Advanced UXSS Analysis

Haviv Cohen
Advanced UXSS Analysis

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Haviv Cohen

Submitted to the Senate of the Technion — Israel Institute of Technology
Kislev 5782 Haifa November 2021
This research was carried out under the supervision of Prof. Roy Friedman and Dr. Nethanel Gelernter, in the Faculty of Computer Science.

Acknowledgements

I would like to thank my advisors, Prof. Roy Friedman and Dr. Nethanel Gelernter, for guiding the research, their attention and support throughout the entire journey.

To my beloved wife, Linoy, who helped me during my studies and research, gave me strength and pushed me to finish my master degree.

To my parents, who taught me from day one to fulfill my dreams and learn in the Technion, and to do so with dedication, vigor, and joy.
# Contents

List of Figures

Abstract

1 Introduction

2 Model and Preliminaries

2.1 The Cross-Site Attacker Model
2.2 Same Origin Policy
2.3 Execution Context
2.4 Chrome Terminology

3 Related Work

3.1 UXSS exploits
3.2 UXSS Classification

4 UXSS Purpose’s Formulation

4.1 Introduction
4.2 Understanding UXSS exploit’s purpose

5 Classification of UXSS Exploits

5.1 Introduction
5.2 Class A - Controlled Scriptable Resource
5.2.1 UXSS Examples of Class A
5.3 Class B - Exploit Cross-Context JS Execution APIs
5.3.1 Subclass B-1 - JS Execution API For Same Origin Documents
5.3.2 Subclass B-2 - JS Execution API For Extensions
5.3.3 Subclass B-3 - JS Execution API For Browser’s Components
5.4 Class C - Exploit Same-Context JS Execution Methods
5.4.1 Subclass C-1 - JS Execution Methods of JS Functions
5.4.2 Subclass C-2 - JS Execution Methods of the Document Element
5.5 UXSS Classification Summary
5.6 Classification Evaluation
List of Figures

2.1 Chrome’s Components .................................................. 9

5.1 openURL’s first exploitation technique ............................. 22
5.2 openURL’s second exploitation technique ......................... 24
5.3 Chrome’s UXSS exploits’ distribution by classes ................. 31
5.4 Safari’s UXSS exploits’ distribution by classes .................. 33

6.1 Chrome’s UXSS exploits’ distribution by vulnerabilities usage ...... 35

8.1 Evaluation of Site Isolation - Summary ......................... 42
8.2 New attack of Chrome on Android’s Site Isolation .......... 46
Abstract

There are many types of security vulnerabilities and exploits that utilize them, and most of them are well studied. Yet, a family of severe security exploits called Universal Cross-Site Scripting (UXSS) has been hardly explored and the foundation required to study them has not been formulated. In this thesis, we focus on this family of exploits.

A UXSS exploit enables the attacker to execute a controlled script in the context of any cross-origin service. UXSS exploits focus solely on the browser implementation and thus bypass any XSS protection implemented in the service’s server-side. Compared to other well-studied exploits, there is neither classification nor basic knowledge about what makes UXSS exploits possible. Because of this, the mitigation techniques implemented in browsers against these exploits are ineffective and inaccurate.

In this thesis, we map the factors that influence the existence of UXSS exploits and achieve a better understanding of them. Analyzing UXSS exploits can be challenging and time-consuming. But, using the results of this research, this process becomes more efficient and much easier. Moreover, we used it to evaluate Site Isolation, which is Chrome’s main mitigation against UXSS exploits. As a result, this research builds the foundations for handling UXSS exploits and other logical browser vulnerabilities.
Chapter 1

Introduction

Cross-Site Scripting [1] (XSS) is a type of web application attack that aims to inject malicious client-side code into the content of trusted websites. It is the most common attack type in web applications [2] and exposes end-users to a range of risks, e.g., sensitive data disclosure, account hijacking, and credentials theft [3]. This attack uses vulnerabilities that depend on input filtering bugs in the trusted website’s code that make the attacker’s code injected and executed in the trusted website’s privileged context. For that reason, XSS exploits are strongly related to the implementation of the trusted website and therefore are service-dependant.

Another attack type that is much less known is Universal XSS (UXSS). Similar to XSS, the UXSS attack type aims to inject malicious code into the content of trusted websites, but in contrast to the former, it utilizes vulnerabilities in the browser’s implementation instead. While XSS exploits make the trusted website inject the malicious code, UXSS exploits inject the malicious code by bypassing security mechanisms implemented in the browser. The term “universal” is used because UXSS exploits are browser-dependent, and thus one UXSS exploit can affect all the domains vulnerable to XSS while surfed by the vulnerable browser.

Consequently, UXSS attacks are more dangerous than XSS attacks, which is why UXSS exploits are considered high-severity exploits [4].

While XSS attacks were researched at the academic level [5–7], the UXSS attack was never academically studied and was barely studied at all. Moreover, a terminology was developed to discuss other high-severity exploits (i.e., the vulnerability types used by them have names and were defined), and the bugs that cause them are known. This is not the case for UXSS exploits. It is crucial to fully understand the exploits that are used by an attack type to protect against it. Therefore, without knowing the bugs and vulnerabilities that are used by UXSS exploits, it will be ineffective to study the causes of UXSS exploits and to prevent UXSS attacks from happening.

The irony is that although UXSS attacks pose a bigger threat than XSS attacks, they were less studied and their mitigation techniques are less advanced. This might be due to the complexity and the efforts that are required to find UXSS exploits, compared
to the simplicity of XSS exploits. To compare this absurd situation about UXSS, let us consider other famous exploits types like XSS and Remote Code Execution \cite{8} (RCE) that were studied at the academic level.

XSS exploits are the result of 3 vulnerabilities types \cite{9}: Reflected XSS, Persistent XSS, and DOM-based XSS. These vulnerabilities depend on input filtering’s bugs in the service’s code. Furthermore, there have been many research studies regarding the prevention of XSS exploits; for example, PathCutter \cite{5} with a server-side approach, Document Structure Integrity \cite{6} with a client-server-side approach, and even a prevention method with a machine learning approach \cite{7}.

RCE exploits use vulnerabilities that can be classified into several types: Stack Overflow \cite{10}, Heap Overflow \cite{11}, Format String \cite{12}, and Use-After-Free \cite{13}. The bugs that cause each of these vulnerability types are well-known, e.g., the vulnerabilities types of Stack Overflow and Heap Overflow can be caused by bugs of missing length checks. Moreover, there have been research studies to explore the exploitation techniques used with each of these vulnerability types; for example, Jump-Oriented Programming \cite{14} (JOP) for Stack Overflow, Heap Spraying \cite{15} for Heap overflow. These vulnerability types and their exploitation techniques have been widely researched and consequently, there are multiple methods for detecting and preventing them: Stack Cookie \cite{16} for Stack Overflow, HeapShield \cite{17} for Heap Overflow, FormatGuard \cite{18} for Format String, and DANGNULL \cite{19} for Use-After-Free.

A high-severity exploit type should be researched according to 5 steps:

1. Formulating the exploit type’s purpose.
2. Analyzing exploits that suit this definition and classifying them into well-defined classes.
3. For each class, defining the vulnerabilities types and the exploitation techniques used by its exploits.
4. For each vulnerability type, analyzing the bugs that cause it.
5. Inventing methods to mitigate this exploit type. A mitigation method can be designed for a specific vulnerability type or exploitation technique to improve its effectiveness.

These steps are necessary and crucial to fully understand the exploit type and to decrease its prevalence.

In the case of UXSS exploits, there were only a few research studies about finding them \cite{20–22} and only one \cite{23} about preventing a specific vulnerability that was used by a UXSS exploit \cite{22}. Furthermore, until recently \cite{24}, there were none for understanding and classifying them.

Undoubtedly, those steps were not taken in the case of UXSS exploits. Consequently, it is unclear what are the causes for UXSS exploits, i.e., what kind of vulnerabilities are used by UXSS exploits and what exploitation techniques are needed to use them. Moreover, a UXSS exploit is usually classified as such based on its outcome.
(which is an XSS caused by a browser’s code) but not based on a deep understanding of what bugs make these vulnerabilities possible.

**Our Contributions:** In this thesis, we map the factors that influence UXSS exploits and achieve a better understanding of what makes these exploits possible. Therefore, this research builds the foundations for handling UXSS exploits and other browser logical vulnerabilities. As mentioned before, in most cases, each UXSS exploit applies to one browser only, and thus we chose to focus on Chrome, which is the most popular browser nowadays [25].

In our research, we first formulated the UXSS’s purpose to properly define which exploits can be considered UXSS exploits. Using this definition, we searched for UXSS exploits in Chrome between versions 41 to 83, with a total of 66 UXSS exploits. After that, we analyzed these exploits, learned what makes them possible, and found common factors between them. Using these findings, we partitioned the UXSS exploits into three classes. Two of these classes were further partitioned into finer sub-classes to get a better precision in their characteristics. We evaluated our classification using Safari’s UXSS exploits to check whether it is relevant to other browsers with different implementations and consequently found out that our taxonomy is also highly applicable to Safari’s UXSS exploits.

Furthermore, for each class, we mapped the exploitation techniques and the vulnerability types used by its exploits. As part of it, we defined five new vulnerability types and created an additional classification based on the used vulnerability type. Although the exploits use a variety of client-side technologies, we were surprised to find out that most of them (56 out of 66) use these five vulnerability types.

The process of understanding and analyzing UXSS exploits can be challenging and time-consuming due to the complexity of the exploits. However, once the UXSS classes were defined and known to us, it became easier to understand additional exploits and classify them. We believe that our classification can serve as the foundation for further studies about UXSS exploits in the future.

Next, we delved into Google’s classification of UXSS exploits [24] and compared it with our classification. The result of this comparison is that our classification is on a larger scale, includes a wider range of UXSS exploits’ types, and can be better utilized to find new UXSS exploits in the future.

Finally, using our classification we evaluated Site Isolation, which is the chosen mitigation used by Google against UXSS exploits. We showed that Site Isolation mitigates against some UXSS classes but still has some major disadvantages, and therefore further work should be done. Also, during the evaluation of Site Isolation, we found a new type of attack in Chrome on Android that exploits weaknesses in its isolation mechanism.
Chapter 2

Model and Preliminaries

2.1 The Cross-Site Attacker Model

This model is used to describe how an attacker uses some of the well-known types of client-side exploits, like UXSS, XSS, and CSRF [26]. Usually, this attacker controls a web page that is visited by the victim user. Sometimes, this attacker uses additional methods (e.g., Social Engineering) to make the victim user visit its web page in the first place. Then, the attacker uses JavaScript (JS) code or other client-side method to make the victim's browser send a cross-origin request to the target web page, e.g., Google or Facebook. These cross-origin requests are usually an essential part of those exploits' exploitation flow.

2.2 Same Origin Policy

Same Origin Policy (SOP) [27] is a security mechanism that is implemented in browsers. This mechanism uses the definition of “origin”. An origin of a client-side context is the combination of the scheme, host, and port of the URL that refers to this context. The SOP mechanism restricts how a script or document can interact (read, write, execute) with a cross-origin context, i.e., with another context that has a different origin.

The SOP mechanism is complex and has gradually evolved over the years. For example, Cross-Origin Resource Sharing (CORS) is a SOP’s sub-mechanism that enables a service to share resources with another cross-origin service by mutual agreement.

Both XSS and UXSS exploits enable a malicious service to execute a controlled script on a cross-origin service, an interaction that should not be possible according to the SOP mechanism. However, they differ in the method they employ to bypass this mechanism.

On the one hand, XSS exploits bypass the SOP mechanism by making the target service serve controlled content. Then, the browser interprets this content as JS code and thus executes it. Therefore, XSS exploits do not cause a SOP violation because
the attacker’s JS code is associated with the target service’s origin.

On the other hand, UXSS exploits bypass this mechanism by exploiting the browser’s implementation of the SOP mechanism. Namely, they exploit various logical flows of the browser that cause unexpected behavior that violates the SOP mechanism.

### 2.3 Execution Context

In general, Execution Context is the environment of the running code. It can be described using several well-known concepts like the execution’s global scope. For a JS code in the browser, there are several context types:

- **Document context** - the most common browser context. Document contexts can be partitioned by the document type (e.g., HTMLDocument, SVGDocument) and by the method they were embedded to the web page (e.g., top frame, iframe).
- **Worker context** - represents web workers that execute scripts in the background.
- **Other contexts** - e.g., Inspector (i.e., DevTools), Extension, and Worklet.

In this thesis, unless otherwise stated, an Execution Context is always of the document type.

### 2.4 Chrome Terminology

**Chromium** - an open-source project developed by Google. Chrome browser is built over the Chromium project.

**Blink** - the rendering engine used by the Chromium project. It started as a fork of the WebKit project. Its main purpose is to transform HTML documents into their interactive visual representation. It is responsible for parsing HTML documents, building the Document Object Model (DOM) and CSS Object Model (CSSOM), building the Render Tree, and further rendering steps (layout and painting). It also manages the lifetime of the document contexts.

**V8** - the JS engine used by the Chromium project. Its main purpose is executing JS code. Its main embedder is the Blink layer. Each instance of the V8 engine is called a V8 Isolate. The relationship between Blink and V8 Isolate is one-to-many: there is one V8 Isolate for all the document contexts and an additional V8 Isolate for each existing worker context. The purpose of this partition is to enable parallel execution of JS code between web worker and their related documents. This is done by assigning different threads for each V8 Isolate. Pay attention that there is another partition inside a V8 Isolate into the concept of “worlds” [28], but this simpler representation is sufficient for our purposes.

**Current Execution Context** - each V8 Isolate has the concept of current Execution Context [28]. It is the context of the JS function that is currently running. JS
functions that are declared in different Execution Contexts can call each other, therefore each V8 Isolate holds a stack data structure with all the current Execution Contexts.

**V8 Binding** - a layer between Blink and V8 layers. Its main purpose is to manage the mapping between Blink’s objects and their V8 representatives (called wrapper objects).

**Process Model** - Chrome’s process model consists of multiple kinds of processes, e.g., Browser, Network Service, Renderer, and GPU. The Browser process is the high privileged process; there is only one Browser process. The Renderer is a sandboxed process (cannot access network stack, File system, etc); there are many Renderer processes per Browser process [29]. The Browser process assigns tasks to its Renderer processes and communicates with them via an Inter-Process Communication (IPC) framework. Examples of these tasks are rendering a document and setting up a Service Worker. Two main layers in the Renderer process are the Blink layer and the V8 layer.

![Diagram of Chrome's Components](image)

**Figure 2.1:** Chrome’s Components
Chapter 3

Related Work

3.1 UXSS exploits

There are several research studies about finding UXSS exploits. Most of these publications have been done outside of academia.

Di Paola et al. [20] presented two UXSS exploits. The first exploited Internet Explorer’s (IE) ActiveX API and the second exploited Adobe Reader, which is a browser plugin.

Rosseta Flash is a UXSS exploit found by Spagnuolo [21] that resulted in multiple patches in both client-side and server-side frameworks. He presented a way to abuse sites that provide JSONP [30] endpoints by loading the resource to a flash object and partially controlling its content via the callback parameter. Spagnuolo showed that it is possible to make a valid SWF file that consists exclusively of alphanumeric characters and thus meets the conditions of the callback parameter. Rosseta Flash is classified as a UXSS exploit because it exploited the lax check of the Flash parser, whose implementation is included in the browser by a plugin. This parser accepted a malformed JSONP payload consisted of two parts: 1. The prefix was an SWF formatted data crafted to include only alphanumeric characters and was passed via the callback parameter. 2. The suffix was a JSON value.

Nava and Lindsay [22] showed how IE’s “XSS filters” can be exploited for a UXSS. This mechanism detects and prevents reflected XSS attacks by scanning outgoing requests using predefined patterns. Whenever a string in the request matches a predefined pattern, it creates a regular expression to detect this string in the response. If there is a match in the response, it alters the response accordingly to prevent the XSS. In their paper, they show how this response manipulation can be exploited to transform a safe response into a malicious response that contains the attacker’s JS code. This UXSS emphasizes that even services that are immune to XSS exploits can still be vulnerable to UXSS exploits.

BEK is a language invented by Hooimeijer et al. [23] that is related to the last example. The BEK language was invented for building sanitizers with an advanced
analysis of the sanitizer behavior and its possible outputs. They used BEK to recheck IE’s XSS filters; in particular, they checked whether strings in XSS cheatsheets could be possible outputs of the XSS filters mechanism, which may detect additional security risks. Furthermore, they suggested an improved solution of using BEK to model the browser’s HTML parser to include all the possible methods to execute JS code. For this particular vulnerability, this research came up with a solution to minimize the differences between the XSS filters mechanism and the HTML parser; hopefully, to prevent another instance of it in the future.

### 3.2 UXSS Classification

Until now, only one research performed a classification of UXSS exploits. It was made by Google researchers Moroz and Glazunov [24]. It examines Chrome’s UXSS exploits reported between the years 2014-2016 and classifies them into eight exploit classes. Also, it suggests several prevention methods and describes the chosen prevention approach implemented in Chrome.

We discuss Google’s classification later in this thesis, in Chapter 7, to compare it relatively to our classification, which is presented in Chapter 5.
Chapter 4

UXSS Purpose’s Formulation

4.1 Introduction

For this formulation, we shall define 2 entities:

- **Attacker** - an entity that executes JS code on a victim user’s browser and wishes to use a UXSS exploit. Usually, the Attacker controls a web service and the initial JS execution is done in its origin.
- **Target** - the web service that is being targeted by the Attacker. In general, the Target entity can also be a strong browser context like the file system or an extension, but in this thesis, we will focus only on the classic Target entity, which is a web service that owns a domain that is not controlled by the Attacker.

Using these definitions, one can define the purpose of the UXSS exploits as follows:

*In a given browser and for each Target entity, a UXSS exploit enables the execution of the Attacker’s JS code* \(^1\) *in the Target origin.*

In the next section, we interpret this purpose in the context of the Chrome browser.

4.2 Understanding UXSS exploit’s purpose

The first half of the UXSS exploit purpose’s definition states its scope limits. UXSS is a logical exploit that usually depends on the browser’s implementation. Therefore, we should add a constraint of “in a given browser” to bind a UXSS exploit to a given browser implementation. In this thesis, the specific browser is Chrome. Moreover, in contrast to XSS exploits that only work on a specific service, UXSS exploits are independent of the Target service and usually work on most web services. For this reason, UXSS exploits assume as little as possible about the Target service; for example, the majority of the UXSS exploits only assume that the Target service provides a URL

\(^1\)In practice, it does not necessarily have to be JS code, and can also be other client-side languages, such as WebAssembly (Wasm) and Flash
that can be embedded into the document context via an iframe\textsuperscript{2}. Because of this, it is necessary to include in the aforementioned definition the limitation of “for each Target entity”, to differentiate it from exploits’ groups that are service dependant, like XSS.

The second half of the definition is the actual requirement. We split it into 2 parts and interpret each one individually. The first part is “the execution of the Attacker’s JS code”. It means that JS source code that was constructed by the Attacker entity is being executed in a V8 Isolate (an instance of the V8 engine in a Renderer process of Chrome). The second part is “in the Target origin”. In the following paragraph, we delve into the meaning of JS code execution in a specific origin in the Chrome browser. We do this by elaborating on low-level V8’s implementation.

The concepts of V8 Isolate and Execution Context are represented in the V8’s source code by the objects Isolate and Context. The Isolate object has a stack of Contexts in which the top item in the stack is the current Execution Context. Each Context holds a GlobalObject object, which represents the global scope of the JS where all the variables in the top scope are stored. Using the V8 Binding layer, the GlobalObject is mapped to its matching object of Execution Context in the Blink layer, which is the Document object in our context. The Document object has a class member of type SecurityOrigin that represents the origin of the document.

A V8 Isolate that executes JS code uses the above-mentioned objects’ chain to get its associated security origin, which is used for security checks. We will refer to this method of retrieving the security origin as “the origin of an executed JS code”.

We can use this definition to redefine the UXSS exploit’s purpose in the Chrome browser as follows:

\textit{A UXSS exploit of the Chrome browser enables the Attacker’s JS code to be executed in a V8 Isolate while it has an origin of the Target service, which can be any web service.}

Let us also define the Attacker’s JS source code as \textit{AttackerJSCode} and the current Execution Context with the target origin as \textit{TargetDocument}. The TargetDocument would always be the first Document with an origin of Target origin that the Attacker managed to execute JS code in its context.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attacker</td>
<td>An entity that wants to use UXSS exploit in victim’s browser.</td>
</tr>
<tr>
<td>Target</td>
<td>An entity that owns a web service and is targeted by the Attacker.</td>
</tr>
<tr>
<td>AttackerJSCode</td>
<td>JS code created by the Attacker.</td>
</tr>
<tr>
<td>TargetDocument</td>
<td>The first Target service’s document that the Attacker manages to execute its AttackerJSCode in it.</td>
</tr>
</tbody>
</table>

Table 4.1: Terms defined in Chapter 4

\textsuperscript{2}I.e., its response does not contain a mechanism like X-Frame-Options \textsuperscript{31} or CSP’s frame-ancestors \textsuperscript{32}.
ControlledResource : A resource used by the TargetDocument that its response is controlled by the Attacker.

JS Execution APIs : Methods for JS execution that are exposed to cross-origin documents.

JS Execution Methods : Methods for JS execution that are designed to be used only by the same document.

TargetIframe : The iframe element embedding the TargetDocument.

ParentDocument : The document containing the TargetIframe.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlledResource</td>
<td>A resource used by the TargetDocument that its response is controlled by the Attacker.</td>
</tr>
<tr>
<td>JS Execution APIs</td>
<td>Methods for JS execution that are exposed to cross-origin documents.</td>
</tr>
<tr>
<td>JS Execution Methods</td>
<td>Methods for JS execution that are designed to be used only by the same document.</td>
</tr>
<tr>
<td>TargetIframe</td>
<td>The iframe element embedding the TargetDocument.</td>
</tr>
<tr>
<td>ParentDocument</td>
<td>The document containing the TargetIframe.</td>
</tr>
</tbody>
</table>

Table 4.2: Terms defined in Chapter 5

In the next chapter we use multiple terms that were defined in both Chapter 4 and Chapter 5. For ease of reference, the most widely used terms are summarized in Table 4.1 and Table 4.2.
Chapter 5

Classification of UXSS Exploits

5.1 Introduction

In this chapter, we present our classification of Chrome’s UXSS exploits. For this purpose, we analyzed 66 UXSS exploits that were chosen as followed. 61 of these exploits were found using Chromium’s bug tracker. We used this tool to search for UXSS exploits that affected Chrome’s versions 41-83 (from 2015 to 2020). Also, we used this tool to track UXSS exploits reported by Project Zero that targeted Chrome’s extensions. To search for UXSS exploits using this tool, we used the query in Code 5.1. Then, we searched for UXSS reports within the results (e.g., filtered out bug reports that the word ‘UXSS’ is only mentioned there).

Code 5.1: Bug tracker query for UXSS exploits

```
uxss status:Fixed,Verified,Duplicate OR universal xss status:Fixed,Verified,
Duplicate OR "cross-origin scripting" status:Fixed,Verified,Duplicate
```

After that, we decided not to include UXSS exploits of Chrome on iOS and Chromecast. Also, we excluded exploits that require uncommon user interaction to trigger the exploit, like drag and drop or copy and paste.

The remaining 5 exploits were found in an attempt to cover a wider range of UXSS exploits that were not reported using Chromium’s bug tracker, mostly because the exploited code did not directly belong to Google. It resulted in 3 UXSS exploits that targeted Chrome’s extensions, a UXSS exploit that targeted the plugin of Adobe Flash Player [21], and Bo0om’s UXSS exploit [33].

To analyze these exploits, first, we learned chromium’s code, and in particular the code of the Blink and V8 layers. This phase included reading big code sections, learning the design of these layers, and delving into complex flows. Then, we analyzed each of these exploits and understood how they work. Many of these exploits are very complex, and thus we used Visual Studio’s built-in debugger to understand what exactly happens in each step of the exploit.

For our classification, we chose to partition the exploits based on the method used by the exploit to execute AttackerJSCode in the TargetDocument. This partition resulted
in 3 classes, which are presented in the following sections.

5.2 Class A - Controlled Scriptable Resource

The loading phase of a document starts with loading its main resource. The main resource’s data is then being handled by a parser based on the resource’s type, e.g., HTML, Flash, and XML. It is determined based on both the embed type and the mime type. The parser can invoke additional requests for sub-resources that are needed for the document’s loading, e.g., scripts, images, and stylesheets.

UXSS exploits in Class A target a TargetDocument’s resource that is being loaded during the document’s loading phase and make it controlled (partially or fully) by the Attacker. We refer to this resource as \textit{ControlledResource}. This requirement alone is not sufficient for the UXSS exploit purpose to be fulfilled because some of the loading contexts for a resource do not provide JS execution, for example:

- Image and stylesheet resources. The parsers for those contexts do not enable JS execution.
- Document’s main resource with a mime type of “text/plain”. The resource’s data is interpreted as plain text by the parser, which causes no further parsing to be made.
- Document’s main resource with an HTML mime type, but with sandbox flags that disallow script execution. In general, the HTML parser enables JS execution, but because of the sandbox flags, all JS execution is blocked.

Due to cases like these, we also require the ControlledResource to be processed by a parser in a way that enables it to execute JS in the document context (i.e., while the TargetDocument is the current Execution Context).

None of the UXSS exploits that were studied during our research targeted sub-resources, therefore we will not delve further into this case. Consequently, from this point onward the type of the ControlledResource will only be the document’s main resource.

In conclusion, there are 2 requirements for a UXSS exploit to be in Class A:

1. Using the exploit, the Attacker controls (partially or fully) the response of TargetDocument’s main resource. This resource will be called the ControlledResource.
2. The ControlledResource is being parsed by a parser that enables JS execution in the document context.

The ControlledResource can be one of 2 types: network resource and non-network resource.

If the ControlledResource is a non-network resource, then its response is declared and is not loaded from the network stack, e.g., iframes with srcdoc or blob URLs. In this case, the document’s origin is usually determined by its parent document (or its
opener document if it is a top-level document). From the Attacker’s perspective, the
challenge here is to make the document’s origin to be equal to the Target service’s
origin. We did not find any UXSS exploits of Chrome that used this option and thus
we do not delve into it here.

If the ControlledResource is a network resource, then its URL must belong to the
Target service. Moreover, as mentioned in Chapter 4, the exploit purpose (and therefore
this class requirements) should be accomplished for every Target service. Therefore, to
fulfill the first requirement, the exploit should provide a way to control the response
of a Target service’s URL for every Target service. This conclusion emphasizes how
challenging it can be to fulfill Class A’s first requirement and probably requires rare
exploitation in Chrome’s network stack or Blink’s resource layer.

5.2.1 UXSS Examples of Class A

Rosseta flash (as described in Chapter 3.1) is an example of an exploit from this
class. It satisfies the two requirements of Class A:

1. JSONP endpoints exist in most of the major services. Using their callback pa-
rameter, the Attacker can partially control the response of a JSONP endpoint’s
URL of the Target service. However, it has two major limitations: 1. It only en-
able to control a prefix of the response, while its suffix is returned by the Target
service. 2. This controlled prefix can only contain alphanumerical data.

2. The Attacker wants to invoke the creation of the TargetDocument and to embed
it in the Attacker’s document in a way that the TargetDocument’s parser would
provide JS execution. The Attacker’s controlled data must be alphanumeric and
must be a prefix of the response data. Therefore, the classic document parser,
an HTML parser, would parse this data as plain text which does not provide JS
execution. For this reason, the Rosseta Flash research studied a way to make
the Flash Parser accept this alphanumerical input as a valid SWF file, parse it,
create a document, and execute the desired JS.

Bo0om’s UXSS [33] is another good example of a UXSS exploit from this class. In
this exploit, the Attacker returns an archive response (i.e., an MHTML file [34]) for its
document. Inside of it, he embeds an iframe with a URL of the Target service. MHTML
is a feature that enables a document to serve its sub-resources (including iframes) along
with its HTML response, to reduce network latency. By using this feature, the main
resource of the Target service’s document is being loaded from the archive, which is
controlled by the Attacker. Under normal circumstances, a document with an archive
and all its descendants’ documents should have full sandbox flags. Therefore, the
Attacker cannot leverage the archive feature to execute controlled JS in a cross-origin
descendant document because JS execution in it is forbidden. However, in this exploit
the Attacker circumvents this problem by using the XSLT feature [35] that enables him
to replace the sandboxed document with a new one. The new document inherits its
origin from the old document (which is the Target service origin) but not its sandbox flags and therefore JS execution in it is enabled.

Bo0om’s UXSS is actually a bug in the XSLT feature that enables sandbox flags bypass. When this feature is combined with the archive feature it can achieve a UXSS exploit. This exploit satisfies Class A’s 2 requirements: the first requirement is satisfied by the MHTML feature that allows the Attacker to control the ControlledResource’s response; the second requirement is satisfied by overriding the sandboxed document with a new one which allows JS execution.

The final example demonstrates how exploits of browser extensions can be used to construct UXSS exploits of Class A. Extensions are privileged contexts that are executed in the browser. Usually, an Attacker initiates its attack on a victim’s browser not by executing JS code in an extension context but on a low-privileged context like a document with the Attacker’s origin. However, a Cross-Context Scripting [36] (XCS) attack, which is usually a DOM-based XSS on the extension context, can be used to execute JS code in a document with the extension origin (i.e., “chrome://<extension-id>”). After the XCS attack, an Attacker can use one of the network interception privileged APIs \(^1\) (in the case that they are permitted in the vulnerable extension’s manifest) to intercept network requests from cross-origin documents. This can be used to intercept the ControlledResource’s request and change its response with one that is controlled by the Attacker; by doing so, we satisfy Class A’s first requirement. The second requirement is obvious if the intercepted request is for an HTML resource. One of the latest reported exploits of this kind used Video Downloader’s extension [37].

5.3 Class B - Exploit Cross-Context JS Execution APIs

Exploits in Class B use existing APIs that enable JS execution in a given document and that are exposed to other contexts, which are not same-origin documents. These exploits use those APIs to execute AttackerJSCode in the TargetDocument. Clearly, those APIs are not allowed to be used from cross-origin documents, and therefore exploitation is needed. We refer to these APIs as “JS Execution APIs”.

Although there are several JS Execution APIs, only 3 APIs’ types were actually used by UXSS exploits, and thus we will only examine them. Each of these APIs’ types has unique characteristics and exploitation techniques and thus is represented by a different subclass.

5.3.1 Subclass B-1 - JS Execution API For Same Origin Documents

This subclass of UXSS exploits is the most widely used (with a total of 26 exploits). Exploits in this subclass use JS Execution APIs that are permitted to be used only by same-origin documents but are also accessible by cross-origin documents. When a

\(^1\)E.g., “webRequestBlocking”, “declarativeNetRequest” or “declarativeWebRequest”.

20
cross-origin document tries to use one of these APIs, a security check is performed and fails, and thus the API’s action is not executed.

Let us assume that TargetDocument is embedded into the page via an iframe element, which we refer to as TargetIframe. We refer to its parent document as ParentDocument. There are several APIs that can be used by the ParentDocument to navigate the TargetDocument to a new URL, for example: changing TargetIframe’s src attribute; changing the href attribute of TargetIframe’s Location (accessed by TargetIframe’s contentWindow.location); simulating a form’s submission or an anchor’s click with the TargetIframe as the target. When the ParentDocument uses one of these APIs and set the URL to a JS URL (a URL with a scheme of “javascript:"), it is interpreted by the Blink layer as a request for JS execution in which the rest of the JS URL is considered as the JS code to be executed. Except two, all the UXSS exploits in this subclass exploited the first mentioned API (of setting the TargetIframe’s src attribute to a JS URL), and thus we will focus on it.

As discussed before, according to the Same Origin Policy, this JS Execution API is allowed to be used only if ParentDocument and TargetDocument are same-origin; otherwise, setting TargetIframe’s src attribute to JS URL should be blocked. In the Blink layer, Code 5.2 is responsible for this behavior.

```
bool ScriptController::canAccessFromCurrentOrigin(...) {
    ... 
    return isolate -> inContext() || BindingSecurity::ShouldAllowAccessToFrame (...);
}

bool HTMLFrameElementBase::isURLAllowed() {
    ... 
    if (protocolIsJavaScript(completeURL)) {
        if (contentFrame() && !ScriptController::canAccessFromCurrentOrigin(...))
            return false;
    }
    ... 
    return true;
}

void HTMLFrameElementBase::openURL(...) {
    if (!isURLAllowed())
        return;
    ...
    loadOrRedirectSubframe(...);
    ...
    // in case it's a JS URL: execute it in the contentFrame
}
```

Code 5.2: Blink code for setting iframe’s src to JS URL

---

2which exploited changing the location.href (CVE-2015-1293) and the window.open API (CVE-2020-6506)

3This code is not the latest version of this class, but a vulnerable version that was used by UXSS exploits.
HTMLFrameElementBase is a base class for HTMLIFrameElement, which is the Blink’s representative for an iframe element. By changing the value of TargetIframe’s src attribute, the openURL method is called. This method would execute JS in TargetDocument only if the isURLAllowed method returns true. In our case, where it is a JS URL, the isURLAllowed method performs an additional check on its contentFrame, which is the iframe’s inner document. This check enforces that if the iframe has an inner document, then the current execution context is permitted to access this document. In the case of JS URL, this method interprets the JS URL as JS code and executes it in the inner document. Therefore, the additional check of isURLAllowed is necessary to guarantee that ParentDocument and TargetDocument are same-origin, and thus ParentDocument is permitted to execute code in the TargetDocument.

There are two techniques to exploit Code 5.2 and that were commonly used among UXSS exploits:

First Exploitation Technique

<table>
<thead>
<tr>
<th>Use vulnerability of:</th>
<th>Standard JS Code to use the vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detached Loaded Frame or Synchronous Navigation to External URL</td>
<td></td>
</tr>
</tbody>
</table>

Create Targetiframe with:
- src attribute – JS URL (AttackerJSCode)
- contentFrame – points to TargetDocument

openURL method on Targetiframe:
- isURLAllowed: returns True (called by parser flow)
- Executes the JS URL (AttackerJSCode) in the Targetiframe’s contentFrame, which is the TargetDocument

---

Figure 5.1: openURL’s first exploitation technique

The first technique exploits the “can access” check which is performed by ScriptController::canAccessFromCurrentOrigin. The call to BindingSecurity::Should AllowAccessToFrame performs the actual same-origin check between the current Execution Context in V8 and the iframe’s inner document. However, because of the OR statement, this check is
performed only if there is a current Execution Context in the V8 Isolate; otherwise, the whole statement returns true, and the SOP check is bypassed. The openURL method is called from two contexts: from V8 flows (setting iframe’s src attribute and adding it to the DOM), and from the document’s parser flows. In the V8 flow, the current Execution Context exists (and therefore it is not interesting in our case) while in the parser’s flow it does not. Code 5.3 is an example of HTML code that causes the document’s parser to trigger this flow.

```html
<html>
<body>
<iframe src='javascript:alert(1)'></iframe>
</body>
</html>
```

Code 5.3: Example of a document’s parser flow to set iframe’s src to JS URL

In this example, the parser would create an HTMLIFrameElement and call its openURL method with the JS URL. The SOP check would be skipped and the call to canAccessFromCurrentOrigin would return true because the flow originated from a parser flow and not from a V8 flow. At first glance, this skip seems safe because the inner document is empty (i.e., about:blank document), which inherits the security properties of its parent document, and therefore the SOP check can be avoided. However, this false assumption was the reason for many UXSS exploits that used document parser’s bizarre flows that enable the inner document to be the TargetDocument instead of an empty document. In conclusion, 3 requirements must be fulfilled to use the first technique:

1. TargetIframe’s openURL method should be called from a document’s parser flow.
2. At the time of the call, TargetIframe’s contentFrame field should point to the TargetDocument (e.g., google.com).
3. At the time of the call, TargetIframe’s src field should contain a controlled JS URL (namely the AttackerJSCode).

There are two popular methods to satisfy these 3 requirements. The first method uses a vulnerability type we called “Detached Loaded Frame” and the second uses another vulnerability type we called “Synchronous Navigation to External URL”. These vulnerabilities are described in Chapter 6. For each of these methods, there is a standard JS code that enables the fulfillment of these 3 requirements using it. Namely, these two fixed code pieces implement the full exploitation techniques of these vulnerabilities. Due to their complexity and low-level details, we do not cover them in this thesis.

In Chrome version 56 a patch came out that closed this exploitation technique. Until then, 19 UXSS exploits used it. This patch ensures that the isURLAllowed method always performs the SOP check, including when it is called by a document’s parser flow. It is worth noting that the generic solution for closing this exploitation technique did
not come from Chromium’s developers but from Mariusz Mlynski, a famous White-Hat researcher who found most of the UXSS exploits in this subclass. This is probably due to the complexity of the exploits and the difficulty to understand what was essential to prevent similar exploits in the future.

Second Exploitation Technique

![Diagram of openURL’s second exploitation technique]

The second exploitation technique is a time-of-check-to-time-of-use [38] (TOCTOU) attack. It relies on what can happen inside the openURL method (in Code 5.2) between isURLAllowed and the call to execute the JS URL. The security check is performed at the start of the method via the isURLAllowed method. This check can be easily passed if the inner document is not a cross-origin document (e.g., about:blank). However, we want to use this API to execute a JS URL in TargetDocument, which is a cross-origin document. Because the call to execute the JS URL is at the end of the openURL method, we can achieve it as follows: at the start of the method, the TargetIframe’s inner document should be a same-origin document, but it should be navigated to TargetDocument until it reaches the end of the method. We refer to this interval as “openURL’s critical interval”.

Both the openURL method and document navigations are executed in the same thread, which is the main thread. For this reason, the navigation cannot be processed
in parallel to the openURL method and thus should be invoked from openURL’s critical interval. This can be done by the call to loadOrRedirectSubframe (from openURL) that can be manipulated to trigger an event handler (which is a synchronous execution of JS code) in the middle of openURL’s critical interval. This JS hook should perform a synchronous navigation to TargetDocument and is required to finish before it returns. Otherwise, the JS URL would be executed in the previous document and not in the TargetDocument. For this JS hook, one of the main approaches is to use a vulnerability of Synchronous Navigation to External URL.

Eventually, a patch came out and closed the second exploitation technique. It changed how the openURL method handles JS URLs. The newer version adds an asynchronous task to the inner document to execute the JS URL in it, and thus the document that is checked in the isURLAllowed method must be the same as the one that executes the JS URL.

5.3.2 Subclass B-2 - JS Execution API For Extensions

Browser’s Extensions are privileged contexts in the browser. One of their privileged APIs is content scripts’ injection that enables an extension to inject JS code into cross-origin documents. UXSS exploits in this subclass manage to execute the AttackerJSCode in the TargetDocument by exploiting a content script injection of a vulnerable extension. The exploits that use this API can be split into two groups: “direct injection” and “indirect injection”, as elaborated below.

In the indirect injection group, the content script which is injected into the TargetDocument is the original content script of the extension and thus it is not controlled by the Attacker. However, this content script is vulnerable to a DOM-based XSS attack. Because this vulnerable content script is injected into almost every created document, it makes every created document vulnerable and thus enables a UXSS in it. For example, a UXSS exploit in this group used the Evernote Web Clipper extension [39]. Its content script was permitted to be injected into every document and was vulnerable to a DOM-based XSS attack.

In the direct injection group, the content script which is injected into the TargetDocument is already controlled by the Attacker, i.e., the content script is the AttackerJSCode. Content scripts can be injected declaratively (declared in the extension’s manifest) or programmatically (by JS API from the extension’s context) [40].

If the AttackerJSCode is injected declaratively, it requires the Attacker to control the response of a JS resource of a vulnerable extension. This probably means that the Attacker used an XSS exploit, and because the URL is fixed in the manifest, it means it must be a persistent-XSS exploit on a JS resource, which is uncommon.

The chrome.tabs.executeScript API (or its newer version chrome.scripting.executeScript) is used for programmatic injection of content scripts. The target documents for this

\[\text{Depends on the match patterns of the content scripts’ field in the manifest.}\]
API usage are limited by the host permissions and can be temporarily expanded to additional documents by the activeTab permission. To be used, it requires the Attacker either to exploit a logical flow of the extension that uses this API, or to use an XCS attack (as described in Chapter 5.2.1) to execute code in the extension’s context, and consequently to use the API as he wishes. Compared to UXSS exploits that use the declarative injection, this UXSS type is more practical. For example, a UXSS exploit of this kind used the Steam Inventory Helper extension [41]. This exploit used a DOM-based XSS with a clickjacking attack for the XCS attack. After that, it was possible to use the executeScript API with the activeTab permission that was permitted in the extension’s manifest.

5.3.3 Subclass B-3 - JS Execution API For Browser’s Components

In the last two sections, we discussed two JS Execution APIs that are designed to be used by documents and extensions, both of which are browser’s Execution Contexts. However, the demand for these APIs is not exclusive to the browser’s Execution Contexts and there are JS Execution APIs that are used by the browser itself. Chrome has some components that can inject JS code into specific documents. An example is the Google Translate button in Chrome’s Omnibox that can trigger a page translation. This functionality is implemented by a JS code that does the translation and is injected by custom JS Execution API into the top document.

UXSS exploits in this subclass exploit these Chrome’s components and inject the AttackerJSCode into the TargetDocument via their custom JS Execution APIs. In our research, we found only two UXSS exploits that match the description of this subclass.

The first UXSS (CVE-2015-1275) exploited a mechanism in Chrome on Android that intercepted Android’s Intents for Google Authenticator application and was implemented by the AuthenticatorHelper object. This application exposes APIs for 2-factor authentication and had a special API (that does not exist anymore) that can be used from within web pages by using Intent’s URL [42] with a specific action’s value. The AuthenticatorHelper was used to parse the Intent’s URL, to send the Intent to the Google Authenticator application, to wait for an Intent with the response, and to send it back to the page that sent the intercepted Intent’s URL. The response was sent back to the page by using a custom JS Execution API that enables to inject JS code into its top document. The JS code was constructed by the AuthenticatorHelper and consisted of a postMessage call with a JSON message that included the original Intent’s URL along with the received response. However, the AuthenticatorHelper did not validate the received Intent’s URL, which is eventually injected into the page as part of the crafted JS code. Therefore, it could be exploited by crafting an Intent’s URL with a trailing single quote followed by the injected JS code that contains the AttackerJSCode. To fulfill the requirements for a UXSS exploit, the JS code should be injected into a Target service’s document. Therefore, this exploit also used a timer to
invoke a top document navigation to a Target service’s URL.

The second UXSS exploit (CVE-2018-6101) is related to Chrome Developer Tools (DevTools) mechanism. This mechanism is designed to be used by developers and enables them to perform high privilege actions in the browser, e.g., to read a local file, to write to a local file, and to inject JS code to any document on any page. Therefore, if an attacker manages to perform actions in the context of Chrome DevTools, he can easily inject its AttackerJSCode to the TargetDocument. Chrome has an option to be executed with a flag of “–remote-debugging-port”, which is used by default in some automation frameworks like Selenium and Puppeteer. This flag causes Chrome to set up a WebSocket server that provides remote access to the Chrome DevTools APIs. This UXSS exploit managed to access the remote debugging’s server from within a web page by scanning localhost’s ports and using a DNS rebinding attack [43].

5.4 Class C - Exploit Same-Context JS Execution Methods

Exploits in Class C aim to execute AttackerJSCode in TargetDocument via existing methods for JS execution that are designed to be used by the same document, i.e., by the TargetDocument. Clearly, these methods are supposed to be used only by the document itself and not by cross-origin documents and therefore exploitation is needed. We refer to these methods as “JS Execution Methods”.

There are two types of JS Execution Methods used by UXSS exploits. Each of them requires different exploitation techniques and thus is represented by a different subclass.

5.4.1 Subclass C-1 - JS Execution Methods of JS Functions

UXSS exploits in this subclass manage to access JS functions of the TargetDocument that enable JS execution (like eval and Function’s constructor) and invoke them. These JS functions are not supposed to be accessed by cross-origin documents and thus their implementation does not contain any security checks. Therefore, if the Attacker finds a way to access one of these JS functions of the TargetDocument, the direct consequence is JS execution in the TargetDocument. The Function’s constructor is the most common JS Execution Method used by this subclass’s UXSS exploits (21 out of 22), and therefore we will focus on it.

Each function in JS is an instance of the JSFunction object and is created by the Function’s constructor, which is also a JSFunction object whose implementation is a native code in the V8 engine. Each Execution Context stores its own Function’s constructor and each Function’s constructor stores the Execution Context it belongs

Obviously, these methods can also be used by same-origin documents that can access the Target-
to. When a JSFunction is executed, it is executed in the Execution Context of the Function’s constructor that created it. If an Execution Context invokes a JSFunction of a different Execution Context, the function’s Execution Context is pushed into the stack of the current Execution Contexts (see Chapter 2.4).

According to this short analysis, if the Attacker has JS access to the Function’s constructor of the TargetDocument, he can use it to create a JSFunction with the AttackerJSCode as the function’s body (it is passed as the first argument of the Function’s constructor) and invoke it. As expected from a JS Execution Method, this will cause the AttackerJSCode to be executed in the TargetDocument and thus a UXSS exploit will be achieved.

Two methods were used to access the TargetDocument’s Function’s constructor. The first method is more common (were used by 19 UXSS exploits) and uses a vulnerability type named “Cross-Origin Object Leak”. This vulnerability type allows the Attacker to leak a JS object that belongs to the TargetDocument. The leaked JS object may be of any type, like an Exception or a Promise object, with two exceptions: WindowProxy⁶ and Location. These two JS objects are well-known to be accessible by cross-origin contexts (they are accessible through the iframe’s contentWindow field) and therefore most of their methods and attributes are protected by SOP security checks. Because of these checks, accessing these two JS objects cannot be used to access additional JS objects of the TargetDocument. These checks do not exist for other types of JS objects (due to efficiency reasons) and thus by leaking one of them one can also leak the TargetDocument’s Function constructor, as we will see in the next paragraph.

Any JS object is constructed by a constructor that can be accessible by its constructor field. This constructor field is a JSFunction object and therefore its constructor field is the Function’s constructor. Therefore, using a leaked object from the TargetDocument, the Attacker can use Code 5.4 to execute AttackerJSCode in TargetDocument.

```
functionConstructor = leakedObject.constructor.constructor;
functionConstructor(AttackerJSCode)();
```

Code 5.4: Leveraging a leaked object into a code execution

The second technique manages to access the Function’s constructor through the WindowProxy object. The WindowProxy is just a wrapper for the Window object that is accessible for cross-origin documents. Under normal circumstances, if a cross-origin document tries to access WindowProxy’s Function attribute (i.e., iframe.contentWindow.Function), an exception is thrown. To understand 2 of the UXSS exploits that exploited this mechanism, we will delve into the V8’s implementation of this access.

As part of the invoking process of a JS Object’s getter, an access check is performed only for the WindowProxy and Location objects. If this access check fails, it means that the accessing context is not permitted to access all the attributes of the JS object

⁶A wrapper for the window object when accessed by the iframe’s contentWindow field.
and then cross-origin interceptors are executed to determine if this particular access is allowed for cross-origin contexts. For example, the WindowProxy has a cross-origin interceptor for the location’s getter and setter but does not have one for the Function getter. Therefore, the Attacker should pass the access check to successfully access the Function’s getter of the WindowProxy.

The access check consists of 3 checks and only if all of them fail, it returns access denied. The 3 checks are as follows:

1. Whether the accessing context and the receiver context (the context which is related to the JS Object) point to the same context.
2. Whether both contexts have the same security token, which is usually the origin of the context represented as a string. This check allows quick and simple SOP check in the V8 layer instead of forwarding the SOP check to the Blink layer.
3. Whether the accessing context may access the receiver context according to the Blink’s SOP check (includes additional cases, like overridden document.domain).

One UXSS exploit in this subclass managed to set the security token of the Attacker’s document to be the same as the security token of the TargetDocument, and thus to passed WindowProxy’s access check. Another UXSS exploit managed to activate a special flow for documents with a “file” scheme to set a flag named “universal access”. While this flag is set, this document can pass any Blink’s SOP check and thus can pass the third check of the WindowProxy’s access check. This bypass in the Blink’s SOP check is very useful and could also be used to pass the security checks in Subclass B-1. Thus, creating a UXSS exploit of a different class according to this classification.

These two UXSS exploits use a vulnerability type we named “Wrong Security Context Evaluation”.

Eventually, in Chrome version 53 a patch came out that added a security check to the native code of the Function’s constructor. This patch adds a check that validates that all Execution Contexts in the stack of the current Execution Contexts have access to the Execution Context of the Function’s constructor by performing an access check identical to WindowProxy’s access check. This patch ensures that a Function’s constructor can only be used by Execution Contexts that have access to its Execution Context. Clearly, this is not the case for UXSS exploits in this subclass that used the leaked Function’s constructor from inside the Attacker’s Execution Context and thus this patch causes the creation of their cross-origin JSFunctions to fail. As discussed in the example of the eval function, the Function’s constructor is not the only JS function that can enable JS execution in a document. Therefore, additional patches were released that gradually covered the other JS functions as well.

5.4.2 Subclass C-2 - JS Execution Methods of the Document Element

UXSS exploits in this subclass use actions on the document element that can trigger JS execution. There are 2 common types of these actions:
1. Actions that navigate the TargetDocument to a JS URL that contains the AttackerJSCode. As discussed before, this navigation is interpreted by Blink as a JS execution, which causes the AttackerJSCode to be executed in the TargetDocument. Examples of such actions: simulating a click on an anchor element with an href’s attribute of JS URL; simulating a submit on a form element with an action’s attribute of JS URL.

2. Actions that add an element to the TargetDocument’s DOM that triggers the execution of AttackerJSCode. For example, adding a script element or adding an img element with a broken URL and an onerror attribute.

The implementations of these actions do not contain security checks because they are expected to be used only by the same document. Therefore, if a JS code that is executed in the Attacker origin triggers one of those actions on the TargetDocument, it results in a UXSS exploit.

In this subclass, all the classified UXSS exploits of Chrome\textsuperscript{7} used the first action’s type, and thus we will focus only on it.

Normally, to perform actions of the first type, the Attacker should be able to access the element’s JS object (e.g., the href element). Therefore, the Attacker should create\textsuperscript{8} an element that belongs to the TargetDocument by using the document.createElement method and thus should have access to the document JS object of the TargetDocument. And indeed, two UXSS exploits used the vulnerability type of Cross-Origin Object Leak to leak the TargetDocument object and used it to create an anchor element with a JS URL and simulate a click on it. However, that kind of leakage (of the Document object) is very rare. Alternatively, there is a second technique that is more commonly used but is also more complex.

The idea behind the second technique is to use one of those actions on an Attacker’s document and then to cause the navigation of the JS URL to be performed on the TargetDocument instead. The only action that was used alongside the second technique is the form submission. The HTMLFormElement object is the Blink’s representative for the form element, and the scheduleFormSubmission method is called as part of the submit action\textsuperscript{9}.

```cpp
void HTMLFormElement::scheduleFormSubmission(...) {
  ...
  if (protocolIsJavaScript(submission->action())) {
    ...
    document().frame()->script().executeScriptIfJavaScriptURL(submission->
    action())
  }
  ...
}
```

Code 5.5: Blink code for HTML form’s submission

\textsuperscript{7}There were UXSS exploits of Safari that used the second action’s type
\textsuperscript{8}There is another option of reusing an existing element, but we do not cover it here.
\textsuperscript{9}This C++ code was taken from an older version of Blink, but the concept behind it stays the same.
Each Document object has a frame field that points to a LocalFrame object. The LocalFrame is used as a container that holds several objects that are necessary for the document to be functional, like the ScriptController object which is responsible for JS execution within it. In the scheduleFormSubmission method, if the submit action is a JS URL, the document’s ScriptController executes the JS URL. Under normal circumstances, if the Attacker’s document has a form and a submit action is triggered on it, the JS URL is executed in the ScriptController of the Attacker’s Document. However, if for some reason the frame field of the Attacker’s document points to the LocalFrame of the TargetDocument, the JS URL is executed in the TargetDocument instead. This behavior (of two Documents point to the same LocalFrame) should not be possible, and thus requires a vulnerability we named “Twin Documents”. This technique was used by 6 UXSS exploits.

5.5 UXSS Classification Summary

In Figure 5.3 we can see the distribution of the 66 UXSS exploits of Chrome among the classes. For more details, see Appendix A.

![Figure 5.3: Chrome’s UXSS exploits’ distribution by classes](image)

In this classification, we partitioned the UXSS exploits of the past few years into 3 classes based on the type of method used to execute AttackerJSCode in the TargetDocument. UXSS exploits in Class A target the loading phase of the TargetDocument and intercept one of its resources whose embed type enables JS execution. UXSS exploits in classes B and C target document mechanisms that enable JS execution. These two classes differ in the purposes and the properties of the used mechanisms.

In Class B, the used mechanisms are called JS Execution APIs and are intended to be used by other contexts and thus are accessible by them. Most of the UXSS exploits in Class B used JS Execution APIs that are intended to be used only by same-origin
documents. These APIs are accessible by both same-origin and cross-origin documents and thus contain security checks to prevent misuse. Therefore, the Attacker can already access these APIs and only need to bypass their security checks. The rest of the UXSS exploits in Class B used JS Execution APIs that are available only for privileged contexts (like extensions and Chrome’s mechanisms) and thus their implementations do not include security checks. Therefore, in this case, the Attacker does not need to exploit the implementation of these APIs but instead needs to find ways to make these contexts use their APIs in an unintended way, i.e., with the AttackerJSCode.

In Class C, the used mechanisms are called JS Execution Methods and are supposed to be used only by the TargetDocument and thus once a context could access them there are no security checks performed as part of their usage. Therefore, UXSS exploits in Class C focus on the way to access these JS Execution Methods, which in most cases involve leaking a TargetDocument’s JS object to the Attacker’s context.

It is worth noting that the clustering technique used by this classification focuses on the possible methods to execute JS code in a given execution context and not only on the methods used by the existing UXSS exploits. This approach raises the probability that this classification would be applicable for future findings of UXSS exploits because each exploit is expected to use an execution method provided by the browser.

Furthermore, this classification may yield advanced techniques for finding new UXSS exploits. Although we only delved into the execution methods that were actually used by UXSS exploits, it is a simple task to expand each class and find all the execution methods that match this class and can be possibly exploited. This can be done by exploring Chrome’s code using Chromium’s open-source project. For example, Class A can be expanded by finding all the mechanisms in Chrome that enable to control the response of cross-origin resources; Subclass B-3 can be expanded by finding all the custom JS execution APIs that are used by Chrome’s components to execute JS code in other documents. As seen in this classification, exploits in the same class usually have similar difficulties and exploitation techniques; this can be helpful to improve the effectiveness of this search process.

5.6 Classification Evaluation

To evaluate our classification, we analyzed 29 UXSS exploits of the Safari browser and then tried to classify them using the new defined classes (for more details, see Appendix A). These exploits were found using Chromium’s bug tracker by filtering Project Zero reports since 2016. In contrast to the search process we did for Chrome, here we did not expand it to find UXSS exploits related to Safari’s extensions. The results are presented in Figure 5.4.

As can be seen in Figure 5.4, all of Safari’s UXSS exploits belong to one of this classification’s classes. Moreover, all the vulnerabilities types we defined in this chapter are used by Safari’s UXSS exploits, and also they share some of the exploitation tech-
Figure 5.4: Safari’s UXSS exploits’ distribution by classes

...niques, e.g., the two exploitation techniques described in Chapter 5.3.1. This finding shows that, although this classification was built based on Chrome’s UXSS exploits, it can also be relevant to other browsers. It is worth noting that the rendering engine’s implementations of both Safari’s WebCore and Chrome’s Blink have a lot in common because Blink is a fork of WebCore. Therefore, in future work, it would be interesting to check how this classification is relevant to other browsers with significantly different implementations, e.g., Firefox.
Chapter 6

Vulnerabilities’ Classification

6.1 Introduction

In the last chapter, we saw for each UXSS class which vulnerabilities types are used by its exploits and the exploitation techniques used to utilize them. In this chapter, we focus on the vulnerabilities types that were only mentioned by name in the last chapter and define them properly. As described in Chapter 1, this step is crucial in building the foundation for understanding UXSS exploits.

Figure 6.1 presents the distribution of Chrome’s UXSS exploits among them.¹

Figure 6.1: Chrome’s UXSS exploits’ distribution by vulnerabilities usage

According to Figure 6.1, most of the classified UXSS exploits (56 out of 66) rely on one of the five defined vulnerabilities types. Moreover, 8 out of the 10 UXSS exploits with an "undefined" vulnerability exploited privileged contexts (i.e., exploited either an

¹CVE-2015-1292 is an exception that used two vulnerabilities, i.e., it used a Twin Documents vulnerability to construct a Cross-Origin Object Leak vulnerability
extension or a browser’s component). Currently, we prefer to refer to the vulnerabilities used by these exploits as context-specific and leave their classification for future work.

This result is unexpected because the classified UXSS exploits use a wide range of exploitation techniques and client-side technologies. A possible explanation is that each of these vulnerabilities breaks a critical Blink’s invariant and thus can be leveraged to unexpected security implications in the Blink layer. We now delve into each of these vulnerability types.

### 6.2 Detached Loaded Frame

*Detached Loaded Frame* is a vulnerability type that enables DOM tree corruption by creating an iframe element that is “detached” and “loaded” at the same time. Detached means the element is not connected to the DOM tree; it is in the same state as an element created by createElement and not added to the DOM by appendChild. Loaded means the iframe contains an inner document. Under normal circumstances, when an iframe is detached from the DOM (e.g., via removeChild), its inner document is closed, destroyed, and would not be created until the iframe is connected once again to the DOM. Instances of this vulnerability type are usually related to Blink’s logic of removing a child Node from a Node object. These vulnerabilities create an unstable Node’s state by executing JS code in the critical sections of the remove method, which is disallowed and enforced by Blink’s security mechanisms.

### 6.3 Synchronous Navigation to External URL

*Synchronous Navigation to External URL* is a vulnerability type that enables to perform a synchronous navigation of a document to an external URL (e.g., “https://www.google.com”). This vulnerability type’s instances are implemented by JS code; they are synchronous because they can perform this navigation and finish before they return. Under normal circumstances, a document navigation to an external URL is an asynchronous task because it is performed by multiple processes (recall that the Renderer is a sandboxed process that uses the Network Service process to access the network) and involves network communication, which is an asynchronous action. Therefore, the JS code should wait until the asynchronous task finishes. Consequently, the following question arises: Why it is not enough to start a document navigation and to perform a busy wait in the JS code until it finishes? The problem is that both JS execution in the document context and document navigation happen in the Renderer process’s main thread. Therefore, performing a busy-wait in the JS code is not sufficient because it makes the main thread occupied, which means it cannot execute the task of finishing the navigation. Moreover, freeing the main thread by finishing the JS execution is disallowed because we can return from the JS code only when the navigation is done due to its synchronous requirement.
Instances of this vulnerability type use flows that call the main thread’s MessageLoop (which allocates the current thread’s tasks) from within the JS code and thus executed the required Blink’s task without exiting the JS code. However, to mitigate this unwanted behavior, when a Blink’s flow calls the MessageLoop there is a security mechanism named ScopedPageSuspender (or ScopedPagePauser in newer versions) that suspends the state of all pages until the execution is returned to Blink. This suspension includes all navigation tasks for documents inside each page. ScopedPageSuspender ensures that a Blink’s task cannot call from itself to another task because it could break important invariants in Blink, which can cause serious security problems. Consequently, these vulnerabilities usually were after ways to bypass this mechanism, e.g., to find uncommon flows in which the ScopedPageSuspender does not suspend newly created pages.

6.4 Cross-Origin Object Leak

Cross-Origin Object Leak is a vulnerability type that enables leaking a JS object from a cross-origin Document context. The leaked JS object may be of any type except the WindowProxy and the Location object (for more details, see Chapter 5.4.1).

Instances of this vulnerability type usually depend on bugs in the V8 binding layer that enable JS objects to be created in the wrong execution context.

6.5 Twin Documents

Twin Documents is a vulnerability type that makes two documents connected to the same LocalFrame. To understand this vulnerability type, we first need a basic understanding of the LocalFrame object. This object is a container that holds objects that are necessary for a document to be functional. Also, each LocalFrame is associated with a “place” inside the browser page. Namely, there is one LocalFrame for the top document and one for each frame owner, which is an HTML element that can embed inner documents within it, like the iframe, object and embed elements. In the case of navigation, the LocalFrame is reused, which means the inner Document is replaced while the place of the Document inside the page (which is represented by the LocalFrame object) stays. When this happens, the previous Document object becomes detached, which means it is destroyed and stops pointing to the LocalFrame. This detach process is necessary because the Blink layer assumes that LocalFrame and Document objects have a one-to-one relationship.

This vulnerability type breaks this assumption and makes two Document objects point to the same LocalFrame. Usually, these vulnerabilities prevent the previous Document from being detached during an iframe’s navigation and thus make the two Document objects share the same LocalFrame.
This vulnerability has another variant with the LocalDOMWindow object (represent the Window interface) instead of the Document object. We do not cover this variant here.

6.6 Wrong Security Context Evaluation

Wrong Security Context Evaluation is a vulnerability type that enables the security context of an execution context to be wrongly evaluated. For the sake of this vulnerability type, the security context includes all the security properties of the execution context, e.g., origin, CSP, security token, and sandbox flags.
Chapter 7

Comparison with other UXSS classifications

Google’s publication [24] is the sole classification of UXSS exploits done besides this research’s classification. It includes 63 reports of UXSS exploits that were reported in 2014 - 2016. The classes of Google’s classification are marked with numbers while this research’s classes are marked with letters; this will be helpful to distinguish between the classes of the two classifications when we refer to them.

The main difference between the two classifications is the clustering technique. The clustering technique of Google’s classification is based on common root causes or exploitation patterns, e.g., both Class 2 and Class 6 consist of UXSS exploits whose root cause is a missing or incorrect usage of access check in the Blink and V8 layers respectively. This differs from this thesis’s clustering technique that is based on the methods that were used by UXSS exploits to execute JS code in cross-origin documents.

This difference makes these two classifications differ in the number of classes (8 to 6, when we compare it with this classification’s sub-classes) and in the distribution of the UXSS exploits between them. Despite that, there are still some similarities between the two classifications. Class 1 includes the exploits from Subclass B-1 that use the first exploitation technique. Class 4 includes most of the exploits in Subclass C-2 and exploits from Subclass B-1 that use the second exploitation technique. Most of the exploits in Subclass C-1 are split in Google’s classification into 3 classes: Class 3, Class 5, and Class 6. This is because they chose to make a deeper analysis of the exploits in this class based on some common factors while we chose to not delve into it more than presented in Chapter 5.4.1. UXSS exploits that belong to Subclass B-2 are not included in Google’s classification at all and thus there is no equivalent class in their classification.

The main disadvantage of Google’s classification stems from its purpose, which is to organize the old UXSS exploits into groups and derive conclusions from that. However, it misses its full potential of laying the foundations for finding new UXSS exploits and understanding their characteristics. For example, there are plenty of UXSS exploits
that are not included in their work and do not have any meaningful class that matches them (except Class 8, which is their default class and has no real meaning), like Bo0om’s UXSS exploit [33], Rosseta Flash [21] and CVE-2020-6506. As explained in Chapter 5.5, this is not the case for our proposed classification, which was designed to support future findings of UXSS exploits and provide new techniques for finding them. The cause for this difference lies in our clustering technique that is not based on the common factors of old UXSS exploits but on the possible methods for executing JS code in a document.

\[1\text{It is the latest reported UXSS exploit that is covered in this thesis's classification and belongs to Subclass B-1}\]
Chapter 8

Analysis of Site Isolation

8.1 Introduction

According to Google’s publication [24], the chosen countermeasure against UXSS exploits is Site Isolation. In this chapter, we analyze this project, evaluate its effectiveness, and present a new type of attack that is related to a variant of Site Isolation on mobile devices.

In this chapter, we rely mainly on two research studies about Site Isolation. The first [29] was published in 2009 and was the first article that introduced the core idea behind the Site Isolation’s project. The second [44] was published a decade after, in 2019, and presented the status of this project in Chrome along with its evaluation.

8.2 Site Isolation

Site Isolation is the implementation of a multi-process browser’s architecture that provides a way to isolate websites into different processes. It defines the term site as the combination of a protocol and a registry-controlled domain name (e.g., the site of the origin “https://www.facebook.com:8080” is “https://facebook.com”). It also defines the term site instance as the set of same-site execution contexts that are connected, i.e., that can access each other via JS code. The core of this architecture is a mapping between browser’s execution contexts and Renderer processes according to their sites. This mapping can be described by two well-known Chrome policies: 1. process-per-site-instance - all the execution contexts in the same site instance must be processed by the same Renderer process. 2. site-per-process - each Renderer process can only process execution contexts of the same site. For example, if a page with a document of site A embeds using an iframe a document of site B, there will be two separate Renderer processes for documents A and B that render the page. This isolation provides some major security and fault tolerance benefits.

One must wonder why this isolation is performed in the granularity of a site and not of an origin. The reason behind this is that the document’s origin is not fixed
and can be changed by changing the document.domain attribute. However, the value of document.domain must always be under the document’s site value. Therefore, the document’s site value stays the same during the whole lifetime of the document. As a result, two cross-site documents always have the same restrictions imposed on two cross-origin documents. These restrictions are implemented through limiting the set of IPC messages between two Renderer processes to those that correspond to the allowed cross-origin APIs, e.g., the postMessage API. This IPC communication between the two Renderer processes goes through the Browser process. This is a great benefit of Site Isolation because this limited set is much smaller than the full APIs’ set that is supported by the Blink and V8 layers for same-origin operations.

In 2018, Site Isolation became enabled by default in Chrome on desktops and laptops. The reason behind this action was the discovery of the Spectre [45] and Meltdown [46] attacks that enable a memory leakage inside a Renderer process. This discovery made it urgent to isolate websites in dedicated processes to reduce the leakage potential of these attacks to a single site. However, isolating websites in dedicated processes also serve another goal, mitigating against UXSS exploits. These exploits are classified in the Site Isolation’s threat model under the “renderer exploit attacker” type along with other well-known exploits, like RCE.

8.3 Evaluation of Site Isolation

Among 94 SOP bypass exploits (which include also UXSS exploits) that were reviewed in the newer Site Isolation’s research [44], the majority relied on a cross-site iframe. When a cross-site iframe is loaded in the same Renderer process as the Attacker’s document it gives the Attacker a large attack surface for the UXSS exploit, which includes both the Blink and V8 layers. Site Isolation changes the picture because the Attacker and Target services are most likely cross-site and therefore their

![Figure 8.1: Evaluation of Site Isolation - Summary](image)
documents will be processed in different processes. As discussed above, this makes the
attack surface much smaller and thus it makes it much harder to find such exploits.
Another big advantage is that all the communication between the two cross-site docu-
ments goes through the Browser process, which is the privileged process that is harder
to manipulate (in contrast to Renderer processes, which are more vulnerable). This
means that the Browser process can validate the communication between them and
thus it can block messages that should not be sent. For example, the Browser pro-
cess validates that a JS URL is not passed in a navigation request originated from a
cross-site document. Navigation requests with JS URLs were the most exploited API
in Subclass B-1 and now this usage is forbidden by the Browser process.

Although Site Isolation is a great mitigation technique against UXSS exploits, it is
still insufficient to block them entirely. We devote the rest of this section to it.

**Products without Site Isolation.** Site Isolation is not enabled by default in all
Chromium-based products, e.g., it is disabled in mobile devices’ products (like Chrome
on Android and Chromium-based WebView). Site Isolation’s architecture causes an
overhead of memory and CPU because it requires additional Renderer processes to
isolate every encountered site. This overhead is a big problem in mobile devices. To
address this problem in Chrome on Android, instead of isolating all sites, only a small
list of sites are isolated. This list is composed of a static list of built-in sites (contains
some popular websites like Google and Twitter) and a dynamic list of sites. The
dynamic list is composed of websites that the user has visited and filled in his password
in them. This is done by detecting form submissions that contain an input element
type password. The assumption behind this dynamic list is that most of the user’s
sensitive data is in sites that the user logged into and therefore these sites are of higher
priority for isolation. Although this assumption is reasonable, it is not necessarily good
enough. For example, some login mechanisms do not use form submissions with a
password and thus would not trigger changes in the dynamic list, like an Single-Sign-
On [47] (SSO) mechanism that uses OAuth, which is widely used. Another example is
websites that the user does not log into but does enter his sensitive data, like credit
card information. Furthermore, in Chapter 8.4, we present a new type of attack that
is related to this dynamic list mechanism.

In Chromium-based WebView not only that Site Isolation is disabled, but also there
is only one Renderer process for the WebView. Compared to Chrome on Android, the
potential overhead that can be caused by additional processes is even worse because
there is a unique instance of a WebView for each mobile application that uses it. It is
worth noting that each WebView instance has its own storage which has no shared data
(e.g., Cookies store) with Chrome application. Therefore, UXSS exploits are considered
less harmful on WebViews because each instance contains less sensitive data. Despite
that, the risk potential still exists; for example, leaking the user’s browsing history
inside a WebView instance constitutes a privacy problem.
UXSS exploits that are not mitigated by Site Isolation. Site Isolation was designed to mitigate UXSS exploits that require both the use of cross-site iframes and the existence of vulnerabilities in the Blink and V8 layers. However, not all UXSS exploits depend on these assumptions:

- Exploits in Class A have a big variance in the methods used to control the TargetDocument’s resource. Therefore, these exploits may or may not be mitigated by Site Isolation, depending on the specific method which is used by the UXSS exploit.
- Exploits in Subclass B-2 usually depend on exploiting extensions’ code rather than Chromium code and thus Site Isolation does not affect them. Exploits in Subclass B-3 do not depend on direct communication with the TargetDocument. Instead, the exploitation is done through a browser’s component that is permitted to do so. Therefore, these exploits are also unaffected by Site Isolation. Before Site Isolation, exploits in Subclass B-1 used APIs that were implemented solely inside the Renderer process. When Site Isolation is enabled, these APIs are implemented by IPC messages between the Renderer processes that goes through the Browser process in case the documents are cross-site. Therefore, these exploits are mitigated by Site Isolation because it forces them to pass extra checks in the Browser process.
- Class C’s exploits are the ones most affected by Site Isolation. These exploits depend on methods that only the same document (or more accurately, the same site instance) should use, and thus it is unlikely that they could be triggered by a different Renderer process.

In summary, we conclude that Site Isolation does not mitigate all types of UXSS exploits; it mostly affects exploits of Class C and Subclass B-1. However, it is worth noting that they are the biggest subclasses in our classification.

UXSS exploits that are mitigated by Site Isolation. Site Isolation mitigation affects the majority of the UXSS exploits in this classification. However, this mitigation has several limitations and disadvantages, such as: 1. Site Isolation only mitigates cross-site attacks, namely where the Attacker service and the Target service are cross-site. Therefore, if an origin under a site is compromised (e.g., a document under this origin is vulnerable to XSS exploit), then a UXSS exploit can be used to compromise the whole site. 2. After Site Isolation came out, a new type of exploits called Bypassing Site Isolation [44] was published. These exploits make the attacker’s document and a cross-site document share the same Renderer process and thus make the document more vulnerable to UXSS exploits. 3. As explained before, Class C’s UXSS exploits depend on JS Execution Methods that are only available for documents in the same Renderer process as the TargetDocument. Therefore, Site Isolation affects them the most. This is not the case for the UXSS exploits in Subclass B-1.
Exploits in Subclass B-1 use JS Execution APIs that are conveniently included in the limited set of APIs that are implemented between two cross-site documents (which are processed by two different Renderer processes). The implementations of these APIs usually include strict checks that are performed by the Browser process. Due to this mitigation, the attack surface of these APIs is not ideal but still exists. None of the UXSS exploits in Subclass B-1 that were covered in this research managed to fully bypass these strict checks when Site Isolation is enabled. However, the latest UXSS exploit in this classification (CVE-2020-6506) managed to bypass these strict checks in the WebView product. Although site Isolation is disabled there, this exploit would have also worked if it were enabled. With a JS URL, it exploited the JS Execution API of window.open that contains validation checks performed by the Browser process.

8.4 A New Attack on Mobile Devices

In the previous section we described how sites are added to the dynamic list of isolated sites in Chrome on Android. This make these sites better protected against UXSS exploits that are mitigated by Site Isolation. However, during our research, we found a new type of attack that decreases the effectiveness of this isolation process.

We discovered that the content of all the client-side long-term storages (including Cache API, Web Storage API, indexedDB, Web SQL API, and even the Cookies store) stays the same even after a site becomes isolated. Therefore, if an Attacker manages to poison one of these storages before a site becomes isolated, then this storage remains poisoned even after the site becomes isolated. Therefore, this new attack affects websites that are not in the static list of built-in sites and that are vulnerable to a DOM-based XSS exploit via one of these storages. An Attacker can use a UXSS exploit that is mitigated by Site Isolation to poison the vulnerable storage of one of these websites in order to execute JS code in its document even after it become isolated.

The flow of the new attack is presented in Figure 8.2. After this attack, the Attacker can use his UXSS exploit once again, but now in the process of the isolated site, to expand his JS execution to other origins under this site.

Although this attack has many prerequisites, it shows that the mechanism of dynamic list of isolated sites has its disadvantages and that the isolated site’s process can be still vulnerable due to actions that happen before its isolation.
Figure 8.2: New attack of Chrome on Android’s Site Isolation

Before the Target service is isolated:

1. The user enters the Attacker website.
2. The Attacker uses a UXSS exploit to execute JS code in the Target service’s document.
3. Then, the Attacker poison one of the client-side storages of the Target service.

The Target service becomes isolated:

1. The user enters his password in a website under the Target service’s site.
2. The site is added to the dynamic list.

After the Target service is isolated:

1. The user enters the Target service’s website.
2. The DOM-based XSS exploit is activated (it depends on the poisoned client-side storage).
3. The Attacker executes JS code in the Target service’s document, which is inside the isolated process.
Chapter 9

Conclusion

In this thesis, we built the foundations for studying UXSS exploits. These exploits are complex and exploring them can be challenging and time-consuming. The classes we defined in section 5 can simplify this process by identifying the characteristics of these exploits. Furthermore, these classes can be used to improve the mitigation techniques against these exploits. As shown in section 5.6, this classification is relevant not only to Chrome but also to other browsers.

Site Isolation is indeed an effective mitigation used in Chrome against UXSS exploits. However, as shown in section 8.3, it has its limitations, and thus there is room for improvement. Moreover, we found that most of the exploits use only five vulnerability types that mainly depend on bugs in the Renderer process’s code (in the Blink and V8 layers). As part of supporting Site Isolation, additional layers of security were added to the Browser process, which can make these vulnerabilities insufficient for a UXSS exploit. This can lead to new vulnerability types that depend on bugs in the Browser process’s code.

We invite other researchers to use our results and join this research topic. Future researches can focus on other browsers, delve deeper into part of the classes or invent a new mitigation technique against the exploits of a specific class.
Appendices
Appendix A

UXSS exploits classifications

We analyzed 66 UXSS exploits of Chrome to build our classification and then we evaluated its effectiveness by analyzing additional 29 UXSS exploits of Safari.

Table A.1 contains the mapping between Chrome’s UXSS exploits to their classes. Table A.3 is the same for Safari’s UXSS exploits. Table A.2 contains the mapping between the vulnerabilities types (defined in section 5) and the Chrome’s UXSS exploits that use them.

In most cases, we referred to the UXSS exploits by their CVE ids. But, in some cases, there was no CVE, and thus we used their bug tracker’s ids (ids under the Chromium project are of the form “tracker-ID”, and ids under Project Zero are of the form “pz-ID”) or just referred to their link in the references section.
<table>
<thead>
<tr>
<th>Year</th>
<th>Class A</th>
<th>Subclass B-1</th>
<th>Subclass B-2</th>
<th>Subclass B-3</th>
<th>Subclass C-1</th>
<th>Subclass C-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020</td>
<td>CVE-2020-6506</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2019</td>
<td>link [37]</td>
<td>CVE-2019-12592</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2018</td>
<td>pz-1555</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CVE-2018-6101</td>
</tr>
<tr>
<td></td>
<td>tracker-61225</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tracker-594383</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tracker-583445</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2016-5164</td>
<td></td>
<td></td>
<td></td>
<td>tracker-638742</td>
<td>tracker-594383</td>
</tr>
<tr>
<td></td>
<td>tracker-656274</td>
<td></td>
<td></td>
<td>tracker-616225</td>
<td>CVE-2016-5128</td>
<td>tracker-607483</td>
</tr>
<tr>
<td></td>
<td>tracker-583445</td>
<td></td>
<td></td>
<td></td>
<td>tracker-688569</td>
<td>tracker-687844</td>
</tr>
<tr>
<td></td>
<td>tracker-594383</td>
<td></td>
<td></td>
<td>tracker-583445</td>
<td>tracker-594383</td>
<td>tracker-583445</td>
</tr>
<tr>
<td></td>
<td>tracker-583445</td>
<td></td>
<td></td>
<td>tracker-583445</td>
<td>tracker-594383</td>
<td>tracker-583445</td>
</tr>
<tr>
<td>2015</td>
<td>CVE-2015-1303</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2015-1301</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2015-1302</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2015-1292</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2015-1286</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2015-1287</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2015-1210</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>CVE-2014-4671</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A.1: Chrome’s UXSS exploits to classes
Table A.2: Chrome’s UXSS exploits to vulnerabilities types

<table>
<thead>
<tr>
<th>Year</th>
<th>Detached Loaded Frame</th>
<th>Synchronous Navigation to External URL</th>
<th>Cross-Origin Object Leak</th>
<th>Twin Documents</th>
<th>Wrong Security Context Evaluation</th>
</tr>
</thead>
</table>

Table A.3: Safari’s UXSS exploits to classes

<table>
<thead>
<tr>
<th>Year</th>
<th>Class A</th>
<th>Subclass B-1</th>
<th>Subclass B-2</th>
<th>Subclass B-3</th>
<th>Subclass C-1</th>
<th>Subclass C-2</th>
</tr>
</thead>
</table>
Bibliography


The study evaluated the effectiveness of a certain mechanism in reducing attacks on websites. The overall idea is that this mechanism, which is implemented in a web browser, allows for isolation of websites that are attacked. In addition, the mechanism can be used in different contexts, such as isolating websites in a domain with limited access.

In conclusion, the mechanism has proven effective in reducing attacks on websites, especially in domains with limited access. The study recommends further research on this mechanism to improve its effectiveness and expand its use in different contexts.
ניתוח זהandrשת עבודה מקדימה של למידה קוד של קרום (Chromium)מבוסס על פרויקט קוד פתוח V8-1 Blink. מתאימה את התוכן ליתרון קרוםatching תוכן עם שבב JavaScript. עטישת תוכן בקרום ובקרום V8 projektオープン עם מקבץ צירוף בקרום. מ Tangent - מחלקה C-1 בקרום인데 תבנית תומך בקרום C-1 תומך בקרום C-1. matemathikon, בתוכן בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך بקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומך בקרום C-1 תומच...
תקציר

In the field of information security, there are several vulnerabilities and exploits that are used, and most of them are studied in universities (UXSS universality). In this regard, it is necessary to study the exploits of these vulnerabilities for the current lack of research institutions that deal with the exploits of this type. The purpose is to be used as a basis for future research. The aim of the work is to conduct a preliminary analysis of the initial stages of the exploit.

JavaScript exploits are divided into UXSS and XSS vulnerabilities that are not studied in universities. The XSS exploits target the website's users, while the UXSS exploits target the website's users. In this regard, there is a difference in the types of exploits, as the XSS exploits are generally used for the server, while the UXSS exploits are used for the client, and are not capable of bypassing the implemented security mechanisms. Therefore, the UXSS exploits are not considered standard. Nonetheless, the UXSS exploits are used in various contexts and are not considered standard by the client. Therefore, a classification of the exploits is necessary to study them.

In this thesis, we focus on the client, as it is the most common platform. First, we define the initial form of JavaScript. After that, we conduct a study of JavaScript exploits from UXSS, which is suitable for the definition of this work. As the result of this study, we found 83-41 JavaScript exploits from UXSS, which are not considered standard by the client. In order to study these exploits, we conducted a more in-depth study of the JavaScript exploits from UXSS, which are not considered standard by the client. We found 66 JavaScript exploits from UXSS, which are not considered standard by the client. As a result, we conducted a more in-depth study of the JavaScript exploits from UXSS, which are not considered standard by the client.

In conclusion, the JavaScript exploits are divided into two types: UXSS and XSS. The XSS exploits are used for the server, while the UXSS exploits are used for the client. Therefore, a classification of the exploits is necessary to study them. In this regard, a more detailed study of the JavaScript exploits from UXSS is necessary to study them.
המחקר בוצע בהנחייתו של פרופסור רועי פרידמן ודוקטור נועה גלצר, במנהלת המחלקה למחשבים והнтерניטisión.

תודות

ברצונתי להודות למנהלי תלמידי מחברים, פרופסורים רועי פרידמן ודוקטור נועה גלצר, על הובלת המחקר, היחס והתמיכה לאורך כל הממחקר, לאמונה והערכה של בני מעבדה, שתרמו לי במתן הלקחיים וההדרכה, תוך כדי חלקות ואמיניות.

לазвание התואר, לעיני שיפורי, עזרתי לי במתן הלקחיים וההדרכה, 사회ית לי בכבוד ולחמלה של ידוה, ות気に את זה להנאה, לתרום לתרבותו ולחברתי, ולה-trade, המרצה והעあたりית.

בหมายות, מרצ והנהיה.
אנליזה מתדרכת של חלשות

תיאור על מחקר

לשכם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעי במדעי המחשב

حبיק כהן

הושא לוגט לוגט — מרכז טכנולוגיה ליווי
כשל התשפ"יב חיפה
נובמבר 2021
UXSS

אנליזה מתכדמת של חוולשות

חביבה כהן