CSP in the Age of Script Gadgets

Martin Johns
m.johns@tu-braunschweig.de
SecAppDev 2019
Me, myself and I

• Prof. Dr. Martin Johns
  • TU Braunschweig, Institute for Application Security (IAS)
  • Since April 2018

• Before joining the wonderful world of academia (2009 - 2018)
  • 9 years at SAP Security Research, Germany
  • Lead for application and web security research

• PhD on Web Security at University of Passau (2004 - 2009)

• Tons of development jobs during the Web 2.0 times (1998 - 2003)
Very brief recall: Cross-site Scripting (XSS)

• XSS is a class of code injection vulnerabilities in web applications

• The attacker can inject HTML/JS into an vulnerable application

• This JS is executed in the browser of the attack’s victim
  • This runs under the victim’s authentication context
  • and has all capabilities that the user himself has
    • Full read access to protected content
    • Creating further (authenticated) HTTP requests and reading responses
    • Forging and interacting with UI elements

• —> Full client-side compromise
The three major causes for XSS
The three major causes for XSS

- Injection of inline script
  - Attacker directly injects complete inline script tags or injects into dynamically created inline scripts

```html
<script>alert('peng');</script>
```
The three major causes for XSS

- **Injection of inline script**
  - Attacker directly injects complete inline script tags or injects into dynamically created inline scripts
    
    ```html
    <script>alert('peng');</script>
    ```

- **Injection of script-tags referencing attacker controlled endpoints**
  
  ```html
  <script src="http://attackr.org"></script>
  ```
The three major causes for XSS

- Injection of inline script
  - Attacker directly injects complete inline script tags or injects into dynamically created inline scripts
    
    ```html
    <script>alert('peng');</script>
    ```

- Injection of script-tags referencing attacker controlled endpoints
  
  ```html
  <script src="http://attackr.org"></script>
  ```

- Injection into dynamic script code generation
  
  ```javascript
  eval(attackerinput);
  ```
XSS is one of the most prevalent menaces on today’s Web

- XSS is caused by insecure programming

- Insufficiently validated data flows from attacker controlled sources to security sensitive sinks

- Thus, our primary response to the problem are
  - Secure development (avoiding)
  - Security testing (detecting)
XSS is one of the most prevalent menaces on today’s Web

- XSS is caused by insecure programming
- Insufficiently validated data flows from attacker controlled sources to security sensitive sinks
- Thus, our primary response to the problem are
  - Secure development
  - Security testing

Does this work?
Prevalence of XSS

- Survey of the CVE database [STREWS 2014]
Prevalence of XSS

Survey of the CVE database [STREWS 2014]

Number of XSS affecting Gmail webmail fixed per quarter

- Q1
- Q2
- Q3
- Q4

NUMBER OF EXPLOITABLE XSS

Q1
Q2
Q3
Q4

2008
2009
2010
2011
2012
2013
2014
2015

2008
2009
2010
2011
2012
2013
2014
2015

Google
Google has paid out a $3,133.7 bounty to a researcher who identified a cross-site scripting (XSS) vulnerability on the recently launched YouTube Gaming website.

YouTube Gaming, quietly launched by YouTube in late August, provides both live-streamed and on-demand gaming videos. The new service competes with Amazon-owned video game streaming website Twitch.

Ashar Javed, a penetration tester with Hyundai AutoEver Europe whose name is in the security hall of fame of several major companies, claims it only took him two minutes to find a reflected XSS vulnerability in YouTube Gaming’s main search bar.
Observation

So, apparently the existing strategies are not enough...

Didn’t we deal with similar circumstances before?

Recall memory corruption:
- Buffer Overflows and co.
- Similar overwhelming number of problems
- Strategy: Attack mitigation
  - Stack guards, non-executable memory, etc.

How can attack mitigation look for XSS?
Observation

So, apparently the existing strategies are not enough…

Didn’t we deal with similar circumstances before?

Recall memory corruption:
- Buffer Overflows and co.
- Similar overwhelming number of problems
- Strategy: Attack mitigation
  - Stack guards, non-executable memory, etc.

How can attack mitigation look for XSS?

Enter CSP
A short history of the Content Security Policy
A first intro to CSP

• What is CSP?
  • Declarative policy to defend against client-side Web attacks

• Main targets
  • Stopping XSS attacks
  • also: (not relevant for this talk)
    • Stopping of information exfiltration
    • Regulation of framing behaviour
    • (proposed) UI consistency enforcement
CSP: Approach

- Scripts execute in the browser
  - Not all scripts in one page come from the same origin
  - New script content can be created on the fly
  - Client-side execution artefacts are invisible for the server

- Thus, mitigation/protection approaches on the server-side work with incomplete information

- CSP
  - Server sets the policy
  - Browser enforces the policy
  - The policy governs with JavaScripts are legitimate, and thus, are allowed to run
The road to CSP

• CSP is build on top of a legacy of research proposals, e.g., the following

• 2007: Jim et al. proposed BEEP [WWW’07]
  • Relevant concept: Browser-enforced policy to stop illegitimate scripts

• 2008: Oda et al. proposed SOMA [CCS’08]
  • Relevant concept: Whitelisting of external script origins

• 2009: Van Gundy and Chen proposed Noncespaces [NDSS’09]
  • Relevant concept: HTML tags carry randomised information, rendering injection impossible

• 2010: Stamm et al. proposed CSP in a research paper [WWW’10]

• 2012: CSP 1.0 W3C Candidate Recommendation
Content Security Policy (CSP) - Level 1

• CSP Level 1 resides on three main pillars
  1. Disallow inline scripts
     - i.e., strict separation of HTML and JavaScript
  2. Explicitly whitelist resources which are trusted by the developer
  3. Disallow on-the-fly string-to-code transformation
     - i.e., forbid eval and aliases

• Text-based policy
  
  ```default-src 'self';
  
  Content-Security-Policy: default-src 'self';
  ```

• CSP is delivered as HTTP header or in meta element in page
CSP - Level 1

• CSP relies on strict separation of HTML and other content
  • This means JavaScript, CSS etc should be loaded via external resources

• For external resources, CSP is structured around directives

• Each directive specifies which content is legal for the respective resource class
  • E.g., script-src, style-src, img-src, font-src, object-src, frame-src, …

• The directive itself is a whitelist
  • i.e, a list of web origins that are permitted to provide said resource
CSP - Directives

- **default-src 'self' | https://* | https://*.example.org | 'none'**
  - controls default policy, can be overwritten by more specific rules
CSP - Directives

- `default-src 'self' | https://* | https://*.example.org | 'none'`
  - controls default policy, can be overwritten by more specific rules

- `script-src, style-src, img-src, font-src, object-src`
  - control allowed origins for scripts, styles, images, fonts, and objects, respectively
CSP - Directives

- `default-src 'self' | https://* | https://*.example.org | 'none'`
  - controls default policy, can be overwritten by more specific rules

- `script-src, style-src, img-src, font-src, object-src`
  - control allowed origins for scripts, styles, images, fonts, and objects, respectively

- `connect-src`
  - whitelists valid XMLHttpRequests targets
**CSP - Directives**

- `default-src 'self' | https://* | https://*.example.org | 'none'`
  - controls default policy, can be overwritten by more specific rules

- `script-src, style-src, img-src, font-src, object-src`
  - control allowed origins for scripts, styles, images, fonts, and objects, respectively

- `connect-src`
  - whitelists valid XMLHttpRequests targets

- `frame-src`
  - restricts from where frames may be shown in document
CSP - Directives

- `default-src 'self' | https://* | https://*.example.org | 'none'`
  - controls default policy, can be overwritten by more specific rules

- `script-src, style-src, img-src, font-src, object-src`
  - control allowed origins for scripts, styles, images, fonts, and objects, respectively

- `connect-src`
  - whitelists valid XMLHttpRequests targets

- `frame-src`
  - restricts from where frames may be shown in document

- `unsafe-inline, unsafe-eval`
  - do exactly what the names suggest...
CSP - Directives

- `default-src 'self' | https://* | https://*.example.org | 'none'`
  - controls default policy, can be overwritten by more specific rules


- `frame-src`
  - restricts from where frames may be shown in document

- `unsafe-inline, unsafe-eval`
  - do exactly what the names suggest...
Why CSP L1 should work (in theory)
Recall: The three major causes for XSS

- Injection of inline script
  - Attacker directly injects complete inline script tags or injects into dynamically created inline scripts

- Injection of script-tags referencing attacker controlled endpoints

- Injection into dynamic script code generation
Recall: The three major causes for XSS

• Injection of inline script
  • Attacker directly injects complete inline script tags or injects into dynamically created inline scripts

  `<script>alert('peng');</script>`

• Injection of script-tags referencing attacker controlled endpoints

  `<script src="http://attackr.org"></script>`

• Injection into dynamic script code generation

  `eval(attackerinput);`
The power of CSP
The power of CSP

• Let’s take this simple, strong CSP
The power of CSP

- Let’s take this simple, strong CSP

```javascript
default-src 'self';
```
The power of CSP

• Let’s take this simple, strong CSP

```javascript
default-src 'self';
```

• Injection of inline script
  • A strong CSP forbids inline scripts
  • (please note javascript:-URLs are a instance of inline scripts)
The power of CSP

• Let’s take this simple, strong CSP

```plaintext
default-src 'self';
```

• Injection of inline script
  • A strong CSP forbids inline scripts
  • (please note javascript:-URLs are a instance of inline scripts)
The power of CSP

- Let’s take this simple, strong CSP

  ```html
  default-src 'self';
  ```

- Injection of inline script
  - A strong CSP forbids inline scripts
  - *(please note javascript: URLs are a instance of inline scripts)*

- Injection of script-tags referencing attacker controlled endpoints
  - The attacker controlled endpoints are not whitelisted
The power of CSP

• Let’s take this simple, strong CSP

  ```
  default-src 'self';
  ```

• Injection of inline script
  • A strong CSP forbids inline scripts
  • (please note javascript:-URLs are an instance of inline scripts)

• Injection of script-tags referencing attacker controlled endpoints
  • The attacker controlled endpoints are not whitelisted
The power of CSP

- Let’s take this simple, strong CSP
  
  ```
  default-src 'self';
  ```

- Injection of inline script
  - A strong CSP forbids inline scripts
  - (please note javascript:-URLs are a instance of inline scripts)

- Injection of script-tags referencing attacker controlled endpoints
  - The attacker controlled endpoints are not whitelisted

- Injection into dynamic script code generation
  - A strong CSP forbids dynamic script code generation
The power of CSP

- Let’s take this simple, strong CSP

```plaintext
default-src 'self';
```

- Injection of inline script
  - A strong CSP forbids inline scripts
  - (please note javascript:-URLs are a instance of inline scripts)

- Injection of script-tags referencing attacker controlled endpoints
  - The attacker controlled endpoints are not whitelisted

- Injection into dynamic script code generation
  - A strong CSP forbids dynamic script code generation
Why CSP L1 did not work
(in practice)
Prohibitive effort for existing code bases
Prohibitive effort for existing code bases

- The Web is not new. We sit on enormous amounts of existing code
- Only very little of this code is naturally compatible with strong CSPs
- Refactoring this code is prohibitively expensive
  - Special problem here: inline event handlers
- Thus, very (!) slow uptake for existing sites
Prohibitive effort for existing code bases

- The Web is not new. We sit on enormous amounts of existing code
- Only very little of this code is naturally compatible with strong CSPs
- Refactoring this code is prohibitively expensive
  - Special problem here: inline event handlers
- Thus, very (!) slow uptake for existing sites
- Potentially easy fix: unsafe-inline
CSP L1 - Adoption in the Wild

[...] only 20 out of the top 1,000 sites in the world use CSP. [...] Unfortunately, the other 18 sites with CSP do not use its full potential.

Incompatible external dependencies
Incompatible external dependencies

• External scripts are not under the control of a site’s developers or security governance

• Thus, if such an external dependency relies on practices that are incompatible with strong CSPs render the deployment of such policies problematic
Incompatible external dependencies

• External scripts are not under the control of a site’s developers or security governance

• Thus, if such an external dependency relies on practices that are incompatible with strong CSPs render the deployment of such policies problematic

• Potentially easy fix: unsafe-eval
Changing whitelists
Changing whitelists

• Web sites are ever changing
  • New external script providers have to be added to the whitelists

• External scripts may include additional scripts from additional origins
  • Not necessary even known to the hosting site
  • E.g., add resellers

• Thus, whitelists have to be constantly maintained
Changing whitelists

- Web sites are ever changing
  - New external script providers have to be added to the whitelists

- External scripts may include additional scripts from additional origins
  - Not necessary even known to the hosting site
  - E.g., add resellers

- Thus, whitelists have to be constantly maintained

- Potentially easy fix: **wildcards in whitelists**
Overly permissive whitelisted origins

- An attacker is still able to inject arbitrary script tags pointing to whitelisted hosts

- Thus, any script on one of these hosts is free game
  - Just, think about how many scripts reside on, e.g., google.com

- Examples for problematic scripts
  - JavaScript frameworks, such as AngularJS
    - Turn markup into script code
  - JSONP endpoints
Excursion: JSONP Concept

http://google.de

https://mail.google.com
Excursion: JSONP Concept

$.getJSON("https://mail.google.com/userdata.json", function (userdata) {
    // handle userdata here
})
Excursion: JSONP Concept

$.getJSON("https://mail.google.com/userdata.json", function (userdata) {
    // handle userdata here
})
Excursion: JSONP Concept

```
$.getJSON("https://mail.google.com/userdata.json", function (userdata) {
    // handle userdata here
});
```

GET /userdata.json
Excursion: JSONP Concept

$.getJSON("https://mail.google.com/userdata.json", function (userdata) {
  // handle userdata here
})

GET /userdata.json

Hostnames do not match
Excursion: JSONP Concept

```
$.getJSON("https://mail.google.com/userdata.json", function (userdata) {
    // handle userdata here
})

<script>
    function read(userdata) {
        // handle userdata here
    }
</script>

<script src="https://mail.google.com/user.js?cb=read"></script>
```

GET /userdata.json

Hostnames do not match
Excursion: JSONP Concept

$.getJSON("https://mail.google.com/userdata.json", function (userdata) {
  // handle userdata here
});

<script>
function read(userdata) {
  // handle userdata here
}
</script>

<script src="https://mail.google.com/user.js?cb=read"></script>

GET /userdata.json

Hostnames do not match

GET /user.js?cb=read

http://google.de

https://mail.google.com
Excursion: JSONP Concept

$.getJSON("https://mail.google.com/userdata.json", function (userdata) {
    // handle userdata here
})

<script>
function read(userdata) {
    // handle userdata here
}
</script>

<script src="https://mail.google.com/user.js?cb=read"></script>
Excursion: JSONP behind the scenes

• Dynamic server-side creation of JS resources

```php
header('Content-Type: application/javascript');
...
$cb = $_GET['cb'];
echo($cb.'(');
$name = "Name";
$Id = "Id";
$Rank = "Rank";
$name = $name;
$Id = $I;
$Rank = $rank;
echo($cb.'}');
?>"
JSONP endpoints

- JSONP relies on the ability of the includer to execute JavaScript
- Hence, no reason to sanitize the callback parameter
- Arbitrary JS can be passed as cb parameter

```
<script src="/path/jsonp?callback=alert(document.domain)//"/>
</script>

/* API response */
alert(document.domain);//{"var": "data", ...});
## Ineffective CSP Policies [CCS16]

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Total</th>
<th>Report Only</th>
<th>Unsafe Inline</th>
<th>Missing object-src</th>
<th>Wildcard in Whitelist</th>
<th>Unsafe Domain</th>
<th>Trivially Bypassable Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique CSPs</td>
<td>26,011</td>
<td>2,591</td>
<td>21,947</td>
<td>3,131</td>
<td>5,753</td>
<td>19,719</td>
<td>24,637</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9.96%</td>
<td>84.38%</td>
<td>12.04%</td>
<td>22.12%</td>
<td>75.81%</td>
<td>94.72%</td>
</tr>
<tr>
<td>XSS Policies</td>
<td>22,425</td>
<td>0</td>
<td>19,652</td>
<td>2,109</td>
<td>4,816</td>
<td>17,754</td>
<td>21,232</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0%</td>
<td>87.63%</td>
<td>9.4%</td>
<td>21.48%</td>
<td>79.17%</td>
<td>94.68%</td>
</tr>
<tr>
<td>Strict XSS Policies</td>
<td>2,437</td>
<td>0</td>
<td>0</td>
<td>348</td>
<td>0</td>
<td>1,015</td>
<td>1,244</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0%</td>
<td>0%</td>
<td>14.28%</td>
<td>0%</td>
<td>41.65%</td>
<td>51.05%</td>
</tr>
</tbody>
</table>

Table 2: Security analysis of all CSP data sets, broken down by bypass categories

Evolution of CSP

- After the first experience with CSP (and the lacking uptake) the mechanism was extended

- Focus of these adaptions was to address the identified usability and security issues
CSP - Relevant changes from Level 1 to Level 2 (I)

- Identified Problem:
  - Overly permissive whitelisted hosts

- Solution: Whitelist resources with paths

```
script-src example.com/scripts/file.js
```

- Remaining Problems
  - Adds further policy complexity and size creep
  - Paths do not address the problem of fluctuations in the set of included origins
  - Path restriction can be circumvented in case the whitelisted origin has an open redirect
CSP - Relevant changes from Level 1 to Level 2 (II)

• Problem:
  • Costly refactoring of inline scripts

• Solution:
  • Allow script tags with hashes or nonces

• Hashes

  script-src 'sha256-B2yPHKaXnvFWtRChIbabYmUBFZdVfKKXHbWtWidDVF8=

• Nonces

  script-src 'nonce-d90e0153c074f6c3fcf53'
CSP - Level 2 Whitelisting with Hashes

• Problem:
  • Costly refactoring of inline scripts

• Solution:
  • Allow script tags with hashes or nonces

```
script-src 'self' https://cdn.example.org
  'sha256-AzQxy7DeWRF9Yq86adG0xLbz7dgM//hBUno53vYK+U='
```
CSP - Level 2 Whitelisting with Hashes

• Problem:
  • Costly refactoring of inline scripts

• Solution:
  • Allow script tags with hashes or nonces

```html
<script-src 'self' https://cdn.example.org
'sha256-AzQxy7DeWRFE9Yq86adG0xLbz7dgM//hBUno53vYK+U='

<script>
alert('My hash is correct');
</script>
```

SHA256 matches value of CSP header
CSP - Level 2 Whitelisting with Hashes

• Problem:
  • Costly refactoring of inline scripts

• Solution:
  • Allow script tags with hashes or nonces

```html
<script-src 'self' https://cdn.example.org
'sha256-AzQxy7DeWRFE9Yq86adG0xLbz7dgM//hBUn053vYK+U='

<script>
alert('My hash is correct');
</script>

<script>
alert('My hash is correct');
</script>
```

SHA256 matches value of CSP header

SHA256 does not match (whitespaces matter)
CSP - Level 2 Whitelisting with Nonces

• Problem:
  • Costly refactoring of inline scripts

• Solution:
  • Allow script tags with nonces

script-src 'self' https://cdn.example.org
'nonce-d90e0153c074f6c3fcf53'
CSP - Level 2 Whitelisting with Nonces

• Problem:
  • Costly refactoring of inline scripts

• Solution:
  • Allow script tags with nonces

```
<script nonce="d90e0153c074f6c3fcf53">
  alert('I will work just fine');
</script>
```

Script nonce matches CSP header
CSP - Level 2 Whitelisting with Nonces

• Problem:
  • Costly refactoring of inline scripts

• Solution:
  • Allow script tags with nonces

\[
\text{\texttt{script-src 'self' https://cdn.example.org 'nonce-d90e0153c074f6c3f53'}}
\]

\[
\text{\texttt{<script nonce="d90e0153c074f6c3f53"> alert('I will work just fine'); </script>}}
\]

\[
\text{\texttt{<script nonce="randomattacker"> alert('I will not work'); </script>}}
\]

Script nonce matches CSP header

Script nonce does not match CSP header
CSP - Relevant changes from Level 2 to Level 3

• Identified problem: Hard to maintain whitelists

• Idea:
  • A trusted script is allowed to add further external scripts, even from not whitelisted origins
  • In combination with nonces, no explicit whitelists are needed
    • Nonced script to bootstrap the script inclusion process

• strict-dynamic
  • allows adding scripts programmatically, eases CSP deployment in, e.g., ad scenario
  • not "parser-inserted"
  • disables host-based whitelisting
CSP - Level 3 strict-dynamic

script-src 'self' https://cdn.example.org
'nonce-d90e0153c074f6c3f53'
'strict-dynamic'
CSP - Level 3 strict-dynamic

script-src 'self' https://cdn.example.org
'nonce-d90e0153c074f6c3fcf53'
'strict-dynamic'

```html
<script nonce="d90e0153c074f6c3fcf53">
    script=document.createElement("script");
    script.src = "http://ad.com/ad.js";
    document.body.appendChild(script);
</script>
```

appendChild is not "parser-inserted"
CSP - Level 3 strict-dynamic

```html
<script nonce="d90e0153c074f6c3fcf53">
  script=document.createElement("script");
  script.src = "http://ad.com/ad.js";
  document.body.appendChild(script);
</script>
```

appendChild is not "parser-inserted"

document.write is "parser-inserted"
Script Gadgets
CSP == Attack Mitigation

- Not: Mitigation of content injection
  - This is an important distinction

- The attacker is still able to exploit the XSS

- But the injected JavaScript code does not execute
Circumvention of Attack Mitigation: Memory Corruption

• Recall: In the beginning of this talk, we drew the parallel to mitigation of memory corruption problems

• Techniques, such as the nx-bit made the direct injection of shell code impossible

• Thus, the attackers started to leverage code already that was already part of the vulnerable application
  • Return-to-LibC
  • Return Oriented Programming
Modern web frameworks

- Modern web frameworks add a lot of custom mark-up and magic

```html
<div data-role="button" data-text="I am a button"></div>

[...]

<script>
  var buttons = $('*[data-role=button]');
  buttons.html(buttons.attr('data-text'));
</script>

<div data-role="button" ...>I am a button</div>
```
Using script gadgets to bypass CSP [CCS17]

```
<?php
echo $_GET['username']
?>

<div data-role="button" data-text="I am a button"></div>
<script nonce="d90e0153c074f6c3fcf53">
  var buttons = $('[data-role=button]');
  buttons.html(button.getAttribute('data-text'));
</script>
```

Attacker cannot guess the correct nonce, so we should be safe here, right?
Using script gadgets to bypass CSP [CCS17]

```
<script nonce="d90e0153c074f6c3fcf53">
  var buttons = $("[data-role=button]"送去
  buttons.html(button.getAttribute("data-text"));
</script>

<div data-role="button" data-text="I am a button"></div>
```

jQuery uses appendChild instead of document.write when adding a script
Using script gadgets to bypass CSP [CCS17]

- Idea: use existing expression parsers/evaluation functions in MVC frameworks

- Lekies et al evaluated widely used frameworks
  - Aurelia, Angular, and Polymer bypass all mitigations via expression parsers

- Often times trivial exploits
  - e.g., Bootstrap `<div data-toggle=tooltip data-html=true title='&lt;script&gt;alert(1)&lt;/script&gt;'&gt;&lt;/div&gt;

- More involved examples require "chains" of calls
  - sometimes depended on a specific function being called, e.g., jQuery's `after` or `html`
Types of script gadget

• Circumventing strict-dynamic
  • The SG queries data from the DOM
  • This data is used to create new, potentially script carrying elements
  • The created code inherits the trust of the SG

• Abusing unsafe-eval
  • The SG queries data from the DOM
  • Within the SG is a data flow into the eval API

• Circumventing nonces or whitelists
  • Sophisticated frameworks contain “expression parsers”
  • In essence, they bring their own JavaScript runtime
How many JavaScript frameworks contain SGs?

- Data collection
  - Trending JavaScript frameworks (Vue.js, Aurelia, Polymer)
  - Widely popular frameworks (AngularJS, React, EmberJS)
  - Older still popular frameworks (Backbone, Knockout, Ractive, Dojo)
  - Libraries and compilers (Bootstrap, Closure, RequireJS)
  - Query-based libraries (jQuery, jQuery UI, jQuery Mobile)

- In total 16 libraries were examined

<table>
<thead>
<tr>
<th>Whitelists</th>
<th>Nonces</th>
<th>CSP</th>
<th>Strict-dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>10</td>
<td>13</td>
</tr>
</tbody>
</table>
Aside: Script Gadget circumvent more than CSP only

- SGs also cause problems for
  - Web Application Firewalls
    - Harmless content is transformed into attacks after rendering
  - XSS Filters
    - No matching between request data and exploit code
  - HTML sanitizers
    - HTML sanitizers remove known-bad and unknown HTML elements and attributes
    - Exploit is in “harmless” data-attributes
Gadgets in custom code

• Fixing a few libraries is easier than fixing all Web sites

• How common are gadgets in user land code?
  • Gadgets might be less common than in libraries
  • Identifying Gadgets in user land code requires automation

Example:
```
<div id="mydiv" data-text="Some random text"></div>

elem.innerHTML = $('#mydiv').attr('data-text');
```
Automatic finding of custom gadgets (I)

- Methodology
  - Usage of a taint-enabled web browser
  - The web browser records all data flows from the DOM into the DOM
    - Taint source: DOM nodes
    - Taint sinks: All applicable APIs that could cause Script Gadgets
  - Crawl of the Alexa top 5000, one level deep

Top 5000 Alexa = 647,085 pages on 4,557 domains
Automatic finding of custom gadgets (II)

- Verification of script gadget
  - Not every flow is vulnerable

- Automatically create exploit
  - Taint-engine provides precise source and sink information
  - Build HTML snippet, that causes the data flow and ends in JS execution

- Simulate XSS problem
  - Insert the HTML snippet in the page on loadtime
  - Record, if the injected JS was executed
Automatic finding of custom gadgets (II)

- Verification of script gadget
  - Not every flow is vulnerable

- Automatically create exploit
  - Taint-engine provides precise source and sink information
  - Build HTML snippet that causes data flow and ends in JS execution

- Simulate XSS
  - Insert the HTML snippet in the page on loadtime
  - Record if the injected JS was executed

285,894 verified gadgets on 906 domains (19.88%)
Study results on CSP (I)

• In the context of this talk, we are mainly interested in SGs that undermine CSP policies
  • Strict-dynamic
  • Unsafe-eval

• Thus, we specifically look for gadgets that:
  • The data flows ending within text, textContent or innerHTML of a script tag
  • The data flow ending within text, textContent or innerHTML of a tag, where the tag name is DOM-controlled (tainted)
  • The data flow ending within script.src
  • The data flow ending in an API which is known for creating and appending script tags to the DOM.
Study results on CSP (II)

• How (in)secure are different CSP keywords?

• CSP unsafe-eval
  • Unsafe-eval is considered secure
  • 48% of all domains have a potential eval gadget

• CSP strict-dynamic
  • Flows into script.text/src, jQuery’s .html(), or createElement(tainted).text
  • 73% of all domains have a potential strict-dynamic gadget.

• Data shows strict-dynamic and unsafe-eval considerably weaken a policy.
Conclusion

• Strong CSPs provide a high level of protection

• Unfortunately strong policies are seldom feasible

• CSP Level 2 + 3 provide flexible tools to ease the adoption of the mechanism
  • But, they have to be handled with care

• Script Gadgets are problematic
  • Not only for CSP but for XSS mitigation / defence in general
  • Research into Script Gadgets is still young
CSP - Report Only Mode

- Implementation of CSP is tedious process
  - removal of all inline scripts and usage of eval
  - tricky when depending on third-party providers
    - e.g., advertisement includes random script (due to real-time bidding)

- Restrictive policy might break functionality
  - remember: client-side enforcement
  - need for feedback channel to developers

- Content-Security-Policy-Report-Only
  - `default-src ....; report-uri /violations.php`
  - allows to field-test without breaking functionality (reports current URL and causes for fail)
  - does not work in meta element
References

- Content Security Policy 1.0, https://www.w3.org/TR/CSP1/
- Content Security Policy Level 2, https://www.w3.org/TR/CSP2/
- Sid Stamm, Brandon Sterne, Gervase Markham: Reining in the web with content security policy. WWW 2010: 921-930