Vetting Browser Extensions for Security Vulnerabilities

Risto Sandvik

Helsinki 28.3.2011

UNIVERSITY OF HELSINKI
Faculty of Science
Department of Computer Science
Modern web browsers have become rich platforms for application development. Arguably the most popular platform for writing web browser extensions is the Mozilla platform, which can be used to write extensions for the Firefox browser. Compared to many other platforms, however, Firefox lacks many security features. One such feature is the compartmentalization of different browser extensions. All extensions can do anything the browser can do, including writing data to the disk and executing other programs.

Before third-party Firefox extensions are made generally available through addons.mozilla.org, they go through a review process during which they are automatically and manually scanned, inter alia, for security vulnerabilities. In this report we examine an attempt by Bandhakavi et al. to improve the automatic scanning part of this process using a tool called “VEX” which employs static information flow analysis on the extension under review.

ACM Computing Classification System (CCS):

D.4.6 [Security and Protection]
Contents

1 Introduction ................................................. 1

2 Security implications of extending the Mozilla platform 2
   2.1 Background of the problem.............................. 2
   2.2 Points of attack........................................... 3

3 Static information flow analysis ........................ 5
   3.1 Improving the extension review process.............. 5
   3.2 Modeling information flows in JavaScript............... 6

4 Evaluation .................................................. 7
   4.1 Implementation............................................ 7
   4.2 Experimental results..................................... 7
   4.3 Example of a vulnerability (Wikipedia Toolbar)... 9

5 Conclusions ............................................... 9

References .................................................. 11
1 Introduction

“it is clear that there are many interacting components involved - - this makes securing a browser quite a monumental task.”
[ViM01]

The modern web browser has evolved from a simple document viewing application to a complex platform that can be used to fulfill most of our everyday computing needs. Current web technologies enable one to produce rich applications that run entirely within the browser. For example, products such as Google Docs can be used to replace distinct word processing and spreadsheet programs that once had to be installed locally on the user's computer.

The functionality of a browser can be further enhanced with extensions. Browser extensions interact with the internal data structures and algorithms of the browser to amend the functionality of the browser in a way desired by the user. The modern web browser's rich Application Programming Interface (API) makes it possible to build almost any conceivable application as an extension of the browser. In this way the web browser has evolved to resemble an operating system rather than a simple application used to view static documents.

This report deals with the Mozilla Firefox web browser and especially the extensions thereof. Firefox is a popular browser and much of its popularity is often attributed to the numerous extensions that are available for it at addons.mozilla.org (AMO). The extensions available for Firefox range from the very simple to, for example, complete FTP and IRC client programs.

Unfortunately, the easy development and deployment of third-party extensions also has its downsides. One clear downside is the inherent complexity of managing the security of an extendable system. It is not trivial to make sure that a program will behave like it should after it has been modified by third-party code that the developer of the program has not yet seen [ViM01]. In this paper we examine the extension architecture of the Mozilla Firefox web browser and the different kinds of vulnerabilities that can be found in the extensions. We also examine the process used by the Mozilla foundation in ensuring the safety of the extensions available at AMO and how this process can be enhanced
by using static information flow analysis as proposed by Bandhakavi et al. [BKM10].

2 Security implications of extending the Mozilla platform

2.1 Background of the problem

Extensions written for the Mozilla Firefox web browser use basically the same technologies as the web browser itself uses. A typical extension is written in Javascript and uses XUL (the “XML User Interface Language”) together with CSS and HTML to implement its user interface. Much of Firefox's low-level functionality is implemented as XPCOM (“Cross Platform Component Object Model”) components. These components are made available to the extension developer through an interface called “XPConnect”. The developer can also write her own XPCOM components using either JavaScript or C++. The downside of using C++ is of course the fact that the extension has to provide separate binaries for all the platforms it wishes to support. This is not necessary when using only JavaScript, which is an interpreted language. A general overview of the architecture of the Mozilla platform is provided in Figure 1.

Firefox has two privilege levels (contexts): page and chrome. The page privilege level is used for the actual content displayed in the browser window (typically an HTML page retrieved from a remote HTTP server) and the chrome privilege level is used for the code of the browser itself and its extensions. Page privileges are restricted in many ways as they are intended for untrusted content. For example, a page loaded from one site cannot access elements of a page loaded from another site. Chrome privileges, however, allow the executed code to access all functionality of the browser and all content of the pages loaded in the browser [BKM10].
Unfortunately, Firefox extensions running with chrome privileges are not compartmentalized in any way. An extension can freely modify the behavior of the browser and/or other installed extensions. The code of an installed extension is fully trusted by Firefox and it is treated as if it was part of the original code of the browser itself. This has lead Liverani and Freeman to describe the security model of Firefox extensions as “nonexistent” [FrL09].

One might think that there isn't much bad an extension written in JavaScript can do. Using the API provided by Firefox, however, code written in JavaScript can perform virtually anything on the target machine, including downloading custom software and executing it with the browser's privileges [Liv10].

2.2 Points of attack

A typical exploit of a security vulnerability in a Firefox extension involves the attacker injecting JavaScript code into a data item residing at the page privilege level and that

Figure 1: Overview of the Mozilla Platform. Available from Mike Schroepfer's blog at http://blog.mozilla.com/schrep/2007/05/16/mozilla-platform/
code getting subsequently executed by the extension at the chrome privilege level [BKM10]. Some typical points of attack are eval(), evalInSandbox(), innerHTML and wrappedJSObject. In the following each of these constructs is briefly explained to give an overview of the various possible security issues in extensions. It has to be noted, however, that this list is not exhaustive as there are other ways to introduce security vulnerabilities in an extension [BKM10, Moz11b].

The eval() function call is a standard construct of the JavaScript language. It interpreters a string passed to it as JavaScript and executes it dynamically thus allowing for great flexibility. Passing unsanitized data to eval() is an obvious security vulnerability as the data will be evaluated as JavaScript code and executed in the chrome privilege level. The eval() function call has many possible uses, but almost all of them are considered to be bad programming practice in the context of writing Firefox extensions [Moz11a, Pal09].

The Mozilla platform provides an alternative to eval() that allows for the secure execution of untrusted dynamic JavaScript in an isolated sandbox. This function is aptly named evalInSandbox(). Unfortunately it is quite easy to use this function in an insecure way. For example, using the “==” operator on the sandbox object leads to a security vulnerability because it calls the toString() or valueOf() method of the object, which could have been modified by the code executed inside the sandbox. Thus, the nontraditional “===” needs to be used [BKM10].

The innerHTML property allows one to easily modify the text that appears between the opening and closing tags of an element in a HTML document. The property can be used to alter existing document object model (DOM) elements or to add new ones. Altering the innerHTML property causes the browser to re-evaluate the document making a code injection attack possible. Sanitizing the string to be inserted as the innerHTML of an element is not trivial as there are numerous ways JavaScript code can be inserted through HTML tags (consider, for example, the following tag: “<img src='does_not_exist' onerror='code_injection_here' />”) [BKM10, FrL09].

As JavaScript objects can be dynamically modified, one has to use great care when dealing with objects that might have been modified by untrusted code. For example,
JavaScript running on a web page can overwrite the `getElementById()` method of the `document` object to return a malicious script. To remedy this, Firefox automatically wraps the `document` object so that the possible modifications made to it are not visible to an extension calling the `getElementById()` method (for example). However, Firefox also makes it possible for the extension to access the unwrapped version of the `document` object. This is done via the `wrappedJSObject` method. As described above, direct references to a content document using the `wrappedJSObject` method are potentially dangerous as any methods or properties of the object may have been overwritten with malicious code [BKM10].

### 3 Static information flow analysis

#### 3.1 Improving the extension review process

The Mozilla foundation has a pool of voluntary reviewers who review all submitted extensions before they are made available to the public on AMO. The review process is documented on the Mozilla developer network site [Moz11a].

During the extension review process the code of the extension is scanned with a grep-like tool for key indicators of bugs or security vulnerabilities. For example, the tool used in the review process will flag all uses of the `eval()` function call. Sections of the extension flagged by the scanner will then be manually inspected by the reviewer [Moz11b].

In their paper, Bandhakavi et al. explore a way to further automate the extension review process described above. As described above, in a typical vulnerability there is a distinct flow of information from an injectable source (in the page context) to an executable sink (in the chrome context). Using static information flow analysis it is possible to track information flows from injectible sources to vulnerable points of attack, thus pointing the manual reviewer to fewer and more probable points of vulnerability than a simple grep-based search [BKM10].

Bandhakavi et al. have identified five source to sink flows that they consider potentially vulnerable:

1. from Resource Description Framework (RDF) to `innerHTML`
2. from content document data to `eval()`

3. from content document data to `innerHTML`

4. `evalInSandbox` return objects used improperly in chrome context

5. `wrappedJSObject` return objects used improperly in chrome context

The first flow above might need some additional explanation. RDF is one of Firefox's ways to store and retrieve persistent data. Extensions can use the framework to store their persistent data or they can retrieve data stored by the browser itself using RDF. Firefox uses RDF to store bookmarks and this is a possible point of malicious code injection as the titles of bookmarked pages are stored un-sanitized. Therefore a user of this data must sanitize it properly before using it in an `innerHTML` statement.

The existence of one of the above mentioned flows in an extension does not automatically mean that there is a security vulnerability in the extension. Obviously information from non-trusted sources can be used if it is correctly sanitized before use [BKM10].

3.2 Modeling information flows in JavaScript

Analyzing JavaScript code statically, especially in the context of a Firefox extension, is not an easy task. A typical Firefox extension has a large number of objects that can be created, modified and deleted dynamically at run-time. Keep in mind that in JavaScript, functions are objects that can be dynamically redefined and passed as parameters like any other objects. Combining these properties of JavaScript with the objects and functionalities available through the DOM and XPCOM, it is no wonder that static information flow analysis has not been a part of the extension review process followed by AMO.

In their paper, Bandhakavi et al. analyze the execution of JavaScript code by keeping track of an abstract heap that tracks JavaScript objects and the relationships between them in the form of a graph. Each node in the graph is a heap location generated by the program and any two nodes `n1` and `n2` are connected by an edge labeled `f` if node `n1`'s property `f` may refer to `n2`. To keep track of the actual information flows between different variables, the analysis also keeps track of all program variables that flow into the nodes of the abstract heap. The authors further define the semantics of core JavaScript syntax (function definitions, assignment statements etc) and the way each of these core
constructs affects the state of the abstract heap used in the analysis. The end result of the analysis is an abstract heap (a graph) representing the objects of the program (nodes in the graph) and the relationships between them (the vertices in the graph). The generated graph can then be examined to determine the possible flows of information through the program. The interested reader is referred to the original paper ([BKM10]) for a more thorough explanation of the definition and construction of the abstract heap used in the analysis.

The analysis takes a few shortcuts when handling certain features used by Firefox extensions. The most obvious of these is the fact that strings that are not known statically (for example strings that are typed in by the user at run-time) passed to the `eval()` statement are completely ignored. Because of this and other simplifications used by the analysis the authors describe their method to be unsound and incomplete. However, this is something that might be expected from an automated static analysis tool [BKM10].

4 Evaluation

4.1 Implementation

For evaluating their method of analysis, the authors implemented the algorithm in a tool called “VEX”. VEX is written in Java and contains about 2000 lines of code. VEX uses the ANTLR parser generator (http://www.antlr.org) to parse the JavaScript code and then performs the static analysis described above. The implementation of VEX described in the paper searches for the five problematic flows enumerated in section 3.1 above. If a problematic flow is found, the tool reports the flow to the user along with the source code locations of the source and the sink of the flow [BKM10].

4.2 Experimental results

For their experiment, the authors downloaded two sets of extensions from AMO. First, in October 2008 they downloaded a random sample of 1827 extensions and in November 2009 they downloaded 699 of the most popular extensions. The two sets had 74 extensions in common, which makes for a total of 2452 extensions.

The extensions inputted to VEX after first being uncompressed. The results of running VEX on the extensions are summarized in figures 2 and 3.
### Flow Pattern

<table>
<thead>
<tr>
<th>Flow Pattern</th>
<th>grep</th>
<th>VEX</th>
<th>Attackable extensions</th>
<th>Source is trusted website</th>
<th>Not attackable</th>
<th>Unanalyzed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content doc to eval</td>
<td>430</td>
<td>13</td>
<td>2*</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Content doc to innerHTML</td>
<td>534</td>
<td>46</td>
<td>0</td>
<td>14</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>RDF to innerHTML</td>
<td>60</td>
<td>4</td>
<td>4**</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Attackable Extensions:** * WIKIPEDIA TOOLBAR V-0.5.7, WIKIPEDIA TOOLBAR V-0.5.9, ** FIZZLE V-0.5, FIZZLE V-0.5.1, FIZZLE V-0.5.2 & BEATNIK V-1.2

Figure 2: Flows from injectible sources to executable sinks [BKM10]

### Unsafe Programming Practices

<table>
<thead>
<tr>
<th>Unsafe Programming Practices</th>
<th>Grep Alerts</th>
<th>VEX Alerts</th>
</tr>
</thead>
<tbody>
<tr>
<td>EvalInSandbox object to == or !=</td>
<td>107</td>
<td>3</td>
</tr>
<tr>
<td>Method call on wrappedJSObject</td>
<td>269</td>
<td>144</td>
</tr>
</tbody>
</table>

Figure 3: Results for unsafe programming practices [BKM10]

In figure 2 the first column shows the number of extensions that were identified for manual inspection using the grep-based search for keywords in use in the AMO review process, whereas the second column shows the number of extensions in which VEX identified a problematic flow. To look for potential attacks, the authors manually reviewed “most” of the extensions flagged by VEX. In this manual review, they were able to confirm security vulnerabilities in six extensions, of which three are different versions of the Fizzle extension and two are different versions of the Wikipedia Toolbar. Three of the found vulnerabilities are thought to have been previously undiscovered. Of the 147 flows alerted by VEX in figure 3, the authors examined 15 and found them all to be “feasible and real” [BKM10].

As can be seen from figure 2, VEX flags considerably fewer extensions for manual inspection than the traditional grep search. Nevertheless, most of the alerts given by VEX in figure 2 seem to be false positives. Especially the flow from content document to innerHTML seems to be problematic for VEX and considerable manual effort is still needed in vetting the extensions for security vulnerabilities. As a concrete example of this manual effort it can be noted that the extensions in the last column of figure 2 were not analyzed because, according to the authors, they were too complex or obscurely written to be analyzed manually. The authors state that they spent an average of two hours per extension trying to confirm the vulnerabilities hinted by VEX [BKM10].
4.3 Example of a vulnerability (Wikipedia Toolbar)

The vulnerability found in the Wikipedia Toolbar extension is a rather simple one. The vulnerable code can be found in figure 4.

```javascript
script = window.content.document.getElementsByTagName("script")[0].innerHTML;
eval(script);
```

Figure 4: Wikipedia toolbar code [BKM10]

The code shown in figure 4 is executed if the user clicks on one of the buttons of the vulnerable version of the Wikipedia Toolbar extension. The code takes the first script element from the currently displayed web page and executes it in the chrome context using `eval()`. The author of the extension probably thought that the user will only click on the buttons of the toolbar while browsing the Wikipedia site. This is not, however, checked in any way and the extension will happily execute the first script on any web page with a click of a button. This might be enough to totally compromise the system on which the browser is running on.

This vulnerability has since been fixed by the author by inserting a check to make sure that the user is actually viewing a page from the Wikipedia domain before executing the first script from the page. This should remedy the issue as long as the content of the pages in the Wikipedia domain can be trusted not to contain malicious scripts [BKM10].

5 Conclusions

The algorithm implemented in VEX helped the authors of [BKM10] find three previously unreported security vulnerabilities on the extensions it examined. Based on this proof-of-concept experiment it appears that potential security vulnerabilities in browser extensions can be found by using static analysis for explicit flows of information. VEX does not, however, provide for a fully automated process. The suspected flows have to manually inspected and many of the flows will be false positives. The number of false positives could be reduced by correctly taking into account the input sanitizing methods used by the extension. Unfortunately this task is far from trivial.

The authors of the paper are aware that the source-sink pairs (flows) of the extensions
examined by VEX are not the only ones leading to possible security risks. The authors have identified other suspicious flows and the same can be deduced from other research in the area, such as [FrL10] and [Liv10].

Unfortunately finding vulnerabilities in Firefox extensions using static information analysis does little to help authors to write secure extensions. Developing extensions is surprisingly easy, but keeping them secure is another story. Further compartmentalization and privilege management of extensions could be used to minimize damages once extension security has been breached. This is an approach used by the Google Chrome browser [BKM10].
References


