Protecting Browsers from Cross-Origin CSS Attacks

Chris Evans
Google
cevans@google.com

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

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

Adam Barth
UC Berkeley
abarth@eecs.berkeley.edu

ABSTRACT

The cross-origin CSS attack is a newly exploited browser vulnerability that allows the attacker to steal the user's secret content on an honest web site from a different domain. In this paper, we expose the cross-origin CSS attack, in which the attacker injects CSS strings into the target domain that contains the secret content, instructs the user's browser to import and parse the document as a style sheet, and then exports the secret string hidden in the imported style sheet to the attacker. This attack is constructed upon the lenient checking of content types and lax parsing of CSS in modern browsers. We detail existing server-side and client-side defense techniques and find shortcomings with each technique. We propose that browsers implement a stricter CSS parsing when the content type is broken, which can provide robust defense for cross-origin CSS attacks while addressing web site compatibility concerns.

Categories and Subject Descriptors

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

General Terms

Security

Keywords

Content-Sniffing, CSS, Same-Origin Policy

1. INTRODUCTION

(This is the introduction.)

2. THREAT MODEL

We define a precise threat model, the web attacker, for reasoning the cross-origin CSS attack. We assume that the attacker's goal is to steal the user's secret content on an honest web site from another domain.

Web Attacker (or Data Theft?)

A web attacker is a malicious principal who owns a domain name, e.g. attacker.com, and operates a web server. To study the browser's defense mechanisms, we assume that the user's browser renders content from the attacker's web site. In addition, the web attacker is given the ability to inject arbitrary strings into the target domain.

- Attacker Abilities: The web attacker can send and receive network messages over network protocols of the attacker's choice, but only from its own machines. Typically, the web attacker operates at least one machine as an HTTP server, which we refer to as attacker.com. The web attacker has no control over the user's network connections. In particular, the web attacker cannot eavesdrop on unencrypted connections and steal session identifiers to hijack the user's session. Neither can the web attacker spoof network frames and impersonate the user to an honest web site.
- User Behavior: We assume that the user visits the attacker's web site with a popular browser. In practice, this assumption can be supported by several techniques for attacting users, e.g. buying web advertisements or sending bulk e-mail to encourage visitors. By visiting attacker.com, the attacker can instruct the user's browser to fetch external style sheets and also export data to remote servers. As long as the user's session with an honest web site haven't expired, the attacker can instruct the user's browser to issue requests with the user's cookies.
- Web Site Behavior: We assume that the attacker can inject arbitrary strings into a target domain that contains the secret content, which we refer to as victim.com. In practice, there are various methods to inject strings into the target domain, e.g. reflection of user controlled strings or via URL parameters. The ability of injecting CSS strings is considered weaker than injecting scripts into a target domain as in common XSS attacks, because scripts may be detected and blocked by client-side XSS filters.

There are a number of related threats that we do not consider in this paper, such as phishing, DNS rebinding, and XSS. (more description...?) The web attacker is incapable of installing malicious software on the user's machine, otherwise, the malmare could replace the browser and bypass any browser-based defenses.

3. ATTACKS

In this section, we study cross-origin CSS attacks. First, we provide some background information. Next, we introduce cross-origin CSS attacks. Then, we discuss the restrictions for constructing this attack and suggest general areas that are exploitable. Finally, we demonstrate an attack against a popular web site as our proof of concept.

3.1 Background

In this section, we provide background information about Cascading Style Sheets (CSS), the same-origin policy (SOP), and content sniffing.

3.1.1 Cascading Style Sheets

Cascading Syle Sheets[4] is a mechanism that lets web developers control the visual appearance of web documents using style sheet language. The design of CSS enables separation of document content from the visual appearance, including layouts, colors, and fonts. This allows web sites to adjusting its appearance, e.g. switching on-screen views to printable views, without requiring modifications to the document. Furthermore, documents are allowed to import external style sheets and let third parties to control a subset of the CSS rules. This feature allows web sites to share style sheets across a number of documents.

Lax Parsing. In modern browsers, the CSS parser is actually very lax. The CSS parser will skip over any amount of invalid syntax in the style sheet until it finds the next valid rule. This is unlike the JavaScript parser that will abort on the first syntax error. A lax CSS parser gives the browser better compatiblity for supporting misconfigured web sites, but unfortunately opens opportunities to attackers and enables the cross-origin CSS attack in this paper.

3.1.2 Same-Origin Policy

The same-origin policy[3] is the main security mechanism in modern browsers that provides isolation of contents between unrelated web sites. The same-origin policy restricts contents to access resources only from the same origin, which applies to scripting interactions, i.e. XMLHttpRequest and Document Object Model (DOM)[5] access. In SOP, the origin of a resource is defined as the protocol, host, and port. The same-origin policy does not apply to fetching and executing remote libraries, including scripts and style sheets, from a different origin.

3.1.3 Content-Sniffing

The HTTP Content-Type header indicates the type of the content that is transmitted using Multipurpose Internet Mail Extensions (MIME)[2] types such as text/html or image/jpeg. Browsers use this MIME type to determine how to handle the contents of HTTP responses. Some web servers fail to provide the correct MIME type in the Content-Type header, therefore browsers use content-sniffing algorithms to guess the correct MIME type by inspecting the contents of HTTP responses and override the server's MIME type. (mention meta tag?)

3.2 Cross-Origin CSS Attack

In a cross-origin CSS attack, the attacker's main objective is to steal the user's secret content on an honest web site from a different domain. The attacker is most likely interested in stealing any sensitive information in a cookie-authenticated web page, or preferrably a secret CSRF token hidden in the document that may allow the attacker to hijack the user's session. (reference for csrf token?) To describe the cross-origin CSS attack, we decompose the attack into three main steps: string injection, style sheet import, and data export.

3.2.1 String Injection

In order to steal content from a different domain, the attacker leverages the fact that browsers allows web pages to import style sheets from cross-origin. Given the ability to inject strings into the target domain which contains the secret content, the attacker may fool the user's browser into loading a carefully crafted non-CSS (e.g. HTML or XML) document as a valid style sheet resource. In practice, there are several methods that may allow a web attacker to inject strings into the target domain, e.g. reflection of user controlled strings or via URL parameters. Normally, random text in a non-CSS document contains syntax errors to the CSS parser and would break the parsing. However, due to the overly lax CSS parsers in modern browsers, portions of a non-CSS document with valid CSS syntax can still be successfully parsed and recognized as style sheet rules.

CSS Properties. To describe the construction of a string injection, suppose that there is s sensitive content, represented with the string "SECRET", in an HTML document on an honest web site that the attacker wants to steal. If the attacker has sufficient influence over the web page to control the text preceding and succeeding the secret string, we can construct the attacker's injection string based on common CSS properties that corresponds to string values, i.e. font-family, background-image, and list-style-image. For example, the HTML document can be crafted by injecting a CSS font-family string as the following:

```
Original:
<HTML>..SECRET..</HTML>

After injection:
<HTML>..{} BODY { font-family: "SECRET" }..</HTML>
```

The crafted HTML document will appear to the CSS parser as containing a valid CSS rule. The fonts of the attacker's page will be styled with a font-family specified as the stolen string "SECRET". Note that the seemingly redundant pair of brackets in the injection string resyncs the CSS parser to make sure that the evil CSS rule parses properly. All the other non-CSS text in the document are skipped by the CSS parser, thus will not effect the user's browser parsing of the evil CSS rule. Once the crafted document is loaded, the attacker can easily steal the secret string from the font-family style of the attacker's page.

3.2.2 Style Sheet Import

When the user visits attacker.com, the next step for the attacker is to instruct the user's browser to fetch and load

the crafted document as a style sheet. In HTML specifications[6], documents can import external style sheets from remote servers by using the CSS "@import" notation, or using the "LINK" element inside the "HEAD" element as the following: .

```
<HTML>
<HEAD>
  <STYLE TYPE="text/css">
    @import url(http://victim.com);
  </STYLE>
  <LINK REL="stylesheet" HREF="http://victim.com">
</HEAD>
</HTML>
```

Cookies. According to the standards, the browser always sends the user's cookie on any load of CSS, including cross-origin. This behavior exploits cookie-authenticated web sites and potentially leaks the user's sensitive information in the server's response.

3.2.3 Data Export

The final step for the attacker is to access the loaded style sheet and export the secret content hidden in the style properties to the attacker. There are various methods to export data to other origins, in this specific attack we are interested in approaches where no user involvement is required. The cross-origin CSS attack is possible by just rendering one evil ad in the user's browser.

JavaScript. One method is using JavaScript to process the data from loaded styles and then send AJAX requests to attacker.com. All major browsers support reading styles in JavaScript, i.e. through DOM access by calling the window.getComputedStyle method, or retrieving the currentStyle object in Internet Explorer. Some browsers (Chrome and Opera) cautiously block DOM access to styles loaded from cross-origin unless the CSS is "well-formed". Other browsers (Internet Explorer, Firefox and Safari) still grant read access to the styles loaded from cross-origin CSS. One of the WebKit-based browsers (Safari) even surrendered the full CSS text of style sheets, including CSS rules loaded from cross-origin (without comments, thankfully). For the Safari browser, the attacker could read the raw text of cross-origin CSS via document.styleSheets[].cssRules[] or window.getMatchedCSSRules(). This behavior violates the same-origin policy and potentially leaks data in pages with semi-valid CSS constructs. Some browsers only permit read access for cross-origin CSS text under restricted conditions. Internet Explorer allows access to cross-origin CSS raw text with document.styleSheets[].rules[] only if the file extension is correct. Chrome and Opera allows access to cross-origin CSS raw text with document.styleSheets[].rules[] only if the external file is well-formed. Chrome also allows access to cross-origin CSS raw text with window.getMatchedCSSRules() if the external file is well-formed.

(draw a table?)

Background-image. Another method is available to exploit users that have disabled JavaScript in their browsers by simply injecting the CSS background-image string as the following:

```
<HTML>..{} BODY { background-image: url(http://att
acker.com/?SECRET); }..</HTML>
```

Using this method, the secret string is appended to the path or query string of the attacker's server URL. Therefore, the background of the attacker's page will be styled with a background image loaded from an URL, the path of which contains stolen data. One important characteristic of the CSS property background-image, which is an URL, is that it will be automatically fetched even if JavaScript is turned off. The stolen data is then harvested by the attacker from their web server logs.

3.3 General Restrictions

The cross-origin CSS attack is less serious than it could be, because there are restrictions of cross-origin data which can be stolen.

- Newlines: Any newline in the stolen string will break CSS parsing.
- Character Escapes: single/double quote, paranthesis, semicolon url(..)
- Injection Points: Need 2 injection points

Exploitation Areas. rich-functionality sites cookie-authenticated URLs e.g. AJAX APIs, JSON responses, XML responses – and in particular – mobile HTML UIs.

IE. fully-patched IE allows newlines allows single/double quotes

UTF-7. UTF7 charset (FF, Safari) header /meta tag http-equiv

3.4 Concrete Attack

Example: Yahoo! Mail

4. DEFENSES

4.1 Server-side Defenses

4.1.1 Newlines

Newlines;

4.1.2 Escaping Quotes

Escaping quotes;

4.1.3 Nosniff

Nosniff Content type header (charset) burden on developers

4.2 Client-side Defenses

4.2.1 Disable Cookies

Prevent sending cookies when importing cross-origin CSS; login status check adjust mobile account, printable view..

4.2.2 Disable Content-Sniffing

Require valid MIME for CSS; if not globally at least for cross-origin loads

4.2.3 DOM SOP

DOM same-origin policy;

4.2.4 Strict CSS Parsing

Strict CSS parsing; ie strict mode??

4.2.5 Strict CSS Parsing for Broken MIME

Proposal: For cross-origin CSS loading, stricter CSS parsing for Broken MIME; -Conservative approach; -Blocks most attacks; Implementation; -Webkit patch (chrome OK, safari 4.0.5)

5. RELATED WORK

5.1 Opera

5.2 Secure Content Sniffing

secure content sniffing[1]

6. CONCLUSIONS

(This is the conclusion.)

7. REFERENCES

[1] 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 IEEE Symposium on Security and Privacy*, May 2009.

[2] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. http://tools.ietf.org/html/rfc2045.

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

[4] W3C. Cascading Style Sheets. http://www.w3.org/Style/CSS/.

[5] W3C. Document Object Model. http://www.w3.org/DOM/.

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