An Empirical Study of Dangerous Behaviors in Firefox Extensions

Jiangang Wang[1], Xiaohong Li[⋆,1], Xuhui Liu[2], Xinshu Dong[2], Junjie Wang[1], Zhenkai Liang[2], and Zhiyong Feng[1]

[1] Department of Computer Science, Tianjin University
xiaohongli@tju.edu.cn
[2] School of Computing, National University of Singapore

Abstract. Browser extensions provide additional functionality and customization to browsers. To support such functionality, extensions interact with browsers through a set of APIs of different privilege levels. As shown in previous studies, browser extensions are often granted more privileges than necessary. Extensions can directly threaten the host system as well as web applications, or bring in indirect threats to web sessions by injecting contents into web pages. In this paper, we make an empirical study to analyze extension behaviors, especially the behaviors that affect web sessions. We developed a dynamic technique to track the behaviors of injected scripts and analyzed the impact of these scripts. We analyzed the behaviors of 2465 extensions and discussed their security implications. We also proposed a solution to mitigate indirect threats to web sessions.

1 Introduction

Browser extensions are widely adopted by modern web browsers to allow users to customize their web browsers. Browser extensions can change display of web pages, improve browsers' behaviors, and introduce new features. To support such functionalities, extensions need to monitor and modify web contents, change browser behaviors and appearances, access stored website passwords, cookies, etc. They can issue HTTP requests on behalf of users, and open sockets to listen to connections or connect to remote servers. They can even access the local file system and launch processes. On one hand, all these privileges enable extensions to extend the functionality of browsers; on the other hand, they increase the attack surface of users' systems.

Threats from extensions can be categorized into two types: *direct threats* and *indirect threats*. *Direct threats* arise from extensions' direct access to critical browser resources or the local system. Extensions can read and write Document Object Module (DOM) of any web page by which page contents and user keystrokes are visible to extensions. By inserting new elements to DOM, extensions can easily push contents to users at the extension developer's will. They also can access browser components directly, such as Firefox's password manager. Extensions can also access network and access local files with current system user's privileges.

Indirect threats are threats to web sessions. By injecting code into a page, browser extensions can easily take full control of a web session. Although this is an intended

⋆ Corresponding author.

D. Gollmann and F.C. Freiling (Eds.): ISC 2012, LNCS 7483, pp. 188–203, 2012.

functionality of browser extensions, malicious extensions can take advantage of it to carry out attacks against web sessions.

To limit extensions' high privileges in browsers, several solutions have been developed. Ter Louw et al. proposed a solution [1, 2] to enhance browser extension security by protecting *the integrity of browser code base* and *the confidentiality and integrity of user data*. It provides an isolation mechanism to prevent one malicious extension from compromising another. Google Chrome has adopted an extension system designed with *least privilege, privilege separation*, and *strong isolation* [3]. Each extension is granted a set of privileges at installation time and it cannot exceed the granted privileges during execution time. A recent proposal [4] further divides privileges of Chrome extensions into micro-privileges, to restrict extensions' capabilities in cross-site requests and DOM element accesses. It also introduces resources requesting new origins. To mitigate threats from over-privileged extensions, Mozilla has developed a new extension development framework Jetpack [5] to assist developers in building extensions following the principle of least authority (POLA).

To gain a better understanding of dangerous behaviors in Firefox extensions, we performed an empirical study of real-world Firefox extensions. Instead of looking for vulnerabilities in browser extensions [6], our focus is on dangerous runtime behaviors of browser extensions. We specifically study indirect threats to web sessions, and discuss what are missing from existing solution in dealing with them.

We developed an automatic testing system based on instrumented Firefox to dynamically investigate extension behaviors. Using this system, we studied Firefox extensions hosted in the Mozilla Addons repository [7]. In total, 2465 extensions had been tested, which were distributed in 13 different categories. We summarize our findings of dangerous behaviors in Firefox extensions, and discuss potential improvements in mitigating indirect threats while maintaining usability.

This paper makes the following contributions:

1. We perform an empirical study of Firefox extension behaviors by monitoring their runtime behaviors, with a focus on indirect threats. From our analysis, we propose improvement to current solutions in mitigating indirect extension threats to web sessions.
2. We designed and implemented an automatic testing system to monitor the browser extension behaviors.

2 Threats from Extension

2.1 Browser Extensions

Browser extensions are tools to extend browsers to enhance their functionality. To obtain highly customizable features, extensions are granted high privileges. They can access almost all components in a browser. The extension privileges can be summarized as follows:

– Monitoring and modifying web contents. All contents displayed in web page are visible to extensions including user inputs. Extension can modify the page by modifying DOM.

- Accessing browser components. Extensions can access browsers' password managers, cookie files, etc.
- Accessing network. Extensions also have full access to network and sockets. It can easily issue HTTP requests.
- Accessing local file systems. Extensions can access local file system with the privileges of the browser process.
- Launching processes. Extensions can launch processes through certain browser API.

As discussed, extensions have powerful privileges. In the following subsections, we will discuss what a malicious extension can do with these powerful privileges. In the rest of the paper, we focus on Firefox extensions.

2.2 Direct Threats

Direct threats are threats from browser extensions, through directly accessing critical resources of browsers or host systems.

- **Direct DOM Access.** Extensions can access all DOM structures in any page, including contents browsed by users, contents in all forms filled and submitted by users, and events on keystrokes and mouse clicks. They can also access all information submitted to the server, including account names and passwords, personal information, finance information, etc. They can not only read DOM, but also modify DOM to create new page contents.
- **Browser Component Access.** Extensions are capable to access all Firefox components, including the password manager and the browsing history, etc. Consequently, they can easily collect account information, such as passwords stored in the password manager and the browsing history.
- **Arbitrary File Access.** Extensions have the ability to access local file systems without any restriction. Once a malicious extension is installed, the whole system is controlled by attackers with the browser user's privilege. Since extensions can access arbitrary files and the extension system does not provide a mechanism to protect the integrity of installed extensions, it is possible for a malicious extension to change other benign extensions' behaviors by modifying their installed files.
- **Network Access.** Extensions have variable ways to access the network, such as sending XMLHttpRequest directly, or requesting resources like images or scripts from arbitrary servers. By attaching information to requests, malicious extensions can send out collected information.
- **Launching Process.** Launching a local process is considered highly dangerous, because by running malicious code, an attacker can take full control of victims' computers and make them part of a botnet.
- **Dynamic Code Execution.** Executing dynamic code is an important feature in the JavaScript language, which enables the program to execute dynamically generated strings as JavaScript code. The strings may originate from an attacker, allowing them execute malicious code.
- **Listening to Keyboard Events.** Listening to keyboard events is common in extensions, but an attacker can use this feature to steal users' inputs.

The root cause of direct threats is the high privileges that a browser grants to extensions. Different browser vendors proposed different mechanisms to mitigate these threats. Firefox browser adopts a Sandbox Review System to force all submitted extensions to be manually reviewed before they are released to the public. Google Chrome adopts a new extension system [3] with "least privileges" principle. The new system requires developers to claim the minimal privileges their extensions need. When an extension is installed, a prompt dialog pops up to warn the users what privileges the extension claims, whether to continue installing or not is determined by users. At run time, an extension cannot exceed privileges it claimed.

2.3 Indirect Threats to Web Sessions

Extensions introduce indirect behaviors through two steps. It first injects contents (script code or HTML elements) into web sessions; The injected contents then result in subsequent behaviors, which may launch attacks against the web sessions. Extensions have various ways to inject contents to web pages. They can call the $write$ or $writeln$ methods of DOM objects, $appendChild$ or $insertBefore$ methods of HTMLElement objects.

Unfortunately, current browsers lack the abilities to distinguish injected contents from original ones. The indirect threats completely bypass permission checking on the extension itself, and can launch attacks to web sessions. These attacks can be launched in all pages, regardless they are vulnerable or not.

Let us use a scenario to demonstrate how it can be exploited by attacks. An extension claims for privileges to access the browsing history and DOM, but it does not require the privilege to access the network. It seems that this claim should be allowed because although the extension may access users' private data, it cannot send them to untrusted third party through network APIs. However, direct access to network has been limited, but this extension can still access network in an indirect way. The extension can modify the DOM tree of benign web pages and inject <script>, <img> or <iframe> tags into pages, which use a malicious site as its source, and attach users' private data or cookies of target web sites as parameters. Such indirect threats have not been well studied by existing research work, and are also not constrained by existing browser extension systems.

All the behaviors conducted by extensions are within the context of browsers. So, it's almost impossible for traditional general protection mechanisms (e.g., anti-virus software or firewall) to judge certain behaviors are conducted by extensions or browsers themselves, let alone to tell these behaviors are benign or not. The protection mechanism again malicious extensions can only be deployed within browsers.

3 Design and Implementation

In this section, we present the design and implementation of our extension testing system that can automatically download, install, and test browser extensions and obtain their behavior information. We first illustrate the architecture of the system, then introduce the design and implementation of modules in our system in detail, and finally explain the testing process of browser extensions.

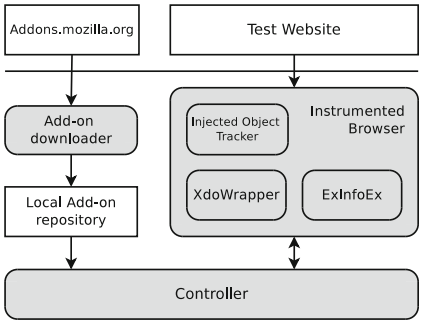

Fig. 1. System architecture of our Firefox extension testing system. The main components are in grey color.

Although our approach and study are based on Firefox, our approach does not depend on specific feature of Firefox, and our study is performed assuming the availability of protection mechanism on all browsers. Our solution and study can be straightforwardly extended to other browsers.

3.1 System Architecture

The system architecture is shown in Figure 3.1. There are six modules in our extension testing system. **Instrumented Browser** monitors the execution of browser extensions. It contains three sub-modules, **Injected Object Tracker** is a module to track whether an HTML element or a piece of JavaScript is originated from web page or injected by extensions; **ExInfoEx** is used for extracting the position information of items (such as menu items) added by tested extension and offering them to simulating module for triggering the events; **XdoWrapper** is a wrapper of Xdotool [8] that makes it easy to simulate many kinds of events at the window level. **Addon-Downloader** is responsible for downloading all the analyzable browser extensions to the local system. **Controller** is a module that connects other modules together and makes the whole test process automatic. We also created a test website to drive the testing of extensions.

3.2 Design and Implementation

We give the detailed introduction about the design and implementation of the major components in our system.

Instrumented Browser. In order to investigate the tested extension's behaviors, we intercept the interfaces or functions that are accessed by the tested extension at runtime using an instrumented Mozilla Firefox 3.5.

In this customized browser, for detecting the direct threats of an extension, we inserted a series of hooks into the browser's source code. We mainly hook four modules of Firefox: the XPConnect module, the DOM module, the Content module, and the

JavaScript engine. XPCOM makes it possible for JavaScript to invoke methods provided by the browser. The XPConnect module is responsible for the communication between JavaScript and XPCOM components. By inserting hooks into this module, we can intercept the events of browser extension accessing XPCOM interfaces, and extract the parameters. In the DOM module and the Content module, we inserted hooks to intercept event listener registration and removal issued by a browser extension. SpiderMonkey [9], the JavaScript engine in Firefox, is also instrumented to intercept several security-related JavaScript native functions for analysis.

The instrumented browser contains three sub-modules:

Injected Object Tracker. To investigate the indirect behaviors of an extension, it is necessary to track the contents injected into web pages by extensions and monitor their subsequent behaviors. For this purpose, we design the Injected Object Tracker to track the source of injected contents. We intercept interfaces used by extensions to inject new contents to web pages. Once extensions inject any object into web pages, the injected object is marked with the source "extension". During behavior monitoring, we can find out whether the behavior is from the original web page or from objects injected by extensions.

ExInfoEx. ExInfoEx (Extension Information Extractor) is a browser extension that identifies the browser user interface elements that are added by the tested extension and transfer the position information of these elements to XdoWrapper module to trigger the events targeted on them. However, due to the technology of "overlay", how to distinguish the elements added by an extension from the elements belonging to browser itself becomes the biggest challenge we faced.

To resolve this problem, we modify the process of overlay loading. When the browser creates elements defined in overlays in the process of loading, an extra attribute is added to each element. The value of this attribute is set to the extension's identifier that is still retained by the browser at this time and can be easily extracted. With this attribute, ExInfoEx can identify those elements by traversing the DOM tree of the browser user interface.

XdoWrapper. For exposing as many behaviors of a browser extension as possible, we need to simulate users' behaviors to trigger the events in web pages and in the chrome area. XdoWrapper is such a module that receives specific instructions from the ExInfoEx module to simulate these users' behaviors, and provides the relevant information about the windows of browser to the Controller module. It is a wrapper of xdotool [8], which can conveniently generate various keyboard and mouse events at the window level and easily manage and manipulate all the windows opened in the operating system.

Controller. The Controller module is used to connect other modules together and coordinate the entire testing process. Specifically, it is mainly responsible for the following work:

- Install/uninstall extensions. Controller is responsible for the installation and uninstallation of each extension.

- Start/shutdown the browser. During the whole test process, there are many cases need to start or shutdown the browser.
- Configure/clean the testing environment. Before or after an extension is tested, the testing environment needs to be configured with the corresponding information or cleaned up by the controller module.
- Coordinate the whole process. All these processes (install, configure, test, and uninstall) should be connected together smoothly, and this is one of the main tasks of Controller.
- Handling exceptions. Controller also handles unexpected issues, such as browser crashes.

Addon Downloader. The Addon Downloader module is responsible for downloading all the testable browser extensions in the online repository. We utilize Htmlcxx [10] (a lightweight HTML and CSS parser for C++) to parse an extension's page and determine whether the extension is testable, and then with the help of Libcurl [11](a client-side file transfer library), download it to our system.

3.3 Testing Process

We first download all the testable browser extensions in the repository. Each extension downloaded to the local system is then installed on the instrumented browser for testing. After installation, essential information about the extension, such as name, ID (if applicable) and installation path, is extracted from the browser profile and recorded into a configuration file, which is used by the browser in the phase of monitoring extension's behaviors. We then restart the browser to begin the behavior monitoring process. At the beginning of restart, the browser reads the information about the tested extension from the configuration file. This information is indispensable for browsers to detect the behavior information of the tested extension. Once the browser obtains this information during restart, the behavior monitoring process is automatically started. For exposing more behaviors of the tested extension, the system first leads browser to visit a particular website designed by ourselves and simulates a variety of events in web pages, such as click on links, keyboard input, and form submission. Then, the system goes through the browser interface, detects the elements (such as menu items, context menu items and status bar items) added by the tested extension, and generates events corresponding to clicks on each these elements. If new windows pop up after clicks, they are simply closed. After that, the browser is shut down, and the tested extension is removed. Relevant configuration information related to this extension is also cleaned up. After all the above procedures are over, the testing process for one extension is completed, and the process for next extension can start.

4 Evaluation and Analysis

The experiment was conducted on a computer with Intel(R) Core(TM)2 Duo CPU at 2.33GH, 250GB 7200RPM disk, and 4GB RAM. Its operating system is Ubuntu 10.04, with the instrumented Firefox 3.5 installed.

Table 1. The ratings of extension behaviors

Rating	Behaviors
high	Arbitrary file access; Process launching; Download; XPinstall; Network access via XPCOM APIs; Update; DOM injection
medium	Password; Login; Cookie; Network access via XMLHttpRequest; Addons management; Changing Firefox preferences; Profile
low	History; Bookmark; Clipboard; Dynamic code execution
none	Accessibility; Browser core; Auto complete; Log to console; Searching; Spell checking; DOM; Editor; Internationalization; Offline cache; XML parser; Network utilities; RSS/RDF; Data types and structures; Streams; Memory management; Thread management; Component management; Additional XPCOM services; JavaScript core; JavaScript debugger; XPConnect; Authentication; Certification; Cryptograph; Additional security interfaces; Document handling; Transaction management; Web worker; Window management; Print; Database access; Images; Zip/jar process; JVM; Plugins

We first investigated the behaviors of Firefox extensions by analyzing the critical XPCOM interfaces and JavaScript functions exposed to extensions, and classified them into different levels according to their potential risks. Then we further studied Firefox extension behaviors by testing 2465 extensions hosted on the Mozilla Addons website [7] using our extension testing system, and analyzed the experimental data from various perspectives.

4.1 Studying and Classifying Extension Behaviors

Through the analysis of critical XPCOM interfaces and JavaScript functions used by Firefox extensions, we summarize them into 54 different behaviors. To investigate extension behaviors from a security perspective, all these behaviors are rated into four levels according to their potential risks. This work is mainly based on Firefox security severity ratings [12], Chromium's severity guidelines for security issues [13] and the work of Barth et al [3]. The ratings are described in Table 1.

Rating *High* includes the behaviors that can possibly download, install, or execute a program. These behaviors are considered the most privileged and dangerous ones, because they could be utilized by an attacker to compromise users' entire operating system. "DOM injection" is also classified into *High* because it can inject DOM contents into web applications that in turn initiate network access. Rating *Medium* consists of behaviors that may access users' private information (e.g., the password or cookie), or access network in a relatively safe way (like through XMLHttpRequest). The behaviors that may modify the critical data of the browser are also rated as this level. The behaviors in Rating *Low* are those that may obtain limited information of users, such as browsing history or bookmark, and those that are likely to cause a vulnerability with low possibility, like executing a dynamically generated string. All other non-sensitive behaviors are classified into Rating *None*. These ratings are used to investigate how many browser extensions exhibit high privileged behaviors and how the distribution of the usage of different behaviors looks like.

The Statistics of Extension Behaviors. To better understand the behaviors of Firefox extensions, we performed a study of statistics on their behaviors according to their ratings. We four that 33% of them had the behaviors belonging to Rating High. The extensions with the behaviors of Rating Medium accounted for nearly 16% of the total and only 1% of the tested extensions possessed the behaviors of Rating Low. A large number of the tested extensions, approximately 50% of them, only demonstrated the behaviors with Rating None, which are invariably benign to users. Note that one extension with higher privileged behaviors does not indicate that it will definitely cause a damage, but means that the extension has such capabilities that, if abused, will bring severe threats to users' data and/or the underlying operating system. We have known that Firefox provides the browser extensions with excessive privileges, and from our experiments we found that the extensions on Mozilla indeed utilize these high privileges widely.

To investigate the extensions' behaviors in detail from a security perspective, Table 2 lists the frequency of security-related behaviors of Ratings *High*, *Medium* and *Low*. In Rating *High*, there were 37 (1.50%) extensions found to access files outside their installation directory. Four (0.16%) extensions launched a process on the local system, and 10 (0.41%) use the download API provided by XPCOM. None of the extensions were found to install other addons through XPInstall API. The most widely used interfaces in this group are XPCOM interfaces for network accessing, accounting for 29.66% of the entire test set. Only one extension in our experiment uses the Update system. Additionally, there were 108 extensions (4.38%) found to inject new contents into web pages.

In Rating *Medium*, we found no extensions that access password information, while the login and cookie information were found to be accessed by some extensions, accounting for 1.78% and 2.80%, of the total number of extensions. There were 441 (17.89%) extensions issuing HTTP requests through the XMLHttpRequest object. The number of extensions that used XPCOM APIs to manage addons (e.g., to search and install addon from the repository or to locate the installation location) were 139 (5.64%). Although changing Firefox preferences may greatly annoy users, there were still 31 (1.26%) extensions doing so. In addition, one extension was observed to access the profile information of the Firefox.

In Rating *Low*, there were 43 (1.74%) and 44 (1.78%) extensions found to access the limited information, the browsing history and bookmarks. 49 of the tested extensions, about 1.99%, were found to manipulate the clipboard through XPCOM APIs. To our surprise, although Mozilla has highly recommended not to execute dynamic generated strings with "eval()" or "Function()", and several safer alternatives are also available, there were still 145 extensions showing such behaviors, accounting for 5.88% of the total number of extensions examined.

4.2 Extension's Indirect Threats to Web Sessions

To investigate the indirect threats from browser extensions, our experiment paid special attentions to *indirect behaviors* of extensions that inject new web contents into web pages. As shown in Table 3, there were 108 browser extensions that inject new contents

Table 2. Frequency of security-related behaviors

Rating	Behavior	Quantity	Frequency
High	Arbitrary file access	37	1.50%
	Process launching	4	0.16%
	Download	10	0.41%
	XPInstall	0	0.00%
	Network access via XPCOM API	731	29.66%
	Update	1	0.04%
	DOM injection	108	4.38%
Medium	Password	0	0.00%
	Login	44	1.78%
	Cookie	69	2.80%
	Network access via XMLHttpRequest	441	17.89%
	Addons management	139	5.64%
	Changing Firefox preferences	31	1.26%
	Profile	1	0.04%
Low	History	43	1.74%
	Bookmark	44	1.78%
	Clipboard	49	1.99%
	Dynamic code execution	145	5.88%

Table 3. Statistics of contents injected into webpages

Injected Element	Quantity	Total	Ratio
script	14	108	12.96%
iframe	8	108	7.41%
a	8	108	7.41%
img	14	108	12.96%
object/embed	3	108	2.78%
others	95	108	87.96%

into web pages, among which 14 extensions inserted “script” tags into pages, eight extensions inserted “iframe” tags into pages, and other 14 extensions inserted “img” tags. There were some other elements that were also found to be injected into pages by extensions, such as “div”, “span” and so on. We focus our analysis on those cases that extensions injected elements that are frequently exploited in attacks, including script, iframe and img tags. By manually examining these cases, we conclude that all the extensions that inject new contents into web pages, were exempted from being malicious.

For the 14 extensions that injected “script” elements into web pages, six of them directly injected JavaScript code into pages, while the remaining eight extensions injected “script” elements that request external JavaScript from the developers’ websites or third-party ones.

For the six extensions that directly injected code, the injected code’s behaviors were rather simple. Three of them simply set certain JavaScript objects’ values to 1, as shown

```
<SCRIPT type="text/javascript">
window.script1309754027588=1;
</SCRIPT>
```

Fig. 2. JavaScript code to assign an object

```
<SCRIPT id="afterthedeadline−dispatchDisable">
  if (window.setTimeout)
    window.setTimeout(function(){
      if ("AtD" in window || "AtDCore" in window) {
        document.AtDdisabled = true;
        var ev = document.createEvent("HTMLEvents");
        ev.initEvent("disableAtD",true,false);
        document.dispatchEvent(ev);
      }
  },100);
</SCRIPT>
```

Fig. 3. JavaScript code to disable extension

in Figure 2; two of them injected empty code; and one of them set a timer event to disable itself, as shown in Figure 3. We also closely examined the eight extensions that injected "script" elements with *src* attributes to refer external JavaScript. We found that one of them accesses the cookie that was believed to belong to its own website. In another case, the injected script called the insecure function "eval" to load dynamic code. However, we have confirmed that this case is also benign.

In the eight extensions that injected "iframe" HTML elements into web pages, the requested pages loaded by iframes all came from the corresponding extension's official website. For the behaviors of these injected pages in iframes, we also find two pages that access the cookie of their own.

Other injection behaviors are also innocuous. All the "a" HTML elements injected into web pages by the eight extensions do not link to other pages. The 14 extensions inject "img" HTML element into web pages, using an image either from their chrome areas or from other sites as the source of "img" tag. All these images used as normal were icons and did not contain malicious event handlers. The three extensions that injected "object" or "embed" HTML elements are to include flash files to web pages.

From the experimental results, we found that subsequent behaviors of web contents injected into web sessions by extensions can be summarized in three groups.

- Network Access. Via $script$, img, $iframe$ or $embedded$ tags.
- Cookie Access. Via $script$ tags.
- Others. Certain injected code assigns simple values to certain objects, or register event listeners. None of these is security related.

Table 4. Statistics of some dangerous functionalities and practices

Dangerous Practice	Quantity	Total	Ratio
Accessing files beyond the directory of its own.	37	2465	1.50%
Launching a process.	4	2465	0.16%
Issuing HTTP requests via XMLHttpRequest.	441	2465	17.89%
Changing the preferences of the browser.	31	2465	1.26%
Capability of dynamic code execution(evel, Function).	145	2465	5.88%
Listening to the keyboard events targeted at web pages.	19	2465	0.77%
Preference names without "extensions." prefix.	266	2465	10.79%

4.3 Direct Threats

Dangerous Functionalities and Practices. In our experimental analysis, we also focused on dangerous functionality and practices. Table 4 lists the statistical results of our experiment.

Accessing files beyond the directory of its own. Browser extensions may have legitimate needs to access local files, so we focused on accesses to files outside the extensions' installation directories by monitoring the invocations to the "initWithPath" function of the "nsILocalFile" interface and its parameter. Among the 37 extensions with such behaviors, 15 extensions access files from "/home/username", and surprisingly there are as many as 7 extensions found to access files from "/usr/bin", 5 accessing "/tmp", and 2 accessing "/bin". After manual inspection, all of them were confirmed to be not malicious.

Launching a process. We monitor process launching by intercepting the "run" method of the "nsIProcess" interface. In these four extensions that launch new processes, two of them (KidZui and KidZui K2) executed the program "/usr/bin/xmodmap" to modify keymaps and pointer button mappings in the X system. Another extension (Zotero) executed the program "/usr/bin/mkfifo" to create a named pipe. Moreover, the extension named "TTS for linux" executed a shell script, which is responsible for converting text to speech by calling KTTSD (Kde Text To Speech Daemon) via DCOP (Desktop COmmunication Protocol).

Issuing HTTP requests via XMLHttpRequest. We found 33 extensions sending current page URLs to remote servers using XMLHttpRequest, which might be due to the fact that browsers do not append the `referrer` header to the HTTP request generated by extensions using XMLHttpRequest by default. Although no abuse of such information was found, it is possible for malicious extensions to use this approach to collect users favors or sense users' behaviors that may not be acceptable for some users.

Changing the preferences of the browser. The modifications to some of the browser preferences can lead to exploit or at least annoying users. To evaluate the situation among Firefox extensions that how many extensions access the critical preferences of the browser, we intercepted the "getBranch" method of the "nsIPrefService" interface and logged its parameters. We found 20 extensions accessed "network.*" preferences, 8 accessed "general.useragent.*" accesses, etc.

Capability of dynamic code execution. We found that 145 (5.88%) extensions still use the dangerous JavaScript features "eval()" or "Function()", despite the existence of safer alternatives. These dangerous practices could leave the door open to attackers. Their existence showed that Mozilla's review process does not completely eliminate dangerous and vulnerable coding practices, although Mozilla's Add-on Review Guide [14] clearly states that any extension using "eval()" to evaluate remote code should be rejected.

Listening to the keyboard events on web pages. We found 19 extensions that monitor the user's keyboard operations. However, after our manual reviewing, considering the particular functionality these extensions need to implement, such event monitoring is considered essential for their functional goals. For example, a translator extension "Nice Translator" accelerates the query speed by recording the user input in real time. And another extension "gleeBox" provides a keyboard-centric approach to navigating the web, which also needs to monitor the keyboard events.

Preference names without the "extensions" prefix. Mozilla suggests that it is a good practice to name extension preferences with an "extensions" prefix; otherwise, extension preferences' names may pollute the namespace of the browser's own preference system and affect the stability of the browser. However, during our experiment, we still found up to 266 extensions violating this rule, accounting for 10.79% of the total.

5 Discussion

5.1 Tracking Principals of Indirect Behaviors

Indirect threats to web sessions come from injected contents by browser extensions. A recent proposal [4] introduces new permissions to have fine-grained control on accesses to DOM elements and the capability to introduce new origins into web sessions. This is a promising direction in mitigating indirect threats from browser extensions. However, there are other cases of indirect threats other than introducing new origins via the `src` attributes of "img" of "iframe" HTML tags. For example, things will get more complicated when a script injected into a web session by a browser extension can dynamically create another script, which in turn modifies the web page's original JavaScript to tamper with their original XMLHttpRequest destinations.

We argue that instead of imposing an allow or disallow option for browser extensions to introduce new origins into web sessions, we need a more systematic approach, which tracks the principals of contents from different sources in web sessions. When injected contents from browser extensions modify original contents in the web page, the principal of the injected content should be propagated to the modified contents. As a result, additional security mechanisms, such as permission checking or access control can be applied on the dynamic principals of different components in the same web session. This approach would not only mitigate indirect threats to web sessions from browser extensions, but would allow legitimate interactions between browser extensions and web sessions.

5.2 Coverage of Extension Behaviors

In our testing system, we setup a simple web site to simulate the real world web sites, and we use XdoWrapper to simulate users' click or keyboard strike events. However,

some extensions are designed to work only on specific web sites and some behaviors will only be triggered by certain user actions. Since we are not able to simulate the exact environments for them, it is possible that certain behaviors are not triggered in our testing system. As our future work, we will work on solutions to achieve better coverage of extension behaviors with assistance from static program analysis.

6 Related Work

We discuss research work related to browser extension security in the following categories.

A. Study of security and privacy in browser extensions. Martin Jr. et al [15] investigate privacy issues in IE 6 extensions, where they found some extensions monitoring users' behaviors or intercepting and disclosing SSL-protected traffic. A more recent study [16] investigates privileges in 25 Firefox extensions that are necessary for extensions' functionalities, and found that only 3 out of the 25 extension would actually require the most powerful capabilities of the privileges Firefox extensions all have, violating the least privilege principle. Compared to them, our work focuses on dangerous behaviors in Firefox extensions, and our study was conducted with an automatic testing tool.

B. Securing browser extensions. Based on the weaknesses of the old Firefox extension system, Barth et el. [3] propose a new browser extension system for Google Chrome, which is designed with least privilege, privilege separation and strong isolation. However, Liu et al. [4] find that the original design of the Chrome extension framework had still violated the principles of least privileges and privilege separation, and they propose improvements to it with micro-privilege management and fine-grained access control to DOM elements. Similarly, Mozilla develops a new extension framework Jetpack [5] to make it easier to develop more secure browser extensions. The basic idea of the new framework is to isolate extensions into a collection of modules, each of which is expected to follow the principle of least authority (POLA). Karim et al. [17] perform a static analysis on 77 core Jetpack framework modules and 359 extensions, where they find 12 and 24 capability leaks, respectively. Although we propose potential improvement to mitigate indirect threats to web sessions from browser extensions, we do not aim for a redesign of browser extension frameworks in this paper. Our focus is on the existing behaviors of Firefox extensions that are currently used by users today.

Sabre [18] is a system that analyzes the browser extensions by monitoring in-browser information-flow. It produces an alert when an extension is found to access some sensitive information in an unsafe way. A similar approach [19] is also used to detect attacks against privilege escalation vulnerabilities in Firefox extensions. Another recent work by Ter Louw et [1, 2] discusses techniques for runtime monitoring of extension behaviors. They try to reduce the threats posed by malicious or buggy Firefox extensions by controlling an extension's access to XPCOM. Similar to this work, to protect users from spy add-ons, SpyShield [20] uses an access-control proxy to control communications between untrusted add-ons and their host application. Some static approaches are also proposed to detect vulnerabilities in JavaScript-based widgets. GATEKEEPER [21] is a static approach for enforcing security and reliability policies for JavaScript programs.

VEX [6] is a framework for identifying potential security vulnerabilities in browser extensions by static information-flow analysis.

Compared to the work above, our focus in this paper is on dangerous behaviors in existing Firefox extensions, rather than detecting vulnerabilities or attacks.

C. Automatic event simulation. Some existing systems for web application testing, such as Selenium [22], Watir [23] and [24] can simulate mouse and keyboard events on webpages. However, they cannot trigger events in Firefox extensions. In this work, we provide additional support for Firefox's internal event simulation.

7 Conclusion

In this paper, we present a large-scale study on dangerous behaviors in Firefox extensions. We focus on investigating the indirect threats posed by extensions by tracking the behaviors of new web content injected by extensions. Through an automatic testing system equipped with an injected object tracker, we tested over 2,465 extensions in 13 different categories from the Mozilla Addon repository. We found that there are 108 extensions in total injecting various contents (such as scripts, iframes, images and so on) into web pages. Although these cases are not malicious, they can be abused to tamper with web sessions and should be tackled with special care. To mitigate this kind of threats, we discuss a solution to apply principal tracking to constrain the behaviors of injected content by extensions, while still maintaining usability for legitimate behaviors.

Acknowledgments. We are grateful to the anonymous reviewers for their insightful comments and suggestions. This research was supported in part by National Natural Science Foundation of China (No.90718023, 91118003), Tianjin Research Program of Application Foundation and Advanced Technology (No.10JCZDJC15700), "985" funds of Tianjin University, and National University of Singapore under NUS Young Investigator Award R-252-000-378-101.

References

1. Ter Louw, M., Lim, J.S., Venkatakrishnan, V.N.: Enhancing web browser security against malware extensions. Journal in Computer Virology 4, 179–195 (2008)
2. Ter Louw, M., Lim, J.S., Venkatakrishnan, V.N.: Extensible Web Browser Security. In: Hämmerli, B.M., Sommer, R. (eds.) DIMVA 2007. LNCS, vol. 4579, pp. 1–19. Springer, Heidelberg (2007)
3. Barth, A., Felt, A.P., Saxena, P., Boodman, A.: Protecting browsers from extension vulnerabilities. In: Network and Distributed System Security Symposium (2010)
4. Liu, L., Zhang, X., Yan, G., Chen, S.: Chrome extensions: Threat analysis and countermeasures. In: Proceeding of the Network and Distributed System Security Symposium, NDSS 2012 (2012)
5. Mozilla. Jetpack, `https://wiki.mozilla.org/Jetpack`
6. Bandhakavi, S., King, S.T., Madhusudan, P., Winslett, M.: Vex: vetting browser extensions for security vulnerabilities. In: Proceedings of the 19th USENIX Conference on Security, Berkeley, CA, USA, p. 22 (2010)

7. Mozilla add-ons, https://addons.mozilla.org/
8. xdotool, http://www.semicomplete.com/projects/xdotool/
9. Spidermonkey, https://developer.mozilla.org/en/SpiderMonkey
10. htmlcxx - HTML and CSS APIs for C++, http://htmlcxx.sourceforge.net/
11. libcurl - the multiprotocol file transfer library, http://curl.haxx.se/libcurl/
12. Security severity ratings, https://wiki.mozilla.org/Security_Severity_Ratings
13. Severity guidelines for security issues, http://dev.chromium.org/developers/severity-guidelines
14. Add-on review guide, https://wiki.mozilla.org/AMO:Editors/EditorGuide/AddonReviews
15. Martin Jr., D.M., Smith, R.M., Brittain, M., Fetch, I., Wu, H.: The privacy practices of web browser extensions. Communications of the ACM (2001)
16. Felt, A.P.: A survey of firefox extension API use. Technical report, University of California at Berkeley (2009)
17. Karim, R., Dhawan, M., Ganapathy, V., Shan, C.-C.: An Analysis of the Mozilla Jetpack Extension Framework. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 333–355. Springer, Heidelberg (2012)
18. Dhawan, M., Ganapathy, V.: Analyzing information flow in javascript-based browser extensions. In: Computer Security Applications Conference, ACSAC (2009)
19. Djeric, V., Goel, A.: Securing script-based extensibility in web browsers. In: Proceedings of the 19th USENIX Conference on Security, USENIX Security 2010, p. 23. USENIX Association, Berkeley (2010)
20. Li, Z., Wang, X.-F., Choi, J.Y.: SpyShield: Preserving Privacy from Spy Add-Ons. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 296–316. Springer, Heidelberg (2007)
21. Guarnieri, S., Livshits, B.: Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In: USENIX Security Symposium (2009)
22. Selenium web application testing system, http://seleniumhq.org/
23. Watir automated webbrowsers, http://wtr.rubyforge.org/
24. Balduzzi, M., Egele, M., Kirda, E., Balzarotti, D., Kruegel, C.: A solution for the automated detection of clickjacking attacks. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, pp. 135–144 (2010)