IS593 Term Project *"Find Universal Cross Site Scripting Vulnerabilities in Brave Browser"*

Antoine RONDELET (20176461) & Khady NGOM(20176440)

ABSTRACT

We study web security mechanisms and Universal Cross-Site Scripting (UXSS), one of the most threatening attack on the web. First, we introduce some background knowledge about web browsers, their role, the companies behind them, their design and their security mechanisms.
We go on to present the UXSS threat and the impacts of such an attack. Then, we study reported UXSS vulnerabilities on the most popular web browsers. From this study, we extract some recurrent patterns found in the different attack scenarios, and conclude about web browser developers' common mistakes.
In a next section, we present our attempts to break the Brave browser and explain our reasoning and our payloads.
We conclude that UXSS vulnerabilities are a major threat to the security of users on the web. The incredible diversity of browsers running on different platforms make UXSS vulnerabilities likely to be found again in the future. Thus, their study should become a major concern for developers and researchers.

Keywords: web security, web browser, Universal Cross-Site Scripting

I. WEB BROWSERS

A. Role

Before introducing our work, we want to define what exactly a browser is and present its intrinsic role.

According to Wikipedia, a web browser (commonly referred to as a browser) is a software application for retrieving, presenting and traversing information resources on the World Wide Web.[1] While the primary role of these pieces of software is to provide an easy way for users to access web resources, most of today's web browsers implement a myriad of features such as the support for secure communications, the possibility to execute Javascript code or even the ability to be extended with plugins, to only name a few.

Around two decades after the *"browser wars"* between Netscape's Navigator and Window's Internet Explorer, the competition in the web browser market is still prevailing. The democratisation of the use of the Web and, more recently, the introduction of HTML5, CSS3, extensive client-side scripting, as well as the widespread use of smartphones and other mobile devices for browsing the web, lead to the apparition of dozens
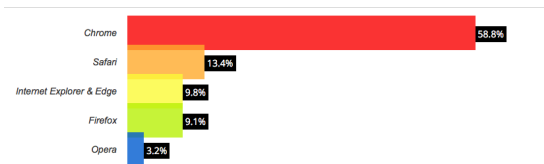

Fig. 1: Web Browser Market Share in October 2017 (from www.w3counter.com)

of browsers. From Mozilla's Firefox[2], Google's Chrome[3], Apple's Safari[4] to Opera[5], for the most famous ones, the number of new browsers never ceases to increase. Today, as we are moving to the 4th industrial revolution - where data is moving astoundingly fast, massively collected and fed into *Intelligent Agents* - some newcomers such as Brave[6] or even Whale[7] are trying to enter this disputed market. They offer fancy and easy-to-use interfaces and new privacy-based features to try to get their slice of the cake. Although, figures 1 and 2 agree on the fact that Google Chrome is the most widespread browser, other companies are working hard and hiring massively to try to dethrone Google's software.
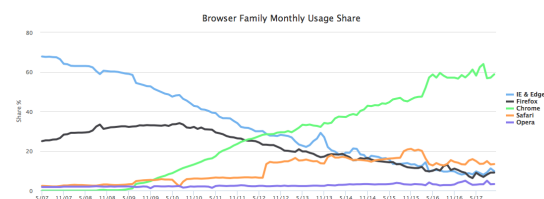

Fig. 2: Web Browser Monthly Usage Share (from www.w3counter.com)

As opposed to Netscape's Navigator, today, most web browsers are free to use. Companies benefit from their browsers by their externalities that help them shape people's computing habits. As the market is quite competitive, developing a web browser can be seen as taking part into a race where the winner is the one who manages to develop the most features. This willingness to be different and attract people's attention often lead web browsers developers to implement features in a hurry. Such frequent releases sometimes unveil

---

[1]https://en.wikipedia.org/wiki/Web_browser

[2]https://www.mozilla.org/en-US/firefox/

[3]https://www.google.com/chrome/index.html

[4]https://www.apple.com/safari/

[5]http://www.opera.com/fr

[6]https://brave.com

[7]http://whale.naver.com/en/

exploitable vulnerabilities and make browsers' architecture more and more complex.

A taste of a web browser's design is provided in the next part of this paper.

B. Architecture

Despite the large variety of browsers available to surf the web, most of them tend to follow a general structure. This architecture, as shown in [4] [5] can be decomposed into eight components, each of which has its very own functionality.

1) User Interface: The component in charge of the UI contains every piece of the browser that is not actually part of the webpage itself. Such piece can either be toolbars or Back and Forward buttons for instance.

2) Browser Engine: The browser engine forms the interface that allows the User Interface to interact and manipulate elements of the rendering engine. It basically marshals actions between the UI and the rendering engine.

3) Rendering Engine: The rendering engine is responsible for producing a graphical representation of the document fetched from a specific URL. It parses and render documents written in HTML and applies styles specified by CSS.

4) Data Persistence: The data persistence subsystem is responsible for insuring the persistence of data when the browser needs it. For instance, a browser needs to save data locally to keep track of the user's bookmarks and cookies.

5) Networking: Since a browser fetches document from the Web, a browser needs to be able to have access to the network. This access is provided by the networking module. It implements file transfer protocols such as HTTP and FTP. Moreover, this subsystem translates between different character sets, and resolves MIME media types for files[8]. In some cases, the networking module might also implement a cache of recently retrieved resources to ensure good performances when a user asks multiple times the same resource.

6) Javascript Interpreter: This module parses and executes Javascript[9] code.

7) XML parser: The XML parser module is in charge of parsing XML documents into a Document Object Model (DOM) tree[10]

8) Display Backend: The display backend subsystem provides primitives for drawing and windows. It also interfaces with the Operating System of the host machine.

While figure 3 shows the general web browser structure explained earlier, it is without saying that each browser has its particularities and its very own version of the aforementioned architecture.

At this point, the reader should be able to have a taste of the complexity of web browsers. Such a compound of subsystems might be likely to expose several vulnerabilities either in the subsystems' implementation or in their interactions.
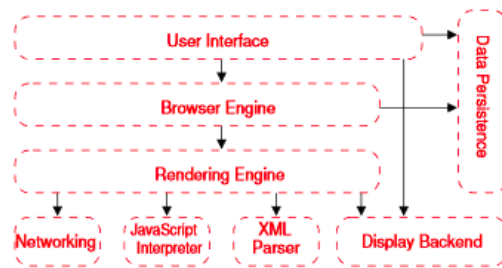In order to get a good appreciation on how the UXSS

Fig. 3: General Web Browser Structure

attack actually work, the next section introduce the security mechanisms used in web browsers.

C. Security mechanisms

While, the Internet is a widespread information infrastructure, the Web is not only used to access and share resources, but is also hosting commercial activities. According to *internetworldstats.com*[11], 51.7% of the world population[12] is now using the Internet on a day to day basis. Albeit, a vast majority of Internet users are undoubtedly honest users, a few of them are malicious and carry out attacks on the network or publish malicious web pages on the web waiting for users to fetch them. In this case where a malicious page is visited by a victim, the attacker's scripts could try to try to retrieve some user's sensitive data.
In fact, in general, many pages tend to be displayed in a user's browser at the same time (into tabs or iframes[13]). Thus, they all "live" in the browser at the same time.

How about using the browser as a bridge for a malicious page to access sensitive data stored in other pages ?

Since browsers are the door that enable us to access web pages from a lot of sources every day, such a scenario would be a major threat for Internet users. Being able to keep browsers safe from such menaces is paramount to keep the web safe. In order to do so, web browsers developers developed very sophisticated security mechanisms to isolate web pages from one another.

Moreover, modern web browsers need to support the latest and constantly evolving web standards. They should be capable of loading resources, while making sure that their level of security remains optimal. In consequence, as explained in [6], web browsers are committed to implement most of the security specifications [7] [8] [9] [10] [11] defined by the World Wide Web Consortium (W3C), such as:

1) Same Origin Policy (SOP): The foundation of browser security mechanisms is the notion of Same Origin Policy (referred to as SOP). Before looking at this important concept, we should first define what an origin is.

In order to access a specific resource, a user has to specify its Uniform Resource Locator (URL)[14]. This web address is a

---

[8]https://en.wikipedia.org/wiki/Media_type
[9]Programming language developed by Netscape in 1995 to make webpages interactive. More details can be found at https://en.wikipedia.org/wiki/JavaScript
[10]https://en.wikipedia.org/wiki/Document_Object_Model

[11]http://www.internetworldstats.com/stats.htm
[12]This represent 3,885,567,619 people
[13]https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe
[14]Reference to a web resource that specifies its location on a computer network and a mechanism for retrieving it.

compound of different elements, as shown on figure 4. The protocol (or scheme) indicates which protocol the browser must use to retrieve the resource. The domain name is an alias for the IP address pointing to the web server that should be requested. The port is a logical construct that identifies a specific process or a type of network service, on the web server that is responsible for serving the web resource. Moreover, the path, parameters and fragment are the path to the resource on the web server, extra parameters provided to the web server and an anchor to a specific part of the resource itself, respectively.

Fig. 4: URL anatomy

With this in mind, defining what an origin is becomes trivial. Looking at the official definition of the W3C[15], an origin is defined by the scheme, host, and port of a URL.

Now that we know what an origin is, the Same Origin Policy can be defined as a policy that ensures that documents retrieved from distinct origins are isolated from each other. Thus, even if users happened to visit malicious web sites, these fraudulent sites would not be able to interfere with the user's session with honest web sites.
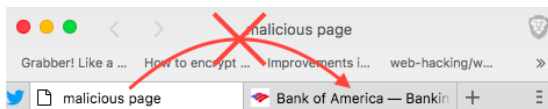
Fig. 5: Same Origin Policy preventing a malicious "tab" to access user's sensitive data

2) Cross-Origin Resource Sharing (CORS): In order to relax the SOP, the W3C specified the Cross-Origin Resource Sharing (CORS) which consists in specifying a white-list of trusted domains allowed to request restricted resources, by extending HTTP with a new origin request header. The CORS mechanism supports secure cross-domain requests and data transfers between browsers and web servers.

Even though, the policy is defined by web server administrators, this security mechanism is used in modern browsers especially on APIs such as *XMLHttpRequest* or *Fetch* to help mitigate the risks of cross-origin HTTP requests and prevent a malicious user from accessing a resource he should not be able to access.

3) Content Security Policy (CSP): The Content Security Policy allows website administrators to define a white-list in a HTTP header, to specify trusted sources for delivering content. That way web authors can control resources the user agent is allowed to load for a given page.

4) Mixed Content Blocking: The mechanism of Mixed Content Blocking blocks insecure content on web pages that are considered secure. A "mixed" content can happen in the case where a user fetches a secure page over HTTPS, containing a HTTP content. Since data transiting over HTTP is not secure and can be eavesdropped, the Mixed Content Blocking mechanism will block this part of the web page. That way the page displayed to the user would be "safe from risks"[16].

5) Subresource Integrity (SRI): Subresource Integrity defines a mechanism by which user agents may verify that a fetched resource has been delivered without being altered. This process allows a User Agent to be sure that the requested resource is the one it received. SRI works by using hash functions. Website developers can associate a hash value with a specific resource. That way web browsers can compute the hash value of the received resource and compare them with the initial value provided by the website. The resource is then displayed only if the hash values are equal, i.e: the resource is valid.

Despite the implementation and support of the security specifications of the W3C, some lower level attacks could be carried out to escape the security mechanisms of today's web browsers. By compromising a browser process and executing arbitrary code, and attacker could either be able to access the host system or bypass the SOP. The next subsections present the concepts of Sandboxes and Process and Origin Isolation. They are used in order to protect users against low level attacks and make sure that SOP would not be "bypassable" at lower levels.

D. Sandboxes

A sandbox is a mechanism which consists in limiting vulnerabilities by isolating components of an application from each other and from the rest of the system. In a sandbox, components run with the minimum access privileges to system resources they need in order to perform their functions. By isolating complex components into sandboxes, web browser developers can limit the damages an attacker

---

[15]https://www.w3.org/Security/wiki/Same_Origin_Policy

[16]This section is placed into quotes because in web security, very few things (not to say nothing) are said to be unconditionally safe.

can cause, and thus, diminish the impact vulnerabilities can have for end-users.

Some additional pieces of information about Google Chrome's sandboxes are provided in [12], and an extract is illustrated in figure 6.

Resource	Access		Resource	Access
Network	Allowed		Network	Blocked
Private networks and loopback	Allowed		Private networks and loopback	Blocked
Port binding	Allowed		Port binding	Blocked
File system	Allowed		File system	Blocked
Registry	Allowed		Registry	Blocked
Process	Allowed		Process	Blocked

Fig. 6: Left: Google Chrome Main Process Sandbox, Right: Google Chrome Render Sandbox

E. Process and Origin Isolation

We studied the mechanism of Same Origin Policy earlier in this paper. This concept restricts access to data across webpages if they have different origins. While this concept is generally implemented at a high level, an attacker able to achieve process-level attacks might be able to bypass the Same Origin Policy. Hence, he would be able to access resources of a different origin hosted in the same process. To avoid such scenarios, developers came up with the idea of hosting each origin in a different process. Preventing direct access between these processes, and applying the same-origin policy in the APIs used by processes for inter-origin communications, enforced SOP at all levels in browsers. The idea here is to ensure a strong site isolation and establish strong security boundaries between web sites [14].
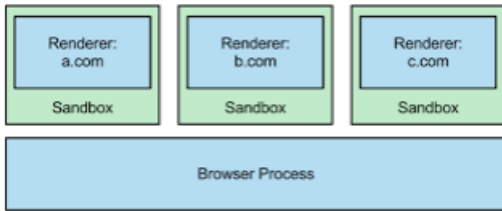

Fig. 7: Site isolation in Chrome

II. UNIVERSAL XSS

Despite all the security mechanisms used in today's browsers, some people still manage to find vulnerabilities that allow them to bypass the same-origin policy. Such vulnerabilities can lead to what is called Universal Cross-site Scripting (UXSS). Before studying some Universal Cross Site Scripting attacks that have been carried out against modern web browsers, we first define the Cross Site Scripting (XSS) attack.

A. "Classic" XSS attack

According to [19], Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into benign and trusted web sites. Since the victim's browser can not know whether the script should be trusted or not, it will execute the script. As a consequence, the injected code is executed at the attacker's target origin. Hence, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and associated with this origin. These scripts can even rewrite the content of the HTML page.

> XSS attacks are injection attacks that exploit vulnerabilities in **web applications**.

In order to carry out a cross-site scripting attack, a malicious user exploit a vulnerability found in the victim web application. Such vulnerability can be a lack of user input sanitization. This allows the attacker to send inline scripts that are going to be executed in the context of the victim origin. Security breaches can also come from regular expressions that could enable a malicious origin to send malicious *postMessages* [20] to a victim website embedded in an *iframe*, for instance.

Since XSS attacks are one of the most popular attacks against web applications, developers are highly encouraged to implement good practices in order to prevent any damage engendered by cross-site scripting attacks. Nevertheless, SOP is implemented by web browsers, so if a vulnerability happened to be found in a web browser, an attacker could bypass every defense mechanism of a web application. Therefore, an attacker would be able to perform malicious payload injections to any web page origin (even in the case where web applications are not vulnerable to XSS).

B. UXSS: A major threat

Universal XSS (UXSS) is a particular type of Cross-Site Scripting that has the ability to be triggered by exploiting flaws in browsers, instead of leveraging the vulnerabilities against insecure web sites.

> UXSS attacks exploit vulnerabilities in **web browsers**.

In UXSS attacks, client-side vulnerabilities are exploited in web browsers or browser extensions, and trigger XSS conditions, which allow malicious code to be executed, despite all protection mechanisms.

The consequences of such an attack are pretty critical. In the case of a successful UXSS, the attacker doesn't just get access to a compromised session on a vulnerable web page, but may get access to any session belonging to web pages currently opened (or cached) by the browser at the time the attack is triggered. UXSS does not need a vulnerable web page in order to trigger, and can penetrate web sessions belonging to secure, well written web pages. Thus, UXSS can create a vulnerability where there isn't one !

While bypassing all the security mechanisms described above in this paper might appear impossible, the attentive reader may remember that web browsers developers are in

a popularity contest. In order to compete, they need to implement numerous features over short periods of time. Even though security features are highly regarded, usability and integration features are usually easier to test and implement, and thus can often be seen as a priority.

Furthermore, UXSS attacks can also be carried out using vulnerabilities in plugins. These make ensuring security of web browsers, an even more difficult task. Since a countless number of extensions are available to users, they all constitute a potential security hole exploitable by attackers.

III. BRAVE

Now that we have defined what UXSS was and studied the consequences of such attacks, we now need to define the environment we used to carry out our project.

In order to conduct our study, we decided to choose the browser Brave[17] which was first announced in January 2016. This free and open-source web browser is based on the Chromium[18] web browser. Brave aims to improve user privacy by blocking website trackers and removing intrusive Internet advertisements, so that fewer data would be shared with advertising customers. One specificity of Brave, is that to make a trade-off with the advertisement incomes, the beta version is testing a rewarding system for publishers, called Brave Payments. According to [22], this system allows users to optionally set a budget that they are willing to donate to the websites they visit. Brave then calculates the percentage assigned to each website through an algorithm and the publisher receives a transfer in bitcoins[19].

Our willingness to carry out our project on Brave was due to the fact that this browser is pretty new (only one year) and still in Beta version. Moreover, the new features included into this browser and the small amount of reported vulnerabilities (see: figure 8) opened the door for more vulnerabilities and made it a good candidate for our project.

A. Environment specifications and software version

Our project has been performed on a MacBook Air running macOS Sierra version 10.12.6, using the Brave version described in figure 9.

B. Widening to other browsers

Despite the fact that we first focused on Brave during this term project, we then decided to broaden our study to the other browsers listed below:

- **Whale** browser by Naver[20]: version 1.0.37.16 (64-bit), released in October 23, 2017
- **Firefox Quantum** by Mozilla[21]: Version 57.0 (64 bits), released in November 14, 2017
- **Safari** by Apple: Version 10.1.2

---

[17]https://brave.com
[18]https://www.chromium.org
[19]https://bitcoin.org/en/
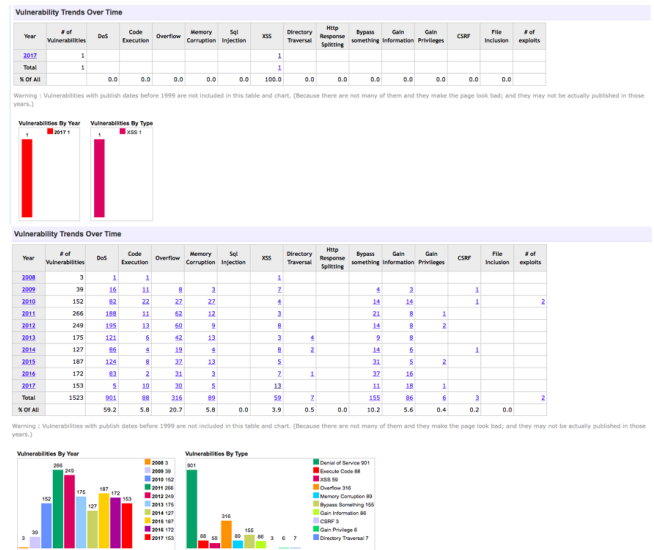[20]http://whale.naver.com/en/
[21]https://www.mozilla.org/en-US/firefox/

Fig. 8: Brave Reported Vulnerabilities (top) compared to Google Chrome Reported Vulnerabilities (bottom) according to cvedetails.com

Fig. 9: Brave version used during our project

IV. CONTENT OF OUR RESEARCH

A. Scope of our study

In the course of the last two months we investigated the causes of Universal Cross-Site Scripting in modern web browsers. As a consequence, we analyzed and studied the reported vulnerabilities that have been discovered on many browsers, in the course of the last years. As a wide panel of platforms are being used in our connected devices nowadays, we decided to eliminate all vulnerabilities reported on mobile platforms such as iOS[22] or Android[23] from our study. Such decision was due to the fact that security mechanisms are not exactly the same in mobile platforms than in laptop operating systems.

---

[22]https://en.wikipedia.org/wiki/IOS
[23]https://en.wikipedia.org/wiki/Android_(operating_system)

In this section of the paper, we try to summarize the attacks we studied, and then, we seek to extract scenario patterns that emerged from this study.

B. Common Vulnerabilities and Exposures (CVE)

In the previous part of the paper we explained how difficult securing a web browser was. Moreover, browser developers have to release new features in small amount of time to stay ahead of the competition. This makes testing an arduous task. Ensuring a full coverage of the software is almost impossible.

In order to prevent malicious users from exploiting potential flaws in web browsers, companies decided to set up vulnerability reward programs, also called *bug bounty programs*[24]. Such scheme consists in giving *"bug hunters"* a financial compensation and a high recognition for the bugs they report. That way, developers cooperate with part of the community and are able to discover and resolve bugs before the general public is aware of them. This helps prevent incidents of widespread abuse.
A majority, not to say all, of reported vulnerabilities have been classified in the Common Vulnerabilities and Exposures system which provides a reference-method for publicly known information-security vulnerabilities and exposures.

Most of the vulnerabilities we studied during this two-month research work are actually extracted from the CVE system.

In total we analyzed a set of nearly 80 vulnerabilities divided up as follow:

- 25 XSS vulnerabilities reported for Mozilla Firefox from 2012 to 2016
- 30 XSS vulnerabilities reported for Google Chrome from 2009 to 2017
- 7 XSS vulnerabilities reported for Internet Explorer and Edge from 2008 to 2016
- 16 XSS vulnerabilities reported for Opera from 2008 to 2013
- 1 XSS vulnerability reported for Brave in 2017

This set of vulnerabilities reported on 5 browsers in a time frame of 8 years, coupled with the disclosed vulnerabilities on web browser plugins, constituted the kernel upon which we based our study. The examination of these reported vulnerabilities helped us understand the different attack vectors used against web browsers. In addition, it helped us understand the security mechanisms that have incrementally been implemented in web browsers in response of these successive attacks.

To get more insight on the vulnerabilities, we took the time to read the discussions the development teams had on the report, during the development of the patches. Not only following the developer's reasoning steps was very interesting, but it also made us realize the strict deadlines also referred to as *"time-to-release"* the teams have to meet.

C. Originality of attack scenarios

The inspection of vulnerabilities revealed an astounding wide range of attack scenarios, all more originals than the others. Such a variety of vulnerabilities is undoubtedly due to the complexity of web browsers and the large amount of possible exploitable flaws. From Cross Origin Drag and Drop [46], to SOP bypass using Reading mode [26], or even exploit of vulnerabilities in plugins [15] [16], attackers are always more and more originals and manage to find breaches that let them execute malicious scripts.

Nevertheless, although we observed a huge variety of attacks, our analysis unveiled some sort of commonalities to multiple successful attacks against web browsers.

D. Patterns of UXSS attacks

An important part of our work has consisted in trying to find patterns in UXSS attacks. Despite the incredible diversity we observed in the CVE vulnerability reports, some vulnerabilities appeared to be exploited more regularly than others.
The following part of this section aims to list the popular attack vectors we found.

Extensions: Browser extensions are plug-ins that extend the features provided by a web browser. Extensions can be downloaded by users and integrated to their browsers to improve the user interface, the security, or for blocking advertisements for instance. Such pieces of software have the right to access everything in the browser. Therefore, one should be really careful when it comes to use extensions. Any malicious plug-in could bypass the SOP and access sensitive data without efforts. To protect their users, many browsers have online stores that allow users to find extensions that have previously been verified. At this point, 3 scenarios arise:

- The user installs a malicious extension from a malicious store: This exposes the user to dramatic attacks and data leaks. The extension developer is able to bypass the browser security mechanisms and access user's private data.
- The user installs a malicious extension from a secure store: This case is less likely than the first case. Companies such as Google deployed mechanisms to detect malicious extensions and prevent them from being published on their store [29]. Nonetheless, in the case where a malicious extension manages to bypass all the checks, the consequences are the same as in the previous case. The user's private pieces of information are totally exposed to the attacker.
- The user installs an official[25] extension from a secure store. This case is the most interesting in our case. In such circumstances the user should be free from risks. However, this is not the case. If the attacker could not manage to find any vulnerability in the browser before, he can now try to find one in the extension. By using this "non-malicious" but vulnerable extension, the user gives the attacker a chance to attack him. Such weaknesses

---

[24]https://en.wikipedia.org/wiki/Bug_bounty_program

[25]By official we mean "non-malicious"

in extensions have been reported in Adobe Reader[30], or Keybase [16] for instance. Finally, vulnerabilities can also be found in the part of the browser responsible for integrating the extensions [31] [32].

Redirects: HTTP redirects are a special kind of HTTP response used for numerous goals such as temporary redirection while site maintenance is ongoing for instance. While this mechanism is, in general, implemented server side, web developers who do not have access over the server, can implement it in their HTML page (HTML redirections) or in their Javascript client (Javascript redirections).
Interestingly enough, carefully timed reloads and redirects can enable an attacker to bypass the SOP [33][26], [34], [35].

Diversity of URIs: In order to ensure good performances, and/or to support some features, web browsers come with numerous URIs (Uniform Resource Identifier). The *data* URI scheme, for instance, provides a way to include data in-line in web pages as if they were external resources. Another example is the *feed* URI scheme that is designed to facilitate subscription to web feeds; or also the *about* URI which is an internal URI scheme implemented in various Web browsers used to reveal internal state and built-in functions. Such a variety of URIs has been widely used by attackers to carry out their attacks. Domainless *about:blank* URIs have been used to bypass the SOP in [36], while *feed:* and *data:* URIs have been used to execute malicious javascript code [37] [38] [39] [40] [41].

XSS Auditor: In order to avoid Cross-site Scripting attacks many web browsers implement a mechanism called XSS Auditor. It uses the *X-XSS-Protection* HTTP header which allows to stop pages from loading when browsers detect reflected XSS attacks. In the case where the header is set to *X-XSS-Protection: 1; mode=block*, once the auditor is triggered the response is blocked and a blank page is shown to the user.[27]. This blocking mode in the XSS Auditor has also been an fairly "popular" attack vector used by attackers. Some scenarios exploited the fact that a page was blocked or not, to infer some secret information on a victim's page or to proceed to a document.referrer leakage [42] [43] [44]

Cross Origin Drag and Drop/Copy and Paste: In HTML5, Drag and Drop has been included as part of the standard. Thus, any element can be set as "draggable", and custom events can be defined by the web developer to refine the behaviour of his page during a drag and drop from the user. However, such a feature can be used in a malicious fashion. Attackers actually found scenarios where cross Drag and Drop were possible and thus lead to XSS conditions [45] [46] [52].

Malicious attributes values for DOM elements: The last method we observed, frequently used to generate XSS conditions in web browsers, was to confuse the browser with malicious names for DOM elements and/or malicious values for DOM elements' attributes.
In such case, the web browser's behavior could be changed and some security mechanisms could be altered.
Attacks that used this technique consisted in redefining global variable (variable clobbering[28]) [48], escaping the browser XSS filters by writing some javascript payloads in the value of some DOM elements [46], or just naming an attribute with the name of a global variable [49] [50] [51]

In addition to these observations, the [28] paper provides an insightful analysis of UXSS vulnerabilities from the *cross-origin JavaScript capability leaks* family. The authors carried out a study on WebKit[29] and tried to detect and exploit when the browser leaked a JavaScript pointer from one security origin to another. They showed that the origins of such vulnerabilities came from the fact that the Document Object Model (DOM) and the JavaScript engine actually enforced the same-origin policy using two different security models: Access Control and Object-Capabilities. While Access Control, used by the DOM, prevents one website from accessing resources allocated to another website, the JavaScript engine, using an object-capability discipline, enforces the same-origin policy by preventing one website from obtaining JavaScript pointers to sensitive objects that belong to a foreign security origin. In their research they showed that the browser leaked a JavaScript pointer from one security origin to another and that this would lead to an exploitable vulnerability to bypass the SOP.

This study could relate with the above section about *Malicious attributes values for DOM elements*. Indeed, in this part we spoke about variable clobbering to prevent some security mechanisms to be triggered, such as frame busting scripts for instance. If, on the other hand, the attacker manages to overwrite some methods from a global Javascript object such as *window* for instance, it would not be foolish to assume that, if invoked inside a victim iframe, this method would execute on behalf of the victim's origin.

Thus, it appears that, depending on the circumstances and the browser, global variables could be seen as *bridges* between origins, and be used by malicious users to attack the web browser.

E. Errors made by developers

After studying the reported vulnerabilities and then extracting some common attack vectors - or patterns - used from them, we noticed some common and easily predictable mistakes made by web browser developers.

These attacks are attacks that could have been "easily" avoided:

- Lack of sanitization of user input: Could lead to script injection in URIs for instance.
- Bad design of regular expression: Could allow a user to bypass some security checks. If the regular expression

---

[26]In the PoC (https://blog.innerht.ml/ie-uxss/) of CVE-2015-0072, the author explains that JavaScript's single-threaded nature is actually used to freeze the browser with an alert box to actually manage to bypass the SOP

[27]https://www.virtuesecurity.com/blog/understanding-xss-auditor/

[28]https://en.wikipedia.org/wiki/Clobbering

[29]https://en.wikipedia.org/wiki/WebKit

is case-sensitive, a user could use upper and lower case letters to short-circuit any checks for instance [52]

- Divergence with specifications: Sometimes the specifications might add a lot of constraints on the developers. It might be tempting to "adapt" them. Nevertheless, such circumstances could create odd situations that could benefit an attacker [53]
- Confusion/Negligence: As browsers support a large panel of extensions, some situations appear when it is difficult to know the party responsible for the implementation of some security mechanisms. In the case where an ambiguity happens, some security checks might be missing, allowing a malicious user to attack the software.
- Inattention: By being focused on developing new features, web browser developers can mistakenly introduce regressions, and vulnerabilities that have already been patched. Thus, an attacker might be able to carry out an old attack on the browser.

Note: This list of potentials errors, is based on the messages exchanged in the bug report pages.

V. ATTEMPTS TO FIND UXSS VULNERABILITIES ON BRAVE BROWSER

In addition to studying the payloads used against modern web browsers, we tried to adapt some of the attacks we observed against the browser Brave[30] in the hope to unveil a UXSS vulnerability. The following section presents our attempts.

Note: For our attacks, we focused on crafting malicious html pages to render, or malicious payloads to inject in the user interface of the browser. Some more refined and advanced attacks, such as some fuzzing against the Javascript interpreter for instance [23], could have been achieved, but they were out of the scope of our project.

A. Proceedings

Executing malicious code in the PDF renderer plugin: As we mentioned earlier, vulnerabilities in plugins can be used in order to bypass some security mechanisms in web browsers. Our first attempt to find UXSS vulnerabilities in *Brave* consisted in trying to exploit plugins. We tried to reproduce and adapt the known vulnerability in Adobe PDF Reader [15] on Brave, which uses PDF Viewer (version 1.9.457). This attempt was, without any surprise, not successful, but it gave us the taste of what a malicious attacker could do to find UXSS vulnerabilities.

```
http://www.axmag.com/download/pdfurl-guide.pdf#<
    maliciousPaylaod>
```

Listing 1: The attack vector we tried to use to execute malicious javascript using the PDF viewer

---

[30]Note that all our attacks were finally widened to the web browsers listed in section III-B. Since all browsers are different, one might have been vulnerable for an attack while others might have not.

Since *Brave* proposes a limited number of extensions, we were extremely restricted. After a couple of research on the supported extensions, and after arguing in finding vulnerabilities, we decided to stop spending a lot of time on extensions. Thus, we moved on trying to attack the browser directly.

Variables clobbering: The next step of our research to find UXSS vulnerabilities consisted in trying to redefine global javascript variables. Indeed, if we managed to redefine a javascript variable such as *window* or *location* for instance, we could be able to either alter the security of the browser (some security checks might fail [48]), or execute our own javascript code in the context of the victim's page [28]. If we redefine the method, of the *window* object for instance, called in a victim page which is embedded in an *iframe*, our malicious code would be executed on behalf of the victim page.
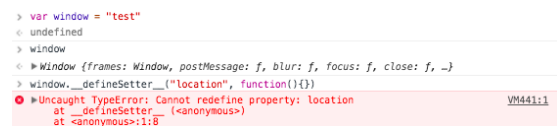

Fig. 10: The browser rejects our attempt to clobber global variables

Figure 10 shows an attempt to redefine the variable *window*. This attempt is not successful. The variable *window* that we create does not shadow the global variable *window*. As a consequence, when we call *window*, our variable is not returned. Moreover, we see on this figure that we cannot redefine the property *location* of the object *window*.

We carried out other attempts, with different payloads, but none of them seemed to unveil a vulnerability.

Exploiting the about: *URI:* In order to support diverse features and in order to manage the settings of the browser, developers use a large panel of different URIs. Among them, the *about* URI is used to reveal internal state and built-in functions. Here, our idea was to use this URI to bypass SOP. Our first objective was to figure out how browsers handled such URIs.
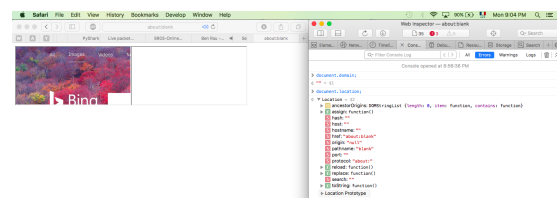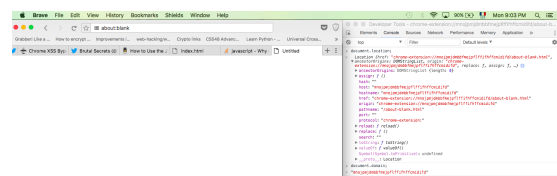

Fig. 11: The origin of *about:blank* in Safari

Fig. 12: The origin of *about:blank* in Brave

As we can observe on figures 11 and 12, in the case of Safari, *about:blank* has no origin (*origin: "null"*). However, in the case of Brave, it has an origin set to some sort of google chrome origin (*origin: "chrome-extension://mnojpmjdmbbfmejpflffifhffcmidifd"*).

Further to this result and some additional research, we figured out that *about:blank* inherits the same security parameters as its caller.

In the case of Safari, we saw that when *about:blank* was opened from the "home page" of the browser, we managed to get a *null* origin. Thus, we tried to use it to see how the browser behaved when a script from a *null* origin tried to access resources of a web page embedded in an iframe. In other words, our goal here was to see whether Safari did strict origin checks or not. To do so, we embedded *bing.com* into an iframe, and tried to execute malicious code in its context.

Unfortunately, even our malicious scripts with a *null* origin couldn't execute in the context of *bing.com*. Nevertheless, we didn't stop here. We tried to define malicious events to see whether they could be executed in the context of our victim *bing.com* web page.

```
// Get the victim iframe
var frame = document.getElementsByTagName('iframe')
    [0]

// Define a malicious event
frame.onmouseover = function() {var script =
    document.createElement('script'); script.type =
    'text/javascript'; var code = 'alert(document.
    cookie);'; script.appendChild(document.
    createTextNode(code)); document.body.appendChild
    (script);}
```

Listing 2: Our attempt to define malicious event on an embbeded victim web page

The idea here was to bind an event to an embedded website in an iframe, and to execute the function of the event every time it happens. For our tests we used the *onmouseover* event, since this event is extremely likely to be triggered by the user. Unfortunately here, the script executes in the context of the parent origin. Thus, we could not access sensitive data from the iframe.

Bookmarking malicious URIs and importing malicious bookmarks: In our "quest" to UXSS vulnerabilities, we tried to challenge all the features we thought about in the browsers. As all browsers provide users with a mean to bookmark the web pages of their choice, we tried to see whether we could find a vulnerability in this feature.

In general, users are provided with a lot of possibilities to manage their bookmarks. They can simply add a page, they can order them and classify them into folders, and they can even export and import them (see figure 13). Knowing this, we were left with many options to find a weakness in the browser.

We first tried to bookmark websites to which we added malicious scripts, as shown on 14. We then tried to enter malicious code in the different fields of the bookmark page
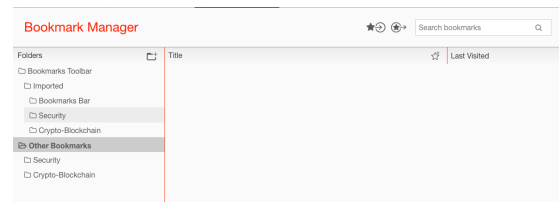

Fig. 13: The bookmark page of Brave

(figure 13). Our idea here was to see if we could be able to run malicious scripts in the context of the browser itself. Indeed, in the case where user input is not correctly handled (not correctly sanitized), the attacker might be able to run javascript with the privileges of the browser.

Fig. 14: A malicious bookmark which points to http://example.com and to which we added malicious javascript in the URL

Our attempts to name bookrmarks folders with javascript code, input javascript in the search bar, and append scripts to URLs before bookmarking them were, once again, not successful.

Bypassing CORS: In our attempt to find a way to bypass the CORS[31] mechanisms, we encountered a lot of difficulties. Thus, we tried to gradually improve our attack in order to finally find an exploitable vulnerability.

The idea we came up with here was to try to set a malicious header in the request, in order to confuse the victim's web server. Indeed, we thought here that if we were able to set *User-Agent*, *Referer* or *Origin* headers, the victim's web server might be fooled, and might send us the web page regardless of our malicious origin. However, it turned out that the browser refused to let us set such HTTP Headers in our requests. Further to this observation, we tried to perform an attack similar to HTTP Response Splitting[32]. What if we could set malicious values to headers? In fact, if we could find a way to change the *User-Agent* indirectly by setting another HTTP header, we would be able to bypass the browser security mechanisms.

Test 1: Bypassing CORS with XMLHTTPRequest

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://www.google.co.kr", true);

// Trying to set a malicious header in our request
var maliciousHeader = "accept-language";
xhr.setRequestHeader(maliciousHeader, "value\%0AUser
    -Agent: curl/7.54.0");

xhr.onreadystatechange = function () {
    if (xhr.readyState == XMLHttpRequest.DONE) {
        if (xhr.status != 200 && xhr.status != 206)
            {
```

---

[31]Cross-Origin Resources Sharing. See section
[32]https://www.owasp.org/index.php/HTTP_Response_Splitting

```
            console.log("Oops ! Something went wrong
                 :( (Status: " + xhr.status + ",
                Response: " + xhr.responseText + ")"
                );
        } else {
            document.getElementById("tFrameWindow").
                innerHTML = xhr.responseText;
        }
        return;
    }
};
xhr.send();
```

Listing 3: Trying to set malicious headers with
XMLHTTPRequest

The idea in the code represented above was to introduce a line break in the header value (%0A is the URL encoding value of the line break[33]) in order to create a malicious *User-Agent* header. That way we would be able to fetch the victim's page. Unfortunately as shown on figure 15, the result was not the one we expected.

Fig. 15: The resulting request headers from our attempt to fake the User-Agent

In the attempt shown above, we tried to set the value of the http header *accept-language* which is a standard HTTP header. However, we know that the *Access-Control-Request-Headers* HTTP header is set in the preflight[34] request when one tries to fetch a cross origin resource. Thus, instead of setting a maliciously crafted header value, we tried to set a malicious custom header name that would be then used as a value for the *Access-Control-Request-Headers* HTTP header (figure 16)

As we can see on figure 16, our attempt to set malicious headers was not successful. It seems that the specifications[35] have been well implemented in the browser. Thus, no vulnerability seems to be usable here to attack the software.

Note that many more headers and names values have been tested to try to bypass the security checks of the browser.

[33]We used the URL encoding line break since headers name and values have to be composed of ASCII characters. Thus, we couldn't use \n nor \r for instance

[34]https://developer.mozilla.org/en-US/docs/Glossary/preflight_request

[35]http://tools.ietf.org/html/rfc2616#section-4.2

Fig. 16: The result of our attempt to set malicious custom header names

Moreover, we also tested the attacks aforementioned using the *Fetch* API[36] with *cors* and *no-cors* modes, however, the results were the same.

Test 2: Bypassing CORS with a Web Worker
After trying to bypass cross origin resource sharing security mechanisms using *XMLHTTPRequest* and *Fetch*, and after studying the CVE-2013-1714, we decided to learn more about web workers and see whether we could use them to fetch a cross origin resource. According to [54], web workers are simple mean for web content to run scripts in background threads. The worker thread can perform tasks without interfering with the user interface.

Thus we tried to see whether a request initiated from a web worker was ruled by the same constraints as "classical" requests, and see whether we could use these workers to attack the browser. The principle of our attack consisted in delegating the *XMLHTTPRequest* to the worker thread a see whether we could fetch a cross origin resource[37]. As shown of figure 17 the request was initiated by our worker (which tried to set malicious headers), but unfortunately the result was the same as the previous attempts presented above.

Fig. 17: The request initiated by our web worker
(blob:null/a036ed6f-601c-433a-994c-ed0e43c28498:26)

Test 3: Bypassing CORS with a Web Worker and a Service Worker
Finally, after all our attempts and some further research, we tried to use a service worker as a proxy of our requests. Such a service worker would intercept all our requests and would modify them in a malicious fashion in order to bypass the CORS security mechanisms.

[36]https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
[37]Our code for this attack scenario can be found at: https://github.com/KAIST-IS593-WEBSEC/UXSS-Vulnerabilities-Project/blob/develop-report/playground/corsBypassingWithWebWorkers.html

Indeed according to Mozilla's web page, service workers essentially act as proxy servers that sit between web applications, and the browser and network (see figures 18 and 19). They are intended to enable the creation of effective offline experiences, intercepting network requests and taking appropriate action based on whether the network is available and updated assets reside on the server.

With this in mind, we tried in this scenario to implement a service worker that would intercept and unmarshal all the requests, and modify them with malicious information (fake/malicious headers for instance) before marshaling them again and executing them.

Fig. 18: The service worker intercepting and replaying the worker request as it is
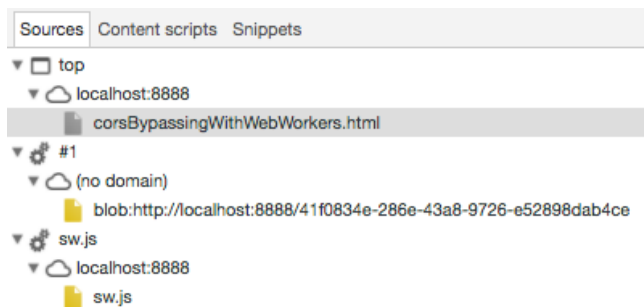

Fig. 19: Our page with the service and the web worker

In order to carry out this attack, we needed to deploy our files on a web server which served pages for the domain *localhost* (with port 8888 in our case). The code of the service worker has been written in the *sw.js* file which is partially presented in the listing 4

```
self.addEventListener('fetch', function(event) {
    event.respondWith(
        modifyRequest(event.request)
    );
});

function modifyRequest(request) {
    serialize(request).then(function(serialized) {
        deserialize(serialized).then(function(
            request) {
            // Send the malicious/modified request
            return fetch(request);
        });
    });
}

function serialize(request) {
    // Create our "own" headers
    var headers = {};
    for (var entry of request.headers.entries()) {
        headers[entry[0]] = entry[1];

        // Add to existing headers, our malicious
            data
        headers["User-Agent"] = "curl";
        headers["Origin"] = "google.com";
```

```
        headers["TestHeader"] = "testValue";
    }
    var serialized = {
        url: request.url,
        headers: headers, // Add our headers to the
            request that will be sent to the web
            server
        method: request.method,
        mode: request.mode,
        credentials: request.credentials,
        cache: request.cache,
        redirect: request.redirect,
        referrer: request.referrer
    };
    if (request.method !== 'GET' && request.method
        !== 'HEAD') {
        return request.clone().text().then(function(
            body) {
            serialized.body = body;
            return Promise.resolve(serialized);
        });
    }
    return Promise.resolve(serialized);
}

function deserialize(data) {
    return Promise.resolve(new Request(data.url,
        data));
}
```

Listing 4: Trying to set malicious headers with a "malicious" service worker

In the code above (listing 4)[38], we tried to execute requests from isolated threads in order to see how the browser handled such cases in term of SOP. The goal here was to see how these workers were considered, whether they had no origin, and if so whether we could use it to bypass the security mechanisms. In this case, we can see on figure 19 that the service worker's origin is also *localhost:8888*. However, despite this, we wanted to see how the browser reacted when we tried to actually modify the content of the request in order to set malicious headers to trick the victim web server. If, despite the origin associated with our service worker, the browser allowed us to modify the requests and replay them, then we would have a full bypass of the CORS security mechanism. This would lead to dramatic vulnerabilities. Indeed, if one manages to bypass the CORS mechanism, this means two things:

- Our request would be executed, and with a good setting of headers, we should be able to fool the victim's web server such that it sends us the web page we want to embed in the HTTP response.
- Upon the reception of the response, we should be able to use our service worker to capture the response first, and modify it by injecting malicious code into it, before it is actually forwarded to the renderer and displayed to the user. At this point, the attacker has an immense power over the victim since he can inject malicious code in the rendered page that will execute on the context of the victim web page's origin, leading to a full bypass of SOP.

This being said, we tried to carry out the attack scenario described above, using the code listed in 4. Unfortunately, after

[38]Our code for this attempt is derived from the answers provided in: https://stackoverflow.com/questions/43813770/how-to-intercept-all-http-requests-including-form-submits

numerous attempts, we couldn't manage to have this attack work as expected.

B. Results and difficulties encountered

In the previous section, we presented our attempts to find UXSS vulnerabilities in the browser Brave[39]. Considering that all of these experiments have proven fruitless this shows that today's web browsers do not seem to be trivially vulnerable. Nonetheless, our misfortune in finding flaws on the web browsers does not mean that they are totally secure. Some vulnerabilities might exists and might be found with further research.

Looking back at the "project proposal" we did a few months ago, the results are pretty different from what was expected. This is due to multiple factors:

- The diversity of vulnerabilities, and the astounding number of different payloads we found: Such diversity made it very difficult for us to build an automated tool to generate potential UXSS payloads. Indeed, as we have shown in this paper, a web browser is an extremely complex piece of software which can be attacked in a countless number of ways.

- Our "lack" of knowledge of web technologies (and javascript) at the beginning of the project: We decided to carry out such a challenging project in order to learn about modern browsers and the security mechanisms that protect us when we browse the web every day. From this point of view, the result is extremely positive. We read a lot, we tested a lot, and we learnt a lot about web security along this project. Nevertheless, this initial poverty of knowledge slowed considerably our progression. Before trying to find vulnerabilities we had to read many documents and reports. In order to have a sufficient basis of knowledge that would allow us to finally come up with our own payloads and ideas, we had to know much more about browsers. A seasoned web security engineer would have had a lot more knowledge about the latest features and latest tools that could be explored to find a flaw in the browser. As an example, we learnt about *web workers* and *service workers* at the very end of the project. And we may still not know many features of web browsers that could have potentially led to UXSS vulnerabilities.

- The diversity and the complexity of web browsers: This point echoes the first ideas of this list. Indeed, today web browsers are extremely complex. Thus trying to attack them without prior knowledge is a pretty tough endeavor. Once again the purpose of this project was to learn about browsers, but having sufficient prior knowledge about modern browsers implementation and architecture would have undoubtedly diminished our workload.

-

Despite the difficulties encountered and the absence of vulnerabilities found, we believe that this project has been

[39]Note that the entire list of our attempts can be found on the Github page of our project: https://github.com/KAIST-IS593-WEBSEC/UXSS-Vulnerabilities-Project/tree/master

extremely interesting. From personal points of view, we learnt a lot. From web browser's architecture, to web security and web programming, this term project exposed us to almost all the notions we studied in the IS593 course. This was also the occasion to be challenged by a "real life" attack scenario, and the occasion to read about previous attacks. The diversity of attacks we studied coupled with the study case taught in IS593 during the term, gave us a taste of what was possible and some lessons about how to attack a system. Being students in information security, such knowledge is definitely extremely valuable and will, without any doubt, play a role when we will have to conceive secure systems by ourselves.

Moreover, from a broader perspective, we noticed that the number of resources about Universal Cross-Site Scripting was pretty limited on the web. We found numerous papers about web browsers security mechanisms, however, the number of papers studying ways to bypass these mechanisms can be counted on one hand. As we mentioned earlier in this paper, most of UXSS vulnerabilities are reported as part of *Bug Bounty Programs*, and if some of them are reported by researcher teams, a majority of such vulnerabilities are reported by passionate people all around the world. That is the reason why, our project could be considered as an invitation to study UXSS in a more academic way. Despite the lack of tangible results, this work could be used as a basis to build further research upon.

VI. CONCLUSION

We saw that today, many web browsers coexists. Thus, the developers maintaining these pieces of software need to develop new numerous features in small amount of time, in order to stay competitive.

Today, the most used web browser is Google Chrome, and many companies are pushing forward with fast development pace to catch up with Chrome. While most of these browsers are free, making sure to have a high number of internet users is a fundamental matter for companies. Indeed, browsers help big companies shape the way people browse the web, and thus prove to have tremendous impact for these firms.

Since we use the internet to achieve sensitive operations, on a daily basis, web browsers play a crucial role in keeping the web safe. They are the door that protect us from the wild and insecure world that is the web. As a consequence, developers need to respect the specifications defined by the W3C, and implement suitable security mechanisms. However, we saw that a tradeoff had to be found between developing features dedicated to improve user's experience and ensure user's security. Such compromise, coupled with the high complexity of web browsers design open a breach for attackers. These flaws can lead to what we called Universal Cross-Site Scripting (UXSS). Moreover, we showed how critical such vulnerabilities could be. In fact, UXSS flaws would provide an attacker with the ability to bypass security mechanisms of the browser, and thus access users' sensitive data despite all protection mechanisms implemented by web application developers.

Finding ways to mitigate such vulnerabilities is thus paramount to keep the web safe. Nonetheless, we showed that protecting a complex software like a browser was extremely hard. That is the reason why companies created *Bug Bounty Programs* which aim to remunerate people who report found vulnerabilities. The reported vulnerabilities are then stored in a database called Common Vulnerabilities and Exposures (CVE) maintained by the MISTRE. This database acts as a record of vulnerabilities, and was the basis we carried out our project upon. The analysis of this database of vulnerability provided us with a myriad of pieces of information, and echoed the different notions studied in IS593 this term. From clickjacking, XSS, CSRF, UI timing attacks, or even browser extensions vulnerabilities, everything has been used to defeat browsers mechanisms.

After studying number of reported vulnerabilities in the CVE, we proposed to extract recurrent patterns used in attacks against different browsers. Furthermore, based on the reading of the core development teams' messages on the bug reports, we worked on pinpointing some common mistakes done by developers that led to vulnerabilities in their software. The next step of our work consisted in crafting our own payloads and finding some potentially dangerous attack scenarios to find UXSS vulnerabilities in the Brave browser. At this point, even after a couple of weeks of studying CVE records and reading online sources, our knowledge basis restrained us and slowed us down. None of the attacks we attempted worked, and we are persuaded that we just scratched the surface of what was possible. Further these results, the outcome of this research has been quite positive. We learnt a lot about cutting edge web security mechanisms and about the web technologies. Furthermore, we studied plenty of attack scenario which truly helped us deeply understand not only the notions broached in IS593, but also on how to attack a software and about the common mistakes developers tend to make.

Looking at the current situation and harsh competition in the sector of web browsers, it would not be foolish to assume that UXSS vulnerabilities are still going to be found and reported. We focused our research on laptop browsers, but it worth mentioning that plenty vulnerabilities have also been found on iOS and Android platforms.
Being aware of such risks is a first step toward more security mechanisms. Even if UXSS vulnerabilities are found in browsers, web applications developers should do their best to follow up the news in security, and keep their web applications secure. The simple fact to set HTTPOnly cookies can make attacker's endeavors much more difficult than expected.

VII. FURTHER RESEARCH

As we spoke about the small amount of academic work about UXSS in this paper, we consider our research as an invitation for more in-depth work about this type of vulnerability. This paper could constitute a basis to carry out further work.

Finally, we showed that UXSS vulnerabilities were critical to keep the web secure. Nevertheless, some other attacks threaten the web as we know it. During our research we studied other sort of attacks that constitute real threats for our privacy on the web. The first paper we wanted to mention is called "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice" and has been written by a team of French and American researchers. This research put the accent on the danger to keep using export-cryptography (DHE_EXPORT) to secure our communications. In addition of explaining their attack (Logjam) used to compute a solution of the Diffie-Hellman Problem[40] in "real time", they explained how dangerous weak keys and how misconfigurations could potentially help states decrypt our HTTPS traffic, and access our personal data. This paper has a fundamental impact since it shows the gap between cryptographers and web server administrators, and pinpoints how misunderstandings can compromise people privacy on the web.

In addition of our research, we also studied how Side-Channel information leakage could actually be huge threats for our privacy on the web. The first paper "Side-Channel Leaks in Web Applications: a Reality Today, a Challenge Tomorrow" by Shuo Chen et. al. and published in 2010 studied how side-channel information such as packet size and packet timestamp could provide an attacker with valuable clues on what the user is doing. The main idea underlined in this paper is that an eavesdropper is able to infer what actions the user undertakes on a web page, despite encryption. Unlike the paper presented above, the idea here is not to "break" encryption but to use other "observable" sources of information.
Last, the paper "Loophole: Timing Attacks on Shared Event Loops in Chrome" by Pepe Vila et. al., published in 2017 exploit a information leak from the shared event loops in Google Chrome to infer the activity of the user on the web.
More than just describing specific attacks, these two papers raise a fundamental idea: not only web browsers should implement security mechanisms and security checks to prevent UXSS attacks from happening, but also, they should protect their users from any kind of Side-Channel attacks.

We see here that a great part of the security on the web depend on web browser developers. Since their services are used by billions of people every day, they should put the emphasis on implementing all the security mechanisms necessary to protect the end-users. However, we just saw in this section that attackers could benefit from misconfigurations of web server administrators to break a weak encryption and access people's sensitive data. Lastly, one can also use some "side-channel" sources of information to infer the actions of a user on a web page, and thus, acquiring sensitive data about the user.
In conclusion, we notice that, keeping the web secure, and protecting the users' privacy is a task that has to be handled by all actors in the industry.

REFERENCES

[1] https://www.brokenbrowser.com/uxss-edge-domainless-world/
[2] MJ Rees, *Evolving the Browser Towards a Standard User Interface Architecture*, 2001

[40]https://en.wikipedia.org/wiki/Diffie-Hellman_problem

[3] W3C, *Document Object Model (DOM)*, https://www.w3.org/DOM/

[4] Alan Grosskurth, and Michael W. Godfrey, *A reference architecture for web browsers*, 2006

[5] Tali Garsiel and Paul Irish, *How Browsers Work: Behind the scenes of modern web browsers*, 2011

[6] Christoph Kerschbaumer, *Enforcing Content Security by Default within Web Browsers*, IEEE, 2016

[7] W3C, *Same Origin Policy (SOP)*, https://www.w3.org/Security/wiki/Same_Origin_Policy

[8] W3C, *Cross-Origin Resource Sharing (CORS)*, https://www.w3.org/Security/wiki/CORS

[9] W3C, *Content Security Policy (CSP)*, https://www.w3.org/TR/CSP/

[10] W3C, *Subresource Integrity*, https://www.w3.org/TR/SRI/

[11] W3C, *Mixed Content*, https://www.w3.org/TR/mixed-content/

[12] Markus Vervier, Michele Orr, Berend-Jan Wever, Eric Sesterhenn, *Browser Security White Paper*, X41 D-SEC GmbH, 2017

[13] Mario Heiderich, *Browser Security White Paper*, Cure53, 2017

[14] The Chromium Projects, *Site Isolation*, https://www.chromium.org/developers/design-documents/site-isolation

[15] Ofer Shezaf, *The Universal XSS PDF Vulnerability*, OWASP IL Chapter leader

[16] *Universal Cross-Site Scripting in Keybase Chrome extension*, https://hackerone.com/reports/232432

[17] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell and Dawn Song, *Towards a Formal Foundation of Web Security*

[18] Nataliia Bielova, *Survey on JavaScript security policies and their enforcement mechanisms in a web browser* The Journal of Logic and Algebraic Programming, 2013

[19] OWASP, *Cross-site Scripting (XSS)*, https://www.owasp.org/index.php/Cross-site_Scripting_(XSS), 2016

[20] Son. et al. *The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites*, NDSS, 2013

[21] Acunetix, *Universal Cross-site Scripting (UXSS): The Making of a Vulnerability*, https://www.acunetix.com/blog/articles/universal-cross-site-scripting-uxss/, 2014

[22] Wikipedia: Brave, *Brave (web browser)* https://en.wikipedia.org/wiki/Brave_(web_browser), 2016

[23] Microsoft Offensive Security Research team, *Browser security beyond sandboxing*, https://blogs.technet.microsoft.com/mmpc/2017/10/18/browser-security-beyond-sandboxing/, 2017

[24] Mozilla, *X-XSS-Protection*, https://developer.mozilla.org/fr/docs/Web/HTTP/Headers/X-XSS-Protection, 2017

[25] CVE-2013-2849, *Cross-Origin Copy & Paste / Drag & Drop allowing XSS*, https://bugs.chromium.org/p/chromium/issues/detail?id=171392, 2013

[26] Manuel Caballero *SOP bypass courtesy of the reading mode (Edge)*, https://www.brokenbrowser.com/sop-bypass-abusing-read-protocol/, 2017

[27] CVE-2014-1747, *UXSS from a local MHTML file*, https://bugs.chromium.org/p/chromium/issues/detail?id=330663, 2014

[28] UC Berkeley, *Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense*

[29] Google, *Trends and Lessons from Three Years Fighting Malicious Extensions*

[30] CVE-2011-2107

[31] CVE-2015-7187

[32] CVE-2017-5020

[33] CVE-2015-0072

[34] CVE-2010-4045

[35] CVE-2009-3013

[36] Manuel Caballero, *SOP bypass / UXSS - Adventures in a Domainless World (Edge)*, 2017

[37] CVE-2012-1966

[38] CVE-2012-1965

[39] CVE-2011-2609

[40] CVE-2012-6463

[41] CVE-2010-2665

[42] CVE-2017-5045

[43] CVE-2013-0909

[44] CVE-2013-2848

[45] CVE-2016-5226

[46] CVE-2013-2849

[47] CVE-2012-0455

[48] Gustav Rydstedt, Elie Bursztein, Dan Boneh and Collin Jackson, *Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites*,

[49] CVE-2012-4209

[50] CVE-2012-3994

[51] CVE-2012-4194

[52] CVE-2012-0455

[53] CVE-2012-3985

[54] Mozilla, *Using Web Workers*, https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers,