J Comput Virol (2008) 4:179–195
DOI 10.1007/s11416-007-0078-5

ORIGINAL PAPER

Enhancing web browser security against malware extensions

Mike Ter Louw · Jin Soon Lim · V. N. Venkatakrishnan

Received: 31 August 2007 / Revised: 27 November 2007 / Accepted: 10 December 2007 / Published online: 12 January 2008

Abstract In this paper we examine security issues of functionality extension mechanisms supported by web browsers. Extensions (or "plug-ins") in modern web browsers enjoy unrestrained access at all times 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 implemented a malware application for the popular Firefox web browser, which we call BROWSERSPY, that requires no special privileges to be installed. BROWSERSPY takes complete control of the user's browser space, can observe all activity performed through the browser and is 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 describe two implementations of this mechanism: a drop-in solution that employs JavaScript and a faster, in-browser solution that makes uses of the browser's native cryptography implementation. We also discuss techniques for runtime monitoring of extension behavior to provide a foundation for defending threats posed by installed extensions.

M. Ter Louw (✉) · J. S. Lim · V. N. Venkatakrishnan
University of Illinois, Chicago, Illinois, USA
e-mail: mter@cs.uic.edu

J. S. Lim
e-mail: jlim@cs.uic.edu

V. N. Venkatakrishnan
e-mail: venkat@cs.uic.edu

1 Introduction

The Internet web browser, arguably the most commonly used application on a network connected computer, is 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.

These uses offer the browser a unique perspective; it can observe and apply contextual meaning to sensitive information provided by the user during very personal activities. Furthermore, the browser has access to this information as *plaintext*, 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 (measured using the `kloc` tool) written in a variety of languages that include C, C++, Java, JavaScript, PHP, 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 subproblem of ensuring confidentiality and integrity within a browser in the presence of browser extensions. We discuss this problem in the context of Mozilla Firefox, the widely used free and open source web browser software, used by about 70 million web users [10].

Browser extensions (or "add-ons") are utilities provided to customize the browser. These extensions alter the browser's

Table 1 Extension support in popular web browsers

Extension capability	Firefox 2.0	Internet Explorer 7.0	Opera 9.02	Safari 2.0.4
Extensions are officially supported	Yes	Yes	Yes	No
User-installed extensions can be disabled easily	Yes	Yes	No	No
User authentication is required for installation	No	No	No	Yes
Installation integrity is verified and enforced	No	No	No	No

behavior by making use of interfaces exported by the browser and other plug-ins. Though each 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.

Given that extensions plug directly into the browser, there is surprisingly no provision currently in Firefox to protect against malicious extensions. One way of providing this protection is to disallow extensions entirely. Firefox can achieve this effect by starting up in debugging mode, which prevents all extension code from loading. However, when started in the normal mode (the default in a typical installation) extensions are allowed to execute. Extensions are widely popular, as indicated by the their download numbers [9], and provide useful functionality to the several thousands of people who employ them. Dismissing the security concerns about extensions by turning them off ignores the threat to these users.

To better understand the impact of running malicious extensions, we set a goal for ourselves to actually craft one. Surprisingly, we engineered a malicious extension for the Firefox browser we call BROWSERSPY, with modest efforts and in less than 3 weeks. Once installed, this extension takes complete control of the browser.

There are two main problems exposed by the actions of our malware 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 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 browser code base 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 when the user authorizes installation of possibly malicious or vulnerable extensions, we augment the browser to support policy-based monitoring of extension behavior. The monitoring and policy enforcement is achieved using interposition mechanisms retrofitted to the Spidermonkey JavaScript engine and other means (Sect. 5).

A key benefit of our solution is that it is targeted to *retrofit* the browser. We consider this property very important, and have traded off otherwise better solutions to achieve it. 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 an expanded version of our previous work [15] and is organized as follows. A discussion of related work appears in Sect. 2. We explain details about our malware extension and the associated threat model in Sect. 3. We present our solution to the extension integrity problem in Sect. 4 and address data confidentiality in Sect. 5. We evaluate these approaches with several Firefox add-ons and discuss their performance in the above sections individually. In Sect. 6 we conclude.

2 Related work

Extensions in various browsers We examined extension support in four contemporary browsers: Firefox, Internet Explorer (IE), Opera and Safari. Table 1 lists the level of extension support for these four browsers. Among the browsers that we studied, only Safari does not natively support the concept of extensions. The remaining three possess extensible architecture but none have adequate security mechanisms addressing extension-based threats.

For instance, IE's primary extension mechanism is through Browser Helper Objects (BHO). BHOs are different from Firefox extensions, as they are primarily implemented using native x86 code. This makes them difficult to analyze due to well known problems with x86 machine code disassembly (e.g., distinguishing code instructions from data). Firefox supports extensions containing native, platform-dependent program code in what are termed *components*, although the vast majority of extensions shun the feature to facilitate portability. Another difference between platforms is BHOs share a common address space with the browser, providing BHO-based spyware greater access to browser resources by enabling direct reading of browser memory. In contrast, Firefox extensions are mostly implemented using interpreted JavaScript code that can not directly address the browser's memory. Their primary access to browser resources is via a managed JavaScript API. Furthermore, the JavaScript source code used in Firefox extensions is available for analysis, thus they are not subject to the disassembly problem.

The PestPatrol malware detection website lists hundreds of malware implemented as BHOs [1]. Despite being a mainstream browser and very little spyware presence, our BROWSERSPY extension and the FormSpy extension [6] (discussed in Sect. 3.2) demonstrate that the problem for Firefox is equally significant and needs further study.

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 the research of ideas such as signed code, static analysis, proof-carrying code (PCC), 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 issues that arise.

Signed code The Firefox browser provides support for signed extensions. This is hardly used in practice, however. A search of extensions in the Firefox extensions repository, http://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. A valid signature only guarantees an extension is from the browser distribution site and was 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 and 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.

JavaScript is a very flexible and dynamic language, and thereby contains features that make static analysis difficult. The difficulty stems from its prototype-based inheritance model, which borrows heavily from the *Self* language described in [17]. The model has a very loose type system, where little can be inferred about the properties of an object without complete knowledge of the object's history since instantiation. For instance, although a programmer may define a JavaScript "class", instances of this class can have individual methods reimplemented at run time, effectively subclassing the original type. To build an accurate control flow graph for the purpose of behavior modeling, tracking these dynamically created types is of critical importance. Without such information, we can not determine method invocation targets with certainty.

It is difficult to employ static analysis for JavaScript code without making conservative assumptions, however. Table 2 gives some more concrete scenarios where a technique based on purely static analysis mechanisms will have difficulty. The first example illustrates the difficulty in 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. A second example involves use of runtime values that pose a difficulty for any static analysis mechanism. Another example (not shown) 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.

Consequently, recent efforts that trace JavaScript code [12,18] use runtime approaches to track references. However, static analysis is additionally employed in [18] to detect *cross-site scripting* (XSS) attacks in cases where dynamic analysis alone is insufficient. They avoid the static analysis pitfalls using a conservative form of tainting. Typical scripts in web pages may not always employ complex JavaScript constructs such as *eval*, and therefore this conservative approach works for the purpose of preventing XSS attacks. If scripts from web pages perform a complex series of actions (such as involving the use of the result of a JavaScript *eval* construct), static analysis is likely to produce false positives. This makes it unsuitable for analyzing browser extension code in JavaScript, as almost half of the extensions that we tested make heavy use of complex *eval* constructs, and all frequently use objects as associative arrays.

PCC and MCC The difficulties surrounding static analysis make frameworks such as PCC [11] 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

Table 2 Some JavaScript statements that make static analysis difficult

Statement category	Example(s)	Problems

Variable assignment to or from an object property or array element	`var object1 = {};`
	…
	`var object2 = object1["property"];`
	`object1["property"] = object2;`
	`var array2 = [];`
	`var object3 = {};`
	…
	`var object4 = array2[i];`
	`array2[i] = object3;`
		We don't know what object references are being assigned to variables object2 and object4.
		We don't know what property of object1 is being assigned the reference to object2.
		We don't know what element of array2 is being assigned the reference to object3.

Object property or array element deletion	`var array3 = [];`
	`var object5 = {};`
	…
	`delete array3[i];`
	`delete object3["property"];`
	`with (object3) delete "property";`
		We don't know which element of array3 is being deleted.
		We don't know which property of object3 is being deleted.

An analyzer would need to track object references in detail along with complex type information for each object. This task becomes difficult when the value of *i* and *"property"* are dependent on expressions that can only be evaluated at runtime

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 runtime check placement.

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 [13] 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 to guarantee exhaustive code coverage, a requirement of approaches based on runtime learning of behavior models.

Execution monitoring Several execution monitoring techniques [19,3,4] have previously looked at the problem of safely executing malicious code. A closely related work is by Hallaraker and Vigna [4]. This was one of the first efforts that looked at the security issues of executing malicious code in a large mainstream browser. Their focus is on protection against web 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 already prevented from performing sensitive actions by execution in a constrained environment (i.e., sandbox). In contrast we address the extension installation integrity problem, as extension code is unmonitored and unrestrained from performing 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 Sect. 5.

Kirda et al. [5] present a detection technique for spyware that hook into IE through its BHO interface. Their technique is based on monitoring the runtime behavior of BHOs in a controlled environment using a series of test inputs. Individual behavior patterns are identified at the level of IE and Windows APIs using a combination of dynamic and static analysis techniques. Their approach does not address threats from BHOs that read from the IE address space directly, however.

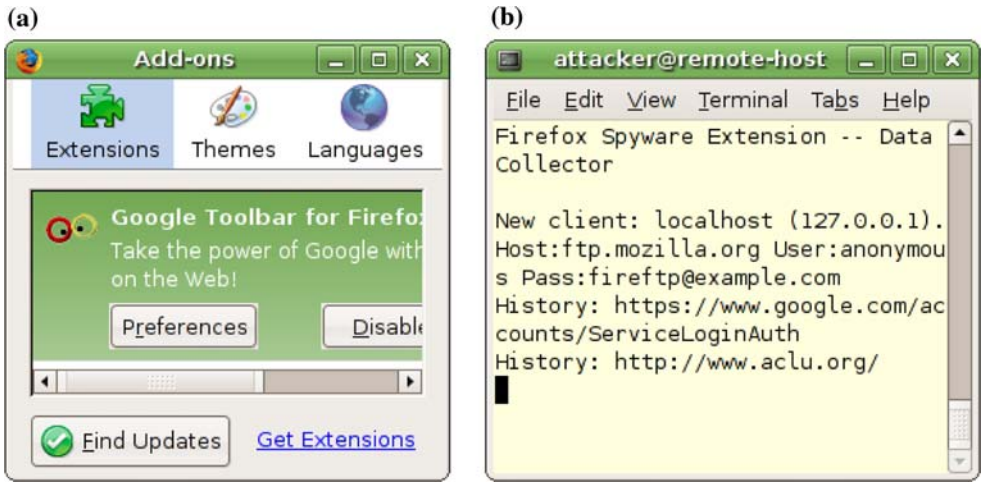

Fig. 1 Two views of the BROWSERSPY extension in operation: **a** extension hiding from the browser, and **b** data collector receiving sensitive information

Spyshield [8] prevents threats from extensions that can directly read browser memory by running extensions in a separate address space. Spyshield's policy enforcement mechanism is based on blocking potentially malicious behavior, and is similar to the technique we describe in Sect. 5 with the difference that they are implemented for separate browser architectures. Theirs is for IE, while our solution is for the web browser Firefox.

Behavior-based detection is a complimentary technique to an effort that is based on blocking suspected malicious behavior on interfaces. Behavior-based detection can be more powerful in identifying certain non-malicious behavior patterns than plain behavior blocking. However, behavior-based detection is crucially dependent on triggering malicious behavior with the right test inputs to a browser extension—a challenging task. Malicious behavior blocking *avoids* this challenge at the risk of denying certain legitimate behavior, but is considerably simpler to integrate in a browser.

3 Attacking the browser

To gain a better understanding of the threat posed by malware extensions, we engaged in the task of actually writing one. The motivations for creating the malicious software were 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) develop a practical understanding of the ease with which malware writers may be able to craft such extensions, and (d) provide a concrete implementation of a malicious Firefox extension to serve as a benchmark for malware analysis.

3.1 A malware extension

The extension we authored, BROWSERSPY, is capable of harvesting every piece of form data (e.g., passwords) submitted by the user, including those transmitted over encrypted connections. Furthermore, once the extension enters the system, it hides and remains undetectable by users (Fig. 1a) as it operates.

Capabilities and behavior Once BROWSERSPY is installed, it begins collecting personal data that will ultimately fall into the hands of an attacker. As a user navigates the Internet, BROWSERSPY harvests the URLs comprising their navigation history and stores them in a cache. Username and password pairs that are stored in Firefox's built-in password manager are retrieved, along with the URL of their associated website. Form data that the user submits finds its way into the extension's cache as well. All of this information is stored and periodically sent over the network to an awaiting adversary at 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 navigation history 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 usernames and passwords. Figure 1b shows a remote window collecting sensitive information about the user.

To mimic a spyware attack more closely, BROWSERSPY employs stealth to prevent the user from knowing that anything unusual is being conducted. The extension uses two

Table 3 The malware extension exploits the use of these XPCOM interfaces to perform attacks

XPCOM interface	Usefulness in performing 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

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 (Fig. 1a). 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 (case in point: the code in the Google Toolbar extension is signed by Google, Inc.) as the browser does not check the integrity of these extensions following installation.

A common technique practiced by malware is covert information flow mechanisms for transmission [7]. 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 BROWSERSPY can be installed this way, it is not the only route by which this malicious extension can be delivered to a browser. It can be injected 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 attacks.

Extension development effort Very little effort was required to create this extension. The lack of security in Firefox's Extension Manager module assisted in its speedy creation. It only took one graduate student (who had no prior experience in developing extensions) 3 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 3 weeks is only an upper bound of effort for creating malicious extensions. Malware writers have more resources, experience and time to create extensions that could be more virulent and stealthy, perhaps employing increasingly sophisticated steganographic techniques for covert 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 interfaces as itemized in Table 3.

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 (`nsIRDFDataSource`) to manipulate an 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 best 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. There is actually a more subtle and fundamental flaw in the implementation of Firefox that allows similar 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

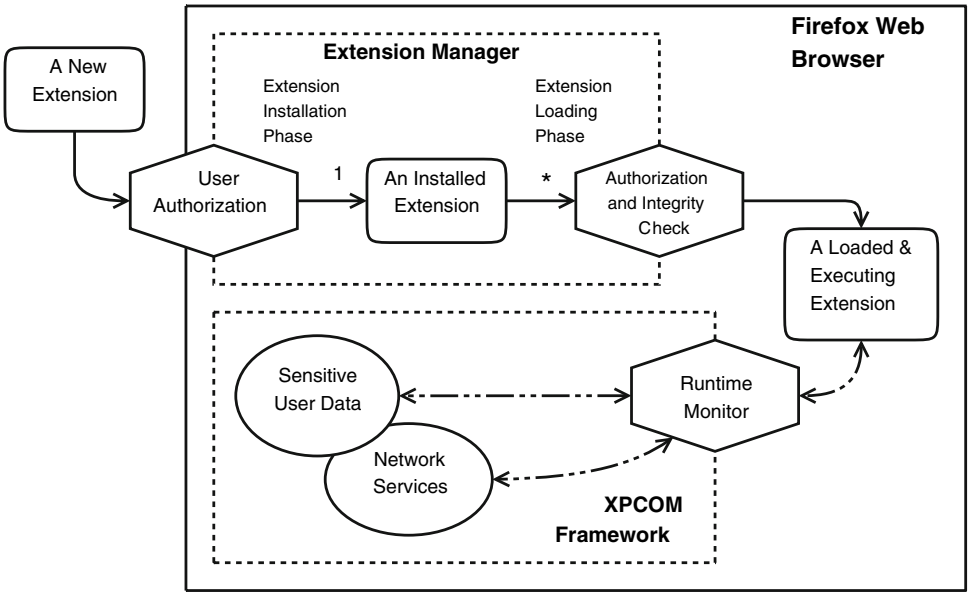

Fig. 2 Overview of Firefox's extensible architecture (hexagons represent functionality added to improve security). Extensions must be user authorized and uncorrupted to get loaded into the browser. Extension access to XPCOM is controlled by policies defined in the runtime monitor

perspective. If the integrity of this file is compromised the browser can easily be attacked. Our BROWSERSPY extension precisely exploits such implementation weaknesses.

3.2 Related developments

Through mostly normal use of the services Firefox provides to extensions, we have been able to concretely demonstrate much cause for concern. BROWSERSPY demonstrates a clear void in the browser's protection against unauthorized extension installation, via its injection attack, allowing it to violate the browser code base integrity. Inadequate protection against malicious extension behavior is also exhibited by BROWSERSPY with its snooping attacks, resulting in violations of user data confidentiality and integrity.

As further testimony of the inherent danger, a recent and similar attack was launched on the Firefox browser using a malware extension known as FormSpy [6], eliciting widespread media coverage and concern about naive users.

A different type of remote vulnerability exploit, brought to media attention by Soghoian [14], is characteristic of a man-in-the-middle attack. This attack targets the Firefox upgrade mechanism, and works whenever an extension upgrades from a server that is not SSL-enabled, thereby providing a vector for the man-in-the-middle attack. By this vector, the user will be open to spoofing attacks and could potentially receive maliciously crafted code. Preventing such man-in-the-middle attacks has a well known solution through the use of authenticated SSL channels for downloading extensions.

The variety and ease of attack vectors on extensible browsers has been shown to be a great risk to the people who rely on them. The situation grows more perilous for users who process greater amounts of sensitive information with their browser.

4 Extension installation and loading enhancements

Firefox's vulnerabilities can be strengthened to make all of the BROWSERSPY extension's attacks, and therefore attacks by similar threats, 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 BROWSERSPY from injecting itself into the browser's code base. Implicitly, this first requirement also disallows unauthorized extensions access to sensitive data, contributing to the fulfillment of Requirement 2.

A high level architecture of our solution is presented in Fig. 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. The topic is presented as discussions of security principles (Sect. 4.1), implementation techniques (Sect. 4.2), security analysis (Sect. 4.3) and performance evaluation (Sect. 4.4). Protection of sensitive information is addressed in Sect. 5 with monitoring

mechanisms to govern extensions' access to the XPCOM framework.

4.1 Security principles

It is important to refine the idea of a browser's code base to clarify the issues surrounding its integrity. Firefox and other extensible browsers, when installed in a fairly secure manner, 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. 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. That is, the files that constitute the browser core are owned by the superuser, and are read-only to all other users and groups. Therefore the browser core is well protected from user-based threats as its files can not be modified by malware (or other ordinary users) possessing only user-level access.

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 portion of the code base, 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, it is capable of installing extensions into the user code base on behalf of the user. The files that constitute the user code base are not stored in the file system with protection against user modification, which makes the user code base vulnerable to malware possessing only user-level access.

A critical aspect to browser security is integrity of the user code base, given that the browser core is well protected. If the user code base of the web browser is compromised, the user can be subjected to attacks such as those employed by BROWSERSPY. 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. (We note that even though this threat exists for other programs present in a user's account, the threat on the browser is especially critical due to the highly sensitive information it processes.) Integrity of the user code base can not be guaranteed unless both are upheld.

As indicated in Sect. 3, the Firefox web browser is vulnerable to attack against both principles. Installing an extension outside of a browser session by emulating 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, 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 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. This makes it easy for BROWSERSPY to inject code into signed extensions. Firefox can not uphold Principle 2 without a fix to maintain code integrity.

A way to detect the addition of code to the user code base is required to enforce Principle 1, even when the added code 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 so that it can differentiate them from new extensions yet to be authorized. Furthermore, 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 authorized by the user.

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 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 indicated 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). 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. For added protection, they may be monitored at run time for adherence to additional security policies as described in Sect. 5.

4.2 Implementation techniques

To prevent malicious extensions from tainting the trusted code base of Firefox, we have developed a prototype implementation of user-signed extensions. 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, and
2. *Extension loading* occurring each time a new browser session begins.

During extension installation, our solution assists the user in signing extension code so that it can be safely incorporated into the user code base. During extension 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 (Fig. 3) for each installed extension which can be used for the purposes of authorization and integrity checking. These certificates are composed of two sections:

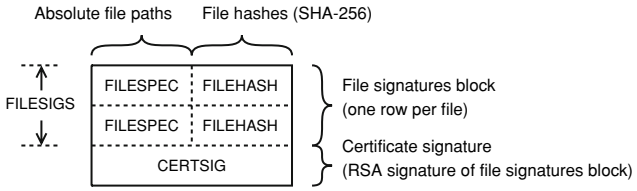

Fig. 3 A user-signed extension certificate, employed to verify authorization and integrity of untrusted code

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

Every file comprised by the extension is represented as 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 its respective FILEHASH, the browser notices if one of the authorized 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. RSA was chosen over better encryption algorithms such as elliptical curve cryptography (ECC) because a mature implementation of RSA already exists within the browser; MD5 was chosen because the hash length is small enough for our implementation to sign it using a 512-bit RSA key. SHA-256, while a more robust cryptographic hash algorithm, generated hashes that were too long. More discussion about the chosen encryption key lengths is presented in Sect. 4.4.

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 occur subsequent to installation.

1. A file is added to or removed from the extension.
2. A file signature is added to or removed from FILESIGS.
3. The content of a file belonging to the extension is modified.
4. One of the file signatures' 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.

4.3 Security analysis

We examine the security properties of our implementation in the areas of private and public key protection, file integrity assurance race conditions, and overall usability.

Key safeguarding Protecting the user from unauthorized modification of trusted extensions requires maintaining the integrity of the extension certificate. The integrity of an authorized extension's files is assured by FILESIGS. The integrity of FILESIGS is assured by CERTSIG, which is in turn assured by the protection of the keys used in signing and verification. Therefore it is necessary to protect these encryption keys, 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. The user is expected to chose a sufficiently strong passphrase such that brute force attacks are unlikely to succeed at breaking the encryption.

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. The cost is a one-time need to authenticate as a privileged user to add this key protection when a user's browser configuration "profile" is created.

Race conditions It is important to ensure that the files whose signatures have been verified during the loading process are the actual ones that are loaded. Otherwise, this opens the possibility for a local race condition attack that would, for instance, replace a verified extension file with a different version before it is loaded, but after the verification step. To fix this problem, the browser extension loader can be augmented to use a mandatory locking mechanism (facilities for mandatory locking are available under Windows Vista or Linux) or use a solution that provides transactional semantics for file operations [16].

Usability It is well understood that a security solution that is invasive or difficult to use will face resistance in user adoption. 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 a passphrase in order to decrypt the private key needed for code signing. The browser prompts the user for this passphrase during installation. This step and the one-time authentication step when the user's browser profile is created, constitute the minimal burden 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. This 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. Our implementation takes this approach. 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 code they are authorizing into their

Table 4 Top 20 most popular extensions from http://addons.mozilla.org tested with our implementation

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

Drawn from the top 23 as we elected to skip platform dependent and non-English language extensions

user code base when they authenticate. If this concern is not addressed, 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 captures of our changes to the installation mechanism can be found on the project website [2].

Another aspect of usability is the integrity checking mechanism performance. We discuss this aspect in the next part of this section.

4.4 Performance evaluation

Our installation integrity assurance 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 4).

Compatibility testing Determining compatibility was done by running the extensions in the test environment and exercising their core functionality while looking for errors. As some extensions provide a large and nuanced feature set, it would have been difficult to exercise their total functionality. In our tests, 18 out of 20 extensions performed flawlessly. One failure occurred in the Forecastfox extension. It elected to force registration of its XPCOM components using an `.autoreg` file, which the browser deletes following registration. The other error occurred in the FoxyTunes extension, which renamed its platform-specific shared library 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 5. It takes the implementation about 17.6 s on average to generate a certificate. The benchmarks indicate over 99.4% 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 808 ms for each certificate to be validated, of which there is one per extension. For validation of the certificate signature, we again observe that, on average, over 92% of the time is spent applying RSA cryptography. It is apparent that the performance impact of the AES and SHA-256 cryptography routines was insignificant compared to RSA. If a faster RSA implementation were employed, the significance of the other two cryptographic functions would likely increase.

Faster integrity checking Our original implementation of the integrity checking mechanism used a JavaScript-based implementation of 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. Due to its nature as an interpreted language, JavaScript is slow running for the computationally intensive algorithms present in RSA. Thus, this implementation suffers performance (and by extension, usability) concerns as explained below.

For a scenario that involves three extensions, note (from Table 5) that the time spent generating certificates is 52.5 s and the time spent on validation is 2.37 s. The certificate generation cost, although it is a one-time cost, is still high

Table 5 Extension authorization and integrity system benchmarks using JavaScript RSA cryptography

Installation/loading performance benchmark	Number of extensions installed				
	1	2	3	4	5
Total time spent generating certificates (s)	17.9	35.2	52.5	70.7	87.0
Average time spent per certificate generated (s)	17.9	17.6	17.5	17.7	17.4
Percent of generation time spent signing certificates	99.4%	99.7%	99.4%	99.6%	99.7%
Total time spent validating certificates (s)	0.85	1.52	2.37	3.14	4.28
Average time spent per certificate validated (ms)	849	760	790	785	856
Percent of validation time spent verifying signatures	87.0%	94.5%	93.6%	94.1%	94.8%

A large percentage of the time was spent performing RSA operations. The test platform was a modified version of Firefox 3.0a5, running on Ubuntu 7.04, on an AMD Athlon 64 X2 3800+ (2 GHz), 2 GB RAM. The extensions evaluated were the top five from Table 4

for this typical scenario to the point where its impact on the user can be clearly perceived when extensions are installed. The main contributing factor to the overhead is the RSA cryptography as quantified in Table 5: over 90% of the overhead is consumed performing RSA cryptography.

To address these performance and usability concerns, we have constructed a second implementation of the cryptographic algorithms needed for signing certificates. This implementation makes use of the Netscape Security Services (NSS) software, part of Firefox's existing code base, to handle the authentication and signature generation/verification tasks. NSS is written in C and compiled into native code for the browser's target platform. Native performance can result in much higher speeds and therefore can increase the usability of our implementation. The primary disadvantage of using a native solution is that it requires a modified distribution of the Firefox browser which contains the integrity checking patches to native code.

We tested our new implementation on the same set of extensions. Resultingly our user code base integrity solution achieved overheads that are almost imperceptible to the end-user, as shown in Table 6.

Browser startup on our test system (using the native cryptography) has an additional delay of about 1/10 s for four simultaneously installed extensions, due to the time spent generating certificates. We contrast this with our JavaScript implementation which required 70.7 additional seconds to start up under the same conditions. With five extensions installed, our native implementation induced around a 1/3 s delay in browser startup due to integrity verification. The JavaScript implementation required an extra 4.28 s to perform the same task.

In Table 7 we compare the performance of the JavaScript implementation against the natively compiled solution. Overall our system functions in excess of 500× faster using the native code. The improvement in time required for signing and validation of certificates also allowed us to double the RSA key length to 1,024 bits, increasing the browser's resilience to attack. This size was infeasible in the JavaScript implementation for performance reasons. At the longer key length, the native solution is still able to sign certificates 2,930× faster. Certificate signing had been the part of our system with the most noticeable overhead, as mentioned previously. The longer key also makes possible an implementation of our solution utilizing a more robust algorithm (e.g., SHA-256) for hashing FILESIGS.

5 Monitoring extension execution

User-signed extensions enable comprehensive protection of the user code base from unauthorized changes, such as installation of a malicious extension without the knowledge or consent of the end-user. With this threat mitigated, the best way to ensure confidentiality and integrity of the user's data is to only allow extensions from highly trusted sources to execute within the browser.

A nefarious extension could, if allowed to run, corrupt the integrity of the user code base or even the browser core by making changes to the browser's runtime state. For example, our BROWSERSPY extension alters the runtime state of the extension manager, a portion of the browser core, to manipulate the displayed list of installed extensions. As noted already, it is a hard problem to statically analyze and understand the actions that will be performed by untrusted JavaScript code in order to take appropriate steps to prevent malicious behavior.

The second phase of our solution therefore endeavors to control an extension's access to critical browser services (XPCOM) via runtime monitoring. (The XPCOM services are discussed in length along with many useful references

Table 6 Extension authorization and integrity system benchmarks using RSA cryptography implemented in C (i.e., native)

Installation/loading performance benchmark	Number of extensions installed				
	1	2	3	4	5
Total time spent generating certificates (ms)	22	47	69	95	182
Average time spent per certificate generated (ms)	22.0	23.5	23.0	23.8	36.4
Percent of generation time spent signing certificates	22.7%	23.4%	23.2%	23.2%	23.6%
Total time spent validating certificates (ms)	50	94	184	242	291
Average time spent per certificate validated (ms)	50	47	61.3	60.5	58.2
Percent of validation time spent verifying signatures	4.0%	3.2%	2.7%	1.2%	2.4%

The overhead is barely perceptible to the end-user. The same test platform and extensions described in Table 5 were used

Table 7 Comparison of two implementations of the extension authorization and integrity system

Installation/loading performance benchmark	JavaScript crypto	C crypto	Improvement factor
Average time spent per certificate generated	17, 620 ms	25.73 ms	685×
Average time spent signing each certificate	17, 540 ms	5.99 ms	2,930×
Percent of generation time spent signing certificates	99.2%	23.3%	4.3×
Average time spent per certificate validated	808 ms	55.4 ms	14.6×
Average time spent verifying each signature	749 ms	1.46 ms	512×
Percent of validation time spent verifying signatures	92.7%	2.6%	35.1×

A JavaScript solution can retrofit existing Firefox installations, though a native, C solution is up to three orders of magnitude faster. The data was drawn from Tables 5 and 6

in [4].) Ordinarily, extensions enjoy unrestricted access to every interface of this framework. Our focus in this section is primarily on the mechanisms and infrastructure needed to implement a runtime monitoring solution that governs all access to the XPCOM interface.

One of the major challenges in implementing a runtime monitor for extensions is described in Sect. 5.1. Implementation techniques are covered in Sect. 5.2, followed by a security analysis of the implementation in Sect. 5.3. Finally the runtime monitor performance is analyzed in Sect. 5.4.

5.1 Problem context

Firefox's default access control security manager for JavaScript implements the *same-origin* and *signed-script* policies. The browser enforces these policies for web content pages, however does not enforce them on internal JavaScript operations. An obstacle in applying web content policies to extension code is browser overlays (explained below), another feature regularly present in extensions. Therefore, a straightforward adaptation of existent enforcement techniques for these policies to protect extensions is not suitable.

The action attribution problem To enforce policies on a per-extension basis, it is necessary to identify the extension requesting each XPCOM service or resource. Unfortunately, Firefox does not have sufficient mechanism in place to establish the requester's identity at each intercepted XPCOM request 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 the information necessary to identify which extension applied the overlay. Through malicious use of the overlay process, an extension can anonymously inject harmful program code into the base functionality of the browser.

5.2 Implementation techniques

To adequately address the action attribution problem, our runtime monitor implementation has two parts. One portion handles policy enforcement for non-overlaid files and the other handles files that are subject to the overlay process. Additionally, a policy specification framework is provided to allow mediation of requests for XPCOM services and resources.

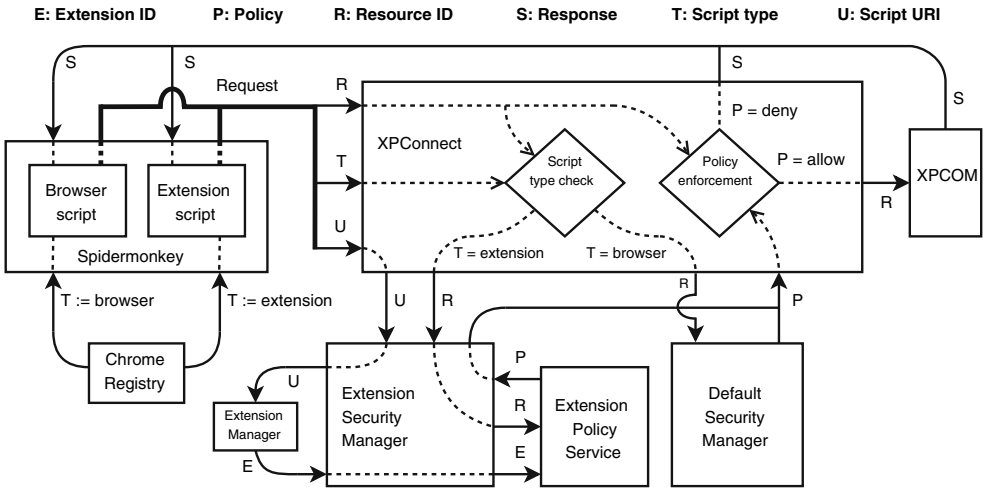

Fig. 4 Policy enforcement using the file name look-up technique: JavaScript programs executing in the Spidermonkey interpreter must access XPCOM resources through a policy-enforcing intermediary, *XPConnect*

Handling non-overlaid files Policy enforcement mechanisms are simpler for non-overlaid files, as the executable statements they contain can be traced back to the extension that issued them. This data is compiled into the set of extensions contributing to any specific operation and used as a basis for policy decisions. The procedure of tracing the origin of a single XPCOM request is shown in Fig. 4, and its performance is discussed later in this section.

Each script loaded into the browser is assigned a type T. A JavaScript statement submits a request $\langle R, T, U \rangle$ to XPConnect for an XPCOM service or resource. If the script type indicates an extension script, a special *Extension Security Manager* is consulted instead of the default security manager. A unique extension identifier E is deduced by looking up the script file name U. If the global and per-extension policies defined in our *Extension Policy Service* allow extension E to access resource R, then the request is forwarded to XPCOM; otherwise a denial response is returned.

Handling overlaid files JavaScript statements present in overlaid files require special handling. When a command is executed from one of these files, the script file name U given to the runtime monitor is the name of the overlay substrate file. This file 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 overlaid 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 bracketed code is then found at the top of the stack. A *try-finally* construct wraps the interposed section 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 [12] 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.

Policies With policy enforcement infrastructure in place, we have implemented six policies on non-overlaid extension code (summarized in Table 8). Complex policies can be composed of these rules to allow only the level of access an extension needs to function. They are representative of the types of policies enforceable using this solution—not an exhaustive

Table 8 The example policies that were created and tested with the runtime monitoring solution

Policy name	What it does	Granularity
XPCOM- ALLOW	Allow all access to a single XPCOM interface	Per extension
XPCOM- DENY	Deny all access to a single XPCOM interface	Per extension
SAME- ORIGIN	Allow network 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

list providing complete coverage of all forms of malicious behavior that extensions are capable of.

The first four policies are extension specific. XPCOM- SAFE and PASS- RESTRICT are conservative policies that disallow access to sensitive data. The policies PASS- RESTRICT and HISTORY- FLOW are enforced globally. The HISTORY- FLOW policy is unique in that it is stateful: if any extension is detected accessing Firefox's history listener interface (`nsIHistoryListener`), that extension will be disallowed further access to `nsIOutputStream` interfaces and their subclasses. This protects writes to files and network sockets over a single session.

5.3 Security analysis

Interposition mechanism defense In cases where the interposition technique is used to trace actions in overlay files, a malicious extension may attempt manipulation of the active extension stack. For instance, it may try 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. Completely 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.

Policy specification limits A common characteristic of the policies we defined is that they are not stateful beyond a single browser session. Consider an extension that collects and stores sensitive data in one session, after which the user terminates the browser process. Later, the user starts up Firefox, upon which the extension retrieves the stored, private information and transmits it over the network. For a policy mechanism to prevent this attack, it must preserve state across browser sessions. The policy state must be stored in a tamper-resistant way such that it can not be manipulated by existing malware on the system. An application of user data signing is one possible solution to this problem which we relegate to future efforts.

5.4 Performance evaluation

We evaluated the performance of our runtime monitor by testing it on BROWSERSPY with policies designed to close the attack vectors it exploits. The implications of this approach are discussed along with the results obtained. Performance overheads were measured to determine the usability impact of policy-based mediation of XPCOM services.

Policy enforcement results We were able to successfully block every surveillance attack employed by BROWSERSPY by implementing the policies PASS- RESTRICT, HISTORY- FLOW and XPCOM- DENY over the XPCOM interfaces it exploited.

In this instance the policy enforcement mechanism proves capable of protecting the user even when a malicious extension is explicitly authorized for inclusion in the user code base. To protect against a wider range of attacks, it is straightforward to apply policies to the complete set of XPCOM interfaces that provide access to sensitive data. Either a blacklist or whitelist approach can be taken to generating the policy rule set.

Doing so may impair normal operation of benign extensions, however. For instance, the FireFTP extension retrieves login credentials from the password manager and communicate that information over the network—a violation of PASS- RESTRICT. More conservatively, a rule set could be implemented that disallows network access to all extensions. This would successfully thwart all surveillance attempts but would conflict with certain extensions' benign network operations. For instance, 12 extensions of the 20 we tested have at

Table 9 Performance micro-benchmarks for the default browser behavior and two different action attribution methods

Extension	Function	Stock (ms)	File look-up (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

The execution time of selected functions within the top five most popular extensions is measured over 1,000 runs. The same test platform described at Table 5 was used

least one feature that requires network connectivity. While it is certainly possible to use these extensions without network access, the particular features that require network access may be unusable if we enforce a policy that denies network access for all of the 20 evaluated extensions. For these reasons it is best to tailor policies on a per-extension basis.

Performance overheads 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, (b) a browser using the file name look-up mechanism and (c) a browser using the interposition technique. The results are summarized in Table 9.

We observed a modest overhead of 8.2% on average to apply our policies using file name look-up. 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 and sparse, 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 code base 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 have integrated our extension integrity checking prototype into a build of Firefox that is currently under review for inclusion in the official release of the browser. Our malicious extension BROWSERSPY is available through private circulation for malware researchers.

Acknowledgments This work was partially supported by National Science Foundation grants CNS-0716584 and CNS-0551660. The first author was partially supported by a Summer of Code internship at the Mozilla Corporation, to work toward incorporating our integrity assurance prototype into the Firefox web browser. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements of the Mozilla Corporation or the U.S. Government.

References

1. eTrust PestPatrol. Pests detected by PestPatrol and classified as browser helper object. http://www.pestpatrol.com/pestinfo%5Cbrowser_helper_object.asp, March 2005
2. Firefox extension security project website. http://alcazar.sisl.rites.uic.edu/wiki/view/Main/ExtensibleWebBrowserSecurity
3. Goldberg, I., Wagner, D., Thomas, R., Brewer, E.A.: A secure environment for untrusted helper applications: Confining the wily

hacker. In: Sixth USENIX Security Symposium, San Jose, CA, USA (1996)
4. Hallaraker, O., Vigna, G.: Detecting malicious JavaScript code in Mozilla. In: 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), Shanghai, China (2005)
5. Kirda, E., Kruegel, C., Banks, G., Vigna, G., Kemmerer, R.A.: Behavior-based spyware detection. In: 15th USENIX Security Symposium, Vancouver, BC, Canada (2006)
6. Kirk, J.: Trojan cloaks itself as Firefox extension. Infoworld magazine, July 2006
7. Lampson, B.W.: A note on the confinement problem. Commun. ACM **16**(10), 613–615 (1973)
8. Li, Z., Wang, X., Choi, J.Y.: SpyShield: Preserving privacy from spy add-ons. In: 10th International Symposium on Recent Advances in Intrusion Detection (RAID), Gold Coast, QLD, Australia (2007)
9. Information from http://addons.mozilla.org
10. Mozilla Firefox at Wikipedia http://en.wikipedia.org/wiki/Mozilla_Firefox
11. Necula, G.C.: Proof-carrying code. In: 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Paris, France (1997)
12. Reis, C., Dunagan, J., Wang, H.J., Dubrovsky, O., Esmeir, S.: BrowserShield: Vulnerability-driven filtering of dynamic HTML. In: 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA, USA (2006)
13. Sekar, R., Venkatakrishnan, V.N., Basu, S., Bhatkar, S., DuVarney, D.C.: Model carrying code: a practical approach for safe execution of untrusted applications. In: 19th ACM Symposium on Operating Systems Principles (SOSP), Bolton Landing, NY, USA (2003)
14. Soghoian, C.: A remote vulnerability in Firefox extensions. http://paranoia.dubfire.net/2007/05/remote-vulnerability-in-firefox.html (2007)
15. Ter Louw, M., Lim, J.S., Venkatakrishnan, V.N.: Extensible web browser security. In: 4th GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assesment (DIMVA), Lucerne, Switzerland (2007)
16. Tsyrklevich, E., Yee, B.: Dynamic detection and prevention of race conditions in file accesses. In: 12th USENIX Security Symposium, Washington, D.C., USA (2003)
17. Ungar, D., Smith, R.B.: Self: The power of simplicity. ACM SIGPLAN Notices **22**(12), 227–242 (1987)
18. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Cross-site scripting prevention with dynamic data tainting and static analysis. In: 14th Annual Network & Distributed System Security Symposium (NDSS), San Diego, CA, USA (2007)
19. Wahbe, R., Lucco, S., Anderson, T., Graham, S.: Efficient software-based fault isolation. In: 14th ACM Symposium on Operating System Principles (SOSP), Asheville, NC, USA (1993)