2013 International Conference on Engineering of Complex Computer Systems

Protecting Web Browser Extensions from JavaScript Injection Attacks

Anton Barua, Mohammad Zulkernine, Komminist Weldemariam
School of Computing
Queen's University, Kingston, Ontario Canada
{barua, mzulker, weldemar}@cs.queensu.ca

***Abstract*—Vulnerable web browser extensions can be used by an attacker to steal users' credentials and lure users into leaking sensitive information to unauthorized parties. Current browser security models and existing JavaScript security solutions are inadequate for preventing JavaScript injection attacks that can exploit such vulnerable extensions. In this paper, we present a runtime protection mechanism based on a code randomization technique coupled with a static analysis technique to protect browser extensions from JavaScript injection attacks. The protection is enforced at runtime by distinguishing malicious code from the randomized extension code. We implemented our protection mechanism for the Mozilla Firefox browser and evaluated it on a set of vulnerable and non-vulnerable Firefox extensions. The evaluation results indicate that our approach can be a viable solution for preventing attacks on JavaScript-based browser extensions. In designing and implementing our approach, we were also able to reduce false positives and achieve maximum backward compatibility with existing extensions.**

***Keywords*—Browser Extension, JavaScript Injection Attack, Randomization, Static Analysis.**

I. Introduction

Web browser extensions add extra features on top of the standard functionalities of a browser. These added features can augment some facilities already present in the browser (e.g., enhance the bookmarking facility [1]), or provide entirely new functionalities (e.g., blocking unwanted advertisements [2]). Browsers also provide powerful APIs that can access highly privileged browser components to facilitate the development of feature-rich extensions. The ability to customize browsers with a wide range of useful features have made extensions extremely popular among users [3].

An extension can contain vulnerabilities which can be exploited by an attentive attacker to gain unauthorized access to sensitive information and even execute arbitrary code in the user's computer [4], [5]. Moreover, browser vendors let developers create extensions using standard web technologies (e.g., HTML, XML, JavaScript and CSS). Thus, an attacker needs nothing more than a rudimentary knowledge of these technologies to exploit an extension vulnerability.

Web browsers enforce an access control mechanism known as the Same Origin Policy (SOP) [6] to prevent the interaction of JavaScript code from one web origin with the content or JavaScript code of another web origin. However, browsers relax this policy for JavaScript-based extensions. In other words, extensions written in JavaScript receive higher privileges than the JavaScript code of webpages. Consequently, an extension can get unrestrained access to webpage resources from various origins and also to the internal components of the browser it extends. A successful exploit can thus utilize the privileges of a vulnerable extension to launch malicious attacks.

JavaScript injection attack is one of the common ways to exploit vulnerabilities in JavaScript-based browser extensions. Such attacks are far more devastating than other web-based attacks that subvert the SOP (e.g., cross site scripting [7], content sniffing attacks [8]). By exploiting a vulnerable extension, an attacker can escalate the privilege level of the injected code to the level of the extension, ultimately misusing the (higher) privilege to successfully control the victim's computer. Note that as JavaScript is a memory-safe language, injection attacks on JavaScript-based extensions can often be executed with guaranteed success as opposed to those written in memory-unsafe languages [9]. A large number of browser extensions are currently developed using JavaScript language[1].

The existing JavaScript security solutions employ static [10], [11], [12] and dynamic [13], [14] analysis techniques. However, static analysis techniques alone can only detect vulnerabilities in an extension before deployment and thus offer no runtime protection. Dynamic analysis techniques commonly employ taint tracking to monitor the flow of untrusted data inside a browser, but incur significant performance overhead and sometimes rely on users' judgement to distinguish vulnerable extensions from the non-vulnerable ones. An alternative proposed solution is to reduce the occurrences of vulnerabilities in extensions with secure extension APIs [15], [16], [17], [18]. However, these APIs require developers to redesign their extensions, and significantly reduce the versatility of extensions by limiting their capabilities.

In addition, even extensions written using secure APIs have been found vulnerable due to unsafe use of some powerful privileges [19], [20]. In addition, browser vendors typically encourage extension developers to upload extensions in a centralized repository after each extension has been reviewed for security defects by volunteers [21]. However, the manual reviewing process is error-prone as demonstrated by the discovery of vulnerabilities in reviewed extensions hosted in such repositories [22]. Moreover, browser extensions (not security-checked) are available for downloading outside the vendor-supported repositories, and many software silently install extensions in browsers.

Extension vulnerabilities manifest themselves after deployment, and current browser protection mechanisms are

[1]Notice that in four out of the top five major browsers, JavaScript is used as the primary programming language for extension development.

978-0-7695-5007-7/13 $26.00 © 2013 IEEE
DOI 10.1109/ICECCS.2013.36

inadequate and fail to protect users from exploitations in an active and transparent way. Thus, the goal of this paper is to develop a runtime protection mechanism that can prevent malicious code execution even in the presence of vulnerabilities in browser extensions. More specifically, we developed a runtime protection mechanism inspired by the Instruction Set Randomization (ISR) approach that transforms the source code of a browser extension to make it distinguishable from malicious attack code. The particular highlight of our approach is that it neither relies on any particular extension API nor it changes the existing extension platforms. Our approach is built based on the syntax of the JavaScript language, and it augments the security mechanisms already present in the browser.

We also developed a static points-to analysis technique for analyzing user supplied input parameters to dynamic JavaScript code generation functions (e.g., `eval()`) used in extensions. This component helps reduce false positives in our runtime protection mechanism which might disable legitimate dynamic JavaScript code generation in extensions. We implemented our approach in an instrumented Mozilla Firefox browser and evaluated it using samples from vulnerable and non-vulnerable Mozilla Firefox extensions. Our evaluation results indicated that the combined runtime protection and static analysis approach is adequate for preventing JavaScript injection attacks on extensions without requiring any changes to the underlying extension platforms.

This paper is organized as follows. The next section provides the technical details of the proposed approach. In Section III, we describe the implementation of our approach and provide the dataset used in evaluating our approach. We discuss the results of our evaluation in Section IV. While in Section V-A our approach is compared with related work, Section VI concludes the paper.

II. Proposed Approach for Runtime Protection

This section describes our proposed approach for protecting browser extensions from JavaScript injection attacks inspired by code transformation and static analysis approaches.

A. Approach Overview

Figure 1 shows an overview of our approach, which consists of two phases: an offline analysis phase and a runtime protection phase. In the offline analysis phase, using our randomization module named *JSRand*, we parse and perform appropriate code transformation on an extension's source code before installing an extension. We also perform source code analysis of the extension for potentially vulnerable dynamic code generation functions using our static analyzer module named *JSPoint*. The runtime protection phase enforces the protection after the extension is loaded and during its execution within the browser. The runtime component applies a reverse transformation on the randomized extension code to distinguish legitimate code from malicious ones.

A browser extension consists of several files and is usually available as a complete package in compressed format. As our target is to prevent JavaScript injection in an extension, we focus on the files that contain executable JavaScript code. Such code can be found in standalone JavaScript code files (i.e., `*.js` files) and also as event handler and inlined code in the user interface files of an extension. We extract all these files from an extension, select the files that might contain code, and feed these files to our code randomizer module.

The JSPoint module performs a static points-to analysis on dynamic code generation functions in case they are detected in the JavaScript code under analysis. Its output is a points-to result that indicates potential vulnerabilities in the dynamic code generation functions. Based on this output, we further transform some portions of the files that were already transformed by the randomizer module. In addition to the files that contain JavaScript code, browser extensions also contain a special configuration file, which contains extension specific configuration details. In particular, we are interested in determining the security origin of an extension a priori. Our *Configuration Parser* module extracts the security configuration which we use in the runtime protection component of *JSRand*.

After the offline transformation using JSRand and JSPoint, we repackage each extension and install them in our instrumented browser. We also supply the extracted configuration information to the instrumented browser for the runtime protection process. Finally, the *Derandomizer* component safeguards a given extension from JavaScript injection attacks while the browser is executing. The module is integrated in the JavaScript engine of our instrumented browser and invoked based on the security configuration information which we extracted earlier. In the remainder of this section, we discuss the details of our approach using examples.

B. (De)Randomization for Runtime Protection

We designed a lexical analyzer to convert a JavaScript source file into a stream of input elements. Our lexical analyzer is based on the *ECMAScript-262* specification —i.e., the standard for JavaScript [23]. We then proceed to randomize a selected class of input elements by appending a random key to the elements using a code transformation technique named *Instruction Set Randomization (ISR)*. ISR is used to alter executable instruction set of a runtime environment to defend against code injection attacks —e.g., see in [24], [25], [26], [7]. It transforms each of the instructions in the compiled code using a randomization algorithm coupled with a secret key. The secrete key is shared with the runtime environment for the reverse derandomization process. Any maliciously injected code into the program would fail to execute as the derandomization process would translate the injected code into invalid instructions —i.e., unrecognizable by the underlying runtime environment.

A straightforward adaptation of the general ISR approach is unsuitable for our specific case. More specifically, the design of the JavaScript engine in modern web browsers, the assignment of security principals to JavaScript code, and ambiguous origins of the JavaScript source files have led us to make the following design choices: i) the randomization process should be amenable to extensibility and provide good security guarantees, ii) randomization has to be applied on cleartext JavaScript code rather than the bytecode representation of the code inside the JavaScript engine, and iii) the origin of the JavaScript source files have to be identified correctly in order to apply derandomization on appropriate JavaScript code.

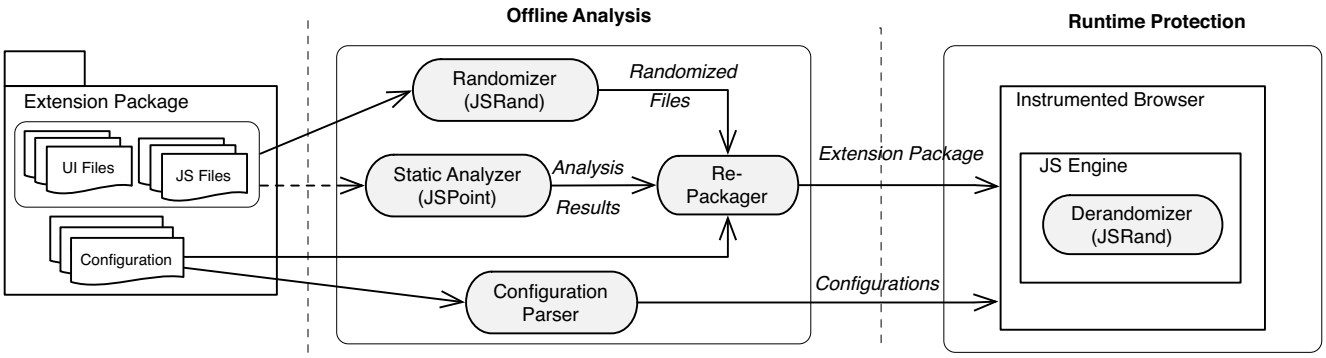

Fig. 1: Overview of the proposed approach.

Our randomization technique can be customized to support different key schemes. Randomizing in this way provides two benefits. First, it allows us to choose a key of arbitrary length as the length of the key is not limited by any implementation factors, e.g., the size of fixed-width instruction sets. Second, as the lexical analyzer can recognize different classes of input elements, it allows us to assign different keys to different input elements. The usage of different randomization schemes provide additional security guarantees over the typical randomization techniques such as XORing a single key of fixed length for each instruction set [24], or XORing the whole cleartext source code with a single random key [27]. For example, if a single key is XOR-ed to the extension code, an attacker only has to apply a brute-force attack powerful enough to discover a key of fixed length. However, as our approach is customizable via the addition of multiple keys and the length of key is not bounded, the complexity of a brute-force attack can raise significantly.

In what follows, we discuss the runtime protection process and then describe how our Derandomization module correctly identifies the sources of the JavaScript.

Runtime Protection Process

The goal here is to protect (vulnerable) browser extensions from JavaScript injection attacks at runtime. Attacker injected code becomes intermixed with valid extension code before it is turned into bytecode by the JavaScript engine. This intermixed code is assigned a privileged security principal and is executed by the JavaScript engine. Therefore, applying the randomization process at the bytecode level will not help us distinguish the code supplied by an attacker from the legitimate extension code. While a given (vulnerable) extension script is randomized with a secret key, the derandomization must take place before assigning any security principals.

After the vulnerable extension code is randomized, an attacker can try to inject malicious code into the randomized script. However, the derandomization process would generate a garbled form of the attack code, as the attack code was not randomized by the attacker using our scheme. Although the same security principal is assigned to this garbled malicious code, the JavaScript engine fails to produce the correct bytecode as the code does not conform to the syntax of the JavaScript language. Therefore, no valid bytecode is produced from the attack script. Ultimately, this would make the injection attacks useless as the derandomization process turns the attack script into invalid code within the execution environment. The JavaScript engine is robust enough to handle erroneous scripts (the garbled code) and it produces appropriate error messages without crashing or halting, making our solution fail-safe. An attempt of a key guessing attack can also be thwarted unless the attacker knows the correct combination of keys, which would be difficult due to our described customizable randomization technique.

Correct Identification of Security Origin

The Derandomization module requires to correctly identify the sources of JavaScript code in order to maintain the integrity of remote web applications and the browser core. The JavaScript engine of the Mozilla Firefox browser identifies remote JavaScript files using the *URL* of web applications. The engine also executes JavaScript originating from extensions and the browser user interface (local JavaScript files). These local JavaScript files are identified using the `chrome://` URL prefix scheme. This prefix is our starting point for correctly identifying the extension JavaScript code to be derandomized. In addition, we utilize the unique identifier of the extension to identify its namespace. For instance, a file `content/user.js` from an extension named `tmp_extension` will have a URL `chrome://tmp_extension/content/user.js` at the entry point of the JavaScript engine. The identifier of the extension can be found in the `chrome.manifest` file of the extension. Our Configuration Parser extracts all the configuration settings from the `chrome.manifest` file so that the Derandomization module utilizes the extracted information to accurately detect the extension script files.

Extensions in the Mozilla Firefox browser have the capability to enhance and customize the user interface of the browser. In particular, an extension can add user interface widgets such as buttons or icons to many parts of the browser window. This customization is enabled through the *overlay* feature of Firefox. Many extensions use this feature to overlay user interface widgets on top of the browser window. This overlay mechanism complicates the detection of JavaScript code for a given extension. For instance, as shown in Figure 2 (a), the `extension.xul` file is overlaid on top of `browser.xul`.

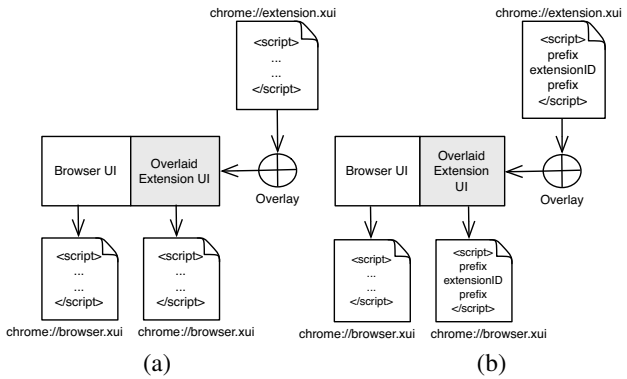

Fig. 2: The overlay process in Mozilla Firefox. (a) Extension scripts become indistinguishable from browser UI scripts due to the overlay process. (b) A unique prefix added by JSRand to overcome the problem.

The script code embedded in `extension.xul` is then identified in the JavaScript engine as if it is coming from the URL `chrome://browser.xul`. However, this URL cannot be used as a signal for derandomizing JavaScript code since the `browser.xul` already contains non-randomized JavaScript code, the core part of the Firefox browser itself.

We circumvent the above problem by using a unique prefix per extension that contains the identifier of the extension. As an extension file is overlaid on the browser user interface, the URL of the extension file is changed to the URL of the user interface. Thus, in the JavaScript engine, we insert the corresponding checks that match the prefix and only proceed to derandomize the code if a match is found. For example, in Figure 2 (b), the JavaScript code of an extension file `extension.xul` is detected as originating from `chrome://browser.xul`. However, the JavaScript code is prefixed with the identifier of the extension. Therefore, the code that follows it is originated from an extension and has to be derandomized. If the unique prefix is not found, we flag the code as originating from the browser and skip the derandomization process.

C. Static Analysis

While JSRand is able to protect extensions from JavaScript code injection attacks, it has no way of distinguishing code generated by dynamic code generation functions of JavaScript. Namely, the dynamic code generated from an extension by the `eval`, `setTimeout` and `setInterval` functions confuses our approach while distinguishing legitimate JavaScript code generated by these functions from malicious code injected by an attacker.

By design, the `eval` function inherits the security origin of the JavaScript code that invokes it. Thus, if an extension's script with origin `chrome://extcode.js` contains a call to `eval`, then at the JavaScript compiler's entry point, the code generated from `eval` would also be detected as originating from `chrome://extcode.js`. However, if the extension is vulnerable, any injected attack code would also be marked as originating from `chrome://extcode.js`. Moreover, as JSRand does not randomize string literals, the derandomizer component of JSRand could turn legitimate code generated from `eval` into malformed code, just as it would do for injected malicious code.

We now describe the design of our points-to component to analyze the JavaScript source code for deducing useful information without executing the program and its operation.

Points-to Analysis

To address this issue, we performed a static analysis to assess whether arguments passed to dynamic code generation functions are malicious or not. We adapted a static points-to analysis technique to analyze the source code of JavaScript-based browser extensions. However, existing points-to analysis approaches (e.g., see [28], [29]) are not directly applicable for our analysis due to the nature of the JavaScript language. Thus, we developed our own points-to static analyzer component (i.e., JSPoint). JSPoint is used to extract semantic information from an extension's JavaScript code so as to analyze and deduce the maliciousness of the arguments of dynamic code generation functions.

The internal machinery of JSPoint is based on an abstract memory model to represent reference variables and heap objects of the JavaScript program of an extension under analysis. While each reference variable is individually represented in the memory, objects are represented as a heap-allocated objects on an abstract global heap. We also modeled primitive values such as numbers and strings as heap allocated objects. Heap objects can also have properties, which are references themselves. These properties can point to other heap-allocated objects, but not to other references. The memory model is continuously updated to reflect the change of evaluating program statements and expressions. JSPoint thus can provide an approximation on the range of values that the parameters of dynamic code generation functions can point to.

Once the abstract memory is constructed, JSPoint proceeds by mimicking the execution model of JavaScript in browsers. In particular, we model the global context of JavaScript and function call invocations as abstract execution contexts. Each execution context is initialized with a parse tree of the code to be analyzed. The parse tree for each execution context is generated using the JavaScript parser that we built for this purpose. We also augment the parser with additional support to extract the code of a function body in order to build a parse tree of a function on the fly during analysis. Consequently, JSPoint evaluates the parsed representation of the expressions and statements and updates the corresponding abstract memory accordingly.

In addition, we incorporated tainting capability to our static analysis in order to isolate potentially untrusted values from trusted ones. JSPoint places a special parameter in each of the heap objects that marks whether the object is tainted or not. Whenever a reference is assigned a value from potentially untrusted sources (e.g., the DOM object `window.content.document`), it taints the heap object. The taint propagated as an object is assigned or passed as a function parameter to other references. At the end of the analysis, if we find that the target reference (e.g., a parameter to a dynamic code generator) points to a tainted object, we

can conclude that the use of such reference might potentially be unsafe.

Operation

Our static analysis phase closely mimics JavaScript execution in browsers. In particular, first we find user interface files of extensions which load potentially vulnerable script files. We then load all the scripts that the user interface file loads. Scripts in a browser are executed in a similar fashion. Namely, first the scripts contained within the `<script>` tags in a document file are loaded and executed. In case of multiple `<script>` tags, the scripts are sequentially loaded and executed to populate and modify the global JavaScript environment. We also need to load all the scripts that are loaded before and analyze them all at once. After feeding the necessary scripts to JSPoint, we mimic the event-driven nature of JavaScript by inserting calls to the functions that contain potentially vulnerable code generating functions. This allows the JSPoint module to reach the vulnerable function calls and generate points-to information for the function arguments.

Once the static analysis is performed, we determine that the parameter to the dynamic code generation function in question is tainted or not. If the parameter is tainted, we conclude that the call might be dangerous and we do not perform further modifications to the parameter. This is because of the recursive nature of the dynamic code generating functions which invoke the JavaScript compiler. The runtime protection mechanism will automatically try to derandomize the string used to generate code dynamically. This would convert the string into garbled JavaScript code, causing any attack attempts through these functions to fail. If, however, JSPoint finds no taint, we add a unique string prefix to the parameter. This allows the randomizer component in the JavaScript Object subsystem to detect and randomize the code fragment passed as a parameter to the dynamic code generation function. Thus, when the non-malicious randomized string reaches the compiler, it is correctly derandomized by the runtime component. In this way, we permit legitimate dynamic code generation to occur, which allows us to safely execute non-vulnerable extensions while causing potentially malicious dynamic code generation to fail.

III. Implementation and Dataset

We implemented the core JSRand and JSPoint modules by utilizing and extending existing toolsets. The core JSRand module includes a JavaScript lexical analyzer and our randomization module. The implementation of JSRand is shared across the offline JavaScript source transformer, document parser (e.g., to transform user interface (XUL) files, XML), as well as the JavaScript engine. The parser of XUL files is built on top of the RapidXML [30] utility.

For the implementation of our static analysis (i.e., JSPoint), we extended JSLint —a JavaScript parsing utility [31]. We also modify the `EvalKernel` function of the JavaScript engine, which is the actual implementation of the `eval` function. Our runtime randomization component is invoked before `EvalKernel` sends the code to the JavaScript compiler. The other dynamic code generation functions (i.e., `setTimeout`, `setInterval`) are not implemented inside the JavaScript engine since they are implemented in the event handler system as part of the DOM system. Hence, instead of appending a unique prefix to the non-malicious parameters to these functions, we randomize the parameter itself if it is a string-type object and is valid a JavaScript code.

We integrated our implementation within an instrumented Mozilla Firefox browser (Version 9.0). This version uses SpiderMonkey as its JavaScript engine. SpiderMonkey compiles JavaScript code embedded in `*.js` files and `<script>` tags using the `CompileScript` function. JavaScript code embedded in event handler functions of DOM elements and extension widgets (e.g., buttons) are compiled using a separate function named `CompileFunctionBody`. These functions take cleartext JavaScript code, the length of the code, the origin of the code, and the security principal to be assigned to the code, along with other parameters as input. Finally, we also modified the compiler and JSObject modules of SpiderMonkey to integrate with the runtime environment.

A number of vulnerable and non-vulnerable Mozilla Firefox browser extensions are used in our experiment —see Table I for some of the samples we used in our experiment. Ten of these extensions contain JavaScript injection vulnerabilities, which we collected from various vulnerability databases such as CVE [32] and OSVDB [33], as well as from academic research [11]. In some of the vulnerable extensions, vulnerabilities are present across multiple versions. The non-vulnerable extensions are downloaded from the Mozilla Add-on repository.

Our evaluation suit contains two extensions that are vulnerable to JavaScript injection attacks that pass untrusted web content as parameter to the `eval` function. The remaining eight vulnerable extensions in our evaluation suite are due to different DOM modification functions and properties (e.g., `document.write()`, `node.innerHTML`).

The number of vulnerable and malicious extensions that we have found in real-world is very few to run large scale experimentation. The reason is that public disclosures about vulnerable extensions are made after the vulnerabilities have been fixed, and for a number of instances, the vulnerable extensions are removed from the Mozilla Add-on repository. This makes it challenging to obtain the copies of these vulnerable extensions for evaluation purposes. Although we were unable to evaluate our approach for the unavailable extensions, we have studied their vulnerability reports and have developed more extensions that mimic those vulnerabilities.

IV. Evaluation

To assess the effectiveness of our approach in protecting browser extension from JavaScript injection attacks, we tested on forty real-world Mozilla Firefox extensions. Our evaluation methodology consists of the following steps.

First, we build attack scenarios tailored for each particular vulnerability. Second, we randomize the extension source code using JSRand. In case dynamic JavaScript generation functions are found within the source code, we invoke JSPoint to perform a points-to analysis in order to reveal potential maliciousness of each input parameter passed to these functions. Based on the results of JSPoint, either we further modify the extension by adding a prefix or by randomizing the input parameters (non-malicious parameter values), or we leave the input parameters

TABLE I: Sample Vulnerable (V) and Non-Vulnerable (NV) Browser Extensions.

Extension Name	Version	Type	Source	Note
Fizzle	0.5 - 0.5.2	V	CVE 2007-1678	Script injection via RSS Feed DOM
Beatnik	1.0, 1.2	V	CVE 2007-3110	Script injection via RSS Feed DOM
Wikipedia Toolbar	0.5.7, 0.5.9	V	CVE 2009-4127	Script injection in `eval()` from the DOM
UpdateScanner	<3.0.3	V	OSVDB 57403	Script injection in update XUL page
Feed Sidebar	3.1, 3.2	V	OSVDB 57404	Script injection via RSS feeds
ScribeFire	<3.4.3	V	OSVDB 57405	Script injection via event handlers in blog editor
Firebug	1.7.2	V	OSVDB 73203	Script injection in NET preview window
Kaizou	0.5.8	V	VEX [11]	Script injection in web editor window
Iplex to AllPlayer	0.7.0	NV	Mozilla Add-on	...
SearchPreview	5.3	NV	Mozilla Add-on	...
Password Exporter	1.2.1	NV	Mozilla Add-on	...
Download Manager Tweak	0.9.5	NV	Mozilla Add-on	...
FB Chat Sidebar Disabler	1.9.7	NV	Mozilla Add-on	...
YouTube to MP3	1.2.3	NV	Mozilla Add-on	...
Google Shortcuts	2.1.7.1	NV	Mozilla Add-on	...

intact (potentially malicious parameter values). Third, we repackage the randomized extensions and install and attack them with exploit scripts. Finally, for both vulnerable and non-vulnerable extensions, we measure the performance overhead.

```
1 var loginManager = Components.classes["
    @mozilla.org/login-manager;1"].getService(
    Components.interfaces.nsILoginManager);
2 var count;
3 var logins;
4 logins = loginManager.getAllLogins({});
5 var stealNamePass = logins[0].username+logins
    [0].password;
6 window.open("http://attacker.com/echonamepass.
    php?namepass="+stealNamePass);
```

Listing 1: Password stealing exploit.

An Exploit Example. Listing 1 shows an example of exploit that steals passwords and sends them to remote servers. In Line 1, the malicious code first accesses the `nsILoginManager` service of Mozilla Firefox, which is a privileged component responsible for maintaining users password storage. In Line 4, the script inserts all the login names and passwords in an array by invoking the `getAllLogins()` function. In Line 5, the script gets the first login name-password pair from the array. In Line 6, the attack script posts this login name-password pair by redirecting the browser to a website named `attacker.com` and posts the stolen information to a server-side script that stores the login name-password pair. Note that, without an extension's privilege, Lines 1, 4 and 5 cannot even be executed. Only by injecting this code in a vulnerable extension, an attack can be carried out.

Similarly, a number of attack scenarios are devised by targeting the particular vulnerabilities for each vulnerable extension. Once the necessary exploits are devised, all the JavaScript and the user interface files contained in `chrome/content` folder (where Mozilla Firefox extensions' main program resides) are randomized. Whenever any dynamic code generation function is detected in a given extension under analysis, our static analysis module performs the necessary points-to analysis. This takes place before invoking the randomization module. During the course of randomization, we use the extracted configuration (from the `chrome.manifest`) to separate overlay files from non-overlay files.

We carried out the above exploit (see Listing 1) on the Wikipedia Toolbar extension by embedding the exploit script as the first `<script>` element in a webpage and clicking one of the toolbar buttons. The exploit is targeted to the vulnerability present in the `eval` function of Wikipedia Toolbar extension. The extension was assumed to be used only when a user is visiting the Wikipedia website. When a user clicks on any of the toolbar buttons of the extension, it extracts the first `<script>` element on a webpage and passes the script to `eval`. Therefore, if the extension's toolbar buttons were clicked while visiting a website that embeds malicious attack script code in its first `<script>` element, the attack executed successfully. Similarly, for other vulnerable extensions, we developed exploits by using suitable mechanisms (e.g., embedding attack script in RSS feeds, hosting malicious content in remote servers).

```
1 function wpedia_prefs(event){
2     script = window._content.document.
3       getElementsByTagName("script")[0].
4       innerHTML;
5     eval(script);
6     if (event.button == 1)
7     ...
8  }
9 wpedia_prefs({});
10 ...
```

Listing 2: Vulnerable portion of the Wikipedia toolbar extension.

The combination of JSRand and JSPoint was able to prevent JavaScript injection attacks on extensions that had vulnerable calls to `eval`. Namely, JSPoint revealed a potentially malicious flow into an `eval` function invocation on Wikipedia Toolbar (see Listing 2). In the listing, the `script` variable is assigned a value originating from a webpage. Later, the variable is passed to `eval` without any input validation. JSPoint detected this potentially malicious parameter assignment as shown in Listing 3. In the listing, the `script` variable points to an object which might contain tainted data ($\{t\}$ means

tainted) when passed to `eval`. Other variables used in the extension are not tainted ($\{n\}$ means non-tainted). In such cases, we do not concatenate a unique prefix to the parameter so that our in-browser component can effectively detect and randomize it.

```
1 wpedia_showhide->Function194{n}
2 wpedia_usertalkpage->Function195{n}
3 wpedia_contribs->Function196{n}
4 wpedia_prefs->Function197{n}
5 script->object{t}
```

Listing 3: JSPoint finds potential malicious parameter values passed to the `eval()` function through the `script` variable (see Listing 2).

Out of the eight vulnerable extensions that can be exploited through different DOM modification functions and properties (e.g., `document.write()`, `node.innerHTML`), seven extensions have been protected by our approach. For instance, we launched an attack using a malicious RSS feed on the Feed Sidebar extension —a Mozilla Firefox extension that lets users to subscribe to RSS feeds and displays the feeds in a sidebar panel. A user subscribes to the feed by clicking the "Subscribe Now" button. The extension displays the feed in the sidebar. When the "click me" button is clicked from the sidebar, the exploit steals username-password pairs and posts them to a server under an attacker's control. We converted the source code of the Feed Sidebar extension and performed points-to analysis, which yielded no malicious parameter values in dynamic code generation functions. We then installed the extension and executed the attack. However, the derandomization process was able to turn the attack code into invalid code, causing the JavaScript engine to throw a syntax error exception. Note that this vulnerability was not discovered by a previous static analysis approach (see [12]) for detecting extension vulnerabilities.

TABLE II: Summary of the performance overhead of our approach. All the values are given in seconds.

Extension	Randomization	Derandomization
Vulnerable	0.93	0.53
Non-Vulnerable	1.43	1.07

Finally, we evaluated the performance overhead of our approach. Specifically, we measured both the time needed to randomize an extension file and the corresponding derandomization time in the browser. A summary of the average overhead times is shown in Table II. During our experimentation, we did not notice any major slowdown in the browsing experience during our evaluation. The observed runtime overhead is mostly one-time, as the browser compiles the extension files when they are first loaded. In fact, the performance overhead increases with the size of the source files. Moreover, our evaluation is done using an instrumented version of the Mozilla Firefox, which is considerably slower than an actual release build. Thus, we believe that the overhead can be reduced significantly with performance tuning and by applying the approach on real production version of the browser.

V. Related Work

In recent years, a number of research efforts have been made to secure browsers and browser extensions from malicious activities. The techniques proposed span dimensions such as heuristic-based analysis, static analysis, and dynamic analysis. In this section, we review related work with a particular focus on efforts to specifically secure browser extensions by applying static, dynamic and randomization techniques.

A. Securing Browser Extensions

Here we discuss approaches that focus on detecting and/or protecting vulnerable JavaScript-based extensions. A comparison of our approach with these approaches is shown in Table III.

VEX [11], [12] applies a static analysis on JavaScript code of Mozilla Firefox extensions to identify vulnerable information flow patterns. The authors used operational semantics of a core subset of JavaScript to generate abstract heaps from the extension code. VEX can detect the existence of information flow patterns from a sensitive source to an executable sink (e.g., dynamic code generation functions). If not detected, such flow patterns can enable an attacker to inject malicious code in a sensitive source that can later be run with elevated privileges at an executable sink. In some cases, VEX generates flows that do not result in exploitable attacks, leading to false positives.

Beacon [10] is another static analysis tool that targets Mozilla's new Jetpack extension framework. It tries to identify capability leaks in Jetpack modules by performing a static points-to analysis on an extension's source code. First, an extension's source code is transformed to a Static Single Assignment (SSA) form, which in turn is used to generate a function call graph. Later, the extension code is converted to a database of facts. In combination with the call-graph, this fact database is used to find out capability leaks in Jetpack [18] extensions. Similar to VEX, Beacon also detects capability leaks that cannot be exploited.

SABRE [13] tracks the flow of JavaScript objects from sensitive sources to sinks inside the Mozilla Firefox browser by employing a dynamic taint analysis technique. Whitelisting is used to separate benign extension flows from malicious ones. However, the whitelist approach essentially delegates the responsibility of deciding the maliciousness of an extension to a user. Similarly, a dynamic taint analysis based approach to detect vulnerable extensions is presented in [14]. This approach attempts to prevent unprivileged data from being compiled into privileged bytecode. It also identifies and prevents privileged caller functions from accidentally calling unprivileged code.

IBEX [17] is a general purpose browser extension development system that provides verifiable security guarantees. It provides an API that lets extensions to use common browser functionalities. Privileges of an extension are specified in a custom policy language. The extension code can then be formally verified against the specified policy. A cross-compiler is available to deploy the same extension in Internet Explorer, Google Chrome, Mozilla Firefox and C3 (an experimental browser). However, to use IBEX, extension developers need to write their extensions in a verifiable language other than JavaScript. Moreover, each browser provides unique APIs

TABLE III: Comparison with other extension vulnerability mitigation approaches.

Work	Technique	Runtime Protection	Requires API Redesign
VEX [11], [12]	Static Information Flow Analysis	No	No
Beacon [10]	Static Points-to Analysis	No	No
SABRE [13]	Dynamic Taint Analysis	Yes	No
Djeric and Goel [14]	Dynamic Taint Analysis	Yes	No
IBEX [17]	Formal Verification	Yes	Yes
Barth *et al.* [15]	Privilege Separation	Yes	Yes
Our Work	Randomization and Points-to Analysis	Yes	No

for its extension system that is constantly updated with new features, making it difficult to develop extensions using a general purpose system like IBEX.

The Google Chrome extension system is proposed with an attempt to restrict extension privileges by adhering to the principle of least privilege and providing strong isolation between the components of an extension [15]. The privileges of an extension are limited by explicitly declaring the privileges during installation. Privileges cannot be changed after installation. As noted in [20], this system cannot prevent developers from writing extensions that arbitrarily request powerful privileges. A recent report also showed that such extension system is vulnerable to injection attacks [34].

B. Randomization to Prevent Injection Attacks

Similar to our work, some research groups have also used randomization techniques to detect injection attacks in system and web applications. For instance, a process-specific instruction-set randomization approach for binary programs is proposed in [24]. Specifically, the code segment of a program executable is transformed using a randomization scheme such as XOR. When the program runs as a process, the runtime environment uses a key (stored in the executable program) to derandomize the encoded machine code instructions. An injection attack targeting the running process fails as the attack instructions are derandomized into a non-executable form. One shortcoming of this approach is the lack of support for dynamically linked libraries, as these libraries have to be randomized beforehand to ensure a proper execution. A similar approach attempting to prevent injection attacks on binary executables is also presented in [25].

The basic ISR approach for protecting binaries suffers from inadequate security and performance, ultimately making it infeasible for practical deployment. Hence, the authors in [35] proposed an approach that provides robust security guarantees with reduced performance overhead. In particular, they use a combination of software dynamic translation and an AES encryption, rather than a simple XOR scheme to randomize binary instructions. The randomized binaries are executed by a virtual machine rather than processor emulators which leads to tolerable performance overheads. SQLRand [36] uses the concepts of ISR to prevent SQL injection attacks. In recent works, researchers have proposed variants of ISR to prevent cross site scripting (XSS) attacks. Noncespaces [7] randomizes the XML namespaces of a document produced by a server to isolate attacker supplied content. xJS [27] randomizes the JavaScript code of an web application at the server side.

So far, we are unaware of any randomization-based protection mechanisms for JavaScript-based browser extensions. Moreover, some of the randomization-based XSS prevention approaches modify the dynamic code generation functions of JavaScript, and consequently require developers to rewrite code using new instructions. Our approach does not require modifying the dynamic code-generation functions, thus relieving the burden of rewriting from developers.

C. JavaScript Static Analysis

Several approaches employ static analysis based technique to analyze JavaScript code for security —e.g., [37], [38], [39], [11], [10]. For instance, Gatekeeper [38] performs a static points-to analysis on JavaScript widgets from widget-hosting portals for detecting malicious widgets. Gatekeeper converts a JavaScript program into a database of facts. These facts are passed to a constraint solver to discover points-to information, which is later used to assess the maliciousness of the widgets. Gulfstream [37] extends Gatekeeper with a graph-based and constraint-solver-based techniques. It performs a staged static analysis composed of an online and offline points-to analysis of JavaScript code. ACTARUS [39] applies points-to analysis combined with taint analysis on JavaScript code to discover code injection vulnerabilities in web applications. VEX [11] and Beacon [10] are two static analysis tools employing dataflow and points-to analysis techniques for discovering vulnerabilities in JavaScript-based browser extensions.

Our work is different from these approaches as we only perform static points-to analysis on demand for some specific JavaScript functions. In particular, we limit the scope of our static analysis to dynamic code generation functions, and we only invoke static analysis if those functions are present in an extension's code.

VI. Conclusion

In this paper, we have presented a runtime protection facility that protects browser extensions from JavaScript injection attacks. The presented approach does not require a change to the underlying extension platform or APIs of the browser, and it is flexible as we allow different randomization schemes of varying complexity.

We implemented our approach in a prototype tool for Mozilla Firefox extensions and evaluated the efficacy and overhead introduced. In designing and implementing the runtime protection, we found that our approach can remove the unnecessary burden of rewriting an extension for a new API from developers and can provide maximum backward

compatibility with existing extensions. The approach can be integrated in a browser for providing transparent runtime protection mechanism with negligible overhead while keeping the normal features of the extensions intact.

Future work will focus on implementing more efficient techniques and in extending our offline analysis and runtime protection system with more advanced techniques. We plan to explore in-browser integration of static analysis techniques which would nullify any possibilities of incorrectly modeling the internal APIs of the browser.

ACKNOWLEDGMENT

This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] "XMarks Sync," https://addons.mozilla.org/en-US/firefox/addon/xmarks-sync/?src=cb-dl-featured, 2012.

[2] "AdBlock Plus," https://addons.mozilla.org/en-US/firefox/addon/adblock-plus/?src=cb-dl-users, 2012.

[3] Z. Li, X. Wang, and J. Y. Choi, "Spyshield: preserving privacy from spy add-ons," in *Proceedings of the 10th international conference on Recent advances in intrusion detection*, ser. RAID'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 296–316. [Online]. Available: http://dl.acm.org/citation.cfm?id=1776434.1776457

[4] J. Wang, X. Li, X. Liu, X. Dong, J. Wang, Z. Liang, and Z. Feng, "An empirical study of dangerous behaviors in firefox extensions," in *Proceedings of the 15th international conference on Information Security*, 2012, pp. 188–203.

[5] R. S. Liverni and N. Freeman, "Abusing Firefox Extensions," in *Proceedings of the DEFCON 17 Hacking Conference*, 2009.

[6] J. Ruderman, "Same Origin Policy for JavaScript," https://developer.mozilla.org/en-US/docs/Same_origin_policy_for_JavaScript, 2012.

[7] M. v. Gundy and H. Chen, "Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks," in *Proceedings of the 16th Network & Distributed System Security Symposium*, 2009.

[8] A. Barua, H. Shahriar, and M. Zulkernine, "Server Side Detection of Content Sniffing Attacks," in *Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering*, 2011, pp. 20 –29.

[9] Y. Ukai, D. Soeder, and R. Permeh, "Environment Dependencies in Windows Exploitation," in *Proceedings of the Black Hat (Japan) Hacking Conference*, 2004.

[10] R. Karim, M. Dhawan, V. Ganapathy, and C.-c. Shan, "An Analysis of the Mozilla Jetpack Extension Framework," in *Proceedings of the 26th European Conference on Object-Oriented Programming*. Springer, 2012, pp. 333–355.

[11] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett, "VEX: Vetting Browser Extensions for Security Vulnerabilities," in *Proceedings of the 19th USENIX Security Symposium*. USENIX Association, 2010, pp. 339–354.

[12] S. Bandhakavi, N. Tiku, W. Pittman, S. T. King, P. Madhusudan, and M. Winslett, "Vetting Browser Extensions for Security Vulnerabilities with VEX," *Communications of the ACM*, vol. 54, no. 9, pp. 91–99, Sep. 2011.

[13] M. Dhawan and V. Ganapathy, "Analyzing Information Flow in JavaScript-Based Browser Extensions," in *ACSAC*. IEEE Computer Society, 2009, pp. 382–391.

[14] V. Djeric and A. Goel, "Securing Script-Based Extensibility in Web Browsers," in *Proceedings of the 19th USENIX Security Symposium*. USENIX Association, 2010, pp. 355–370.

[15] A. Barth, A. Porter Felt, P. Saxena, and A. Boodman, "Protecting Browsers from Extension Vulnerabilities," in *Proceedings of the 17th Annual Network & Distributed System Security Symposium*, 2010.

[16] "Google Chrome Extensions API," http://developer.chrome.com/extensions/api_index.html, august, 2012.

[17] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy, "Verified Security for Browser Extensions," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2011, pp. 115–130.

[18] "Mozilla Developer Network - Jetpack," https://developer.mozilla.org/en-US/docs/Jetpack, 2012.

[19] A. P. Felt, "Security Bugs in Google Chrome Extensions (and How to Avoid Them)," http://www.adrienneporterfelt.com/blog/?p=226, 2011.

[20] A. P. Felt, K. Greenwood, and D. Wagner, "The Effectiveness of Application Permissions," in *Proceedings of the 2nd USENIX Conference on Web Application Development*. USENIX Association, 2011.

[21] "Add-ons for Firefox," https://addons.mozilla.org/en-US/firefox/, august, 2012.

[22] "Vulnerable Firefox Extensions," http://secunia.com/community/advisories/search/?search=firefox+extension, 2012.

[23] "ECMAScript Language Specification," http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf, June 2011, august, 2012.

[24] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks with Instruction-Set Randomization," in *CCS*. ACM, 2003, pp. 272–280.

[25] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," in *CCS*. ACM, 2003, pp. 281–289.

[26] A. N. Sovarel, D. Evans, and N. Paul, "Where's the FEEB? the effectiveness of instruction set randomization," in *Proceedings of the 14th conference on USENIX Security Symposium*. USENIX Association, 2005.

[27] E. Athanasopoulos, V. Pappas, A. Krithinakis, S. Ligouras, E. P. Markatos, and T. Karagiannis, "xJS: Practical XSS Prevention for Web Application Development," in *Proceedings of the 2010 USENIX Conference on Web Application Development*. USENIX Association, 2010.

[28] J. Whaley and M. S. Lam, "Cloning-based Context-Sensitive Pointer Alias Analysis using Binary Decision Diagrams," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. ACM, 2004, pp. 131–144.

[29] B. Steensgaard, "Points-to Analysis in Almost Linear Time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1996, pp. 32–41.

[30] "RapidXML," http://rapidxml.sourceforge.net/.

[31] "JSLint," http://www.jslint.com/.

[32] "CVE - Common Vulnerabilities and Exposures," http://cve.mitre.org/, august, 2012.

[33] "The Open Source Vulnerability Database," http://www.osvdb.org/, august, 2012.

[34] N. Carlini, A. P. Felt, and D. Wagner, "An Evaluation of the Google Chrome Extension Security Architecture," in *Proceedings of the 21st USENIX Security Symposium*. USENIX Association, 2012.

[35] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill, "Secure and Practical Defense Against Code-Injection Attacks using Software Dynamic Translation," in *Proceedings of the 2nd International Conference on Virtual Execution Environments*. ACM, 2006, pp. 2–12.

[36] S. W. Boyd and A. D. Keromytis, "SQLrand: Preventing SQL Injection Attacks," in *Proceedings of the Second International Conference on Applied Cryptography and Network Security*. Springer, 2004, pp. 292–302.

[37] S. Guarnieri and B. Livshits, "Gulfstream: Incremental Static Analysis for Streaming JavaScript Applications," in *Proceedings of the USENIX Conference on Web Application Development*. USENIX Association, 2010.

[38] Guarnieri, Salvatore and Livshits, Benjamin, "Gatekeeper: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code," in *Proceedings of the USENIX Security Symposium*. USENIX Association, 2009, pp. 151–168.

[39] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the World Wide Web from Vulnerable JavaScript," in *Proceed-*

ings International Symposium on Software Testing and Analysis. ACM, 2011, pp. 177–187.