Redefining Impossible: XSS without arbitrary JavaScript

Redefining Impossible: XSS without arbitrary JavaScript - Luan Herrera, PortSwigger Research.

- Published: 2020-09-23
- Original: <https://portswigger.net/research/redefining-impossible-xss-without-arbitrary-javascript>
- Preserved from: https://portswigger.net/research/redefining-impossible-xss-without-arbitrary-javascript (stored) on 2026-08-11
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Redefining Impossible: XSS without arbitrary JavaScript | PortSwigger Research

Redefining Impossible: XSS without arbitrary JavaScript

[image: Luan Herrera]

Luan Herrera

Researcher

@lbherrera_

-

**Published: **Wednesday, 23 September 2020 at 13:01 UTC

-

**Updated: **Wednesday, 20 July 2022 at 09:16 UTC

-

We recently updated our impossible XSS labs series with a new challenge. For this scenario your injection occurs within a single quoted JavaScript string and you can only use the charset `a-zA-Z0-9'+.` `

`<script> x = 'injection'; </script>`

The objective was to steal a cookie and send it to a remote server. One month after posting this challenge, we received an amazing and imaginative solution from Luan Herrera. We asked if he'd like to write it up as a guest post on our site and he agreed. Over to you Luan.

Breaking it down

My initial approach was to define the problem clearly. Understand the challenge's restrictions, and from there, work towards a solution.

Restrictions

-

The input was being reflected inside a single quoted string.

-

The charset was restricted to `a-zA-Z0-9'+.` `.

Takeaways

-

Given the injection context, in order to escape the quoted string it is necessary to make use of the ' and + characters.

-

Because of the + operator, whatever payload was used after escaping the single quoted string would be concatenated with two other strings, and then assigned to the variable x — which meant that `toString()` would always be automatically called: `x = '' + injection + '';`

-

It was possible to call functions through Tagged templates — this was achieved by using the backtick character, albeit there were restrictions:

-

It's only possible to pass one argument to the function.

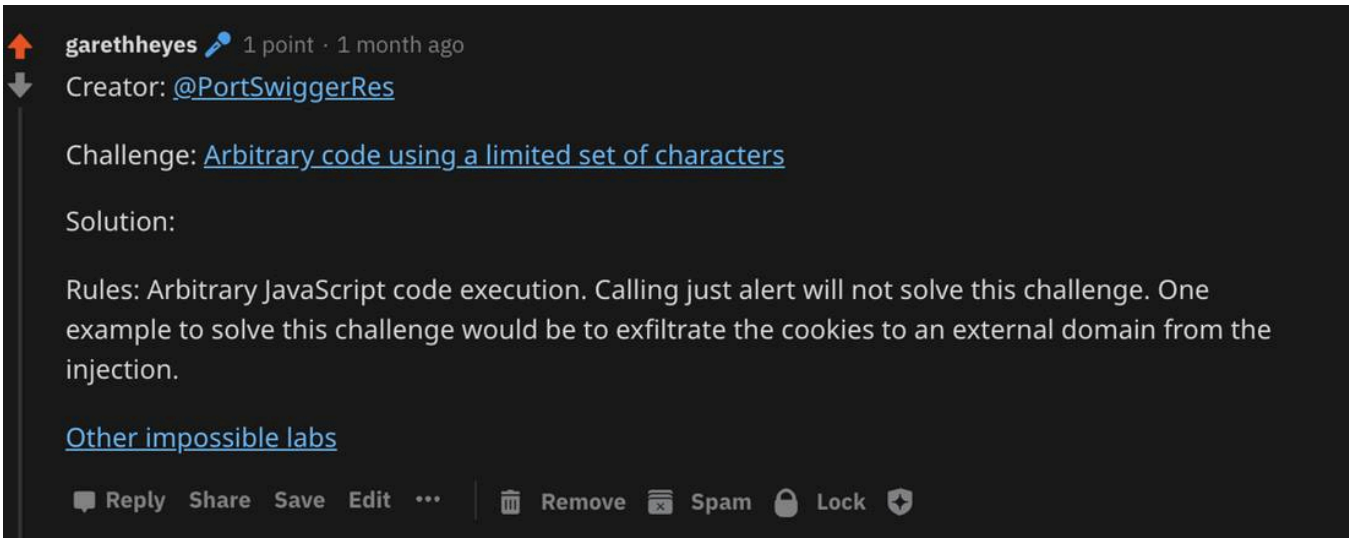
-

It's only possible to pass a literal string as the argument, which prevented passing the value of variables to functions, as the ${} characters were not allowed.

After frustratingly long hours diving deeper into Javascript's native functions, trying to chain functions in a way that allowed me to pass arbitrary variables to Tagged templates, or even trying to find a clever way to make assignments, I wasn't any the wiser and concluded this was indeed an impossible challenge =(

New perspective

Weeks later, while reading a thread at /r/Slackers subreddit looking whether new challenges had been released, I noticed Gareth's message:



Perhaps achieving arbitrary code execution wasn't expressly necessary, if I could leak the data some other way it might prove to be sufficient.

Side-channel attacks

With a new approach in mind, I decided to revisit the challenge. What other means were there to exfiltrate cross-origin content under a restrictive charset? I immediately thought of side-channel attacks.

I considered whether it was possible to construct some sort of oracle using only the allowed charset, and then use one of the many side-channel leaking techniques to exfiltrate said information.

With the permitted charset I had access to the references of any window property (document.cookie, DOM contents), and through string manipulation I had the ability to inspect one character at a time.

`// document.cookie = "secret=1337"; document.cookie.charAt 0 // "s" document.cookie.charCodeAt 0 // 115`

The Oracle

The next step was to create some comparison mechanism in which the behaviour responded one way if the “selected” char equated to the char being tested against it, and in a different way if it didn’t. This divergence in behaviour would have to be detectable by one of the side-channel leaking techniques.

`String.prototype.split()` combined with `String.prototype.repeat()` turned out to be a good candidate for the oracle-like behaviour.

By repeating the targeted char before splitting it, it was possible to arbitrarily inflate the execution time difference between a matched char vs one that didn’t.

In the example below, when the selected char matched the one being targeted, the execution time was much higher than when it didn’t.

`let init = performance.now(); "s".repeat(40000000).split("s"); // (40000001) ["", "", "", "", ...] console.log(performance.now() - init); // 1098.244999999224

let init = performance.now(); "s".repeat(40000000).split("a"); // ["ssssss...ssssss"] console.log(performance.now() - init); // 31.080000000656582`

This higher execution time was also reflected in the time that it took for the page to fully load and for the load event of an iframe to trigger, for example — and naturally, this time difference could be leveraged to exfiltrate information from the page.

There was one caveat: because of the limited charset, not all characters could be explicitly passed to the split function — this required a small tweak in the way the chars were being represented.

For instance, the “>” char could not be passed directly as a parameter to the split function because it wasn’t a valid char of the given charset.To solve this, each character's char code was converted to a string.

This allowed me to compare any character stored within the window’s properties using only chars in the permitted charset. I chose hex as the base because all ASCII characters can be represented as two bytes, ranging from 00 to FF.

`// document.cookie = "secret=1337"; document.cookie.charCodeAt 0 // 115 .toString 16 // "73" .charAt 0` // "7"

`document.cookie.charCodeAt 0 // 115 .toString 16 // "73" .charAt 1 // "3"`

The timing-based proof of concept tests all characters of the challenge’s `document.cookie` one by one; each character, converted into a hex string, has its bytes resolved individually by testing if they match values from 0 to F. Leveraging the higher execution time of matching queries leads to detectable leaks.

The timing-based proof of concept can be found here.

Improving the exploit

Though the exploit worked, I felt like there was room to improve its efficiency; time-based side-channel exfiltrations are known to be susceptible to network latency. Also, because it was necessary to hang the page long enough when the chars matched (for detection purposes), the exploit ended up being slow.

I revisited the oracle and tried to rewrite the exploit to remove the time-based aspect of the exfiltration. One realisation was that instead of `split()` and `repeat()`, I could use `String.prototype.match()` to match the characters being tested against the target one by one.

The trick was that when there was a match, an Array containing match-related content was returned, otherwise it returned null. The second leap was in recognising that Arrays had access to the `toString()` method and that null didn't.

`// document.cookie = "secret=1337"; document.cookie.charCodeAt 0 // returns 115 .toString 16 // converts 115 to "73", hexadecimal base .match 73 // returns a match ["73", index: 0, input: "73", groups: undefined] .toString` // evaluates the earlier match to "73"

// document.cookie = "secret=1337"; document.cookie.charCodeAt `0` // returns 115 .toString `16` // converts 115 to "73", hexadecimal base .match `74` // returns null .toString` `// VM528:2 Uncaught TypeError: Cannot read property 'toString' of null at <anonymous>:2:54`

As demonstrated, invoking `Object.prototype.toString()` will either:

-

If the char matched, return the char tested against itself.

-

If that char didn't match, throw an exception as the method doesn't exist.

This difference in behaviour allowed me to make boolean queries over any content of the window properties, as further exemplified below:

"Is `document.cookie[0]` equal to 's'?"

-

If yes, the expression returns 's' itself.

-

If not, the script triggers an exception by trying to use `toString()` on a null.

There only remained the issue of exfiltration. With the earlier oracle it was possible to arbitrarily inflate the execution time of a successful query and measure it with `performance.now()`. That was no longer the case.

The initial detection idea I had was that if the chars matched, `toString()` wouldn't throw an exception and a redirect would occur, if the chars didn't match, calling `toString()` on a null would lead to an error and location.assign would never be reached, therefore no redirect.

`x = '' + 's'.match s .toString + location.assign 1 + '';`

Since I was loading the challenge's page inside an iframe I thought I would be able to detect whether the redirect happened by listening to the iframe's load event and counting the number of times it was triggered (a known XS-Leak technique).

Assumption [if there was a match]:

-

Load event triggers when the challenge's page is loaded in the iframe.

-

Match occurs, script doesn't crash, location.assign redirects to /1.

-

Load event is triggered a second time when the redirected page fully loads.

-

The two loads are detected by listening to the iframe's load event.

On the same note, if the script had thrown on `toString()`, `location.assign` would never be reached and neither would the redirect to /1 happen. In this case, the load event ends up being triggered only a single time.

`[...]`

Unfortunately this wasn't quite what happened; I failed to take into account that the load event was only triggered when the page was fully loaded.

One of two things happened instead:

-

If there was a match, the script didn't crash, the redirect happened before the page was fully loaded; when the redirected page fully loaded the load event was triggered.

-

If there wasn't a match, the script crashed, the challenge's page finished loading and the load event was triggered.

In both instances the load event was triggered only once, preventing me from using it as a side-channel to leak the oracle's result.

Stop!

One obscure method of window is `stop()` which "stops further resource loading in the current browsing context, equivalent to the stop button in the browser".

If it was possible to prevent the page from fully loading, then it followed that the load event wouldn't be triggered; and that's exactly what happened!

Because any script in the same script block in which `stop()` is called will still be executed, it proved possible to incorporate it into the injected oracle.

`<script> x = '' + stop + s .match s .toString + location.assign 1 + ''; </script>`

Unlike the earlier case where the load event was triggered once regardless of whether there was a match or not, with `stop()` what happened instead was that it prevented further resources from loading past the current script block - stopping the iframe's load event from triggering.

This was helpful because:

If there was not a match:

-

The script crashed, the redirect never happened.

-

`stop()` prevented the page from loading, the load event of that iframe was never triggered.

If there was a match:

-

The script didn't crash, the redirect still happened (since it was triggered within the same script block).

-

The load event triggered upon load completion of the redirected page.

Through this we can both infer information much more reliably and leak which are all the characters in any given window property (e.g document.cookie, DOM contents, etc).

Summary

The technique can be summarised by the following points:

-

We can sequentially test all characters from a given window property by accessing it with `charCodeAt(i)`;

-

A redirect happens when the "selected" char matches against a char from a window property since `location.assign()` will be called — a load event will be triggered due to that;

-

A load event won't be triggered when the "selected" char doesn't match against a char from a window property due to `stop()` being called and preventing the page from loading;

-

We can observe this difference in behaviour by detecting if or when the iframe's load event was triggered cross-origin;

That's the concept for a working (and more efficient) exploit! All that was left was to implement the proof of concept with the techniques discussed here, and automating the same process for each character.

You can find the final load-based proof of concept here.