Large-Scale Analysis of Style Injection by Relative Path Overwrite

Sajjad Arshad
Northeastern University
arshad@ccs.neu.edu

Seyed Ali Mirheidari
University of Trento
seyedali.mirheidari@unitn.it

Tobias Lauinger
Northeastern University
p672@tobias.lauinger.name

Bruno Crispo
University of Trento
bruno.crispo@unitn.it

Engin Kirda
Northeastern University
ek@ccs.neu.edu

William Robertson
Northeastern University
wkr@ccs.neu.edu

ABSTRACT

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 differences in how browsers and web servers interpret relative paths (i.e., *path confusion*) 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.

In this paper, we present the first large-scale study of the Web to measure the prevalence and significance of style injection using RPO. Our 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 analyze in detail various impediments to successful exploitation, and make 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.

KEYWORDS

Relative Path Overwrite; Scriptless Attack; Style Injection

ACM Reference Format:
Sajjad Arshad, Seyed Ali Mirheidari, Tobias Lauinger, Bruno Crispo, Engin Kirda, and William Robertson. 2018. Large-Scale Analysis of Style Injection by Relative Path Overwrite. In *WWW 2018: The 2018 Web Conference, April 23–27, 2018, Lyon, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3178876.3186090

1 INTRODUCTION

Cross-Site Scripting (XSS) [37] 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 [14] and exfiltrate secrets from a site.

The aforementioned injection attacks typically arise due to the lack of separation between code and data [11], 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. 研究意义一

There are many existing techniques of how style directives could be injected into a site [14, 18]. A relatively recent class of attacks is Relative Path Overwrite (RPO), first proposed in a blog post by Gareth Heyes [17] 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 [48]. 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 [17]. 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 paper, 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 [9], 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 [3] in the Common

WWW 2018, April 23–27, 2018, Lyon, France

ACM ISBN 978-1-4503-5639-8/18/04.
https://doi.org/10.1145/3178876.3186090

尽管脚本注入攻击比基于样式注入的攻击更强大，但它们作为一种威胁也更广为人知，而且web开发人员更有可能采取措施使其变得更困难。从攻击者的角度来看，在不可能进行脚本注入的情况下，样式注入攻击可能是一种选择

跨站脚本攻击（Cross-Site Scripting，XSS）一直是最常见的Web攻击之一，
传统的XSS攻击就是攻击者将可执行脚本注入站点中，然后受害者的Web浏览器去执行这一段被注入的脚本，
但是近期的研究表明，脚本攻击（Script Injection） 不一定就是有效攻击，
在一定的条件下，通过注入构造好的CSS样式指令同样可以实现脚本攻击的效果。

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 these could be used to mitigate these vulnerabilities.

The contributions of this paper are summarized as follows:

- We present the first automated and large-scale study of the prevalence and significance of RPO vulnerabilities in the wild.
- We discuss a range of factors that prevent a vulnerability from being exploited, and find that simple countermeasures exist to mitigate RPO.
- We link many exploitable pages to installations of Content Management Systems (CMSes), and notify the vendors.

2 BACKGROUND & RELATED WORK

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.

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 [40]. The inclusion of unsanitized inputs could occur on the server side or client side, and in a persistent *stored* or volatile *reflected* way [37]. 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.

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 [5] and Content Security Policy [45, 50] 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 [19, 29], exfiltrating arbitrary content [23], reading HTML attributes [16, 24], and bypassing Clickjacking defenses [16]. 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. [14] 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 [39] without script execution.

Rather than using layout-based information leaks to exfiltrate data from a page, Huang et al. [18] 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 [49]. 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 [55].

2.3 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, 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://example.com/rpo/test.php/ with an added trailing slash, 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 [17], 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 [34], escalating style injection to XSS [54]. Another blog post by Dalili [10] 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 [22] coins the term Path Relative StyleSheet Import (PRSSI) for a specific subset of RPO attacks, introduces a PRSSI vulnerability scanner for Burp Suite [7], and proposes countermeasures. Terada [48] provides more exploitation techniques for various browsers or certain web applications, and [55] 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 [12, 13]. 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.

2.4 Preconditions for RPO Style Attacks

For the purpose of this paper, 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.

没太懂？？

- 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 [44]; 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 4.3.1 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 [22]. 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 3 tests how often these preconditions hold in the wild in order to quantify the vulnerability and exploitability of pages with respect to RPO attacks.

2.5 Related Work

In the previous sections, 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 [7] 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.

和他人工作的区别

The separate class of script-based attacks has been studied extensively, such as systematic analysis of XSS sanitization frameworks [53], detecting XSS vulnerabilities in Rich Internet Applications [2], large-scale detection of DOM-based XSS [27, 30], and bypassing XSS mitigations by Script Gadgets [25, 26]. An array of XSS prevention mechanisms have been proposed, such as XSS Filter [41], XSS-Guard [6], SOMA [36], BluePrint [31], Document Structure Integrity [35], XSS Auditor [5], NoScript [32], Context-Sensitive Auto-Sanitization (CSAS) [43], DOM-based XSS filtering using runtime taint tracking [46], preventing script injection through software design [20], Strict CSP [52], and DOMPurify [15]. However, the vulnerability measurements and proposed countermeasures of these works on script injection do not apply to RPO-based style injection.

但是，这些关于脚本注入的工作的漏洞度量和建议的对策并不适用于基于rpo的风格注入

3 METHODOLOGY

？什么技术

Our methodology consists of three main phases. 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.

采用这些技术的目的

3.1 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:

数据来源

数据筛选

(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. For example, we would group example.com/?lang=en and example.com/?lang=fr.

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

最后的实验数据规模是来自222千个站点的137万个页面

3.2 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.

在高级别上，爬虫模拟浏览器扩展相对路径的方式，并测试是否可以使用路径混淆将样式指令注入作为样式表加载的资源。

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 2.3). The crawler then requests each unique stylesheet URL until one has been found to reflect the injected style in the response.

检查response中是否包含<base>标签，如果包含可以说明该页面是安全的。

The style directive we inject to test for reflection vulnerabilities is shown in the legend of Figure 1. 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 2.3. 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.

图1a显示了一个通用示例，其中URL中没有参数。
由于爬虫不知道有效参数的名称，它只需将有效负载作为子目录附加到URL的末尾

正常的URL
attacker构造过的URL
假设每个HTML页面都引用一个样式表../style/css，浏览器在解析样式表路径会像第三个URL一样解析

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

(a) Path Parameter (Simple)

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

(b) Path Parameter (PHP or ASP)

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

(c) Path Parameter (JSP)

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

(d) Encoded Path

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

%3F就是对？进行编码后的结果

(e) Encoded Query

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

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

(f) Cookie

path confusion和style injection的各种变种

Figure 1: 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.

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.

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 1a 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.

请在示例中，附加了两个斜杠，
以便浏览器在将样式表引用扩展到父目录(./style.cs s)时不会从URL中删除PayLoad。

在实际的爬行中，我们总是附加20个斜杠，以避免必须考虑不同数量的父目录。

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 1b, PHP or ASP) or semicolons (Figure 1c, 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.

这种技术针对的是像IIS（还有Apache、Ngnix）这样的网络服务器，
这些服务器解码URL中编码的斜杠以进行目录遍历，而Web浏览器则没有。

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 1d). 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 1e. 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 1f, 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.

如果攻击者可以创建新的cookie或篡改现有的cookie(与其他技术相比，这是一种强有力的假设)，并且页面在响应中反映cookie值，那么就可以进行CSS注入

Payload被注入每个cookie值，
并由爬虫作为HTTP请求头发送

3.3 Exploitability Analysis

在上面通过Python Crawler挑选出易受攻击的页之后，还要在真实的浏览器，如Chrome上进一步检查这些页面是否真的能够在真正的浏览器上利用

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.

为此，我们基于GoogleChrome建了一个爬虫，并使用Remote Debugging Protoca驱动浏览器并记录HTTP请求和响应。此外，我们开发了一个Chrome扩展，用我们的payload填充CSS样式表请求中的cookie头。

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 1 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

样式指令使用随机生成的URL为HTML主体页面加载背景图像。
通过检查是否有向外部请求img图像的HTTP请求流量来判断是否注入成功

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 [22]. 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.

3.4 Limitations

RPO是一类攻击，我们的方法只涵盖其中的一个子集

文章中进行的实验都是反射型的css样式注入，而不是存储型(stored)样式注入

相比于爬虫查找，人工查找可能可以发现更多的注入风格。

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.

缺点1

缺点2

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

一个伦理问题是，注入的CSS可能存储在服务器上，而不是反映在响应中，因此可能破坏站点。我们采取了几个谨慎的步骤，以尽量减少任何有害的副作用，我们调查的网站。

3.5 Ethical Considerations

1. 我们没有尝试登录到网站，我们只在页面的公开版本上测试了RPO。

2. 不提交任何表单

3.为了避免触发Java脚本事件，我们没有单击页面中的任何按钮或链接

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.

我们注意到在使用方面存在一些不一致之处。

4 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 1: 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

Tested Pages是从page groups中随机选出的测试页面

Table 1 shows a summary of our dataset. *Tested Pages* refers to the set of randomly selected pages from the page groups as discussed in Section 3.1. For brevity, we are referring to *Tested Pages* wherever we mention pages in the remainder of the paper.

十个最大的网站中有六个在我们的候选集中。
也就是说，它们包含在公共爬行中，并且具有相对的样式路径

4.1 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 2. 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.

我们的候选集包含较大站点的较高比例和较小站点的较低比例。

While all the pages in the candidate set contain at least one relative stylesheet path, Figure 3 shows that 63.1 % of them contain multiple relative paths, which increases the chances of finding a successful RPO and style injection point.

虽然候选集中的所有页面至少包含一个相对样式表路径，但是图3显示63.1%的页面包含多个相对路径，这增加了找到成功的RPO和样式注入点的机会。

4.2 Vulnerable Pages

We consider a candidate page vulnerable if one of the style injection techniques of Section 3.2 succeeds. In other words, the server's response should 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.

攻击成功标准

Table 2 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 2, 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.

？没太懂

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

！！！！
服务器响应中的基标签可以防止路径混淆，因为它指示浏览器应该如何扩展相对路径。

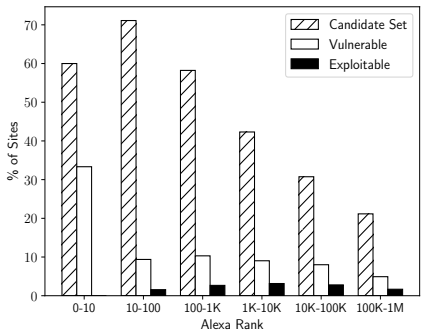

Figure 2: Percentage of the Alexa site ranking in our candidate set (exponentially increasing bucket size).

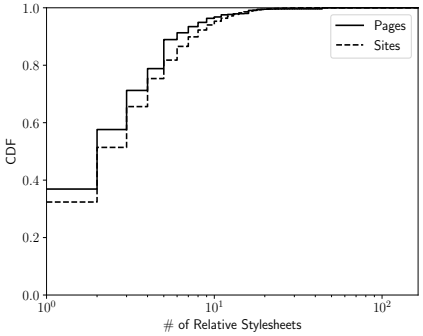

Figure 3: CDF of total and maximum number of relative stylesheets per web page and site, respectively.

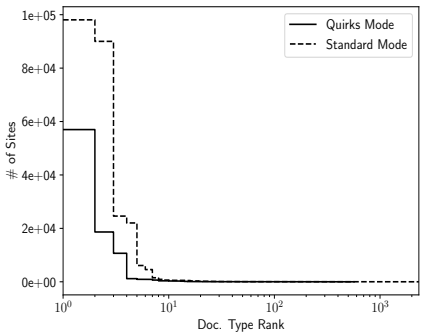

Figure 4: Number of sites containing at least one page with a certain document type (ordered by doctype rank).

Table 2: Vulnerable/exploitable pages and sites in the candidate set (IE using framing).

Technique	Vulnerable		Exploitable (Chrome)		Exploitable (Internet Explorer)	
	Pages	Sites	Pages	Sites	Pages	Sites
Path Parameter	309,079 (1.0%)	9,136 (4.1%)	6,048 (<0.1%)	1,025 (0.5%)	52,344 (0.2%)	3,433 (1.5%)
Encoded Path	53,502 (0.2%)	1,802 (0.8%)	3 (<0.1%)	2 (<0.1%)	24 (<0.1%)	5 (<0.1%)
Encoded Query	89,757 (0.3%)	1,303 (0.6%)	23 (<0.1%)	20 (<0.1%)	137 (<0.1%)	43 (<0.1%)
Cookie	15,656 (<0.1%)	1,030 (0.5%)	4,722 (<0.1%)	81 (<0.1%)	2,447 (<0.1%)	238 (0.1%)
Total	377,043 (1.2%)	11,986 (5.4%)	10,781 (<0.1%)	1,106 (0.5%)	54,853 (0.2%)	3,645 (1.6%)

Table 3: 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: 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 %)

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

<base> tag 在IE表现得有点异常

4.3 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. This test succeeded for 2.9 % of vulnerable pages; 0.5 % of sites in the candidate set had at least one exploitable page (Table 2).

In the following, 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.

为了测试一个脆弱的页面是否可以被利用，我们在Chrome中打开它，
注入一个带有图像引用的样式负载(随机生成的URL)，并检查图像是否确实被加载

4.3.1 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.

都是啥类型？
怎么这么多？

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 3.2. 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

Table 5: 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%)

injected CSS to be parsed and the injected background image to be downloaded.

Table 3 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 shows the most frequently used document types that force all the browsers into quirks mode, 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 3.3. 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 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 5), which is one of the preconditions for successful RPO.

4.3.2 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 2 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 2). 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.

4.3.3 Anti-Framing Techniques. Some sites use a range of techniques to prevent other pages from loading them in a frame [42]. 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.

Furthermore, developers may use JavaScript code to prevent framing of a page. Yet, techniques exist to bypass this protection [38]. 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.

4.3.4 MIME Sniffing. A consequence of self-reference in the type of RPO studied in this paper 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` [4, 21, 33].

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 4.3.1. For each browser in Table 3, 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.

在分析数据集中可利用的页面时，我们注意到许多页面似乎属于知名的CMSes。
由于这些web应用程序通常安装在数千个站点上，
因此修复这些应用程序中的RPO弱点可能会产生很大的影响。

4.4 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 [51]. 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 online 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.

不是很懂？

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 [8, 28, 47]. 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.

除了实验中用到的数据以外，Wappalyzer还在Internet上检测到这些CMS有将近32K的安装，这表明有更多的站点可能会存在RPO漏洞。

5 MITIGATION TECHNIQUES

根本原因和浏览器对于URL解析存在差异导致的

Relative path overwrites rely on the web server and the web browser interpreting URLs differently. 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. 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.

防御方式1

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 paper, 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) [48].

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.

6 CONCLUSION

This paper 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.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation (NSF) under grant CNS-1703454 award, and Secure Business Austria.

REFERENCES

[1] 2017. Chrome Remote Debugging Protocol. https://chromedevtools.github.io/devtools-protocol/. (2017).

[2] Steven Van Acker, Nick Nikiforakis, Lieven Desmet, Wouter Joosen, and Frank Piessens. 2012. FlashOver: Automated Discovery of Cross-site Scripting Vulnerabilities in Rich Internet Applications. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*.

[3] Alexa. 2016. Top Sites. http://www.alexa.com/topsites. (2016).

[4] Adam Barth, Juan Caballero, and Dawn Song. 2009. Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves. In *IEEE Symposium on Security and Privacy (S&P)*.

[5] Daniel Bates, Adam Barth, and Collin Jackson. 2010. Regular Expressions Considered Harmful in Client-Side XSS Filters. In *International World Wide Web Conference (WWW)*.

[6] Prithvi Bisht and V. N. Venkatakrishnan. 2008. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.

[7] Burp Suite. 2017. https://portswigger.net/burp/. (2017).

[8] Orcun Cetin, Carlos Ganan, Maciej Korczynski, and Michel van Eeten. 2017. 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)*.

[9] Common Crawl. 2016. https://commoncrawl.org/. (August 2016).

[10] Soroush Dalili. 2015. 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).

[11] Adam Doupe, Weidong Cui, Mariusz H. Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. 2013. deDacota: Toward Preventing Server-Side XSS via Automatic Code and Data Separation. In *ACM Conference on Computer and Communications Security (CCS)*.

[12] Omer Gil. 2017. Web Cache Deception Attack. In *Black Hat USA*.

[13] Omer Gil. 2017. Web Cache Deception Attack. http://omergil.blogspot.com/2017/02/web-cache-deception-attack.html. (2017).

[14] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. 2012. Scriptless Attacks - Stealing the Pie Without Touching the Sill. In *ACM Conference on Computer and Communications Security (CCS)*.

[15] Mario Heiderich, Christopher Späth, and Jörg Schwenk. 2017. DOMPurify: Client-Side Protection Against XSS and Markup Injection. In *European Conference on Research in Computer Security (ESORICS)*.

[16] Gareth Heyes. 2009. The Sexy Assassin: Tactical Exploitation using CSS. https://docs.google.com/viewer?url=www.businessinfo.co.uk/labs/talk/The_Sexy_Assassin.ppt. (2009).

[17] Gareth Heyes. 2014. RPO. http://www.thespanner.co.uk/2014/03/21/rpo/. (2014).

[18] Lin-Shung Huang, Zack Weinberg, Chris Evans, and Collin Jackson. 2010. Protecting Browsers from Cross-Origin CSS Attacks. In *ACM Conference on Computer and Communications Security (CCS)*.

[19] Artur Janc and Lukasz Olejnik. 2010. Feasibility and Real-World Implications of Web Browser History Detection. In *Web 2.0 Security and Privacy (W2SP)*.

[20] Christoph Kern. 2014. Securing the Tangled Web. *Commun. ACM* 57, no. 9 (2014), 38–47.

[21] Christoph Kerschbaumer. 2016. Mitigating MIME Confusion Attacks in Firefox. https://blog.mozilla.org/security/2016/08/26/mitigating-mime-confusion-attacks-in-firefox/. (2016).

[22] James Kettle. 2015. Detecting and Exploiting Path-Relative Stylesheet Import (PRSSI) Vulnerabilities. http://blog.portswigger.net/2015/02/prssi.html. (2015).

[23] Masato Kinugawa. 2015. CSS based Attack: Abusing Unicode-Range of @font-face. http://mksben.l0.cm/2015/10/css-based-attack-abusing-unicode-range.html. (2015).

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

[25] Sebastian Lekies, Krzysztof Kotowicz, Samuel Grob, Eduardo A. Vela Nava, and Martin Johns. 2017. Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets. In *ACM Conference on Computer and Communications Security (CCS)*.

[26] Sebastian Lekies, Krzysztof Kotowicz, and Eduardo Vela Nava. 2017. Breaking XSS mitigations via Script Gadgets. In *Black Hat USA*.

[27] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 Million Flows Later - Large-scale Detection of DOM-based XSS. In *ACM Conference on Computer and Communications Security (CCS)*.

[28] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Michael Bailey, Damon McCoy, Stefan Savage, and Vern Paxson. 2016. You've Got Vulnerability: Exploring Effective Vulnerability Notifications. In *USENIX Security Symposium*.

[29] Bin Liang, Wei You, Liangkun Liu, Wenchang Shi, and Mario Heiderich. 2014. Scriptless Timing Attacks on Web Browser Privacy. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[30] Nera W. C. Liu and Albert Yu. 2014. Ultimate DOM Based XSS Detection Scanner On Cloud. In *Black Hat Asia*.

[31] Mike Ter Louw and V.N. Venkatakrishnan. 2009. BLUEPRINT: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *IEEE Symposium on Security and Privacy (S&P)*.

[32] Giorgio Maone. 2009. NoScript. https://noscript.net/. (2009).

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

[34] Microsoft. 2015. Understanding the Compatibility View List. https://msdn.microsoft.com/en-us/library/gg699485(v=vs.85).aspx. (2015).

[35] Yacin Nadji, Prateek Saxena, and Dawn Song. 2009. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Network and Distributed System Security Symposium (NDSS)*.

[36] Terri Oda, Glenn Wurster, P. C. van Oorschot, and Anil Somayaji. 2008. SOMA: Mutual Approval for Included Content in Web Pages. In *ACM Conference on Computer and Communications Security (CCS)*.

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

[38] OWASP. 2017. Clickjacking Defense Cheat Sheet. https://www.owasp.org/index.php/Clickjacking_Defense_Cheat_Sheet. (2017).

[39] OWASP. 2017. Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet. (2017).

[40] OWASP. 2017. XSS (Cross Site Scripting) Prevention Cheat Sheet. https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet. (2017).

[41] David Ross. 2008. IE 8 XSS Filter Architecture / Implementation. https://blogs.technet.microsoft.com/srd/2008/08/19/ie-8-xss-filter-architecture-implementation/. (2008).

[42] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. 2010. Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites. In *IEEE Oakland Web 2.0 Security and Privacy (W2SP)*.

[43] Mike Samuel, Prateek Saxena, and Dawn Song. 2011. Context-Sensitive Auto-Sanitization in Web Templating Languages Using Type Qualifiers. In *ACM Conference on Computer and Communications Security (CCS)*.

[44] Henri Sivonen. 2013. Activating Browser Modes with Doctype. https://hsivonen.fi/doctype/. (2013).

[45] Sid Stamm, Brandon Sterne, and Gervase Markham. 2010. Reining in the Web with Content Security Policy. In *International World Wide Web Conference (WWW)*.

[46] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. 2014. Precise Client-side Protection against DOM-based Cross-Site Scripting. In *USENIX Security Symposium*.

[47] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. 2016. Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification. In *USENIX Security Symposium*.

[48] Takeshi Terada. 2015. A Few RPO Exploitation Techniques. https://www.mbsd.jp/Whitepaper/rpo.pdf. (2015).

[49] W3C. 2011. CSS Syntax and Basic Data Types. http://www.w3.org/TR/CSS2/syndata.html. (2011).

[50] W3C. 2015. Content Security Policy Level 2. https://www.w3.org/TR/CSP2/. (2015).

[51] Wappalyzer. 2017. Identify technologies on websites. https://www.wappalyzer.com/. (2017).

[52] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *ACM Conference on Computer and Communications Security (CCS)*.

[53] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. 2011. An Empirical Analysis of XSS Sanitization in Web Application Frameworks. In *European Conference on Research in Computer Security (ESORICS)*.

[54] XSS Jigsaw. 2015. CSS: Cascading Style Scripting. http://blog.innerht.ml/cascading-style-scripting/. (2015).

[55] XSS Jigsaw. 2016. RPO Gadgets. http://blog.innerht.ml/rpo-gadgets/. (2016).