UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Computer Science

Kristjan Krips

The Security Analysis of Browser Extensions

Bachelor's thesis (6 EAP)

Supervisor: Sven Laur, D.Sc. (Tech.)

Author: .... "........" June 2010
Supervisor: .... "........" June 2010

Admitted to thesis defense

Professor .... "........" June 2010

Tartu 2010

Contents

Introduction

The evolution of Internet has brought browsers, which can be compared to operating systems. Users are allowed to run multiple tasks simultaneously and therefore the browser must control the memory management and scheduling. Additionally, the browser must assure that the computer and the user's information is not compromised. Thus, the browsers applies constraints to the web content. A web content that is rendered inside the browser should not have access to the hard drive and should not have access to modify the way other web pages are being rendered. The focus of the browsers security features has been on restricting web content's rights. This should have changed as browser extensions were introduced. Now, three most popular web browsers include the possibility to extend their functionalities. Internet Explorer, Google Chrome and Firefox allow users to install binary extensions and these extensions have access to the hard drive. The new threat posed by the extensions is not widely acknowledged.

In this work, we analyse the security models of browser extensions. We view the extension models of Mozilla Firefox 3.6, Internet Explorer 8 and Google Chrome 5.0.360. Because browsers are providing functionalities similar to operating systems, we analyse these extension models as we would analyse an operating system. We show that the current security models can be abused with little effort. A browser with a compromised extension may result in the whole computer being compromised. To support our claims, we tested most of the attacks that are described in this analysis. The source code of these attacks is not included in the thesis. Thus, due to previously mentioned risks, we want to stress the importance of the threat that extensions pose to the security of browsers.

The thesis is divided into three parts. The first part describes Mozilla Firefox's security model, the second part contains analysis of Google Chrome's extension model and the third part describes Internet Explorer's security model. Each browser's analysis starts with the description of the extension model. Then we present the weaknesses of current model and show ways of compromising it. The feasibility of creating malware extensions is analysed for each browser individually. Based on the analysis we propose possible attack vectors for each browser. Finally, we suggest ways to improve the current security model and give advice to the users.

Chapter 1

Mozilla Firefox 3.6

Mozilla Firefox is a free open source browser. Mozilla project was started in January 1998, when Netscape publicly released the source code of Netscape Communicator 4.0. With the launch of the Mozilla project, a new cross-platform layout engine named Gecko was released. The Mozilla Firefox project was created as an experimental branch of the Mozilla project. In 2003 Mozilla Foundation was established in order to continue the work on the Mozilla project. In November 2004, Firefox 1.0 was released. The browser was built on Gecko 1.7, a successor to the layout engine that was created by Netscape Communications Corporation. Since then the popularity of the browser has grown and as of March 2010 Firefox holds 24.52% of web browsers market share according to Net Applications [BMS]. The company monitors a network of 40000 web sites over the world and the results are based on an approximate of 160 million unique visitors per month. Because of the browser's modular architecture, it is possible to extend its functionalities and that can weaken the security architecture of the browser.

1.1 Overview of the extensions

Add-ons are software components that the user can add to Firefox. They are divided into themes, extensions and plugins. Add-ons are created by third-party developers who use Firefox's extension framework to add functionality to the browser. Themes change the visual appearance of the user interface.

Plugins are native code libraries, which are used to provide the user with external functionalities. Thus, plugins are binary applications that run inside the browser's memory. Extensions differ from plugins as they extend or modify the browser's functionalities and also need only one set of privileges. Extensions can be written in JavaScript or C++ and they are able to use the XPCOM API [TO].

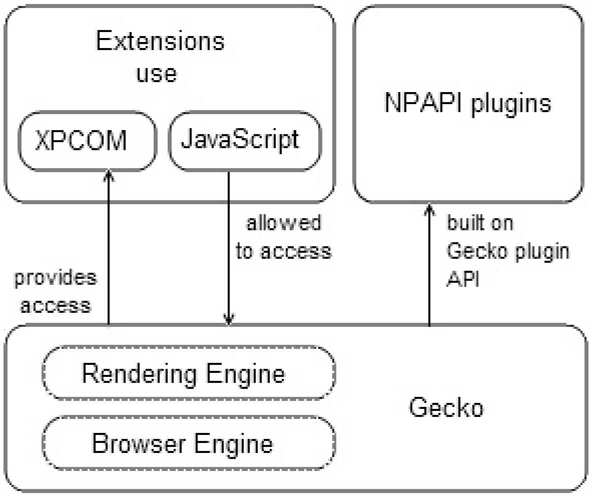
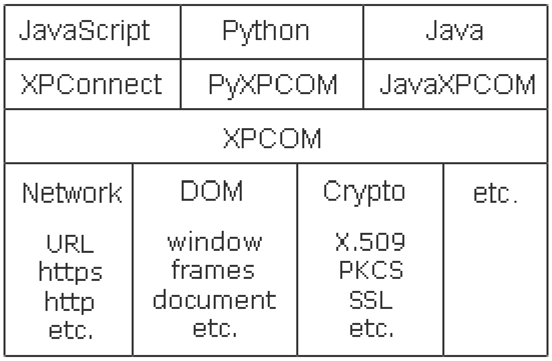


Figure 1.1: Extensions and plugins relation with Gecko.

Browser's high level architecture is depicted in Figure 1.1. Extensions can access the Gecko engine, which is a cross-platform layout engine. Access to the engine is provided through a middle layer named XPConnect, which allows JavaScript to interfere with Cross Platform Component Object Model (XPCOM).

The component object model makes the resources of Gecko available as a series of components, or reusable cross-platform libraries, which can be accessed from the web browser or extension [XPCb]. The functionality of the browser is defined in XPCOM components and accessed by means of those component interfaces. For example the various XPCOM components provide the functionality for navigation, window management, managing cookies, bookmarks, security, searching, rendering, and other features [TO]. Since the XPCOM components are written in C or C++, an additional bridging layer is needed to enable the other programming languages to access these components, see Figure 1.2. Applications that want to access the XPCOM components (Network, DOM, Security, etc.) use a special language binding layer of XPCOM called XPConnect, that reflects the library interfaces into JavaScript. Other language bindings exist for Python (PyXPCOM), Java (JavaXPCOM) and bindings for Perl and Ruby is being developed [Bin].



JavaScript | Python | Java

XPConnect | PyXPCOM | JavaXPCOM

XPCOM

Network: URL, https, http, etc. | DOM: window, frames, document, etc. | Crypto: X.509, PKCS, SSL, etc. | etc.

Figure 1.2: The XPCOM components are scriptable and JavaScript can access these components via XPConnect.

Firefox extensions can be divided into two: binary extensions and extensions that are based on XUL(XML User Interface Language) and JavaScript. The new functionalities are applied by overloading the browser's code at startup or by adding additional binary components to the browser. Extensions that have been installed on Firefox have full browser privileges. An extension has the rights to read, write and execute files on the user's computer. This means that if an attacker can exploit a vulnerability in an extension then he may get the control over the browser. An attacker may also create a malicious extension and trick the user install it.

1.2 Security architecture

Firefox uses single process memory model. All the windows, tabs, plugins and extensions run inside the same process. Threads that run inside the process share the address space and thus there are no borders between threads.

1.2.1 Firefox's memory space and threads

Threads allow multiple executions to take place in the same process environment, so that they are to a large degree independent of one another. Having multiple threads running in parallel in one process is analogous to having multiple processes running in parallel in a computer. However, the separation of threads is weaker than the separation of processes.

Threads inside a process are not as independent as different processes running in an operating system. All threads use the same address space, which means that they also share the same global variables. Thus, a thread can access every memory address within the process's address space, see Figure 1.3. Every thread is able to read, write, or even wipe out another thread's stack. Different processes may have different owners and run with lower or higher privileges and therefore a process needs to be protected from other processes. Contrary to processes, a thread does not need protection from other threads as threads run with the same privileges and have a single owner. Thus, there is no protection between threads. Therefore, in addition to sharing an address space, all the threads share the same set of open files, child processes, alarms, and signals, etc [Tan01]. For further details we refer to the handbook [Tan01].

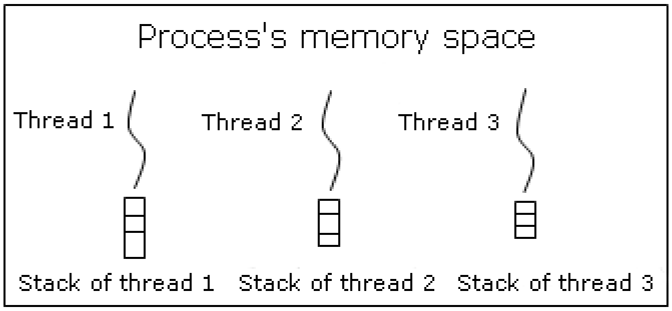


Figure 1.3: All threads inside a process share the same memory space, therefore a thread can access another thread's stack.

To summarise, a thread inside the Firefox process is able to read and write to the memory addresses outside of the thread's memory. Because of that, it is possible to make an extension that is able to read and modify the memory of another extension. The developers have tried to create borders between threads because of the thread safety issues. This is done in the API level, for example, calling the plug-in API is supported only from the main thread. Background threads are not allowed to modify the main thread, requests that wish to modify the user interface are handed to the main thread. This is implemented to protect the browser from crashing [Thr]. It is possible to use a XPCOM component to have a call execute in another thread, but this can lead to a crash [Tur].

Despite that there are some borders between threads, extensions can use all the APIs the browser can. If an exension runs in a background thread, then it can pass the request to the main thread. Thus extensions can do everything the browser does. Therefore, the current memory system does not separate extensions nor provides a way to handle privilege management. UC Berkeley security group examined the extensions privilege management in the API level [Fel09]. They examined the use of APIs in 25 extensions recommended by Mozilla. After examining the behavior of the extensions, they determined that only 3 of the 25 extensions actually needed access to the most powerful capabilities of the Firefox extension system. Then they compared extension behavior to the interfaces used to implement it and found a privilege gap between the desired functionality and the actual interfaces [Fel09]. This shows that by the current model extensions have higher privileges than are required for their functionalities.

With the current APIs, reducing the privileges of extensions in the Firefox extensions system is difficult because the functionalities needed for an extension are accessed through interfaces that give access to different privileges [BFSB09]. For example, Firefox does not have a low privilege file storage interface and thus extensions use a file system interface, which grants read and write access to arbitrary files. Also, an extension that stores its preferences using a preference service is able to modify the preferences of the browser and other extensions. For that reason it is difficult to design an extension with limited privileges.

1.2.2 Firefox's scheduler

As Firefox is a multitasking system it owns a scheduler, that provides available processing time to the browser's tasks. Local threads are scheduled within a process only and are handled entirely by The Netscape Portable Runtime (NSPR) API. Firefox also supports global threads, that are scheduled by the host OS and correspond to native threads on the host OS.

NSPR threads are scheduled by priority and can be preempted or interrupted. These threads are interruptible, with some constraints and inconsistencies. To interrupt a thread, the caller of `PR_Interrupt` must have the NSPR reference to the target thread. `PR_Interrupt` requests that a thread would stop performing its task and return to a control point. When a thread is interrupted, it is rescheduled from the point at which it was blocked. A thread may be interrupted only if it is waiting on a condition variable or waiting on I/O. In the latter case, interruption does cancel the I/O operation. In neither case, when a thread interrupted, it is not terminated [NSPa]. A thread can be shut down by calling its shutdown method from another thread. This stops events from being dispatched to the thread, but any pending events will run to completion [nsI].

For the implementation of NSPR, different strategies are used on different platforms. On some platforms the NSPR threads map directly to the native threads on the platform, while on others NSPR supports both threads that are scheduled by NSPR and the native threads. NSPR version 4.8.3 uses pthreads library on all Unix platforms and on Windows platforms NSPR multiplexes user-level threads on top of native, kernel-level threads. This model is also called a combined MxN model, with Windows threads and fibers [NSPb]. It means that there are many local threads inside a Windows thread and these local threads are scheduled by NSPR.

To summarize, the scheduling is not done by the operating system and if Firefox gets compromised, then also the scheduler gets compromised. Thus, if a malicious extension is installed, then it is able to control the execution of a random thread inside the Firefox process.

1.2.3 Firefox's code space

By default the contents of Firefox's installation folder can be modified only with administrative rights. This folder contains the browser's source files and globally installed extensions and plugins. User profile files are located in a folder that is accessible without administrative privileges, see Figure 1.4. This folder contains user's bookmarks, preferences, passwords, extensions, etc.

Most of the browser's functionality is made available through the use of XPCOM components. The official way to add additional XPCOM components to the browser is to create an extension with the components inside a components directory. With this approach the user will be notified about the modifications and will be able to disable the component from the extensions manager. Also, old components will not be installed on newer Firefox versions if the extension is not compatible with the newer Firefox's version.

Firefox's code space in most part is not protected from modifications. It is possible to modify the user interface files, the configuration files and extension files. Before Firefox 3.6 it was possible to add binary components to Firefox's components directory and these components were automatically loaded at startup. This was a security issue as these new components were invisible to the users [Nig]. There was also a problem with Firefox crashing after an update from version 3.0 to 3.5 because the old binary components were not compatible with the new version.

This was somewhat fixed by adding a whitelist of approved components, that the browser is allowed to load automatically. The whitelist is implemented as a text file, containing the names of the allowed binary components. Each time the browser is updated, a new components list is created, which negates any unallowed changes in the file. It should not be possible to load components that are not in the whitelist, because it is a security risk. The current implementation of the whitelist allows a third party program to add binary components to Firefox the same way as it was done before. It is possible because the whitelist is not protected from modifications.

To test this hypothesis we created a new binary component and tested the functionality of the whitelist's implementation by adding a new component to the current whitelist. We saw that the current whitelist does not provide additional security as it is possible to force the browser to load the new component at startup.

To carry out this attack, a third party program must be installed on the computer that carries out the component installation. The program must have read and write access to the Firefox's installation directory, which means that administrative privileges are required. This is required to modify the whitelist and to register the new component after the whitelist has been changed. Firefox is forced to check for new components when it starts if its `.autoreg` file is modified. We added our component to the whitelist and then we modified the autoreg file to force Firefox to scan the whitelist. Using these steps we were able to quietly load our new binary component the next time the browser started. We also found out that all new components are whitelisted if the file containing the whitelist is removed from the components directory.

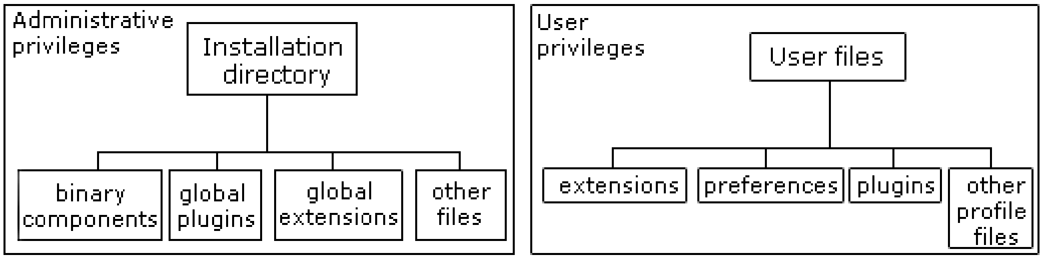


Figure 1.4: The overview of Firefox's code space.

These security risks could be abused to compromise the browser in a manner that the user would not notice. There is little that the user could do against the silent component installation because most users will not check, which binary components are installed. If a threat abusing the current model gets popular, then it is possible to block it. The developers of Firefox have created blacklist of components that the browser cannot load. The blacklist is renewed with every browser update. With this approach it is possible to add malware components to the blacklist of the next update, which would break the malware. This approach can be avoided by the malware if it disables the updating of the blacklist. Thus, the method is effective only in preventing the installation of malware components.

The files of extensions, plugins and preferences are not protected and can be overwritten by the user. This means that a third party application is able to modify the functionality of the browser. For example, it is possible to change the functionality of the extension by modifying its source files. Thus, a user cannot be sure that the installed extensions function the same way they did after their installation. This can lead to an information leak if a compromised component is placed on the user's computer or if an existing extension's functionalities are modified.

1.2.4 Update system

Before Gecko 1.9 (before Firefox 3), the extension update system was partially vulnerable to a man-in-the-middle attack. Man-in-the-middle attack allows an attacker to listen the traffic between the user and the server and intercept it. For example, this attack could be done by hijacking a public Wi-Fi network, because the connection between the computer and the public Wi-Fi router is usually not encrypted and thus unsafe. Therefore, the attacker may be able to impersonate a server and send false information to the user. The vulnerability of the update system was possible because the updates were sometimes delivered via an insecure protocol, without using further security measures. This allowed an attacker to intercept traffic in a public Wi-Fi hotspot and interfere with the update process. Gecko 1.9 introduced security measures to solve the problem. Now, every extension that uses an insecure protocol for updating, must have a digital signature in the update manifest. If such an extension does not have a digital signature, then Firefox will not check for updates for that extension. Also, a https page must use a valid signature for the update process to succeed. To verify the integrity of the downloaded extension, the update must be hashed. More specifically, the update information is hashed using a SHA-512 hashing algorithm and the hash is digitally signed with the extension creator's private key [HAS]. These security measures will prevent an attacker from succeeding in a man-in-the-middle attack against the update process.

1.2.5 JavaScript sandboxing

By current design, JavaScript that is used in an extension has the same privileges as the extension. This could lead to a security breach, as privileged JavaScript may interact with a hostile web page. A hostile web page might use the design flaws of an extension and inject malicious code to the web page, which could result in running the attacker's code on the user's computer. Therefore, it is advised to sandbox the JavaScript that the extension injects into web pages.

Applications and extensions that inject JavaScript to the DOM of untrusted (web page) content need to be secured against hostile web sites. As a web site's DOM might contain malicious content, the extensions must make sure that the information that they use is really coming from the DOM API and not from JavaScript properties, getter functions, and setter functions defined by a malicious page [Saf]. Since Firefox 1.5, an API (`Components.utils.Sandbox`) exists to evaluate JavaScript code with restricted privileges. Code that is running in this sandbox will always execute with restricted privileges, as on a normal web page [eva]. Still, privileged JavaScript has to access some parts of unprivileged web content. By creating a sandbox and then loading a document into an `XPCNativeWrapper` within the sandbox, an area can be made, in which JavaScript has restricted privileges. `XPCNativeWrapper` is a way to wrap up an object, so that it's safe to access it from privileged code. It limits access to the properties and methods of the object it wraps, thus preventing a web page from redefining extension's methods and properties [XPCa]. Extension that needs to use a sandbox to run user scripts should create a sandbox area, add a document to the `XPCNativeWrapper`, load JavaScript from local files (or generate it) and finally use `evalInSandbox()` to run that code on the document. To use `evalInSandbox()`, first a sandbox object must be created using its constructor, `Components.utils.Sandbox`. The sandbox must be initialized with an origin URI, a DOM window or a `nsIPrincipal` object. Manually creating `XPCNativeWrapper` objects by using its constructor is necessary only if the extension is designed to work on versions prior to Firefox 1.5. Newer Firefox versions automatically wrap privileged JavaScript whenever it accesses less privileged objects. The code that runs using `evalInSandbox()` will be able to call API functions, so it must be protected from JavaScript running in the original web page: for this reason many functions in the sandbox are unavailable [eva]. This kind of sandbox allows an extension to access certain parts of the hostile environment like DOM, but it does not allow malicious scripts to interfere with privileged scripts or intercept references to privileged functions. However, security problems may arise when using `evalInSandbox()`. It may not be safe to use properties of the objects that are executed in the sandbox. Also, it is not safe to call privileged functions from within the sandbox.

1.2.6 Blocklisting extensions

Firefox has a built-in security feature, that should protect the users from installing compromised plugins and extensions. A blocklist of extensions and plugins has been created. This list should contain all known extensions and plugins that are vulnerable to an attack or that have been created by attackers. The list is updated daily and if a match is found, then the extension or plugin will be disabled and the user will be notified about the threat. This approach can protect the users from installing widely known threats and insecure extensions, but it does not provide any protection after an extension has been installed because the extension can disable this feature. Thus, if a compromised extension is installed, it can guarantee that no new blocklists will be downloaded. The compromised extension can modify the preferences file in the user's profile folder and set `extensions.blocklist.enabled=false` [Blo]. Additionally, an extension is able to modify the `blocklist.xml` file and can allow dangerous extensions or plugins to be installed. The extensions should not have the right to disable this feature in order for it to be successful and provide protection against malware.

1.3 Attack scenarios

1.3.1 Creating a keylogger

An extension can be used as a keylogger. It is fairly simple to write a few lines of JavaScript that logs every keystroke on the browser window or in dialog boxes. Using `XMLHttpRequest` it is possible to send the collected data to a listening server. What makes this dangerous is the simplicity to implement it. It is not required to create or to install a new extension to add the keylogger to the browser. An attacker or a third party software that is able to modify the installed extensions can add additional functionality to already installed extensions and thus compromise them. This is dangerous as the user will not be able to detect the change, extensions will not loose their functionality and no new extensions appear in the extensions manager. Besides compromising the browser, it is possible to compromise the whole computer. Firefox's extensions are not sandboxed, they have privileges to write to the hard drive and to read from the hard drive. An extension can contain a binary component that is able to listen to every keystroke made when the browser is running.

We tested if it would be possible to add a binary file to an already existing extension and it was. A third party application or an attacker can add binary files to already installed extensions without leaving a trace to the user. This is possible as by the current model extensions load all binary files that are included in their components folder. An attacker who is using these methods can probably collect information about the user without getting noticed. Using virtual keyboard will not protect the user from these keyloggers as the keystrokes are intercepted from the browser. The keylogger does not use operation system APIs to hook itself to other programs, it is based only on JavaScript. Thus, firewalls or antivirus programs will not detect this keylogger because it is a simple JavaScript extension that has become a part of the browsers functionality and runs inside the browser's process. Therefore, if Firefox is not blocked by the firewall, then the keylogger is also not blocked, as they use the same port. The keylogger script that is part of the extension cannot be distinguished from safe extensions, as an extension is allowed to use JavaScript and send information to a remote server.

1.3.2 Website defacement

While browsing a web page, users believe that the content they see is authentic. This is why attackers would like to modify the content of web pages. An attacker could alter the information displayed or change the behavior of the web site and users would likely not notice the difference. In Firefox it is possible to change the way a web page is being displayed while it is being loaded. JavaScript with a specific event listener is enough to change the content of a web page being displayed. `DOMContentLoaded` is the event listener that could be used. Fired on a `Window` object when a document's DOM content is finished loading, but unlike "load", does not wait until all images are loaded [Gec].

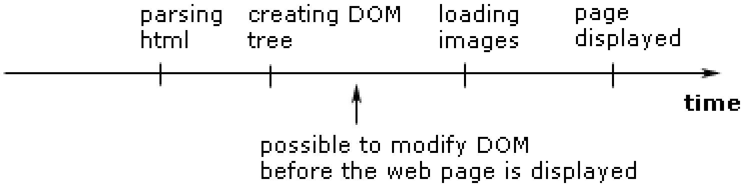


Figure 1.5: This scheme shows how a web page is being loaded.

We made an extension to test if we could use these methods for changing the way an https page is being displayed. We wanted to test, if it would be possible to add or remove content from an https page. The results revealed that after the DOM had been loaded, it was possible to modify the content before it was displayed. The extension was able to both add and remove content. The result shows that website defacement is achievable and that Firefox does not guarantee that a verified web page is always being displayed correctly. This property can be used for creating a man-in-the-browser attack, which is described in the second chapter. Implementing the attack in Firefox is identical to the one described for Google Chrome.

1.3.3 Phishing attacks

Phishing is a type of fraud, which tricks users to give away sensitive information, for example usernames and passwords. Usually phishing is done by directing the user to a fake web site, which seems identical to the legitimate one. This could be used by an attacker for gathering confidential information. For example, the attacker could get the login details if the user would try to log into his account on a fake web page. To be able to trick a user to think that a web page is secure, it is needed to make him see no difference between the real page and the fake one. Firefox uses several methods to change the user interface so that the user sees certain icons or extra information when using a secure site. Firefox classifies web pages into three categories: pages with no identity information, pages with basic identity information and pages with complete identity information. The last two are both encrypted pages whose domain has been verified, but the latter also has the information about the owner of the site. To visualize the categorization, it provides a colored button on the left side of the address bar since version 3.0, see Figure 1.6. This area is called the site identity button. If the page has no identity information, then the button is colored grey, when it is verified and uses encryption but does not have the information about the owner it is colored blue and when the page has complete identity information it is colored green.

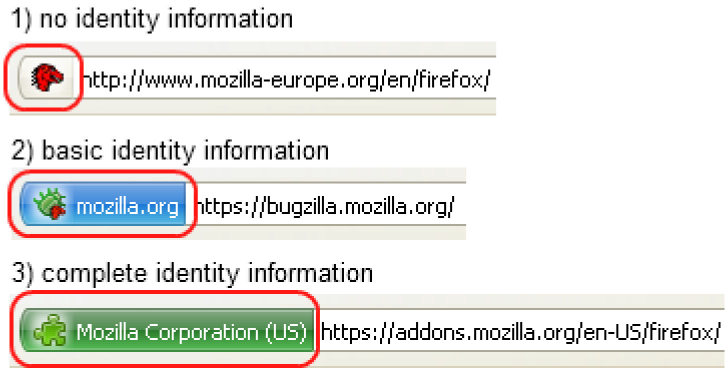

1) no identity information
http://www.mozilla-europe.org/en/firefox/
2) basic identity information
mozilla.org
https://bugzilla.mozilla.org/
3) complete identity information
Mozilla Corporation (US)
https://addons.mozilla.org/en-US/firefox/

Figure 1.6: It shows how the color of the identity button changes depending of the available identity information.

It should not be possible to change the color of the site identity button, as this is the first thing, which makes the user notice that he is using a secure site. But using only a few lines of code, it is possible to overlay the site identity button's background color to make it look identical to the ones used on secure sites.

Another security feature is the padlock icon, which is shown on secure sites. Firefox places the icon on the right side of the status bar. There are no restrictions on the extensions overlaying the skin of the browser. It is possible to disable the icon with one line of code and add a false icon with a few more lines, see Figure 1.7.

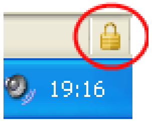


Figure 1.7: A padlock image is created on the status bar every time an https page is being loaded.

1.3.4 Stealing saved passwords

Firefox allows users to save passwords in order to make browsing more comfortable. Most users are not aware that these passwords are saved to an encrypted file on the user's profile folder and that the file that contains the keys necessary for decrypting these passwords is inside the same folder. These files can be copied to a new profile folder on another computer and then viewed using the browsers password manager. An extension is able to read the contents of these files and send the information to a remote server and thereby reveal the saved usernames and passwords to a third party. There is a way to avoid the leakage of passwords by using a master password. If the user has set a master password, then the files of passwords and keys are encrypted with another key. When a master password is used, then it must be entered every time, when the user wants to use the saved passwords. This eliminates the convenience of not entering passwords. For that reason, the feature of protecting already saved passwords is not widely used. Even if the user protects his passwords, there is a chance that a compromised extension is able to read the saved passwords. The compromised extension might read the keystrokes, find out the master password and send the collected strings with the encrypted files to a remote server. An extension could also read the password from the filled forms, that are shown after the master password is used to access a password protected website. Considering these weaknesses in the password protection, it is not advised to save passwords. Passwords that are not protected can be viewed in a few seconds by anyone who is able to access the computer. Even if the passwords are protected, then there still remains a threat from the compromised extensions. Disallowing the password file to be decrypted on other computers would protect the user's passwords from being copied. Besides that, Firefox could use the password managers that are provided by operating systems.

1.3.5 Using Firefox as a botnet

Firefox could be used as a botnet client. Most difficult in creating this kind of a botnet is installing botnet extensions to thousands of computers. If this could be done, then the extensions could guide the browser on behalf of the botnet owner. The extensions could read the instructions from stated websites. This can be done by using `XMLHttpRequest`, that queries the contents of a html page without loading it. The received html can be parsed and instructions can be found. After reading the commands the extension could start to send spam following the instructions or to start requesting a certain URL to perform a DoS attack. DoS attack could be done by using `XMLHttpRequest`. The website, which is being queried, has to allow `XMLHttpRequests` or the results are not returned. If the website does not allow `XMLHttpRequests`, then the requests still go through, making it possible to clog the website. As the botnet client would run inside the Firefox process, it would not be found by detecting software. Also, a Firewall would not stop it because Firefox is allowed to pass it. The botnet client could be optimized to utilise only a part of the computer's resources, to remain unnoticed.

1.3.6 Risk assessment

Considering the threats we classified them by severity ratings provided by Mozilla.

- Critical - It is possible to run code with user privileges and without the user's knowledge.
- High - Access to confidential data and ability to inject data without the user's knowledge.
- Moderate - Access to sensitive information that does not expose the user or organization to immediate risk.
- Low - Leaks of non-sensitive information.
- DoS - Temporary denial of service attacks that may result in the crash of the application.

	Changing the functionalities	Keylogger	Website defacement	Phishing attacks	Botnet client	Faulty extensions
Critical	x					
High				x		
Moderate		x			x	
Low			x			
DoS						x

1.4 Ways for compromising Firefox

1.4.1 Cross-Site Scripting

Cross-Site Scripting attacks are a type of injection problem, in which malicious scripts are injected into the otherwise benign and trusted web sites. Cross-site scripting (XSS) attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user [XSS]. According to the presentation of Roberto Suggi Liverani and Nick Freeman, any input rendered in the chrome is a potential XSS injection point [LF]. XSS in chrome is privileged code, so there are no same origin policy restrictions [LF].

The Firebug extension versions prior to 1.04 had vulnerabilities, which allowed running arbitrary script code in chrome. Firstly, the input passed to the `console.log()` function was not properly sanitised and could have been exploited to execute arbitrary script code within the "chrome": context by tricking a user into visiting a malicious website. Secondly, results of the `toString` method when processing function objects were not properly sanitised before being used. This could have been exploited to e.g. execute arbitrary script code within the "chrome:" context by overriding the `toString` method with a specially crafted function [Fir].

This means that some extensions may have security holes, allowing a web page to inject scripts, which could alter the behavior of another web site. Usually cross-site scripting is restricted by the same origin policy, which does not allow a script to access the methods and properties of a different site, but the code loaded from chrome is privileged with no restrictions on XSS. Because Firefox's extensions do not run inside a sandbox, it is possible that an insecure extension could allow an attacker to take control over the whole computer.

1.4.2 Installing a compromised extension

A third party software or a trusted person is able to install extensions. For example, the cleaning staff or computer repair team may have access to the browser. The extension that is installed may camouflage itself and behave as a well known extension. Very few of the extensions are currently signed, the user is not able to verify the origin of the installed extension. As a result of limited identity information the change may stay undetected.

1.4.3 Modifying the installed extensions

We have demonstrated that is possible to modify the functionalities of installed extensions. An attacker using this property can hide the malware's code inside the existing extensions. A third party software intended to compromise the extensions could be used. The user might detect the third party software and remove it, but probably would not detect the change in the extensions directory. Such an attack vector would compromise the computer, mislead the user and leave the threat undetected. As the malware has become a part of the browser, the detection programs will not find the infection.

1.5 Solutions

The easiest solution for these problems is to use the browser in safe mode. Namely, when Firefox is started in safe mode, it disables all extensions and thus there is no threat from the extensions. But that approach would destroy the current functionalities provided by the extensions. This solution would provide security only to a fraction of users, as most are not aware of the threats posed by extensions.

If leaving aside the safe mode, the current extension model needs to be made safer. The survey of Firefox extension API use by Adrienne Porter Felt showed that extensions use too powerful APIs [Fel09]. These APIs provide access to file system and the possibility to execute files. This kind of behavior is required by a minority of extensions. Thus, the extensions have access to the APIs, which they do not need to use. Currently, it is not possible to allow an extension only partial API access. To give the extensions lowest possible privileges, current APIs would have to be redesigned.

It should not be possible to change the functionalities of already installed extensions. One solution would be to create a hash for every extension hosted on the official Mozilla's extensions website. Every extension would include a hash code, which would be compared to a corresponding hash in the Mozilla's server every time the browser is started. A secure protocol would be used to compare the hashes. This solution would make the startup of the browser a bit slower but would compensate it by guaranteeing that the extensions downloaded from the official website can not be modified.

Extensions should not have the right to modify the content of an https page. Users should always feel safe when browsing trusted and encrypted web sites. The API should not allow the modification of DOM on sites that are using https. The downside of this would be the impossibility to use extensions that modify DOM for blocking advertisements, as they are also a part of the content of the page. This is a sacrifice the users would have to make in order to receive the guaranteed content.

Chapter 2

Google Chrome

Extension support for Windows platform was added to Google Chrome in version 4.0. At the same time, extensions were enabled for Linux and since version 5.0.307.7 extensions are available for Mac OS X 10.5 or newer releases. In this chapter we analyse the extension model of Google Chrome version 5.0.356. The analysis is applicable for all Google Chrome versions that support extensions.

2.1 Overview of the extensions

Extensions are small programs that can modify and enhance the functionality of Google Chrome. They can be written using technologies like HTML, JavaScript, and CSS. Extensions are essentially web pages, and thus they can use all the APIs that the browser provides to web pages, for example `XMLHttpRequest`, `JSON` and `HTML5` local storage. Extensions are allowed to modify the user interface of Google Chrome by using browser actions or page actions. Browser actions allow to put icons in the main Google Chrome toolbar. Page actions allow to put icons inside the address bar. Page actions represent actions that can be run on the current page, but that are not applicable to all pages. Extensions can interact with some browser features, such as bookmarks and tabs. They can also interact with web pages or servers by using content scripts or cross-origin `XMLHttpRequests` [GCO]. Extension's security architecture differs from the architecture of the plugins. Plugins are written using Netscape Plugin Application Programming Interface (NPAPI), which is a cross-browser API for plugins. Each plugin is a binary that runs in its own process, but the process is not sandboxed. This is because plugins need to access the operating system. Google Chrome also supports binary extensions. A NPAPI plugin can be bundled into an extension, which allows to call into native binary code from JavaScript. Code running in an NPAPI plugin has the full permissions of the current user and is not sandboxed or shielded from malicious input by Google Chrome in any way [GCN].

2.2 High-level architecture

Until recently, web browsers could be compared to single-user, co-operatively multi-tasked operating systems. In such an operating system a poorly designed application or a hung process could bring the whole system to a halt. Web browsers that use a single-process architecture face the same problems. If a web browser runs all of its components in one process, then the crash of one component can cause the crash of the whole browser. A misbehaving web page, extension or a plugin can take down the entire browser and thus all of the running tabs [GCA]. Google Chrome uses separate processes for browser tabs to protect the overall application from bugs and glitches in the rendering engine. It also restricts access from each rendering engine process to other rendering engine processes and to the rest of the system. This brings to web browsing the benefits similar to memory protection and access control in operating systems.

A main process controls all other processes. This process runs the user interface and manages the extension, plugin and rendering processes, see Figure 2.1. By default, each instance of a web page runs in its own renderer process, which guarantees that web page is not able to modify the way another web page behaves. Different subdomains are considered as a part of the same web page to allow JavaScript access between the subdomains as defined in the Same Origin Policy, see [GCP]. The number of renderer processes is limited by the computer's resources, the average limit is 20. If the limit is reached then every succeeding tab will share a process with a randomly chosen renderer process. This behavior somewhat weakens the process separation model, but the impact is not severe as the shared process will be chosen randomly. The process limit can be avoided if a limited number of tabs is used.

Chromium's rendering engines are executed within a sandboxed process, thus limiting access to the user's computer. These sandboxed processes do not have direct access to the user's filesystem, display, or most other resources. Therefore, if a rendering process gets compromised, it cannot compromise the computer. The sandboxed renderers can gain access to permitted resources only through the browser process, which can impose security policies on this access. As a result, Chromium's browser process can mitigate the damage that an exploited rendering engine can do [GCP].

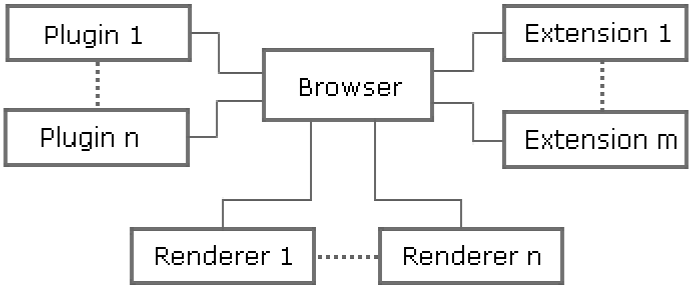


Figure 2.1: Google Chrome's main process manages the other processes.

2.3 The security architecture of extensions

The security architecture gives to the extensions minimal required privileges and restricts access to the file system. The idea is to limit the possible exploits of an extension's design that would allow a web page to take over the browser. More precisely, every extension runs in a separate operating system process. Such a process is isolated from the other operating system processes and thus an extension cannot compromise the browser's kernel. As every web page runs in a sandboxed process, it is not easy for a malicious web site to compromise an extension. Besides that, every extension contains a manifest file named `manifest.json` which describes the extension's privileges. This file is located in the user's extensions folder and is therefore accessible without administrative privileges. It is not digitally signed and thus can be modified. The manifest file defines with which web sites the extension is allowed to interact. The limited privileges will not allow an attacker to exploit the security holes of an extension to access content it was not meant to use.

Every extension is divided into two basic blocks: content scripts and a background page. Content scripts are files that run JavaScript in the context of web pages. Background page is an invisible HTML page that runs in the extension process and holds the main logic of the extension. For example, it may contain a long-running script to manage some task or state, for further details see [GCB]. In addition to a background page, an extension may contain other HTML pages, for example a browser action can contain a popup that is implemented by a HTML page.

The division into content scripts and background page is done to achieve further privilege separation. Content scripts are allowed to interact directly with the web pages and to modify the DOM, but they cannot use most of `chrome.*` APIs. In addition, content scripts cannot make cross-site `XMLHttpRequests` and cannot use the variables or functions defined by web pages, by other content scripts or extension pages. A content script that is injected into a web page is not able to see any other JavaScript executing on that page. The reverse also holds, JavaScript running on the same web page cannot call any functions or access any variables defined by the content script [GCC].

Background pages can use the `chrome.*` APIs, but are not allowed to directly contact the web pages. Content script needs to communicate with the background page in order to ask it to use the privileged APIs. Communication between extensions and content scrips is implemented by message passing, see Figure 2.2. There is an API that provides the message passing capabilities. This implementation limits the content scripts privileges while maintaining the functionalities of the extensions [GCM]. For example, a content script is able to collect data from the DOM, but it needs to ask the background page to use `XMLHttpRequest` to send this data to a remote server.

Every extension package is forced to be signed by the creator and for that a unique key pair is assigned as the extension is packaged. This is done in order to guarantee secure update process. A new version of the extension must be signed with the assigned private key. Every extension contains an update URL and the assigned public key in a manifest file, which is used to verify the origin of the update. For that reason the update can be received via an insecure connection. When Google Chrome checks for updates, it will request a XML document from the update URL and make sure whether the document refers to a newer version of the extension. If a newer version is available, then the browser will download it and check if the extension package is signed with the same private key as the current extension. If the signature is correct, then the installation of the update will succeed. However, this approach will not protect the installed extensions from being modified. Thus, having access to the user's profile folder will allow replacing the update URL and the corresponding public key.

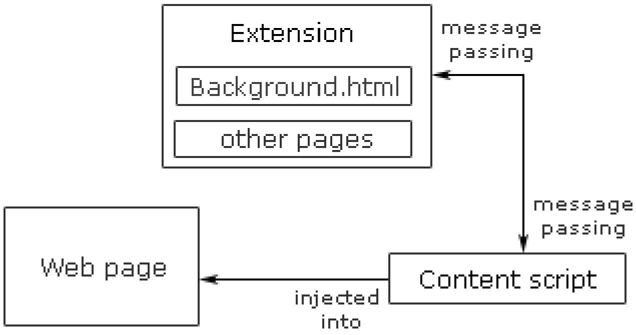


Figure 2.2: Content scripts and the background page communicate via message passing.

2.3.1 Chrome's memory space

In Google Chrome the memory of browser kernel, web pages, extensions and plugins is isolated. The isolation is achieved through the use of different processes, which are natively isolated from each other. The communication between different processes is done by message passing. Every extension runs in its own process and is therefore protected from other extensions. Thus, almost all of the extension's code is separated from other processes, with the exception of user scripts, which run in whichever renderers they apply to. The extension process can be considered as a special type of renderer that controls rendering of the various pieces of user interface, that an extension wishes to add to the browser. The extension process is not able to interact with the web page content directly, it has to inject user scripts into a web page and then communicate with these scripts [GCE]. This means that the extension is isolated from web content and cannot directly access the functions and variables of a web page, therefore it is protecting itself from threats. An extension is not able to directly access the variables or functions of another process, including the main browser process. Thus, if an extension gets hijacked, it cannot modify the way the browser or other extensions function.

2.3.2 Google Chrome's code space

Google Chrome's code space is not protected. It is possible to modify extension files and the preference files. These files are located in Google Chrome's user profile folder and thus accessing and modifying them does not require administrative rights. We tested the possibility to overwrite the preferences file and the extension files and found out that it was possible. Additional content can be injected into the extension files, but to make all the changes work the preferences file has to be modified first. This is because the preferences file contains the copies of extension manifests and if the extension's manifest file was modified the preferences have to be changed similarly. In order to modify the preferences file, the browser must not be running or the default state will be restored after the browser is closed. Overwriting these files and thus changing the functionality of the extensions required only user privileges. Therefore, a third party application, which is running with limited privileges is able to change the functionalities of an already installed extensions.

2.4 Attack scenarios

2.4.1 Creating a keylogger

As Google Chrome's extensions can use content scripts, it is trivial to write a keylogger. In order to create a keylogger it should be possible to access the user profile folder, which does not require administrative privileges. A content script needs to listen for keystokes and save them. In order to do that, special rights must be given to the content script in the manifest file, to allow the content script to be injected into every web page. Because the content script cannot use `XMLHttpRequest` to send data to a remote server, it has to delegate the task to the background page. Message passing will allow to send the saved keystrokes to the background page, which is able to send the data to a remote server. We created this kind of a keylogger and found out that the code can be copied into an already installed extension. This can be dangerous, as the user will not be able to detect the change. The code can be injected by a third party application or by someone who has brief access to the computer.

2.4.2 Man in the browser attack

This attack is feasible if a compromised JavaScript can be installed to the browser. This attack is based on the possibility to actively modify https pages via DOM interface. As a result, a malicious extension could manipulate a bank's web site or the web site of an e-mail service provider. First, the extension would have to recognize the web page it will attack. This can be done as the extension can query the URL of a web page. The attack would begin when the user navigates to a https page listed in the extension. The extension would already know the structure of these web pages. Thus, it could recognize a frame or a web page, where user is required to fill forms. In that page or frame the extension could inject an additional function to the send button and collect and replace the data entered into the forms when the send button is pressed. Thus, the extension could modify the DOM after the modifications send the data to the server. The modification in the functionalities of the send button is invisible to the user as the visible user interface does not change. To test our claim, we created an extension, which was able to inject code into a https page and modify its DOM. The attacking extension might also replace the DOM of the result page, but that might create a flicker on the screen. A flicker might be noticeable because Google Chrome does not support `DOMContentLoaded` event the way Firefox does. Namely, Google Chrome may load images before the `DOMContentLoaded` event is fired.

Contrary to Firefox, it is possible to disable extensions in Google Chrome during a browsing session. This can be done due to the isolated memory of Chrome. For example, a button could be added to the user interface, which would disable all extensions on the current web site. Chrome already provides the possibility to choose on which web sites plugins are allowed and this could also be extended to extensions. Currently it is possible to disable one extension at a time from the extension manager, but not all extensions. Also, it should be possible to create a list of web pages on which extensions are automatically disabled, thereby guaranteeing the integrity of the web page.

2.4.3 Using Google Chrome as a botnet

A Google Chrome's extension could be used as a botnet client. The implementation would be a bit more difficult than in Firefox due to the content script's restrictions, but in principal the same. An extension could read the commands from fixed web sites and then send spam by exploiting the possibilities of `XMLHttpRequest`.

2.4.4 Risk assessment

- Critical - It is possible to run code with user privileges and without the user's knowledge.
- High - Access to confidential data and ability to inject data without the user's knowledge.
- Moderate - Access to sensitive information that does not expose the user or organization to immediate risk.

- Low - Leaks of non-sensitive information.
- DoS - Temporary denial of service attacks that may result in the crash of the application.

	Critical	High	Moderate	Low	DoS
Changing the functionalities	x				
JavaScript keylogger			x		
Man in the browser attack		x			
Botnet client				x	
Faulty extensions					x

2.5 Ways for compromising Google Chrome

2.5.1 Installing a compromised extension

The easiest way to compromise the browser is to trick the user into installing a compromised extension. This risk can be limited by providing signed and verified extensions, but it is not possible to protect the user from third party extensions that are not hosted in the Google's extension gallery. Extensions that have a binary NPAPI component are thoroughly tested by Google before they are allowed into the extensions gallery. This is necessary as binary component are not sandboxed and have access to the file system. For that reason, Google Chrome notifies the user about the potential security risk when a binary extension is being installed. However, it is not possible to protect the user after a malicious binary extension has been installed. Installing an unknown binary extension is considered to be the user's risk.

2.5.2 Modifying the installed extensions

It is possible to modify the source code of an installed JavaScript extension. This makes it possible to hide malicious extensions from the user as they can live inside another extension. The injected code will not break the original functionalities of the extension, which guarantees that the user will not notice the change. Thus a third party application can add code to already existing extensions in order to hide it's behavior.

2.6 Solutions

Currently, there is no option on the user interface to disable all extensions on certain web sites. This would be an important option, as the user could be guaranteed that an extension has not changed the integrity of a web page. There is a possibility to turn off the extensions by running Google Chrome with the flag `--disable-extensions`, but this approach is not acceptable for an average user.

Google Chrome does not notify the user, when an extension's source code has been modified and this allows the malicious code to be hidden. Furthermore, it should not be possible to unnoticeably change the privileges of an extension by modifying the manifest file. These vulnerabilities could be disabled by digitally signing the extension's files. The signature would have to be recalculated every time an update is received, as it is the only time when the extension's files are modified. This would be computationally feasible. Another solution would be to use a hash function to protect the whole code tree. Hash function SHA-256 would be sufficient for the task. The code tree could be hierarchically hashed and the top hash kept in a protected location, which would be accessible only with administrative privileges.

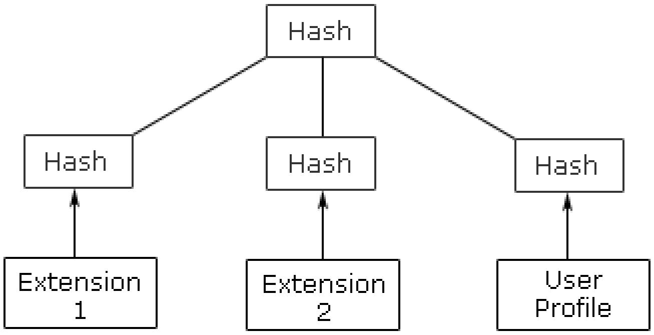


Figure 2.3: Hierarchically hashed code space.

Chapter 3

Internet Explorer 8

Internet Explorer is a closed source browser that is developed by Microsoft. The browser is built only for the Microsoft Windows operating systems. As of April 2010, Internet Explorer holds 59.95% of web browsers market share according to Net Applications, making it the most popular web browser [IES]. The statistics by Net Applications includes mobile browsing and is gathered from a network of 40000 web sites over the world.

3.1 Overview of the extensions

Browser extensions, which were introduced in Microsoft Internet Explorer 5, allow to add functionality to the browser. Furthermore, extensions can modify the user interface in a way that is not directly related to the viewable content of web pages. The extensibility of the browser makes it possible for the users to shape the browser according to their needs, for further details see [IEE].

Internet Explorer's extensions can be divided into the categories of browser extensions and content extensions. Browser extensions can be used to add additional functionality to the browser's content. It includes features such as shortcut menu extensions, custom toolbars, Explorer Bars, and Browser Helper Objects (BHOs). Content extensions are used to extend the types of content that can be parsed and displayed. ActiveX Controls and active documents are a part of content extensions [IEA]. For example, ActiveX Controls are used for running Flash and Microsoft Silverlight in Internet Explorer.

With BHOs it is possible to write components, that Internet Explorer will load each time it starts up. Specifically, these components are in-process Component Object Model (COM) components. These objects run in the same memory context as the browser and can perform any action on the available windows and modules. Such actions include detecting the browser's typical events like GoBack, GoForward and DocumentComplete; accessing the browser's menus and toolbars and modifying them [IEE]. Additionally, a Browser Helper Object can create windows on the currently viewed page and install hooks to monitor messages and actions [IEE]. This means that it can modify the functionality of the browser by adding binary components. A Browser Helper Object is a dll module that runs within Internet Explorer and offers additional services.

ActiveX controls can be embedded into a web page and used as an application. ActiveX control has full access to the operating system and thus it can be used to spread malware. A web page can initialize the installation process of the ActiveX control, which is a potential security threat. This is called a Drive-By-Download. A page may ask the user to install a malware extension by claiming it to be an useful plugin and trick the user to install it. If the browser is running in protected mode then the installation will prompt the user with a request to use administrator privileges.

3.2 High-level architecture

Internet Explorer architecture is based on a Component Object Model, a binary-interface standard for software componentry introduced by Microsoft in 1993. According to Microsoft Developer Network library the architecture of Internet Explorer is divided into six modules [IEA].

- `IExplore.exe` is a small application that relies on the other main components of Internet Explorer to do the work of rendering, navigation, protocol implementation, and so on.

- `Browsui.dll` is responsible for the user interface. This includes the address bar, status bar, menus, and so on.
- `Shdocvw.dll` exposes ActiveX Control interfaces, provides navigation functionality and history functionality. This component can provide all the functionalities of the Internet Explorer except the user interface.
- `Mshtml.dll` contains the Trident rendering engine. It is responsible for displaying web pages and handling the DOM.
- `Urlmon.dll` is responsible for handling Multipurpose Internet Mail Extensions (MIME) and download of web content.
- `WinInet.dll` handles all network communication over HTTP, HTTPS and FTP.

In previous Internet Explorer versions tabs, BHOs, ActiveX controls and toolbar extensions were running in the same process as the browser window. This caused crashes and security problems as the memory space was shared. Loosely-coupled Internet Explorer (LCIE) architecture was introduced in Internet Explorer 8, see Figure 3.1. This architecture isolates the browser's frame and its tabs into different processes and allows to use The Windows Vista Integrity Mechanism on a per-process basis. This means that it is possible to run tabs in a protected mode, while allowing exceptions. Therefore, tabs can be run with different mandatory integrity levels (MIC) inside the same browser window. Additionally, the browser kernel is isolated from the tab processes, which execute scripts and extensions, therefore providing additional protection for the browser kernel. The protected mode is designed to prohibit write privileges but the confidentiality of the file system is not protected, as read access is not restricted. This also holds for Internet Explorer 7. By default, the possible number of Internet Explorer's processes is limited by the amount of available physical memory. Once the limit is reached then the new tabs are forced to share a process with an already existing tab. The limit may be changed manually by editing the value of `TabProcGrowth` in the Windows registry. When compared to Google Chrome, which fixes a process limit depending on the available physical memory, Internet Explorer determines when its tabs will share a process by using a context-based algorithm. This algorithm uses the amount of installed physical memory to calculate a pattern, which is used for creating new processes. Therefore, tabs in Internet Explorer may share a process before the process limit is reached.

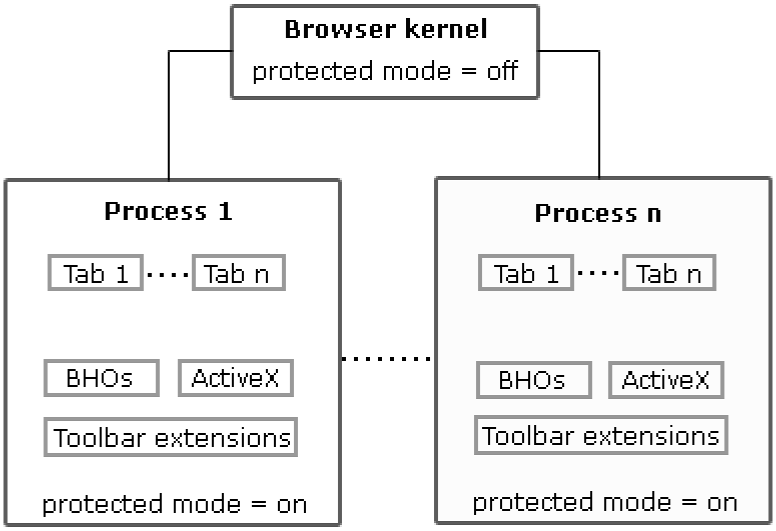


Figure 3.1: Loosely-coupled Internet Explorer architecture.

3.3 The security architecture of extensions

The core of the security architecture is based on the functionalities of Windows Vista or newer versions of Windows. The architecture is based on privilege separation and the ability to create processes with lower privileges. It allows to run the browser with limited rights, which protects the computer if the browser gets compromised. This approach is necessary, as the extensions run with the same privileges as the browser.

In Windows Vista, IE 7 and IE8 run in protected mode, which helps protect users from attack by running the Internet Explorer process with greatly restricted privileges. Protected mode significantly reduces the ability of an attacker to write, alter or destroy data on the user's machine or to install malicious code [SB]. Protected mode uses the Windows Vista integrity mechanism to run the Internet Explorer process at low integrity [SB].

Thus, if the browser or a tab runs in a protected mode a malicious extension does not have write access to the data on the file system that has a higher integrity level. By default, all files and registry entries have medium integrity level and thus Internet Explorer in protected mode does not have write access to them. However, the protected mode does not deny read access and because of that a malicious extension would still be able to read the data with a higher integrity level. This could lead to a security breach, as a malicious extension could gather sensitive information and send it to a remote server.

The majority of the extensions are digitally signed, which guarantees that the updates are not tampered with and come from an authentic source.

3.3.1 Internet Explorer's memory space

Internet Explorer does not create new processes for extensions and plugins. Extensions are loaded as the browser starts, but they run in the process from which they are called. The level of memory isolation depends on the amount of created tabs as the number of new processes is limited. As the number of tabs grows, new web pages are forced to share the process with other web pages. Thus, the extension called from shared memory space will have direct access to the other web sites in that process. This is somewhat different from Google Chrome's behavior, as Chrome runs each extension in a different process. In Internet Explorer an extension that runs in a shared process is not able to directly access web pages that are located in other processes. However, in Google Chrome an extension with proper privileges may inject scripts to every open web site.

A low privilege process cannot compromise a higher privilege process. This is achieved by User Interface Privilege Isolation (UIPI), which prevents lower privilege processes from accessing higher privilege processes by blocking possibly dangerous behavior [UIP]. For example, it is not possible to `SendMessage` or `PostMessage` to higher privilege application windows. This is disabled because window messages cover a wide range of information and requests, including messages for mouse and keyboard input, dialog box input and window creation and management [UIP]. Also, using hooks to attach to a higher privilege process or to monitor a higher privilege process is not allowed. These restrictions are used to protect the privileged applications as a lower privileged process may not listen or modify their behavior. In addition, a low privilege process cannot perform DLL injection to a higher privilege process, as it would allow the lower privilege process to run code with higher privileges.

3.3.2 Internet Explorer's code space

Internet Explorer that is running in Windows Vista or a newer Windows platform is by default running in protected mode. Thus, IE has write access only to low integrity objects. These include the cookies folder, history folder, temporary files folder and favorites folder. An extension that runs in protected mode can write only to certain subfolders of the previously mentioned low integrity objects. For example, an extension can write to `Temp\Low` and `Cookies\Low`. With those restrictions extensions are not able to write to system locations such as the `Program Files folder` ,`HKEY_CLASSES_ROOT` or `HKEY_LOCAL_MACHINE` subtrees. Also, extensions that try to gain write access to the Internet Explorer binary files will receive access denied errors [SB]. Thus, extensions that run in protected mode are not able to compromise the IE code space. Thereby, comparing to Firefox and Google Chrome, Internet Explorer running in protected mode is able to prevent extensions from modifying the code space.

3.4 Attack scenarios

3.4.1 Creating a keylogger

It is possible to write a BHO that logs keystrokes and monitors user's behavior. By doing a quick search on Internet, we found several downloadable BHO keyloggers and tutorials for creating them. As Internet Explorer does not have a simple JavaScript based extension system it is somewhat harder to write a simple keylogger. For example, a BHO could save user's keystrokes in a temporary file that is accessible from the browser. The collected data could be sent to a remote server, when enough information has been saved. While it was possible for the user to read the source code of the Firefox's and Google Chrome's JavaScript extensions, it is not possible in Internet Explorer because the user has access to a compiled binary file. Thus, the user cannot verify the functionalities of the BHO.

3.4.2 Website defacement

Website defacement cannot be done in an invisible way, like it is possible in Firefox. In Firefox, JavaScript can interact with the DOM before the web page has been displayed. In Internet Explorer, the state when the DOM has been loaded but is not displayed cannot be detected, see Figure 3.2. That is not possible because the scripts and media in Internet Explorer are loaded as they are downloaded and do not wait until the DOM is finished. Thus a web page would have to be loaded in order to modify the DOM and the user could notice it. Besides that, to manipulate the DOM a binary plugin (BHO) would have to written, which is more complex than writing a JavaScript extension. Therefore the original DOM cannot be replaced by trivial methods.

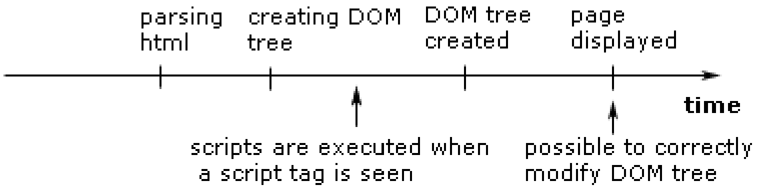

parsing html
creating DOM tree
DOM tree created
page displayed
time
scripts are executed when a script tag is seen
possible to correctly modify DOM tree

Figure 3.2: This scheme shows how a web page is being loaded.

3.4.3 Man in the browser attack

Website defacement is feasible only if invisible elements are modified. Man in the browser attack is an example of such a website defacement. This attack is feasible if a compromised BHO can be installed on the computer. Thus, a previous infection is probably required for this attack to succeed. Man in the Browser attack is similar to a man-in-the-middle attack but in this case the malware is running inside the browser. A BHO is able to modify the communication between the user and the server by replacing some parts in DOM before a data packet is sent to the server.

For example, the compromised BHO could manipulate the information in Internet banking transactions. The BHO contains a list of URL-s that are being targeted and it compares these URL-s with the ones that the user visits. The attack is initialized when the user navigates to a web page that is in the targets list. The extension already knows the structure of the bank's web page and is able to recognize the transactions page. In the transactions page the extension collects the data from the forms, as the send button is clicked and replaces it in DOM. After the DOM is changed the extension calls the function, which sends the modified information to the bank. For these steps the functionality of the send button is changed by adding an additional function to the DOM, when the transactions page has been loaded. This is invisible to the user, as the visible user interface does not change. When the bank sends a reply to confirm the transaction the extension replaces the false data with the correct data in the reply's DOM. The latter may be noticed by the user, as the modification of the DOM might leave a flicker on the screen. However, it is not likely that the user will understand what the flicker meant.

This attack is possible because the BHO is able to access and modify the DOM of the web pages. It is also hard to avoid because the user passes all the security checks before the transmission data is sent to the bank.

3.4.4 Risk assessment

- Critical - It is possible to run code with user privileges and without the user's knowledge.
- High - Access to confidential data and ability to inject data without the user's knowledge.
- Moderate - Access to sensitive information that does not expose the user or organization to immediate risk.

- Low - Leaks of non-sensitive information.
- DoS - Temporary denial of service attacks that may result in the crash of the application.

	Drive by downloads	Installing a compromised binary	Man in the browser attack	Faulty extensions
Critical	x	x		
High			x	
Moderate				
Low				
DoS				x

3.5 Ways for compromising Internet Expolorer

3.5.1 Installing a compromised extension

The risk of installing an unknown BHO or an ActiveX control equals the risk of running an executable file. Thus, before installing an ActiveX control the user should be warned of the threats. A compromised ActiveX could be accidentally installed when a web site is initiating an installation. Also, an ActiveX control may have security flaws that could allow the computer to be compromised. For example, in July 2009 `MPEG2TuneRequest` ActiveX control in `msvidctl.dll` in DirectShow allowed remote attackers to execute arbitrary code via a crafted web page [CVE]. This was possible because the `msvidctl.dll` contained a design flaw, which allowed stack-based buffer overflow in the `CComVariant::ReadFromStream` function in the Active Template Library (ATL) to be used to execute arbitrary code [CVE].

Many popular BHO extensions are not hosted on the official extensions web site. These extensions are created by third party developers who might have added malware to the BHO. For example, the user would probably not notice a keylogger inside a binary that blocks advertisement.

3.6 Solutions

Currently, the security architecture does not prevent a compromised extension from reading the data on the file system and sending this data to a remote server. The extensions should not have read access to random files on the file system. A solution for this would be to limit the read access and force the extension to ask read access only to a specific location, when the extension is installed. The extensions could have a file containing a list of locations with read access and the user would see the list while the extension is being installed.

A protection against threats arising from extensions can be achieved by running Internet Explorer with all extensions disabled. Internet Explorer will start without extensions if the user runs `iexplore -extoff`. With this approach extensions are not loaded as the browser starts. If the browser was started in normal mode, then the extensions are loaded, but they can be disabled from the add-ons manager, which is located in the tools menu.

Conclusive comparison

One of the biggest differences between these browsers is the fact that Google Chrome and Firefox are built on open source code, while Internet Explorer's source code is not available. Therefore, the community of developers and enthusiast are able to find security holes in Google Chrome and Firefox. As Internet Explorer is based on closed source, it may hide security holes, which the community is not able to find. When a severe vulnerability is found in a browser, then usually a fast fix is provided in the next update. All three browsers behave similarly when downloading updates for extensions. The update is digitally signed and thus cannot be tampered with. However, Google Chrome and Firefox get updates for their extensions from sources that are defined by the extensions. Updateable extensions contain a manifest file with an update URL and a public key, but the manifest file is not protected from being modified. Thus, if the computer is already compromised, then it is possible to change the source of the updates and replace the existing public key, which is used for authenticating the update.

Memory space

When comparing the three browsers, Firefox is the only one without isolated memory. Google Chrome and Internet Explorer both use additional processes for rendering web content. They also isolate the browser's kernel from the extensions and web content, thereby protecting it from being compromised. Contrary to Google Chrome and Internet Explorer, Firefox could be compromised by a malicious extension or by a specially crafted web page. As Firefox currently follows a single process model, extensions are able to access and modify the stack of a random thread inside the process. Firefox does not provide any isolation between extensions as an extension is able to modify the functionality of other extensions. In order to enhance security and performance, a multi-process version of Firefox is currently being developed.

Code space

The architecture of Google Chrome's extensions prevents a non-binary extension from accessing file system. Also, the sandboxed rendering processes are denied read and write access to the file system. Internet Explorer sets similar restrictions with the exception that read access is allowed. This privilege could lead to a leak of sensitive data if a browser containing malicious extension is used for browsing. Unlike Google Chrome and Internet Explorer, Firefox's architecture gives JavaScript extensions read and write access to the file system. Thus, a JavaScript extension is able to collect sensitive data from the hard drive and publish it. Neither Firefox nor Google Chrome guarantee that an already installed extension's source files are not tampered with. Both browsers are not able to detect an injection of a keylogger script into installed extensions. Moreover, such an extension's original functionalities remain intact, therefore not giving the user a reason for getting suspicious.

Brauseri laienduste turvaanalüüs

Kristjan Krips

Bakalaureusetöö (6 EAP)

Kokkuvõte

Paljud tänapäevased brauserid võimaldavad funktsionaalsuse lisamist või muutmist laienduste kaudu. Rohkete võimaluste tõttu on laiendused muutunud kasutajate hulgas populaarseks ja see on toonud kaasa uued ründevektorid, mis ohustavad kasutajate turvalisust. Töös analüüsime populaarsemate veebibrauserite laienduste turvaarhitektuuri. Vaatleme Firefox 3.6, Google Chrome 5.0.360 ja Internet Explorer 8 laienduste ehitust ja nende turvalisust. Töö annab ülevaate vastavate brauserite laienduste arhitektuurilisest turvalisusest ja kirjeldab võimalikke ründevektoreid. Selgitame, kuidas on vastavate veebibrauserite koodiruum ja mälu kaitstud ja teeme kindlaks missuguseid õiguseid brauserite laiendused omavad. Uurime, kuidas on praegust laienduste arhitektuuri kasutades võimalik brausereid kompromiteerida ja kirjeldame sellega kaasnevaid riske. Selleks demonstreerime laiendusi, mis kompromiteerivad brauseri, näitamaks olemasoleva arhitektuuri puudujääke. Näitame erinevaid ründevektoreid ja kirjeldame nendele vastavaid ründestsenaariumeid. Töö tulemusena selguvad brauserite laienduste turvaarhitektuuri nõrkused. Nende leevendamiseks pakume välja lahendusi, mis parandavad turvaarhitektuuri. Töö tulemusena on võimalik brauserite kasutajaid informeerida olemasolevatest ohtudest ja teadvustada turvalisuse olulisusest.

Glossary

BHOs Browser Helper Objects are DLL modules designed for extending Internet Explorer's functionalities. Pages 36, 37, 38, 41, 42, 43, 44

COM Component Object Model is an architecture, allowing applications to be built from binary software components. Page 37

CSS Cascading Style Sheets are used by Firefox's and Google Chrome's extensions for creating browser themes. Page 26

DOM Document Object Model is a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents. Pages 8, 15, 16, 18, 24, 29, 32, 33, 38, 42, 43

DoS Denial of Service is a type of an attack, which attempts to make a computer resource unavailable to its intended users. Pages 21, 22, 34, 44

Gecko An open source layout engine used in Mozilla Firefox. Pages 6, 7, 14

JavaXPCOM A technology, which enables simple interoperation between XPCOM and Java. Page 8

LCIE Loosely-coupled Internet Explorer architecture isolates Internet Explorer's frame and its tabs into different processes. Page 38

MIC Mandatory Integrity Control is a security feature introduced in Windows Vista, which adds integrity levels to processes. Page 38

NPAPI Netscape Plugin Application Programming Interface is a cross-platform plugin architecture used in Firefox and Google Chrome. Pages 27, 34

NSPR Netscape Portable Runtime provides platform independence for threads, thread synchronization, file and network I/O and basic memory management. Page 11

PyXPCOM A technology, which enables simple interoperation between XPCOM and Python. Page 8

UIPI User Interface Privilege Isolation is a security feature that was introduced in Windows Vista. It is used to prevent lower privilege processes from accessing higher privilege processes. Page 40

XPCOM A cross platform component object model that is used in Firefox. Pages 7, 8, 10, 12

XPConnect A technology, which enables simple interoperation between XPCOM and JavaScript. Pages 7, 8

XSS Cross-Site Scripting is a type of an attack, which allows an attacker to inject unwanted code into a web site. Pages 22, 23

XUL XML User Interface Language, which is used in Firefox. Page 8

Bibliography

[BFSB09] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting Browsers from Extension Vulnerabilities. Technical Report UCB/EECS-2009-185, EECS Department, University of California, Berkeley, December 2009. Available from http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-185.html.

[Bin] Language Bindings. http://mdn.beonex.com/en/XPCOM/Language_Bindings. Last checked on 15 May 2010.

[Blo] Extensions blocklist. http://kb.mozillazine.org/Extensions.blocklist.enabled. Last checked on 1 May 2010.

[BMS] Browser market share. http://marketshare.hitslink.com/browser-market-share.aspx?qprid=0&qpcal=1&qpstick=1. Last checked on 29 April 2010.

[CVE] Cve-2008-0015. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0015. Last checked on 23 April 2010.

[eva] Components.utils.evalinsandbox. https://developer.mozilla.org/en/Components.utils.evalInSandbox. Last checked on 25 March 2010.

[Fel09] Adrienne Porter Felt. A Survey of Firefox Extension API Use. Technical Report UCB/EECS-2009-139, EECS Department, University of California, Berkeley, October 2009. Available from http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-139.html.

[Fir] Mozilla firefox firebug extension two cross-context scripting vulnerabilities. http://secunia.com/advisories/24743/. Last checked on 2 November 2009.

[GCA] Multi-process architecture. http://dev.chromium.org/developers/design-documents/multi-process-architecture. Last checked on 2 April 2010.

[GCB] Background pages. http://code.google.com/chrome/extensions/background_pages.html. Last checked on 3 April 2010.

[GCC] Content scripts. http://code.google.com/chrome/extensions/content_scripts.html. Last checked on 3 April 2010.

[GCE] Extension process model. http://www.chromium.org/developers/design-documents/extensions/process-model. Last checked on 3 April 2010.

[GCM] Message passing. http://code.google.com/chrome/extensions/messaging.html. Last checked on 3 April 2010.

[GCN] Npapi plugins. http://code.google.com/chrome/extensions/npapi.html. Last checked on 2 April 2010.

[GCO] Google chrome extensions overview. http://code.google.com/chrome/extensions/overview.html. Last checked on 2 April 2010.

[GCP] Process models. http://www.chromium.org/developers/design-documents/process-models. Last checked on 3 April 2010.

[Gec] Gecko-specific dom events. https://developer.mozilla.org/en/Gecko-Specific_DOM_Events. Last checked on 2 November 2009.

[HAS] Extension versioning, update and compatibility. https://developer.mozilla.org/en/Extension_Versioning,_Update_and_Compatibility. Last checked on 1 May 2010.

[IEA] Internet explorer architecture. http://msdn.microsoft.com/en-us/library/aa741312.aspx. Last checked on 10 April 2010.

[IEE] Browser extensions. http://msdn.microsoft.com/en-us/library/aa753587(VS.85).aspx. Last checked on 10 April 2010.

[IES] Browser market share. http://marketshare.hitslink.com/browser-market-share.aspx?qprid=0&qpcal=1&qpstick=1. Last checked on 1 May 2010.

[LF] Roberto Suggi Liverani and Nick Freeman. Abusing Firefox Extensions. Available from http://www.defcon.org/images/defcon-17/dc-17-presentations/defcon-17-roberto_liverani-nick_freeman-abusing_firefox.pdf. Last checked on 2 November 2010.

[Nig] Johnathan Nightingale. Component directory lockdown. Available from https://developer.mozilla.org/devnews/index.php/2009/11/16/component-directory-lockdown-new-in-firefox-3-6/. Last checked on 1 May 2010.

[nsI] nsithread. https://developer.mozilla.org/en/nsIThread. Last checked on 15 May 2010.

[NSPa] Introduction to nspr. https://developer.mozilla.org/en/NSPR_API_Reference/Introduction_to_NSPR. Last checked on 1 April 2010.

[NSPb] Nspr 4.8.3 release. http://www.mozilla.org/projects/nspr/release-notes/nspr31.html. Last checked on 1 April 2010.

[Saf] Safely accessing content dom from chrome. https://developer.mozilla.org/en/Safely_accessing_content_DOM_from_chrome. Last checked on 1 May 2010.

[SB] Marc Silbey and Peter Brundrett. Understanding and working in protected mode internet explorer. Available from http://msdn.microsoft.com/en-us/library/bb250462(VS.85).aspx. Last checked on 15 April 2010.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, second edition, 2001.

[Thr] Using the thread manager. https://developer.mozilla.org/en/The_Thread_Manager#Using_the_Thread_Manager. Last checked on 24 March 2010.

[TO] Doug Turner and Ian Oeschger. Using XPCOM components. Available from https://developer.mozilla.org/en/Creating_XPCOM_Components/Using_XPCOM_Components. Last checked on 29 April 2010.

[Tur] Doug Turner. nsisupports proxies. Available from https://developer.mozilla.org/en/nsISupports_proxies. Last checked on 1 May 2010.

[UIP] Windows vista application development requirements for user account control (uac). http://msdn.microsoft.com/en-us/library/aa905330.aspx. Last checked on 15 April 2010.

[XPCa] Xpcnativewrapper. https://developer.mozilla.org/en/XPCNativeWrapper. Last checked on 25 March 2010.

[XPCb] Xpcom. https://developer.mozilla.org/en/xpcom. Last checked on 15 May 2010.

[XSS] Cross-site scripting (xss). `http://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`. Last checked on 2 November 2009.