UNIVERSITÀ DEGLI STUDI
DI TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE
ICT International Doctoral School
University of Trento, Italy

Confused by Path: Analysis of Path Confusion Based Attacks

Seyed Ali Mirheidari

Advisor:

Prof. **Bruno Crispo**
Università degli Studi di Trento, Italy

Examiner:

Prof. **Silvio Ranise**
Fondazione Bruno Kessler, Trento, Italy

Prof. **Stefano Calzavara**
Università Ca' FoscariVenezia, Italy

November 2020

Abstract

URL parser and normalization processes are common and important operations in different web frameworks and technologies. In recent years, security researchers have targeted these processes and discovered high impact vulnerabilities and exploitation techniques. In a different approach, we will focus on semantic disconnect among different framework-independent web technologies (e.g., browsers, proxies, cache servers, web servers) which results in different URL interpretations. We coined the term "Path Confusion" to represent this disagreement and this thesis will focus on analyzing enabling factors and security impact of this problem.

In this thesis, we will show the impact and importance of path confusion in two attack classes including Style Injection by Relative Path Overwrite (RPO) and Web Cache Deception (WCD). We will focus on these attacks as case studies to demonstrate how utilizing path confusion techniques makes targeted sites exploitable. Moreover, we propose novel variations of each attack which would expand the number of vulnerable sites and introduce new attack scenarios. We will present instances which have been secured against these attacks, while being still exploitable with introduced Path Confusion techniques.

To further elucidate the seriousness of path confusion, we will also present the large scale analysis results of RPO and WCD attacks on high profile sites. We present repeatable methodologies and automated path confusion crawlers which detect thousands of sites that are still vulnerable to RPO or WCD only

with specific types of path confusion techniques. Our results attest the severity of path confusion based class of attacks and how extensively they could hit the clients or systems. We analyze some browser-based mitigation techniques for RPO and discuss that WCD cannot be dealt as a common vulnerability of each component; instead it arises when an ecosystem of individually impeccable components ends up in a faulty situation.

Keywords

[Path Confusion, Relative Path Overwrite, Scriptless Attack, Content Delivery Network, Web Cache Deception]

Acknowledgments

First and foremost, I am grateful to my supervisor, Bruno Crispo, for his support, encouragement and valuable insights during my PhD career. I, also, want to thank the professors in my thesis committee, who read my thesis and provided me with their valuable feedbacks.

Next, I would like to thank my friend, Sajjad Arshad, for collaborating with me. It was a great opportunity and I respect his set of skills. I thank Engin Kirda, William Robertson, Kaan Onarlioglu and all members of SecLab in North Eastern University for being such supportive during our collaboration period.

I dedicate this dissertation to the memory of my grandfather Mohammad Ali (Babaei), who not only was the first one to pull me to the field of computer science but also taught me lessons for my whole life journey. I, also, dedicate it to my beloved grandmother Sedigheh (Mamani), for her love, support and being like a second mother for me.

I could not have made it this far without endless love and support of the best mother Nahid who provided for me, always sacrificially. I would like to thank my father Reza and my sister Maryam who believed in me and encouraged me during my education and career path. I would like to dedicate this thesis to the memory of my mother in law, Maryam, who was so enthusiastic to be there for my defense, miss you and rest in peace.

Finally, I dedicate this work to my beloved wife, Mehrnoosh, whose love, unconditional support and care helped me to move forward every single day.

Contents

1 Introduction **3**
1.1 Thesis Contributions 6
1.2 Thesis Outline 8

2 Background & Related Works **9**
2.1 Cross-Site Scripting 9
2.2 Scriptless Attacks 10
2.3 Web Caches 12

3 Path Confusion **15**
3.1 Introduction 15
3.2 Relative Path Overwrite 16
3.2.1 Threat Model 18
3.2.2 Preconditions for RPO Style Attacks 18
3.2.3 Motivation 20
3.3 Web Cache Deception 20
3.3.1 Threat Model 22
3.3.2 Motivation 23

4 Relative Path Overwrite **25**
4.1 Introduction 25
4.2 Methodology 27
4.2.1 Candidate Identification 28

4.2.2 Vulnerability Analysis 29
4.2.3 Exploitability Analysis 33
4.2.4 Limitations 34
4.2.5 Ethical Considerations 35
4.3 Analysis 35
4.3.1 Relative Stylesheet Paths 36
4.3.2 Vulnerable Pages 36
4.3.3 Exploitable Pages 39
4.3.4 Document Types 40
4.3.5 Internet Explorer Framing 43
4.3.6 Anti-Framing Techniques 43
4.3.7 MIME Sniffing 44
4.3.8 Content Management Systems 45
4.4 Countermeasures 46
4.4.1 Observed Mitigations 47
4.4.2 Suggested Defense Techniques 48
4.5 Chapter Summary 48

5 Web Cache Deception **51**
5.1 Introduction 51
5.2 Methodology 54
5.2.1 Stage 1: Measurement Setup 55
5.2.2 Stage 2: Attack Surface Detection 56
5.2.3 Stage 3: WCD Detection 57
5.2.4 Verification and Limitations 59
5.2.5 Ethical Considerations 60
5.3 Measurement Study 62
5.3.1 Data Collection 63
5.3.2 Measurement Overview 63

5.3.3 Vulnerabilities 68
5.3.4 Practical Attack Scenarios 71
5.3.5 Study Summary 74
5.4 Variations on Path Confusion 75
5.4.1 Path Confusion Techniques 77
5.4.2 Results 78
5.5 Empirical Experiments 82
5.5.1 Cache Location 82
5.5.2 Cache Expiration 83
5.5.3 CDN Configurations 85
5.5.4 Lessons Learned 87
5.6 Chapter Summary & Discussion 88

6 Conclusions and Future Directions **91**

Bibliography **93**

List of Tables

4.1 Sample Grouped Web pages. 29
4.2 Narrowing down the Common Crawl to the candidate set used in our analysis (from left to right). 36
4.3 Vulnerable pages and sites in the candidate set. 39
4.4 Exploitable pages and sites in the candidate set (IE using framing). 39
4.5 Quirks mode document types by browser. 41
4.6 Most frequent document types causing all browsers to render in quirks mode, as well as the sites that use at least one such document type. 41
4.7 Summary of document type usage in sites. 42

5.1 Sample URL grouping for attack surface discovery. 56
5.2 Summary of crawling statistics. 63
5.3 Pages, domains, and sites labeled by CDN using HTTP header heuristics. These heuristics simply check for unique vendor-specific strings added by CDN proxy servers. 66
5.4 Response codes observed in the vulnerable data set. 66
5.5 Cache headers present in HTTP responses collected from vulnerable sites. 67
5.6 Types of vulnerabilities discovered in the data. 68
5.7 Attack scenarios. 71

5.8 Response codes observed with successful WCD attacks for each path confusion variation. 79
5.9 Number of unique pages/domains/sites exploited by each path confusion technique. Element (i, j) indicates number of many pages exploitable using the technique in row i, whereas technique in column j is ineffective. 79
5.10 Vulnerable targets for each path confusion variation. 80
5.11 Default caching behavior for popular CDNs, and cache control headers honored by default to prevent caching. 84

List of Figures

3.1 Conversion of Relative Path to Absolute URL 16

3.2 Path Confusion Due to Server-Side URL Mapping 17

3.3 An illustrated example of web cache deception. Path confusion between a web cache and a web server leads to unexpected caching of the victim's private account details. The attacker can then issue a request resulting in a cache hit, gaining unauthorized access to cached private information. 21

4.1 Methodology Overview. *Candidate* pages use relative stylesheet paths, *vulnerable* pages reflect injected style directives, and *exploitable* pages parse and render the injected style when opened in a browser. 27

4.2 Various techniques of **path confusion** and **style injection**. In each example, the first URL corresponds to the regular page, and the second one to the page URL crafted by the attacker. Each HTML page is assumed to reference a stylesheet at `../style.css`, resulting in the browser expanding the stylesheet path as shown in the third URL. **PAYLOAD** corresponds to `%0A{}body{background:NONCE}` (simplified), where `NONCE` is a randomly generated string. 31

4.3 Percentage of the Alexa site ranking in our candidate set. . . 37

4.4 CDF of total and maximum number of relative stylesheets per web page and site, respectively. 38

4.5 Number of sites containing at least one page with a certain document type (ordered by doctype rank). 40

5.1 A high-level overview of our WCD measurement methodology. 54
5.2 Distribution of the measurement data and vulnerable sites across the Alexa Top 5K. 64
5.3 Categories 65
5.4 Five practical **path confusion** techniques for crafting URLs that reference **nonexistent file names**. In each example, the first URL corresponds to the regular page, and the second one to the malicous URL crafted by the attacker. More generally, **nonexistent.css** corresponds to a nonexistent file where **nonexistent** is an arbitrary string and **.css** is a popular static file extension such as .css, .txt, .jpg, .ico, .js etc. 76

List of Publications

Significant portions of this thesis built upon following publications with some revisions and updates.

[1] Seyed Ali Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. "Cached and Confused: Web Cache Deception in the Wild." *In 29th USENIX Security Symposium (USENIX Security 20), pp. 665-682. 2020.*

[2] Sajjad Arshad, Seyed Ali Mirheidari, Tobias Lauinger, Bruno Crispo, Engin Kirda, and William Robertson. "Large-Scale Analysis of Style Injection by Relative Path Overwrite." *In Proceedings of the 2018 World Wide Web (WWW) Conference, pp. 237-246. 2018.*

Chapter 1

Introduction

Today's World Wide Web has grown fully fledged and heterogeneous technologies and frameworks have been introduced to increase security and availability. Most of these technologies (e,g. browser, cache, load balancer, web application firewall, web server, etc.) need to deeply understand and parse HTTP requests to serve the response to clients. More precisely, parsing and normalization are considered as initial and important parts of almost all of HTTP processing-chain technologies.

A Uniform Resource Locator (URL) is composed of different components, e.g., scheme, authority, host, path, query and fragment. Any technology uses its own method to parse a URL and extract different components in order to apply pre-defined rules and policies. For more than a decade, security experts and researchers have targeted each URL parser implementation. Different classes of vulnerabilities have been discovered and exploited for each specific technology and web framework. For instance, Orange Tsai presented a series of exploitation techniques that take advantage of the quirks of built-in URL parsers in popular programming languages and web frameworks [129, 130].

The important point here is that each HTTP processing-chain technology autonomously parses and normalizes the HTTP requests and URLs. A major overlooked problem arises when distinct parts of the processing chain

have disagreements on their interpretations of the URL components, hence different paths would be established. In other words, in this class of vulnerabilities, there is no problem about individual interpretation but the whole established path would not be correctly induced. Despite studies focused on each component, literature is so sparse on how drastic the result of such disagreements in path interpretation could be.

A fitting example of this category of vulnerability is the one which targets the `Host` header in a HTTP request. Chen et al [23] shows how inconsistent interpretation of a HTTP request with ambiguous host fields (e.g., with multiple `Host` headers) between two different technologies can impose security risks. This disagreement could be studied in different components of a URL and might expose the system to different attack scenarios and security risks. Our focus in this thesis is URL `Path` component.

Web application attacks such as cross-site scripting or Cross-site request forgery, are well-understood and studied by both academics and the general security community. However, as mentioned previously, the security implications of path confusion have started to garner attention only recently, and academic literature on the subject is sparse. Even though these attacks have been dealt with as separate scenarios, we realized them to be all stemming in the disagreement among different web components' URL interpretation. We coined the name *Path Confusion* and investigate the problem more in depth in this thesis. In the following, we will discuss Relative Path Overwrite and Web Cache Deception as two samples of path confusion based attacks.

Relative Path Overwrite. Relative Path Overwrite (RPO) is a recent technique to inject style directives into sites even when no style sink or markup injection vulnerability is present. It exploits path confusion differences in how browsers and web servers interpret relative paths to make a HTML page reference itself as a stylesheet; a simple text injection vulnerability along with

browsers' leniency in parsing CSS resources results in an attacker's ability to inject style directives that will be interpreted by the browser. Even though style injection may appear less serious a threat than script injection, it has been shown that it enables a range of attacks, including secret exfiltration.

Web Cache Deception. Web cache deception (WCD) is an attack proposed in 2017, where an attacker tricks a caching proxy into erroneously storing private information transmitted over the Internet and subsequently gains unauthorized access to that cached data. Web cache technologies may be configured to make their caching decisions based on complex rules such as pattern matches on file names, paths, and header contents. Launching a successful WCD attack requires an attacker to craft a malicious URL that triggers a caching rule, but also one that is interpreted as a legitimate request by the web server.

For the first time, we extend the exploitability of this class of attacks by developing novel attack techniques. We evaluate them through novel and repeatable methodologies in large scales that show to which extent these attacks could be impactful. This area of research is so sparse and possible attacks and their impacts are overlooked. We explore mentioned path confusion based attacks and introduce new variations of them that would expand the number of vulnerable systems. Moreover, we propose unprecedented path confusion based techniques and analyze them at scale.

We shed light on path confusion based attacks as well as their severity by analyzing them in the wild. Moreover, we introduce novel variations of path confusion to increase the likelihood of exploitability. Descriptions of our thesis contributions and outline are detailed in the following sections.

1.1 Thesis Contributions

In this thesis, we investigate two different path confusion based attack scenarios which stem in path interpretation disagreement among different system components, each of which is not necessarily faulty but would result in a malfunctioning system, vulnerable to path confusion attacks. Our proposed attack scenarios and contributions in each part have been specified as follows.

- We present the first large scale study of the Web through a repeatable methodology to measure the prevalence and significance of style injection using RPO in the wild. Our work shows that around 9 % of the sites in the Alexa Top 10K contain at least one vulnerable page, out of which more than one third can be exploited.
- We introduce new variations of RPO and study the popularity of them among top 1M Alexa. We also conduct an assessment of novel path confusion techniques and their impact on exploitation of RPO vulnerabilities.
- We analyze and discuss in detail various impediments and range of factors that prevent a RPO vulnerability from being successfully exploited, and make recommendations for remediations.
- We propose a novel, repeatable methodology to detect sites impacted by WCD at scale. Unlike existing WCD scan tools that are designed for site administrators to test their own properties in a controlled environment, our methodology is designed to automatically detect WCD in the wild.
- We present findings that quantify the prevalence of WCD in 295 sites among the Alexa Top 5K, and provide a detailed breakdown of leaked information types. Our analysis also covers security tokens that can be stolen via WCD as well as novel security implications of the attack, all areas left unexplored by existing WCD literature.

- We introduce the new variation of WCD using different path confusion techniques that would expand the number of vulnerable systems. We conduct a follow-up measurement over 340 sites among the Alexa Top 5K that show variations on the path confusion technique to make it possible to successfully exploit sites that are not impacted by the original WCD attack. Our proposed attack and research was voted and led to an award as the *top web hacking technique of 2019* by the hacking and research community and the PortSwigger panel [22, 108].
- We analyze the default settings of popular CDN providers and document their distinct caching behavior, highlighting that mitigating WCD necessitates a comprehensive examination of a website's infrastructure.

1.2 Thesis Outline

This dissertation is organized as follows. Chapter 2 discusses preliminary concepts and related attacks which would be later be utilized during the thesis. In Chapter 3, we provide a concise description of the Path Confusion problem which would be studied in more detail in the wild throughout the next chapters. Chapter 4 investigates RPO in the wild by introducing novel variation of path confusion techniques in an automated methodology and present results of large scale analysis of style injection. In Chapter 5, we present the large-scale measurement and a detailed analysis of WCD and investigate the root cause of it among popular CDNs. We conclude the thesis by discussions and conclusions in Chapter 6.

Chapter 2

Background & Related Works

2.1 Cross-Site Scripting

Many sites have vulnerabilities that let attackers inject malicious script. Dynamic sites frequently accept external inputs that can be controlled by an attacker, such as data in URLs, cookies, or forms. While the site developer's aim would have been to render the input as text, lack of proper sanitization can result in the input being executed as script [106]. The inclusion of unsanitized inputs could occur on the server side or client side, and in a persistent *stored* or volatile *reflected* way [103]. To the victim's web browser, the code appears as originating from the first-party site, thus it is given full access to the session data in the victim's browser. Thereby, the attacker bypasses protections of the Same-Origin Policy.

The script-based attacks has been studied extensively, such as systematic analysis of XSS sanitization frameworks [140], detecting XSS vulnerabilities in Rich Internet Applications [3], large-scale detection of DOM-based XSS [76, 85], persistent client-side XSS [119], and bypassing XSS mitigations by Script Gadgets [75, 74]. An array of XSS prevention mechanisms have been proposed, such as XSS Filter [110], XSS-Guard [16], SOMA [100], BluePrint [86], Document Structure Integrity [95], XSS Auditor [14], NoScript [88], Context-Sensitive Auto-Sanitization (CSAS) [112], DOM-based

XSS filtering using runtime taint tracking [120], preventing script injection through software design [64], Strict CSP [139], ScriptProtect [94], and DOMPurify [50]. However, the vulnerability measurements and proposed countermeasures of these works on script injection do not apply to scriptless injection attack.

2.2 Scriptless Attacks

Cross-Site Scripting is perhaps the most well-known web-based attack, against which many sites defend by filtering user input. Client-side security mechanisms such as browser-based XSS filters [14] and Content Security Policy [118, 136] also make it more challenging for attackers to exploit injection vulnerabilities for XSS. This has led attackers (and researchers) to investigate potential alternatives, such as *scriptless* attacks. These attacks allow sniffing users' browsing histories [83, 59], exfiltrating arbitrary content [70], reading HTML attributes [52, 73], and bypassing Clickjacking defenses [52]. In the following, we highlight two types of scriptless attacks proposed in the literature. Both assume that an attacker cannot inject or execute script into a site. Instead, the attacker abuses features related to Cascading Style Sheets (CSS).

Heiderich et al. [49] consider scenarios where an attacker can inject CSS into the context of the third-party page so that the style directives are interpreted by the victim's browser when displaying the page. That is, the injection sink is either located inside a style context, or the attacker can inject markup to create a style context around the malicious CSS directives. While the CSS standard is intended for styling and layout purposes such as defining sizes, colors, or background images and as such does not contain any traditional scripting capabilities, it does provide some context-sensitive features that, in combination, can be abused to extract and exfiltrate data.

If the secret to be extracted is not displayed, such as a token in a hidden form field or link URL, the attacker can use the CSS attribute accessor and content property to extract the secret and make it visible as text, so that style directives can be applied to it. Custom attacker-supplied fonts can change the size of the secret text depending on its value. Animation features can be used to cycle through a number of fonts in order to test different combinations. Media queries or the appearance of scrollbars can be used to implement conditional style, and data exfiltration by loading a different URL for each condition from the attacker's server. Taken together, Heiderich et al. demonstrate that these techniques allow an attacker to steal credit card numbers or CSRF tokens [105] without script execution.

Rather than using layout-based information leaks to exfiltrate data from a page, Huang et al. [56] show how syntactically lax parsing of CSS can be abused to make browsers interpret an HTML page as a "stylesheet." The attack assumes that the page contains two injection sinks, one before and one after the location of the secret in the source code. The attacker injects two CSS fragments such as `{}*{background:url('//attacker.com/?` and `');}`, which make the secret a part of the URL that will be loaded from the attacker's server when the directive is interpreted. It is assumed that the attacker cannot inject markup, thus the injected directive is not interpreted as style when the site is conventionally opened in a browser. However, the CSS standard mandates that browsers be very forgiving when parsing CSS, skipping over parts they do not understand [135]. In practice, this means that an attacker can set up a site that loads the vulnerable third-party site *as a stylesheet*. When the victim visits the attacker's site while logged in, the victim's browser loads the third-party site and interprets the style directive, causing the secret to be sent to the attacker. To counter this attack, modern browsers do not load documents with non-CSS content types and syntax errors as stylesheets when they originate from a different domain than the

including page. Yet, attacks based on tolerant CSS parsing are still feasible when both the including and the included page are loaded from the same domain. Relative Path Overwrite attacks can abuse such a scenario [143].

2.3 Web Caches

Repeatedly transferring heavily used and large web objects over the Internet is a costly process for both web servers and their end-users. Multiple round-trips between a client and server over long distances, especially in the face of common technical issues with the Internet infrastructure and routing problems, can lead to increased network latency and result in web applications being perceived as unresponsive. Likewise, routinely accessed resources put a heavy load on web servers, wasting valuable computational cycles and network bandwidth. The Internet community has long been aware of these problems, and deeply explored caching strategies and technologies as an effective solution.

Today web caches are ubiquitous, and are used at various—and often multiple—steps of Internet communications. For instance, client applications such as web browsers implement their own *private* cache for a single user. Otherwise, web caches deployed together with a web server, or as a man-in-the-middle proxy on the communication path implement a *shared* cache designed to store and serve objects frequently accessed by multiple users. In all cases, a cache hit eliminates the need to request the object from the origin server, improving performance for both the client and server.

In particular, web caches are a key component of Content Delivery Networks (CDN) that provide web performance and availability services to their users. By deploying massively-distributed networks of shared caching proxies (also called *edge servers*) around the globe, CDNs aim to serve as many requests as possible from their caches deployed closest to clients, offloading the

origin servers in the process. As a result of multiple popular CDN providers that cover different market segments ranging from simple personal sites to large enterprises, web caches have become a central component of the Internet infrastructure. A recent study by Guo et al. estimates that 74% of the Alexa Top 1K make use of CDNs [47].

The most common targets for caching are static but frequently accessed resources. These include static HTML pages, scripts and style sheets, images and other media files, and large document and software downloads. Due to the shared nature of most web caches, objects containing dynamic, personalized, private, or otherwise sensitive content are not suitable for caching. We point out that there exist technologies such as Edge Side Includes [131] that allow caching proxies to assemble responses from a cached static part and a freshly-retrieved dynamic part, and the research community has also explored caching strategies for dynamic content. That being said, caching of non-static objects is not common, and is not relevant to WCD attacks. Therefore, it will not be discussed further in this chapter.

The HTTP/1.1 specification defines `Cache-Control` headers that can be included in a server's response to signal to all web caches on the communication path how to process the transferred objects [40]. For example, the header `‘‘Cache- Control: no-store’’` indicates that the response should not be stored. While the specification states that web caches *MUST* respect these headers, web cache technologies and CDN providers offer configuration options for their users to ignore and override header instructions. Indeed, a common and easy configuration approach is to create simple caching rules based on resource paths and file names, for instance, instructing the web cache to store all files with extensions such as `jpg`, `ico`, `css`, or `js` [35, 28].

Caching mechanisms in many Internet technologies (e.g., ARP, DNS) have been targeted by *cache poisoning* attacks, which involve an attacker storing a malicious payload in a cache later to be served to victims. For example,

James Kettle recently presented practical cache poisoning attacks against caching proxies [68, 69]. Likewise, Nguyen et al. demonstrated that negative caching (i.e., caching of 4xx or 5xx error responses) can be combined with cache poisoning to launch denial-of-service attacks [98]. Although the primary goal of a cache poisoning attack is malicious payload injection and not private data disclosure, these attacks nevertheless manipulate web caches using mechanisms similar to web cache deception. Hence, these two classes of attacks are closely related.

More generally, the complex ecosystem of CDNs and their critical position as massively-distributed networks of caching reverse proxies have been studied in various security contexts [122, 47]. For example, researchers have explored ways to use CDNs to bypass Internet censorship [54, 145, 41], exploit or weaponize CDN resources to mount denial-of-service attacks [128, 24], and exploit vectors to reveal origin server addresses behind proxies [134, 61]. On the defense front, researchers have proposed techniques to ensure the integrity of data delivered over untrusted CDNs and other proxy services [78, 80, 91].

Chapter 3

Path Confusion

3.1 Introduction

Traditionally, URLs referenced web resources by directly mapping them to a web server's filesystem structure, followed by a list of query parameters. For instance, `example.com/home/index.html?lang=en` would correspond to the file `home/index.html` at that web server's document root directory, and `lang=en` represents a parameter indicating the preferred language.

Since two syntactically different URLs could be correspondent, URL normalization process has been utilized to determine whether different URLs are referring to the same web resource location on different frameworks, e.g. web browsers, proxies, cache servers and web servers. As every different web component has implemented URL normalization differently from others, there are different attacks which may target any of these components.

While web applications grew in size and complexity, web servers introduced sophisticated URL rewriting mechanisms to implement advanced application routing structures as well as to improve usability and accessibility. In other words, web servers parse, process, and interpret URLs in ways that are not clearly reflected in the externally-visible representation of the URL string. Consequently, browser, the rest of the communication endpoints and man-in-the-middle entities may remain oblivious to this additional layer of

```
 Page URL: http://example.com/rpo/test.php
Style Path: dist/styles.css

Style URL: http://example.com/rpo/dist/styles.css
```

Figure 3.1: Conversion of Relative Path to Absolute URL

abstraction between the resource filesystem path and its URL, and process the URL in an unexpected—and potentially unsafe—manner. We coin the term *path confusion* to describe this phenomenon.

We inspect the path confusion based attacks in two classes of Relative Path Overwrite (RPO) and Web cache deception (WCD) in the following sections.

3.2 Relative Path Overwrite

Relative Path Overwrite vulnerabilities can occur in sites that use relative paths to include resources such as scripts or stylesheets. Before a web browser can issue a request for such a resource to the server, it must expand the relative path into an absolute URL. For example in Figure 3.1, assume that a web browser has loaded an HTML document from `http://example.com/rpo/test.php` which references a remote stylesheet with the relative path `dist/styles.css`. Web browsers treat URLs as file system-like paths, that is, `test.php` would be assumed to be a file within the parent directory `rpo/`, which would be used as the starting point for relative paths, resulting in the absolute URL `http://example.com/rpo/dist/styles.css`.

However, the browser's interpretation of the URL may be very different from how the web server resolves the URL to determine which resource should be returned to the browser. The URL may not correspond to an actual server-side file system structure at all, or the web server may internally rewrite parts of the URL. For instance, when a web server receives a request for `http:`

```
 Page URL: http://example.com/rpo/test.php/
Style Path: dist/styles.css

Style URL: http://example.com/rpo/test.php/dist/styles.css
```

Figure 3.2: Path Confusion Due to Server-Side URL Mapping

`//example.com/rpo/test.php/` with an added trailing slash as shown in Figure 3.2, it may still return the same HTML document corresponding to the `test.php` resource. Yet, to the browser this URL would appear to designate a directory (without a file name component), thus the browser would request the stylesheet from `http://example.com/rpo/test.php/dist/styles.css`. Depending on the server configuration, this may either result in an error since no such file exists, or the server may interpret `dist/styles.css` as a parameter to the script `test.php` and return the HTML document. In the latter case, the HTML document includes itself as a stylesheet. Provided that the document contains a (text) injection vulnerability, attackers can carry out the scriptless attacks; since the stylesheet inclusion is same-origin, the document load is permitted.

The first account of RPO is attributed to a blog post by Gareth Heyes [53], introducing self-referencing a PHP script with server-side URL rewriting. Furthermore, the post notes that certain versions of Internet Explorer allow JavaScript execution from within a CSS context in the *Compatibility View* mode [92], escalating style injection to XSS [142]. Another blog post by Dalili [32] extends the technique to IIS and ASP.Net applications, and shows how URL-encoded slashes are decoded by the server but not the browser, allowing not only self-reference but also the inclusion of different resources. Kettle [66] coins the term Path Relative StyleSheet Import (PRSSI) for a specific subset of RPO attacks, introduces a PRSSI vulnerability scanner for Burp Suite [18], and proposes countermeasures. Terada [125] provides more exploitation techniques for various browsers or certain web applications,

and [143] discusses an example chaining several vulnerabilities to result in a combination of RPO and a double style injection attack. Gil shows how attackers can deceive web cache servers by using RPO [42, 43]. Some of the attacks discussed in the various blog posts are custom-tailored to specific sites or applications, whereas others are more generic and apply to certain web server configurations or frameworks.

3.2.1 Threat Model

The general threat model of Relative Path Overwrite (RPO) resembles that of Cross-Site Scripting (XSS). Typically, the attacker's goal is to steal sensitive information from a third-party site or make unauthorized transactions on the site, such as gaining access to confidential financial information or transferring money out of a victim's account. The attacker carries out the attack against the site indirectly, by way of a victim that is an authorized user of the site. The attacker can trick the victim into following a crafted link, such as when the victim visits a domain under the attacker's control and the page automatically opens the manipulated link, or through search engine poisoning, deceptive shortened links, or through means of social engineering.

3.2.2 Preconditions for RPO Style Attacks

In this thesis, we focus on a generic type of RPO attack because its preconditions are less specific and are likely met by a larger number of sites. More formally, we define a page as *vulnerable* if:

- The page includes at least one stylesheet using a relative path.
- The server is set up to serve the same page even if the URL is manipulated by appending characters that browsers interpret as path separators.

- The page reflects style directives injected into the URL or cookie. Note that the reflection can occur in an arbitrary location within the page, and markup or script injection are not necessary.

- The page does not contain a `<base>` HTML tag before relative paths that would let the browser know how to correctly expand them.

This attack corresponds to style injection by means of a page that references itself as a stylesheet (PRSSI). Since the "stylesheet" self-reference is, in fact, an HTML document, web servers would typically return it with a `text/html` content type. Browsers in standards-compliant mode do not attempt to parse documents with a content type other than CSS even if referenced as a stylesheet, causing the attack to fail. However, web browsers also support *quirks mode* for backwards compatibility with non-standards compliant sites [115]; in this mode, browsers ignore the content type and parse the document according to the inclusion context only.

We define a vulnerable page as *exploitable* if the injected style is interpreted by the browser–that is, if an attacker can force browsers to render the page in quirks mode. This can occur in two alternative ways:

- The vulnerable HTML page specifies a *document type* that causes the browser to use quirks mode instead of standards mode. The document type indicates the HTML version and dialect used by the page; Section 4.3.4 provides details on how the major web browsers interpret the document types we encountered during our study.

- Even if the page specifies a document type that would usually result in standards mode being used, quirks mode parsing can often be enforced in Internet Explorer [66]. Framed documents inherit the parsing mode from the parent document, thus an attacker can create an attack page with an older document type and load the vulnerable page into a frame.

This trick only works in Internet Explorer, however, and it may fail if the vulnerable page uses any anti-framing technique, or if it specifies an explicit value for the `X-UA-Compatible` HTTP header (or equivalent).

Our measurement methodology in Section 4.2 tests how often these preconditions hold in the wild in order to quantify the vulnerability and exploitability of pages with respect to RPO attacks.

3.2.3 Motivation

In the previous section, we surveyed a number of style-based attacks in the scientific literature, and several blog posts discussing special cases of RPO. We are not aware of any scholarly work about RPO, or any research about how prevalent RPO vulnerabilities are on the Web. To the best of our knowledge, Burp Suite [18] is the first and only tool that can detect PRSSI vulnerabilities based on RPO in web applications. However, in contrast to our work, it does not determine if the vulnerability can be exploited. Furthermore, we are the first to provide a comprehensive survey of how widespread RPO style vulnerabilities and exploitabilities are in the wild.

3.3 Web Cache Deception

The widespread use of *clean URLs* (also known as *RESTful URLs*) help illustrate the issues resulting from different interpretations of a URL. Clean URL schemes use structures that abstract away from a web server's internal organization of resources, and instead provide a more readable API-oriented representation. For example, a given web service may choose to structure the URL `example.com/index.php?p1=v1&p2=v2` as `example.com/index/v1/v2` in clean URL representation.

Now, consider the case where a user accesses the same web service using

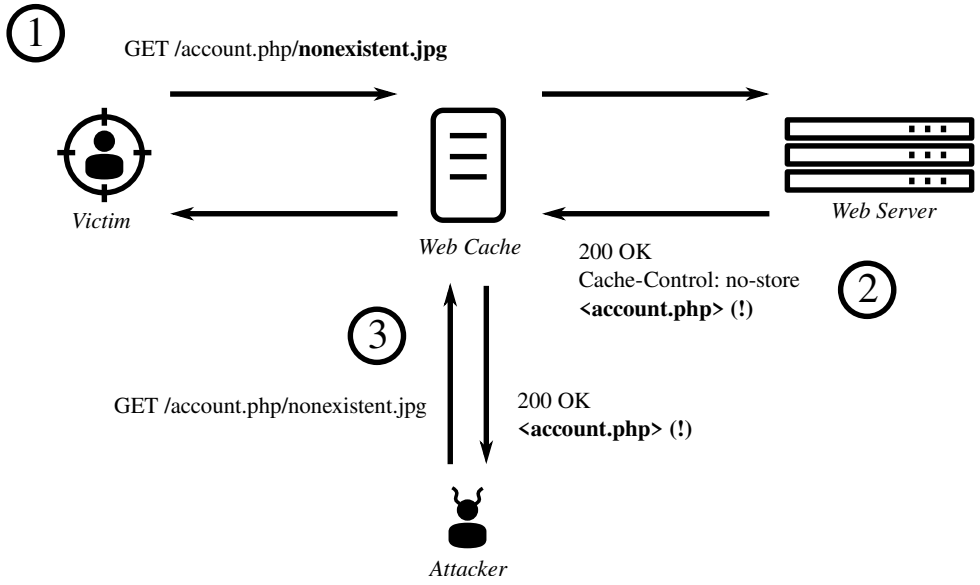

Figure 3.3: An illustrated example of web cache deception. Path confusion between a web cache and a web server leads to unexpected caching of the victim's private account details. The attacker can then issue a request resulting in a cache hit, gaining unauthorized access to cached private information.

the URL `example.com/index/img/pic.jpg`. While different URL normalizations might be applied by different components of the system, there is still possible for the path to be inconsistently interpreted by different partners. The user's browser and all technologies in the communication path (e.g., the web browser, caches, load balancers, proxies, web application firewalls) are likely to misinterpret this request, expect an image file in return, and treat the HTTP response accordingly (e.g., web caches may choose to store the response payload).

However, in reality, the web service will internally map this URL to `example.com/index.php?p1=img&p2=pic.jpg`, and return the contents of `index.php` with an HTTP 200 status code. Note that even when `img/pic.jpg` is an arbitrary resource that does not exist on the web server, the HTTP 200 status code will falsely indicate that the request was successfully handled as intended. Described behaviour is the root cause of Web Cache Deception

attack.

WCD is a recently-discovered manifestation of path confusion that an attacker can exploit to break the confidentiality properties of a web application. This may result in unauthorized disclosure of private data belonging to end-users of the target application, or give the attacker access to sensitive security tokens (e.g., CSRF tokens) that could be used to facilitate further web application attacks by compromising authentication and authorization mechanisms. Gil proposed WCD in 2017, and demonstrated its impact with a practical attack against a major online payment provider, PayPal [44, 45].

3.3.1 Threat Model

In order to exploit a WCD vulnerability, the attacker crafts a URL that satisfies two properties:

1. The URL must be interpreted by the web server as a request for a non-cacheable page with private information, and it should trigger a successful response.

2. The same URL must be interpreted by an intermediate web cache as a request for a static object matching the caching rules in effect.

Next, The attacker uses social engineering channels to lure a victim into visiting this URL, which would result in the incorrect caching of the victim's private information. The attacker would then repeat the request and gain access to the cached contents. Fig. 3.3 illustrates these interactions.

In *Step 1* , the attacker tricks the victim into visiting a URL that requests `/account.php/nonexistent.jpg`. At a first glance this appears to reference an image file, but in fact does not point to a valid resource on the server.

In *Step 2* , the request reaches the web server and is processed. The server in this example applies rewrite rules to discard the non-existent part of the requested object, a common default behavior for popular web servers

and application frameworks. As a result, the server sends back a success response, but actually includes the contents of `account.php` in the body, which contains private details of the victim's account. Unaware of the URL mapping that happened at the server, the web cache stores the response, interpreting it as a static image.

Finally, in *Step 3* , the attacker visits the same URL which results in a cache hit and grants him unauthorized access to the victim's cached account information.

Using references to non-existent cacheable file names that are interpreted as path parameters is an easy and effective path confusion technique to mount a WCD attack, and is the original attack vector proposed by Gil. However, we discuss novel and more advanced path confusion strategies in Sec. 5.4. Also note that the presence of a `Cache-Control: no-store` header value has no impact in our example, as it is common practice to enable caching rules on proxy services that simply ignore header instructions and implement aggressive rules based on path and file extension patterns (see Sec. 2.3).

3.3.2 Motivation

WCD garnered significant media attention due to its security implications and high damage potential. Major web cache technology and CDN providers also responded, and some published configuration hardening guidelines for their customers [84, 17, 15]. More recently, Cloudflare announced options for new checks on HTTP response content types to mitigate the attack [25].

Researchers have also published tools to scan for and detect WCD, for instance, as an extension to the Burp Suite scanner or as stand-alone tools [57, 116]. We note that these tools are oriented towards penetration testing, and are designed to perform targeted scans on web properties directly under the control of the tester. That is, by design, they operate under certain pre-conditions, perform information disclosure tests via simple similarity and

edit distance checks, and otherwise require manual supervision and interpretation of the results. This is orthogonal to the methodology and findings we present in Chapter 5. Our experiment is, instead, designed to discover WCD vulnerabilities at scale in the wild, and does not rely on page similarity metrics that would result in an overwhelming number of false positives in an uncontrolled test environment.

Chapter 4

Relative Path Overwrite

4.1 Introduction

Cross-Site Scripting (XSS) [103] attacks are one of the most common threats on the Web. While XSS has traditionally been understood as the attacker's capability to inject script into a site and have it executed by the victim's web browser, more recent work has shown that script injection is not a necessary precondition for effective attacks. By injecting Cascading Style Sheet (CSS) directives, for instance, attackers can carry out so-called *scriptless* attacks [49] and exfiltrate secrets from a site.

The aforementioned injection attacks typically arise due to the lack of separation between code and data [36], and more specifically, insufficient sanitization of untrusted inputs in web applications. While script injection attacks are more powerful than those based on style injection, they are also more well-known as a threat, and web developers are comparatively more likely to take steps to make them more difficult. From an attacker's point of view, style injection attacks may be an option in scenarios where script injection is not possible.

There are many existing techniques of how style directives could be injected into a site [49, 56]. A relatively recent class of attacks is Relative Path Overwrite (RPO), first proposed in a blog post by Gareth Heyes [53] in

2014. These attacks exploit the semantic disconnect between web browsers and web servers in interpreting relative paths (*path confusion*). More concretely, in certain settings an attacker can manipulate a page's URL in such a way that the web server still returns the same content as for the benign URL. However, using the manipulated URL as the base, the web browser incorrectly expands relative paths of included resources, which can lead to resources being loaded despite not being intended to be included by the developer. Depending on the implementation of the site, different variations of RPO attacks may be feasible. For example, an attacker could manipulate the URL to make the page include user-generated content hosted on the same domain [125]. When an injection vulnerability is present in a page, an attacker could manipulate the URL such that the web page references itself as the stylesheet, which turns a simple text injection vulnerability into a style sink [53]. Among these attack instantiations, the latter variant has preconditions that are comparatively frequently met by sites. Our work focuses on this variant of RPO.

To date, little is known about how widespread RPO vulnerabilities are on the Web. Especially since the attack is more recent and less well-known than traditional XSS, we believe it is important to characterize the extent of the threat and quantify its enabling factors. In this chapter, we present the first in-depth study of style injection vulnerability using RPO. We extract pages using relative-path stylesheets from the Common Crawl dataset [31], automatically test if style directives can be injected using RPO, and determine whether they are interpreted by the browser. Out of 31 million pages from 222 thousand Alexa Top 1 M sites [8] in the Common Crawl that use relative-path stylesheets, we find that 377 k pages (12 k sites) are vulnerable; 11 k pages on 1 k sites can be exploited in Chrome, and nearly 55 k pages on over 3 k sites can be exploited in Internet Explorer. We analyze a range of factors that prevent a vulnerable page from being exploited, and discuss how

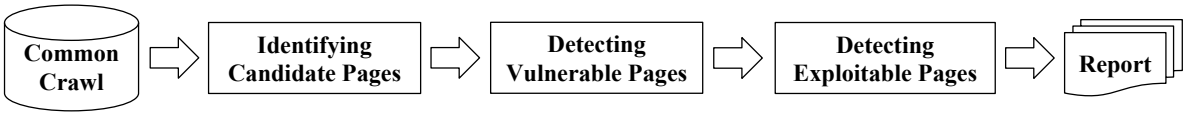

Figure 4.1: Methodology Overview. *Candidate* pages use relative stylesheet paths, *vulnerable* pages reflect injected style directives, and *exploitable* pages parse and render the injected style when opened in a browser.

these could be used to mitigate these vulnerabilities.

The contributions of this chapter are summarized as follows:

- First automated and large-scale study of the prevalence and significance of RPO vulnerabilities in the wild.
- Analysis of various path confusion techniques and their impact on exploitation of RPO vulnerabilities.
- Discussing a range of factors that prevent a vulnerability from being exploited, and find that simple countermeasures exist to mitigate RPO.
- Linking many exploitable pages to installations of Content Management Systems (CMSes), and notify the vendors.

4.2 Methodology

Our methodology consists of three main phases , as shown in Figure 4.1: we seed our system with pages from the Common Crawl archive to extract *candidate* pages that include at least one stylesheet using a relative path. To determine whether these candidate pages are *vulnerable*, we attempt to inject style directives by requesting variations of each page's URL to cause *path confusion* and test whether the generated response reflects the injected style directives. Finally, we test how often vulnerable pages can be *exploited* by checking whether the reflected style directives are parsed and used for rendering in a web browser.

4.2.1 Candidate Identification

For finding the initial seed set of candidate pages with relative-path stylesheets, we leverage the Common Crawl from August 2016, which contains more than 1.6 billion pages. By using an existing dataset, we can quickly identify candidate pages without creating any web crawl traffic. We use a Java HTML parser to filter any pages containing only inline CSS or stylesheets referenced by absolute URLs, leaving us with over 203 million pages on nearly 6 million sites. For scalability purposes, we further reduce the set of candidate pages in two steps:

1. We retain only pages from sites listed in the Alexa Top 1 million ranking, which reduces the number of candidate pages to 141 million pages on 223 thousand sites. In doing so, we bias our result toward popular sites– that is, sites where attacks could have a larger impact because of the higher number of visitors.

2. We observed that many sites use templates customized through query strings or path parameters. We expect these templates to cause similar vulnerability and exploitability behavior for their instantiations, thus we can speed up our detection by grouping URLs using the same template, and testing only one random representative of each group.

   In order to group pages, we replace all the values of query parameters with constants, and we also replace any number identifier in the path with a constant. We group pages that have the same abstract URL as well as the same document type in the Common Crawl dataset. Table 4.1 shows instances of grouped web pages by either query values or path parameters.

Since our methodology contains a step during which we actively test whether a vulnerability can be exploited, we remove from the candidate set

Table 4.1: Sample Grouped Web pages.

Group By	**Web page**
Query Value	http://www.example.com/?lang=**en** http://www.example.com/?lang=**it**
Path Parameter	http://www.example.com/**028** http://www.example.com/**142**

all pages hosted on sites in `.gov`, `.mil`, `.army`, `.navy`, and `.airforce`. The final candidate set consists of 137 million pages (31 million page groups) on 222 thousand sites.

4.2.2 Vulnerability Analysis

To determine whether a candidate page is vulnerable, we implemented a lightweight crawler based on the Python Requests module. At a high level, the crawler simulates how a browser expands relative paths and tests whether style directives can be injected into the resources loaded as stylesheets using path confusion.

For each page group from the candidate set, the crawler randomly selects one representative URL and mutates it according to a number of techniques explained below. Each of these techniques aims to cause path confusion and taints page inputs with a style directive containing a long unique, random string. The crawler requests the mutated URL from the server and parses the response document, ignoring resources loaded in frames. If the response contains a `<base>` tag, the crawler considers the page not vulnerable since the `<base>` tag, if used correctly, can avoid path confusion. Otherwise, the crawler extracts all relative stylesheet paths from the response and expands them using the mutated URL of the main page as the base, emulating how browsers treat relative paths (see Section 3.2). The crawler then requests each unique stylesheet URL until one has been found to reflect the injected

style in the response.

The style directive we inject to test for reflection vulnerabilities is shown in the legend of Figure 4.2. The payload begins with an encoded newline character, as we observed that the presence of a newline character increases the probability of a successful injection. We initially use `%0A` as the newline character, but also test `%0C` and `%0D` in case of unsuccessful injection. The remainder of the payload emulates the syntax of a simple CSS directive and mainly consists of a randomly generated string used to locate the payload in the body of the server response. If the crawler finds a string match of the injected unique string, it considers the page vulnerable.

In the following, we describe the various URL mutation techniques we use to inject style directives. All techniques also use RPO so that instead of the original stylesheet files, browsers load different resources that are more likely to contain an injection vulnerability. Conceptually, the RPO approaches we use assume some form of server-side URL rewriting as described in Section 3.2. That is, the server internally resolves a crafted URL to the same script as the "clean" URL. Under that assumption, the path confusion caused by RPO would result in the page referencing itself as the stylesheet when loaded in a web browser. However, this assumption is only conceptual and not necessary for the attack to succeed. For servers that do not internally rewrite URLs, our mutated URLs likely cause error responses since the URLs do not correspond to actual files located on these servers. Error responses are typically HTML documents and may contain injection sinks, such as when they display the URL of the file that could not be found. As such, server-generated error responses can be used for the attack in the same way as regular pages.

Our URL mutation techniques differ in how they attempt to cause path confusion and inject style directives by covering different URL conventions used by a range of web application platforms.

```
example.com/page.asp
example.com/page.asp/PAYLOAD//
example.com/page.asp/PAYLOAD/style.css
```

(a) Path Parameter (Simple)

```
example.com/page.php/param1/param2
example.com/page.php/PAYLOADparam1/PAYLOADparam2//
example.com/page.php/PAYLOADparam1/PAYLOADparam2/style.css
```

(b) Path Parameter (PHP or ASP)

```
example.com/page.jsp;param1;param2
example.com/page.jsp;PAYLOADparam1;PAYLOADparam2//
example.com/page.jsp;PAYLOADparam1;PAYLOADparam2/style.css
```

(c) Path Parameter (JSP)

```
example.com/dir/page.aspx
example.com/PAYLOAD/..%2Fdir/PAYLOAD/..%2Fpage.aspx//
example.com/PAYLOAD/..%2Fdir/PAYLOAD/..%2Fpage.aspx/style.css
```

(d) Encoded Path

```
example.com/page.html?k1=v1&k2=v2
example.com/page.html%3Fk1=PAYLOADv1&k2=PAYLOADv2//
example.com/page.html%3Fk1=PAYLOADv1&k2=PAYLOADv2/style.css
```

(e) Encoded Query

```
example.com/page.php?key=value
example.com/page.php//?key=value
example.com/page.php/style.css

Original Cookie: k1=v1; k2=v2
 Crafted Cookie: k1=PAYLOADv1; k2=PAYLOADv2
```

(f) Cookie

Figure 4.2: Various techniques of **path confusion** and **style injection**. In each example, the first URL corresponds to the regular page, and the second one to the page URL crafted by the attacker. Each HTML page is assumed to reference a stylesheet at `../style.css`, resulting in the browser expanding the stylesheet path as shown in the third URL. **PAYLOAD** corresponds to `%0A{}body{background:NONCE}` (simplified), where `NONCE` is a randomly generated string.

Path Parameter. A number of web frameworks such as PHP, ASP, or JSP can be configured to use URL schemes that encode script input parameters as a directory-like string following the name of the script in the URL. Figure 4.2a shows a generic example where there is no parameter in the URL. Since the crawler does not know the name of valid parameters, it simply appends the payload as a subdirectory to the end of the URL. In this case, content injection can occur if the page reflects the page URL or referrer into the response. Note that in the example, we appended two slashes so that the browser does not remove the payload from the URL when expanding the stylesheet reference to the parent directory (`../style.css`). In the actual crawl, we always appended twenty slashes to avoid having to account for different numbers of parent directories. We did not observe relative paths using large numbers of `../` to reference stylesheets, thus we are confident that twenty slashes suffice for our purposes.

Different web frameworks handle path parameters slightly differently, which is why we distinguish a few additional cases. If parameters are present in the URL, we can distinguish these cases based on a number of regular expressions that we generated. For example, parameters can be separated by slashes (Figure 4.2b, PHP or ASP) or semicolons (Figure 4.2c, JSP). When the crawler detects one of these known schemes, it injects the payload into each parameter. Consequently, in addition to URL and referrer reflection, injection can also be successful when any of the parameters is reflected in the page.

Encoded Path. This technique targets web servers such as IIS that decode encoded slashes in the URL for directory traversal, whereas web browsers do not. Specifically, we use `%2F`, an encoded version of '`/`', to inject our payload into the URL in such a way that the canonicalized URL is equal to the original page URL (see Figure 4.2d). Injection using this technique

succeeds if the page reflects the page URL or referrer into its output.

Encoded Query. Similar to the technique above, we replace the URL query delimiter '?' with its encoded version %3F so that web browsers do not interpret it as such. In addition, we inject the payload into every value of the query string, as can be seen in Figure 4.2e. CSS injection happens if the page reflects either the URL, referrer, or any of the query values in the HTML response.

Cookie. Since stylesheets referenced by a relative path are located in the same origin as the referencing page, its cookies are sent when requesting the stylesheet. CSS injection may be possible if an attacker can create new cookies or tamper with existing ones (a strong assumption compared to the other techniques), and if the page reflects cookie values in the response. As shown in Figure 4.2f, the URL is only modified by adding slashes to cause path confusion. The payload is injected into each cookie value and sent by the crawler as an HTTP header.

4.2.3 Exploitability Analysis

Once a page has been found to be vulnerable to style injection using RPO, the final step is to verify whether the reflected CSS in the response is evaluated by a real browser. To do so, we built a crawler based on Google Chrome, and used the Remote Debugging Protocol [1] to drive the browser and record HTTP requests and responses. In addition, we developed a Chrome extension to populate the cookie header in CSS stylesheet requests with our payload.

In order to detect exploitable pages, we crawled all the pages from the previous section that had at least one reflection. Specifically, for each page we checked which of the techniques in Figure 4.2 led to reflection, and crafted the main URL with a CSS payload. The CSS payload used to verify exploitability

is different from the simple payload used to test reflection. Specifically, the style directive is prefixed with a long sequence of `}` and `]` characters to close any preceding open curly braces or brackets that may be located in the source code of the page, since they might prevent the injected style directive from being parsed correctly. The style directive uses a randomly-generated URL to load a background image for the HTML body. We determine whether the injected style is evaluated by checking the browser's network traffic for an outgoing HTTP request for the image.

Overriding Document Types. Reflected CSS is not always interpreted by the browser. One possible explanation is the use of a modern document type in the page, which does not cause the browser to render the page in quirks mode. Under certain circumstances, Internet Explorer allows a parent page to force the parsing mode of a framed page into quirks mode [66]. To test how often this approach succeeds in practice, we also crawled vulnerable pages with Internet Explorer 11 by framing them while setting `X-UA-Compatible` to `IE=EmulateIE7` via a `meta` tag in the attacker's page.

4.2.4 Limitations

RPO is a class of attacks and our methodology covers only a subset of them. We target RPO for the purpose of style injection using an HTML page referencing itself (or, accidentally, an error page) as the stylesheet. In terms of style injection, our crawler only looks for reflection, not stored injection of style directives. Furthermore, manual analysis of a site might reveal more opportunities for style injection that our crawler fails to detect automatically.

For efficiency reasons, we seed our analysis with an existing Common Crawl dataset. We do not analyze the vulnerability of pages not contained in the Common Crawl seed, which means that we do not cover all sites, and we do not fully cover all pages within a site. Consequently, the results presented

in this chapter should be seen as a lower bound. If desired, our methodology can be applied to individual sites in order to analyze more pages.

4.2.5 Ethical Considerations

One ethical concern is that the injected CSS might be stored on the server instead of being reflected in the response, and it could break sites as a result. We took several cautionary steps in order to minimize any damaging side effects on sites we probed. First, we did not try to login to the site, and we only tested RPO on the publicly available version of the page. In addition, we only requested pages by tainting different parts of the URL, and did not submit any forms. Moreover, we did not click on any button or link in the page in order to avoid triggering JavaScript events. These steps significantly decrease the chances that injected CSS will be stored on the server. In order to minimize the damaging side effects in case our injected CSS was stored, the injected CSS is not a valid style directive, and even if it is stored on the server, it will not have any observable effect on the page.

In addition, experiment resulted in the discovery of vulnerable content management systems (CMSes) used world-wide, and we contacted them so they can fix the issue. We believe the real-world experiments that we conducted were necessary in order to measure the risk posed by these vulnerabilities and inform site owners of potential risks to their users.

4.3 Analysis

For the purposes of our analysis, we gradually narrow down the seed data from the Common Crawl to pages using relative style paths in the Alexa Top 1 M, reflecting injected style directives under RPO, and being exploitable due to quirks mode rendering.

Table 4.2 shows a summary of our dataset. *Tested Pages* refers to the set

Table 4.2: Narrowing down the Common Crawl to the candidate set used in our analysis (from left to right).

	Relative CSS	**Alexa Top 1M**	**Candidate Set**
All Pages	203,609,675	141,384,967	136,793,450
Tested Pages	53,725,270	31,448,446	30,991,702
Sites	5,960,505	223,212	222,443
Doc. Types	9,833	2,965	2,898

of randomly selected pages from the page groups as discussed in Section 4.2.1. For brevity, we are referring to *Tested Pages* wherever we mention pages in the remainder of the chapter.

4.3.1 Relative Stylesheet Paths

To assess the extent to which our Common Crawl-seeded candidate set covers sites of different popularity, consider the hatched bars in Figure 4.3. Six out of the ten largest sites according to Alexa are represented in our candidate set. That is, they are contained in the Common Crawl, and have relative style paths. The figure shows that our candidate set contains a higher fraction of the largest sites and a lower fraction of the smaller sites. Consequently, our results better represent the most popular sites, which receive most visitors, and most potential victims of RPO attacks.

While all the pages in the candidate set contain at least one relative stylesheet path, Figure 4.4 shows that 63.1 % of them contain multiple relative paths, which increases the chances of finding a successful RPO and style injection point.

4.3.2 Vulnerable Pages

We consider a candidate page vulnerable if one of the style injection techniques of Section 4.2.2 succeeds. In other words, the server's response should

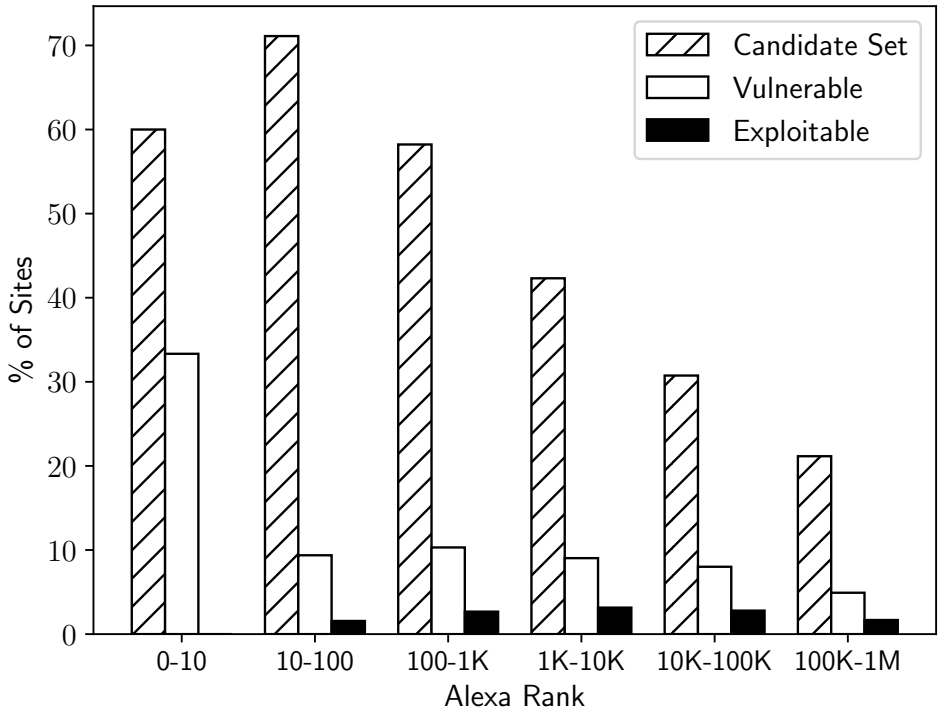

Figure 4.3: Percentage of the Alexa site ranking in our candidate set.

reflect the injected payload. Furthermore, we conservatively require that the response not contain a `base` tag since a correctly configured base tag can prevent path confusion.

Table 4.3 shows that 1.2 % of pages are vulnerable to at least one of the injection techniques, and 5.4 % of sites contain at least one vulnerable page. The path parameter technique is most effective against pages, followed by the encoded query and the encoded path techniques. Sites that are ranked higher according to Alexa are more likely to be vulnerable, as shown in Figure 4.3, where vulnerable and exploitable sites are relative to the candidate set in each bucket. While one third of the candidate set in the Top 10 (two out of six sites) is vulnerable, the percentage oscillates between 8 and 10 % among the Top 100 k. The candidate set is dominated by the smaller sites in the ranks between 100 k and 1 M, which have a vulnerability rate of 4.9 % and push down the average over the entire ranking.

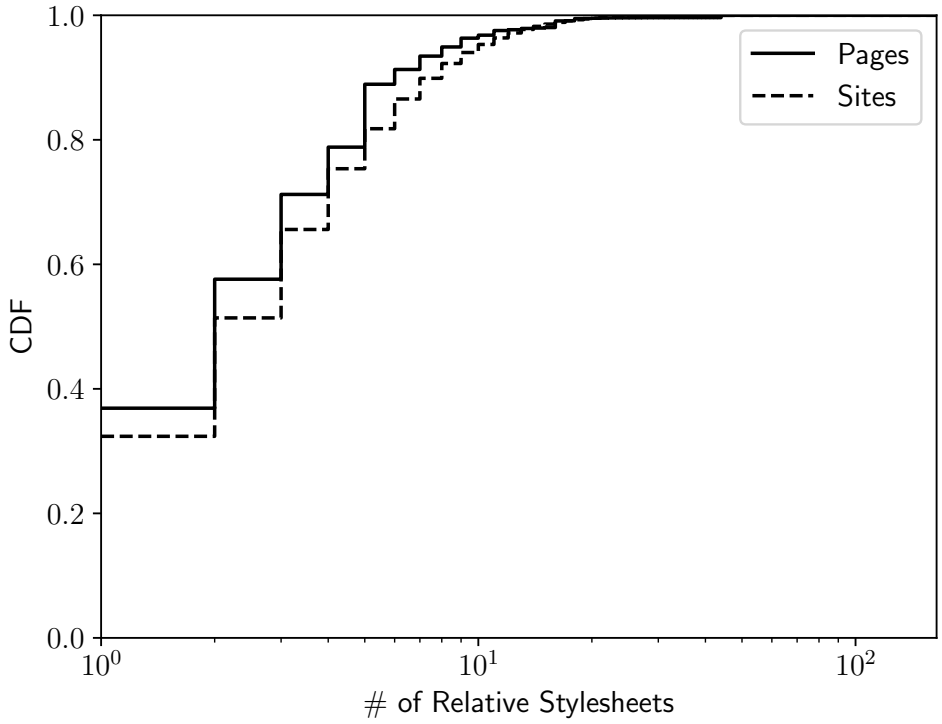

Figure 4.4: CDF of total and maximum number of relative stylesheets per web page and site, respectively.

A `base` tag in the server response can prevent path confusion because it indicates how the browser should expand relative paths. We observed a number of inconsistencies with respect to its use. At first, 603 pages on 60 sites contained a `base` tag in their response; however, the server response after injecting our payload did not contain the tag anymore, rendering these pages potentially exploitable. Furthermore, Internet Explorer's implementation of the `base` tag appears to be broken. When such a tag is present, Internet Explorer fetches two URLs for stylesheets—one expanded according to the base URL specified in the tag, and one expanded in the regular, potentially "confused" way of using the page URL as the base. In our experiments, Internet Explorer always applied the "confused" stylesheet, even when the one based on the `base` tag URL loaded faster. Consequently, `base` tags do not appear to be an effective defense against RPO in Internet Explorer (They

Table 4.3: Vulnerable pages and sites in the candidate set.

Technique	**Vulnerable Pages**	**Vulnerable Sites**
Path Parameter	309,079 (1.0%)	9,136 (4.1%)
Encoded Path	53,502 (0.2%)	1,802 (0.8%)
Encoded Query	89,757 (0.3%)	1,303 (0.6%)
Cookie	15,656 (<0.1%)	1,030 (0.5%)
Total	377,043 (1.2%)	11,986 (5.4%)

Table 4.4: Exploitable pages and sites in the candidate set (IE using framing).

Technique	**Exploitable (Chrome)**		**Exploitable (Internet Explorer)**	
	Pages	**Sites**	**Pages**	**Sites**
Path Parameter	6,048 (<0.1%)	1,025 (0.5%)	52,344 (0.2%)	3,433 (1.5%)
Encoded Path	3 (<0.1%)	2 (<0.1%)	24 (<0.1%)	5 (<0.1%)
Encoded Query	23 (<0.1%)	20 (<0.1%)	137 (<0.1%)	43 (<0.1%)
Cookie	4,722 (<0.1%)	81 (<0.1%)	2,447 (<0.1%)	238 (0.1%)
Total	10,781 (<0.1%)	1,106 (0.5%)	54,853 (0.2%)	3,645 (1.6%)

seem to work as expected in other browsers, including Edge).

4.3.3 Exploitable Pages

To test whether a vulnerable page was exploitable, we opened it in Chrome, injected a style payload with an image reference (randomly generated URL), and checked if the image was indeed loaded. As shown in Table 4.4, this test succeeded for 2.9 % of vulnerable pages; 0.5 % of sites in the candidate set had at least one exploitable page.

In Section 4.3.4 to 4.3.7, we explore various factors that may impact whether a vulnerable page can be exploited, and we show how some of these partial defenses can be bypassed.

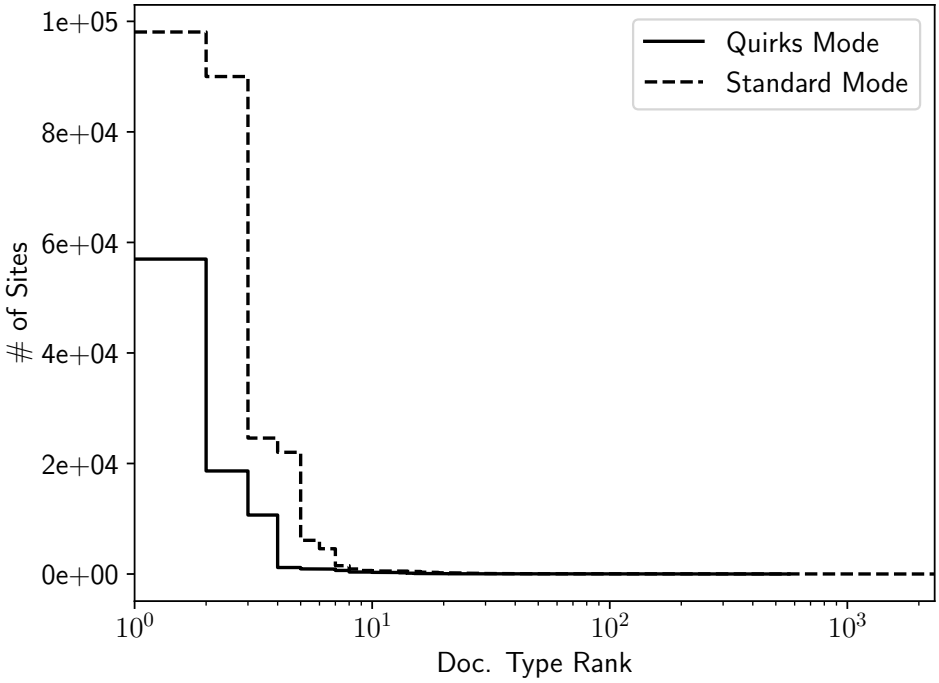

Figure 4.5: Number of sites containing at least one page with a certain document type (ordered by doctype rank).

4.3.4 Document Types

HTML document types play a significant role in RPO-based style injection attacks because browsers typically parse resources with a non-CSS content type in a CSS context only when the page specifies an ancient or non-standard HTML document type (or none at all). The pages in our candidate set contain a total of 4,318 distinct document types. However, the majority of these unique document types are not standardized and differ from the standardized ones only by small variations, such as forgotten spaces or misspellings.

To determine how browsers interpret these document types (i.e., whether they cause them to render a page in standards or quirks mode), we designed a controlled experiment. For each unique document type, we set up a local page with a relative stylesheet path and carried out an RPO attack to inject CSS using a payload similar to what we described in Section 4.2.2.

Table 4.5: Quirks mode document types by browser.

Browser	**Version**	**Operating System**	**Doc. Types**
Chrome	55	Ubuntu 16.04	1,378 (31.9 %)
Opera	42	Ubuntu 16.04	1,378 (31.9 %)
Safari	10	macOS Sierra	1,378 (31.9 %)
Firefox	50	Ubuntu 16.04	1,326 (30.7 %)
Edge	38	Windows 10	1,319 (30.5 %)
Internet Explorer	11	Windows 7	1,319 (30.5 %)

Table 4.6: Most frequent document types causing all browsers to render in quirks mode, as well as the sites that use at least one such document type.

Doc. Type (shortened)	**Pages**	**Sites**
(none)	1,818,595 (5.9 %)	56,985 (25.6 %)
"-//W3C//DTD HTML 4.01 Transitional//EN"	721,884 (2.3 %)	18,648 (8.4 %)
"-//W3C//DTD HTML 4.0 Transitional//EN"	385,656 (1.2 %)	11,566 (5.2 %)
"-//W3C//DTD HTML 3.2 Final//EN"	22,019 (<0.1 %)	1,175 (0.5 %)
"-//W3C//DTD HTML 3.2//EN"	10,839 (<0.1 %)	927 (0.4 %)
All	3,046,449 (9.6 %)	71,597 (32.2 %)

We automatically opened the local page in Chrome, Firefox, Edge, Internet Explorer, Safari, and Opera, and we kept track of which document type caused the injected CSS to be parsed and the injected background image to be downloaded.

Table 4.5 contains the results of this experiment. Even though the exact numbers vary among browsers, roughly a third of the unique document types we encountered result in quirks mode rendering. Not surprisingly, both Microsoft products Edge and Internet Explorer exhibit identical results, whereas the common Webkit ancestry of Chrome, Opera, and Safari also show identical results. Overall, 1,271 (29.4 %) of the unique document types force all the browsers into quirks mode, whereas 1,378 (31.9 %) of them cause at least one browser to use quirks mode rendering. Table 4.6 shows the most frequently used document types that force all the browsers into quirks mode,

Table 4.7: Summary of document type usage in sites.

Doc. Type	At Least One Crawled Page	All Crawled Pages
None	56,985 (25.6%)	19,968 (9.0%)
Quirks	27,794 (12.5%)	7,720 (3.5%)
None or Quirks	71,597 (32.2%)	30,040 (13.5%)
Standards	192,403 (86.5%)	150,846 (67.8%)

which includes the absence of a document type declaration in the page.

To test how often Internet Explorer allows a page's document type to be overridden when loading it in an `iframe`, we created another controlled experiment using a local attack page framing the victim page, as outlined in Section 4.2.3. Using Internet Explorer 11, we loaded our local attack page for each unique document type inside the frame, and tested if the injected CSS was parsed. While Internet Explorer parsed the injected CSS for 1,319 (30.5 %) of the document types in the default setting, the frame override trick caused CSS parsing for 4,248 (98.4 %) of the unique document types.

While over one thousand document types result in quirks mode, and around three thousand document types cause standards mode parsing, the number of document types that have been standardized is several orders of magnitude smaller. In fact, only a few (standardized) document types are used frequently in pages, whereas the majority of unique document types are used very rarely. Figure 4.5 shows that only about ten standards and quirks mode document types are widely used in pages and sites. Furthermore, only about 9.6 % of pages in the candidate set use a quirks mode document type; on the remaining pages, potential RPO style injection vulnerabilities cannot be exploited because the CSS would not be parsed (unless Internet Explorer is used). However, when grouping pages in the candidate set by site, 32.2 % of sites contain at least one page rendered in quirks mode (Table 4.7), which is one of the preconditions for successful RPO.

4.3.5 Internet Explorer Framing

We showed above that by loading a page in a frame, Internet Explorer can be forced to disregard a standards mode document type that would prevent interpretation of injected style. To find out how often this technique can be applied for successful RPO attacks, we replicated our Chrome experiment in Internet Explorer, this time loading each vulnerable page inside a frame. Around 14.5 % of vulnerable pages were exploitable in Internet Explorer, five times more than in Chrome (1.6 % of the sites in the candidate set).

Figure 4.3 shows the combined exploitability results for Chrome and Internet Explorer according to the rank of the site. While our methodology did not find any exploitable vulnerability on the six highest-ranked sites in the candidate set, between 1.6 % and 3.2 % of candidate sites in each remaining bucket were found to be exploitable. The highest exploitability rate occurred in the ranks 1 k through 10 k.

Broken down by injection technique, the framing trick in Internet Explorer results in more exploitable pages for each technique except for cookie injection (Table 4.4). One possible explanation for this difference is that the Internet Explorer crawl was conducted one month after the Chrome crawl, and sites may have changed in the meantime. Furthermore, we observed two additional impediments to successful exploitation in Internet Explorer that do not apply to Chrome. The framing technique is susceptible to frame-busting methods employed by the framed pages, and Internet Explorer implements an anti-MIME-sniffing header that Chrome appears to ignore. We analyze these issues below.

4.3.6 Anti-Framing Techniques

Some sites use a range of techniques to prevent other pages from loading them in a frame [111]. One of these techniques is the `X-Frame-Options` header. It

accepts three different values: `DENY`, `SAMEORIGIN`, and `ALLOW-FROM` followed by a whitelist of URLs. In the vulnerable dataset, 4,999 pages across 391 sites use this header correctly and as a result prevent the attack. However, 1,900 pages across 34 sites provide incorrect values for this header, and we successfully attack 552 pages on 2 sites with Internet Explorer.

A related technique is the `frame-ancestors` directive provided by Content Security Policy. It defines a (potentially empty) whitelist of URLs allowed to load the current page in a frame, similar to `ALLOW-FROM`. However, it is not supported by Internet Explorer, thus it cannot be used to prevent the attack. Out of all vulnerable pages, 52 pages across 3 sites used `frame-ancestors`. Although we expected these pages to be exploitable in Internet Explorer, none of them would let an attack to proceed successfully as they used `X-Frame-Options` header as well.

Furthermore, developers may use JavaScript code to prevent framing of a page. Yet, techniques exist to bypass this protection [104]. In addition, the attacker can use the HTML 5 `sandbox` attribute in the `iframe` tag and omit the `allow-top-navigation` directive to render JavaScript frame-busting code ineffective. However, we did not implement any of these techniques to allow framing, which means that more vulnerable pages could likely be exploited in practice.

4.3.7 MIME Sniffing

A consequence of self-reference in the type of RPO studied in this section is that the HTTP content type of the fake "stylesheet" is `text/html` rather than the expected `text/css`. Because many sites contain misconfigured content types, many browsers attempt to infer the type based on the request context or file extension (*MIME sniffing*), especially in quirks mode. In order to ask the browser to disable content sniffing and refuse interpreting data with an unexpected or wrong type, sites can set the header

`X-Content-Type-Options: nosniff` [12, 65, 90].

To determine whether the injected CSS is still being parsed and executed in presence of this header while the browser renders in quirks mode, we ran an experiment similar to Section 4.3.4. For each browser in Table 4.5, we extracted the document types in which the browser renders in quirks mode, and for each of them, we set up a local page with a relative stylesheet path. We then opened the page in the browser, launched an RPO attack, and monitored if the injected CSS was executed.

Only Firefox, Internet Explorer, and Edge respected this header and did not interpret injected CSS in any of the quirks mode document types. The remaining browsers did not block the stylesheet even though the content type was not `text/css`. With an additional experiment, we confirmed that Internet Explorer blocked our injected CSS payload when `nosniff` was set, even in the case of the framing technique.

Out of all the vulnerable pages, 96,618 pages across 232 sites had a `nosniff` response header; 23 pages across 10 sites were confirmed exploitable in Chrome but not in Internet Explorer, since the latter browser respects the header while the former does not.

4.3.8 Content Management Systems

While analyzing the exploitable pages in our dataset, we noticed that many appeared to belong to well-known CMSes. Since these web applications are typically installed on thousands of sites, fixing RPO weaknesses in these applications could have a large impact.

To identify CMSes, we visited all exploitable pages using Wappalyzer [138]. Additionally, we detected two CMSes that were not supported by Wappalyzer. Overall, we identified 23 CMSes on 41,288 pages across 1,589 sites. Afterwards, we manually investigated whether the RPO weakness stemmed from the CMS by installing the latest version of each CMS (or using the on-

line demo), and testing whether exploitable paths found in our dataset were also exploitable in the CMS. After careful analysis, we confirmed four CMSes to be exploitable in their most recent version that are being used by 40,255 pages across 1,197 sites.

Out of the four exploitable CMSes, one declares no document type and one uses a quirks mode document type. These two CMSes can be exploited in Chrome, whereas the remaining two can be exploited with the framing trick in Internet Explorer. Beyond the view of our Common Crawl candidate set, Wappalyzer detected nearly 32 k installations of these CMSes across the Internet, which suggests that many more sites could be attacked with RPO. We reported the RPO weaknesses to the vendors of these CMSes using recommended notification techniques [81, 121, 21]. Thus far, we heard back from one of the vendors, who acknowledged the vulnerability and are going to take the necessary steps to fix the issue. However, we have not received any response from the other vendors.

4.4 Countermeasures

In this section we discuss potential defense techniques against RPO. The observed mitigations, although not intentionally designed to prevent RPO, but were successful to alleviate the exploitation probability. We, also, discuss some defense techniques which could be successfully employed to safeguard the system against RPO. It worths mentioning that since RPO is a complicated attack which relies on multiple prerequisites to be successful, its countermeasures could be simpler compared to XSS or other common attack techniques. Therefore, obstructing one of the preconditions, which include using relative path stylesheet, reflect inserted directives in the input, and rendering the response in quirk mode, would suffice to barricade the attack.

4.4.1 Observed Mitigations

Relative Path Overwrites rely on the web server and the web browser interpreting URLs differently. Based on our experiments, using absolute path and including the `<base>` tag are observed to be preventive against RPO. Even though pre-mentioned techniques were not intended to prevent RPO, those pages unintentionally afforded protection against it.

HTML pages can use only absolute (or root-relative) URLs, which removes the need for the web browser to expand relative paths. Alternatively, when the HTML page contains a `<base>` tag, browsers are expected to use the URL provided therein to expand relative paths instead of interpreting the current document's URL. Both methods can remove ambiguities and render RPO impossible if applied correctly. Specifically, base URLs must be set according to the server's content routing logic. It should be mentioned that if developers choose to calculate base URLs dynamically on the server side rather than setting them manually to constant values, there is a risk that routing-agnostic algorithms could be confused by manipulated URLs and re-introduce attack opportunities by instructing browsers to use an attacker-controlled base URL. Furthermore, Internet Explorer does not appear to implement this tag correctly.

Instead of preventing RPO and style injection vulnerabilities, the most promising approach could be to avoid exploitation. In fact, declaring a modern document type that causes the HTML document to be rendered in standards mode makes the attack fail in all browsers except for Internet Explorer. Web developers can harden their pages against the frame-override technique in Internet Explorer by using commonly recommended HTTP headers: `X-Content-Type-Options` to disable "content type sniffing" and always use the MIME type sent by the server (which must be configured correctly), `X-Frame-Options` to disallow loading the page in a frame, and

`X-UA-Compatible` to turn off Internet Explorer's compatibility view.

4.4.2 Suggested Defense Techniques

Web developers can reduce the attack surface of their sites by eliminating any injection sinks for strings that could be interpreted as a style directive. However, doing so is challenging because in the attack presented in this chapter, style injection does not require a specific sink type and does not need the ability of injecting markup. Injection can be accomplished with relatively commonly used characters, that is, alphanumeric characters and `(){}/"`.

Experience has shown that despite years of efforts, even context-sensitive and more special character-intensive XSS injection is still possible in many sites, which leads us to believe that style injection will be similarly difficult to eradicate. Even when all special characters in user input are replaced by their corresponding HTML entities and direct style injection is not possible, more targeted RPO attack variants referencing existing files may still be feasible. For instance, it has been shown that user uploads of seemingly benign profile pictures can be used as "scripts" (or stylesheets) [125].

4.5 Chapter Summary

This chapter presented a systematic study of CSS injection by RPO in the wild. We showed that over 5 % of sites in the intersection of the Common Crawl and the Alexa Top 1M are vulnerable to at least one injection technique. While the number of exploitable sites depends on the browser and is much smaller in relative terms, it is still consequential in absolute terms with thousands of affected sites. RPO is a class of attacks, and our automated crawler tested for only a subset of conceivable attacks. Therefore, the results of our study should be seen as a lower bound; the true number of exploitable

sites is likely higher.

Compared to XSS, it is much more challenging to avoid injection of style directives. Yet, developers have at their disposal a range of simple mitigation techniques that can prevent their sites from being exploited in modern browsers.

Chapter 5

Web Cache Deception

5.1 Introduction

Web caches (also called HTTP caches or web accelerators) have become an essential component of the Internet infrastructure with numerous use cases such as reducing bandwidth costs in private enterprise networks and accelerating content delivery over the World Wide Web. Today caching is implemented at multiple stages of Internet communications, for instance in popular web browsers [93, 126], at caching proxies [133, 117], and directly at origin web servers [9, 97].

In particular, Content Delivery Network (CDN) providers heavily rely on effective web content caching at their edge servers, which together comprise a massively-distributed Internet overlay network of caching reverse proxies. Popular CDN providers advertise accelerated content delivery and high availability via global coverage and deployments reaching hundreds of thousands of servers [29, 7]. A recent scientific measurement also estimates that more than 74% of the Alexa Top 1K are served by CDN providers, indicating that CDNs and more generally web caching play a central role in the Internet [47].

While there exist technologies that enable limited caching of dynamically-generated pages, web caching primarily targets static, publicly accessible content. In other words, web caches store static content that is costly to de-

liver due to an object's size or distance. Importantly, these objects *must not* contain private or otherwise sensitive information, as application-level access control is not enforced at cache servers. Good candidates for caching include frequently accessed images, software and document downloads, streaming media, style sheets, and large static HTML and JavaScript files.

In 2017, Gil presented a novel attack called *web cache deception (WCD)* that can trick a web cache into incorrectly storing sensitive content, and consequently give an attacker unauthorized access to that content [44, 45]. Gil demonstrated the issue with a real-life attack scenario targeting a high profile site, PayPal, and showed that WCD can successfully leak details of a private payment account. Consequently, WCD garnered significant media attention, and prompted responses from major web cache and CDN providers [84, 25, 17, 15, 99, 26].

At its core, WCD results from *path confusion* between an origin server and a web cache. In other words, different interpretations of a requested URL at these two points lead to a disagreement on the cacheability of a given object. This disagreement can then be exploited to trick the web cache into storing non-cacheable objects. WCD does not imply that these individual components—the origin server and web cache—are incorrectly configured per se. Instead, their hazardous interactions as a system lead to the vulnerability. As a result, detecting and correcting vulnerable systems is a cumbersome task, and may require careful inspection of the entire caching architecture. Combined with the aforementioned pervasiveness and critical role of web caches in the Internet infrastructure, WCD has become a severely damaging issue.

In this chapter, we first present a large-scale measurement and analysis of WCD over 295 sites in the Alexa Top 5K. We present a repeatable and automated methodology to discover vulnerable sites over the Internet, and a detailed analysis of our findings to characterize the extent of the prob-

lem. Our results show that many high-profile sites that handle sensitive and private data are impacted by WCD and are vulnerable to practical attacks. We then discuss additional path confusion methods that can maximize the damage potential of WCD, and demonstrate their impact in a follow-up experiment over an extended data set of 340 sites.

To the best of our knowledge, this is the first in-depth investigation of WCD in a scientific framework and at this scale. In addition, the scope of our investigation goes beyond private data leakage to provide novel insights into the severity of WCD. We demonstrate how WCD can be exploited to steal other types of sensitive data including security tokens, explain advanced attack techniques that elevate WCD vulnerabilities to injection vectors, and quantify our findings through further analysis of collected data.

Finally, we perform an empirical analysis of popular CDN providers, documenting their default caching settings and customization mechanisms. Our findings underline the fact that WCD is a *system safety* problem. Site operators must adopt a holistic view of their infrastructure, and carefully configure web caches taking into consideration their complex interactions with origin servers.

To summarize, we make the following contributions:

- We propose a novel, repeatable methodology to detect sites impacted by WCD at scale. Unlike existing WCD scan tools that are designed for site administrators to test their own properties in a controlled environment, our methodology is designed to automatically detect WCD in the wild.

- We present findings that quantify the prevalence of WCD in 295 sites among the Alexa Top 5K, and provide a detailed breakdown of leaked information types. Our analysis also covers security tokens that can be stolen via WCD as well as novel security implications of the attack, all areas left unexplored by existing WCD literature.

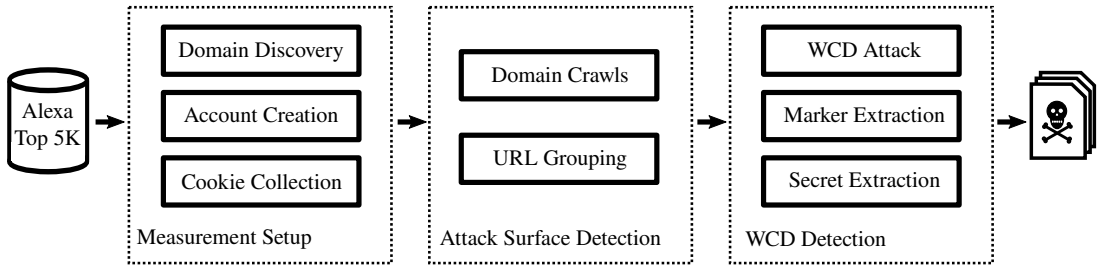

Figure 5.1: A high-level overview of our WCD measurement methodology.

- We conduct a follow-up measurement over 340 sites among the Alexa Top 5K that show variations on the path confusion technique make it possible to successfully exploit sites that are not impacted by the original WCD attack.

- We analyze the default settings of popular CDN providers and document their distinct caching behavior, highlighting that mitigating WCD necessitates a comprehensive examination of a website's infrastructure.

Ethical Considerations. We have designed our measurement methodology to minimize the impact on scanned sites, and limit the inconvenience we impose on site operators. Similarly, we have followed responsible disclosure principles to notify the impacted parties, and limited the information we share in this paper to minimize the risk of any inadvertent damage to them or their end-users. We discuss details of the ethical considerations pertaining to this work in Sec. 5.2.5.

5.2 Methodology

We present our measurement methodology in three stages: *(1)* measurement setup, *(2)* attack surface detection, and *(3)* WCD detection. We illustrate this process in Fig. 5.1. We implemented the tools that perform the described tasks using a combination of Google Chrome and Python's Requests

library [109] for web interactions, and Selenium [113] and Google Remote Debugging Protocol [46] for automation.

5.2.1 Stage 1: Measurement Setup

WCD attacks are only meaningful when a vulnerable site manages private end-user information and allows performing sensitive operations on this data. Consequently, sites that provide authentication mechanisms are prime targets for attacks, and thus also for our measurements. The first stage of our methodology identifies such sites and creates test accounts on them.[1]

Domain Discovery. This stage begins by visiting the sites in an initial measurement *seed pool* (e.g., the Alexa Top n domains). We then increase site coverage by performing sub-domain discovery using open-source intelligence tools [102, 2, 51]. We add these newly-discovered sub-domains of the primary sites (filtered for those that respond to HTTP(s) requests) to the seed pool.

Account Creation. Next, we create two test accounts on each site: one for a *victim*, and the other for an *attacker*. We populate each account with unique dummy values. Next, we manually explore each victim account to discover data fields that should be considered private information (e.g., name, email, address, payment account details, security questions and responses) or user-created content (e.g., comments, posts, internal messages). We populate these fields with predefined *markers* that can later be searched for in cached responses to detect a successful WCD attack. On the other hand, no data entry is necessary for attacker accounts.

[1] In the first measurement study we present in Sec. 5.3, we scoped our investigation to sites that support Google OAuth [107] for authentication due to its widespread use. This was a design choice made to automate a significant chunk of the initial account setup workload, a necessity for a large-scale experiment. In our follow-up experiment later described in Sec. 5.4 we supplemented this data set with an additional 45 sites that do not use Google OAuth. We discuss these considerations in their corresponding sections.

Table 5.1: Sample URL grouping for attack surface discovery.

Group By	URL
Query Parameter	http://example.com/?lang=**en** http://example.com/?lang=**fr**
Path Parameter	http://example.com/**028** http://example.com/**142**

Cookie Collection. Once successfully logged into the sites in our seed pool, crawlers collect two sets of cookies for all victim and attacker accounts. These are saved in a cookie jar to be reused in subsequent steps of the measurement. Note that we have numerous measures to ensure our crawlers remain authenticated during our experiments. Our crawlers periodically re-authenticate, taking into account cookie expiration timestamps. In addition, the crawlers use regular expressions and blacklists to avoid common logout links on visited pages.

5.2.2 Stage 2: Attack Surface Detection

Domain Crawls. In the second stage, our goal is to map from domains in the seed pool to a set of pages (i.e., complete URLs) that will later be tested for WCD vulnerabilities. To this end, we run a recursive crawler on each domain in the seed pool to record links to pages on that site.

URL Grouping. Many modern web applications customize pages based on query string or URL path parameters. These pages have similar structures and are likely to expose similar attack surfaces. Ideally, we would group them together and select only one random instance as a representative URL to test for WCD in subsequent steps.

Since performing a detailed content analysis is a costly process that could generate an unreasonable amount of load on the crawled site, our URL group-

ing strategy instead focuses on the structure of URLs, and approximates page similarity without downloading each page for analysis. Specifically, we convert the discovered URLs into an abstract representation by grouping those URLs by query string parameter names or by numerical path parameters. We select one random instance and filter out the rest. Table 5.1 illustrates this process.

This filtering of URLs significantly accelerates the measurements, and also avoids overconsumption of the target site's resources with redundant scans in Stage 3. We stop attack surface detection crawls after collecting 500 unique pages per domain for similar reasons.

5.2.3 Stage 3: WCD Detection

In this final stage, we launch a WCD attack against every URL discovered in Stage 2, and analyze the response to determine whether a WCD vulnerability was successfully exploited.

WCD Attack. The attack we mount directly follows the scenario previously described in Sec. 3.3 and illustrated in Fig. 3.3. For each URL:

1. We craft an attack URL that references a non-existent static resource. In particular, we append to the original page "`/<random>.css`"[2]. We use a random string as the file name in order to prevent ordinary end-users of the site from coincidentally requesting the same resource.

2. We initiate a request to this attack URL from the *victim* account and record the response.

[2]Our choice to use a style sheet in our payload is motivated by the fact that style sheets are essential components of most modern sites, and also prime choices for caching. They are also a robust choice for our tests. For instance, many CDN providers offer solutions to dynamically resize image files on the CDN edge depending on the viewport of a requesting client device. Style sheets are unlikely to be manipulated in such ways.

3. We issue the same request from the *attacker* account, and save the response for comparison.

4. Finally, we repeat the attack as an *unauthenticated user* by omitting any session identifiers saved in the attacker cookie jar. We later analyze the response to this step to ascertain whether attackers without authentication credentials (e.g., when the site does not offer open or free sign ups) can also exploit WCD vulnerabilities.

Marker Extraction. Once the attack scenario described above is executed, we first check for private information disclosure by searching the attacker response for the *markers* that were entered into victim accounts in Stage 1. If victim markers are present in URLs requested by an attacker account, the attacker must have received the victim's incorrectly cached content and, therefore, the target URL contains an exploitable WCD vulnerability. Because these markers carry relatively high entropy, it is probabilistically highly unlikely that this methodology will produce false positives.

Secret Extraction. We scan the attacker response for the disclosure of secret tokens frequently used as part of web application security mechanisms. These checks include common secrets (e.g., CSRF tokens, session identifiers) as well as any other application-specific authentication and authorization tokens (e.g., API credentials). We also check for session-dependent resources such as dynamically-generated JavaScript, which may have private information and secrets embedded in them (e.g., as explored by Lekies et al. [77]).

In order to extract candidates for leaked secrets, we scan attacker responses for name & value pairs, where either *(1)* the name contains one of our keywords (e.g., `csrf`, `xsrf`, `token`, `state`, `client_id`), or *(2)* the value has a random component. We check for these name & value pairs in hidden HTML form elements, query strings extracted from HTML anchor elements,

and inline JavaScript variables and constants. Similarly, we extract random file names referenced in HTML script elements. We perform all tests for randomness by first removing dictionary words from the target string (i.e., using a list of 10,000 common English words [63]), and then computing Shannon entropy over the remaining part.

Note that unlike our checks for private information leaks, this process can result in false positives. Therefore, we perform this secret extraction process only when the victim and attacker responses are identical (a strong indicator of caching), or otherwise when we can readily confirm a WCD vulnerability by searching for the private information markers. In addition, we later manually verify all candidate secrets extracted in this step.

5.2.4 Verification and Limitations

Researchers have repeatedly reported that large-scale Internet measurements, especially those that use automated crawlers, are prone to being blocked or served fake content by security solutions designed to block malicious bots and content scrapers [101, 137]. In order to minimize this risk during our measurement, we used a real browser (i.e., Google Chrome) for most steps in our methodology. For other interactions, we set a valid Chrome user-agent string. We avoided generating excessive amounts of traffic and limited our crawls as described above in order to avoid triggering rate-limiting alerts, in addition to ethical motivations. After performing our measurements, we manually verified *all* positive findings and confirmed the discovered vulnerabilities.

Note that this measurement has several important limitations, and the findings should be considered a potentially loose lower bound on the incidence of WCD vulnerabilities in the wild. For example, as described in Sec. 5.3, our seed pool is biased toward sites that support Google OAuth, which was a necessary compromise to automate our methodology and render a large-scale measurement feasible. Even under this constraint, creating accounts on some

sites required entering and verifying sensitive information such as credit card or US social security numbers which led to their exclusion from our study.

Furthermore, decisions such as grouping URLs based on their structure without analyzing page content, and limiting site crawls to 500 pages may have caused us to miss additional instances of vulnerabilities. Similarly, even though we manually filtered out false positives during our secret token extraction process and verified all findings, we do not have a scalable way of detecting false *negatives*. We believe that these trade-offs were worthwhile given the overall security benefits of and lessons learned from our work. We emphasize that the results in this measurement represent a lower bound.

5.2.5 Ethical Considerations

Here, we explain in detail important ethical considerations pertaining to this work and the results we present.

Performance Considerations. We designed our methodology to minimize the performance impact on scanned sites and inconvenience imposed on their operators. We did not perform repeated or excessive automated scans of the targeted sites, and ensured that our measurements did not generate unreasonable amounts of traffic. We used only passive techniques for sub-domain enumeration and avoided abusing external resources or the target site's DNS infrastructure.

Similarly, our stored modifications to crawled web applications only involved creating two test accounts and filling out editable fields with markers that we later used for data leakage detection. We believe this will have no material impact on site operators, especially in the presence of common threats such as malicious bots and credential stuffing tools that generate far more excessive junk traffic and data.

Security Considerations. Our methodology entirely avoids jeopardizing the security of crawled sites or their end-users. In this work, we never injected or stored any malicious payload to target sites, to web caches on the communication path, or otherwise maliciously tampered with any technology involved in the process. Likewise, the experiments we performed all incorporated randomized strings as the non-existent parts of URLs, thereby preventing unsuspecting end-users from accidentally accessing our cached data and receiving unexpected responses.

Note that this path randomization measure was used to prevent inconveniencing or confusing end-users; since we never exploited WCD to leak real personal data from a web application or stored a malicious payload, our work never posed a security risk to end-users.

Our experiments did not take into account robots.txt files. This was a risk-based decision we consciously made, and we believe that ignoring exclusion directives had no negative impact on the privacy of these sites' visitors. Robots.txt is not a security or privacy mechanism, but is intended to signal to data aggregators and search engines what content to index – including a directive to exclude privacy sensitive pages would actually be a misuse of this technology. This is not relevant to our experiments, as we only collect content for our analysis, and we do not index or otherwise publicly present site content.

Responsible Disclosure. In this chapter, we present a detailed breakdown of our measurement findings and results of our analysis, but we refrain from explicitly naming the impacted sites. Even though our methodology only utilized harmless techniques for WCD detection, the findings point at real-world vulnerabilities that could be severely damaging if publicly disclosed before remediation.

We sent notification emails to publicly listed security contacts of all im-

pacted parties promptly after our discovery. In the notification letters we provided an explanation of the vulnerability with links to online resources and listed the vulnerable domain names under ownership of the contacted party. We informed them of our intention to publicly publish these results, noted that they will not be named, and advised that they remediate the issue as adversaries can easily repeat our experiment and compromise their sites. We also explicitly stated that we did not seek or accept bug bounties for these notifications.

We sent the notification letters prior to submitting this work for review, therefore giving the impacted parties reasonably early notice. As of this writing, 12 of the impacted sites have implemented mitigations.

Repeatability. One of the co-researchers of this measurement is affiliated with a major CDN provider at the time of writing. However, the measurements and results we present in this chapter do not use any internal or proprietary company information, or any such information pertaining to the company's customers. We conducted this work using only publicly available data sources and tools. Our methodology is repeatable by other researchers without access to any CDN provider internals.

5.3 Measurement Study

We conducted two measurement studies to characterize web cache deception (WCD) vulnerabilities on the Internet. In this first study we present in this section, the research questions we specifically aim to answer are:

(Q1) What is the prevalence of WCD vulnerabilities on popular, highly-trafficked domains? (§5.3.2)

(Q2) Do WCD vulnerabilities expose PII and, if so, what kinds? (§5.3.3)

Table 5.2: Summary of crawling statistics.

	Crawled	**Vulnerable**
Pages	1,470,410	17,293 (1.2%)
Domains	124,596	93 (0.1%)
Sites	295	16 (5.4%)

(Q3) Can WCD vulnerabilities be used to defeat defenses against web application attacks? (§5.3.3)

(Q4) Can WCD vulnerabilities be exploited by unauthenticated users? (§5.3.3)

In the following, we describe the data we collected to carry out the study. We discuss the results of the measurement, and then consider implications for PII and important web security defenses. Finally, we summarize the conclusions we draw from the study. In Sec. 5.4, we will present a follow-up experiment focusing on advanced path confusion techniques.

5.3.1 Data Collection

We developed a custom web crawler to collect the data used in this measurement. The crawler ran from April 20-27, 2018 as a Kubernetes pod that was allocated 16 Intel Xeon 2.4 GHz CPUs and 32 GiB of RAM. Following the methodology described in Sec. 5.2, we configured the crawler to identify vulnerable sites from the Alexa Top 5K at the time of the experiment. In order to scalably create test accounts, we filtered this initial measurement seed pool for sites that provide an option for user authentication via Google OAuth. This filtering procedure narrowed the set of sites considered in this measurement to 295. Table 5.2 shows a summary of our crawling statistics.

5.3.2 Measurement Overview

Alexa Ranking. From the 295 sites comprising the collected data set, the crawler identified 16 sites (5.4%) to contain WCD vulnerabilities. Fig. 5.2

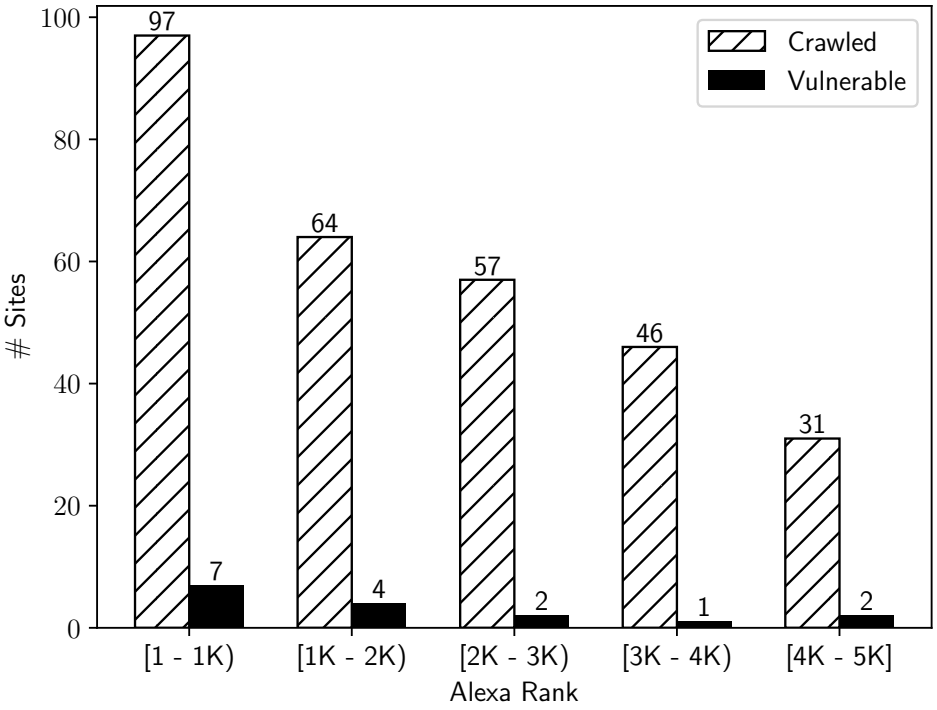

Figure 5.2: Distribution of the measurement data and vulnerable sites across the Alexa Top 5K.

presents the distribution of all sites and vulnerable sites across the Alexa Top 5K. From this, we observe that the distribution of vulnerable sites is roughly proportional to the number of sites crawled; that is, our data does not suggest that the incidence of WCD vulnerabilities is correlated with site popularity.

Category. Figure 5.3 shows the categories extracted from McAfee SmartFilter [89]. Overall, 13 out of 49 categories had at least one vulnerable site in our dataset. It is noteworthy that we found 16 vulnerable sites (as seen in Table 5.2) while it seems to have totally 20 vulnerable ones as per statistics illustrated in Figure 5.3. This difference is because of classifying some of the sites in more than one category at the same time based on McAfee SmartFilter.

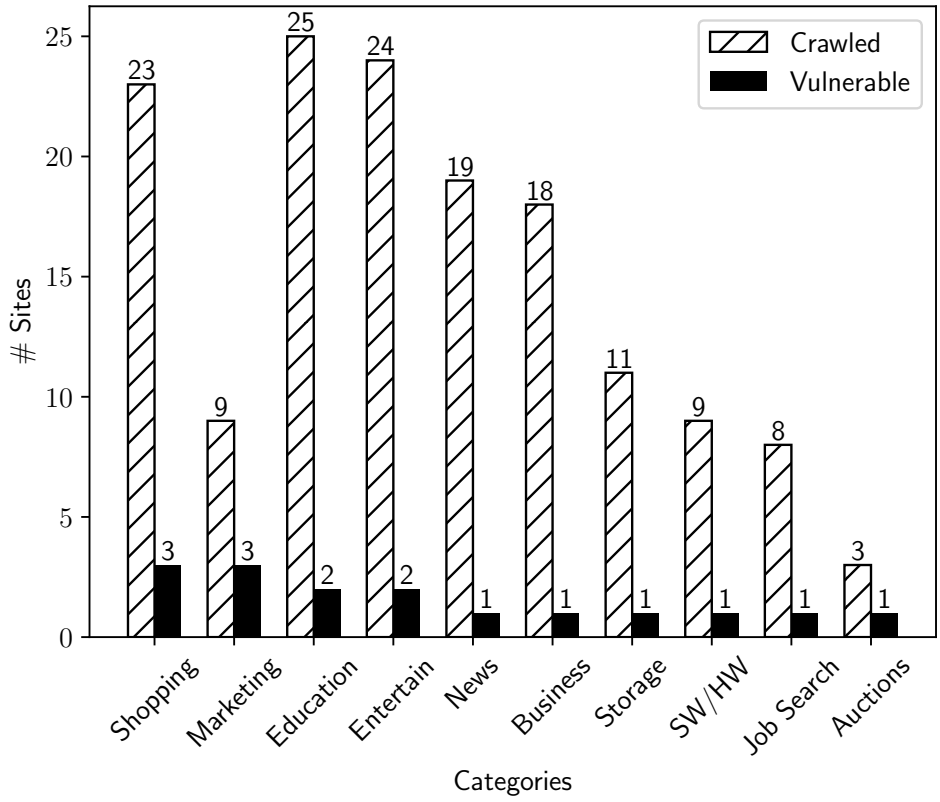

Figure 5.3: Categories

Shopping and Marketing categories have the highest rank of vulnerable sites. While Marketing observed to be in the second rank in total, but we found 3 out of 11 (27.2%) of this category to be vulnerable which is the highest among all other categories. Shopping had 3 out of 16 vulnerable sites (18.75%), however, 3 out of 24 (12.5%) crawled sites in the category was detected to be vulnerable. It is worth to notice that Business and News categories have the least portion of vulnerable sites with regard to their number of sites.

Content Delivery Networks (CDNs). Using a set of heuristics that searches for well-known vendor strings in HTTP headers, we labeled each domain and site with the corresponding CDN. Table 5.3 shows the results of this labeling.

Table 5.3: Pages, domains, and sites labeled by CDN using HTTP header heuristics. These heuristics simply check for unique vendor-specific strings added by CDN proxy servers.

CDN	**Crawled**			**Vulnerable**		
	Pages	**Domains**	**Sites**	**Pages**	**Domains**	**Sites**
Cloudflare	161,140 (11.0%)	4,996 (4.0%)	143 (48.4%)	16,234 (93.9%)	72 (77.4%)	8 (50.0%)
Akamai	225,028 (15.3%)	16,473 (13.2%)	100 (33.9%)	1,059 (6.1%)	21 (22.6%)	8 (50.0%)
CloudFront	100,009 (6.8%)	10,107 (8.1%)	107 (36.3%)	2 (<0.1%)	1 (1.1%)	1 (6.2%)
Other CDNs	244,081 (16.6%)	2,456 (2.0%)	137 (46.4%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
Total CDN Use	707,210 (48.1%)	33,675 (27.0%)	244 (82.7%)	17,293 (100.0%)	93 (100.0%)	16 (100.0%)

Table 5.4: Response codes observed in the vulnerable data set.

Response Code	**Pages**	**Domains**	**Sites**
404 Not Found	17,093 (98.8%)	82 (88.2%)	10 (62.5%)
200 Ok	205 (1.2%)	19 (20.4%)	12 (75.0%)

Note that many sites use multiple CDN solutions, and therefore the sum of values in the first four rows may exceed the totals we report in the last row.

The results show that, even though WCD attacks are equally applicable to any web cache technology, all instances of vulnerable pages we observed are served over a CDN. That being said, vulnerabilities are not unique to any one CDN vendor. While this may seem to suggest that CDN use is correlated with an increased risk of WCD, we point out that 82.7% of sites in our experiment are served over a CDN. A more balanced study focusing on comparing CDNs to centralized web caches is necessary to eliminate this inherent bias in our experiment and draw meaningful conclusions. Overall, these results indicate that CDN deployments are prevalent among popular sites, and the resulting widespread use of web caches may in turn lead to more opportunities for WCD attacks.

Table 5.5: Cache headers present in HTTP responses collected from vulnerable sites.

Header	**Pages**	**Domains**	**Sites**
Expires:	1,642 (9.5%)	23 (24.7%)	13 (81.2%)
Pragma: no-cache	652 (3.8%)	11 (11.8%)	6 (37.5%)
Cache-Control:	1,698 (9.8%)	26 (28.0%)	14 (87.5%)
max-age=, public	1,093 (6.3%)	10 (10.8%)	7 (43.8%)
max-age=	307 (1.8%)	1 (1.1%)	1 (6.2%)
must-revalidate, private	102 (0.6%)	1 (1.1%)	1 (6.2%)
max-age=, no-cache, no-store	67 (0.4%)	3 (3.2%)	2 (12.5%)
max-age=, no-cache	64 (0.4%)	4 (4.3%)	1 (6.2%)
max-age=, must-revalidate	51 (0.3%)	1 (1.1%)	1 (6.2%)
max-age=, must-revalidate, no-transform, private	5 (<0.1%)	3 (3.2%)	1 (6.2%)
no-cache	5 (<0.1%)	2 (2.2%)	1 (6.2%)
max-age=, private	3 (<0.1%)	1 (1.1%)	1 (6.2%)
must-revalidate, no-cache, no-store, post-check=, pre-check=	1 (<0.1%)	1 (1.1%)	1 (6.2%)
All	1,698 (9.8%)	26 (28.0%)	14 (87.5%)
(none)	15,595 (90.2%)	67 (72.0%)	3 (18.8%)

Response Codes. Table 5.4 presents the distribution of HTTP response codes observed for the vulnerable sites. This distribution is dominated by `404 Not Found` which, while perhaps unintuitive, is indeed allowed behavior according to RFC 7234 [40]. On the other hand, while only 12 sites leaked resources with a `200 OK` response, during our manual examination of these vulnerabilities (discussed below) we noted that more PII was leaked from this category of resource.

Cache Headers. Table 5.5 shows a breakdown of cache-relevant headers collected from vulnerable sites. In particular, we note that despite the presence of headers whose semantics prohibit caching—e.g., "`Pragma: no-cache`", "`Cache-Control: no-store`"—pages carrying these headers are cached regardless, as they were found to be vulnerable to WCD. This finding provides evidence that site administrators indeed take advantage of the configuration

Table 5.6: Types of vulnerabilities discovered in the data.

Leakage	**Pages**	**Domains**	**Sites**
PII	17,215 (99.5%)	88 (94.6%)	14 (87.5%)
User	934 (5.4%)	17 (18.3%)	8 (50.0%)
Name	16,281 (94.1%)	71 (76.3%)	7 (43.8%)
Email	557 (3.2%)	10 (10.8%)	6 (37.5%)
Phone	102 (0.6%)	1 (1.1%)	1 (6.2%)
CSRF	130 (0.8%)	10 (10.8%)	6 (37.5%)
JS	59 (0.3%)	5 (5.4%)	4 (25.0%)
POST	72 (0.4%)	5 (5.4%)	3 (18.8%)
GET	8 (<0.1%)	4 (4.3%)	2 (12.5%)
Sess. ID / Auth. Code	1,461 (8.4%)	11 (11.8%)	6 (37.5%)
JS	1,461 (8.4%)	11 (11.8%)	6 (37.5%)
Total	17,293	93	16

controls provided by web caches that allow sites to override header-specified caching policies.

A consequence of this observation is that user-agents cannot use cache headers to determine with certainty whether a resource has in fact been cached or not. This has important implications for WCD detection tools that rely on cache headers to infer the presence of WCD vulnerabilities.

5.3.3 Vulnerabilities

Table 5.6 presents a summary of the types of vulnerabilities discovered in the collected data, labeled by manual examination.

PII. 14 of the 16 vulnerable sites leaked PII of various kinds, including names, usernames, email addresses, and phone numbers. In addition to these four main categories, a variety of other categories of PII were found to be leaked. Broad examples of other PII include financial information (e.g., account balances, shopping history) and health information (e.g., calories

burned, number of steps, weight). While it is tempting to dismiss such information as trivial, we note that PII such as the above can be used as the basis for highly effective spearphishing attacks [37, 58, 55, 20].

Security Tokens. Using the entropy-based procedure described in Sec. 5.2, we also analyzed the data for the presence of leaked security tokens. Then, we manually verified our findings by accessing the vulnerable sites using a browser and checking for the presence of the tokens suspected to have been leaked. Finally, we manually verified representative examples of each class of leaked token for exploitability using the test accounts established during the measurement.

6 of the 16 vulnerable sites leaked CSRF tokens valid for a session, which could allow an attacker to conduct CSRF attacks despite the presence of a deployed CSRF defense. 3 of these were discovered in hidden form elements used to protect POST requests, while an additional 4 were found in inline JavaScript that was mostly used to initiate HTTP requests. We also discovered 2 sites leaking CSRF tokens in URL query parameters for GET requests, which is somewhat at odds with the convention that GET requests should be idempotent.

6 of the 16 vulnerable sites leaked session identifiers or user-specific API tokens in inline JavaScript. These session identifiers could be used to impersonate victim users at the vulnerable site, while the API tokens could be used to issue API requests as a victim user.

Authenticated vs. Unauthenticated Attackers. The methodology we described in Sec. 5.2 includes a detection step intended to discover whether a suspected WCD vulnerability was exploitable by an unauthenticated user by accessing a cached page without sending any stored session identifiers in the requests. In only a few cases did this automated check fail; that is,

in virtually every case the discovered vulnerability was exploitable by an unauthenticated user. Even worse, manual examination of the failure cases revealed that in each one the crawler had produced a false negative and that in fact all of the remaining vulnerabilities were exploitable by unauthenticated users as well. This implies that WCD, as a class of vulnerability, tends not to require an attacker to authenticate to a vulnerable site in order to exploit those vulnerabilities. In other words, requiring strict account verification through credentials such as valid SSNs or credit card numbers is not a viable mitigation for WCD.

Similarity. As a fundamental assumption, we expect the attacker to receive the exact same page in WCD as the one which has been sent to the victim and stored on CDN. It is worth mentioning that some WCD detection tools use the same assumption to detect the vulnerability. Since our crawler has recorded the response pages which has been sent to the victim along with the pages sent to both authenticated and unauthenticated attackers, we tried to verify this similarity assumption.

With the same assumption, we expected the page which has been sent to the victim to be the same as the one saved on CDN and sent to attacker. Interestingly, we observed only 99.2% similarity between the pages in 14 out of 16 (87.5%) sites sent to victims and the ones received by the attacker while the rest had slight changes in comparison with their corresponding pages. We investigated to find the reason behind this difference and we observed the difference had appeared in pages stored on Cloudflare CDN. Cloudflare uses the "Email Address Obfuscation" protection to avoid bots and scrappers from accessing the stored emails on the pages by replacing email addresses with some randomly generated numbers [27]. Since Python bots parse the page they cannot find the saved email address, while the requesting browser can render the page and produce the stored email address correctly. Because

Table 5.7: Attack scenarios.

Attack Scenario	**Vulnerable**	**Exploitable**
Session Hijacking	6	6
Cross-site Request Forgery	5	4
oAuth Redirection URI CSRF	1	1
Cross-site Script Inclusion (XSSI)	N/A	2

email's replacement is different for every access to the page contents stored on CDN, the returned page would be unexpectedly slightly different when attacker tries to access the same page.

Therefore, the pages seen by victim and attacker have not to be necessarily identical in all cases. This experiments gives us the takeaway that WCD protection tools which try to detect the attack only by relying on the similarity assumption, would experience false negatives due to any complicated network and protection settings on CDNs.

5.3.4 Practical Attack Scenarios

We studied the feasibility and exploitability of leaked security tokens with respect to potential attack scenarios to provide a better understanding of WCD risks and impacts. We randomly picked one site within each category of leakages listed in Table 5.6 and manually verified the exploitability of leaked security tokens. Outcomes of this analysis have been listed in Table 5.7.

Session Hijacking. We successfully exploited the session hijacking attack utilizing the leaked victim's session IDs. Notably all 6 selected sites were exploitable and we did not face any difficulty or defense-in-depth security protection during the exploitation process. In other words, the ease of exploitability and reproducibility imposes a critical risk for sites and increases the chance of account takeover or session hijacking attacks.

Circumvent CSRF Token. Using random tokens in any state changing requests are one of the most popular and recommended solution to mitigate CSRF [13, 132]. Based on our manual analysis, almost all vulnerable domains protected the state change requests with per-session CSRF Tokens. We did not observe any per-request CSRF token albeit the values might not be the same for different sites. These observations indicate the higher chance of exploitability when the token is leaked. It is worth mentioning that per-request CSRF token decreases the chance of successful CSRF attacks using leaked values in that it is situational and highly depends on frequency of user interactions with the site.

We have found 10 domains with leaked CSRF tokens including JS, POST and GET. We randomly picked 5 out of POST and GET and exploited the victim's browser to ensure that leaked CSRF token could be utilized in a practical CSRF attack. We did not select any leaked CSRF in inline JavaScript in that the vulnerable pages usually contain the leaked session which could be used in session hijacking with a higher damage potential, which has been tested in previous subsection.

As shown in different studies there may be other in-depth defense mechanisms besides the CSRF token such as Same-Site Cookies [141], Referer or Origin header check [13] or Double Submit Cookie [71]. However, there are studies which show some of mentioned in-depth defense mechanisms have pitfalls [62, 87, 72]. We did not find any of them during the exploitation scenario which could protect that victim from the attack. The only exception was one site which checked the Referer header and as a result our exploitation could not be succeeded. Lack of in-depth defense solutions would make WCD attack even more severe.

OAuth Redirect URI CSRF. We found one site which used a CSRF token to generate an OAuth *state* token in order to maintain the state between

the request and callback in Facebook's login process. This approach is the recommended CSRF protection for OAuth redirection URIs [11, 19, 48]. The leaked *state* token provides the attacker the opportunity to exploit the CSRF in OAuth redirection URI. Following the same attack scenario, we connected victim's account to attacker's Facebook account. Consequently, user's information may be saved or shared to a resource which is trusted and controlled by the attacker [114, 144, 123, 82, 39].

Cross-Site Script Inclusion. There are sites which usually generate and customize session-state dependent dynamic JavaScripts for each user on-the-fly. Dynamic JavaScripts may include sensitive data such as PII or security tokens and open a door for Cross-Site Script Inclusion (XSSI) attack [77]. XSSI requires an external dynamic script to be included in the attacker's web page via a HTML `script` tag which is not subject to *Same-Origin Policy* (SOP) [96]. Using an unguessable dynamic script URLs (random file name or random token in GET parameter) is a practical protection against this attack, which makes it impossible for the attacker to know the exact URL of the specific dynamic script. In WCD attack, because the whole victim's page contents are revealed, previously-mentioned defense mechanisms would no longer be effective to prevent XSSI.

We observed thousands of web pages through our experiments which contained unguessable dynamic script URLs, however, we considered it out of scope to investigate all those URLs to see if they are session-state dependent dynamic scripts with sensitive information included. We randomly picked few of the leaked unguessable JavaScript URLs for manual analysis and detected two pages utilizing session-dependent dynamic JavaScripts. We found one of these domains using dynamic script along with PII. However, the PII was also observable in the contents revealed through WCD. Nonetheless, there may be some cases in which the leaked web page through WCD attack has no

sensitive information contained in it, but the unguessable URL included in the leaked web contents would provide the attacker the opportunity of using that URL and run an XSSI attack, which brings him the PII or session ID.

Cache Poisoning. Assume an authenticated attacker who stored a malicious payload in her profile or other pages. The stored payload is just accessible within a valid attacker's session, so it would be unaccessible for a victim who is not authorized to access the injected page. An attacker can lure the cache server to store mentioned page by asking her browser to load `example.com/account.php/nonexistent.css`. As described in section 3.3, the web server would return a response for `account.php` with `nonexistent.css` as a parameter. The vulnerable cache server may consider the HTML response as a stylesheet file and cache (store) it. Henceforth, `nonexistent.css` which contains attacker's profile page along with malicious payload would be accessible for all users. Finally, an attacker will lure victim's browser to visit malicious page `account.php/nonexistent.css` and the store payload get executed.

In other words, The attack scenario is the reverse of the WCD scenario in some sense: unlike the sensitive information leakage, the attacker's goal is not to make the server to reveal stored users information to her browser. The goal, however, is to save her malicious script on the cache server and deceive the user to run it [68, 34].

5.3.5 Study Summary

Summarizing the major findings of this first experiment, we found that 16 out of 295 sites drawn from the Alexa Top 5K contained web cache deception (WCD) vulnerabilities. We note that while this is not a large fraction of the sites scanned, these sites have substantial user populations as to be expected with their placement in the Alexa rankings. This, combined with the

fact that WCD vulnerabilities are relatively easy to exploit, leads us to conclude that these vulnerabilities are serious and that this class of vulnerability deserves attention from both site administrators and the security community.

We found that the presence of cache headers was an unreliable indicator for whether a resource is cached, implying that existing detection tools relying on this signal may inadvertently produce false negatives when scanning sites for WCD vulnerabilities. We found vulnerable sites to leak PII that would be useful for launching spearphishing attacks, or security tokens that could be used to impersonate victim users or bypass important web security defenses. Finally, the WCD vulnerabilities discovered here did not require attackers to authenticate to vulnerable sites, meaning sites with restrictive sign-up procedures are not immune to WCD vulnerabilities.

5.4 Variations on Path Confusion

Web cache technologies may be configured to make their caching decisions based on complex rules such as pattern matches on file names, paths, and header contents. Launching a successful WCD attack requires an attacker to craft a malicious URL that triggers a caching rule, but also one that is interpreted as a legitimate request by the web server. Caching rules often cannot be reliably predicted from an attacker's external perspective, rendering the process of crafting an attack URL educated guesswork.

Based on this observation, we hypothesize that exploring variations on the path confusion technique may increase the likelihood of triggering caching rules and a valid web server response, and make it possible to exploit additional WCD vulnerabilities on sites that are not impacted by the originally proposed attack. To test our hypothesis, we performed a second round of measurements fourteen months after the first experiment, in July, 2019.

Specifically, we repeated our methodology, but tested payloads crafted

```
example.com/account.php
example.com/account.php/nonexistent.css
```

(a) Path Parameter

```
example.com/account.php
example.com/account.php%0Anonexistent.css
```

(b) Encoded Newline (\n)

```
example.com/account.php;par1;par2
example.com/account.php%3Bnonexistent.css
```

(c) Encoded Semicolon (;)

```
example.com/account.php#summary
example.com/account.php%23nonexistent.css
```

(d) Encoded Pound (#)

```
example.com/account.php?name=val
example.com/account.php%3Fname=valnonexistent.css
```

(e) Encoded Question Mark (?)

Figure 5.4: Five practical **path confusion** techniques for crafting URLs that reference **nonexistent file names**. In each example, the first URL corresponds to the regular page, and the second one to the malicous URL crafted by the attacker. More generally, **nonexistent.css** corresponds to a nonexistent file where **nonexistent** is an arbitrary string and **.css** is a popular static file extension such as .css, .txt, .jpg, .ico, .js etc.

with different path confusion techniques in an attempt to determine how many more pages could be exploited with path confusion variations. We used an extended seed pool for this study, containing 295 sites from the original set and an additional 45 randomly selected from the Alexa Top 5K, for a total of 340. In particular, we chose these new sites among those that *do not* use Google OAuth in an attempt to mitigate potential bias in our previous measurement. One negative consequence of this decision was that we had to perform the account creation step entirely manually, which limited the number of sites we could include in our study in this way. Finally,

we revised the URL grouping methodology by only selecting and exploiting a page among the first 500 pages when there is at least one marker in the content, making it more efficient for our purposes, and less resource-intensive on our targets. In the following, we describe this experiment and present our findings.

5.4.1 Path Confusion Techniques

Recall from our analysis and Table 5.4 that our WCD tests resulted in a `404 Not Found` status code in the great majority of cases, indicating that the web server returned an error page that is less likely to include PII. In order to increase the chances of eliciting a `200 OK` response while still triggering a caching rule, we propose additional path confusion techniques below based on prior work [127, 129, 130]), also illustrated in Fig. 5.4. Note that *Path Parameter* in the rest of this section refers to the original path confusion technique discussed in this work.

Encoded Newline (`\n`). Web servers and proxies often (but not always) stop parsing URLs at a newline character, discarding the rest of the URL string. For this path confusion variation, we use an encoded newline (`%0A`) in our malicious URL (see Fig. 5.4b). We craft this URL to exploit web servers that drop path components following a newline (i.e., the server sees `example.com/account.php`), but are fronted by caching proxies that instead do not properly decode newlines (`account.php%0Anonexistent.css`). As a result, a request for this URL would result in a successful response, and the cache would store the contents believing that this is static content based on the nonexistent file's extension.

Encoded Semicolon (`;`). Some web servers and web application frameworks accept lists of parameters in the URL delimited by semicolons; however, the

caching proxy fronting the server may not be configured to recognize such lists. The path confusion technique we present in Fig. 5.4c exploits this scenario by appending the nonexistent static file name after a semicolon. In a successful attack, the server would decode the URL and return a response for `example.com/account.php`, while the proxy would fail to decode the semicolon, interpret `example.com/account.php%3Bnonexistent.css` as a resource, and attempt to cache the nonexistent style sheet.

Encoded Pound (`#`). Web servers often process the pound character as an HTML fragment identifier, and therefore stop parsing the URL at its first occurrence. However, proxies and their caching rules may not be configured to decode pound signs, causing them to process the entire URL string. The path confusion technique we present in Fig. 5.4d once again exploits this inconsistent interpretation of the URL between a web server and a web cache, and works in a similar manner to the encoded newline technique above. That is, in this case the web server would successfully respond for `example.com/account.php`, while the proxy would attempt to cache `example.com/account.php%23nonexistent.css`.

Encoded Question Mark (`?`). This technique, illustrated in Fig. 5.4e, targets proxies with caching rules that are not configured to decode and ignore standard URL query strings that begin with a question mark. Consequently, the web server would generate a valid response for `example.com/account.php` and the proxy would cache it. More precisely, the proxy may misinterpret the same URL as `example.com/account.php%3Fname=valnonexistent.css`.

5.4.2 Results

We applied our methodology to the seed pool of 340 sites, using each path confusion variation shown in Fig. 5.4. We also performed the test with the

Table 5.8: Response codes observed with successful WCD attacks for each path confusion variation.

Technique	Pages		Domains		Sites	
	200	**!200**	**200**	**!200**	**200**	**!200**
Path Parameter	3,870	25,932	31	93	13	7
Encoded `\n`	1,653	24,280	79	76	9	7
Encoded `;`	3,912	25,576	91	92	13	7
Encoded `#`	7,849	20,794	102	85	14	7
Encoded `?`	11,282	26,092	122	86	17	8
All Encoded	11,345	31,063	128	94	20	9
Total	12,668	32,281	132	97	22	9

Table 5.9: Number of unique pages/domains/sites exploited by each path confusion technique. Element (i, j) indicates number of many pages exploitable using the technique in row i, whereas technique in column j is ineffective.

Technique	**Path P.**	**Encoded `\n`**	**Encoded `;`**	**Encoded `#`**	**Encoded `?`**
Path P.	-	4,390 / 26 / 7	1,010 / 5 / 4	5,691 / 11 / 3	5,673 / 12 / 3
Encoded `\n`	521 / 9 / 4	-	206 / 5 / 3	3,676 / 5 / 3	3,668 / 5 / 3
Encoded `;`	696 / 7 / 4	3,761 / 24 / 6	-	4,881 / 9 / 2	4,863 / 8 / 0
Encoded `#`	4,532 / 17 / 4	6,386 / 28 / 7	4,036 / 13 / 3	-	90 / 1 / 1
Encoded `?`	13,245 / 39 / 8	15,109 / 49 / 11	12,749 / 33 / 5	8,821 / 22 / 5	-
All Encoded	13,456 / 45 / 11	16,472 / 58 / 12	12,917 / 39 / 9	13,762 / 35 / 8	5,031 / 14 / 4

Path Parameter technique, which was an identical test case to our original experiment. We did this in order to identify those pages that are not vulnerable to the original WCD technique, but only to its variations.

We point out that the results we present in this second experiment for the Path Parameter technique differ from our first measurement. This suggests that, in the fourteen-month gap between the two experiments, either the site operators fixed the issue after our notification, or that there were changes to the site structure or caching rules that mitigated existing vulnerabilities or exposed new vulnerable pages. In particular, we found 16 vulnerable sites in the previous experiment and 25 in this second study, while the overlap

Table 5.10: Vulnerable targets for each path confusion variation.

Technique	Pages	Domains	Sites
Path Parameter	29,802 (68.9%)	103 (69.6%)	14 (56.0%)
Encoded `\n`	25,933 (59.9%)	86 (58.1%)	11 (44.0%)
Encoded `;`	29,488 (68.2%)	105 (70.9%)	14 (56.0%)
Encoded `#`	28,643 (66.2%)	109 (73.6%)	15 (60.0%)
Encoded `?`	37,374 (86.4%)	130 (87.8%)	19 (76.0%)
All Encoded	42,405 (98.0%)	144 (97.3%)	23 (92.0%)
Total	43,258 (100.0%)	148 (100.0%)	25 (100.0%)

between the two is only 4.

Of the 25 vulnerable sites we discovered in this experiment, 20 were among the previous set of 295 that uses Google OAuth, and 5 among the newly picked 45 that do not. To test whether the incidence distributions of vulnerabilities among these two sets of sites show a statistically significant difference, we applied Pearson's χ^2 test, where vulnerability incidence is treated as the categorical outcome variable and OAuth/non-OAuth site sets are comparison groups. We obtained a test statistic of 1.07 and a p-value of 0.30, showing that the outcome is independent of the comparison groups, and that incidence distributions do not differ significantly at typically chosen significance levels (i.e., $p > 0.05$). That is, our seed pool selection did not bias our findings.

Response Codes. We present the server response codes we observed for vulnerable pages in Table 5.8. Notice that there is a stark contrast in the number of `200 OK` responses observed with some of the new path confusion variations compared to the original. For instance, while there were 3,870 success codes for Path Parameter, Encoded `#` and Encoded `?` resulted in 7,849 and 11,282 success responses respectively. That is, two new path confusion techniques were indeed able to elicit significantly higher numbers of successful server responses, which is correlated with a higher chance of returning

private user information. The remaining two variations performed closer to the original technique.

Vulnerabilities. In this experiment we identified a total of 25 vulnerable sites. Table 5.10 shows a breakdown of vulnerable pages, domains, and sites detected using different path confusion variations. Overall, the original path confusion technique resulted in a fairly successful attack, exploiting 68.9% of pages and 14 sites. Still, the new techniques combined were able to exploit 98.0% of pages, and 23 out of 25 vulnerable sites, showing that they significantly increase the likelihood for a successful attack.

We next analyze whether any path confusion technique was able to successfully exploit pages that were not impacted by others. We present these results in Table 5.9 in a matrix form, where each element (i, j) shows how many pages/domains/sites were exploitable using the technique in row i, whereas utilizing the technique listed in column j was ineffective for the same pages/domains/sites.

The results in Table 5.9 confirm that each path confusion variation was able to attack a set of unique pages/domains/sites that were not vulnerable to other techniques, attesting to the fact that utilizing a variety of techniques increases the chances of successful exploitation. In fact, of the 25 vulnerable sites, 11 were only exploitable using one of the variations we presented here, but not the Path Parameter technique.

All in all, the results we present in this section confirm our hypothesis that launching WCD attacks with variations on path confusion, as opposed to only using the originally proposed Path Parameter technique, results in an increased possibility of successful exploitation. Moreover, two of the explored variations elicit significantly more `200 OK` server responses in the process, increasing the likelihood of the web server returning valid private information.

We stress that the experiment we present in this section is necessarily lim-

ited in scale and scope. Still, we believe the findings sufficiently demonstrate that WCD can be easily modified to render the attack more damaging, exploiting unique characteristics of web servers and caching proxies in parsing URLs. An important implication is that defending against WCD through configuration adjustments is difficult and error prone. Attackers are likely to have the upper hand in devising new and creative path confusion techniques that site operators may not anticipate.

5.5 Empirical Experiments

Practical exploitation of WCD vulnerabilities depends on many factors such as the caching technology used and caching rules configured. In this section, we present two empirical experiments we performed to demonstrate the impact of different cache setups on WCD, and discuss our exploration of the default settings for popular CDN providers.

5.5.1 Cache Location

While centralized server-side web caches can be trivially exploited from any location in the world, exploiting a distributed set of CDN cache servers is more difficult. A successful WCD attack may require attackers to correctly target the same edge server that their victim connects to, where the cached sensitive information is stored. As extensively documented in existing WCD literature, attackers often achieve that by connecting to the server of interest directly using its IP address and a valid HTTP `Host` header corresponding to the vulnerable site.

We tested the impact of this practical constraint by performing the *victim* interactions of our methodology from a machine located in Boston, MA, US, and launching the attack from another server in Trento, Italy. We repeated this test for each of the 25 sites confirmed to be vulnerable in our second

measurement described in Sec. 5.4.

The results showed that our attack failed for 19 sites as we predicted, requiring tweaks to target the correct cache server. Surprisingly, the remaining 6 sites were still exploitable even though headers indicated that they were served over CDNs (3 Akamai, 1 Cloudflare, 1 CloudFront, and 1 Fastly).

Upon closer inspection of the traffic, we found headers in our Fastly example indicating that a cache miss was recorded in their Italy region, followed by a retry in the Boston region that resulted in the cache hit, which led to a successful attack. We were not able to explore the remaining cases with the data servers exposed to us.

Many CDN providers are known to use a tiered cache model, where content may be available from a parent cache even when evicted from a child [38, 5]. The Fastly example above demonstrates this situation, and is also a plausible explanation for the remaining cases. Another possibility is that the vulnerable sites were using a separate centralized server-side cache fronted by their CDN provider. Unfortunately, without a clear understanding of proprietary CDN internals and visibility into site owners' infrastructure, it is not feasible to determine the exact cache interactions.

Our experiment confirms that cache location is a practical constraint for a successful WCD attack where a distributed set of cache servers is involved, but also shows that attacks are viable in certain scenarios without necessitating additional traffic manipulation.

5.5.2 Cache Expiration

Web caches typically store objects for a short amount of time, and then evict them once they expire. Eviction may also take place prematurely when web caches are under heavy load. Consequently, an attacker may have a limited window of opportunity to launch a successful WCD attack until the web cache drops the cached sensitive information.

Table 5.11: Default caching behavior for popular CDNs, and cache control headers honored by default to prevent caching.

CDN	Cached Objects	Honored Headers		
		no-store	**no-cache**	**private**
Akamai	Objects with a predefined list of static file extensions only.	✗	✗	✗
Cloudflare	Objects with a predefined list of static file extensions, **AND** all objects with cache control headers `public` or `max-age > 0`.	✓	✓	✓
CloudFront	All objects.	✓	✓	✓
Fastly	All objects.	✗	✗	✓

In order to measure the impact of cache expiration on WCD, we repeated the *attacker* interactions of our methodology with 1 hour, 6 hour, and 1 day delays. [3] We found that 16, 10, and 9 sites were exploitable in each case, respectively.

These results empirically demonstrate that exploitation is viable in realistic attack scenarios, where there are delays between the victim's and attacker's interactions with web caches. That being said, caches will eventually evict sensitive data as expected, meaning that attacks with shorter delays are more likely to be successful. We also note that we performed this test with a randomly chosen vulnerable page for each site as that was sufficient for our purposes. In practice, different resources on a given site may have varying cache expiration times, imposing additional constraints on what attacks are possible.

[3] We only tested 19 sites out of 25, as the remaining 6 had fixed their vulnerabilities by the time we performed this experiment.

5.5.3 CDN Configurations

Although any web cache technology can be affected by WCD, we established in Sec. 5.3.2 that CDNs play a large role in cache use on the Internet. Therefore, we conducted an exploratory experiment to understand the customization features CDN vendors offer and, in particular, to observe their default caching behavior. To that end, we created free or trial accounts with four major CDN providers: Akamai, Cloudflare, CloudFront, and Fastly. We only tested the basic content delivery solutions offered by each vendor and did not enable add-on features such as web application firewalls.

We stress that major CDN providers offer rich configuration options, including mechanisms for site owners to programmatically interact with their traffic. A systematic and exhaustive analysis of CDN features and corresponding WCD vectors is an extremely ambitious task beyond the scope of this chapter. The results we present in this section are only intended to give high-level insights into how much effort must be invested in setting up a secure and safe CDN environment, and how the defaults behave.

Configuration. All four CDN providers we experimented with offer a graphical interface and APIs for users to set up their origin servers, apply caching rules, and configure how HTTP headers are processed. In particular, all vendors provide ways to honor or ignore Cache-Control headers, and users can choose whether to strip headers or forward them downstream to clients. Users can apply caching decisions and time-to-live values for cached objects based on expressions that match the requested URLs.

Akamai and Fastly configurations are translated to and backed by domain-specific configuration languages, while Cloudflare and CloudFront do not expose their back-end to users. Fastly internally uses Varnish caches, and gives users full control over the Varnish Configuration Language (VCL) that governs their setup. In contrast, Akamai appears to support more powerful

HTTP processing features than Varnish, but does not expose all features to users directly. Quoting an Akamai blog post: *"Metadata [Akamai's configuration language] can do almost anything, good and bad, which is why WRITE access to metadata is restricted, and only Akamai employees can add metadata to a property configuration directly."* [6]

In addition to static configurations, both Akamai and Cloudflare offer mechanisms for users to write programs that execute on the edge server, and dynamically manipulate traffic and caches [30, 4].

In general, while Cloudflare, CloudFront, and Fastly offer free accounts suitable for personal use, they also have paid tiers that lift restrictions (e.g., Cloudflare only supports 3 cache rules in the free tier) and provide professional services support for advanced customization. Akamai strictly operates in the business-to-business market where configuration is driven by a professional services team, as described above.

Cacheability. Next, we tested the caching behavior of these CDN providers with a default configuration. Our observations here are limited to 200 OK responses pertaining to WCD; for an in-depth exploration of caching decisions involving 4xx or 5xx error responses, we refer readers to Nguyen et al. [98]. We summarize our observations in Table 5.11, which lists the conditions for caching objects in HTTP responses, and whether including the relevant Cache-Control headers prevent caching.

These results show that both Akamai and Cloudflare rely on a predefined list of static file extensions (e.g., .jpg, .css, .pdf, .exe) when making cacheability decisions. While Cloudflare allows origin servers to override the decision in both directions via Cache-Control headers, either to cache non-static files or prevent caching static files, Akamai's default rule applies unconditionally.

CloudFront and Fastly adopt a more aggressive caching strategy: in the absence of Cache-Control headers all objects are cached with a default time-

to-live value. Servers behind CloudFront can prevent caching via Cache-Control headers as expected. However, Fastly only honors the `private` header value.

5.5.4 Lessons Learned

The empirical evidence we presented in this section suggests that configuring web caches correctly is not a trivial task. Moreover, the complexity of detecting and fixing a WCD vulnerability is disproportionately high compared to launching an attack.

As we have seen above, many major CDN vendors do not make RFC-compliant caching decisions in their default configurations [40]. Even the more restrictive default caching rules based on file extensions are prone to security problems; for example, both Akamai and Cloudflare could cache dynamically generated PDF files containing tax statements if configured incorrectly. On the other hand, we do not believe that these observations implicate CDN vendors in any way, but instead emphasize that CDNs are not intended to be plug & play solutions for business applications handling sensitive data. All CDNs provide fine-grained mechanisms for caching and traffic manipulation, and site owners must carefully configure and test these services to meet their needs.

We reiterate that, while CDNs may be a prominent component of the Internet infrastructure, WCD attacks impact all web cache technologies. The complexity of configuring CDNs correctly, the possibility of multi-CDN arrangements, and other centralized caches that may be involved all imply that defending against WCD requires site owners to adopt a holistic view of their environment. Traditional security practices such as asset, configuration, and vulnerability management must be adapted to take into consideration the entire communication infrastructure as a system.

From an external security researcher's perspective the challenge is even

greater. As we have also discussed in the cache location and expiration experiments, reasoning about a web cache system's internals in a black box fashion is a challenging task, which in turn makes it difficult to pinpoint issues before they can be exploited. In contrast, attackers are largely immune to this complexity; they often do not need to disentangle the cache structure for a successful attack. Developing techniques and tools for reliable detection of WCD—and similar web cache attacks—is an open research problem. We believe a combination of systems security and safety approaches would be a promising research direction, which we discuss in Chapter 6.

5.6 Chapter Summary & Discussion

In this chapter, we presented the first large-scale investigation of WCD vulnerabilities in the wild, and showed that many sites among the Alexa Top 5K are impacted. We demonstrated that the vulnerable sites not only leak user PII but also secrets that, once stolen by an attacker, can be used to bypass existing authentication and authorization mechanisms to enable even more damaging web application attack scenarios.

Alarmingly, despite the severity of the potential damage, these vulnerabilities still persist more than two years after the public introduction of the attack in February 2017. Similarly, our second experiment showed that in the fourteen months between our two measurements, only 12 out of 16 sites were able to mitigate their WCD vulnerabilities, while the total number of vulnerabilities rose to 25.

One reason for this slow adoption of necessary mitigations could be a lack of user awareness. However, the attention WCD garnered from security news outlets, research communities, official web cache vendor press releases, and even mainstream media also suggests that there may be other contributing factors. In fact, it is interesting to note that there exists no technology or tool

proposed to date that allows site operators to reliably determine if any part of their online architecture is vulnerable to WCD, or to close their security gaps. Similarly, there does not exist a mechanism for end-users and web browsers to detect a WCD attack and protect themselves. Instead, countermeasures are largely limited to general guidance by web cache vendors and CDN providers for their users to configure their services in consideration of WCD vectors, and the tools available offer limited manual penetration-testing capabilities for site operators with domain-specific knowledge.

We assert that the above is a direct and natural consequence of the fact that WCD vulnerabilities are a *system safety* problem. In an environment with WCD vulnerabilities, there are no isolated faulty components; that is, web servers, load balancers, proxies, and caches all individually perform the functionality they are designed for. Similarly, determining whether there is human error involved and, if so, identifying where that lies are both non-trivial tasks. In fact, site operators often have legitimate needs to configure their systems in seemingly hazardous ways. For example, a global corporation operating hundreds to thousands of machines may find it technically or commercially infeasible to revise the Cache-Control header settings of their individual web servers, and may be forced to instruct their CDN provider to perform caching based purely on file names.

These are all strong indicators that the growing ecosystem of web caches, in particular CDN-fronted web applications, and more generally highly distributed Internet-based architectures, should be analyzed in a manner that captures their security and safety properties as a system. As aforementioned, venerable yet still widely-used *root cause analysis* techniques are likely to fall short in these efforts, because there is no individual system component to blame for the failure. Instead, security researchers should adopt a systems-centric security analysis, examining not only individual system components but also their interactions, expected outcomes, hazardous states, and acci-

dents that may result. Modeling and analyzing WCD attacks in this way, drawing from the rich safety engineering literature [79] is a promising future research direction that will help the security community understand and address similar systems-level attacks effectively.

Chapter 6

Conclusions and Future Directions

In this section, directions for further investigations would be discussed which could be followed to continue the research in this overlooked but important area of research. We conclude the thesis by summarizing our findings in two explored path confusion based attacks proposed in this thesis.

This thesis is based on different interpretations which create inconsistent perception of path component in URLs. WCD is a path confusion problem in web cache technologies; RPO is the result of path confusion in browsers. A promising future direction could be investigating security impacts of path confusion in other technologies like load balancer, proxies or even web application firewalls. Another potential direction would be investigating interpretation disagreements which occur for other components of a URL (query, fragment and host).

In chapter 4, we presented the first large-scale study of the Web to measure the prevalence and significance of style injection using RPO. This work shows that around 9 % of the sites in the Alexa Top 10,000 contain at least one vulnerable page, out of which more than one third can be exploited. We analyzed in detail various impediments to successful exploitation, and made recommendations for remediation. In contrast to script injection, relatively simple countermeasures exist to mitigate style injection. However,

there appears to be little awareness of this attack vector as evidenced by a range of popular Content Management Systems (CMSes) that we found to be exploitable.

In Chapter 5, we present the first large-scale study that quantifies the prevalence of WCD in 340 high-profile sites among the Alexa Top 5K. Our analysis reveals WCD vulnerabilities that leak private user data as well as secret authentication and authorization tokens that can be leveraged by an attacker to mount damaging web application attacks. Furthermore, we explore WCD in a scientific framework as an instance of the path confusion class of attacks, and demonstrate that variations on the path confusion technique used make it possible to exploit sites that are otherwise not impacted by the original attack. Our findings show that many popular sites remain vulnerable two years after the public disclosure of WCD. Our empirical experiments with popular CDN providers underline the fact that web caches are not plug & play technologies. In order to mitigate WCD, site operators must adopt a holistic view of their web infrastructure and carefully configure cache settings appropriate for their applications.

Bibliography

[1] Chrome remote debugging protocol. https://chromedevtools.github.io/devtools-protocol/, 2017.

[2] Ahmed Aboul-Ela. Sublist3r. https://github.com/aboul3la/Sublist3r.

[3] Steven Van Acker, Nick Nikiforakis, Lieven Desmet, Wouter Joosen, and Frank Piessens. FlashOver: Automated discovery of cross-site scripting vulnerabilities in rich internet applications. In ACM Symposium on Information, Computer and Communications Security (ASIACCS), 2012.

[4] Akamai Developer. Akamai EdgeWorkers. https://developer.akamai.com/akamai-edgeworkers-overview.

[5] Akamai Developer. Content Caching. https://developer.akamai.com/legacy/learn/Caching/Content_Caching.html.

[6] Akamai Developer – Jay Sikkeland. Advanced Metadata: A Brief Overview. https://developer.akamai.com/blog/2017/04/28/advanced-metadata-brief-overview.

[7] Akamai Technologies. Facts & Figures. https://www.akamai.com/us/en/about/facts-figures.jsp.

[8] Alexa. Top sites. `http://www.alexa.com/topsites`, 2016.

[9] Apache HTTP Server Project. Apache HTTP Server Version 2.4 – Caching Guide. `https://httpd.apache.org/docs/2.4/caching.html`.

[10] Sajjad Arshad, Seyed Ali Mirheidari, Tobias Lauinger, Bruno Crispo, Engin Kirda, and William Robertson. Large-Scale Analysis of Style Injection by Relative Path Overwrite. In International World Wide Web Conference, 2018.

[11] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis 1. Journal of Computer Security, 22(4):601–657, 2014.

[12] Adam Barth, Juan Caballero, and Dawn Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In IEEE Symposium on Security and Privacy (S&P), 2009.

[13] Adam Barth, Collin Jackson, and John C Mitchell. Robust defenses for cross-site request forgery. In Proceedings of the 15th ACM conference on Computer and communications security (CCS), pages 75–88. ACM, 2008.

[14] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side xss filters. In International World Wide Web Conference (WWW), 2010.

[15] Shay Berkovich. ProxySG and Web Cache Deception. Symantec Connect, 2017. `https://www.symantec.com/connect/blogs/proxysg-and-web-cache-deception`.

[16] Prithvi Bisht and V. N. Venkatakrishnan. XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), 2008.

[17] Benjamin Brown. On Web Cache Deception Attacks. The Akamai Blog, 2017. https://blogs.akamai.com/2017/03/on-web-cache-deception-attacks.html.

[18] Burp Suite. https://portswigger.net/burp/, 2017.

[19] Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Clara Schneidewind, Marco Squarcina, and Mauro Tempesta. Wpse: Fortifying web protocols via browser-side security monitoring. In 27th USENIX Security Symposium, pages 1493–1510, 2018.

[20] Deanna D. Caputo, Shari Lawrence Pfleeger, Jesse D. Freeman, and M. Eric Johnson. Going Spear Phishing: Exploring Embedded Training and Awareness. In IEEE Security & Privacy, 2014.

[21] Orcun Cetin, Carlos Ganan, Maciej Korczynski, and Michel van Eeten. Make notifications great again: Learning how to notify in the age of large-scale vulnerability scanning. In Workshop on the Economics of Information Security (WEIS), 2017.

[22] Catherine Chapman. Web cache deception named top web hacking technique of 2019, 2019. https://portswigger.net/daily-swig/web-cache-deception-named-top-web-hacking-technique-of-2019.

[23] Jianjun Chen, Jian Jiang, Haixin Duan, Nicholas Weaver, Tao Wan, and Vern Paxson. Host of troubles: Multiple host ambiguities in http implementations. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 1516–1527, 2016.

[24] Jianjun Chen, Jian Jiang, Xiaofeng Zheng, Haixin Duan, Jinjin Liang, Kang Li, Tao Wan, and Vern Paxson. Forwarding-Loop Attacks in Content Delivery Networks. In The Network and Distributed System Security Symposium, 2016.

[25] Ka-Hing Cheung. Web Cache Deception Attack revisited. Cloudflare Blog, 2018. https://blog.cloudflare.com/web-cache-deception-attack-revisited/.

[26] Catalin Cimpanu. Web Cache Deception Attack Tricks Servers Into Caching Pages with Personal Data. Bleeping Computer, 2017. https://www.bleepingcomputer.com/news/security/web-cache-deception-attack-tricks-servers-into-caching-pages-with-personal-data/.

[27] Cloudflare. What is Email Address Obfuscation? https://support.cloudflare.com/hc/en-us/articles/200170016-What-is-Email-Address-Obfuscation-.

[28] Cloudflare. Origin Cache-Control. https://support.cloudflare.com/hc/en-us/articles/115003206852s.

[29] Cloudflare. The Cloudflare Global Anycast Network. https://www.cloudflare.com/network/.

[30] Cloudflare Developers. Cloudflare Workers Documentation. https://developers.cloudflare.com/workers/.

[31] Common Crawl. https://commoncrawl.org/, August 2016.

[32] Soroush Dalili. Non-root-relative path overwrite (RPO) in IIS and .Net applications. https://soroush.secproject.com/blog/2015/02/non-root-relative-path-overwrite-rpo-in-iis-and-net-applications/, 2015.

[33] Soroush Dalili. Non-Root-Relative Path Overwrite (RPO) in IIS and .Net Applications, 2015. https://soroush.secproject.com/blog/2015/02/non-root-relative-path-overwrite-rpo-in-iis-and-net-applications/.

[34] Louis Dion-Marcil, Laurent Desaulniers, and Olivier Bilodeau. Edge Side Include Injection: Abusing Caching Servers into SSRF and Transparent Session Hijacking. Black Hat USA, 2018. https://i.blackhat.com/us-18/Wed-August-8/us-18-Dion_Marcil-Edge-Side-Include-Injection-Abusing-Caching-Servers-into-SSRF-and-Transparent-Session-Hijacking-wp.pdf.

[35] Akamai Documentation. Caching, 2019. https://learn.akamai.com/en-us/webhelp/ion/oca/GUID-AAA2927B-BFF8-4F25-8CFE-9D8E920C008F.html.

[36] Adam Doupe, Weidong Cui, Mariusz H. Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. deDacota: Toward preventing server-side XSS via automatic code and data separation. In ACM Conference on Computer and Communications Security (CCS), 2013.

[37] Julie S. Downs, Mandy B. Holbrook, and Lorrie Faith Cranor. Decision Strategies and Susceptibility to Phishing. In Symposium On Usable Privacy and Security, 2006.

[38] Fastly – Hooman Beheshti. The truth about cache hit ratios. https://www.fastly.com/blog/truth-about-cache-hit-ratios.

[39] Daniel Fett, Ralf Küsters, and Guido Schmitz. The web sso standard openid connect: In-depth formal security analysis and security guidelines. arXiv preprint arXiv:1704.08539, 2017.

[40] Roy T. Fielding, Mark Nottingham, and Julian F. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Caching. IETF – RFC 7234, 2014. https://www.rfc-editor.org/info/rfc7234.

[41] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. Blocking-Resistant Communication Through Domain Fronting. In Privacy Enhancing Technologies, 2015.

[42] Omer Gil. Web cache deception attack. In Black Hat USA, 2017.

[43] Omer Gil. Web cache deception attack. http://omergil.blogspot.com/2017/02/web-cache-deception-attack.html, 2017.

[44] Omer Gil. Web Cache Deception Attack, 2017. https://omergil.blogspot.com/2017/02/web-cache-deception-attack.html.

[45] Omer Gil. Web Cache Deception Attack. Black Hat USA, 2017. https://www.blackhat.com/us-17/briefings.html#web-cache-deception-attack.

[46] Google. Chrome Remote Debugging Protocol. https://chromedevtools.github.io/devtools-protocol/.

[47] Run Guo, Jianjun Chen, Baojun Liu, Jia Zhang, Chao Zhang, Haixin Duan, Tao Wan, Jian Jiang, Shuang Hao, and Yaoqi Jia. Abusing CDNs for Fun and Profit: Security Issues in CDNs' Origin Validation. In IEEE International Symposium on Reliable Distributed Systems, 2018.

[48] Dick Hardt. The oauth 2.0 authorization framework. IETF – RFC 6749, 2012. https://www.rfc-editor.org/info/rfc6749.

[49] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. Scriptless attacks - stealing the pie without touching the sill. In ACM Conference on Computer and Communications Security (CCS), 2012.

[50] Mario Heiderich, Christopher Späth, and Jörg Schwenk. Dompurify: Client-side protection against xss and markup injection. In European Conference on Research in Computer Security (ESORICS), 2017.

[51] Michael Henriksen. AQUATONE. `https://github.com/michenriksen/aquatone`.

[52] Gareth Heyes. The sexy assassin: Tactical exploitation using CSS. `https://docs.google.com/viewer?url=www.businessinfo.co.uk/labs/talk/The_Sexy_Assassin.ppt`, 2009.

[53] Gareth Heyes. RPO. `http://www.thespanner.co.uk/2014/03/21/rpo/`, 2014.

[54] John Holowczak and Amir Houmansadr. CacheBrowser: Bypassing Chinese Censorship Without Proxies Using Cached Content. In ACM Conference on Computer and Communications Security, 2015.

[55] Jason Hong. The State of Phishing Attacks. Communications of the ACM, 55(1):74–81, 2012.

[56] Lin-Shung Huang, Zack Weinberg, Chris Evans, and Collin Jackson. Protecting browsers from cross-origin CSS attacks. In ACM Conference on Computer and Communications Security (CCS), 2010.

[57] Arbaz Hussain. Auto Web Cache Deception Tool, 2017. `https://medium.com/@arbazhussain/auto-web-cache-deception-tool-2b995c1d1ab2`.

[58] Tom N. Jagatic, Nathaniel A. Johnson, Markus Jakobsson, and Filippo Menczer. Social Phishing. Communications of the ACM, 50(10):94–100, 2007.

[59] Artur Janc and Lukasz Olejnik. Feasibility and real-world implications of web browser history detection. In Web 2.0 Security and Privacy (W2SP), 2010.

[60] XSS Jigsaw. RPO Gadgets, 2016. https://blog.innerht.ml/rpo-gadgets/.

[61] Lin Jin, Shuai Hao, Haining Wang, and Chase Cotton. Your Remnant Tells Secret: Residual Resolution in DDoS Protection Services. In IEEE/IFIP International Conference on Dependable Systems and Networks, 2018.

[62] Martin Johns and Justus Winter. Requestrodeo: Client side protection against session riding. In Proceedings of the OWASP Europe 2006 Conference, 2006.

[63] Josh Kaufman. 10,000 Most Common English Words, 2013. https://github.com/first20hours/google-10000-english.

[64] Christoph Kern. Securing the tangled web. Communications of the ACM, 57, no. 9:38–47, 2014.

[65] Christoph Kerschbaumer. Mitigating MIME confusion attacks in firefox. https://blog.mozilla.org/security/2016/08/26/mitigating-mime-confusion-attacks-in-firefox/, 2016.

[66] James Kettle. Detecting and exploiting path-relative stylesheet import (PRSSI) vulnerabilities. http://blog.portswigger.net/2015/02/prssi.html, 2015.

[67] James Kettle. Detecting and Exploiting Path-Relative Stylesheet Import (PRSSI) Vulnerabilities. PortSwigger Web Security Blog, 2015. https://portswigger.net/blog/detecting-and-exploiting-path-relative-stylesheet-import-prssi-vulnerabilities.

[68] James Kettle. Practical Web Cache Poisoning. PortSwigger Web Security Blog, 2018. https://portswigger.net/blog/practical-web-cache-poisoning.

[69] James Kettle. HTTP Desync Attacks: Request Smuggling Reborn. PortSwigger Web Security Blog, 2019. https://portswigger.net/blog/http-desync-attacks-request-smuggling-reborn.

[70] Masato Kinugawa. CSS based attack: Abusing unicode-range of @font-face. http://mksben.l0.cm/2015/10/css-based-attack-abusing-unicode-range.html, 2015.

[71] Manideep Konakandla, Dave Wichers, Paul Petefish, Eric Sheridan, and Dominique Righetto. Cross-site request forgery (csrf) prevention cheat sheet. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet.

[72] Krzysztof Kotowicz. Stripping referrer for fun and profit. http://blog.kotowicz.net/2011/10/stripping-referrer-for-fun-and-profit.html, 2011.

[73] Sebastian Lekies. How to bypass CSP nonces with DOM XSS. http://sirdarckcat.blogspot.com/2016/12/how-to-bypass-csp-nonces-with-dom-xss.html, 2016.

[74] Sebastian Lekies, Krzysztof Kotowicz, Samuel Grob, Eduardo A. Vela Nava, and Martin Johns. Code-reuse attacks for the web: Breaking

cross-site scripting mitigations via script gadgets. In ACM Conference on Computer and Communications Security (CCS), 2017.

[75] Sebastian Lekies, Krzysztof Kotowicz, and Eduardo Vela Nava. Breaking xss mitigations via script gadgets. In Black Hat USA, 2017.

[76] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later - large-scale detection of DOM-based XSS. In ACM Conference on Computer and Communications Security (CCS), 2013.

[77] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. The Unexpected Dangers of Dynamic JavaScript. In USENIX Security Symposium, 2015.

[78] Chris Lesniewski-Laas and M. Frans Kaashoek. SSL Splitting: Securely Serving Data from Untrusted Caches. In USENIX Security Symposium, 2003.

[79] Nancy G. Leveson. Engineering a Safer World. The MIT Press, Cambridge, MA, USA, 2011.

[80] Amit Levy, Henry Corrigan-Gibbs, and Dan Boneh. Stickler: Defending against Malicious Content Distribution Networks in an Unmodified Browser. In IEEE Security & Privacy (S&P), 2016.

[81] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Michael Bailey, Damon McCoy, Stefan Savage, and Vern Paxson. You've got vulnerability: Exploring effective vulnerability notifications. In USENIX Security Symposium, 2016.

[82] Wanpeng Li and Chris J Mitchell. Security issues in oauth 2.0 sso implementations. In International Conference on Information Security, pages 529–541. Springer, 2014.

[83] Bin Liang, Wei You, Liangkun Liu, Wenchang Shi, and Mario Heiderich. Scriptless timing attacks on web browser privacy. In IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2014.

[84] Joshua Liebow-Feeser. Understanding Our Cache and the Web Cache Deception Attack. Cloudflare Blog, 2017. https://blog.cloudflare.com/understanding-our-cache-and-the-web-cache-deception-attack/.

[85] Nera W. C. Liu and Albert Yu. Ultimate DOM based XSS detection scanner on cloud. In Black Hat Asia, 2014.

[86] Mike Ter Louw and V.N. Venkatakrishnan. BLUEPRINT: Robust prevention of cross-site scripting attacks for existing browsers. In IEEE Symposium on Security and Privacy (S&P), 2009.

[87] Rich Lundeen. The deputies are still confused. Blackhat EU, 2013.

[88] Giorgio Maone. NoScript. https://noscript.net/, 2009.

[89] McAfee. Customer URL Ticketing System. https://www.trustedsource.org/.

[90] MDN. X-Content-Type-Options. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Content-Type-Options, 2018.

[91] Nikolaos Michalakis, Robert Soulé, and Robert Grimm. Ensuring Content Integrity for Untrusted Peer-to-Peer Content Distribution Networks. In USENIX Symposium on Networked Systems Design & Implementation, 2007.

[92] Microsoft. Understanding the compatibility view list. https://msdn.microsoft.com/en-us/library/gg699485(v=vs.85).aspx, 2015.

[93] Mozilla. MDN web docs – HTTP Cache. https://developer.mozilla.org/en-US/docs/Mozilla/HTTP_cache.

[94] Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, and Martin Johns. Scriptprotect: mitigating unsafe third-party javascript practices. In Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, pages 391–402, 2019.

[95] Yacin Nadji, Prateek Saxena, and Dawn Song. Document structure integrity: A robust basis for cross-site scripting defense. In Network and Distributed System Security Symposium (NDSS), 2009.

[96] MOZILLA DEVELOPER NETWORK. Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[97] NGINX. NGINX Content Caching. https://docs.nginx.com/nginx/admin-guide/content-cache/content-caching/.

[98] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack. In ACM Conference on Computer and Communications Security, 2019.

[99] Mark Nottingham. How (Not) to Control Your CDN, 2017. https://www.mnot.net/blog/2017/06/07/safe_cdn.

[100] Terri Oda, Glenn Wurster, P. C. van Oorschot, and Anil Somayaji. SOMA: Mutual approval for included content in web pages. In ACM Conference on Computer and Communications Security (CCS), 2008.

[101] Kaan Onarlioglu. Security Researchers Struggle with Bot Management Programs. Dark Reading, 2018. https://www.darkreading.com/perimeter/security-researchers-struggle-with-bot-management-programs/a/d-id/1332976.

[102] OWASP. Amass. https://github.com/OWASP/Amass.

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

[104] OWASP. Clickjacking defense cheat sheet. https://www.owasp.org/index.php/Clickjacking_Defense_Cheat_Sheet, 2017.

[105] OWASP. Cross-site request forgery (csrf) prevention cheat sheet. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet, 2017.

[106] OWASP. XSS (cross site scripting) prevention cheat sheet. https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet, 2017.

[107] Google Identity Platform. Using OAuth 2.0 to Access Google APIs. https://developers.google.com/identity/protocols/OAuth2.

[108] PortSwigger. Top 10 web hacking techniques of 2019, 2019. https://portswigger.net/research/top-10-web-hacking-techniques-of-2019.

[109] Kenneth Reitz. Requests: HTTP for Humans. http://docs.python-requests.org/en/master/.

[110] David Ross. IE 8 XSS filter architecture / implementation. https://blogs.technet.microsoft.com/srd/2008/08/19/ie-8-xss-filter-architecture-implementation/, 2008.

[111] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting frame busting: a study of clickjacking vulnerabilities on popular sites. In IEEE Oakland Web 2.0 Security and Privacy (W2SP), 2010.

[112] Mike Samuel, Prateek Saxena, and Dawn Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In ACM Conference on Computer and Communications Security (CCS), 2011.

[113] SeleniumHQ. Selenium – Web Browser Automation. `https://www.seleniumhq.org/`.

[114] Ethan Shernan, Henry Carter, Dave Tian, Patrick Traynor, and Kevin Butler. More guidelines than rules: Csrf vulnerabilities from non-compliant oauth 2.0 implementations. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), pages 239–260. Springer, 2015.

[115] Henri Sivonen. Activating browser modes with doctype. `https://hsivonen.fi/doctype/`, 2013.

[116] Johan Snyman. Airachnid: Web Cache Deception Burp Extender. Trustwave – SpiderLabs Blog, 2017. `https://www.trustwave.com/Resources/SpiderLabs-Blog/Airachnid--Web-Cache-Deception-Burp-Extender/`.

[117] Squid. Squid: Optimising Web Delivery. `http://www.squid-cache.org/`.

[118] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In International World Wide Web Conference (WWW), 2010.

[119] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild. 2019.

[120] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. Precise client-side protection against DOM-based cross-site scripting. In USENIX Security Symposium, 2014.

[121] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. Hey, you have a problem: On the feasibility of large-scale web vulnerability notification. In USENIX Security Symposium, 2016.

[122] Volker Stocker, Georgios Smaragdakis, William Lehr, and Steven Bauer. The growing complexity of content delivery networks: Challenges and implications for the Internet ecosystem. Telecommunications Policy, 41(10):1003–1016, 2017.

[123] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of oauth sso systems. In Proceedings of the 2012 ACM conference on Computer and communications security, pages 378–390. ACM, 2012.

[124] Takeshi Terada. A Few RPO Exploitation Techniques, 2015. `https://www.mbsd.jp/Whitepaper/rpo.pdf`.

[125] Takeshi Terada. A few RPO exploitation techniques. `https://www.mbsd.jp/Whitepaper/rpo.pdf`, 2015.

[126] The Chromium Projects. HTTP Cache. `https://www.chromium.org/developers/design-documents/network-stack/http-cache`.

[127] Aleksei Tiurin. A Fresh Look On Reverse Proxy Related Attacks, 2019. `https://www.acunetix.com/blog/articles/a-fresh-look-on-reverse-proxy-related-attacks`.

[128] Sipat Triukose, Zakaria Al-Qudah, and Michael Rabinovich. Content Delivery Networks: Protection or Threat? In European Symposium on Research in Computer Security, 2009.

[129] Orange Tsai. A New Era of SSRF - Exploiting URL Parser in Trending Programming Languages! Black Hat USA, 2017. https://www.blackhat.com/us-17/briefings.html#a-new-era-of-ssrf-exploiting-url-parser-in-trending-programming-languages.

[130] Orange Tsai. Breaking Parser Logic: Take Your Path Normalization off and Pop 0days Out! Black Hat USA, 2018. https://www.blackhat.com/us-18/briefings/schedule/index.html#breaking-parser-logic-take-your-path-normalization-off-and-pop-days-out-10346.

[131] Mark Tsimelzon, Bill Weihl, Joseph Chung, Dan Frantz, John Brasso, Chris Newton, Mark Hale, Larry Jacobs, and Conleth O'Connell. ESI Language Specification 1.0. World Wide Web Consortium (W3C), 2001. https://www.w3.org/TR/esi-lang.

[132] Tom Van Goethem, Ping Chen, Nick Nikiforakis, Lieven Desmet, and Wouter Joosen. Large-scale security analysis of the web: Challenges and findings. In International Conference on Trust and Trustworthy Computing, pages 110–126. Springer, 2014.

[133] Varnish. Varnish HTTP Cache. https://varnish-cache.org/.

[134] Thomas Vissers, Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. Maneuvering Around Clouds: Bypassing Cloud-based Security Providers. In ACM Conference on Computer and Communications Security, 2015.

[135] W3C. Css syntax and basic data types. http://www.w3.org/TR/CSS2/syndata.html, 2011.

[136] W3C. Content security policy level 2. https://www.w3.org/TR/CSP2/, 2015.

[137] David Y. Wang, Stefan Savage, and Geoffrey M. Voelker. Cloak and Dagger: Dynamics of Web Search Cloaking. In ACM Conference on Computer and Communications Security, 2011.

[138] Wappalyzer. Identify technologies on websites. https://www.wappalyzer.com/, 2017.

[139] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In ACM Conference on Computer and Communications Security (CCS), 2016.

[140] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. An empirical analysis of XSS sanitization in web application frameworks. In European Conference on Research in Computer Security (ESORICS), 2011.

[141] M. West and M. Goodwin. Same-site cookies. https://tools.ietf.org/html/draft-ietf-httpbis-cookie-same-site-00, 2016.

[142] XSS Jigsaw. CSS: Cascading style scripting. http://blog.innerht.ml/cascading-style-scripting/, 2015.

[143] XSS Jigsaw. RPO gadgets. http://blog.innerht.ml/rpo-gadgets/, 2016.

[144] William Zeller and Edward W Felten. Cross-site request forgeries: Exploitation and prevention. The New York Times, pages 1–13, 2008.

[145] Hadi Zolfaghari and Amir Houmansadr. Practical Censorship Evasion Leveraging Content Delivery Networks. In ACM Conference on Computer and Communications Security, 2016.