Monkey-in-the-browser: Malware and Vulnerabilities in Augmented Browsing Script Markets

Steven Van Acker, Nick Nikiforakis, Lieven Desmet, Frank Piessens, Wouter Joosen
{firstname.lastname}@cs.kuleuven.be
iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium

ABSTRACT

With the constant migration of applications from the desktop to the web, power users have found ways of enhancing web applications, at the client-side, according to their needs.

In this paper, we investigate this phenomenon by focusing on the popular Greasemonkey extension which enables users to write scripts that arbitrarily change the content of any page, allowing them to remove unwanted features from web applications, or add additional, desired features to them. The creation of script markets, on which these scripts are often shared, extends the standard web security model with two new actors, introducing novel vulnerabilities.

We describe the architecture of Greasemonkey and perform a large-scale analysis of the most popular, community-driven, script market for Greasemonkey. Through our analysis, we discover not only dozens of malicious scripts waiting to be installed by users, but thousands of benign scripts with vulnerabilities that could be abused by attackers. In 58 cases, the vulnerabilities are so severe, that they can be used to bypass the Same-Origin Policy of the user's browser and steal sensitive user-data from all sites. We verify the practicality of our attacks, by developing a proof-of-concept exploit against a vulnerable user script with an installation base of 1.2 million users, equivalent to a "Man-in-the-browser" attack.

Keywords

Augmented browsing; browser extension; Greasemonkey; script market; userscripts.org; malware; vulnerabilities; DOM-based XSS; large-scale analysis

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; H.3.5 [**Information Storage and Retrieval**]: Web-based services

1. INTRODUCTION

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASIA CCS'14, June 4–6, 2014, Kyoto, Japan.
Copyright 2014 ACM 978-1-4503-2800-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2590296.2590311.

The web has evolved from a collection of purely static pages to entire web applications, making the browser the medium of choice for delivering new software and services. With this migration, many power users who used to customize their operating system and install their applications of choice, now feel the desire to customize the applications inside their browser, in a way that fits their needs. These customizations usually result in an enhanced form of browsing the web, which is called "augmented browsing".

Probably the most well-known instance of augmented browsing software is the *Greasemonkey* [8] browser extension, which, at the time of this writing, ranks fifth in the list of most popular Firefox extensions [20]. Greasemonkey users can write *user scripts*, i.e., small JavaScript programs, that manipulate loaded webpages on the client-side in any way desired. User scripts can, among others, hide ads, change the content layout of a page, and make cross-origin HTTP requests to create client-side mashups. In contrast with typical browser extensions, user scripts are comprised of a single JavaScript file and are not packaged in any way, making them easy to inspect and modify.

Due to the popularity of Greasemonkey and the large number of user scripts created for it, the Greasemonkey developers created `userscripts.org` [21], a community-driven script market, on which members can exchange user scripts.

The creation of a script market brings along some unique security issues, because it extends the standard web attacker model with new actors. In the regular model, a website is visited by a client and an attacker can either attack the website by exploiting server-side vulnerabilities, or the visitor through client-side vulnerabilities, like XSS or CSRF. In the augmented browsing scenario, however, the model is extended with the inclusion of a user script in the visitor's browser, a script market, and a script author creating and sharing user scripts through the script market.

In this paper, we perform an in-depth analysis of this extended script ecosystem. First, we consider the script author as a malicious actor, having the ability to create user scripts with malicious functionality, and upload them to the script market where they may be downloaded and installed by victim users. We report on the prevalence of malicious scripts, the discovered malice, and whether this malice was identified by the `userscripts.org` community.

Second, we shift our focus to the possibility of conducting attacks on poorly coded user scripts. We find many instances of benign scripts whose authors, even though they had no bad intentions, unwillingly introduced vulnerabilities which could be used to attack websites that are otherwise

secure. Using straightforward static-analysis techniques, we identify more than 100 user scripts, with millions of installations, vulnerable to DOM-based XSS [12]. We also show that a certain type of user script vulnerability can be abused to launch attacks even against the Greasemonkey engine itself, leading to powerful global XSS attacks, where an attacker can steal a user's data from all sites.

Our main contributions are:

1. We evaluate the Greasemonkey browser extension, focusing on the functionality with negative security consequences.
2. We analyze the most popular, community-driven script market for Greasemonkey and describe the difficulties of relying on the community to define and identify maliciousness.
3. We demonstrate novel attacks that take advantage of benign Greasemonkey scripts to attack, otherwise secure, websites.

2. GREASEMONKEY

In this section, we describe the Greasemonkey engine, its uses, and the structure of Greasemonkey scripts. Finally we examine how Greasemonkey affects the security and isolation of scripts in the browser.

2.1 Greasemonkey engine

Greasemonkey is a popular browser add-on for augmented browsing. Using Greasemonkey, users can, on the client side, modify the appearance and functionality of any webpage. This is done by JavaScript programs that are injected in arbitrary webpages and have access to privileged functionality, not available to normal JavaScript programs. Through these Greasemonkey scripts and with the help of the browser's DOM, users can arbitrarily edit a webpage, including the removal of content, e.g., ads, or the addition of new content, e.g., adding missing functionality to a web application, or creating mashups using content from multiple domains.

While Greasemonkey was originally a Firefox-specific extension, there are also ports of the extension to other browsers, like Tampermonkey for Google Chrome. According to the extension markets of Mozilla Firefox and Google Chrome, at the time of this writing, there are almost three million users who have the Greasemonkey and Tampermonkey extensions installed. Moreover, due to the popularity of the extension, a subset of the Greasemonkey functionality is, by default, supported in many modern browsers, where Greasemonkey scripts are treated as a special case of browser extensions.

In general, Greasemonkey scripts can be considered lightweight browser extensions. Users can write their own scripts, or find scripts written by other users, either dispersed on the web, or concentrated on community-driven script markets, much like the aforementioned popular extension stores. As further explained in the next section, Greasemonkey scripts are single-file JavaScript programs, without Manifest files and directory structures, which users can inspect and edit from within the Greasemonkey extension.

2.2 Greasemonkey scripts

In this section, we demonstrate the basic structure and syntax of Greasemonkey user scripts, and the necessary concepts for the comprehension of the rest of the paper.

Listing 1 Example of a Greasemonkey user script

```
// ==UserScript==
// @name          Hello World
// @description   Description of this script
// @namespace     http://author.com/gmscripts
// @include       http://example.com/*
// @include       http://*.example.com/*
// @exclude       http://login.example.com/*
// @grant         GM_xmlhttpRequest
// ==/UserScript==

alert("Hello World");
GM_xmlhttpRequest({
  method: "GET",
  url: "http://www.shopping.com/",
  onload: function(response) {
    alert(response.responseText);
  }
});
```

2.2.1 *Structure of scripts*

Listing 1 shows a simple example of a user script. Notice that before the actual functionality of the script, there is script-specific meta-data in the form of comments enclosed by `// ==UserScript==` and `// ==/UserScript==`

The Greasemonkey engine will recognize the comments containing `@` signs and read-in the appropriate values. The `@name` and `@description` directives specify the title of a script and a user-readable description of what the script does. The `@include` and `@exclude` directives allow the script authors to specify the domains and webpages that their script should execute on. The `@grant` directive specifies that the listed function should be added to the Greasemonkey sandbox.

The actual code of the user script starts where the meta-data comment block ends. In our example, the first call is to the standard `alert` function provided to JavaScript from the Browser Object Model and used to display message boxes to the user. The second function call, however, is towards a special Greasemonkey-specific function. *Greasemonkey API* functions have the `GM_` prefix and are typically able to do operations not allowed by standard JavaScript code. In this case, the script performs a cross-domain HTTP request to `http://www.shopping.com`, an operation that is otherwise forbidden by the *Same Origin Policy* (SOP), the browser's default security policy, for security and privacy reasons. Other Greasemonkey functions allow a user script to store and retrieve persistent data, access script-specific resources, and register menu commands in the browser.

Figure 1 shows the Greasemonkey dialog that is displayed to the user trying to install our example user script. Notice that at the bottom of the dialog, the user is warned that the scripts can violate the user's security and privacy, and that the user is supposed to install scripts only from trusted sources.

2.2.2 *Important meta-data*

In this section, we expand upon some of the aforementioned Greasemonkey directives with security and privacy consequences.

`@include`. As described earlier, Greasemonkey consults the `@include` directive to determine which pages a user script

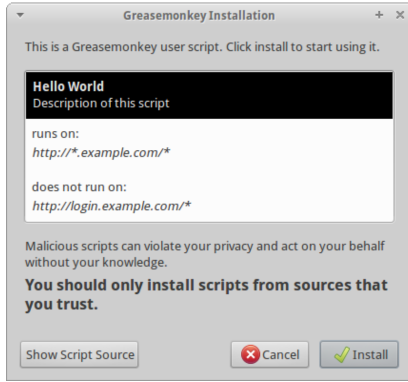

Figure 1: Greasemonkey Script Installation Dialog

should be injected in. Greasemonkey uses regular expressions to match the `@include` header against the entire URL, allowing a lot of flexibility. The script author might for instance add `@include http*://*.example.com` to allow the script to run on both HTTP and HTTPS versions of the example.com sub-domains, and the script author is even allowed to specify `@include *` to run the script on any website. If no `@include` directive is present, Greasemonkey will default to `@include *` for that user script.

`@grant`. Recognizing the merits of the least-privilege principle, Greasemonkey allows script authors to specify which functions of the Greasemonkey API should be added to the Greasemonkey sandbox, using the `@grant` header.

Consider again the user script listed in Listing 1, displaying the usage of this `@grant` header to request access to the `GM_xmlhttpRequest` function. The special directive `@grant none` is used to indicate that the script uses no Greasemonkey API functions at all, and thus none should be added to the sandbox. In the absence of `@grant` headers, Greasemonkey will attempt to infer the necessary API functions by analyzing the user script.

2.3 Attack surface

At this point, it should be evident that the extra functionality of user scripts, unfortunately comes with room for extra vulnerabilities. We consider three different attack scenarios: a) malicious user scripts abusing the pages in which they are injected, b) attackers abusing benign but vulnerable user scripts to attack webpages and, c) malicious pages trying to abuse the Greasemonkey engine and gain access to privileged functions.

In the first scenario, a victim installs a user script that advertises some functionality, e.g., automatically hiding ads on all webpages. This script may be a trojan horse which, next to hiding ads, steals private data from pages, the user's cookies, or even capture all of the user's keystrokes.

In the second scenario, an attacker can take advantage of vulnerabilities introduced by user scripts on pages that otherwise would have no exploitable vulnerabilities, e.g., the exploitation of a DOM-based XSS vulnerability on a webmail application introduced by the added functionality of a Greasemonkey user script.

In the third scenario, an attacker can take advantage of user script vulnerabilities, not just to inject code in a benign page, but to inject code in Greasemonkey's sandbox. Greasemonkey makes use of sandboxing to protect the privileged `GM_` functions from possibly malicious scripts running on a website. Despite this sandbox and additional stack-inspecting mechanisms of Greasemonkey, a poorly-written user script can still introduce unsafe code in the sandboxed environment, e.g., by `eval`-ing a string from a malicious page without performing the proper sanity checks. When this happens, a malicious script can get access to the powerful Greasemonkey API and circumvent the SOP.

3. COMMUNITY-DRIVEN SCRIPT MARKETS

Script markets facilitate the sharing of Greasemonkey scripts by providing script authors with a disseminating platform and a feedback mechanism, and consumers of scripts with comments and ratings about the quality and utility of a particular script.

We retrieved a total of 86,358 user scripts together with their accompanying meta-data, from `userscripts.org`. This meta-data includes how many times a user script has been installed, and whether or not the user script was flagged as "Harmful" by the community.

More information about the dataset can be found in [23].

4. MALWARE ASSESSMENT

Greasemonkey scripts are more powerful than traditional JavaScript programs, because they can manipulate and retrieve private data in a user's browser without SOP restrictions. Consequently, such scripts can be an attractive infection vector for malware authors, who can create malicious user scripts and trick users into installing them.

In this section, we discuss malware in Greasemonkey user scripts and how the `userscripts.org` community is currently attempting to deal with malicious scripts.

4.1 Userscripts.org issue reporting

The `userscripts.org` community website has a community-based, manual reviewing process to detect malicious user scripts. When a malicious user script is detected, the user can flag it as "harmful" and, optionally, explain her vote in the comment section.

In our dataset of 86,358 scripts, 626 (0.7%) are marked as "harmful" by at least one user of the `userscripts.org` community. Of those 626 scripts, 592 have at least as many votes in favor of "harmful" as votes against it.

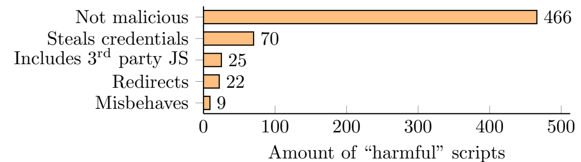

Figure 2: Categories of malware found in the 592 scripts labelled as "harmful" in the `userscripts.org` dataset. Almost 80% is harmless.

To determine the quality of this manual review process, we performed a manual analysis of these scripts to determine what users regard as "harmful". From the 592 "harmful"

scripts, we could not find any trace of malice in 466 (78.7%) of them. For our purposes, we defined malice as the attempt to steal private data from a user, or trick the user into performing an action with potential monetary benefits for the attacker. We will refer to the remaining 126 scripts that do contain malware, as the *verified harmful dataset*.

A breakdown of the entire harmful dataset according to the reason the scripts were flagged, is shown in Figure 2. The largest fraction with verified maliciousness is comprised by 70 scripts containing malware designed to steal credentials in some form, from the user. The next largest fraction of scripts (25) includes third party JavaScript into a loaded page [16]. Twenty-two scripts simply redirect the user to another website, with the possible intent to lure the user into a drive-by-download scenario and install malware that way. Only five of these "redirect" scripts were reported by users to be the cause of a drive-by-download attack. Finally, there remain nine scripts which "misbehave", and do not fit in the previous categories.

Shifting our attention to the 466 benign scripts that were mislabeled as malicious, the given reasons indicate that the concept of "harmful" is not always clear to the members of the community, and that there should be a clearer definition.

A more detailed breakdown of the reported malware, as well as observations that can help in the detection of malware, can be found in [23].

5. ATTACKING WEAK SCRIPTS

In the previous section, we discussed scripts that are malicious by design, giving their authors the ability to harm those scripts' users. Because Greasemonkey injects user scripts into visited webpages, these user scripts unfortunately increase the attack surface of the user.

In this section, we discuss two vulnerabilities that occur in user scripts: DOM-based XSS and overly generic `@include` directives. Through these vulnerabilities, an attacker can trick a victim's browser into executing code on webpages onto which a user script acts, or even any webpage he wants, and potentially even gain access to powerful Greasemonkey API functions.

5.1 DOM-Based XSS

DOM XSS vulnerabilities present in user scripts are more dangerous than regular DOM XSS vulnerabilities because the vulnerability will be injected into any page on which the user script code is included. Attackers can then exploit these vulnerabilities and gain access to the Greasemonkey API, which is not bound by the SOP.

DOM-based XSS analysis setup. To determine whether any DOM-based XSS vulnerabilities occur in our user scripts dataset, we screen all scripts using a lightweight static-analysis method. Using SpiderMonkey [19], we parsed all scripts in our dataset and obtained a simplified AST for each one of them. Using a list of sources and sinks [7], we searched for sources used directly in the argument list of sinks. As such, all the results reported in the next sections are lower bounds of vulnerabilities.

Results. The results of our DOM-based XSS analysis on the full dataset retrieved from `userscripts.org`, are shown in Table 1. From the 86,358 scripts in our dataset, our analysis revealed 1,736 that contain a DOM-based XSS. The majority of scripts are vulnerable through the `e.innerHTML` sink (1,654 or 95.3%) and the various sources originating from the `document` object (99.7%).

	`document`			`window`	**Total**
	`.body`	`.location`	*other*	*.name*	
`doc.write(x)`	0	1	2	0	3
`eval(x)`	3	0	76	0	79
`innerHTML`	721	83	846	5	1,654 (*)
Total	724	84	924	5	1,736 (*)

Table 1: Scripts with detected DOM-based XSS vulnerabilities according to the used sources and sinks. (*) Total reflects the amount of unique scripts for the given sink, not the row sum.

Note that some sources might require the ability of an attacker to place persistent data onto a website. From the dataset, 101 scripts are vulnerable to DOM-based XSS with sources not bound by this requirement.

The most prominent, vulnerable to DOM-based XSS, user script that we discovered is the fourth most popular script on the `userscripts.org` script market, with almost 40 million installations.

5.2 Overly generic `@include`

As explained in Section 2.2, the `@include` directive specifies which webpages a user script is injected in. The `@include` directive allows wildcards, and uses regular expression to test the entire URL of the webpage being visited.

If the `@include` wildcard is used in a too generic way, this can lead to a security problem. For instance, reconsider the introductory example in Listing 1. In this script, the directive `@include http://*.example.com/*` is used. An attacker can construct the URL `http://www.mybank.com/#x.example.com/abc` and trick a user of this script to visit it. Greasemonkey's regular expression will then match the `@include` directive against this crafted URL and falsely assume that the author of the script wants the script to be executed on `http://www.mybank.com/`. The attacker has caused the script to run on a webpage for which it was not intended, by abusing the `@include` wildcard.

`@match`. The developers of Google Chrome, in their adaptation of the Greasemonkey engine, recognized that the wildcard `*` in the `@include` directive, was not strict enough and could lead to insecure situations. For this reason, they created the `@match` [4] directive which provides the same functionality as `@include`, but in a safer way.

Google Chrome's `@match` wildcard is context-sensitive and is applied by splitting a URL into three parts: a scheme, a host and a path. A `*` wildcard can occur within each part, but cannot match anything that violates the borders between the parts.

To be compatible with user scripts for Google Chrome, Greasemonkey adopted the `@match` directive alongside its `@include` directive. In cases where both `@include` and `@match` directives are used, the `@include` directive is handled first.

	@match	No @match	Total
@include securely	670	33,775	34,445
@include insecurely	770	39,955	40,725
No @include	884	10,304	11,188
Total	2,324	84,034	86,358

Table 2: @include and @match directive usage, "insecurely" means an overly generic @include

Usage of the @include and @match directives. Table 2 divides the scripts in our dataset with regard to @include and @match directives. From the 86,358 scripts in our dataset, 75,170 (87.0%) contain a @include directive, of which 40,725 insecurely with a too generic wildcard. Since scripts without an explicit @include directive automatically obtain a @include * directive, this means that 51,913 scripts or 60.1% of the full dataset can be tricked into executing on a different domain than the one they were designed for.

Only 2,324 specify a @match directive, of which 1,440 also specify an @include directive. Of those 1,440, 770 have insecure @include directives, meaning the @match directive's security advantage over a @include, is completely negated.

The most popular script with an unsafe @include directive is, at the same time, the most popular script on userscripts.org, a social networking script with more than 250 million downloads. It uses an overly generic wildcard @include directive of the form @include http://*website.com/*.

5.3 Resulting malicious capabilities

Global XSS. The combination of a DOM-based XSS vulnerability, and an overly generic @include directive, results in a critical vulnerability. Scripts which contain this combination of vulnerabilities allow attackers to execute malicious code on any webpage of choice, by having victims visit appropriately constructed URLs.

From the 1,736 vulnerable scripts revealed from our analysis to be vulnerable to DOM-based XSS vulnerabilities, 944 (54.3%) also use overly generic @include directives and can thus be used to perform global XSS attacks.

Privileged XSS. The case of a DOM-based XSS where attacker-controlled data find its way into an eval(x) sink reveals an extra security issue because it allows malicious code to execute inside the Greasemonkey sandbox and gain access to the Greasemonkey API.

Consider for instance the example in Listing 1. The example script uses GM_xmlhttpRequest to get access to cross-origin resources from http://www.shopping.com/. This API function will be present in the sandbox where the user script executes, because @grant GM_xmlhttpRequest is used to request it. If this example script also contained a DOM-based XSS vulnerability with an eval(x) sink, then a malicious website could trigger this vulnerability, executing code inside the Greasemonkey sandbox and get access to the powerful GM_xmlhttpRequest function.

From the 79 scripts that contain a DOM-based XSS with an eval(x) sink, 60 execute in a sandboxed environment with access to the Greasemonkey API and can thus leak that API to a malicious website which may abuse it.

Privileged, global XSS. To aggravate the problem further, it is possible to combine all three vulnerabilities: a script with an overly generic @include directive, vulnerable to a DOM-based XSS attack where attacker-controlled data flow into eval(x), thereby exposing the Greasemonkey API.

A script such as this can be abused by an attacker by luring victims to a specially crafted URL, causing attacker-controlled code to be executed, with access to the powerful Greasemonkey API. Since the Greasemonkey API functions are not bound by the Same Origin Policy, an attacker could then abuse them to steal private data from the victim's browser, across all sites. From the 60 scripts we identified as being vulnerable to a DOM-based XSS with an eval(x) sink and which also expose the Greasemonkey API, 58 use an overly generic @include directive.

The most prominent example is a script installed by 1.2 million users, which can be forced to run on any website, due to its overly generic use of wildcards. Moreover, the script makes insecure use of eval allowing an attacker to execute arbitrary code in the Greasemonkey sandbox. We created a proof-of-concept exploit which amounts to a Man-in-the-Browser attacker, i.e., we can conduct requests towards all websites (together with the user's cookies), read the responses, and inject malicious JavaScript on any domain.

We are in the process of disclosing the vulnerabilities to the involved parties.

6. RELATED WORK

To the best of our knowledge, this paper is the first one that tries to shed light on alternative, community-driven, JavaScript markets. Closely related, however, is research done in identifying malicious and vulnerable browser extensions from the official extension markets of Mozilla Firefox [3] and Google Chrome [9], showing that extensions often request more privileges than needed. VEX [2] analyzes Firefox extensions, such as Greasemonkey, for privilege escalation vulnerabilities, but does not analyze the user scripts used by Greasemonkey itself.

While malicious browser extensions are typically written in JavaScript, malicious JavaScript, today, has a different connotation, that of code which exploits some vulnerability in the browser or in one of the browser plugins to eventually lead to drive-by downloads. Due to the great magnitude of the problem, there has been a significant body of research in detecting malicious JavaScript, using honeypots [15], dynamic analysis of JavaScript code [5, 11, 13, 17], and hybrid systems [6, 18] which utilize both static and dynamic techniques to analyze JavaScript code.

The main difference of this type of malicious JavaScript with the types of malicious Greasemonkey scripts analyzed in this paper, is that in our case, maliciousness is context-specific. That is, it may only be discoverable when the user is on a specific page of a specific website, making dynamic detection of context-specific maliciousness significantly harder to define, as well as detect.

Typical JavaScript sandboxing techniques [1, 10, 14, 22] attempt to isolate malicious code in a controlled environment and prevent references to powerful functionality from leaking inside the sandbox. In contrast, Greasemonkey creates a sandbox with its powerful API inside and attempts to prevent the leakage of references to this API to the outside.

7. CONCLUSION

In this paper, we analyzed the Greasemonkey browser extension and the `userscripts.org` script market, searching for evidence of malware and vulnerabilities, as well as documenting the ways with which community-driven script markets deal with malicious scripts. Through this process, we find that the review process of `userscripts.org` is ineffective in 78% of the cases.

Moreover, we identify and analyze two types of vulnerabilities found in user scripts, which could allow an attacker to use the restricted and powerful Greasemonkey functions to, among others, bypass the Same Origin Policy, and force a user script to run on any website.

We found that DOM-based XSS vulnerabilities are present in 2% of user scripts and that 60.1% of user scripts can be forced to run on any webpage. Finally, we show how an attacker can combine many vulnerabilities to launch powerful privileged, global XSS attacks and discover 58 scripts that are susceptible to this attack. We verify this attack through a proof-of-concept exploit for one of these user scripts, installed by over a million users, allowing an attacker to steal user data across all sites.

Acknowledgements

This research was performed with the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE), the Research Fund KU Leuven, the FP7 projects STREWS, NESSoS and WebSand, and the IWT project SPION.

8. REFERENCES

[1] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: Complete client-side sandboxing of third-party javascript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 1–10. ACM, 2012.

[2] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium*, pages 339–354. USENIX Association, 2010.

[3] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *NDSS*. The Internet Society, 2010.

[4] Match Patterns - Google Chrome. `https://developer.chrome.com/extensions/match_patterns.html`.

[5] M. Cova, C. Kruegel, and G. Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proceedings of the World Wide Web Conference (WWW)*, 2010.

[6] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[7] DOM XSS Test Cases Wiki Cheatsheet Project. `https://code.google.com/p/domxsswiki/`.

[8] Greasemonkey. `https://addons.mozilla.org/en-US/firefox/addon/greasemonkey/`.

[9] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 115–130, Washington, DC, USA, 2011. IEEE Computer Society.

[10] L. Ingram and M. Walfish. Treehouse: JavaScript sandboxes to help web developers help themselves. In *Proceedings of the USENIX annual technical conference*, 2012.

[11] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. In *Proceedings of USENIX Security*, 2013.

[12] A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. `http://www.webappsec.org/projects/articles/071105.shtml`, April 2005.

[13] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 443–457, Washington, DC, USA, 2012. IEEE Computer Society.

[14] J. Magazinius, P. Phung, and D. Sands. Safe wrappers and sane policies for self protecting javascript. In *15th Nordic Conference on Secure IT Systems*, 2010.

[15] Y. min Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of 13th Network and Distributed Systems Security Symposium (NDSS '06)*, 2006.

[16] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2012.

[17] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 169–186, Berkeley, CA, USA, 2009. USENIX Association.

[18] K. Rieck, T. Krueger, and A. Dewald. Cujo: Efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 31–39, New York, NY, USA, 2010. ACM.

[19] Mozilla SpiderMonkey. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`.

[20] Mozilla add-ons - featured extensions. `https://addons.mozilla.org/en-US/firefox/extensions/`.

[21] Userscripts.org. `http://userscripts.org/`.

[22] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. WebJail: least-privilege integration of third-party components in web mashups. ACSAC '11, pages 307–316, New York, NY, USA, 2011. ACM.

[23] S. Van Acker, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen. Monkey-in-the-browser: Malware and vulnerabilities in augmented browsing script markets – extended version. Technical report, Mar. 2014.