Security Issues about Web Browser Add-ons

Andreas Holzammer
(aholza@cs.tu-berlin.de)

Seminar "Internet Sicherheit" ,
Technische Universität Berlin

June 6, 2008(SS 2008)

Abstract

The authors of [15] discuss several security issues in Web browser extensions and propose a number of solutions to these. First, they introduce an experimental malicious extension and describe in detail its operation. Some of the mechanisms of this extension could be blocked by the proposed signing of each installed extension through the user. Signatures are checked when code is loaded. Another proposed countermeasure is to monitor execution at runtime while enforcing certain access policies.

1 Introduction

This essay summarizes the publication *Extensible Web Browser Security* by Mike Ter Louw, Jin Soon Lim and V.N. Venkatakrishnan [15]. It was presented at the GI International Conference on Detection of Intrusions & Malware and Vulnerability Assessment (DIMVA) in Lucerne, Switzerland in 2007.

The authors start by implementing an experimental malicious browser extension which they call BrowserSpy. It runs within the Firefox browser. They present a feasibility study of it, including the amount of work required. On the other hand, they investigate techniques for preventing the browser from being hijacked. A signature-based mechanism that prevents extensions from corrupting each other's installations is proposed. They also present a mechanism for monitoring the browser's extension interfaces that allows for the enforcement of certain security policies. A thorough performance evaluation of the *user signed extensions* and the *runtime enforcement of security polices* is done.

Web browsers, the technology discussed in this paper, are widely spread. Every personal computer that is sold with Microsoft Windows has a integrated web browser called Microsoft Internet Explorer. According to a census done at The Counter [1] site, the most used web browsers these days are: Internet Explorer (77%), Mozilla Firefox (16%), Safari (3%) and Opera (1%).

Today users are employing web browsers intensively for sensitive jobs like shopping and home banking or the more benign news reading. The browser handles the sometimes very sensitive information in cleartext, so if the browser is hijacked than private information like credit card numbers, passwords or banking information's could leak.

Almost every web browser supports extensions, called browser helper objects(BHO) by Internet Explorer or extensions by Firefox or plug-ins by Safari and Opera. In the

remaining of this essay we will use the term *extension* for all these add-ons. The authors sustain that Safari does not support extensions which is in contrast to the existence of Safari plug-ins, as in [8]. With such an extension the user can personalize its web browser, add toolbars such as the Google Toolbar or even make the browser show the actual weather in the status bar. One evidence that these extensions are used are the download rates of these extensions. Almost everybody that uses a browser uses a browser extension e. g. Google Toolbar.

It is very easy to install an browser extension. An inexperienced user could install a browser extension by accident, without knowing the consequences.

Malicious extensions like FormSpy or BrowserSpy attack mostly the following two security features that users normally take for granted:

1. *Integrity of the browser code base*: a malicious extension can take over the control of the browser and hide itself after its installed or loaded.
2. *Confidentiality and integrity of user data*: harmful extensions can gather sensitive information and send it back to the attacker, even if this data is elsewhere encrypted when in transit. This attack is well demonstrated by the BrowserSpy.

The authors present a set of solutions that retrofit the browser in order to ensure these security aspects.

The studies we discuss here are based on Mozilla Firefox, which is a popular and open source browser. Access to its code base allows for easy experimentation. Moreover, a lot of documentation about its workings exist. There is less information about Internet Explorer, but the authors also do an assessment of how secure this other browser is.

In Firefox/Internet Explorer all extensions can be disabled, or some Extension, but almost nobody can tell if an extension has malicious behavior or not. To say disable all extensions because they are insecure would be ignoring the actual threat.

2 Related Work

Execution Monitoring The authors use execution monitoring to enhance the security of the browser. Execution monitoring is described later in detail. Execution monitoring is also used in protection against pages with malicious content [11]. This is a very similar problem to ours.

PCC and MCC Another approach than execution monitoring is proof-carrying code (PCC) [12]. PCC is a method to verify properties about an application via a formal proof that accompanies the application's executable code. It is difficult to use for programs that heavily use the eval statement. Normally PCC transforms the original script with runtime checks that enforce desired policies and produce a proof that the transformed program respects the policy. The proof is used to verify the correctness of the placement of the runtime checks. For the solution they present that would mean transformations need to be made before all eval statements.

Another solution is model-carrying-code(MCC) [14], which has a learning algorithm for code behavior that will be downloaded. The problem is the learning process which requires large test suites for exhaustive code coverage.

3 Malware

Malicious browser extensions are widely spread as one can see on the actual list of malicious browser helper objects in the Spyware Encyclopedia [5]. The authors used for this purpose a service that is not anymore online [2], called pest patrol.

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: XPCOM interfaces that can be used for malicious extensions. This table is taken from the paper [15].

It is very easy to program an extension. There are many tutorials on the Internet where is is described how to program a extension [7].

Mozilla Firefox extensions are written in JavaScript and XUL(XML User Interface Language), which makes the extensions platform independent. XUL is a XML user interface markup language, which is used to design Graphical User Interfaces(GUIs). These files are interpreted with the help of Spidermonkey. Spidermonkey is a JavaScript engine that can compile, interpret, decompile and garbage collect.

Browser helper object, which extends the Internet Explorer are created with Visual Studio. Microsoft describes how to create an browser helper object in detail here [6].

It is also possible that a malicious extension can hide from the extension list. A malicious extension can modify another extension to get the malicious code executed undiscovered. The browser does not check the integrity of an extension.

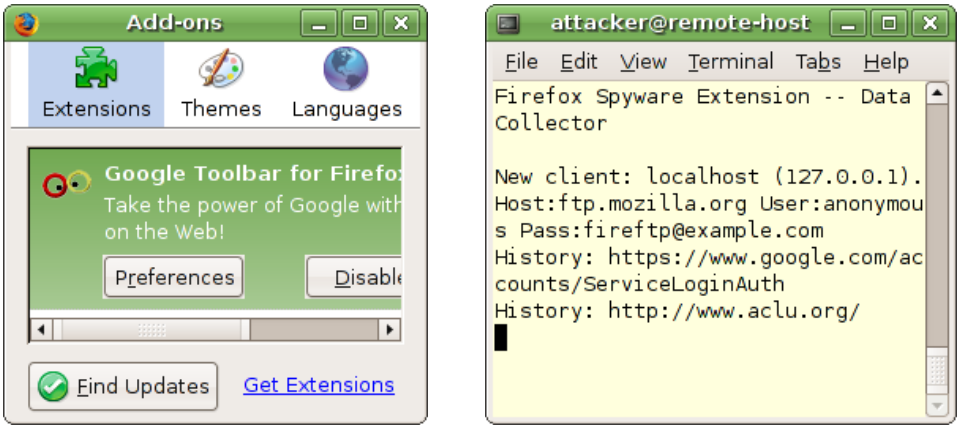
Malware extensions can be installed by the user, but can also installed by an other program, such as an trojan or other malicious program. The Browser does not check if an extension was installed with the Browser or externally.

3.1 Malware under Firefox

It is very easy to program an extension that can steal sensitive data, it took one of the authors without prior knowledge 3 weeks part time to deploy such an extension [15]. It is called BrowserSpy. It is capable to retrieve passwords that are stored Firefox's built-in password manager, retrieve browser history, intercept form fields of credit card numbers, street adresses, Social Security Numbers, and other sensitive information. It injects itself into the Google Toolbar to hide itself. BrowserSpy gets its information through the XPCOM(Cross Platform Component Object Model) framework. The XPCOM framework provides a variety of services within the browser such as file access abstraction, Component management, Object message passing and Memory management. Table 1 shows XPCOM interfaces that could be used to program a malicious extension. It is only needed to change the `chrome.manifest`(a configuration file of an extension where it stores information about loading the extension) to inject code into an existing extension. With no restrictions to access these files it is very easy to inject another extension. Figure 1 shows a screenshot during the operation of BrowserSpy.

3.2 Malware under Internet Explorer

Most BHOs are installed through ActiveX, which makes it very easy to install a malicious BHO. A user just needs to open a Page where such an ActiveX installer is on and the Browser asks if the user wants to install the ActiveX component, and if the user clicks yes, the BHO is installed. So such malicious BHOs can be hidden in normal Sites and the user thinks the BHO is doing other things than what is described on the page.



(a) Extension hiding from the browser UI. (b) Data collector receiving sensitive information.

Figure 1: Two views of the BrowserSpy extension in operation (taken from [15]).

3.3 Installation and Loading Extensions

Before we discuss the security enhancements, we look how the installation and loading of an extension works. It is also possible to install a extension externally, for an extension for Firefox it is only needed to add it to the configuration file and for BHO an addition of an registry key is needed.

Firefox loads all its extensions that are installed without checking the integrity(described below) and Firefox is not checking if the user installed this extension on purpose(authentification, also described below). The Internet Explorer does the same. So it is easy for a malicious program, that is already on the computer to inject himself into the browser in order to access sensitive information. There are many ways that a malicious program gets onto a system, that is not discussed in this paper.

4 Security Features

Before we start to discuss how to protect browsers from hijacked to get sensitive information we will introduce some key futures, after that we will discuss how all these key futures are used to secure a web browser from malicious browser extension.

4.1 Integrity

Integrity checking of code creates signature information of the code to determine changes. The integrity of an extension should be tested if the extension is installed or loaded, so nobody can inject code in this extension. The fact that a file was not modified in time can be checked by comparing of a previous hash summary. Hash functions map large chunks of data to much shorter ones. They behave almost as bijective functions. The most used hashing algorithm is MD5, but it is proven that the algorithm is only partially secure [17]. An other hashing algorithm is SHA256. This algorithm has been studied in 2003 from Gilbert and Handschuh and they found no weaknesses [9].

4.2 Authorization

Authorization is the concept of allowing access to resources only to those permitted to use them. So the authorization process allows a program/person to access data or functions,

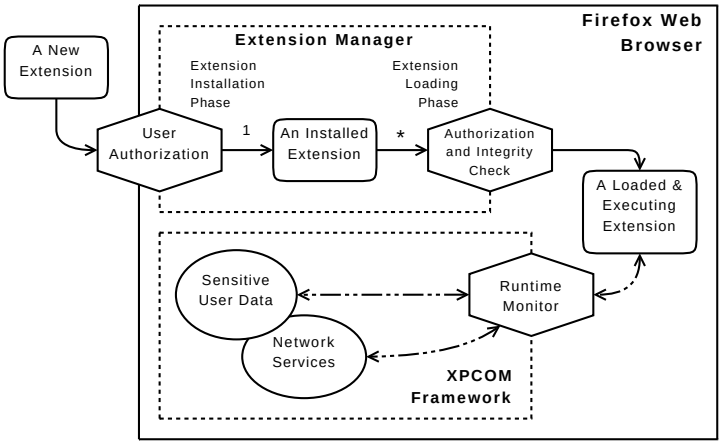


Figure 2: Shows how Extensions are installed and loaded with the Firefox(hexagons represent functionality added to improve security). Figure from [15].

if the authorization process decides it. This means if a person/program authenticates herself, it has access to the data or functions.

4.3 Code Signing

Code signing relies on authentification. Authentification is understood as a verification of the identity of a person or process. In a communication system, authentication verifies that messages really come from their stated source, like the signature on a (paper) letter. So that means for code signing that the code is from the given source and guarantee that the code has not been altered or corrupted since it was signed.

Firefox has a support for code signing but it is not widely used. The authors searched for signed extensions on the central repository at `addons.mozilla.org` and found only two among thousands.

4.4 Static Code Analysis

Static analysis is a method what a program does and/or accesses of a program that is performed without the running program. The static analysis could decide if an extension is malicious or not. This is a good attempt for security reasons and is has no overhead by executing a script. It is only needed to check the integrity of the code, but it is very hard to run a static analysis on JavaScript, e.g. because it has the capability to build up executable code at runtime(using the *eval* statement). Almost half(45%) of the extensions they tested make heavy use of complex eval statements. These programming languages are called *Dynamic programming languages*.

4.5 Execution Monitoring

Execution monitoring is a technique to verify that a program is doing indeed what it is supposed to do. Mostly used is the least privileged method as it is described in this paper [13]. This principle means that a program should run with least privileges necessary to complete the job that the program is supposed to do. There is also the principle of the most privilege. This principle means that the program should have all privileges that to not harm the system/sensitive information. There is much research an the topic execution monitoring for untrusted code. Most studies use the sandbox principle [10] to do execution monitoring, that means they take a secure environment to execute a program.

5 Enforcing Security

5.1 Extension installation and loading

To ensure that the browser can not be hijacked or to get on sensitive information we need to enforce principles to ensure that.

1. Extension can not be installed without user's authorization
2. Ensure that an extension can not inject itself into another trusted extension
3. Protect sensitive data from being accessed/manipulated by extensions.

In the paper from Louw, Lim and Venkatakrishnan [15] they developed an architecture for the Firefox to ensure these principles, as seen in Figure 2. If a extension is installed the browser should ask the user to authorize the extension to be loaded with the browser start and the integrity of the extension should be checked. If an extension is loaded or executed it should be monitored, that the extension does not violate the policies.

To enforce these principles we need authorization, code signing and integrity checking, for the installing process. It is also needed that the user can not inject code into the browsers core, so the user should not have access rights to the browser executables/libraries.

Extensible browsers, like the Firefox, when installed in a secure way should have two different code bases:

1. Browser core, code that is directly from browser executable
2. User code base, add-on code that is loaded if the browser starts from users profile.

The browser core has to have full privileges within the browser. The browser core should be protected from being modified, this can be done if the browser core is owned by the a superuser and the browser is running just in user mode. The user code base can not be restricted that way, because we want to install extensions within a browser session. If the this requirement is met, we have also a security issue. If the user code base is compromised, the sensitive data is also insecure what is proven by BrowserSpy. To secure the code bases, we need the authorization of the user if an extension is installed, and if an extension tries to modify another extension, as already mentioned above.

As mentioned above, Firefox does not enforce this principles yet. The Firefox installation process can be emulated for extensions to load an extension that the user did not authorize. An extension can be modified by another extension without users authorization.

One solution for that problem is code signing. It could be required that all extensions should be signed by a trusted entity. With code signing it can be determined if an executable file is modified or not. Firefox supports code signing, but it is not implemented in a secure way, such that the signature is only checked when an extension is installed and not if it is loaded. It is also a problem that only very few extensions are signed. This means the browser has to be redesigned in this section.

We also have to enforce that a extension is not installed externally by emulating the installation process of the Firefox. That could be done with user signed code. A user signs the extensions that he wants to be authorized for loading, and Firefox only loads the extensions that are signed by the user. This also repairs the problem of not signed extensions by trusted entity.

Louw, Lim and Venkatakrishnan have developed a prototype implementation of user signed extensions. The Firefox code has been modified in these two sections:

1. Extension installation, to check integrity and authorization if an extension is installed or updated

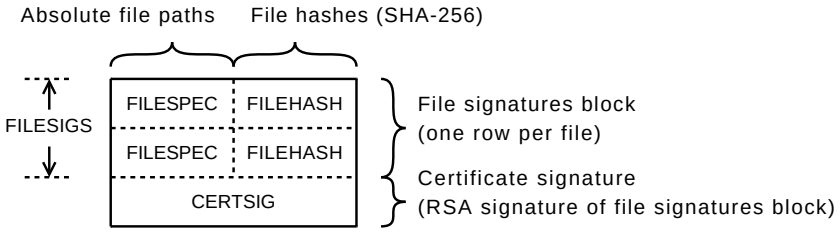


Figure 3: A user signed extension certificate(taken from [15]).

2. Extension is loaded, to check integrity and authorization of extension each time a new browser session begins

During installation of an extension the user is asked(he needs to type his password) if he wants to authorize the extension for loading. During loading of the Firefox, he checks the integrity and authorization of the extension by itself and needs no user interaction. If the integrity and authorization fails the extension is not loaded. A scheme of this prototype is seen in Figure 2.

Now we want to describe the details how all this works. During the installation of the extension a certificate is generated for it(see Figure 3), which consist of file signatures and a certificate signature. A file signature consists of an absolute file path and a file hash(SHA256), to verify the integrity of the extension file. The certificate signature ensures the integrity of the file checksums. The certificate signature is the RSA signed MD5 hash value of the file signatures. The user has to sign the certificate signature via RSA public-key cryptography. With this extension certificate the browser can verify the integrity and authorization of the extension, while loading extensions.

It is necessary to secure the public/private key for the RSA public-key cryptography. If an attacker gets the private key he can break our security system, by signed his own extension without users permission or modify another extension etc. To protect the private key, this key is encrypted by using AES(Advanced Encryption Standard). So the user have to provide a passphrase every time the private key is needed. An other approach is to exchange the public key to sign extensions etc. So we have to protect the public key with superuser privileges, such that it can not be changed by an user account.

5.2 Extension Execution

User extension signing is not enough to secure the browser, because the user could install an malicious extension accidentally. Once installed, the malicious extension can do any action that is provided by the framework XPCOM. We need to restrict the access to the XPCOM framework to secure the browser. Restricting access to the XPCOM framework is discussed in detail by Hallaraker and G. Vigna([11]). Firefox enforces the following policies for JavaScript *same origin policy* and *signed script policy* for web content pages, but not for internal JavaScript operations such as extensions. We have to do runtime monitoring to prevent malicious extensions from using critical operations of the XPCOM framework.

There is a problem with runtime monitoring with Firefox, the Firefox supports file overlays. File overlays allow plug-ins to extend the core functionaries of the browser. With these file overlays the runtime monitor can not determine the origin of the code. If we want per extension policies, we need to know which extension is requesting each XPCOM operation. The browser core has to have full privileges. So we have to split the problem into two parts: for overlay and non-overlay files.

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 2: The example policies that were created and tested with the runtime monitoring solution. This table is taken from the paper [15].

For non-overlay files it is easy to determine the origin of the code as it is maintained by the browser's JavaScript interpreter Spidermonkey.

For overlay files it is a little bit more complicated. The origin that is maintained by the browser's JavaScript interpreter Spidermonkey now points to the target of the overlay, that means usually to the browser core that is overlayed. To fix this problem we need to associate the actions of overlay files with their extension origin. Their approach to this problem is automatic interposition *delimiting statements*, which mark blocks of code with entry and end points. These statements allows us to determine the origin for each code block. This has to be done in the installing process of the extension.

At the opening statement we push an extension identifier to the stack. But at the end of the code we have to clear the identifier. We also need to pop the identifier if the code block throws an exception, so we use a *try-finally* construct.

Spidermonkey is supposed to to the interposition. It has the ability to compile and decompile code to and from bytecode. The decompile function is used to do the interposition. So we first compile the code, after that we run our decompile function to to the interposition.

Six policies on non-overlaid extension code were implemented. They are shown in the Table 2.

The first four policies are extension specific and the remaining are global policies. The XPCOM-ALLOW and XPCOM-DENY policy allows/disallows all access to the XPCOM interface. The SAME-ORIGIN policy allows only access to the same-origin domain. The XPCOM-SAFE policy denies all access to XPCOM interface while SSL encryption is in use. The global policy PASS-RESTRICT denies the access to the password manager. The global policy HISTORY-FLOW prevents that the URL history is saved to a file or send through a channel. Policies could be combined to have a more granular security system that prevents security leaks.

These policies are only a starting point, the authors say that there have to be future studies to enhance the security of browsers.

5.3 Usability

Usability is very important, because if the security enhancements are uncomfortable to use, users will avoid them, like the signing of browser extensions of the Firefox. For the security enhancements for the Firefox there is only little user interaction required, so should be usable.

A problem could be that the user needs to type in his passphrase every time he wants to install a extension. So that means if the user wants to install 10 extensions at the same time he has to type in his passphrase 10 times, that is inefficient. It should be possible to install all extensions in one process. They authors propose that the user has to know about what extensions he is installing, so a list of all extension that the user wants to installed should be shown to the user, before the user types in his passphrase.

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. This table is taken from the paper [15].

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 4: 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 in Table 3 was used. This table is taken from [15].

It is also important that as few as possible extensions are able to run with these security enhancements without showing any problems. They also tested the compatibility with 20 extensions, 18 of these extensions performed flawlessly.

5.4 Performance Evaluation

The performance is also a very big factor if a security enhancement is used or not so we want to look how much overloads we produce for our security enhancements. The Table 3 shows the overheads for the user signed extension technique. The cryptography implemented uses SHA256 for file hashing, 512-bit keys for RSA, 128-bit passphrases for AES and MD5 for hash FileSigs for use in generating CertSig. It takes about 18.7 seconds an average to generate a certificate. The most time is spend on signing the certificate using RSA(over 99.5%). While extension loading it takes 751 ms for each extension to be validated. Again 90% of the time is spend applying RSA cryptography. A JavaScript RSA cryptography method is used to sign/verify certificates. It should be faster if the native RSA implementation of the Firefox is used. The Table 4 shows the overhead for the runtime policy enforcement technique. It is observed: (a) an unmodified browser, (b) a browser using filename lookup mechanism and (c) a browser using the interposition technique on overlayed files. A average overhead from 8.2% is observed for filename lookup. For the interposition method a average overhead from 17.0% is observed.

6 Conclusion

We review [15] that shows that browser extensions have the potential to be a real threat. Although some of them are signed by their original authors, Firefox extensions are able to interfere with each other's code. A mechanism that protects the integrity of browser and existing extension code was proposed, together with another one that monitors the

browser's extension interfaces and enforces security policies. The publication presents a feasible way of consistently improving Firefox' security, with a minimum performance impact.

References

[1] Browser statistics. http://www.thecounter.com/stats/2008/May/browser.php.

[2] etrust pest patrol. http://www.pestpatrol.com/pestinfo2005.

[3] Firefox extension webpage. https://addons.mozilla.org/de/firefox.

[4] Internet explorer addon webpage. http://www.windowsmarketplace.com/category.aspx?bcatid=3500.

[5] List of malicious browser helper objects. http://www.ca.com/us/securityadvisor/pest/browse.aspx?cat=Browser%20Helper%20Object.

[6] Microsofts description of how to program a browser helper object. http://msdn.microsoft.com/en-us/library/bb250436.aspx.

[7] Programming web browser extensions. http://www.libsuccess.org/index.php?title=Web_Browser_Extensions.

[8] Safari plugin webpage. http://pimpmysafari.com/.

[9] Henri Gilbert and Helena Handschuh. Security analysis of sha-256 and sisters. In *Selected Areas in Cryptography*, pages 175–193, 2003.

[10] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, 1996.

[11] 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.

[12] George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[13] Fred B. Schneider. Least privilege and more. *IEEE Security and Privacy*, 01(5):55–59, 2003.

[14] R. Sekar, V.N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, and Daniel C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 15–28, New York, NY, USA, 2003. ACM.

[15] Mike Ter Louw, Jin Soon Lim, and V. N. Venkatakrishnan. Extensible web browser security. In *GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assesment (DIMVA)*, Lucerne, Switzerland, 2007.

[16] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2007.

[17] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *EUROCRYPT*, pages 19–35, 2005.