CrossFire: An Analysis of Firefox Extension-Reuse Vulnerabilities

Ahmet Salih Buyukkayhan
Northeastern University
bkayhan@ccs.neu.edu

Kaan Onarlioglu
Northeastern University
onarliog@ccs.neu.edu

William Robertson
Northeastern University
wkr@ccs.neu.edu

Engin Kirda
Northeastern University
ek@ccs.neu.edu

***Abstract*—Extension architectures of popular web browsers have been carefully studied by the research community; however, the security impact of interactions between different extensions installed on a given system has received comparatively little attention. In this paper, we consider the impact of the lack of isolation between traditional Firefox browser extensions, and identify a novel *extension-reuse* vulnerability that allows adversaries to launch stealthy attacks against users. This attack leverages capability leaks from legitimate extensions to avoid the inclusion of security-sensitive API calls within the malicious extension itself, rendering extensions that use this technique difficult to detect through the manual vetting process that underpins the security of the Firefox extension ecosystem.**

We then present CrossFire, a lightweight static analyzer to detect instances of extension-reuse vulnerabilities. CrossFire uses a multi-stage static analysis to efficiently identify potential capability leaks in vulnerable, benign extensions. If a suspected vulnerability is identified, CrossFire then produces a proof-of-concept exploit instance – or, alternatively, an exploit template that can be adapted to rapidly craft a working attack that validates the vulnerability.

To ascertain the prevalence of extension-reuse vulnerabilities, we performed a detailed analysis of the top 10 Firefox extensions, and ran further experiments on a random sample drawn from the top 2,000. The results indicate that popular extensions, downloaded by millions of users, contain numerous exploitable extension-reuse vulnerabilities. A case study also provides anecdotal evidence that malicious extensions exploiting extension-reuse vulnerabilities are indeed effective at cloaking themselves from extension vetters.

I. Introduction

Major web browsers, including Firefox, Chrome, Internet Explorer, Safari, and Opera, provide extension mechanisms that allow third parties to modify the browser's behavior, enhance its functionality and GUI, and integrate it with popular web services. A large pool of browser extensions are published in centralized repositories such as Firefox Add-ons [26] and the Chrome Web Store [11], and are downloaded by millions of users.

NDSS '16, 21-24 February 2016, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23149

As a result of their increasing popularity, browser extensions have also become increasingly targeted by attackers. Extensions can often access private browsing information such as cookies, history and password stores, and also system-wide resources. For instance, Firefox exposes a rich API to its extensions through XPCOM (Cross Platform Component Object Model) [29] that allows nearly-unrestricted access to sensitive system resources such as the filesystem and network. Consequently, malicious extensions, or attacks directed at legitimate extensions, pose a significant security risk to users. The research community has recognized this threat, presented studies and tools that analyze the security properties of extensions [3], [4], [7], [8], [13], [16], [37], and proposed various defenses [31], [35], [38].

However, despite the abundance of research focusing on the security of browser extensions in isolation, to the best of our knowledge, the possible interactions between multiple browser extensions have not been well-studied from a security perspective. In particular, the Firefox extension architecture allows all JavaScript extensions installed on a system to share the same JavaScript namespace, hence making it possible for an extension to invoke the functionality (or modify the state) of others. This problem has long been recognized as a namespace pollution problem that can introduce errors if multiple extensions define identical global names [27]. However, its impact on security has not been studied so far.

In this paper, we first introduce a new class of Firefox extension attacks that exploit *extension-reuse vulnerabilities*. These vulnerabilities allow a seemingly innocuous extension to reuse security-critical functionality provided by other legitimate, benign extensions to stealthily launch confused deputy-style attacks. Malicious extensions that utilize this technique would be significantly more difficult to detect by current static or dynamic analysis techniques, or extension vetting procedures. The malicious extension itself does not make any sensitive API calls or resource accesses, which allows the malicious behavior to stay hidden. In addition, automated analysis of such malicious extensions would require covering the code from the entire extension pool available to Firefox users since the attack could utilize code from any and multiple extensions, which would considerably increase the complexity of the analysis task.

Next, we present a lightweight methodology to automatically discover possible extension-reuse vulnerabilities, which involves static data-flow analysis to identify flows between globally accessible identifiers defined in extensions and security-sensitive XPCOM calls. Using these flows, a

malicious extension can indirectly access the XPCOM API. We implement this technique in a tool that we call CROSSFIRE. Our system produces two types of output: automatically-generated exploits that can immediately be used to validate the presence of vulnerabilities, and *exploit templates* that can be adapted by users of the tool to rapidly construct working proof-of-concept exploits. Similarly, CROSSFIRE's output could be utilized by extension developers to identify and secure the vulnerable code sections.

Finally, we study the prevalence of extension-reuse vulnerabilities in a large pool of Firefox extensions, evaluate the effectiveness of CROSSFIRE in discovering vulnerabilities, and demonstrate that extension-reuse vulnerabilities have real-life impact through concrete examples. The results of our analysis and a random sample study show that many exploitable extension-reuse vulnerabilities exist among the top 2,000 Firefox extensions. In particular, 9 of the top 10 extensions are exploitable. Furthermore, we present anecdotal evidence that demonstrates the potential for malicious extensions that exploit extension-reuse vulnerabilities to bypass Mozilla's vetting process.

In summary, this paper makes the following research contributions.

- We present a novel class of attacks that abuse the lack of isolation between Firefox extensions to perform *extension-reuse*. This technique allows an outwardly benign, but actually malicious, browser extension to reuse functionality available in other legitimate extensions to launch stealthy attacks.
- We introduce a lightweight static analysis, implemented in a tool called CROSSFIRE, to automatically discover extension-reuse vulnerabilities, generate exploits that confirm the presence of vulnerabilities, and output exploit templates to assist users of the tool in rapidly constructing proof-of-concept exploits.
- We provide a detailed analysis of the top 10 Firefox extensions to report on the automatically-generated and human-crafted exploits discovered, estimate the effort required to construct a working exploit from exploit templates, and demonstrate the practical impact of the generated attacks.
- We analyze a pool of the top 2,000 Firefox extensions, and examine in detail a random sample of 323 (i.e., targeting a 5% confidence interval at a 95% confidence level). We estimate the occurrence of extension-reuse vulnerabilities, and report false positive rates for CROSSFIRE.
- We present anecdotal evidence we obtained by crafting a sample extension that exploits an extension-reuse vulnerability in a popular extension, NoScript, and show that it could pass the extension vetting process undiscovered. (We highlight the fact that our sample extension did not actually contain a malicious payload.)

II. PROBLEM STATEMENT

In this section, we briefly present some background information on Firefox extension development, define the problem of extension-reuse vulnerabilities and attacks in detail, and explain our threat model.

A. Firefox Extensions

Firefox extensions, also called add-ons in Mozilla parlance, add new functionality to the browser, change its behavior, enhance the GUI, and interact with web page contents. Firefox gives extensions access to a powerful API through XPCOM [29], which is a framework that provides various services to applications built on the Mozilla platform. As a result, extensions can access sensitive system resources such as the filesystem and network with the same privileges that the browser process runs with.

Firefox extensions are written in JavaScript and XUL (XML User Interface Language, which is an XML dialect used by Mozilla to define graphical user interfaces). They communicate with XPCOM through a glue layer called XPConnect, which exposes the various XPCOM components' interfaces to JavaScript. Extension developers may also utilize functionality from third-party binaries, or they may create their own binary XPCOM components; as a result, certain extensions contain a mix of JavaScript and native code.

Recently, Mozilla developed an alternative framework for extension development, called the Add-on SDK, as part of the Jetpack project [30]. This framework provides extension authors with a high-level API for an easy development process, and addresses some of the security issues associated with regular, legacy Firefox extensions, by restricting access to XPCOM and isolating extension modules from each other. While Mozilla encourages the use of the Jetpack framework, a large body of popular legacy extensions are still in use. Moreover, the simplified API of the Jetpack framework is not feature-complete and, therefore, various extensions use a mix of the legacy extension development techniques and Add-on SDK to access more powerful XPCOM features where necessary. In fact, a recent study [32] shows that in June 2014, only 10.6% of the top 1,000 Firefox extensions were built using the Add-on SDK. We have also performed a similar preliminary experiment to verify those results. In particular, we crawled the Mozilla Add-ons website for extensions tagged "Jetpack". Our results show that, as of October 2014, 12.0% of the top 2,000 Firefox extensions are developed using the Jetpack framework, while the remaining 88.0% are legacy extensions. While these results indicate that the adoption of the Jetpack framework may be increasing, a clear majority of the top extensions are still using the legacy extension development methods.

B. Extension-Reuse Vulnerabilities

Firefox extensions share the same JavaScript namespace – in other words, every extension installed on a system can freely access all of the JavaScript names defined in the global scope by each extension. This problem has been identified by the Mozilla community in the past, and it has been recommended that each extension define its own namespace to avoid JavaScript name collisions [27]. However, the security implications thereof has been left largely unexplored so far. In particular, this shared JavaScript namespace makes it possible for extensions to read from and write to global variables defined by others, call or override all global functions, and modify instantiated objects.

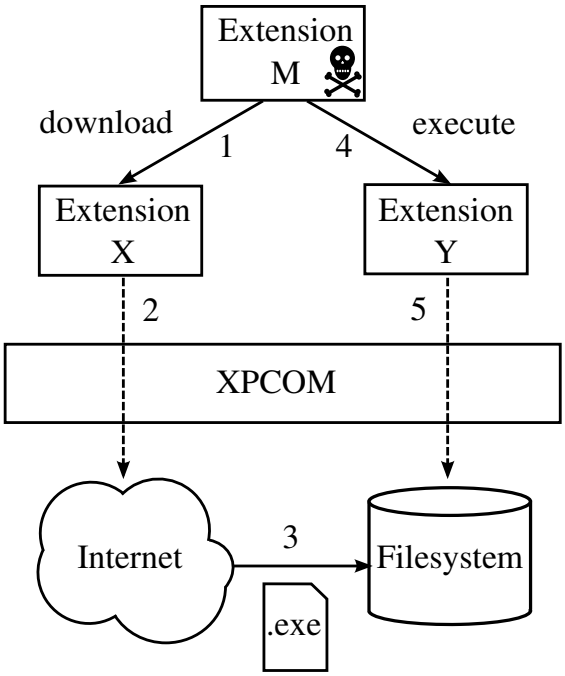

Figure 1. A sample extension-reuse attack showing the malicious extension *M* reusing functionality from two legitimate extensions to indirectly access the network and filesystem. In this way, the malicious extension discreetly downloads a malicious file and executes it.

While a malicious extension might attempt to perform an attack simply by invoking the corresponding API calls from its own code, this malicious functionality is likely to be detected before the extension can be made available on Mozilla's online repository. This is due to the requirement Mozilla imposes on every extension to pass a review process that involves functional testing and source code reviews by human extension vetters [25]. Moreover, the security research community has presented numerous analysis systems that can automatically vet extension code to identify or block suspicious behavior (see Section VI).

We observe that Firefox's shared JavaScript namespace can be exploited by a malicious extension to stealthily launch attacks on the system, and bypass the countermeasures described above. We define an *extension-reuse vulnerability* in a given extension as a control or data flow from a global JavaScript name to a security-critical API call (e.g., one that provides access to the filesystem or the network, or allows arbitrary code execution) that results in a *capability leak*. Since global JavaScript names are available to all extensions, an attacker can often identify a sufficient set of capability leaks to write a malicious extension that indirectly invokes critical APIs through other, legitimate extensions, to mount a confused deputy-style attack. This attack scenario is illustrated in Figure 1. Here, (1) a malicious extension *M* exploits a capability leak from a legitimate extension *X*, (2,3) and uses it to download a malware executable to disk. Next, (4) extension *M* exploits another capability leak from a different extension *Y* to access the filesystem, and (5) executes the previously downloaded malware.

We can state the described attack scenario more formally as follows.

We let:

$\mathcal{C}$ be the set of all capabilities provided by the extension framework,
E be the set of extensions installed on the system,
$C_e \subseteq \mathcal{C}$ be the set of capabilities leaked by $e \in E$,
$A \subseteq \mathcal{C}$ be the set of capabilities required to launch an attack a.

Then, an attacker can write a malicious extension to launch the extension-reuse attack a if:

$$A \subseteq \bigcup_{e \in E} C_e$$

Since the malicious extension's code does not contain direct calls to the APIs that enable the attack, detecting such malicious activity through human reviews requires evaluation of the malicious extension in the context of all possible Firefox extensions, or extending automated analysis to cover the entire extension code base, which renders the task costly or infeasible.

Finally, we note that while it is possible to combine multiple extension-reuse vulnerabilities in this way to craft complex attacks, it is often sufficient to use a single vulnerability to successfully launch damaging attacks, making this attack practical even when a very small number of extensions are installed on a system. For example, an attacker can simply redirect a user that visits a certain URL to a phishing website, or automatically load a web page containing a drive-by-download exploit. In this paper, we assume that every instance of an exploitable extension-reuse vulnerability may potentially be used to compromise the security of a Firefox user, and refer to all such exploits as *attacks* for brevity.

We demonstrate later in Section IV-A and Section IV-B that many extensions downloaded by millions of users contain exploitable extension-reuse vulnerabilities.

C. Threat Model

The threat model we consider for this work primarily involves the common scenario in which an extension developer writes an extension, and submits it to Mozilla's online extension repository to make it publicly available. Users then download and install the extensions to their systems.

In this scenario, we assume that an attacker has access to the extension pool published on Mozilla Add-ons web page, and that she can download and analyze them offline to identify any extension-reuse vulnerabilities they might contain. Subsequently, the attacker crafts a malicious extension that exploits a set of extension-reuse vulnerabilities in any number of popular legitimate extensions to perform the desired malicious activity, and submits it to Mozilla. We assume that the malicious extension is subjected to Mozilla's regular extension review process, which includes functional testing and human code reviews, before it is made available online. We do not make any assumptions about whether the attacker makes any deliberate attempts to make the extension review more difficult, such as obfuscating the source code. However, we assume that the attacker takes care to adhere to the minimum

requirements for Mozilla reviewers to plausibly consider an extension for acceptance. For example, we assume that the malicious extension implements some innocuous functionality as a cover.

On the user's side, we assume that all installed extensions have full access to XPCOM APIs, and that they can run with the same privileges as the browser process, as all Firefox extensions normally do. We also assume that the attacker takes the necessary precautions so that the malicious extension fails silently when the required set of vulnerable extensions to perform the attack is not installed on the user's system.

While the attack technique described in this work is also applicable to certain Jetpack extensions that improperly export global names, and those that mix the use of Add-on SDK and the low-level APIs, this paper primarily focuses on legacy Firefox extensions which constitute the majority of the popular Firefox extensions (i.e., 88.0%), as explained previously. However, we also briefly discuss how extension-reuse attacks could be adapted to Jetpack extensions, and the results of our preliminary experiments with them in Section V.

Finally, note that this work does not consider attacks on non-JavaScript components of extensions, such as binary executables packaged together with the extension. Similarly, binary browser *plug-ins* (e.g., Flash player, PDF viewers), which are distinct from *extensions*, are outside the scope of this paper.

III. Analysis with CrossFire

In this section, we present an overview of our tool called CrossFire, describe how we utilize static control- and data-flow analysis to detect and exploit extension-reuse vulnerabilities in Firefox extensions, explain several example vulnerabilities found by this analysis, and discuss limitations of the analysis.

A high-level overview of the main components that comprise CrossFire is presented in Figure 2. The system takes as input the target extension's source code and a database of security-sensitive browser API calls that represent potential sinks in the data-flow analysis. First, a JavaScript parser module processes the code and generates the corresponding abstract syntax tree (AST) representation. Next, the vulnerability analyzer component processes this AST in two stages. Stage 1 is a basic pass over the AST to compute a simple approximation of the call graph, and to collect information essential to performing the subsequent, more involved analysis. Using this information, the Stage 2 analyzer performs a taint analysis from globally-exposed JavaScript names to any security-sensitive APIs contained in the database. Finally, the results of the analysis are fed into an *exploit generator* component, which produces either an exploit to validate the presence of the vulnerability, or provides the user with *exploit templates* to assist in manually crafting malicious extensions. The core components of CrossFire are further discussed in this section.

A. Vulnerability Analysis

Our approach to detecting extension-reuse vulnerabilities is primarily an example of static data-flow analysis, where sources are globally-accessible JavaScript identifiers and sinks are security-sensitive calls to XPCOM and other browser APIs. In our attack model, an adversary can interact with legitimate extensions in three ways: (i) by modifying the contents of global variables (which might contain JavaScript primitives or more complex objects) that flow into security-sensitive APIs as call arguments, (ii) by directly invoking globally exposed functions that, in turn, invoke those APIs later during execution, or (iii) by overriding globally defined functions (e.g., callbacks for security-sensitive event listeners). All of these methods allow an attacker to indirectly access security-sensitive APIs and, therefore, our analysis needs to consider all globally defined identifiers as analysis sources. The goal of the analysis, then, is to determine whether any of these global variables and functions – which can be directly accessed by a malicious extension – can allow an adversary to invoke a sensitive API functions with attacker-controlled arguments.

Table I. Examples of security-sensitive XPCOM and browser APIs used by CrossFire as data-flow sinks during vulnerability analysis.

Operation	API call
Code Execution	initWithPath, launch eval
File I/O	initWithPath, asyncCopy, asyncFetch
Network Access	loadURI, saveURI, open
Clipboard Access	getTransferData
Cookie Store Access	getCookieString
Bookmarks Access	exportBookmarksHTML getBookmarkURI
Password Store Access	getAllLogins
Preference Access	getBranch
Event Listener Registration	addEventListener

A non-exhaustive list of the prominent sinks that are considered by our analysis is presented in Table I. Flows from global identifiers that are accessed or tampered with by the attacker into these sinks could lead to attacks involving: binary and JavaScript code execution; file and network I/O operations; key logging by hooking the appropriate event listeners; and access to and modification of private browsing data, stored credentials, clipboard contents, and other potentially sensitive information.

The static analysis proceeds in two stages. In Stage 1, CrossFire traverses the AST of the extension code to build a more compact representation of the program suitable for detecting extension-reuse vulnerabilities. This stage involves a lightweight *context-insensitive* analysis to identify all globally exposed JavaScript identifiers and generate an under-approximation of the function call graph. During this stage, CrossFire also performs an intraprocedural analysis on each function it encounters to generate function summaries. The summaries produced by CrossFire capture data dependencies between function arguments and return values, dependencies and side effects on global variables, the presence of any sinks in the function and, if present, whether any arguments

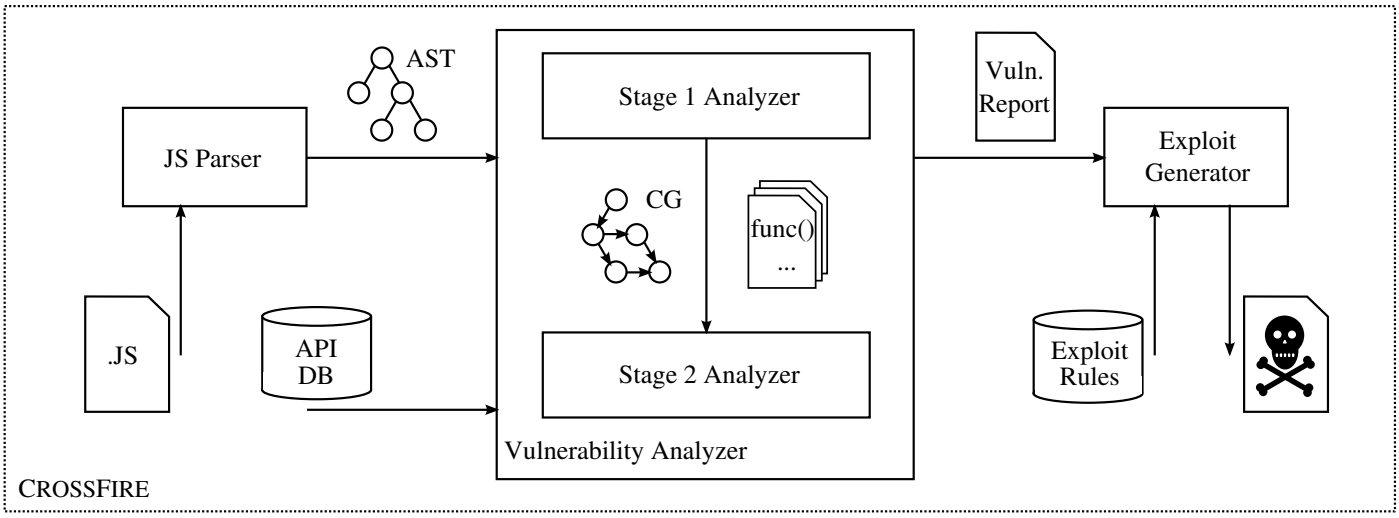

Figure 2. An overview of the core components of CROSSFIRE.

or globals flow into those sinks. This intraprocedural analysis uses a classic data-flow worklist algorithm that iterates until convergence to a fixpoint is reached.

Next, CROSSFIRE uses the information collected in Stage 1 of the analysis to perform the second stage. In Stage 2, CROSSFIRE inspects the call graph, identifies all possible paths between sources and sinks, and discards the remaining paths. On this set of retained paths, the system performs a forward *context-sensitive* taint analysis, where context is defined as a bounded call chain of depth k. For each non-constant global identifier, a unique taint label is assigned. Then, the following taint propagation policy is applied: (i) assignments to primitive variables are tainted with the union of taint labels from the right-hand side; (ii) assignments to an object field result in propagation of the union of taint labels from the right-hand side to the entire object; (iii) function invocations result in a transfer of taint from parameters and referenced global variables to any return value or modified global variables according to the function summaries recovered from the prior intraprocedural analysis.

If the analysis detects that a tainted value flows into a sink, it performs various sink-specific checks to ensure that an actual vulnerability exists. For instance, if the sink is an event listener registration, CROSSFIRE checks the other arguments to the corresponding API call to determine whether the event listened to is security-sensitive (e.g., a key press event could be used for key logging).[1] Once a vulnerability is confirmed, CROSSFIRE passes this information to the exploit generator component.

B. Exploit Generation

The exploit generator is invoked when a vulnerability is found during static analysis. It uses predefined, sink-specific rulesets to generate exploit samples. These rulesets specify the sensitive arguments of the sinks, their types, and semantic meaning (e.g., to indicate that a certain argument should be replaced with a malicious URL, or with the path to a binary). The generated exploits take the form of simple variable assignments when the taint source is a global variable, or function calls with malicious arguments if the source is a global function definition.

While in certain cases the above approach directly yields a working exploit, more complex data flows can make it difficult to automatically generate attacks without performing a more rigorous and precise analysis. These cases occur, for instance, when the taint status of a variable cannot be tracked accurately due to complex, nested control-flow structures, when tainted values are sanitized before they reach the sinks, or when invoking a vulnerable global function requires the attacker to specify additional arguments, the types of which are unknown. In such cases, CROSSFIRE instead produces an exploit template to assist the user with manual rapid development of a working proof-of-concept. In particular, the exploit template includes the corresponding path in the call graph, relevant source code line numbers, names of tainted identifiers and their flows, and the target sinks reached. In effect, this template declares that while a potential vulnerability exists, it cannot be confirmed due to the unsoundness and imprecision inherent in our approach, and manual intervention is required to confirm it. We discuss in Section IV-C how long it takes for a human analyst to create working exploits from these templates.

C. Example Vulnerabilities

In the following, we provide examples of vulnerabilities found in extensions listed among top 10 on the Mozilla Add-ons repository at the time of writing, and show exploits generated by CROSSFIRE that can be used in concrete attack scenarios.

1) Open URL: Flash Video Downloader is an extension that allows users to extract and download multimedia content embedded inside Flash files. We discovered a vulnerability in this extension that allows opening and displaying the contents of a URL in a new browser tab. Since the effects of exploiting this vulnerability is visible to the user, it would best be used in conjunction with attacks that require user interaction, such as opening a phishing page when the user attempts to visit a

[1] We note that, in general, identifying specific events that listener functions are registered for requires a string analysis. However, in practice, events are virtually always specified directly as string literals in the function invocation.

specific legitimate URL. The code presented below is one of the simpler vulnerabilities to exploit, only requiring a method invocation on a global object `fvd_single`, with a single parameter passed by the attacker. It is an example of a working proof-of-concept exploit that was automatically generated by CROSSFIRE.

```
// Attacker simply calls the global function
// below with a malicious value $url
fvd_single.navigate_url($url);
```

2) Send HTTP Request: This vulnerability in *Web of Trust*, an extension that crowdsources security ratings for websites, sends an HTTP request to an attacker-specified URL. However, unlike the previous example, it does not display the contents to the user. As such, it can be used to communicate with, or exfiltrate data to, an attacker-controlled server using query strings. CROSSFIRE provides a detailed exploit template for this vulnerability, and crafting the code shown below only requires a quick manual analysis to determine the types of arguments that should be passed to the method call.

```
// Attacker sets a global server $url
wot_api_comments.server = $url;
wot_api_comments.call("", "", {});
```

3) Download File: This is a vulnerability in *FlashGot Mass Downloader*, an extension that integrates various external download managers with Firefox, which allows an attacker to download a list of files. Unlike the other exploits that can send a GET request to a URL to achieve the same task, exploiting this vulnerability does not display Firefox's download prompt. Instead, the files are downloaded silently, in a completely transparent manner. This could be exploited, for instance, together with a file execution vulnerability to download and run malware. For the exploit code shown below, CROSSFIRE provides an exploit template that indicates the relationship between the object array passed to the vulnerable method and the sink it flows into, but the specific structure of the objects inside the array cannot be detected automatically and must be determined through manual analysis.

```
// Attacker creates an array of
// file $url and $path combinations
var files = [{
  href: $url,
  description: "",
  fname: $path,
  noRedir: true
  },
  // ...more files if needed...
];
gFlashGotService.download(files);
```

4) Execute File: Here, we present two vulnerabilities that could be used to execute binary files, the first one in *Firebug*, an extension that provides a set of web developer tools, and the other in *Greasemonkey*, which allows users to modify the displayed website content using custom JavaScript code.

The first vulnerability results from exploiting code that is normally used by Firebug for opening a file of interest in an external editor. By changing the editor's path, the attacker can control which file to execute.

```
// Attacker specifies the path to an
// executable as $exe and its command
// line arguments as $args
var malicious_exe = {};
malicious_exe.executable = $exe;
malicious_exe.cmdline = $args;

// The first argument needs to be a valid
// local file or directory on the system;
// a standard root  directory will do.
Firebug.ExternalEditors.open(
  "file:///C:/", malicious_exe);
```

The latter vulnerability, in Greasemonkey, is found in code that provides similar functionality for modifying script files in an external editor. However, this time the path to the external editor is set by changing a preference value.

```
// Attacker chooses a path $exe
var gPrefMan = new GM_PrefManager();
gPrefMan.setValue("editor", $exe);
GM_util.openInEditor();
```

D. Implementation

We implemented CROSSFIRE's static analyzer and exploit generator components in approximately 1.2K lines of JavaScript code. For JavaScript parsing and AST generation, we used a modified version of Esprima [1], a popular JavaScript parsing framework. Our modifications to Esprima serve to adapt the parser to Mozilla-specific JavaScript language extensions, and make the tool resilient to certain types of syntax errors in extension code that we encountered frequently during our experiments.

E. Limitations

The primary goals of this paper is to introduce and highlight extension-reuse vulnerabilities, quantify the prevalence of the problem among popular extensions, and demonstrate its impact on current extension vetting procedures. As such, the static analyzer component of CROSSFIRE we describe in this section is specifically tailored for discovering extension-reuse vulnerabilities as opposed to striving for a sound and precise analysis. In fact, the analysis we describe here is decidedly unsound: we do not attempt to tackle all of the well-known challenges of analyzing JavaScript programs, such as inferring dynamic types, handling prototype-based inheritance, resolving variable scopes, or handling string evaluation performed by `eval` or `setTimeout` statements.

While this lack of soundness and precision can be viewed as a deficiency, we argue that instead – in the spirit of "soundiness" [23] – it is a strength. Indeed, we explicitly trade off traditional goals of static program analysis in favor of efficiency, as our particular goal is oriented more towards best-effort discovery of extension-reuse vulnerabilities and less towards proving the absence of these vulnerabilities. We also point out that, as our evaluation demonstrates in

Section IV, our analysis successful identifies numerous real-world instances of extension-reuse vulnerabilities, and in many cases automatically generates a working proof-of-concept attack despite its inherent limitations.

IV. Evaluation

In this section, we survey the extension-reuse vulnerabilities we have discovered in the top 10 Firefox extensions, present an analysis of 323 extensions randomly sampled from the Firefox Add-ons extension repository, quantify the performance of CrossFire and the human effort required to write working extension-reuse exploits, and present a case study showing anecdotal evidence that extension-reuse vulnerabilities have practical impact.

A. Vulnerabilities in Top Extensions

As a first step in understanding the impact and prevalence of extension-reuse vulnerabilities, we ran CrossFire on the top 10 most downloaded Firefox extensions (excluding those that use the Jetpack framework). Furthermore, we investigated all of the reported vulnerabilities manually, and classified them as either true alerts or false positives. Detailed results of this analysis are presented in Table II.

These results indicate that 9 out of the top 10 Firefox extensions contain several examples of extension-reuse vulnerabilities, with only Adblock Plus being impervious to this attack. CrossFire was able to automatically generate at least one working exploit for five of the tested extensions, while we constructed many other working exploits through manual analysis with the help of CrossFire's exploit templates. (We discuss the human effort required for the manual analysis task later in Section IV-C.) Given that the extensions we tested have been downloaded millions of times according to the Mozilla Add-ons website, we surmise that a large number of Firefox users are affected by extension-reuse vulnerabilities.

One interesting observation we obtained through this experiment is the large number of global variables and function definitions in all of the tested extensions. Indeed, according to Mozilla, JavaScript namespace pollution is one of the most encountered issues during extension reviews [25]. This suggests that attempting to mitigate extension-reuse vulnerabilities through new guidelines for developers that discourage the use of globals, or through a more involved code review process for vetters to manually verify the secure use of globals, would not be effective solutions to the problem. This highlights once again that manual human analysis, while capable of discovering classes of vulnerabilities that elude the most sophisticated automated analysis, is nevertheless fallible. In particular, manual review simply cannot achieve the scale or consistency that sophisticated analyses promise.

Finally, we observed that the number of false positives, or non-exploitable vulnerabilities reported by CrossFire, varied with the tested extensions, ranging from 0% to 100% of the detected vulnerabilities. However, we stress that even when the false positive rates were high, the actual number of false vulnerabilities reported were small (e.g., four vulnerabilities were found for Adblock, yielding a 100% false positive rate), making their management and elimination via manual analysis an easy and quick task. We revisit the discussion of false positives on a larger dataset in the next section.

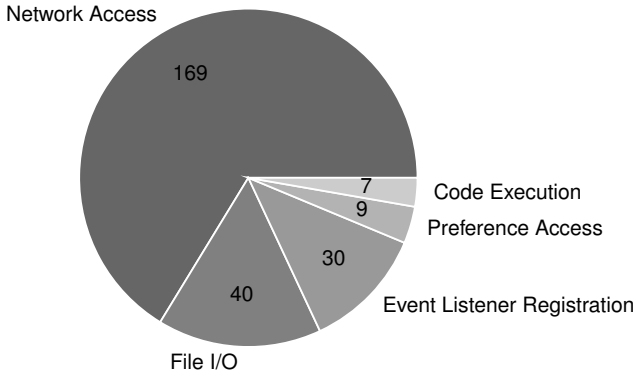

Figure 3. Breakdown of true positive vulnerabilities discovered by CrossFire by category.

B. Random Sample Study of Extensions

After our analysis of the top 10 extensions, in order to better understand how widespread extensions-reuse vulnerabilities are and how CrossFire performs in terms of false positives, we selected a random sample of extensions from those available on Mozilla Add-ons website and analyzed them.

In this experiment, we chose to limit our population to the top 2,000 Firefox extensions. This was due to our observation that as the extension popularity further decreased, we frequently encountered outdated extensions that were not compatible with modern versions of Firefox we used in our experiments. When choosing our sample size, we targeted a confidence interval of 5% at a 95% confidence level. According to the standard theory on confidence intervals for proportions (e.g., [15], Chapter 13), a sample size of 323 is sufficient to reach this target accuracy. Consequently, we selected a random sample consisting of 323 extensions for our experiment.

First, we ran CrossFire on our random sample, which yielded a total of 351 extension-reuse vulnerabilities. Next, we conducted a detailed manual analysis to identify vulnerability reports that represent false positives. The breakdown of our analysis is presented in Table IV, and the number of true positive vulnerabilities discovered in each category is shown in Figure 3.

We also ran CrossFire on the full dataset of the top 2,000 extensions. A summary of the statistical properties of the dataset is presented in Table III. Based on the estimated true positive rate of 72.65% we obtained in the random sample study within a 5% confidence interval, and given that CrossFire found 4,462 potential vulnerabilities in the full dataset, we estimate that more than **3018** of these vulnerabilities are exploitable on the lower bound of our confidence interval with 95% confidence.

We point out that while the obtained false positive rate of 27.35% seems relatively high, the actual number of false vulnerabilities reported per analyzed extension is low, and as is evidenced by the human analysis time estimates presented in the next section, they can be quickly filtered out.

Table II. DETAILED ANALYSIS RESULTS OF THE TOP 10 FIREFOX EXTENSIONS.

Extension Name	Globals		Sinks	Exploits			Attack Types
	Var.	Func.		Auto	Manual	False Pos.	
Adblock Plus	218	570	17	0 (0.0%)	0 (0.0%)	4 (100.0%)	–
Video DownloadHelper	46	707	74	0 (0.0%)	15 (100.0%)	0 (0.0%)	Code exec., File, Network
Firebug	71	378	40	0 (0.0%)	1 (100.0%)	0 (0.0%)	Code exec.
NoScript	40	1142	33	2 (22.2%)	5 (55.6%)	2 (22.2%)	Code exec., Network
DownThemAll!	53	632	14	0 (0.0%)	5 (100.0%)	0 (0.0%)	Network, Preference
Greasemonkey	121	362	17	1 (16.7%)	3 (50.0%)	2 (33.3%)	Code exec., File, Network
Web of Trust	56	601	275	1 (2.0%)	33 (67.4%)	15 (30.6%)	File, Network, Cookie
Flash Video Down.	50	123	79	4 (66.7%)	1 (16.7%)	1 (16.7%)	File, Network, Preference
FlashGot Mass Down.	36	555	53	3 (17.7%)	5 (29.4%)	9 (52.9%)	Code exec., File, Network
Down. YouTube Videos	2	22	6	0 (0.0%)	2 (66.7%)	1 (33.3%)	File, Preference

Table III. FIVE-NUMBER SUMMARIES, MEAN, AND TOTAL VALUES OF CROSSFIRE'S STATIC ANALYSIS RESULTS. THE EXPERIMENT IS PERFORMED ON THE TOP 2,000 FIREFOX EXTENSIONS.

Metric	Min	Q_1	Median	Mean	Q_3	Max	**Total**
Global Variables	0.00	1.00	2.00	11.32	9.00	422.00	22626
Global Functions	0.00	4.00	21.00	80.94	77.75	5460.00	161728
Sinks	0.00	0.00	2.00	6.33	7.00	278.00	12641
Vulnerabilities	0.00	0.00	0.00	2.23	2.00	238.00	4462

Table IV. SUMMARY OF THE TRUE AND FALSE POSITIVES DETECTED BY CROSSFIRE WHEN ANALYZING 323 EXTENSIONS RANDOMLY SAMPLED FROM TOP 2,000 FIREFOX EXTENSIONS.

Total Vulnerabilities	351	
True Positives	255	72.65%
Automated	51	14.53%
Manual	204	58.12%
False Positives	96	27.35%

C. Performance & Manual Effort

We characterize the overall performance of CROSSFIRE using two metrics: automatic vulnerability analysis runtime, and the manual effort required to construct working proof-of-concept exploits from exploit templates generated by CROSSFIRE.

For the first metric, we ran our system on our entire dataset of the top 2,000 Firefox extensions using a commodity desktop computer (3.40 GHz Intel Core i7-4770 CPU, 16 GB memory, running Ubuntu 14.0.1), and recorded the runtime required to analyze each extension. To quantify the human analysis time, during the random sample experiment described in Section IV-B, we timed our analyst's manual analysis sessions for each extension to obtain an estimate value. Each extension was analyzed by the same analyst, and our calculations for this metric exclude those extensions in our random sample for which no vulnerability was detected. The analyst performing the task was a graduate computer science student that had less than one year of experience with JavaScript programming and Firefox extension development.

Note that for our performance computations on the latter metric, we do not use the analysis timings obtained for each individual vulnerability. Instead, we estimate the analysis time for individual vulnerabilities in an extension by first measuring the total analysis time for that extension, and then computing its average over all working exploits found. We perform this operation due to our observation that manual analysis of the first reported vulnerability in an extension often takes significantly longer than investigating the rest. This is because the analyst spends extra time to understand the code during the initial analysis, and then performs the subsequent analyses much faster in light of this contextual knowledge. This results in a small number of long session durations, followed by a large number of very short analysis sessions. Instead of reporting biased results, we believe our estimation approach reflects the human analysis burden more accurately. The five-number summaries and arithmetic averages of these performance metrics are presented in Table V. The results for the automatic static analysis performance show that CROSSFIRE can analyze the majority of extensions in less than a second. Note that, here, the mean analysis time is much larger than the median, and there is a large gap between the third quartile (i.e., Q_3) and the maximum value. This is due to a small number of extensions in our dataset that contain unusually large code bases. For example, the extension *Local Load* [5], which allows its users to use local copies of common third-party JavaScript libraries instead of downloading them when loading web pages, contains the entire source code for several complex libraries and their various popular versions. As a result, our analysis of Local Load required 763.91 seconds, the maximum in our experiment. Despite these outliers, that CROSSFIRE performs efficiently with most extensions is further illustrated by the fact that the 95^{th} percentile in our measurements is 1.42 seconds, and the 99^{th} percentile is 6.80 seconds.

For the human analysis time measurements, the results indicate that, on average, a working proof-of-concept attack can be crafted in less then 10 minutes from the template generated by CROSSFIRE. In our experiments, the longest analysis session was still shorter than 40 minutes, which suggests that the human burden of working toward actual attacks from exploit templates is a manageable task even with a single analyst.

Table V. FIVE-NUMBER SUMMARIES AND THE MEAN TIME MEASUREMENTS OF AUTOMATIC STATIC ANALYSIS BY CROSSFIRE, AND HUMAN ANALYSIS TO CRAFT WORKING EXPLOITS FROM EXPLOIT TEMPLATES.

Performance Metric	Min	Q_1	Median	Mean	Q_3	Max
Static Analysis Runtime	0.05 sec	0.18 sec	0.28 sec	1.06 sec	0.51 sec	763.91 sec
Human Analysis Burden	0.50 min	3.20 min	4.50 min	6.31 min	9.18 min	36.00 min

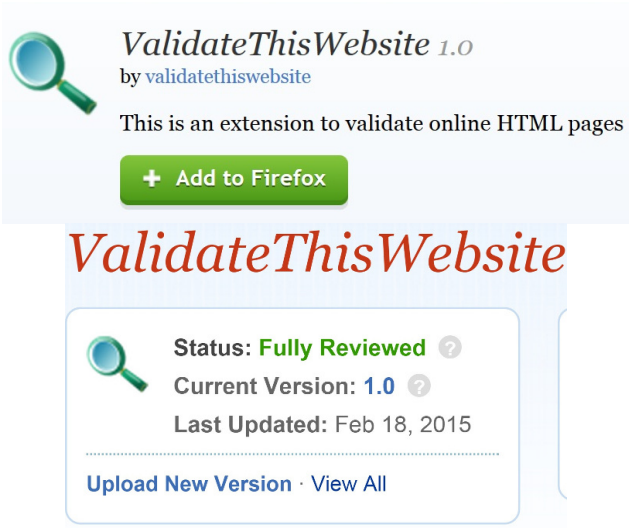

Figure 4. Screenshots from Mozilla Add-ons website showing the accepted extension and its fully reviewed status.

D. Case Study: Submitting an Extension to Mozilla Add-ons Repository

Due to the ethical issues surrounding the testing of extension-reuse exploits in a real-world setting, we were unable to conduct a detailed scientific study of the attack in practice. Instead, we opted to perform a simple case study to anecdotally demonstrate the attack's practical impact, and to encourage more rigorous future studies under controlled environments.

For this case study, we developed an extension called *ValidateThisWebsite*, which allowed users to automatically run a public markup validation service on the displayed web page when a button on the browser's toolbar is clicked. However, we also embedded in the code a cross-extension call to the popular script and plug-in blocker extension NoScript [14], which allowed our extension to stealthily connect to a URL of our choosing. This cross-extension call, made via the global variable `noscriptBM` defined in NoScript's source code, is presented below.

```
// Attacker chooses $url
noscriptBM.placesUtils.__ns.__global__.ns.
          loadErrorPage(window[1], $url);
```

Our extension consisted of approximately 50 lines of JavaScript code, and did not contain code obfuscation or any other attempt to hinder analysis. We submitted our extension to the Mozilla Add-ons repository, and opted for the *full review* option. This option represents the highest degree of scrutiny offered by Mozilla, and involves functional testing and human code reviews for security [25].

Our extension successfully passed the initial automated analysis upon submission, and subsequently passed the full review process without receiving any security warnings. We were notified of its acceptance to the online repository two days after its submission; see Figure 4 for screenshots of the listing on the Mozilla Add-ons website. We downloaded and tested the online version of our extension, and verified that the cross-extension call indeed works as intended. This case study, while only a single data point, serves as an existence proof that malicious extensions exploiting extension-reuse vulnerabilities can indeed pass the vetting process undetected, and that they pose a real threat to Firefox users.

Ethical Considerations. This case study was designed in the same vein as those presented in two recent prominent security research publications [33], [39].

We stress that the extension we developed did not actually contain a malicious URL, but instead a harmless link. Specifically, we registered the domain name "validatethis.website" for this case study, and set up our cross-extension call to open "http://validatethis.website/" which did not link to any content. Note that while harmless, this approach is still representative of an actual attack, because an attacker could use a similar strategy to first include an empty link in the extension, only to update the URL with malicious content after passing the vetting process.

We have never publicly advertised our extension, and we took it down from the repository promptly after receiving the acceptance notification. We did not record or otherwise track any activity on the sample domain that might have taken place during the vetting of our extension. Finally, we performed this case study only once to avoid unnecessarily burdening the extension reviewers.

V. DISCUSSION

In this section, we touch upon some of the interesting questions left open in this paper, and discuss possible future research directions.

A. Extension-Reuse Vulnerabilities in Jetpack Extensions

While this work primarily focused on the vulnerabilities in legacy Firefox extensions due to their popularity and prevalence, we must stress that Jetpack extensions are not immune to extension-reuse attacks. Here, we briefly discuss a variation of the attack that specifically targets Jetpack extensions, and report on the results of our preliminary experiments with them.

Jetpack extensions are developed and packaged as a collection of isolated modules. In order to include and reuse these modules in extension code, module authors are first required to explicitly *export* variables and functions defined in their modules. Later, extension developers can load these modules

into their code using the provided *require()* function, and freely access the exported interfaces. In this setting, exported interfaces are analogous to the global variables that could be exploited in legacy extensions. In other words, an attacker can analyze the modules included in a Jetpack extension to identify data flows from exported variables and functions into security-sensitive APIs, and reuse these capability leaks to craft extension-reuse attacks.

To verify the practicality of this attack, we modified CROSSFIRE to inspect the modules in Jetpack extensions and apply its static analysis to detect data flows from exported module interfaces into critical APIs. We analyzed the entire set of 1,028 Jetpack extensions hosted on the Mozilla Add-ons repository and found that 5 of those extensions contained a total of 8 vulnerabilities. We did not encounter false positives in this experiment, possibly due to the narrower attack surface. CROSSFIRE generated working attacks for six of the vulnerabilities automatically, while two required manual work assisted by exploit templates.

As an example, one of the vulnerable extensions is *Live Stream Notifier*, which shows a notification when a Twitch stream goes live. The concrete exploit shown below retrieves the contents of an attacker-specified file on disk.

```
// Attacker loads vulnerable module...
var utils = require("utils");

// ...and chooses a path $file
utils.getFileContents($file);
```

These findings demonstrate that Jetpack extensions are effective at narrowing down the attack surface by limiting the number of globally exposed interfaces. However, they are still not immune to extension-reuse vulnerabilities through explicitly-exported variables and functions, and developers must take care to prevent dangerous capability leaks.

B. Implications on Current Extension Vetting Procedures

Our finding that extension-reuse attacks are possible and pose a threat to Firefox users – even for Jetpack extensions, as described above – has direct implications for the current vetting scheme used for the Firefox browser. Naturally, we do not intend our work to be interpreted as an attack on the efforts of Firefox's cadre of extension vetters, who have an important and difficult job. However, since the vetting process is *the* fundamental defense against malicious extensions in the Firefox ecosystem, we believe it is imperative that (i) extension vetters be made aware of the dangers posed by extension-reuse vulnerabilities, and that (ii) tool support be made available to vetters to supplement the manual analyses and testing they perform. We are not the first to propose automated techniques for discovering extension vulnerabilities; we touch on this area of related work in Section VI. Nevertheless, our experiments demonstrate that current tooling is insufficient to handle this class of attack, and the techniques we propose can serve as a first step towards bolstering the vetting process to detect extension-reuse vulnerabilities.

C. Future Work

One important issue we omitted in this work is the likelihood that an attacker would be able to find a sufficient set of extension-reuse vulnerabilities to launch a desired attack on a target system. On one hand, given that many practical attacks are possible by exploiting only one or two vulnerabilities, and that nine of the top 10 extensions contain a large number of such vulnerabilities, we intuitively expect the possibility of a successful attack to be high in many cases. On the other hand, a scientific quantification of this issue would require a large-scale survey of Firefox users, and a detailed study of their extension usage behavior.

Another promising venue for future research is extension-reuse attack detection and mitigation techniques. Clearly, the highest assurance against such attacks would be possible by directly fixing the root cause of the issue, in other words by isolating the JavaScript contexts of Firefox extensions. However, the complexity and cost of such an intrusive change to the browser's extension architecture needs to be investigated further. Moreover, it is not clear whether the shared JavaScript namespace has any legitimate functionality, or if it is mandatory for the browser or certain extensions to work correctly.

A simpler, albeit less effective, detection or mitigation approach would be extending the existing solutions for browser extension analysis, verification, and runtime policy enforcement (e.g., those described later in Section VI) to detect cross-extension interactions, for instance by devising more accurate call site provenance techniques.

Finally, Mozilla announced on August 21, 2015 upcoming major changes to Firefox extensions, including the implementation of a new add-on API called WebExtensions [28]. Although details and security implications of these changes were not clear at the time of writing, we expect that a systematic security analysis of WebExtensions would be a promising future research direction.

VI. RELATED WORK

The security community has produced a large body of work investigating the security properties of browser extension mechanisms. Barth et al. [4] present a study of 25 Firefox extensions and point out that most of them have unnecessarily high privileges. Subsequently, they propose a security-hardened extension architecture for Chrome, designed around the principles of least privilege and privilege separation. Carlini et al. [6] and Liu et al. [21] further scrutinize Chrome's extension architecture, present additional security threats, and propose various countermeasures. Karim et al. [18] instead analyze Firefox's more recent Jetpack framework, identify modules with capability leaks and over-privileged extensions, and present a methodology to convert legacy Firefox extensions into Jetpack extensions. While these efforts direct their attention to desktop browsers, Marston et al. [24] focus on securing Firefox extensions on Android devices. Despite the abundance of research in the field, our paper represents the first work introducing and specifically addressing the problem of extensions-reuse vulnerabilities.

Another class of work proposes static and dynamic analysis techniques to identify security flaws in browser extensions.

Kapravelos et al. [16] describe Hulk, a dynamic analysis system that monitors extension activities through the use of fuzzing techniques and *HoneyPages* that adapt to extensions' expectations. They analyze more than 48K Chrome extensions and report on the malicious extensions they encountered. Bandhakavi et al. [2], [3] propose a static information flow analysis framework for JavaScript extensions called VEX, and analyze more than 2K Firefox extensions. Guha et al. [13] present IBEX, a framework that allows extension developers to create fine-grained access control and data-flow policies, and a static analysis methodology to verify these. Djeric et al. [8] and Dhawan et al. [7] propose dynamic analyses to track untrusted data inside the browser, and detect extensions that attempt to compromise the system's security. Similarly, Wang et al. [37] examine the behavior of Firefox extensions using an instrumented browser. Some of the analyses described in these papers could potentially be extended to detect extension-reuse vulnerabilities or malicious extensions that exploit the same. However, carrying out this task reliably would require incorporating the entire extension pool available to users into the analysis, which would almost certainly present problems of scalability and questions of coverage.

Other researchers have proposed execution monitors for runtime policy enforcement on browser extensions. Onarlioglu et al. [31] describe Sentinel, a lightweight XPCOM policy enforcer for JavaScript Firefox extensions. An extended version of this work [32] provides a partial and limited defense against extension-reuse attacks by protecting global variables against tampering; however, reuse of globally-exposed sensitive functions (e.g., attacks those described in Section III-C) remain unaddressed. Ter Louw et al. [34], [35] present an extension integrity checker and an XPCOM policy enforcement framework built into Firefox. As opposed to the previously mentioned work that offers a flexible policy framework, Wang et al. [38] propose an approach that targets two specific policies. Malicious extensions that exploit extension-reuse vulnerabilities would be able to bypass the defenses described in this class of work because, in our attack model, malicious extensions do not violate security policies but instead reuse functionality from legitimate extensions that are not subject to policy restrictions in a confused deputy-style attack. However, as before, policy enforcement systems could potentially be adapted to this new attack model through techniques that can determine the provenance of security-critical operations more accurately across different extensions.

Recent work by Karim et al. [17] presents a technique for transforming legacy Firefox extensions to use the Jetpack framework. As previously discussed, Jetpack extensions are not immune to extension-reuse vulnerabilities; however, techniques that allow for automatically porting legacy extensions to modern extension frameworks could potentially reduce exploitable capability leaks.

Freeman and Liverani [9], [22] have released two whitepapers that describe *Cross Context Scripting (XCS)* vulnerabilities, and demonstrate attack scenarios targeting Firefox. XCS constitutes a distinct class of attacks that deal with executing untrusted content retrieved from web pages inside the browser's trusted zone, and is not addressed in our paper.

Earlier work on web browsers mostly focused on securing native plug-ins and third-party applications that run within browsers, such as Adobe Flash player. For example, Li et al. [20] and Kirda et al. [19] present techniques to contain spyware-like behavior in Internet Explorer's *Browser Helper Objects*. Other work [10], [12], [36], [40] provides secure execution environments inside browsers through sandboxing and isolation concepts borrowed from the field of operating systems research. This line of work targets a different problem from the setting of browser extension security addressed in this paper.

VII. Conclusions

In this paper, we introduced a novel class of attacks stemming from *extension-reuse* vulnerabilities, which arises from the lack of isolation between Firefox extensions, and results in capability leaks through global identifiers defined in the shared JavaScript namespace of the browser. We then presented CrossFire, a lightweight static analysis tool that can quickly analyze a large pool of extensions, automatically detect extension-reuse vulnerabilities they contain, and, finally, generate proof-of-concept exploits and exploit templates that can be used for rapid exploit construction by a human analyst to validate reported vulnerabilities. We also experimented with CrossFire in order to characterize its false positive rate due to the inherent limitations of our static analysis, as well as the human effort required to eliminate false vulnerability reports and produce working exploit code from exploit templates. Our results indicate that, on average, a single human analyst can produce an exploit under 10 minutes and, despite a relatively high false positive rate, the absolute false positive numbers remain low and manageable.

Our detailed analysis of the top 10 extensions, a random sample study of the top 2,000 extensions, and a case study demonstrating the difficulty of manually identifying extension-reuse exploits all support our claim that extension-reuse vulnerabilities are real, practical, and are present in large numbers in popular extensions downloaded by millions of users. In addition, our experiments with vulnerable Jetpack extension show that, even though Jetpack extensions have a narrower attack surface compared to legacy extensions, they are not immune to extension-reuse attacks.

Acknowledgments

This work was supported by the Office of Naval Research (ONR) under grant N000141310102, National Science Foundation (NSF) under grant CNS-1116777, and Secure Business Austria.

References

[1] Ariya Hidayat, "Esprima," http://esprima.org/.

[2] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett, "VEX: Vetting Browser Extensions for Security Vulnerabilities," in *Proceedings of the USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2010.

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

[4] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, "Protecting Browsers from Extension Vulnerabilities," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2010.

[5] Brian LePore, "Local Load," http://www.getlocalload.com/.

[6] N. Carlini, A. P. Felt, and D. Wagner, "An Evaluation of the Google Chrome Extension Security Architecture," in *Proceedings of the USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2012.

[7] M. Dhawan and V. Ganapathy, "Analyzing Information Flow in JavaScript-Based Browser Extensions," in *Proceedings of the Annual Computer Security Applications Conference*, 2009.

[8] V. Djeric and A. Goel, "Securing Script-Based Extensibility in Web Browsers," in *Proceedings of the USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2010.

[9] N. Freeman and R. S. Liverani, "Exploiting Cross Context Scripting Vulnerabilities in Firefox," http://www.security-assessment.com/files/whitepapers/Exploiting_Cross_Context_Scripting_vulnerabilities_in_Firefox.pdf, 2010.

[10] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker," in *Proceedings of the USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 1996.

[11] Google, "Chrome Web Store," https://chrome.google.com/webstore/category/extensions.

[12] C. Grier, S. Tang, and S. T. King, "Secure Web Browsing with the OP Web Browser," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2008.

[13] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy, "Verified Security for Browser Extensions," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2011.

[14] InformAction, "NoScript," http://noscript.net/.

[15] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, Apr. 1991.

[16] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, "Hulk: Eliciting Malicious Behavior in Browser Extensions," in *Proceedings of the USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2014.

[17] R. Karim, M. Dhawan, and V. Ganapathy, "Retargetting Legacy Browser Extensions to Modern Extension Frameworks," in *Proceedings of the European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer, 2014.

[18] R. Karim, M. Dhawan, V. Ganapathy, and C.-c. Shan, "An Analysis of the Mozilla Jetpack Extension Framework," in *Proceedings of the European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer, 2012.

[19] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer, "Behavior-Based Spyware Detection," in *Proceedings of the USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2006.

[20] Z. Li, X. Wang, and J. Y. Choi, "SpyShield: Preserving Privacy from Spy Add-ons," in *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*. Berlin, Heidelberg: Springer, 2007.

[21] L. Liu, X. Zhang, G. Yan, and S. Chen, "Chrome Extensions: Threat Analysis and Countermeasures," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2012.

[22] R. S. Liverani, "Cross Context Scripting with Firefox," http://www.security-assessment.com/files/whitepapers/Cross_Context_Scripting_with_Firefox.pdf, 2010.

[23] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhotak, J. N. Amaral, B.-Y. E. Chang, S. Guyer, U. Khedker, A. Moller, and D. Vardoulakis, "In Defense of Soundiness: A Manifesto," *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, Jan. 2015.

[24] J. Marston, K. Weldemariam, and M. Zulkernine, "On Evaluating and Securing Firefox for Android Browser Extensions," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*. New York, NY, USA: ACM, 2014.

[25] Mozilla, "Add-on Documentation - Review Process," https://addons.mozilla.org/en-US/developers/docs/policies/reviews.

[26] ——, "Add-ons for Firefox," https://addons.mozilla.org/.

[27] Mozilla Add-ons Blog, "Firefox Extensions: Global Namespace Pollution," http://blog.mozilla.org/addons/2009/01/16/firefox-extensions-global-namespace-pollution/, 2009.

[28] ——, "The Future of Developing Firefox Add-ons," http://blog.mozilla.org/addons/2015/08/21/the-future-of-developing-firefox-add-ons/, 2015.

[29] Mozilla Developer Network, "XPCOM," https://developer.mozilla.org/en-US/docs/XPCOM.

[30] Mozilla Wiki, "Jetpack," https://wiki.mozilla.org/Jetpack.

[31] K. Onarlioglu, M. Battal, W. Robertson, and E. Kirda, "Securing Legacy Firefox Extensions with Sentinel," in *Conference on Detection of Intrusions and Malware & Vulnerability Assessment*. Springer, Jul. 2013.

[32] K. Onarlioglu, A. S. Buyukkayhan, W. Robertson, and E. Kirda, "Sentinel: Securing Legacy Firefox Extensions," *Computers & Security*, vol. 49, pp. 147–161, Mar. 2015.

[33] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2014.

[34] M. Ter Louw, J. S. Lim, and V. N. Venkatakrishnan, "Extensible Web Browser Security," in *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment*. Berlin, Heidelberg: Springer, 2007.

[35] ——, "Enhancing Web Browser Security against Malware Extensions," in *Journal in Computer Virology*. Springer-Verlag, 2008, vol. 4, pp. 179–195.

[36] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter, "The Multi-Principal OS Construction of the Gazelle Web Browser," in *Proceedings of the USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2009.

[37] 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 Information Security Conference*. Berlin, Heidelberg: Springer, 2012.

[38] L. Wang, J. Xiang, J. Jing, and L. Zhang, "Towards Fine-Grained Access Control on Browser Extensions," in *Proceedings of the International Conference on Information Security Practice and Experience*. Berlin, Heidelberg: Springer, 2012.

[39] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee, "Jekyll on iOS: When Benign Apps Become Evil," in *Proceedings of the USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2013.

[40] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2009.