CSP & Co. Can Save Us from a Rogue Cross-Origin Storage Browser Network! But for How Long?

Juan D. Parra Rodriguez
University of Passau
dp@sec.uni-passau.de

Joachim Posegga
University of Passau
jp@sec.uni-passau.de

ABSTRACT

We introduce a new browser abuse scenario where an attacker uses local storage capabilities without the website's visitor knowledge to create a network of browsers for persistent storage and distribution of arbitrary data. We describe how security-aware users can use mechanisms such as the Content Security Policy (CSP), sandboxing, and third-party tracking protection, i.e., CSP & Company, to limit the network's effectiveness. From another point of view, we also show that the upcoming Suborigin standard can inadvertently thwart existing countermeasures, if it is adopted.

CCS CONCEPTS

• **Security and privacy** → **Browser security**; • **Networks** → **Web protocol security**;

KEYWORDS

Web Security; WebRTC; PostMessage; Browser Security; Content Security Policy; Suborigins; Parasitic Computing

ACM Reference Format:
Juan D. Parra Rodriguez and Joachim Posegga. 2018. CSP & Co. Can Save Us from a Rogue Cross-Origin Storage Browser Network! But for How Long?. In *CODASPY '18: Eighth ACM Conference on Data and Application Security and Privacy, March 19–21, 2018, Tempe, AZ, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3176258.3176951

1 INTRODUCTION

Browsers provide access to computational resources such as local storage, network and processing power. This can be helpful for several applications; however, rogue developers can also abuse browser's resources, e.g., for mining crypto-currencies in the browser (crypto-jacking). Meanwhile, the Web security community is engaged in an arms race against attackers exploiting data or session related vulnerabilities such as cross-site scripting (XSS), therefore leaving the browser resource abuse problem unattended.

We discuss how malicious third-party code can make its way into honest Web applications and force their visitor's browsers to join a browser network to store and distribute arbitrary data without letting the browser's user know. Also, we show that some CSP directives, and other mechanisms, can be effective to thwart the browser network attack, even though their rationale protects against other attacks. Further, countermeasures presented are analyzed from the perspective of the two potential victims of a browser resource attack, namely the browser's user visiting the website, and the website (developer or administrator). From another point of view, we show that existing countermeasures are threatened by upcoming standards and can be reversed as their attacker model does not include browser resource abuse scenarios.

2 EXTERNAL SCRIPTS ON THE WEB

Websites commonly instruct the browser to fetch resources from different servers across the Internet. Thus, to cope with various security threats, the browser has an **Origin-based security model** which isolates resources. In most cases, an important factor is whether the code is served by a different Origin[1] than the site visited by the user. For brevity, we call this an *external resource*.

Nonetheless, in the Origin-based security model ***external resources* can inherit the Origin from the page including them** depending on how the resources are included. As shown by rows 1 and 2 of Table 1, sites including external resources with a script tag share their Origin with the resource. Conversely, Iframes obtain their Origin from their source location, i.e., rows 3 and 4 of Table 1, regardless of the Origin of the website including them.

Also, Table 1 shows the Origin assigned to the resource, which determines the Local Storage instance available for it. For example, the Iframe of row number 4 of Table 1 obtains a different storage object than the site including it. On the contrary, scripts included as shown by rows 1, 2 and 3 share the Local Storage instance with the site including them.

Leaving aside how external scripts are executed, **external code can go rogue and abuse a website visitor**. In our case, it is important to mention that abusive external code can be included in a website by a developer either intentionally or unintentionally. Some sites may intentionally sacrifice their reputation by using "unconventional" means to earn money, e.g., crypto-jacking. However, popular sites are not keen to destroy their reputation by intentionally abusing browsers. All in all, there are a number of ways leading to abusive code execution because developers include external code without being aware of the consequences; for example, through CMS widgets[2], advertisements, or JavaScript libraries.

[1]An Origin can be seen as a tuple containing the protocol, host and port for a particular website; for example, the origin for https://a.com/home/ the tuple (https, a.com, 80)
[2]Wordpress removed a plugin from their marketplace, as it abused thousands of browsers to do crypto-jacking without the website's owners and browser's users knowledge. For more information see https://www.wordfence.com/blog/2017/11/wordpress-plugin-banned-crypto-mining/

Row No.	Inclusion code from visited site: code from Origin https://a.com	Same Origin as the visited site	Local Storage Instance
1	`<script src="https://a.com/script.js" ... >`	yes	https://a.com (shared)
2	`<script src="https://b.com/script.js" ... >`	yes	https://a.com (shared)
3	`<iframe src="https://a.com/script.js" >`	yes	https://a.com (shared)
4	`<iframe src="https://b.com/script.js" >`	no	https://b.com (not shared)

Table 1: External resource inclusion (from a site hosted by https://a.com), its Origin and Local Storage

3 THE ATTACK

In addition to inadvertently including malicious external code, developers can introduce XSS vulnerabilities allowing the execution of malicious scripts abusing local storage, *i.e., Abusive Scripts.*

As shown by row 4 in Table 1, the **Origin-based security model** introduced in Section 2 provides Iframes loaded from a different Origin with a Local Storage instance in a different Origin as the site including them. Although this Local Storage separation is beneficial for data isolation, it can be abused when a website includes *n* Iframes from different Origins consuming up to *n* times the Local Storage quota (see Figure 1). This approach has already been used to fill the user's disk[3].

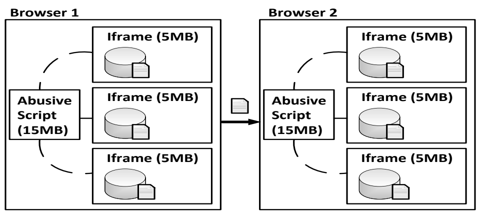

Figure 1: Rogue Storage Network (using 30 MB)

This way of bypassing the storage quota can be extended by using a cross-Origin **inter-window messaging** mechanism, i.e., postMessage represented by dotted lines. This lets each Iframe communicate with the parent window (Abusive Script) allowing it to access their Local Storage to create a database with *n* frames, and therefore using *n* times the Iframe quota.

Furthermore, an attacker can coerce browsers to share information from Local Storage with other browsers executing the Abusive Script without the user's knowledge through **inter-browser messaging communication**, i.e., WebRTC Data Channels. In the case of WebRTC, the Same Origin Policy is not applicable when browsers interact with each other, so an attacker can connect browsers he controls through different sites in a single cross-Origin browser network.

We have previously evaluated the feasibility of the browser network through several experiments with real-life browsers and automation tools in a technical report [1]. We analyzed how visitor return rates and time between visits affect availability of the information distributed across browsers. Further, we also analyzed the amount of traffic exchanged between browsers and servers to conclude that network overhead for servers is significantly less than for browsers.

[3]http://www.bbc.com/news/technology-21628622

4 ATTACKER MODEL

Before defining countermeasures, we analyze possible attack scenarios. Figure 2 depicts how an attacker can use a site (marked with X) to deliver malicious code while making the Origin-based isolation implemented by browsers explicit.

The **first scenario** shows an attacker compromising the website visited by the user. This scenario takes place when the attacker embeds the Abusive Script directly within the Origin of the visited site. This can, for example, happen when an attacker controls a third-party library included using the script tag or when an attacker exploits an XSS bug. The **second and third scenario** show when the Abusive Script included in a "safe" manner, i.e., different Origin than the website being visited. These cases happen when the external content is included in an Iframe, e.g., advertisements. The main difference between attack scenario 2 and 3 is that an attacker could make a site host the Abusive Script or the Iframes performing storage. In practice the attacker can do both in one Origin, but we separate them for the sake of clarity in relation to the possible countermeasures.

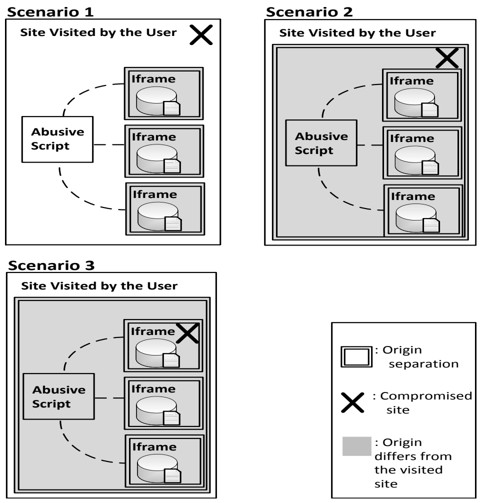

Figure 2: Scenarios based on compromised sites and Origin separations

5 COUNTERMEASURES

The victim most affected by browser resource attacks is the browser's user. However, sites used to host Abusive Scripts without their knowledge can be considered victims of the attack also as their reputation could suffer. Thus, we split countermeasures based on who can take action (browser's user or website developers); further, for each actor deploying a countermeasure, we reference for which attack scenarios from Figure 2 they are protected.

Users can protect themselves against all attack scenarios if they enable their browser's third-party tracking protection. This protects the user's privacy from advertisements by blocking persistence APIs for JavaScript running in Origins different from the site loaded. This removes any persistent storage mechanism available for resources outside of the visited website's Origin, i.e. row number 4 on Table 1. Practically speaking, this measure denies local storage access to any resource with gray background in Figure 2.

Website's developers can use HTTP response headers or HTML keywords for security directives when they include external resources. For instance, the `sandbox` HTML keyword ensures that an Iframe is not allowed to execute JavaScript, unless the `allow-scripts` keyword is used. Moreover, even if developers use the `allow-scripts` keyword, sand-boxed Iframes cannot use their Local Storage instance because each Iframe gets assigned to a random, invalid, origin. As a result, all Origin checks fail and prevent the Iframe from using Local Storage or cookies. By including the `sandbox` directive when including the site containing the Abusive Script, developers can prevent the second and third attack scenarios.

Developers can use the `script-src` and `frame-src` CSP directives in the HTTP response headers to specify which scripts or frames are loaded by a particular site. Thus, a restrictive policy allowing only secure scripts makes it impossible for the attacker to execute his Abusive Script or the storage Frame functionality. If this countermeasure is properly implemented, it would protect the site against the first and second attack scenarios. Although it is less likely, an attacker compromising a website to host the Iframe source (Local Storage part) would not be prevented by the `script-src` directive from including this compromised script in another Origin, i.e., third scenario.

When an attacker has compromised a site and is using it as an Iframe within an Abusive Script, i.e., third scenario, a developer could set the `frame-ancestors` CSP directive in the HTTP response headers to ensure that the site can only be embedded in resources loaded from a list of origins. Therefore, if a security-aware developer specifies a restrictive list of frame ancestors for his site, this would prevent an attacker who has compromised the site from including this particular site as the storage Iframe in the Abusive Script.

6 CONCLUSION

Currently, security-aware users and developers can configure browsers and their websites to avoid being used by a rogue cross-Origin storage network such as the one described in this paper. However, there are three issues that could revert existing advantages. *First*, CSP directives are designed for different purposes than preventing an attack such as the one presented in this paper. For instance, `script-src` and `frame-ancestors` protect against XSS and click-jacking; therefore, there is no guarantee that future countermeasures, e.g., future specifications of CSP, will still protect users against resource abuse attacks. *Second*, the most successful mechanism from CSP, i.e., `script-src`, has faced challenges when retrofitting existing applications [3]; moreover, most of the CSP policies using script-src are easily circumvented today [2]. *Third*, the browser's third-party tracking protection currently prevents all attack scenarios; however, if the Suborigins[4] specification is adopted, attackers can abuse browsers with less effort than today and overcome the third-party tracking protection. The main problem with the Suborigins specification is that it allows a single Origin to define separate Suborigins. Each Suborigin is provided with a Local Storage instance and inter-window communication capabilities. Therefore, an attacker can increase the number of Iframes used to store information on the browser side without constraints. More to the point, separate Suborigins created from the same Origin are not considered third-party content[5], so the third-party tracking protection would not prevent the attack when Suborigins are used. However, this kind of attack requires a more powerful attacker who can set HTTP response headers.

ACKNOWLEDGMENTS

This research has been supported by the EU under the H2020 AGILE (Adaptive Gateways for dIverse muLtiple Environments), grant agreement number H2020-688088.

REFERENCES

[1] Juan David Parra Rodriguez and Joachim Posegga. 2016. *Abusing Web Browsers for Hidden Content Storage and Distribution.* Technical Report MIP-1603. University of Passau. http://www.fim.uni-passau.de/fileadmin/files/forschung/mip-berichte/MIP_1603.pdf

[2] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16).* ACM, New York, NY, USA, 1376–1387. https://doi.org/10.1145/2976749.2978363

[3] Michael Weissbacher, Tobias Lauinger, and William Robertson. 2014. Why is CSP failing? Trends and challenges in CSP adoption. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8688 LNCS (2014), 212–233. https://doi.org/10.1007/978-3-319-11379-1_11

[4]https://w3c.github.io/webappsec-suborigins/
[5]https://github.com/w3c/webappsec-suborigins/issues/43