Extensible Web Browser Security

Mike Ter Louw, Jin Soon Lim, and V.N. Venkatakrishnan

Department of Computer Science,
University of Illinois at Chicago
mter,jlim,venkat@cs.uic.edu

Abstract. In this paper we examine the security issues in functionality extension mechanisms supported by web browsers. Extensions (or "plug-ins") in modern web browsers enjoy unlimited power without restraint and thus are attractive vectors for malware. To solidify the claim, we take on the role of malware writers looking to assume control of a user's browser space. We have taken advantage of the lack of security mechanisms for browser extensions and have implemented a piece of malware for the popular Firefox web browser, which we call BROWSER-SPY, that requires no special privileges to be installed. Once installed, BROWSER-SPY takes complete control of a user's browser space and can observe all the activity performed through the browser while being undetectable. We then adopt the role of defenders to discuss defense strategies against such malware. Our primary contribution is a mechanism that uses code integrity checking techniques to control the extension installation and loading process. We also discuss techniques for runtime monitoring of extension behavior that provide a foundation for defending threats due to installed extensions.

1 Introduction

The Internet web browser, arguably the most commonly used application on a network connected computer, is becoming an increasingly capable and important platform for millions of today's computer users. The web browser is often a user's window to the world, providing them an interface to perform a wide range of activity including email correspondence, shopping, social networking, personal finance management, and professional business.

This usage gives the browser a unique perspective; it can observe and apply contextual meaning to sensitive information provided by the the user during very personal activities. Furthermore, the browser has access to this information *in the clear*, even when the user encrypts all incoming and outgoing communication. This high level of access to sensitive, personal data warrants efforts to ensure its complete confidentiality and integrity.

Ensuring that the entire code base of a browser addresses the security concerns of confidentiality and integrity is a daunting task. For instance, the current distribution of the Mozilla Firefox browser has a build size of 3.7 million lines of code (as measured using the `kloc` tool) written in a variety of languages that include C, C++, Java, JavaScript and XML. These challenges of size and implementation language diversity make it difficult to develop a “one-stop shop” solution for this problem. In this paper, we focus on the equally significant sub-problem of ensuring confidentiality and integrity in a browser in the presence of browser extensions. We discuss this problem in the context of Mozilla Firefox, the widely used free (open source) software browser, used by about 70 million web users [1].

Browser extensions (or “add-ons”) are facilities provided to customize the browser. These extensions make use of interfaces exported by the browser and other plug-ins to alter the browser’s behavior. Though the build of Firefox is platform-specific (such as one for Windows XP, Linux or Mac OS X), extensions are primarily platform-independent based on the neutral nature of JavaScript and XML, the predominant languages used to implement them.

Even though extensions plug directly into the browser, there is no provision currently in Firefox to provide protection against malicious extensions. One way to do this is to disallow extensions altogether. Firefox is able to do this when started in debugging mode, which prevents any extension code to be loaded. However, typical installation and execution in the normal mode allow extensions to be executed. Extensions offer useful functionality, as evidenced by the popularity of their download numbers [2], to several thousands of users who use them. Dismissing the security concerns about extensions by turning them off ignores the problem.

To understand the impact of running a malicious extension, we set for ourselves the goal of actually crafting one. Surprisingly, we engineered a malicious extension for the Firefox browser we call BROWSERSPY, with modest efforts and in less than three weeks. Once installed, this extension takes complete control of the browser. As further testimony, a recent attack was launched on the Firefox browser using a malware extension known as FormSpy [8], that elicited widespread media coverage and concern about naive users.

There are two main problems raised by the presence of our malware extension and the FormSpy extension:

- *Browser code base integrity* A malicious extension can compromise the integrity of the browser code base when it is installed and loaded. We demonstrate (by construction) that a malicious extension can subvert the installation process, take control of a browser, and hide its presence completely.
- *User data confidentiality and integrity* A malicious extension can read and write confidential data sent and received by the user, even over an encrypted secure connection. We demonstrate this by having our extension collect sensitive data input by a user while browsing and log it to a remote site.

In this paper we present techniques that address these problems. To address extension integrity, our solution empowers the end-user with complete control of the process by which code is selected to run as part of the browser, thereby disallowing installation integrity threats due to malware. This is done by a process of *user authorization* that detects and refuses to allow the execution of extensions that are not authorized by the end user.

To address the second challenge of data confidentiality and integrity, we augment the browser with support for policy-based monitoring of extensions by interposition mechanisms retrofitted to the Spidermonkey JavaScript engine and other means (Section 5).

A key benefit of our solution is that it is targeted to *retrofit* the browser. We consider this benefit very important, and have traded off potentially better solutions to achieve this benefit. Other benefits of our approach are that it is convenient, user-friendly and poses very acceptable overheads. Our implementation is robust, having been tested with several Firefox extensions.

This paper is organized as follows. A discussion of related work appears in Section 2. We present the main details behind our malware extension in Section 3. We present our solution to the extension integrity problem in Section 4 and address data confidentiality in Section 5. We evaluate these approaches with several Firefox add-ons and discuss their performance in the above sections individually. In Section 6 we conclude.

2 Related work

We examined extension support in four contemporary browsers: Firefox, Internet Explorer (IE), Safari and Opera. Among the four browsers that we studied, only the Safari browser does not support the concept of extensions. The remaining three possess extensible architecture but do not have security mechanisms addressing extension-based threats. For instance, IE primary extension mechanism is through Browser Helper Objects (BHO). The PestPatrol malware detection website lists hundreds of malware that use BHOs [5]. Furthermore, the integrity and confidentiality of the end-user's private data used in the browser is also not addressed in recent mechanisms such as "protected-browser-mode" [4] in Windows Vista.

The problem of safely running extensions in a browser is in many ways similar to the problem of executing downloaded untrusted code in an operating system. This is a well known problem, and has propelled research in ideas such as signed code, static analysis, proof-carrying code, model-carrying code and several execution monitoring approaches. Below, we discuss the applicability of these solutions to the browser extension problem highlighting several technical and practical reasons.

Signed code The Firefox browser provides support for signed extensions; however, this is hardly used in practice. A search of extensions in the Firefox extensions repository `addons.mozilla.org` revealed several thousand unsigned extensions and only two that were signed. In addition, we note that signed extensions merely offer a first level of security. They only guarantee that they are from the browser distribution site and are unmodified in transit; no assurance is provided regarding the security implications of running the extension.

Static analysis A very desirable approach for enforcing policies on extension code is by use of static analysis. Static analysis has been employed in several past efforts in identifying vulnerabilities or malicious intent. The primary advantages of using static analysis are the absence of execution overhead and runtime aborts, which are typical of dynamic analysis based solutions.

It is difficult to employ static analysis for JavaScript code without making conservative assumptions, however. A first example is the *eval* statement in JavaScript that allows a string to be interpreted as executable code. Without knowing the runtime values of the arguments to the *eval* statement, it is extremely difficult—if not impossible—to determine the runtime actions of the script. Another problem is tracing the flow of object references in a prototype-based object oriented language such as JavaScript. For instance, variable assignment to or from an array element or object property (when the object is indexed as an associative array) can decisively hamper the tracking of object reference flow as references are stored or retrieved.

Consequently, recent efforts that trace JavaScript code [11] use runtime approaches to track references. An exception is [13] that employs static analysis for JavaScript for detecting *cross-site scripting* (XSS) attacks. In their approach, scenarios like the above are handled by a conservative form of tainting. This is suitable for their purpose of preventing XSS attacks as evidenced by their experimental results, and the fact that typical scripts from web pages are not expected to have complex *eval* constructs. This approach is unsuitable for statically analyzing extension code in JavaScript, however. Almost half (45%) of the extensions that we tested make heavy use of complex *eval* constructs, while all generously use objects as associative arrays, making static analysis very hard.

PCC and MCC The difficulties for static analysis make frameworks such as proof-carrying code (PCC) [10] unsuitable for this problem. It will be difficult to produce proofs for extensions that make heavy use of constructs such as *eval* as part of their code. The typical approach to employ PCC in scenarios that require runtime data is to: (a) transform the original script with runtime checks that enforce the desired security property, and (b) produce a proof that the transformed program respects this property. The proof in this case is primarily used to demonstrate the correctness of the placement of runtime checks.

In the browser situation, transformation needs to be made before all *eval* statements. Policy enforcement would still be carried out by runtime checks, and therefore we did not adopt this route of using PCC. Another solution is model-carrying-code [12] which employs runtime techniques to learn the behavior of code that will be downloaded. The difficulty in using this approach is in obtaining test suites for exhaustive code coverage required for approaches based on runtime learning of models.

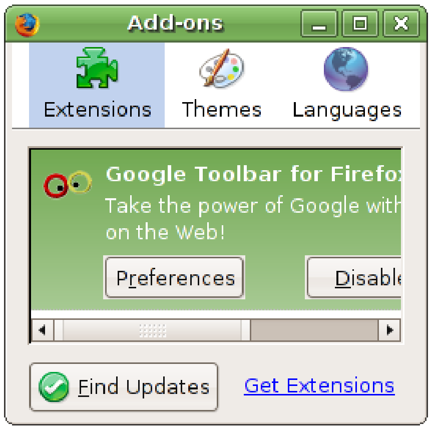
Execution monitoring Several execution monitoring techniques [14, 6, 7] have previously looked at the problem of safely executing malicious code. The closest related project to our approach is by Hallaraker and Vigna [7]. This was the first work that looked at the security issues of executing malicious code in a large mainstream browser. Their focus is on protection against pages with malicious content rather than the ensuring the integrity of a browser's internal operations. For them it is not necessary to address the problem of browser code integrity, as scripts from web pages are sandboxed to prevent them from performing sensitive actions. In contrast we address the extension installation integrity problem, as extension code is unmonitored and can perform many sensitive operations.

To effectively regulate extension behavior, a runtime monitor must be able to determine the particular extension responsible for each operation. A direct adaptation of their execution monitoring approach does not provide this ability, and is therefore not suited for runtime supervision of extensions. To fill this void we describe two new *action attribution* mechanisms making use of browser facilities and JavaScript interposition in Section 5.

3 A Malware Extension

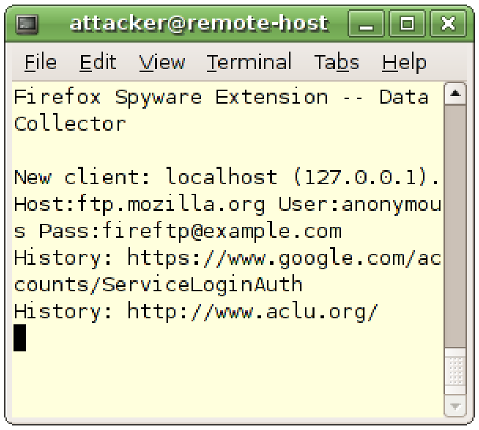
To gain a better understanding of the threat posed by a malware extension, we set ourselves the task of actually writing one. The motivations for creating the malicious software are to: (a) help us identify the scope of threats malicious extensions pose by understanding the facilities available to an extension in a browser, (b) increase our understanding of architecture-level and implementation-level weaknesses in the browser's extension manager, (c) give us a practical estimate in understanding the ease with which malware writers may be able to craft such extensions, and (d) provide a concrete implementation of a malicious extension to serve as a benchmark for malware analysis.

Extension Capabilities BROWSERSPY, the extension we authored, is capable of harvesting every piece of form data (e.g., passwords) submitted by the user, including those sent over encrypted connections. Furthermore, once the extension enters the system, it ensures that it remains undetectable by users (Figure 1 (a)).



Google Toolbar for Firefo
Take the power of Google with
on the Web!

(a) Extension hiding from the browser UI.



```
Firefox Spyware Extension -- Data
Collector

New client: localhost (127.0.0.1).
Host:ftp.mozilla.org User:anonymou
s Pass:fireftp@example.com
History: https://www.google.com/ac
counts/ServiceLoginAuth
History: http://www.aclu.org/
```

(b) Data collector receiving sensitive information.

Fig. 1. Two views of the BROWSERSPY extension in operation.

Once BROWSERSPY is installed, it begins collection of personal data that will ultimately fall into the hands of an attacker. As a user navigates the Internet, BROWSERSPY harvests the URLs comprising their browsing history and stores them in a cache. Any username and password pairs that are stored in Firefox's built-in password manager are retrieved, along with the URL of the site they pertain to. Form data that the user submits finds its way into the extension as well. All of this information is stored and periodically sent over the network to a remote host.

Given enough data the spy can effectively steal the identity of the person using the browser. Intercepted form fields can give an attacker credit card numbers, street addresses, Social Security Numbers, and other highly sensitive information. The username / password pairs can readily provide access to the user's accounts on external sites. The history items can give the attacker a profile of the victim's browsing patterns, and serve as candidate sites for further break-in attempts using the retrieved set of username / password pairs. Figure 1 (b) shows a remote window collecting sensitive information about the user.

To mimick a spyware attack more closely, BROWSERSPY employs stealth to prevent the user from knowing that anything unusual is being conducted. The extension uses two techniques to shroud itself from Firefox's installed extensions list. First, the extension simply removes itself from the list so that the user won't see it. Second, it injects itself into a (presumably benign) extension, Google Toolbar (Figure 1 (a)). The latter method serves as a technique to guard the extension from being discovered should the user inspect the files on her system. The injection process is even successful at infecting code signed browser extensions,[1] as the browser does not check the integrity of these extensions following installation.

A common technique practiced by malware is covert information flow mechanisms [9] for transmission. To mimic this behavior, our final stealth tactic deliberately delays delivery of sensitive data to the remote host. We cache the information and send it out in periodic bursts to offset the network activity from the event that triggers it, making it harder for an observant user to correlate the added traffic with security sensitive operations. Thus, the composite effect of some relatively easy measures employed by our extension is alarming.

Extension entry vectors The typical process of extension installation requires the user to download and install the extension through a browser interface window. Though the BROWSERSPY extension can be installed this way, this is not the only route by which this malicious extension can be delivered to a browser. It can be delivered by preexisting malware on the system without involving the browser. It can also be delivered outside the browser given user account access for a short duration. These entry vectors are all too common with unpatched systems, public terminals, and naive users who do not suspect such threats.

Extension development effort Very little effort was required to create this extension. The lack of any security in the browser's Extension Manager module assisted in its speedy creation. It only took one graduate student (who had no prior experience in developing extensions) three weeks working part time to complete this extension. We present this information merely to argue the ease with which this task can be accomplished. We note that this period of three weeks is merely an upper bound of effort for creating malicious extensions. Malware writers have more resources, experience and time to create extensions that could be more stealthy, perhaps employing increasingly sophisticated covert mechanisms for information transmission.

Our implementation techniques We started by studying the procedure of how extensions are created, installed and executed in the system. Firefox extensions make use of the *Cross-Platform Component Object Model* (XPCOM) framework, which provides a variety of services within the browser such as file access abstraction. We carefully studied interfaces to the XPCOM framework available for use by an extension, and discerned that one could easily program event observers for various operations performed by the browser. We implemented the spying features based on four of these event observers as itemized in Table 1.

We make unconventional use of the XPCOM framework to achieve hiding mechanisms in our spyware implementation. To simply disappear from the browser user interface, we use a standard interface (Table 1) to manipulate an

[1] Case in point, the code in the Google Toolbar extension is signed by Google, Inc.

XPCOM Interface	Usefulness to perform malicious behavior
nsIHistoryListener	By attaching an event listener of this type to each open document, the browser notifies the malware when a new document is opened.
nsIHttpChannel	By attaching an event listener to this interface, the browser grants the malware a chance to inspect query parameters before submission.
nsIPasswordManager	The malware invokes a method provided by this interface which reveals all of the user's stored passwords.
nsIRDFDataSource	This interface provides the malware with write access to one of the Extension Manager's critical internal data objects.

Table 1. The malware extension exploits the use of these XPCOM interfaces to perform attacks.

internal data object belonging to Firefox's Extension Manager. This exposes a flaw in the browser implementation, full access to an object is exported where it should remain at most read-only to the extension code base.

Injecting the BROWSERSPY extension into another extension requires copying a file into the target's directory and then appending some text to the target's `chrome.manifest` (a file containing declarations instructing the browser how to load an extension). The absence of file access restrictions on extension code easily allow this injection attack. It is actually a more subtle and fundamental flaw in the implementation of Firefox that allows such attacks to be carried out with ease. Instead of storing user preferences in a data file and reading them for later use, the browser generates JavaScript code every time the user changes her preferences, and executes this file on startup. This is poor design from a security perspective. If the integrity of this file is compromised the browser can easily be attacked. Our BROWSERSPY extension precisely exploits such implementation weaknesses.

Through mostly normal use of the services Firefox provides to extensions, we have been able to concretely demonstrate much cause for concern.

4 Our approach to enhance security

Firefox's vulnerabilities can be strengthened to make all of the BROWSERSPY attacks unsuccessful. As mentioned in the introduction section, this requires us to enforce the following requirements:

Requirement 1 Ensure the integrity of the browser's code base.
Requirement 2 Protect sensitive user data from being accessed or modified by the extension code base.

A browser that adheres to Requirement 1 prevents the BROWSERSPY extension from injecting itself into the browser's code base. Implicitly, this first requirement also disallows unauthorized extensions to access sensitive data, contributing to the fulfillment of Requirement 2.

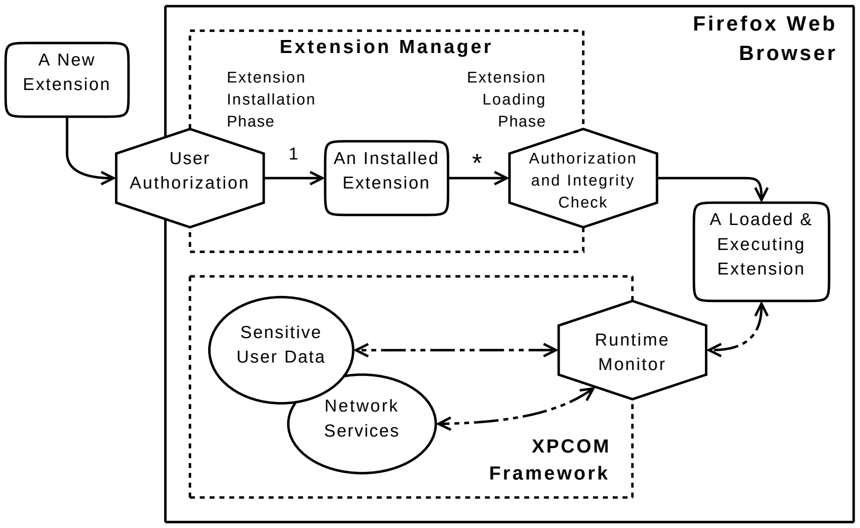


Fig. 2. Overview of Firefox's extensible architecture (hexagons represent functionality added to improve security). Extensions must be user authorized and uncorrupted to get loaded into the browser. Extension access to XPCOM is controlled by policies defined in the runtime monitor.

A high level architecture of our solution is presented in Figure 2. Browser code base integrity is addressed in our approach by a mechanism of user authorization which we describe in the remainder of this section. Protection of sensitive information is addressed in Section 5 through monitoring mechanisms that control extensions' access to the XPCOM framework.

4.1 Extension installation and loading

It is important to understand a browser's code base to clarify the issues surrounding its integrity. Firefox and other extensible browsers, when installed in a fairly secure fashion, have at least two components to their code base:

1. *Browser core*, the code directly loaded by invoking the browser executable.
2. *User code base*, additional program code loaded from among the user's files as the browser starts up.

We analyze the browser core and user code base to determine how the concepts of code authorization apply to each.

By default the code in the browser core must be granted full privileges within the browser, and we say that the user has authorized this by the basic act of installing the browser. This authorization is typically enforced by making the browser core not modifiable by an ordinary (unprivileged) user account. The files that constitute the core have their owner set to the superuser, and do not allow write privileges for any other users and groups. In addition to the code directly loaded by invoking the browser executable, there can be extensions installed as part of the browser core. For the purposes of this paper we include them in the browser core, as they share authorization properties with it.

The code that makes up the user code base is also authorized by the act of installation. This typically takes the form of the user confirming the install of an extension via the browser's graphical user interface. As the browser runs with the privileges of the user who invoked it, the browser is capable of installing extensions into the user code base on behalf of the user.

A critical aspect to browser security is the integrity of user code base, given that the browser core is well protected. If the user code base of the web browser is compromised, the end user is vulnerable to attack by malware such as the BROWSERSPY extension. The following two principles are fundamental to the user code base integrity:

Principle 1 Code should not be introduced into the user code base of the browser without the user's authorization.

Principle 2 Code that is part of the user code base of the browser should not be modified without the user's authorization.

It is necessary that browsers with an extensible architecture enforce these principles.[2] Integrity of the user code base can not be guaranteed unless both are upheld.

As indicated in Section 3, the Firefox web browser is vulnerable to attack against both principles. Installing an extension outside of a browser session by emulating the Firefox's installation routine is one way of introducing code into the user code base. This can be done without the user's knowledge or consent, thus betraying Principle 1. Furthermore, modification of the user code base of the browser can be realized by conducting an injection attack on a trusted extension, as the BROWSERSPY extension does. The injection is performed without the authorization of the user, which violates Principle 2.

Code signing One potential solution to this problem is to require all extensions to be delivered to the user's browser with their code signed by a trusted entity. In this scenario, the user code base can be validated at any time to determine if its integrity has been undermined. However, Firefox has a design flaw in its current implementation of signed extensions that precludes the effectiveness of this solution. It only validates code at the time an extension is installed; when the browser loads an extension for execution, no integrity check is performed, making it easy for the BROWSERSPY extension to inject code into signed exten-

[2] We note that even though this threat exists for other programs that are present in the user's account, the threat on the browser is especially critical due to nature of sensitive information available to it.

sions. Firefox can not uphold Principle 2 without a fix to maintain code integrity, and a solution to this requires re-architecting the browser.

A way to detect the addition of code to the user code base is required to enforce Principle 1, even for code that has been signed by a trusted provider. The detection mechanism must implement an indicator of what extensions are currently part of the user code base, enabling newly introduced extensions to be differentiated from the set of previously authorized ones. This indicator needs to be secure against tampering by an agent other than the user, as the user is the *sole* authorizing agent with respect to what extensions are part of the user code base. A system based simply on remotely signed extensions does not provide these facilities, and thus can not ensure that all additions to the user code base are user authorized.

User signed extensions A solution aimed at providing a better protection layer for the browser code base must certainly allow for unsigned extensions in order for it to offer any practical benefit. As previously mentioned, an extension distributor such as Mozilla may not be willing to provide any assurances with regard to third-party code by signing extensions on their behalf. Yet, users still want to allow such extensions into their user code base, as evidenced by the popularity of unsigned extension downloads. This poses a dilemma.

Our solution to this dilemma is to empower the user with the ability to sign extensions that are included in her user code base. Once the user has indicated approval for the unsigned code to become integrated into the user code base, we provide tools for the user to sign and suitably transform the code so that at any point its integrity can be verified. After the conversion to a user-signed extension is accomplished, we augment the browser with support for maintaining assurance of its user code base integrity. This thwarts injection attacks by malicious code (e.g., BROWSERSPY). These user signed extensions thus enhance resiliency of the browser code base to unauthorized modification.

User signed extensions also enable a convenient mechanism for the user to tightly control what is allowed into the user code base. Extensions can be allowed execution based on whether or not they have been signed by the user.[3]

Implementation approach To prevent a malicious extension from tainting the trusted code base of Firefox, we have developed a prototype implementation of user signed extensions. The remainder of this section describes this implementation in detail.

The default behavior of the browser core has been augmented in two places:

1. *Extension installation*, performed once each time an extension is installed or updated to a more recent version.

[3] They may be further monitored for confidentiality and integrity policies as described in Section 5.

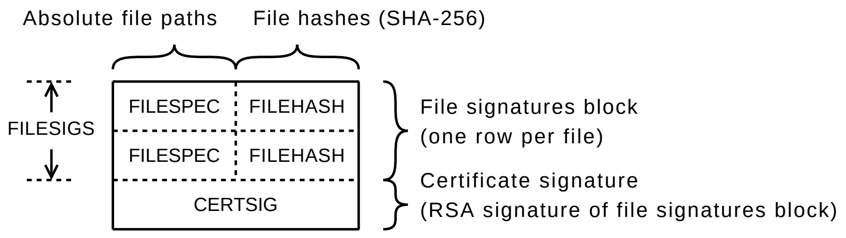


Fig. 3. A user signed extension certificate, employed to verify authorization and integrity of untrusted code.

2. *Extension loading*, occurring each time a new browser session is begun.

During installation, our solution assists the user in signing extension code so that it can be safely incorporated into the user code base. During loading, each extension loaded from the user code base is tested for code integrity before allowing it to be introduced into the browser session. If an extension has not been signed by the user, Firefox will not load it. Loading will also be denied to any extension for which integrity verification has failed. Figure 2 displays these steps as an extension makes its way into a browser session for execution.

Extension certificates User extension signing is performed by generating a certificate (Figure 3) for each installed extension which can be used for the purposes of authorization and integrity checking. These certificates are composed of two sections:

1. FILESIGS, used to verify the integrity of the extension's files.
2. CERTSIG, used to verify the integrity of FILESIGS.

Every file that comprises the extension is represented by a signature in the FILESIGS section of the certificate. Each file's signature is composed of its absolute path (FILESPEC) and a SHA-256 content hash (FILEHASH). By comparing the list of files present in the extension at load time with FILESIGS, the browser detects if a file has been added to or removed from the user code base without authorization. Through comparison of each file's hash value at load time with their respective FILEHASH, the browser notices if one of the trusted files has been illicitly modified. Firefox will refuse to load any extension that is revealed by these detection mechanisms to have violated user code base integrity.

The certificate signature CERTSIG is the RSA signed MD5 hash value of FILESIGS. As an extension is loaded, the browser generates another FILESIGS corresponding to the load-time state of the extension's root directory. The browser is then able to determine whether the file signatures represented in the certificate are valid by computing a hash of the extension's load-time FILESIGS and comparing it to the hash stored in CERTSIG. This check will fail if any of the following events have occurred subsequent to installation:

1. a file is added to or removed from the extension
2. a file is added to or removed from FILESIGS
3. one of an extension's files is modified
4. one of the certificate's FILEHASH is modified

Upon detection of these forms of corruption, the browser will rule not to load the extension.

The integrity of a certificate signature is protected by having the user sign it via RSA public-key cryptography. This signing by the user is what explicitly authorizes the extension to become part of the user code base. If the signature is tampered with, the browser will not be able to derive the hash value of FILESIGS, which must be decoded from CERTSIG to validate the certificate. In such a case, the browser will refuse to load the extension.

Key safeguarding It is necessary to protect the user's public and private keys that are used in this solution, as they are the root of the security provided by user signed extension certificates.

An attacker can circumvent authorization if he gains access to the private key. He can modify user signed extensions and sign them himself by emulating the browser's certificate generation process. Since we expect extension based browser attacks to be launched by a malicious agent with user level access, and the user's private key is likely to be stored in local file space under user control, additional security is needed to protect the private key.

The enhanced protection is provided by encoding the private key using AES encryption. This encoded private key is made available to the browser, which prompts the user for her AES passphrase whenever the RSA private key is needed for extension installation. As only the user knows the passphrase, the private key is not accessible to attackers.

A different exploit is possible if an attacker is able to overwrite the user's public key with one of his own. In this scenario the browser is fooled into accepting extensions that are signed by the attacker and refusing those that are signed by the user. This privilege swap attack is possible because only the public key is used in the certificate validation process (private key safeguards do not come into play).

To protect the public key from this attack, our solution stores the key file as part of the browser core; writing to the key file requires administrative (root) privileges, though reading can still be performed by the user. This makes the key invulnerable to attack by an agent with only user level access.

Usability It is well understood that a security solution that is invasive or difficult to use will face resistance in user uptake. If users decide they would rather not use the security solution then the benefits it provides can not be realized.

With this concept in mind, the solution presented here is implemented in the least intrusive way possible.

Recall that the user must provide an AES passphrase in order to decrypt the private key needed for code signing. The browser prompts the user for this passphrase during installation. This is the minimal burden that our integrity mechanism imposes on the user.

The browser could require the user to authorize each extension when it is loaded, which would require the user to authenticate every time the browser starts up. Instead, authorization is performed only during extension installation. This way the user has to authenticate only on rare occasion: when installing or upgrading an extension. The system is just as secure as one which performs load-time authentication, and exhibits greater usability.

Another usability concern is apparent during the certificate generation phase of extension installation. As the user is performing the infrequent activity of adding a new extension, she may decide to add more than one. A multiple installation situation is especially likely when a periodic software update is triggered by the browser. Considering that each certificate generated requires the user to authenticate, installing several extensions could frustrate the user by repeatedly prompting for her password.

The obvious solution is to authenticate the user once, and then perform the certificate generation in bulk. This is the approach taken by our implementation. Once the user has decrypted the private key needed for signing, it is used to sign all the necessary certificates before being zeroed and deallocated.

Care must be taken when performing multiple installation based on a single authentication. It is highly important that the user always know what is signed as a result of authentication. If this issue is not regarded, a malicious extension could be injected among the other extensions to be installed without the user's knowledge. To defend against this attack, our implementation displays a list of all extensions that will be signed on the user's behalf before authentication is required. The user can decide to generate certificates for all extensions that are pending installation, or for none of them. Additionally, authorization decisions can be made per-extension. Screen shots of our changes to the installation mechanism can be found on the project website [3].

4.2 Install protection experimental analysis

The solution was tested to determine its compatibility with popular browser extensions and its impact on Firefox's speed. The 20 most popular extensions for Firefox were used as a basis for our performance evaluation (listed in Table 2).

Compatibility testing Determining compatibility was done by running the extensions in the test environment and exercising their core functionality. As some

1. Download Statusbar	6. Forecastfox	11. Web Developer	16. Map+
2. FlashGot	7. Tab Mix Plus	12. Cooliris Previews	17. StumbleUpon
3. NoScript	8. VideoDownloader	13. DownThemAll!	18. Foxmarks
4. Adblock Plus	9. Foxytunes	14. FireBug	19. Clipmarks
5. FireFTP	10. Fasterfox	15. Torrent Search	20. Answers

Table 2. Top 20 most popular extensions from `addons.mozilla.org` tested with our implementation. Drawn from the top 23 as we elected to skip platform dependent and non-English language extensions.

Installation / Loading performance benchmark	Number of extensions installed				
	1	2	3	4	5
Total time spent generating certificates (s)	18.6	38.1	53.5	75.6	94.7
Average time spent per certificate generated (s)	18.6	19.1	17.8	18.9	18.9
Percent of generation time spent signing certificates	99.5	99.5	99.9	99.9	99.8
Total time spent validating certificates (s)	0.75	1.50	2.30	3.00	3.70
Average time spent per certificate validated (ms)	748	750	767	750	740
Percent of validation time spent verifying signatures	90.5	92.3	95.8	95.3	96.4

Table 3. Extension installation integrity system benchmarks. The system used was a modified version of Firefox 2.0, running on Ubuntu 6.06 LTS, on an AMD Athlon 64 X2 3800+ (2GHz), 2GB RAM. The extensions tested were the top five from Table 2.

extensions provide a large feature set, it would have been difficult to exercise their total functionality. In our tests, 18 out of 20 extensions performed flawlessly. The extension Forecastfox elected to force registration of its XPCOM components using an `.autoreg` file, which the browser deletes following registration. The Foxytunes extension renamed its platform-specific component DLL to remove the platform identifier. These two user code base integrity violations are the result of actions taken that other extensions were able to avoid through different approaches to the same task. We also note that in general it is not possible for automated mechanisms to reason about the safety of these file manipulation operations, and hence the only option is to disallow them.

Performance overheads To evaluate performance in terms of speed, we benchmarked the extension loading and installation phases under five conditions. For each test, we installed from one to five extensions and measured:

1. the time needed to generate the user signed certificates during installation,
2. the time needed to validate the certificates during loading, and
3. the time spent performing RSA cryptography during items 1 and 2.

The cryptography implemented uses 128-bit passphrases for AES, 512-bit keys for RSA, and SHA-256 for file hashing. MD5 is used to hash FILESIGS for use in generating CERTSIG.

The results of our speed tests are shown in Table 3. It takes the implementation about 18.7 seconds on average to generate a certificate. The benchmarks indicate over 99.5% of that time is spent signing the certificate using RSA.

Extension loading takes a little longer than a stock installation of Firefox. It takes about 751 ms for each certificate to be validated, of which there is one per extension. For validation of the certificate signature, we again observe that over 90% of the time is spent applying RSA cryptography.

We estimate that by using the browser's native RSA implementation, a significant benefit to performance would be gained. The RSA implementation we use is written in JavaScript. Due to its nature as an interpreted language, JavaScript is slow running for computationally intensive algorithms as present in RSA. We chose RSA in JavaScript as it is the easiest drop-in solution for our purpose of generating a stand-alone patch for the browser. If Firefox were modified such that its internal C++ public key cryptography routines be made available for use by the Extension Manager, we believe the performance of certificate generation could be greatly enhanced. Improved RSA performance would also allow us to use greater length keys, increasing the browser's resilience to attack.

It is apparent that the AES and SHA-256 cryptography routines do not noticeably impact performance. If a faster RSA implementation were employed, the significance of the other two cryptographic functions would likely increase.

We acknowledge the importance that the solution be optimized for greater extension load-time performance, as it is the common case when contrasted against extension installation. When comparing the speed of loading to installation in our prototype, it is conspicuous that the system is optimized for loading.

5 Extension execution

User signed extensions disallow the user code base from unauthorized changes. However, this doesn't address the threat of a malicious extension installed with user consent. Once installed, it can corrupt the integrity of the user code base or even the browser core by making changes to the runtime state of the browser. Our BROWSERSPY extension hides itself using this mechanism; it alters the runtime state of the Extension Manager affecting display of the list of installed extensions.

The second phase of our solution therefore involves controlling an extension's access to critical browser services (XPCOM) via runtime monitoring. (The XPCOM services are discussed in length [7] along with many useful references.) Ordinarily, extensions enjoy unrestricted access to every interface of this framework. Our focus in this section is mainly on the mechanisms and infrastructure needed for a runtime monitoring solution governing access to the XPCOM interface.

The default security manager for JavaScript uses policies such as the *same origin* policy and the *signed script* policy. Firefox enforces these policies for web content pages, however does not so restrain internal JavaScript operations.

Moreover, these policies are oblivious to browser overlays (explained below), another regular feature present in extensions. A straightforward adaptation of the use of these policies on extensions is not suitable for these reasons.

The action attribution problem To enforce policies on a per-extension basis, it is necessary to identify the extension requesting each XPCOM operation. Unfortunately, Firefox does not have sufficient mechanism in place to establish identity due to the presence of *file overlays*. Extensions can provide these overlays to core portions of the web browser, which may extend and selectively mask the browser core. This integration is handled by Firefox in a way that does not retain means to identify the extension that applied the overlay. Through a malicious execution of this procedure, an extension can anonymously inject program code into the base functionality of the browser. Therefore, our solution to the action attribution problem has two parts: for overlay and non-overlay files.

Handling non-overlaid files Policy enforcement mechanisms are comparatively simpler for non-overlaid files. In this case, the executable statements they contain are traced back to the extension that issued them. This data is compiled into the set of extensions contributing to any specific operation, used for the basis of policy decisions. The procedure of tracing the origin of a single operation involves getting the URL of the currently executing script (maintained by the browser's JavaScript interpreter *Spidermonkey*) and deriving an extension identifier from it. We have implemented this action attribution mechanism and discuss the performance later in this section.

Handling overlaid files JavaScript statements present in overlay files require special handling. When a command is executed from one of these files, the script filename available to the runtime monitor points to the target of the overlay. This target is usually part of the browser core—not part of the extension that the code originated from—resulting in the action attribution problem explained earlier.

We have devised a way to provide the means of associating actions of overlay files with their extension of origin. Our approach is based on automatic interposition of "delimiting statements" around blocks of code that qualify as entry and exit points. These statements enable us to identify the executing extension for all code evaluated within the enclosed block.

The opening statements that we interpose manipulate a stack (maintained in the browser core) by pushing an extension identifier onto it. The interposed closing statements subsequently access the stack to pop the identifier off. An indicator of which extension is issuing the intermediate code is then found at the top of the stack. A *try-finally* construct wraps the function body to ensure that we pop the stack in the event of a return statement or thrown exception.

Spidermonkey is adapted to perform the interposition. Out of the box it provides us with an API capable of compiling JavaScript statements into bytecode (in preparation for execution), and another interface to decompile bytecode back into its original JavaScript. Specifically, support was added to the decompiler so that it can do the needed interposition.

The technique employed is to compile JavaScript into bytecode, then feed the bytecode into the interposing decompiler. This procedure is conducted once per extension, at the time the extension is installed. The performance of this operation is comparable to the rewriting technique in [11] with the advantage that it does not have to run every time the browser application is launched.

The above infrastructure is sufficient to handle overlaid code. However, a total solution to the overlay problem requires stripping the JavaScript code from overlay files, transforming it using interposition, and stitching the file back together. We are currently adding support to our infrastructure to handle this straightforward operation, and therefore report the performance of this interposition mechanism on non-overlaid files later in this section.

Analysis of interposition mechanisms Under the interposition technique, a malicious extension may attempt manipulation of the active extension stack to spoof its extension ID, allowing access to XPCOM with elevated privileges and circumvention of stateful policies. Although this attack is theoretically possible, significant effort is required to mount it in our environment where user code base integrity is assured. Assuring a segment of code will not exploit this deficiency is difficult, stemming from the challenges in static analysis of JavaScript code.

Protection against this attack can be provided by an additional security layer that uses randomization. A signed, extension-specific magic number is given as an argument to the interposed stack manipulation code. This makes it harder for a generic attack to be constructed that is successful for more than a single targeted user. To achieve generality, the extension must self-modify by discovering and incorporating the magic number into its attack code. This is hard to do in our environment where extension integrity is continually enforced, as the malware must morph prior to installation.

Policies With infrastructure in place, we have implemented six policies on non-overlaid extension code. They are representative of the types of policies enforceable using this solution, and are described in Table 4.

The first four policies are extension specific. Complex policies can be composed of these rules to allow only the level of access an extension needs to function. The policies XPCOM-SAFE and PASS-RESTRICT are conservative policies that disallow access to sensitive data. PASS-RESTRICT and HISTORY-FLOW are enforced globally. HISTORY-FLOW is unique in that it is stateful. If any extension is detected accessing Firefox's URL history interface, that extension will be

Policy name	What it does	Granularity
XPCOM-ALLOW	Allow all access to a XPCOM interface	per extension
XPCOM-DENY	Deny all access to a XPCOM interface	per extension
SAME-ORIGIN	Allow access to same-origin domains	per extension
XPCOM-SAFE	Deny all access to XPCOM while SSL is in use.	per extension
PASS-RESTRICT	Deny access to the password manager	all extensions
HISTORY-FLOW	Prevent URL history leaks via output streams	all extensions

Table 4. The example policies that were created and tested with the runtime monitoring solution.

Extension	Function	Stock (ms)	File Lookup (ms)	Overhead (%)	Interposition (ms)	Overhead (%)
Adblock Plus	abp_init()	14.1	14.5	2.8	15.4	9.2
Download Statusbar	init()	4.5	4.7	4.4	5.0	11.1
FireFTP	changeDir()	26.4	29.4	11.4	32.6	23.5
FlashGot	getLinks()	4.2	4.4	4.8	4.6	9.5
NoScript	nso_save()	14.2	16.7	17.6	18.7	31.7
Average				8.2		17.0

Table 5. Performance micro-benchmarks for the default browser behavior and two different action attribution methods. The execution time of selected functions within the top five most popular extensions is measured over 1000 runs. The same test platform described at Table 3 was used.

disallowed further access to interfaces of type `nsIOutputStream`. This protects writes to files and network sockets over a single session.

Performance To evaluate the performance of our approach to action attribution, we wrapped functions within the five most popular Firefox extensions with benchmarking code. One thousand iterations of each function were performed in: (a) an unmodified browser, and (b) a browser using the filename lookup mechanism , and (c) a browser using the interposition technique on overlay files.

We observed a modest overhead of 8.2% on average to apply our policies using filename lookup. The interposition mechanism was slightly slower, imposing an overhead of 17.0% on average. The additional impact is not detrimental considering that overlay code is typically short, thus causing minimal difference in the overall user experience. Our experience in operating the browser with active runtime monitoring and policy enforcement did not indicate perceivable overhead.

6 Conclusion

We authored a malicious extension as proof-of-concept that security concerns exist in modern extensible web browsers. We selected the open source browser Firefox as our target platform, because it suffers many of these flaws.

The threat of malicious extensions was addressed using two mechanisms: (1) a mechanism by which the installation integrity of extensions is validated at load-time, and (2) infrastructure for runtime monitoring and policy enforcement of extensions to further prevent attacks on browser core integrity and sensitive data confidentiality.

Our changes to Firefox insure that the browser allows only extensions installed by the user to be loaded, and detects unauthorized changes made to installed extensions. This modification seals the outside installation vector for malicious extensions by disallowing standard and injection type installations external to a browser session. We enabled the browser to monitor a significant portion of extension code at runtime and effect policy on a per-extension basis. The monitoring infrastructure and the set of policies that we have created represent only a starting point. More research is needed for designing a comprehensive suite of policies that can be enforced on extensions with acceptable overheads on usability.

We are currently pursuing efforts to integrate our extension integrity checking prototype into the Firefox browser main source tree. Our malicious extension is available through private circulation for malware researchers.

References

1. Information from http://en.wikipedia.org/wiki/Mozilla_Firefox.
2. Information from http://addons.mozilla.org.
3. Project website. http://research.mike.tl/view/Research/ExtensibleWebBrowserSecurity.
4. Protected mode in vista ie7. http://blogs.msdn.com/ie/archive/2006/02/09/528963.aspx.
5. eTrust Pest Patrol. Pests detected by pestpatrol and classified as browser helper object. http://www.pestpatrol.com/pestinfo2005.
6. I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *USENIX Security Symposium*, 1996.
7. O. Hallaraker and G. Vigna. Detecting Malicious JavaScript Code in Mozilla. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 85–94, Shanghai, China, June 2005.
8. J. Kirk. Trojan cloaks itself as firefox extension. Infoworld magazine, July 2006.
9. B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10), 1973.
10. G. C. Necula. Proof-carrying code. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 106–119. ACM SIGACT and SIGPLAN, ACM Press, 1997.
11. C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.
12. R. Sekar, V. N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model carrying code: A practical approach for safe execution of untrusted applications. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.
13. P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, , and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Network and Distributed System Security Symposium (NDSS)*, San Diego 2007.
14. R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium of Operating System Principles*, 1993.