Internet Security Seminar

Analyzing Information Flow in JavaScript-based Browser Extensions

Julien LIRONCOURT
Technical University of Berlin, Germany
e-mail: j.lironcourt@tu-berlin.de
Ecole Nationale de l'Electronique et de ses Applications, Cergy, France
e-mail: julien.lironcourt@ensea.fr

Introduction

The Web browser is one of the most used, and most exploited application. Today, browsers run not only on desktop computers and laptops, but also on Tablet PCs, PDAs and mobile phones and people can browse at any time and in any place. When seeing this intensive use, it becomes obvious that the question of security will be at the core of the debate. Indeed, as browsers are widely used and deal with particurlaly confidential information (such as bank account, or e-shopping...), they will be a particular target for attackers and so are exposed to many risks. Since the first browser was written by Tim Berners-Lee at the CERN (European Nuclear Research Center) in 1990, things have evolved quickly and browsers become more and more complex with more and more functionalities.
But these new functionalities -enabled by plug-ins and extensions- also mean more risks as they create new security weaknesses for the user. According to a recent report by Symantec [12], during the first half of 2007, Internet Explorer had 93 security vulnerabilities, Mozilla browsers had 74, Safari had 29 vulnerabilities and Opera had 9. And an additional 301 other vulnerabilities in browser plug-ins and extensions.
In this paper we study a new type of security weakness caused by browser extensions which can be very pernicious. After a general overview about the way browsers work, as a prerequisite for our study, and seeing their lack in security, we study the Sabre system (Security Architecture for Browser Extensions) which tracks the information flow inside the browser to analyse malicious extensions, and which is an answer proposed to the big security lacks that the use of extensions in browsers represent. Having understood the way Sabre was working, can we then see how relevant it is, and if its answer to security problems is the most appropriate.

1 Generalities about Web Browsers

1.1 Browser Structure

Many different web browsers exist, Internet Explorer, Firefox , Safari, Opera, Google Chrome, and others which are less known and used. Every Web browser has a common general structure, even if this structure can be slightly different in some details from one to another.
We can see on figure 1 this architecture with eight major subsystems.

The User Interface subsystem is the layer between the user and the browser engine.
The Browser Engine subsystem is a component that provides a high-level interface to the rendering Engine. It loads an URL and supports the basic browsing actions like forward, back and reload.
The Rendering Engine subsystem produces all the visual presentation of an URL (dealing with HTML or XML documents, optionnally CSS, and managing to calculate the exact layout of the web page). This subsystem includes the HTML parser.
The Networking subsystem implements file transfer

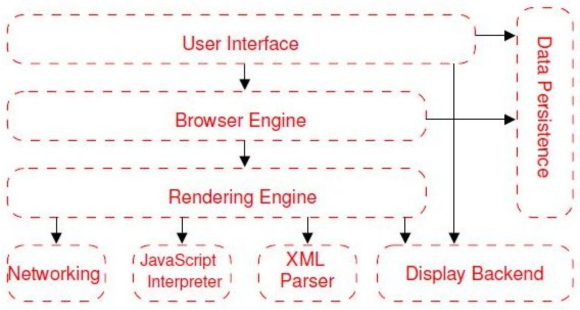

Figure 1: Browser structure scheme [2]

protocols such as HTTP and FTP.
The JavaScript Interpreter evaluates JavaScript code, which may be embedded in web pages.
The XML Parser subsystem parses XML documents into a Document Object Model (DOM) tree.
The Display Backend subsystem provides drawing and windowing primitives, a set of user interface widgets, and a set of fonts. It may be tied closely with the operating system.
The Data Persistence subsystem stores various data associated with the browsing session on disk.
We can show now the Mozilla structure on the picture below and see that it is based on the general structure previously presented. We can see on the

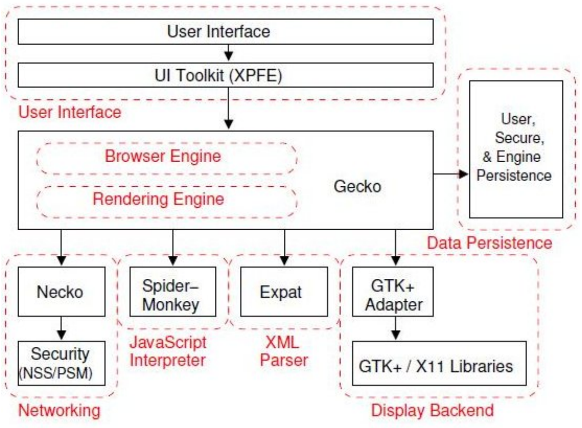

Figure 2: Mozilla Architecture [2]

picture above that all the main elements previously desrcribed are in this structure, split or integrated in Mozilla's tools to implement the key functions.
Firefox's conceptual structure is similar to Mozilla's, but Firefox contains one notable feature which is not shown in the architecture : a powerful extension facility.
We will then present some of key functions in the browser architecture which are useful to understand the security lacks that we will deal with further in this work.

DOM

The Document Object Model (or DOM) is a convention from the W3C. It is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents [19]. The document can be further processed and the results of that processing can be incorporated back into the presented page. Aspects of the DOM (such as its "Elements") may be addressed and manipulated within the syntax of the programming language in use. The public interface of a DOM is specified in its Application Programming Interface (API). The DOM is programmatically accessible, commonly through JavaScript. It can be seen as a tool to give access to HTML and XML documents. To enable to modify the content and visual presentation of the document, it provides a structural representation of the document [20]. Thanks to it, the developer can both : give a structured representation of the document, codify the way a script can access this structure. Indeed it is mainly a means to link a web page to a coding language or a script language.
As an interface, it allows to standardize the means of access to a tagged document, in particular a web page. All the methods and all the events available for the programmer to manipulate and create dynamic pages are organized as a hierarchy of objects [21]. A marking language such as HTML, or any other language based on XML, can be schematized as a hierarchy in tree structure. The different components of this tree structure are the nodes. That is why the main object of the DOM model is the node object.

XUL

I also present here the concept of the XUL as we will evoke it later. XUL, the XML User Interface Language, is an XML user interface markup language developed by the Mozilla project [15]. XUL operates in Mozilla cross-platform applications such as Firefox to build the User Interface. The Mozilla Gecko layout engine provides an implementation of XUL used in the Firefox browser.

Same origin policy

The same origin policy prevents a document or script loaded from one origin from getting or setting properties of a document from another origin [17]. This policy dates all the way back to Netscape Navigator 2.0. Mozilla considers two pages to have the same origin if the protocol, port (if one is specified), and host are the same for both pages [18]. This mechanism bears a particular significance for modern web applications that extensively depend on HTTP cookies to maintain authenticated user sessions, as servers act based on the HTTP cookie information to reveal sensitive information or take state-changing actions. A strict separation between content provided by unrelated sites must be maintained on the client side to prevent the loss of data confidentiality or integrity.

Sandboxing
In computer security, sandboxing is a security mechanism for separating running programs. A sandbox limits, or reduces, the level of access its applications have -it is a container, hence its name. It is often used to execute untested code, or untrusted programs from unverified third-parties, suppliers and untrusted users.
As an example, if a process P runs a child process Q in a sandbox, then Q's privileges would typically be restricted to a subset of P's. For example, if P is running on the "real" system (or the "global" sandbox), then P may be able to look at all processes on the system. Q, however, will only be able to look at processes that are in the same sandbox as Q. Barring any vulnerabilities in the sandbox mechanism itself, the scope of potential damage caused by a misbehaving Q is reduced [22].

1.2 Extensions

In this paper, we define the term extensions with the meaining of Mozilla. Extensions add new functionality to Mozilla applications such as Firefox, SeaMonkey and Thunderbird. They can add anything from a toolbar button to a completely new feature. They allow the application to be customized to fit the personal needs of each user if they need additional features, while keeping the applications small to download [24]. Extensions are different from plugins, which help the browser display specific content like playing multimedia files. Extensions are also different from search plugins, which plug additional search engines in the search bar. The range of functionalities that extensions can cover can be very wide : from layout to security features, including multimedia content. Most of extensions are coded with scripting language (JavaScript), but it is also possible to write some code in low level languages (C/C++) or high-level languages (Java,Python) . They can contain native libraries (DLL on Windows) and call their functions, but they can also contain native programs (such as EXE files on Windows) and launch them. As Firefox is Opensource, extensions are available at https://addons.mozilla.org [13]. Before being released, extensions will go through the Sandbox review system to check that it is safe. As its principle shows it on the picture above, the sandbox review system is a Mozilla's process to test addd-ons before they are released.
Extensions can be installed from the browser using

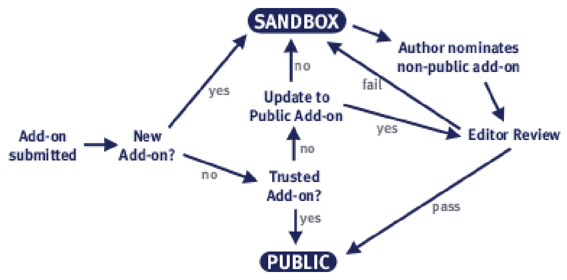

Figure 3: Sandbox validation in Mozilla [17]

the add-on manager, or from web pages. But it is also possible to install extensions with an external program without any user agreement.
Even if we will shortly mention other types of extensions, such as plug-ins or Browser Helper Objects, to draw some comparison and parallel, we will focus in this paper on JavaScript-based browser extensions (JSEs). A Browser Helper Object (BHO) is a DLL module designed as a plug-in for Microsoft's Internet Explorer web browser to provide added functionalities. JSEs are written in JavaScript and are very popular and widely available (as add-ons) for Firefox, making it so famous. Such JavaScript extensions can be seen as a script injection. If a web page is loaded, all scripts extensions will be internally injected to the page.
Other browsers like Microsoft Internet Explorer or Google Chrome also support extensions which can interact with JavaScript code ; they would be scriptable plug-ins or Active-X controls. But surprisingly, the lack of security of ActiveX controls is well known whereas the lack of security of Firefox extensions seems to be ignored. In that extend, Firefox extensions have the potential to be much more dangerous than ActiveX controls since much more pernicious. That is why we will focus on Firefox extensions in this paper. It is important to understand that extensions, as said and studied in this paper, are instructions for an interpreter or a virtual machine, and so they are not considered as executables, but as bytecode or scripts, contrary to plug-ins or BHOs which are really executable binaries.

1.3 Extensions and Security

First of all, extensions can be installed by dropper programs, which is a first reason to worry when it comes to security. Moreover, extensions can disappear from the add-ons manager, and do so thanks to several techniques. If the extension is installed in the system's extensions directory, there is a hidden property to activate in the extension's installation manifest install.rdf. In the user profile directory, the

file's extensions.rdf is an XML document which lists the visible extensions. An extension can thus edit its entry in this file. And finally the extension can modify the code of the add-ons manager (overlay), and hide its presence by modifying the CSS of the manager.
Firefox has an Application Programming Interface called XPCOM (Cross Platform Component Object Model [14]) which is very powerful. It can be called directly from JavaScript code. The Firefox source code contains many examples with this API, for instance there is a documented way to create server sockets directly from JavaScript code. These functionalities allow to interact with Firefox and with the underlying system. They allow to read or write access to the Document Object Model (DOM) of the currently opened documents. Therefore, that gives the possibility to access browser entities such as the cookies or passwords store, read and write access to the file system, access to the network through client and server sockets, access to native code and libraries, granting the same rights on the system as Firefox. It is so, because the browser executes JSEs with elevated privileges in order to support the rich set of functionalities that they have to provide.
In opposition, JavaScript code in a web application is executed with restricted privileges, as sandboxing is used. Sandboxing is a popular technique for creating confined execution environments, which could be used for running untrusted programs. Indeed, JavaScript code in a web application can only access DOM elements and issue network requests (like XMLHttpRequest) in accordance with the same-origin policy and cannot access the file system. Then, even if a JSE is not malicious by itself, vulnerabilities in the browser and in JSEs can allow a malicious website to access and modify the privileges of a JSE and for example, either accessing the file system or executing commands on the host machine.
What seems particularly worrying, is the simplicity with which such extensions can be created. Indeed, in the reference paper on which we are working, they needed malicious extensions to test their system. They give this task of writing malicious extension to an undergraduate student, who had only a rudimentary knowledge of JavaScript. That is confirmed by this other document, a bachelor thesis of a Dutch student [3], in which he studies extensions and their big security lacks and show what is possible to do through examples, using code that he built himself and having the potential to steal bank data of a user, or even hacking a banking system. The point is that the JavaScript language is so powerful that you can make very strong actions with only a few lines of code. Some examples of malicious extensions (at least the core of their code) seen while making this work was not much longer than ten lines (a procedure code in JavaScript to steal a password for an example). The procedure to release an extension, even for malicious ones, is done here (again through the thesis of this Dutch student [3]). After creating a malicious extension, you will have to bribe a developer of a popular extension. Then, add the malicious code to the popular extension, and upload the new extension as a new update. Every user that has already installed this extension will get a pop-up asking to install the update when Firefox restarts (which is done automatically with only two clicks from the user). Such things, unknown from the great majority of users, are quite frightening at first sight, hence today's research to fix that.

2 SABRE

2.1 General overview

As seen previously, there are important lacks in browsers security. Hence this question : can we prevent from the possible attacks using browser weaknesses through JavaScript extensions ? We will see the example of Sabre (Security Architecture for Browser Extensions) which is a system that tracks the information-flow inside the browser to analyze JSEs.
The general idea on which Sabre is built is to associate each JavaScript object with a label which will determine if it contains or not sensitive information. If a JSE attempts to send a JavaScript Object containing sensitive data over the network or write it to a file, or to execute a script received from an untrusted domain with elevated privileges, Sabre will raise an alert.
Information-flow tracking has already been used to analyze plug-ins and browser helper objects, but Sabre tracks information-flow at the level of JavaScript instructions and within the browser which allows it to report better information about suspicious actions. Besides, JSEs use a lot of cross-domain calls to access web documents (DOM), local storage and the network. That is the reason why Sabre precisely tracks cross-domain flows,that means flows for which a JavaScript object is passed between different browser subsystems. Thus, Sabre tracks DOM nodes which are accessed by a JSE, putting labels in both DOM and JavaScript Interpreter. We will see the way it works deeper later and through examples.

2.2 How Sabre has been made

Sabre was implemented by modifying SpiderMonkey to track information flow. SpiderMonkey is the code name for the Mozilla's C implementation of JavaScript. It is basically the JavaScript interpreter for Firefox. The representation of JavaScripts object in SpiderMonkey was modified to support security labels. In order to propagate Information flow, the interpretation of JavaScript bytecode has also been enhanced to be able to modify labels. Moreover, some browser subsystems have also been modified, such as the DOM and XPCOM, to store and propagate security labels, that allows to track information flow in browser subsystems. Thanks to this approach, all the previous main goals are respected and controlled. All JavaScript code is executed by the interpreter, and so entirely labeled and controlled. Then associating security labels with JavaScript objects allows to track these labels inside the interpreter and every browser subsystems.

2.3 How Sabre works

Sabre has three main goals.
First, controlling all JavaScript code executed by the browser. It includes code in web applications, JSEs and JavaScript code executed in the browser core. This aim of controlling every JavaScript code is very important. Indeed, imagine that a malicious JSE copies data into a XUL element, which may then be read and transmitted by JavaScript in a web application. In such a case, it is important to track the flow of sensitive data through the JSE to the XUL element and into the Web application. That is why, as an attack can involve JavaScript code in multiple browser subsystems, it is so important to track all JavaScript code.
Then, Sabre has also to be tolerant not to alter the normal flow in the browser. Indeed, Sabre may raise an alert concerning a suspect information flow by a JSE, whereas it is part of the normal advertisement that this JSE provides. In such a case, we need an analyst to choose to trust the JSE and whitelist the flow ; this analyst must be able to quickly locate the JavaScript code that caused the information flow and determine whether or not it must be whitelisted. That is what is called : easy action attribution. To support declassification of sensitive information, Sabre extends the JavaScript interpreter with the ability to declassify flows. A security analyst supplies a declassification policy, which specifies how the browser must declassify a sensitive object.
Finally, Sabre also has to track information flow across browser subsystems. Indeed, JavaScript code in a browser and its JSE interacts heavily with other browser subsystems, as we already saw it, as the DOM, and persistent storage such as cookies or saved passwords, and even the local file system. Sabre has precisely to control very tightly information flows across these subsystems because attacks launched by JSEs involve browser subsystems.
One of the key concepts of Sabre is the Security labels. Sabre also modifies the interpreter to create security labels. It associates every JavaScript object with two security labels. One for tracking the flow of sensitive information. The other one to track the flow of low-integrity information. In the first case, the aim is to detect violations of confidentiality. In the second case, the violation in integrity. Each security label contains three pieces of information. First, a sensitivity level, to determine whether or not the label stores sensitive information. Then, a Boolean flag, to determine whether the object was modified by some JavaScript code in a JSE. And finally the names of the JSEs and web domains that may have modified the object. The sensitivity level is used to determine possible information flows violation, but Sabre will raise an alert only if the object was modified by a JSE. This policy of raising an alert only when an object is modified by a JSE is crucial to avoid false alerts. As Sabre tracks the execution of all JavaScript code, including code in web application and in the browser core, it can report confidentiality violations when sensitive data is accessed in a legal way, such as when JavaScript in a web application accesses cookies. That is the reason why Sabre has been designed to report an information-flow violation only when a sensitive object is modified by JavaScript code in a JSE.
The efficiency of the information flow tracking of Sabre is due to its Security labels. Sabre associates a security label with each JavaScript object, including objects of base type, such as Boolean or int, as well as complex objects such as arrays or compound objects with properties. For complex objects, Sabre associates additional labels, that is to say that every element of an array, or every component of a compound objects will have its own security label. And the label of the entire array is the aggregate of the different values. Sabre stores security labels by directly modifying the interpreter's data structures that represent JavaScripts objects. The interest of doing so is that it can use the object-oriented principle of heritage, which eases a lot the label propagation rule.
As JavaScript in a browser can interact with browser subsystems, especially the DOM, Sabre also modifies the browser's DOM subsystem to store security labels. Each DOM node has an

associated security label. This label is accessed and transmitted by the browser to the JavaScript interpreter when the DOM node is accessed in a JSE.

We already saw that Sabre modifies the JavaScript Interpreter to raise an alert when a sensitive object is trying to be sent through an output channel (which is called a low-sensitivity sink) through another subsystem of the host machine able to transfer data (like a smtp server). Sabre also modifies the DOM to raise an alert when the DOM node that stores sensitive information is sent over the network, as if a form or script element containing data which is labeled as sensitive is transmitted over the network. We have to notice that the file system is listed as both sensitive and low a sensitivity-sink. Indeed, a JSE is potentially able to steal some data from a web application and store it on the file system. Then another JSE can access the file system. But, the point is that the browser is continuously using the file system to read or write, for example for bookmarks and user preferences. So, no to raise an alert on benign flows, Sabre reports an information flow violation only if an object is written to the file system by a JSE.
In addition to propagating sensitivity values, Sabre uses the provenance of each JavaScript instruction to determine whether a JavaScript object is modified by a JSE. If so, it sets a Boolean flag and records the name of the JSE in the security label of the object for diagnostics. Because the JavaScript interpreter can precisely determine the source file containing the bytecode currently being executed, this approach allows Sabre to determine quite safely the provenance of an instruction.

2.4 Examples

Greasemonkey is a popular JSE that allows user-defined scripts to customize the background of web pages. Greasemonkey provides API functions which execute with elevated privileges because user-defined scripts must have the ability to read and modify arbitrary web pages. For example, the GM_XMLHttpRequest function allows a user-defined script to execute an XMLHttpRequest to an arbitrary web domain, and is not constrained by the same-origin policy. Vulnerabilities in Firefox and Greasemonkey allow scripts loaded on a web page to access the GM API functions and execute them with elevated privileges, and for example could allow access to the boot.ini file from the local system (through a GET request). An attacker could use a post to transmit this data over the network for example. Sabre would detect this attack because sensitive user data are used in an unsafe way. Sabre marks as sensitive every data that a JSE reads in a pre-defined set of sources (including the local file system). So, after a request from the script to access the local file system, the response will also be marked as sensitive (response from the GM_XMLHTTPRequest). And then the DOM nodes which stores this data will also then be marked as sensitive. When the browser will try to send content from the DOM over the network, Sabre will raise an alert for the user, since some code with a sensitivity label has tried to be executed.
Firebug is another popular JSE. It provides a development and debugging environment for HTML, CSS and JavaScript code. As a code development aid, Firefox exports a console interface which can be used by the scripts loaded in the browser can use to display messages within the firebug console. But vulnerabilities can allow a malicious web page to inject JavaScript code into the Firebug console. Remind that this injected code executes with chrome privileges and could reach for example the nsIProcess or nsLocalFile interfaces exported by XPCOM and start a process or either read or modify the content of a file on the local host. As it considers all data received from the console as untrusted, since this interface is exposed to web applications, Sabre would report an alert when the nsProcess or nsLocalFile is invoked with untrusted parameters (like some that are derived from data received through the console Interface).
Finally, we have an example of a real malicious JSE : FFsniFF (for Firefox Sniffer). This JSE can steal every entry that a user made on a HTML forms, including password entries, and save their values. And then it can transmits everything by mail by using smtp commands. On top of that, FFsniFF also hides itself from the user by exploiting a Firefox vulnerability in the Firefox extension Manager to delete its entry from the add-ons list presented by Firefox. Sabre is able to detect FFsniFF because it considers all data received from form fields on a web page as sensitive. Sabre will alert the user when FFsniFF will attempt to send the sensitive data by smtp.

3 How can we evaluate Sabre ?

3.1 Evaluation

Effectiveness We consider here the examples detailed earlier of JSE with malicious flows for the tests. First, let us have a look to the case of Greasemonkey and Firebug to study effectiveness of Sabre for vulnerable JSE. As we saw before, Sabre precisely identified the information flow violation, as we have already described it before. As also evocated in the presentation of Sabre's principle, Sabre did not raise any alert when we used a JSE-enhanced browser to visit benign web pages. Then, we consider FFsniFF to study the case of malicious JSE. AS we have also already seen it before, Sabre records the provenance of every JavaScript bytecode instruction executed and raised an alert when the malicious JSE attempted to transmit the data stolen to a remote attacker through the network. Many other JavaScript malicious JSEs have also been elaborated and tested to evaluate Sabre's capacity to detect them, and that successfully.

Performance The performance tests revealed that the runtime overhead with Sabre could be quite significant (6,1x) The memory consumption of Sabre was also measured, by browsing web pages with a large amount of JavaScript code. Opening four tabs, Google Maps with street views enabled, CNN, BBC and iGoogle for a fifteen minute browsing session, the memory consumed by the Sabre enabled-browser was approximately 74% more than that of an unmodified browser, which is very important.

3.2 Future of Sabre

Sabre appears as the most advanced tool in the domain of information flow tracking. Some prior work has already been done on the subject. Some techniques to detect spyware in browser extensions has been built [5, 6, 7], especially in plugins and BHOs. Like Sabre, these projects worked on information flow tracking [5] and on controlling plugin-browser interactions [29, 30]. But Sabre goes further, since it adds to the monitoring information function some supplements to trace sensitivity and integrity of JavaScript objects, and it performs information-flow tracking within the browser. The idea of information-flow tracking applied specifically on JavaScript is also not new. We have the example of Eich et al [8,9] and a lot has already been done on information-flow tracking for web applications, with SQL injection and script injection attacks. But Sabre with its cross-domain information tracking provides a much finer analysis of the information flows, which has never been done by now.
Other recent works consist in exploring the sandboxing possibilities of browser extensions, but this is only applicable to extensions such as plug-ins or BHOs which are binary executables. We can also mention Ter-Louw et al. [10] who opened the way in the study of security of JSEs. But their work was based on monitoring XPCOM and so Sabre goes much deeper in this approach.
Future research for Sabre will consist in studying the possibility of sandboxing and in improving performance by exploring static analysis of JavaScript code.

3.3 Limits of Sabre

If Sabre manages quite good to detect and block malicious JSEs, it can have some problems to deal with unknown JSEs, and has a bad natural tendency to block to easily a benign activity. The main problem is that malicious JSEs have a very similar behavior to safe JSEs, and it is impossible to make a clear and systematic distinction. So the distinction must be done by studying each case, which has to be done manually. For example PwdHash is a JSE which customizes passwords to prevent phishing attacks. In its procedure, it will read the password from an HTML form modifies it (converting it into SHA-1, a advanced crypting method) and writes it back to the HTML form. Such a JSE raises an alert for Sabre, that has to be declassified. But the difference is really tight between safe JSE and malicious. And we have many examples of this kind, at least one for each main function exploited by a JSE. As we have just seen the example in details for interaction with HTML forms, we have the same kind of examples for sending data over an HTTP channel, interacting with the file system, loading a URL. As an example, Web-of-Trust is a JSE that performs an XMLHttpRequest for each URL that a user visits, in order to fetch security ratings for that URL from its server. Such a behavior can potentially be misused by malicious JSEs to compromise user privacy. Then, we can also mention DownloadHelper and Greasemonkey which are JSEs made to download content from the network on to the file system (mediafiles and users scripts respectively) and ForecastFox which reads user preferences, such as Zip codes, from the file system and sens an XMLHttpRequest to receive weather updates from accuweather.com. Combining malicious JSEs misusing such behaviors and working together could really be dreadful in terms of security, both to download malicious files on the host and to steal personal user information. These are many examples to show that behavior can be so close from a safe to a malicious JSE that it seems to be impossible to decide automatically, and that an analyst has to do it every time manually. And I do not think that, in such an extent, we can consider Sabre has a reliable system with a long-term consideration. Alerts always raisen, and then systematically white-listing is no proof of a good engine. That is why, even if Sabre seems very good in its main job of tracking information-flow, it does not appear so obvious to use it in the everyday life, and so by a basic user who does not know anything about programming. Hence the need for something which could go deeper in this extent of browsing security and do more.

3.4 Related work

The actual structure of browsers is not safe. That is the point. Every attempt to develop tools to enhance greater security, can for sure improve the situation, but does not seem as a reliable solution with long-term consideration, as it will be, as always, a perpetual fights with attackers. Moreover, such tools (as Sabre or the other previous works evoked before) appear as attempts to try plugging the gaps of an unsecured system anyway. Indeed, modern web browser design still has roots in the original model of browser usage in which users viewed static pages and the browser itself was the application. Now, much more is required from a browser (as displaying multimedia content for example) and the old structure seems to reach its limits. Modern browser, keeping this basic-structure, have become much more complex, dealing with lots of plug-ins and extensions. The traditional same-origin policy, which may be the most used and main security policy for browsers, is not applicable anymore when it comes to the new complexity of browsers and their extensions. It is too restrictive to use with browser's plug-ins. We already saw all this in this paper, and studying the Sabre system and its complexity to deal with JSE and exceptions proves that once more. That is the reason why, a research-team from the University of Illinois had a much more radical approach, which is not without ambition : designing and implementing a new secure web browser.
The OP Browser has been made with this general policy to break the browser into several distinct and isolated components, and make all interactions between these components explicit. At the heart of its design is a browser kernel that manages each of our components and interposes on communications between them. This model provides a clean separation between the implementation of the browser components and the security of the browser. This approach results in the following architecture, that we can see on Figure 4 below.

We can see five main subsystems in this architecture: the Web page subsystem, a network component, a stor-age component, a user-interface (UI) component, and a browser kernel. Each of these subsystems runs within a separate OS-level process, and the Web page subsystem is broken into several different processes. The browser kernel manages the communication between the subsystems and between processes,

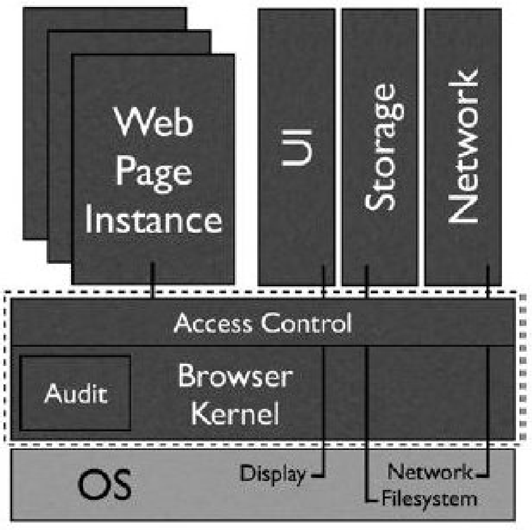

Figure 4: Overall architecture of the OP Web browser [11]

and it also manages interactions with the underlying operating system. Sandboxing techniques are used to limit the interactions of each subsystems with the underlying operating system.
We will not go deeper into this architecture, as the study of the OP browser is not the subject of this work, but we will see more in details what is the web page instance subsystem. We can see its breakdown on the following figure.

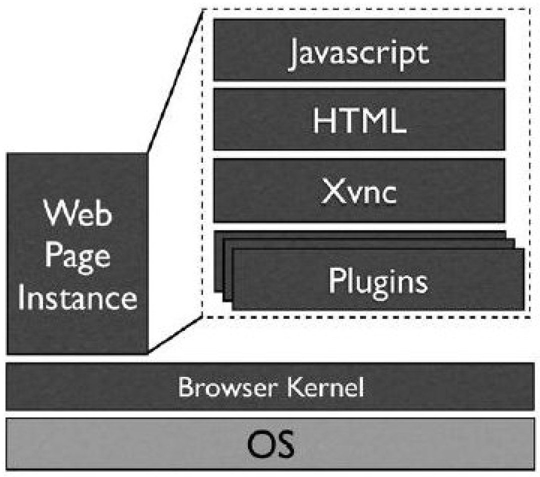

Figure 5: Web page Instance details [11]

The browser kernel creates a new Web page instance each time a user opens a new web page. For each Web page instance a new set of processes to build the Web page is created. Each Web page instance consists of an HTML parsing and rendering engine, a JavaScript interpreter, plug-ins, and an X server for rendering all visual elements included within the

page. The HTML engine represents the root HTML document for the Web page instance. The HTML engine delegates all JavaScript interpretation to the JavaScript component, which communicates back with the HTML engine to access any elements from the DOM. We run each plug-in object in an OS-level process and plug-in objects also access DOM elements through the HTML engine.
Such principles seem to be really interesting basics to solve some problems raised by our study of Sabre. We should study more in details the OP browser to judge whether or not it is really more reliable for a safe browsing. The point is that this new approach, with a new architecture, design to deal with extensions appears as the right one compared to all the problems and exceptions that Sabre had to face. The OP browser may also need a information-flow tracker to prevent it from attacks, but the idea to separate every elements in the structure should help a lot to make things easier than with a traditional browser.

Conclusion

As we saw at the very beginning, the basic structure of browsers is quite old and dates back to the very beginning of the internet. Since this time, the use of Information Technology has evolved a lot and so has the use of internet, and the content that browsers have now to display is much more various (including multimedia for example) and the functionalities that a browser needs are much more elaborated. To manage to offer these new functionalities, browsers work with extensions, and especially JavaScript extensions (JSE). But this use of extensions can become dangerous due to some important security failures.
As a proposal to solve the problem, Sabre provides an information flow tracking tool, which is really deeply thought and elaborated to fill the gaps of security of browsers. Its main principle is to study cross-domain information flows from one browser subsystem to another, and to do so it uses the concept of security label to determine if an element is safe or not, and sent an alert if yes. The way it deals with labels and associates them, linked to the JavaScript interpreter gives great results for detecting malicious extensions. But the only snag is that malicious extensions can be very close to safe extensions in the way they work, and that these safe extensions have to be whitelist by Sabre to run on the system. At the end, either Sabre blocks almost every extension looking a little suspicious (even if it is not necessary) or it exposes to be able to be hacked too. The work on Sabre is impressive in its way to be able to report any suspicious information flow, but the alert raised then and the rest of the procedure does not really provide a sustainable solution for an everyday use. It seems that the user will still have to make some concessions between functionalities and security. If he wants more functionalities, he must accept that his browsing is more risky.
A better solution may be to rethink fundamentally the structure of a web browser to adapt it to the security constraints the evolution requires. The OP browser developed by researchers of the University of Illinois is doing so and the approach seems really interesting. We will have to wait and see for the real results and improvements that it can bring.

References

[1] Mohan Dhawan, Vinod Ganapathy. Analyzing Information Flow in JavaScript-based browser Extensions. Rutgers University DCS Technical Report 648, April 2009.

[2] Alan Grosskurth, Michael W. Godfrey. Architecture and evolution of the modern web browser. June 2006.

[3] Julian Verdurmen. Firefox extension Security. Bachelor thesis. January 28th, 2008.

[4] Philippe Beaucamps, Daniel Reynaud. Malicious Firefox extensions. In *Symp sur la securite des technologies de l'information et des communications*, June 2008.

[5] M.Egele, C. Kruegel, E.Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *USENIX Annual Technical*, June 2007.

[6] E. Kirda, C. Kruegel, G.Banks, G. Vigna and R. Kemmerer. Behavior-based spyware detection. In *USENIX Security*, August 2006.

[7] Z. Li, X. Wang, and J.Y. Choi. Spychield : Preserving privacy from spy add-ons. In *RAID*, September 2007.

[8] B. Eich. Better security for JavaScript, March 2009. Dagstuhl Seminar 09141 : Web Application Security.

[9] B. Eich. JavaScript Security : Let's fix it. May 2009. Web 2.0. Security and Privacy Workshop.

[10] M. Ter-Louw, J.S. Lim, and V.N. Venkatakrishnan. Enhancingweb browser security against malware extensions. *Journal of Computer Virology*, 4(3), August 2008

[11] Chris Grier, Shuo Tang, and Samuel T. King. Building a more secure Web browser. August 2008.

[12] D. Turner, Symantec Internet Security Threat Report : Trends for January-June 07, Technical Report, Symantec Inc., 2007

[13] Firefox Add-ons. http://addons.mozilla.org

[14] Mozilla.org XPCOM. http://mozilla.org/project/xpcom

[15] XML User interface language (XUL) projects. http://mozilla.org/project/xul

[16] J. Rudermann. The same-origin policy. August 2001. http://mozilla.org/project/security/components/same-origin.html

[17] Sand Review system. https://addons.mozilla.org/en-US/firefox/pages/sandbox

[18] Same-origin policy for JavaScript. https://developer.mozilla.org/en/Same_origin_policy_for_JavaScript

[19] Document Object Model from W3C. http://www.w3.org/DOM

[20] DOM et JavaScript. https://developer.mozilla.org/fr/Le_DOM_et_JavaScript

[21] DOM http://www.gchagnon.fr/cours/dhtml/introdom.html

[22] Sandboxing. http://www.kernelthread.com/publications/security/sandboxing.html

[23] SpiderMonkey. http://www.mozilla.org/js/spidermonkey

[24] Firefox extensions. https://developer.mozilla.org/en/Extensions