookmarklet itself does not store any secrets and is identical for all users. The bookmarklet simply inserts a `<script>` tag into the current document that requests a script from pwdmngr.com over HTTPS. The password manager's server then validate the master secret and `Referer` header.

If the master secret is missing (perhaps because it was evicted or was deleted), the script navigates the user to https://pwdmngr.com/login, where the user can enter his or her master passphrase. (In particular, the password manager must not prompt the user for the passphrase in a window under the control of the attacker because the attacker can steal the user's master passphrase by drawing a malicious password entry control on top of the honest control.)

6. Conclusion

Bookmarklets provide web developers a simple and rapidly deployable mechanism for interacting with third-party content. However, bookmarklets run in the JavaScript environment of the third-party page, which might have been manipulated by the page. Because JavaScript environments are so pliable, an attacker can exploit this plasticity to replace many of the native JavaScript objects with replicas that behave maliciously. These simulated environments are analogous to rootkits in conventional operating systems.

We examined six commercial login bookmarklets that handle sensitive information, such as passwords. We discover that an attacker can steal the user's password from all six systems Although the concrete attacks we demonstrate are simple, often only one or two lines of JavaScript, we describe general techniques for mounting more sophisticated attacks. Our proposed solution has been adopted by three of the six vendors.

References

[1] Google AdSense, https://www.google.com/adsense/.

[2] Google Maps, http://maps.google.com/.

[3] Microsoft, Windows Live contacts control (beta), http://dev.live.com/contactscontrol/.

[4] S. Kangas, About bookmarklets, http://www.bookmarklets.com/about/.

[5] M. Wu, R. C. Miller, and G. Little, Web Wallet: Preventing phishing attacks by revealing user intentions, in *Proc. of the 2006 Symposium On Usable Privacy and Security (SOUPS '06)*.

[6] K.-P. Yee and K. Sitaker, Passpet: Convenient password management and phishing protection, in *Proc. of the 2006 Symposium On Usable Privacy and Security (SOUPS '06)*.

[7] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch, SubVirt: Implementing malware with virtual machines, in *Proc. of the 2006 IEEE Symposium on Security and Privacy*.

[8] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, Cloaker: Hardware supported rootkit concealment, in *Proc. of the 2008 IEEE Symposium on Security and Privacy*.

[9] D. Crockford *et al.*, ECMAScript 3.1 goals, February 2008, http://wiki.ecmascript.org/doku.php?id=es3.1:es3.1_goals.

[10] A. Barth, C. Jackson, and J. C. Mitchell, Securing frame communication in browsers, in *In Proceedings of the 17th USENIX Security Symposium (USENIX Security 2008)*.

[11] J. Ruderman, JavaScript Security: Same Origin, http://www.mozilla.org/projects/security/components/same-origin.html.

[12] R. Sayre *et al.*, Sites not loading—redeclaration const JSON error on console, https://bugzilla.mozilla.org/show_bug.cgi?id=459293.

[13] S. Maffeis, J. C. Mitchell, and A. Taly, An operational semantics for JavaScript, in *Proc. of the Sixth ASIAN Symposium on Programming Languages and Systems (APLAS 2008)*.

[14] VeriSign Inc., Personal identity protector - using one-click sign-in, https://pip.verisignlabs.com/1click_howtouse.do.

[15] A. Barth, C. Jackson, and J. C. Mitchell, Robust defenses for cross-site request forgery, in *To appear at the 15th ACM Conference on Computer and Communications Security (CCS 2008)*.

[16] Passpack, Fixed: 1 click login button, October 2008, http://passpack.wordpress.com/2008/10/11/reinstall-1click-login-button/.

[17] R. Fielding *et al.*, Hypertext transfer protocol – http/1.1, http://www.ietf.org/rfc/rfc2616.txt.