Monkey-in-the-browser: Malware and vulnerabilities in augmented browsing script markets – extended version

Steven Van Acker
Nick Nikiforakis
Lieven Desmet
Frank Piessens
Wouter Joosen
Report CW 657, March 2014

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Monkey-in-the-browser: Malware and vulnerabilities in augmented browsing script markets – extended version

Steven Van Acker
Nick Nikiforakis
Lieven Desmet
Frank Piessens
Wouter Joosen
Report CW 657, March 2014

Department of Computer Science, KU Leuven

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 newly identified types of 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 have discovered several of these severely vulnerable scripts, with over a million installations, and created a proof-of-concept exploit that successfully launches a novel "Man-in-the-browser" attack against an installed vulnerable script with an installation base of 1.2 million users.

Keywords : Augmented browsing, browser extension, Greasemonkey, community driven, script market, userscripts.org, malware, vulnerabilities, DOMXSS, large-scale analysis, javascript sandbox
CR Subject Classification : K.6.5

Monkey-in-the-browser: Malware and vulnerabilities in augmented browsing script markets – extended version

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 newly identified types of 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 have discovered several of these severely vulnerable scripts, with over a million installations, and created a proof-of-concept exploit that successfully launches a novel "Man-in-the-browser" attack against an installed vulnerable script with an installation base of 1.2 million users.

1. INTRODUCTION

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. For many users, the desktop appears to do little more than house their browser and manage their Internet connection. 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* [11] browser extension, which, at the time of this writing, ranks fifth in the list of most popular Firefox extensions [26]. 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. Overall, Greasemonkey and user scripts tend to a different audience than the neatly-packaged browser extensions available on the traditional extension markets.

Due to the popularity of Greasemonkey and the large number of user scripts created for it, the Greasemonkey developers created a community website on which members can exchange user scripts: a community-driven, script market known as `userscripts.org` [28].

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 with user scripts, 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.

Next, we briefly look at specific scenarios allowing targeted attacks against script users without their knowledge, either at script installation time or any other time during the lifetime of a script within their augmented browser.

Last, 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. 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:

- We evaluate the Greasemonkey browser extension, focusing on the functionality with negative security consequences.
- 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.
- 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 page of the web. 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. Greasemonkey scripts are different from other browser extensions, in that they target a different crowd of users. As further explained in the next section, Greasemonkey scripts are single-file JavaScript programs, without Manifest files and directory structures. Their lightweight nature allows them to be much more website-specific than normal browser extensions, e.g., disabling ads by hiding one specific HTML object on the user's favorite website, or game helping scripts for specific games on popular social networks. In addition, unlike browser extensions, the JavaScript nature of each script is not hidden in archive files. Instead, users can inspect and edit the code of their installed scripts from within the Greasemonkey extension.

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/*
// @require      http://author.com/lib.js
// @updateURL    http://author.com/hw.meta.js
// @downloadURL  https://author.com/hw.user.js
// @grant        GM_xmlhttpRequest
// ==/UserScript==

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

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.

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 `@namespace` directive allows for the separation of scripts that have the same filename. The `@include` and `@exclude` directives allow the script authors to specify the domains and webpages that their script should execute on. The `@require` directive allows the script author to include external JavaScript code and use it from the user script. Both `@updateURL` and `@downloadURL` are used during the automatic user script update process. 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, among others, store and retrieve persistent data, access script-specific resources and register menu commands in the browser.

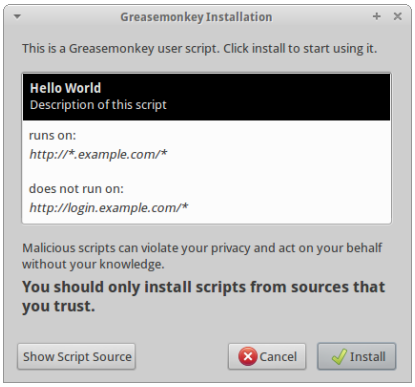

Figure 1: Greasemonkey Script Installation Dialog

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 from only trusted sources.

2.2.2 *Important meta-data*

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

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

`@require`. Greasemonkey allows script authors to base their scripts on external JavaScript libraries through the `@require` directive in the script header. When a script with a `@require` directive is installed, the URL argument of this header is used to download the specified external JavaScript library and store it alongside the installed script. At runtime, the local copy of the external JavaScript library is executed together with the script code.

Consider the user script listed in Listing 1. During the installation of this script, Greasemonkey will find the `@require` header pointing to `http://author.com/lib.js`, download the referenced library script and store it alongside this user script for execution at runtime.

`@updateURL` *and* `@downloadURL`. Greasemonkey has built-in functionality to automatically install updates for installed user scripts, when it detects that an update is available. To make use of this feature, script authors can specify the `@updateURL` and `@downloadURL` directives, as shown in Listing 1.

The `@updateURL` header is used to specify where the latest meta-data for a user script can be found. If this meta-data reveals the availability of a new version of a user script, the update process is triggered. The `@downloadURL` header lists the URL from which the updated script is to be downloaded, once the update process has been triggered.

If `@updateURL` or `@downloadURL` are not found in the script header, Greasemonkey automatically infers them from the location from which the script was installed. Both `@updateURL` and `@downloadURL` can use the HTTP scheme, but only when `@downloadURL` uses an HTTPS scheme, will the update be automatic.

`@grant`. Based on the least-privilege principle, powerful functions like the ones in the Greasemonkey API should be available to user scripts, only if they are absolutely necessary. Recognizing the merits of this 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 acts as a keylogger and captures all of the user's keystrokes. Since JavaScript allows for extensive minification and obfuscation of code, identifying malice by simply inspecting the source code of a script can be an arduous and technically challenging procedure, which the majority of users will most likely not be able to perform.

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, however, of 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.

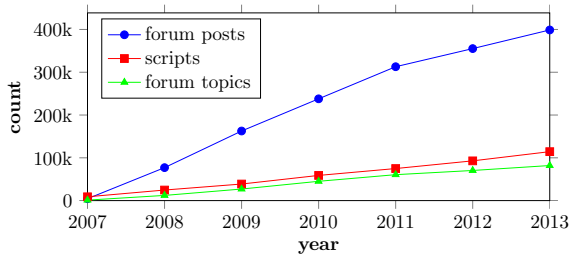

Figure 2: Historical data of `userscripts.org` on user scripts, forum topics and forum posts

When this happens, a malicious script can, for instance, get access to the `GM_xmlhttpRequest` function of Greasemonkey which allows the attacker to send arbitrary requests towards any website, with the user's cookies embedded in them, and read the corresponding responses.

3. COMMUNITY-DRIVEN SCRIPT MARKETS

An augmented browser extension, such as Greasemonkey, allows power users to create user scripts and use them in their daily browsing. Once written, a user script can be useful and generic enough, to be of value to other users. Script markets facilitate the sharing of such 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.

In this section, we discuss `userscripts.org`, the official script market for Greasemonkey scripts, and describe some general features and historical information. In addition, we also report on the building and categorizing of a dataset of user scripts and meta-data from `userscripts.org` that will be used throughout the rest of this paper.

3.1 Userscripts.org

Userscripts.org is an online community established in 2005, which hosts community-provided Greasemonkey scripts and is the official script market associated with Greasemonkey.

The website allows members to upload, update and delete their user scripts. A forum hosted on the website allows the community members to communicate amongst themselves, discussing ideas and user scripts. User scripts can also be reviewed or flagged for further review by flagging them with issues. There are 5 categories of issues, namely "Broken", "Copy", "Harmful", "Spam" and "Vague". Scripts are characterized as "Vague" when the script authors do not adequately describe the purpose of their extensions. Members can vote on whether a flagged issue is present or not, and leave comments to support their vote.

The website also tracks several pieces of meta-data for each user script, among which, is a counter indicating how many times a user script was downloaded and installed. At the time of writing, the website hosts more than 114,000 Greasemonkey scripts written by more than 90,000 registered users. The websites forum contains about 400,000 forum posts spanning more than 82,000 forum topics.

Figure 2 plots data based on historical records [14] indicating the number of user scripts hosted by the community, the number of forum topics and forum posts since 2007. This data shows that the website has grown steadily since its creation. On average, the website has grown by about 48 user scripts, 37 forum topics and 179 forum posts per day, indicating that the Greasemonkey engine and its associated scriptmarket are active, despite the growth of more traditional browser extension markets.

3.2 Gathered dataset and statistics

To gain better insight into the user scripts provided by `userscripts.org`, we retrieved a total of 86,358 user scripts together with their accompanying meta-data.

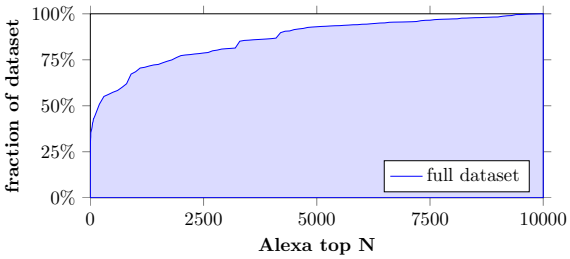

Figure 3: From the 37,893 user scripts for which the @include domain could be analyzed, 29.9% was designed for the Alexa top 3

The authors of the user scripts in this dataset designed their scripts with a specific environment and purpose in mind. By analyzing the meta-data in each user script, we aimed to discover which websites these user scripts are meant to run on, as well as the category of each discovered website.

First, we correlated the meta-data of the 37,893 user scripts in our dataset that are designed for the Alexa [2] top 10,000, "high-profile", websites.

Figure 3 shows the fraction of these "high-profile" user scripts related to the top Alexa domains they target. 11,313 (29.9%) of them are designed for the Alexa top 3 (Facebook, Google and YouTube), while 25,958 (68.5%) are designed for domains in the Alexa top 1,000.

Category	**count**	
Social Networking	12,238	32.3%
Search Engines / Portals	4,570	12.1%
Games	4,271	11.3%
Blogs / Web Communications	2,992	7.9%
Computers / Internet	2,396	6.3%
Total	26,467	69.8%

Table 1: Top five categories to which the "high-profile" user scripts belong, according to the domain for which they were designed.

Next, we used data from Trendmicro SiteSafety [27] to split the user scripts into categories, according to the domain they are designed for. From the 37,893 "high-profile" user scripts in our dataset, 69.8% belong to the five categories shown in Table 1.

The data gathered from our dataset indicates that most user scripts are designed for "high-profile" websites aimed at entertaining and informing users, such as social networking sites, portals, games and blogs. These results show that

Listing 2 This Greasemonkey script can be both malicious or non-malicious, depending on whether the user is aware of its actions

```
var enemies = ["Cylon"];
for (e in enemies)
    game.destroyColony(enemies[e]);
```

many users are willing to install Greasemonkey scripts that operate on websites with valuable private data, like `facebook.com`. As we discuss in later sections, this willingness to trust user scripts can be abused by malicious script authors, in order to gain access to a user's private user data and perform actions on the user's behalf.

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, why it is difficult to automate malware detection in user scripts, and how the `userscripts.org` community is currently attempting to deal with malware. We also analyze the subset of user scripts on `userscripts.org` that was labelled "harmful" by the community review process and provide some observations about this malware to improve the malware detection process.

4.1 Defining and detecting malware in user scripts

Automatic malware detection is certainly a desirable mechanism that could, in theory, be used to protect Greasemonkey users from malicious user scripts, by screening new or updated user scripts, and marking them as malicious. Before this can happen, however, there first needs to be a clear definition of what exactly constitutes a malicious user script. A malicious script can not just be defined as the presence of malicious code in the user script. The context and meta-data also need to be considered.

For instance, consider Listing 2, a code fragment inspired by an existing Greasemonkey script, containing a game helper, i.e., a script that assists a user while playing a specific game and gives him a competitive advantage over other users. When executed in a certain game, the code fragment destroys all colonies of type "Cylon".

If "Cylon" is an opposing team in the game, then the user would probably consider this script harmless and argue that the user script works as intended or advertised. If, on the other hand, the user's team name is "Cylon", then the script would sabotage her game and she would consider the script malicious. Without this additional contextual information, a malware detector would have to resort to the semantics of the script alone, and it would be unclear whether this code fragment should be classified as malicious or harmless.

To avoid such occasions, each script has a description field in its meta-data, where the script author can describe the purpose of the script and thus provide this additional contextual information. A malicious script could then be defined as a script whose semantics do not match its description and is performing some action that the user finds undesirable.

Unfortunately, verifying whether a user script's semantics match its description, is a task requiring non-trivial natural language processing, which in turn, relies on the user's verbosity and writing style. In addition, a user script's semantics are context-specific, and requires a deeper understanding of the web-application it was designed for. As such, automating this malware detection process appears to be a difficult task, which is likely not to produce good results. Based also on our experience reverse-engineering Greasemonkey user scripts, we believe that the task of identifying malice should, at the moment, be left to human reviewers.

4.2 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. Due to the increased issue-related activity around these scripts, we focus on them for the rest of this section.

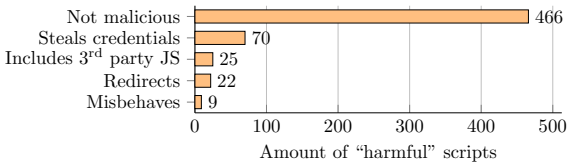

Figure 4: Categories of malware found in the 592 scripts labelled as "harmful" in the `userscripts.org` dataset. Almost 80% is harmless.

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 4. Ignoring, for the time being, the scripts that we discovered to be not malicious, the largest fraction with 70 scripts contains malware designed to steal credentials in some form, from the user. This category contains scripts that steal cookies (29 scripts), steal username and password directly (22), steal username and password through phishing (14) and scripts that log and leak keystrokes (5).

The next largest fraction with 25 scripts include third party JavaScript into a loaded page. From the URL alone, it is not always clear what type of malware these scripts contain, if any. The included code could be dynamically generated and used to target specific users. We further investigate the possibility of targeted attacks in Section 5.

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 simply "misbehave", and can not be summarized in the previous categories. Their behavior is best described as making fraudulent transactions: sending spam on social networks, destroying online game assets, making a PayPal donation, ... etc.

Shifting our attention to the 466 benign scripts that were mislabeled as malicious, the main reasons for this labeling were "bad practices", e.g. providing custom update functionality instead of using the proper built-in functionality of Greasemonkey, and copied user scripts being mislabeled as "Harmful" instead of "Copy". For other scripts the issue reporter claims, among others, that the user script attracts copyright violations, destroys online communities, and even censors freedom of speech. These 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.

In addition, we found some scripts that at one point included malware, but had the malware removed from the latest revision by its author. Such scripts, although now clean, still carry the "harmful" label because the labelling is not always reset on new revisions.

4.3 Malware observations

As mentioned earlier, a completely automatic malware-detection mechanism is not likely to produce good results for Greasemonkey scripts. However, from our manual review of user scripts in the verified harmful dataset, we observed certain patterns that kept on reoccurring in many of the malicious scripts. From these patterns, we derived two detection methods that could assist human reviewers in prioritizing possibly malicious scripts, in the review process.

Malware insertion. During our analysis we observed that malware authors often copy an existing popular script and then add some malicious code, without modifying the original, surrounding code. The resubmitted malicious script is likely to appear during the search for scripts offering a specific functionality and installed by victim users, instead of the original script.

To determine the feasibility of detecting malware by identifying similarities between scripts, we set up an experiment to determine which scripts are copies of other scripts on `userscripts.org`. Comparing each script with every other script is a time-consuming process of complexity $O(n^2)$. Therefore, we limited the scope of our search to scripts of approximately the same size. For example, the size of the *Abstract Syntax Tree (AST)* of the largest piece of malware found during our manual analysis was 5,656 bytes. We doubled this amount and compared each script in our dataset to all older scripts with a maximum of 10KB size difference in the AST. This size-filter reduces the amount of comparisons by about 90%, from 3.7 billion to about 400 million.

Our comparison technique operates as follows: we consider that code is only inserted in one specific location in the script, and calculate what fraction of the new script is derived from an ancestor script.

Figure 5 shows the cumulative percentage of scripts in the full and verified harmful datasets, ordered by their similarity

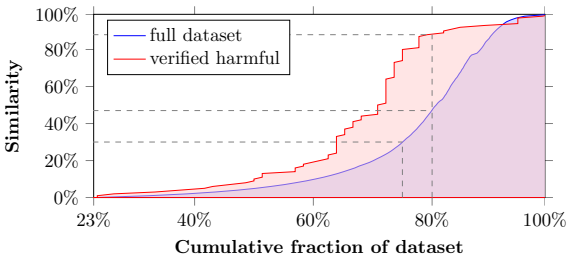

Figure 5: Cumulative amount of scripts in our datasets that are similar to older scripts of the full dataset. Notice that the verified harmful dataset contains more scripts with higher similarity to other scripts, than the full dataset

with older scripts. For the full dataset (blue line), we can see that almost 75% of the scripts have less than 30% in common with the other scripts.

Comparing this to the similarities of the verified harmful dataset (red line), we can clearly see that the malicious scripts are almost always "above" the full dataset, indicating a consistently higher similarity rating for most scripts in this dataset. For instance, we can see that 80% of the full dataset has a similarity rating of about 45%, while the verified harmful dataset has a similarity rating of about 90% for the same fraction of the dataset. Thus, the similarity of a new script with existing ones can be used to guide a human reviewer towards malicious scripts. Our findings are in line with the findings of Kapravelos et al. [15], who notice that authors of traditional JavaScript malware try to evade detection by copying popular JavaScript libraries (like jQuery) and injecting them with malicious code.

Malware reuse. In addition to copying popular user scripts, we also observed that malware authors occasionally recycle malware fragments. Motivated by this observation, we ran the following experiment to uncover additional text-strings that are indicative of malware. From the set of all verified harmful scripts, we extracted all strings of length ten or more, comprised of alphanumeric characters plus '.', '-' and '_'. We then searched for these strings in the full dataset.

Text-string	total	harmful	
`voxDve.indexOf`	7	6	(85.7%)
`voxDve.substr`	7	6	(85.7%)
`xVDs.iterateNext`	7	6	(85.7%)
`eleNew.nextSibling`	23	20	(87.0%)
`eleNew.parentNode.insertBefore`	23	20	(87.0%)

Table 2: Five text-strings appearing in ten or more scripts, of which at least 50%, but less than 100% are verified harmful

From the results of this experiment, we only retained those strings which occur in ten or more scripts, of which at least half are verified harmful, yielding a total of 18 strings. For brevity, we only show five of those in Table 2.

The string `eleNew.parentNode.insertBefore` was found in 20 scripts in the "harmful" dataset all of which were as-

sociated with a malicious cookie-grabber. There are, however, 23 scripts in the full dataset that contain this specific string. The extra three also contain the malware but were not flagged by the community.

This experiment indicates the value of a simple text-search for the community's review process. As we have shown, it is straightforward to extract indicative strings from a base set with known malware. Since a text-search on new scripts for these text-strings is equally simple, this detection method is very effective.

5. TARGETED ATTACKS

In Section 4 we considered that a malicious script author can add malware to a user script. This method has some drawbacks from the malicious author's perspective. The main one is that the malware-containing script has to be uploaded to the script market and is thus available for analysis by the rest of the community.

Another method of spreading malware, which does not require exposing the malware to an online community, is by infecting users during installation of the script, or during the update process. A malicious script author, using one of these methods, can effectively infect users with malware without exposing the malware to an online community and even allow him to cherrypick which users to infect, leading to targeted attacks.

In this section, we discuss these targeted attacks during installation and during the update process In addition, we measure how many scripts in our dataset are susceptible to these attacks.

During user script installation. Consider that the author of the example listed in Listing 1 is malicious and is determined to target a new user of his user script with malware by taking advantage of the `@require` directive.

A review of this user script by the community could show that the script, by itself, does nothing harmful. To review the `@required` JavaScript library, the reviewer would need to download the library from `author.com`, whose server-side code could determine that it is under review and return harmless code. The review of this downloaded JavaScript library will then equally indicate it is harmless.

Reassured by the community review, the targeted user could decide to install the script. During installation of this script, a request will be sent from the user's browser towards `author.com`, requesting the specified JavaScript library. At this point, code running on that webserver can again determine where the request is coming from, e.g. by geo-locating the IP address or fingerprinting the user's browser [10], and reply with custom malware for the targeted user.

Through user script updates. Similarly, the Greasemonkey update process can be abused by a malicious script author to infect a targeted user of his user script, with malware.

Consider again the example in Listing 1 where both `@updateURL` and `@downloadURL` headers are used. At regular intervals, Greasemonkey will initiate the update process for all installed user scripts. If the update request originates from a targeted user, the server-side can pretend there is an update available and push malicious code to the Greasemonkey extension, a fact which will be invisible for any other user of that user script.

Appearance in the dataset. Although these attacks are possible, we can not easily detect whether our dataset contains user scripts that covertly install malware during the installation or update process. Such scripts, after all, would be specially crafted to resist this kind of review. Nevertheless, we are interested in discovering user scripts in our dataset which could be used to covertly install malware.

In the full dataset, 10,866 (12.6%) scripts have a valid `@require` directive. Of these, 3,264 scripts `@require` JavaScript exclusively from `userscripts.org`, 6,897 download them exclusively from third-party domains, and 705 use both. This means that 7,602 scripts (8.8% of the full dataset) `@require` JavaScript libraries from third-party domains and may covertly install malware during the installation process.

The three most popular third-party domains from which external JavaScript is loaded are `googleapis.com` (3,738 scripts), `sizzlemctwizzle.com` (1,339 scripts) and `google-code.com` (868 scripts). The most popular user script in our dataset, which `@require`s an external JavaScript library is a Farmville script with over 60 million installations, and `@requires` JavaScript from `sizzlemctwizzle.com`.

Although these domains can be considered trusted due to their popularity, there are many third-party domains that only occur a handful of times in an `@require` directive in the dataset, indicating that they are most likely tied exclusively to the script's author. Such domains can potentially serve malware covertly.

Shifting our focus to the update mechanism, 1,135 user scripts provide a valid `@updateURL`, 516 provide a valid `@downloadURL` and 481 provide both. As mentioned in Section 2.2, the remaining 85,188 scripts without either a `@updateURL` or `@downloadURL` have the respective URL derived from the location from which the script was installed, which in this case is `userscripts.org`.

From the 516 scripts that provide a valid `@downloadURL`, 462 are located in the `userscripts.org` domain, while 54 point elsewhere. Most of the `@downloadURL`s use the HTTPS scheme (371) while 145 use HTTP. This data shows that from the 516 scripts with an explicit `@downloadURL` in our dataset, 334 point to `https://userscripts.org` and will automatically update whenever an update is available. For all the rest, the Greasemonkey engine will automatically set the `userscripts.org` domain as the update domain accessible using the HTTPS scheme, meaning that for the vast majority of scripts, updates will be performed silently.

The three most popular third-party domains from which updates are downloaded, both for HTTP and HTTPS, are `github.com` (27 scripts), `zanloy.com` (4 scripts) and `google.com` (3 scripts). The most popular script in our dataset, which updates from a third-party domain over the HTTPS protocol, is an IMDB script with more than 50,000 installations, updating from `https://github.com`.

6. 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. Thus, even if scripts are not malicious by commission, they may still cause harm to their users due to vulnerabilities, by omission.

In this section, we discuss two vulnerabilities that occur in user scripts: DOM-based XSS and overly generic `@include`

Listing 3 Example script vulnerable to DOM-based XSS

```
var d = document.createElement("div");
d.innerHTML = "This page is located at " +
    document.location.href;
document.body.appendChild(d);
```

Listing 4 Resulting div of DOM-based XSS attack

```
<div>
This page is located at http://example.com/?<
    script>alert(1);</script>
</div>
```

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.

6.1 DOM-Based XSS

DOM-Based XSS, is an XSS attack in which a payload is executed that is somehow stored in the DOM of the victim's browser. This is in contrast with reflected or persistent XSS, where the payload is placed inside the visited website.

Consider the example shown in Listing 3, which appends a newly created `div` tag to the loaded webpage and writes the current page's location into it. This code fragment contains a DOM-based XSS vulnerability because it allows an attacker-controlled string to be inserted into the HTML page of the currently loaded website.

If this code fragment is used on `http://example.com/`, a victim's browser visiting `http://example.com/?<script>alert(1);</script>` would generate the HTML code shown in Listing 4. The attacker payload, in this case `alert(1);` would be executed as JavaScript in the `example.com` origin.

The DOM-based XSS vulnerability in the previous example is restricted to the webpage on which the code in Listing 3 is present. Using the same code in a Greasemonkey script potentially lifts this restriction. The vulnerability will then be injected into any page on which the Greasemonkey code is included. An attacker with knowledge of this situation, has thus a much larger target: every page on which this Greasemonkey script is executed, becomes vulnerable.

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 the Parser API [21] in SpiderMonkey [24], Mozilla's standalone JavaScript engine, we parsed all scripts in our dataset and obtained a simplified AST for each one of them. Using the list of sources and sinks listed in Table 3, 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 4. From the 86,358 scripts in our dataset, our analysis revealed 1,736 that contain a DOM-based XSS. The

Sources:	`document`, `document.{baseURI,body, documentURI,forms,links,location, referrer,scripts,title,URL, URLUnencoded}`, `window.name`
	✕
Sinks:	`document.write(`x`)`, `document.writeln(`x`)`, `eval(`x`)`, e`.innerHTML =` x

Table 3: Sources and sinks used in the lightweight static analysis performed to look for DOM-based XSS

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%).

Note that not all sources are under the control of any attacker and might require the ability to place persistent data onto a website. The four sources that can be influenced by an attacker are `document.cookie`, `document.location`, `document.URL` and `window.name`. From the dataset, 101 scripts are vulnerable to DOM-based XSS involving those four sources.

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. The script is designed for a popular massively multiplayer online strategy game called *Ikariam*. We created a proof-of-concept exploit where, through the clicking on a specially-crafted URL, similar to the one used in the example in the previous section, we could inject JavaScript in authenticated pages of users.

6.2 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 the use of a wildcard, and uses regular expression matching 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 might craft 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` [7] 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

	document								window.name	Total
	.body	.cookie	.forms	.links	.location	.title	.URL	*other*		
document.write(x)	0	0	0	0	1	1	0	1	0	3
eval(x)	3	2	1	0	0	0	0	73	0	79
e.innerHTML = x	721	4	5	17	83	14	6	800	5	1,654 (*)
Total	724	6	6	17	84	15	6	874	5	1,736 (*)

Table 4: Breakdown of amount of scripts with detected DOM-based XSS vulnerabilities according to the used sources and sinks. Only those sources and sinks with any results are shown. (*) the totals do not reflect the sum on each row, but rather the amount of total unique scripts for the given sink

between the parts. For instance, in the directive `@match http://*.example.com/about.html` the wildcard is located in the host part and can only match characters associated with a host. Unlike with `@include`, the `http://www.mybank.com/#x.example.com/about.html` URL will not be matched, since `/` and `#` are not valid characters for a hostname. Likewise, the wildcard in `*://www.example.com/` can only match `http` or `https`.

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 5: `@include` and `@match` directive usage, "insecurely" means an overly generic `@include`

Usage of the `@include` *and* `@match` *directives.* Table 5 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 installations. It uses an overly generic wildcard `@include` directive of the form `@include http://*website.com/*`.

6.3 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 an attacker to execute malicious code on any webpage that the attacker chooses, by crafting a specific URL.

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. As explained in Section 2.3, malicious websites can leverage such a DOM-based XSS vulnerability in a user script, to 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, even though is meant to run on a gaming site, 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.

7. 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 and Google Chrome.

Barth et al. [5] criticize the all-permissive Firefox extension system showing that only three out of 25 investigated extensions required full system access. The authors propose an alternative extension architecture that requires extensions to explicitly ask permission for access to resources and also compartmentalize the browser so that a vulnerability in a "benign-but-buggy" extension does not necessarily mean arbitrary code execution with the permissions of the user running the browser process. Guha et al. [12] study the Google Chrome market and show that a significant fraction of extensions request more permissions than they require. The authors set out to create a more fine-grained policy system to describe access to resources, as well as a statically-verifiable, platform-independent language for writing extensions which are then automatically compiled to JavaScript and other platform-dependent code.

Liu et al. [17] remind that next to benign-but-buggy extensions, malicious extensions pose real threats to the security and privacy of users. The authors present some proof-of-concept extensions that send spam emails, steal bank credentials, and perform distributed denial-of-service attacks on demand. As a defense against malicious extensions, the authors propose the use of micro-privileges, such as `inject_script` and `cross_site` in order to further increase the granularity of Chrome's fine-grained policy system.

VEX [3] 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, i.e., achieve remote code execution and download arbitrary malicious executables on the victim's machine. According to a recent study by Barracuda Labs, the visitors of the 25,000 most popular sites on the Internet, got exposed to more than 10 million such exploits, on February of 2012 alone [4]. Due to the great magnitude of the problem, there has been a significant body of research in detecting malicious JavaScript, using honeypots [20], dynamic analysis of JavaScript code [8, 22, 16, 15], and hybrid systems [9, 23] which utilize both static and dynamic techniques to analyze JavaScript code. Purely static analysis of JavaScript has met with limited success due to the large degree of obfuscation that can be achieved in the JavaScript language. As such, purely static techniques [6] are best used as lightweight filters which can separate the "definitely benign" from the "possibly malicious". The latter can be used as input in more resource-intensive dynamic systems while the former can be safely ignored.

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. Thus malicious Greasemonkey scripts are more likely to interact in an abusing way with a specific web application, rather than trying to trigger a vulnerability in the browser. As such, they may be only 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 [19, 25, 29, 1, 13, 18] 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. The vulnerabilities exposed in this paper allow an attacker in some cases to inject malicious code inside the sandbox, causing a situation similar to the "inverse sandbox" effect described in [29].

8. CONCLUSION

As more and more applications move from the desktop to the web, power users turn to augmented-browsing tools, to personalize their web applications.

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 automated malware detection in a script market is difficult because of the context-sensitive nature of malice, and that the review process of `userscripts.org` is ineffective in 78% of the cases. Next to the discovery of malicious scripts, we identify ways in which malicious authors can bypass the community review process and covertly infect user script users with malware.

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 demonstrate this attack through a proof-of-concept exploit for one of these user scripts, installed by over a million users, allowing us to steal their data across all sites.

The purpose of our work is to highlight the inherent difficulties of securing script markets against malicious actors, and the possibility of weaponizing benign scripts against otherwise secure websites.

Responsible disclosure

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

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.

9. 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] Alexa - the web information company. `http://www.alexa.com/`.

[3] 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.

[4] Barracuda Labs. When good sites go bad. `http://www.barracudalabs.com/goodsitesbad/`, 2012.

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

[6] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 197–206, New York, NY, USA, 2011. ACM.

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

[8] 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.

[9] 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.

[10] P. Eckersley. How Unique Is Your Browser? In *Proceedings of the 10th Privacy Enhancing Technologies Symposium (PETS)*, 2010.

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

[12] 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.

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

[14] Internet Archive: Wayback Machine. `http://archive.org/web/`.

[15] 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.

[16] 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.

[17] L. Liu, X. Zhang, G. Yan, and S. Chen. Chrome Extensions: Threat Analysis and Countermeasures. In *Proceedings of the 19th Network and Distributed Systems Security Symposium (NDSS '12)*, 2012.

[18] 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.

[19] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja - safe active content in sanitized JavaScript. Technical report, Google Inc., June 2008.

[20] 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.

[21] SpiderMonkey - Parser API. `https://developer.mozilla.org/en-US/docs/SpiderMonkey/Parser_API`.

[22] 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.

[23] 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.

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

[25] M. Ter Louw, K. T. Ganesh, and V. N. Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *19th USENIX Security Symposium*, 2010.

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

[27] Trend Micro Site Safety Center. `http://global.sitesafety.trendmicro.com/`.

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

[29] 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.