CSE480: Web Security

9. Content Security Policy

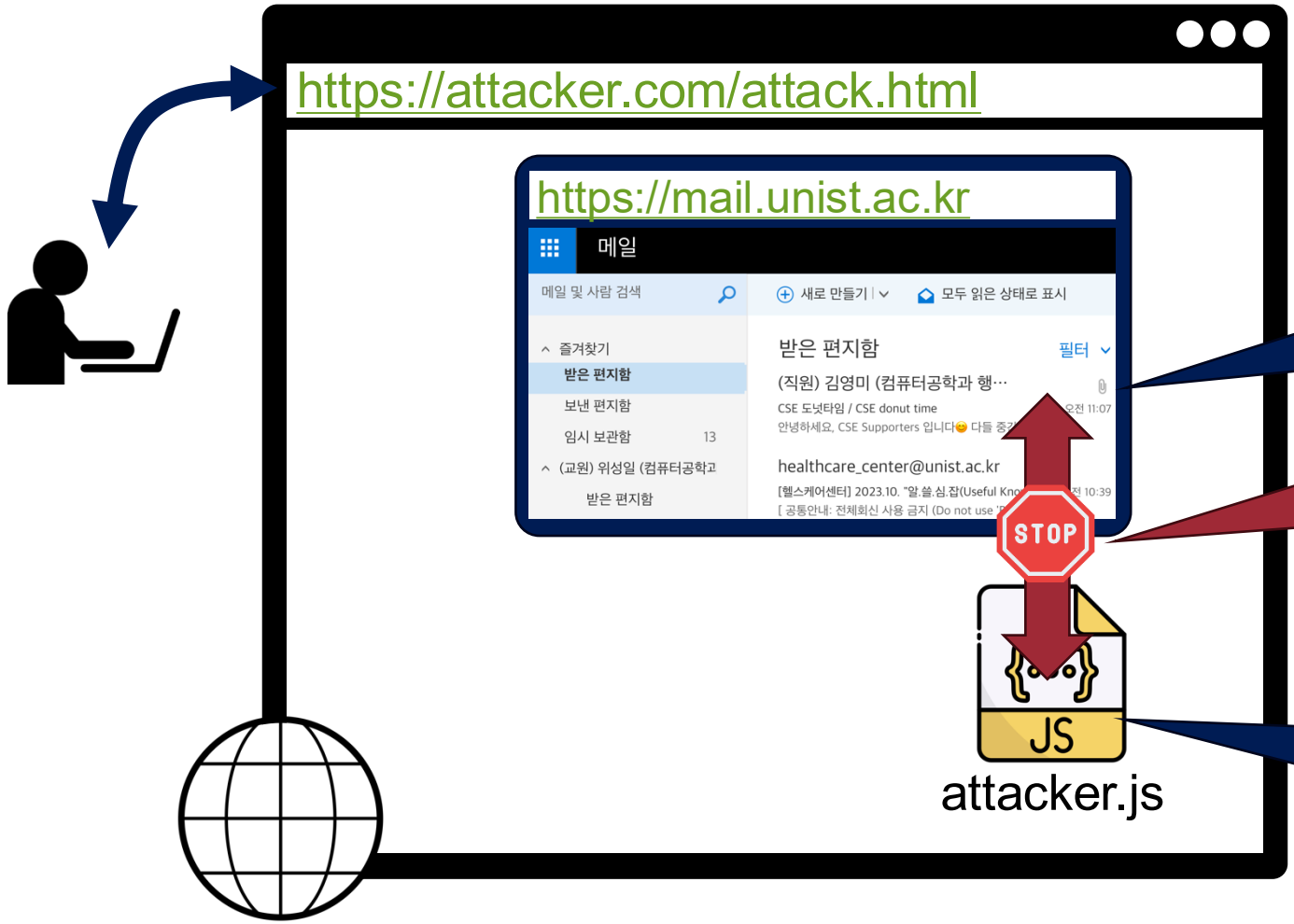
Seongil Wi



Department of Computer Science and Engineering

Recap: Same Origin Policy (SOP)

- Restricts scripts on one origin from accessing data from another origin



Any resource has its own origin

Resources located at the *https://mail.unist.ac.kr* origin

A JS runs with a origin cannot access other origin resources

JavaScript runs with a *https://attacker.com* origin

Recap: What is an Origin?

- **Origin = Protocol + Domain Name + Port**
- Two URLs have the same origin if the **protocol**, **domain name** (not subdomains), **port** are the same for both URLs
  - All three must be equal origin to be considered the same

Recap: Cross-Site Scripting (XSS)

- A code injection attack
- Malicious scripts are injected into benign and trusted websites
- Injected codes are executed at **the attacker's target origin**

Recap: Cross-Site Scripting (XSS)

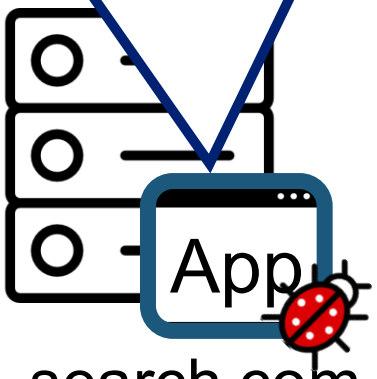
```
<html>
  <body>
    Search result for <?php echo $_GET['query'];?>
    <?php
      // get results from DB and print them
    ?>
```

https://search.com?query=<script>alert('hi');</script>




```
<html>
  <body>
    Search result for <script>alert('hi')</script>
    ...
  </body>
</html>
```

Injected malicious codes are executed at the https://search.com origin



search.com
(vulnerable web app)

Recap: XSS Type (IMPORTANT!!)

- Reflected XSS (Server-side XSS)
- Stored XSS
- DOM-based XSS (Client-side XSS)
- Universal XSS

Recap: How to Prevent XSS Attacks?

#1: Input validation/sanitization

- Any user input must be preprocessed before it is used inside HTML
- Option 1-1: Implement your own sanitization logic (not recommended)

- Option 1-2: Use the good escaping libraries
  - E.g., `htmlspecialchars(string)`, `htmlentities(string)`, …

#2: Content Security Policy (CSP)

- A new security mechanism supported by modern browsers

Today's Topic!

Content Security Policy (CSP)



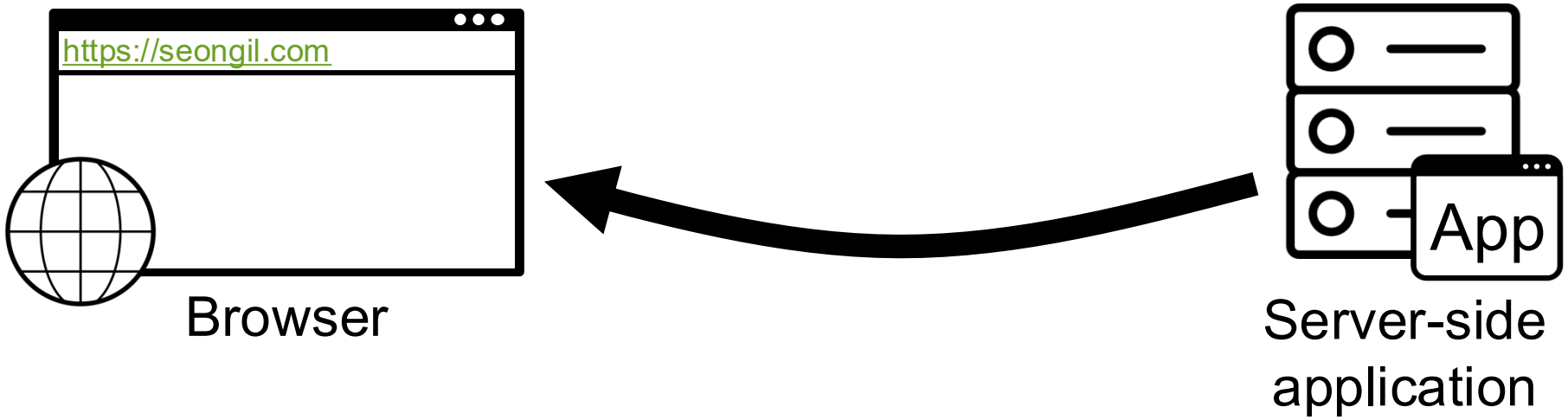
Content Security Policy (CSP)



- Explicitly allow resources which are trusted by the developer
  - Servers declare trusted sources

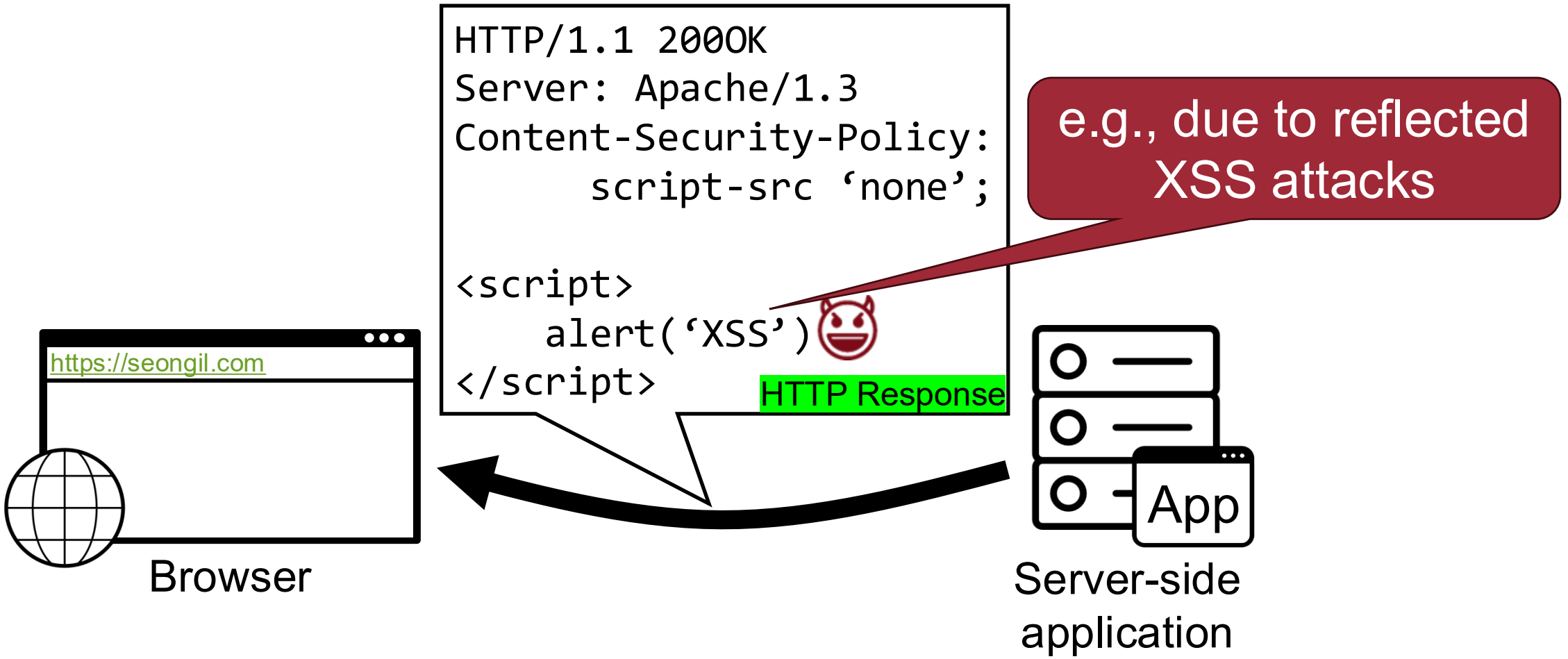
CSP Workflow

- Explicitly allow resources which are trusted by the developer
  - Servers declare trusted sources



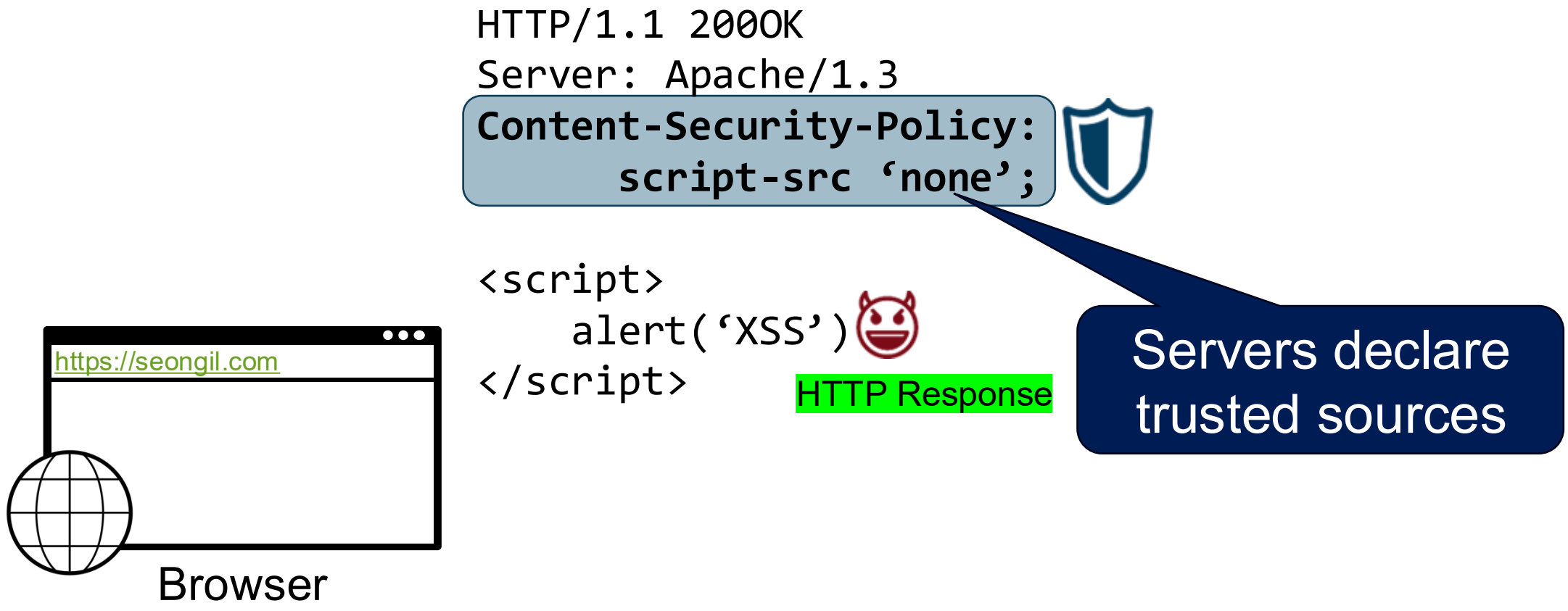
CSP Workflow

- Explicitly allow resources which are trusted by the developer
  - Servers declare trusted sources



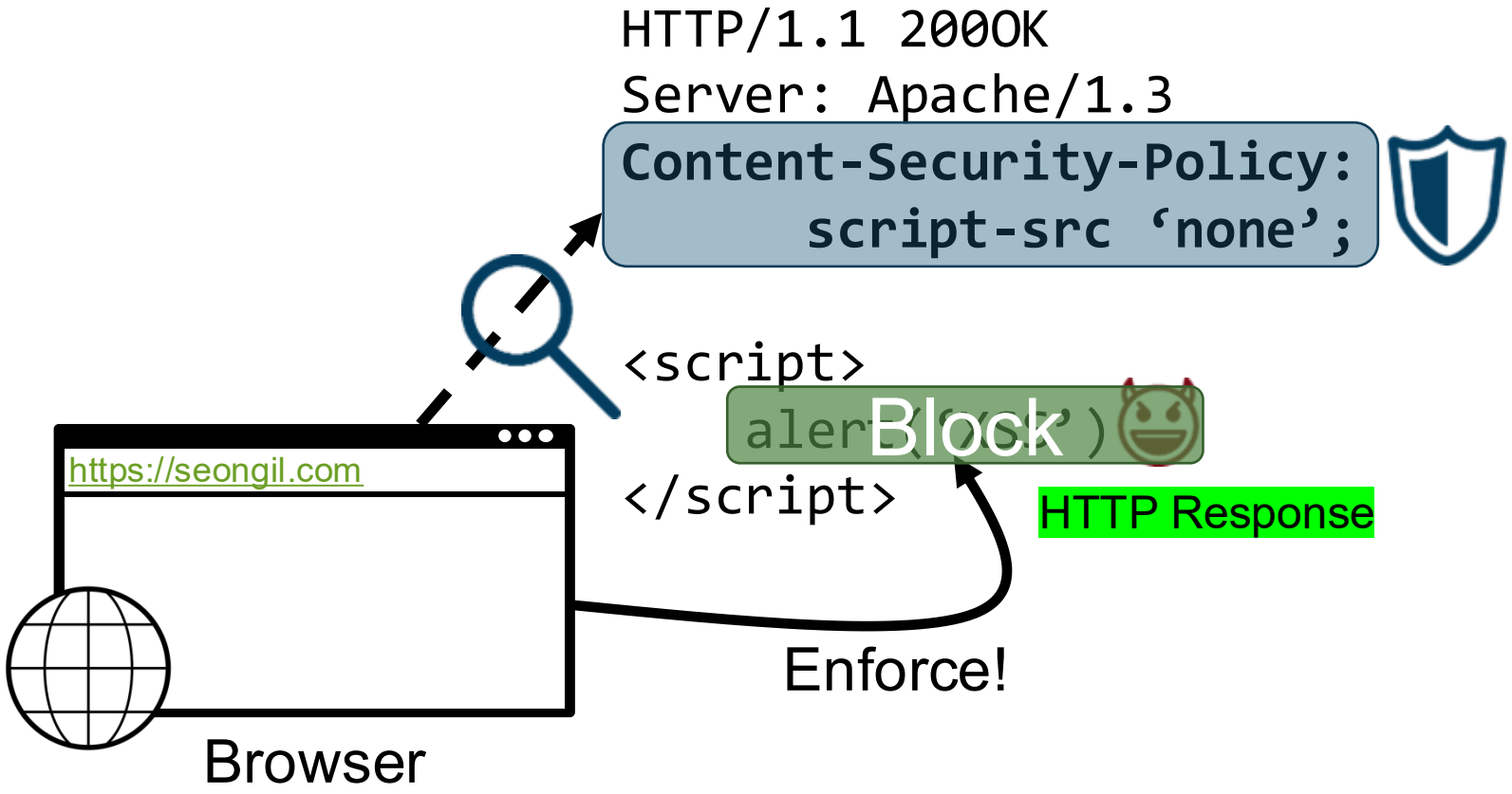
CSP Workflow

- Explicitly allow resources which are trusted by the developer
  - Servers declare trusted sources



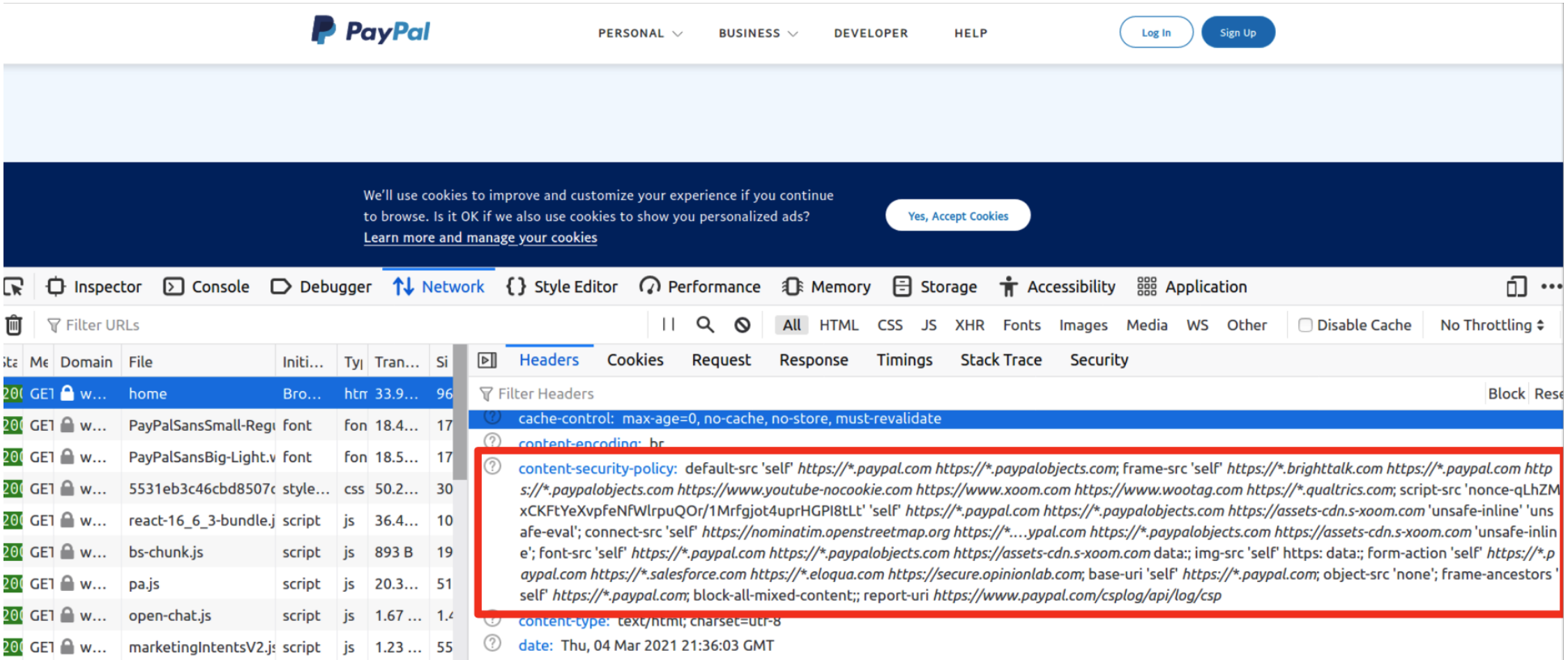
CSP Workflow

- Explicitly allow resources which are trusted by the developer
  - Servers declare trusted sources



Example Policy on paypal.com

Demo: https://www.paypal.com/home



Content Security Policy (CSP)

- Explicitly allow resources which are trusted by the developer
  - Servers declare trusted sources

- Disallow dangerous JS constructs like eval or event handlers
- Delivered as HTTP header or in meta element in page
  - **HTTP header:** `Content-Security-Policy: default-src …`
  - **Meta element**: `<meta http-equiv="Content-Security-Policy" content="default-src…">`

- **Enforced by the browser** (**all policies** must be satisfied)
  - Your browser must support CSP for its use
- First candidate recommendation in 2012, currently at Level 3

Browser Support

Chrome

- `Content-Security-Policy` CSP Level 3 - Chrome 59+ Partial Support
- `Content-Security-Policy` CSP Level 2 - Chrome 40+ Full Support Since January 2015
- `Content-Security-Policy` CSP 1.0 - Chrome 25+
- `X-Webkit-CSP` Deprecated - Chrome 14-24

Safari

- `Content-Security-Policy` CSP Level 3 - Safari 15.4+ Partial Support
- `Content-Security-Policy` CSP Level 2 - Safari 10+
- `Content-Security-Policy` CSP 1.0 - Safari 7+
- `X-Webkit-CSP` Deprecated - Safari 6

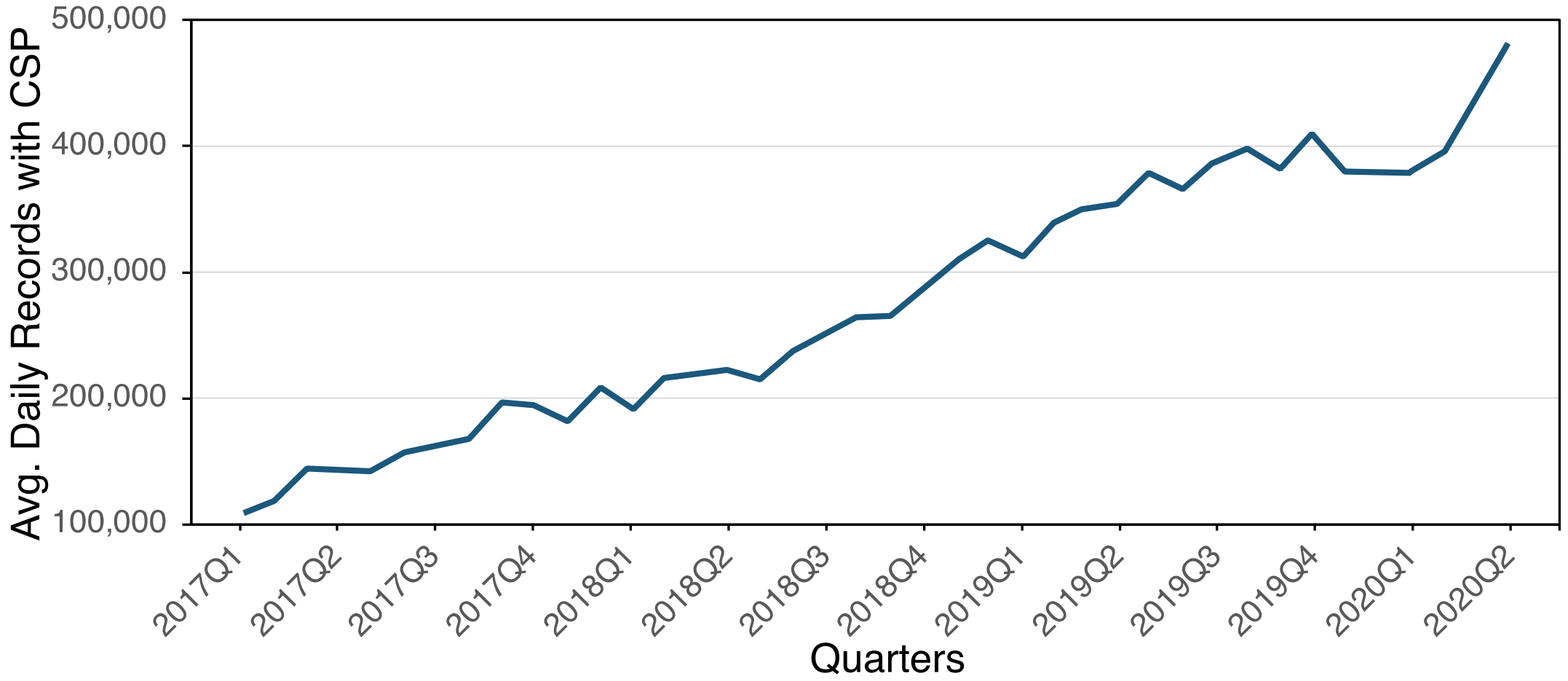
Firefox

- `Content-Security-Policy` CSP Level 3 - Firefox 58+ Partial Support
- `Content-Security-Policy` CSP Level 2 - Firefox 31+ *Partial Support* since July 2014
- `Content-Security-Policy` CSP 1.0 - Firefox 23+ Full Support
- `X-Content-Security-Policy` Deprecated - Firefox 4-22

Edge

- `Content-Security-Policy` CSP Level 3 - Edge 79+ Partial Support
- `Content-Security-Policy` CSP Level 2 - Edge 15+ Partial, 76+ Full
- `Content-Security-Policy` CSP 1.0 - Edge 12+

CSP Popularity



https://www.bitsight.com/blog/content-security-policy-limits-dangerous-activity-so-why-isnt-everyone-doing-it

Format of CSP


```
Policy := [directive [value1];]...
```

A list of pairs of directive and values

CSP Level 1 – Controlling Scripting Resources

```
Policy := [directive [value1];]...
```

✓ **Directive**: `script-src`
- Specifically controls where scripts can be loaded from
- **If provided, inline scripts and eval will not be allowed**

✓ **Value**: Many different ways to control sources
- **‘none’** – no scripts can be included from any host
- **‘self’** – only own origin
- **https://domain.com** – allow the script from this origin
- **https://*.domain.com** – any subdomain of domain.com, any script on them
- **https:** – any origin delivered via HTTPs
- **‘unsafe-inline’ / ‘unsafe-eval’** – reenables inline handlers and eval

CSP Level 1 – Controlling Additional Resources

- **`img-src, style-src, font-src, object-src, media-src`**
  - Controls non-scripting resources: images, CSS, fonts, objects, audio/video
- **`frame-src`**
  - Controls from which origins frames may be added to a page
- **`connect-src`**
  - Controls XMLHttpRequest, WebSockets (and other) connection targets
- **`default-src`**
  - Serves as fallback for all fetch directives (all of the above)
  - **Only used when specific directive is absent**

CSP Level 1 – Limitations

- If our goal is to allow scripts from own origin and inline scripts
  - Solution: `script-src 'self' 'unsafe-inline'`
- Problem: bypasses literally any protection
  - Attacker can inject inline JavaScript
- One possible solution: removing inline script and converting it into an external script

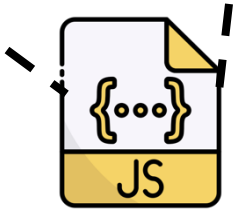
For each inline script…

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

converting

```
<script src="myscript.js">
</script>
```

```
alert(1)
```



myscript.js

Problem: Removing inline script is painful

CSP Level 2 – Nonces and Hashes

- Proposed improvement in CSP Level 2: **nonces and hashes**
- Allows every inline script adds nonce property

```
–script-src ‘nonce-$value’ ‘self’
```

- Allows inline scripts based on their SHA hash (SHA256, SHA384, or SHA512)

```
–script-src ‘sha256-$hash’ ‘self’
```

CSP Level 2 – Example


```
script-src 'self' https://cdn.example.org
‘nonce-d90e0153c074f6c3fcf53' 'sha256-
5bf5c8f91b8c6adde74da363ac497d5ac19e4595fe39cbdda22cec8445d3814c'
```

```
<script>
alert(“My hash is correct”)
</script>
```

```
<script>
alert(“incorrect”)
</script>
```

SHA256 hash value:
5bf5c8f91b8c6adde74da363ac497d5ac19
e4595fe39cbdda22cec8445d3814c

CSP Level 2 – Example


```
script-src 'self' https://cdn.example.org
‘nonce-d90e0153c074f6c3fcf53' 'sha256-
5bf5c8f91b8c6adde74da363ac497d5ac19e4595fe39cbdda22cec8445d3814c'
```

```
<script>
alert(“My hash is correct”)
</script>
```

✓

```
<script>
alert(“incorrect”)
</script>
```

✗

SHA256 matches value of CSP header

SHA256 does not match

CSP Level 2 – Example


```
script-src 'self' https://cdn.example.org
‘nonce-d90e0153c074f6c3fcf53' 'sha256-
5bf5c8f91b8c6adde74da363ac497d5ac19e4595fe39cbdda22cec8445d3814c'
```

```
<script>
alert(“My hash is correct”)
</script>
```



```
<script>
  alert(“My hash is correct”)
</script>
```



SHA256 matches value of CSP header

SHA256 does not match (whitespaces matter)

CSP Level 2 – Example


```
script-src 'self' https://cdn.example.org
‘nonce-d90e0153c074f6c3fcf53' 'sha256-
5bf5c8f91b8c6adde74da363ac497d5ac19e4595fe39cbdda22cec8445d3814c'
```

```
<script nonce=“d90e0153c074f6c3fcf53”>
alert(“It’s all good”)
</script>
```

✓

```
<script nonce=“nocluehackplz”>
alert(“I will not work”)
</script>
```

✗

Script nonce matches CSP header

Script nonce does not match CSP header

CSP Level 2 – Additional Changes

- `child-src`
  - Deprecates frame-src, also valid for Web Workers
- `base-uri`
  - Controls whether `<base>` can be used and what it can be set to
- `form-action`
  - Ensured that forms may only be sent to specific targets
  - Does not fall back to `default-src` if not specified

CSP Level 2 – Limitations


```
script-src 'self' https://cdn.example.org
‘nonce-d90e0153c074f6c3fcf53'
```

```
<script nonce=“d90e0153c074f6c3fcf53”>
  script=document.createElement("script");
  script.src = "http://ad.com/ad.js";
  document.body.appendChild(script);
</script>
```

Add new script element without nonce

Does this script work under a nonce-based policy?

No!

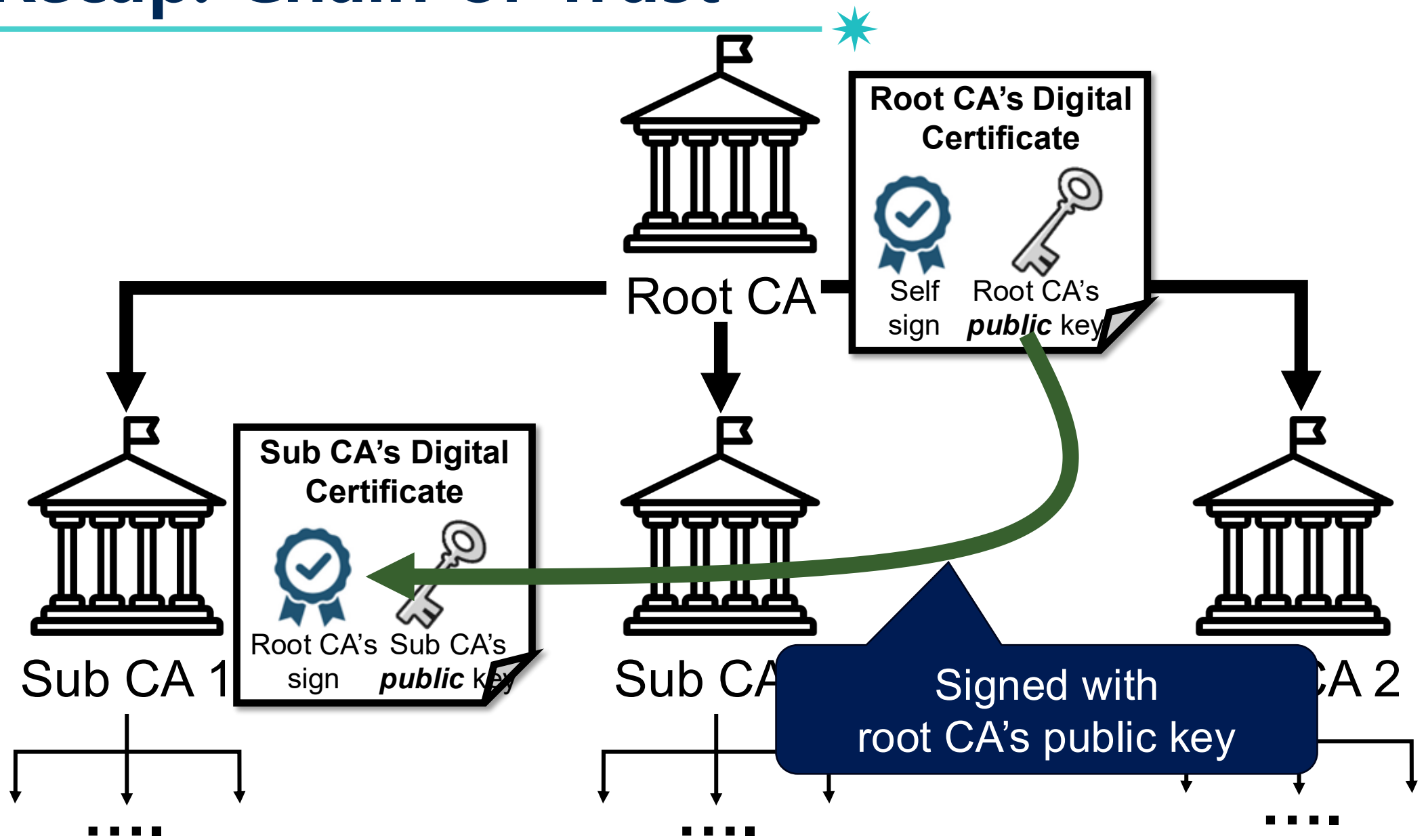
Changes from Level 2 to Level 3: `strict-dynamic`

- Additional changes: add **`strict-dynamic`**
  - Allows adding scripts programmatically, eases CSP deployment in, e.g., ad scenario

- Mostly due to dynamic ADs
  - 1st page load: script from ads.com → fancy-cars.com
  - 2nd page load: script from ads.com → cheap-ads.net → dealsdeals.biz

- **Idea: propagate trust**
  - If we trust ads.com, let's also trust whoever ads.com load script from

Recap: Chain of Trust



CSP Level 3 – strict-dynamic Example


```
script-src 'self' https://cdn.example.org
‘nonce-d90e0153c074f6c3fcf53’ ‘strict-dynamic’
```

```
<script nonce=“d90e0153c074f6c3fcf53”>
  script=document.createElement("script");
  script.src = "http://ad.com/ad.js";
  document.body.appendChild(script);
</script>
```

We trust this script

Propagate trust: we also trust this script, so we allow it to execute

Changes from Level 2 to Level 3: `strict-dynamic`

- Additional changes: add **`strict-dynamic`**
  - Allows adding scripts programmatically, eases CSP deployment in, e.g., ad scenario

- Not “parser-inserted”

CSP Level 3 – strict-dynamic


```
script-src ‘nonce-d90e0153c074f6c3fcf53’ ‘strict-dynamic'
```

```
<script nonce=“d90e0153c074f6c3fcf53”>
  script =
     document.createElement(“script”);
  script.src = “http://ad.com/ad.js”;
  document.body.appendChild(script);
</script>
```

✓

appendChild is not “parser-inserted”

```
<script nonce=“d90e0153c074f6c3fcf53”>
  document.write(“<script
      src = “http://ad.com/ad.js”>
      </script>”)
</script>
```

✗

document.write is “parser-inserted”

CSP – Reporting Functionality

- report-uri <url>
  - Sends JSON report to specified URL

- report-to <endpoint>
  - Requires separate definition through Report-To HTTP header

Important!

- CSP does not stop XSS, tries to mitigate its effects



How can we bypass CSP?

CSP – Bypasses



- Problem #1: User input at the trusted script

Problem #1: User Input


```
script-src ‘nonce-random123’ ‘strict-dynamic’;
```

- What if the injection happens directly at nonced script elements?

```
<script nonce=“random123”>
  script=document.createElement("script");
  script.src = user_input + “valid.js";
  document.body.appendChild(script);
</script>
```

Problem #1: User Input


```
script-src ‘nonce-random123’ ‘strict-dynamic’;
```

- What if the injection happens directly at nonced script elements?

```
<script nonce=“random123”>
  script=document.createElement("script");
  script.src = user_input + “valid.js";
  document.body.appendChild(script);
</script>
```

Executes on the target origin

Attacker can inject attacker.com

CSP Bypass: JSONP

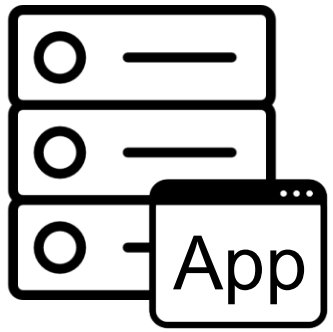
- Any allowed site with JSONP endpoint is potentially dangerous
  - E.g., `https://allowed.com/jsonp?callback="my malicious code here"//`

Recap: JSONP XSS Attacks

- What if an attacker has a change to inject some string value in the JSONP URL?

https://vulnerable.com

```
<script>
  function read(json) {
    document.write(json.temp)
  }
</script>
<script
  src="http://weather.com/
        jsonp?callback=
        alert('xss');read">
</script>
```



weather.com
web server

```
weather.com/jsonp?callback=
                alert('xss');read

alert('xss');read([{
"temp": 36
"location": "ULSAN"
}])
```

CSP Bypass: Base Tag Bypass

Origin: https://victim.com

```
Content-Security-Policy: script-src 'nonce-abc123'
```

```
<html>



    <script nonce="abc123" src="/js/app.js"></script>
</html>
```

Allow to execute
https://victim.com/js/app.js

CSP Bypass: Base Tag Bypass

Origin: https://victim.com

```
Content-Security-Policy: script-src 'nonce-abc123'
```

```
<html>
    <base href="https://attacker.com/">

    <script nonce="abc123" src="/js/app.js"></script>
</html>
```

Injected HTML snippet

Allow to execute
https://attacker.com/js/app.js

CSP – Bypasses



- Problem #1: User input at the trusted script

- Problem #2: Developer’s mistake/misconfiguration

Problem #2: Developer's Mistake/Misconfiguration

- Developer's ***mistake***

```
defalt-src 'self'
```

  - Typo in the first directive leads to the `default-src` directive being missing from the policy (Content Security Problems?, ***CCS'16***)

- Developer's ***misconfiguration***

```
default-src 'unsafe-inline' *
```

  - Defining CSP is hard!
  - Many website developers just allow all of the inline script and all hosts
  - **94.72% of all website bypassible (e.g., misconfigured their CSP)** (CSP Is Dead, Long Live CSP!, ***CCS'2016***)

CSP Is Dead, Long Live CSP!, *CCS'16*

- The first in-depth analysis of the security of CSP deployments across the web

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

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 inves-

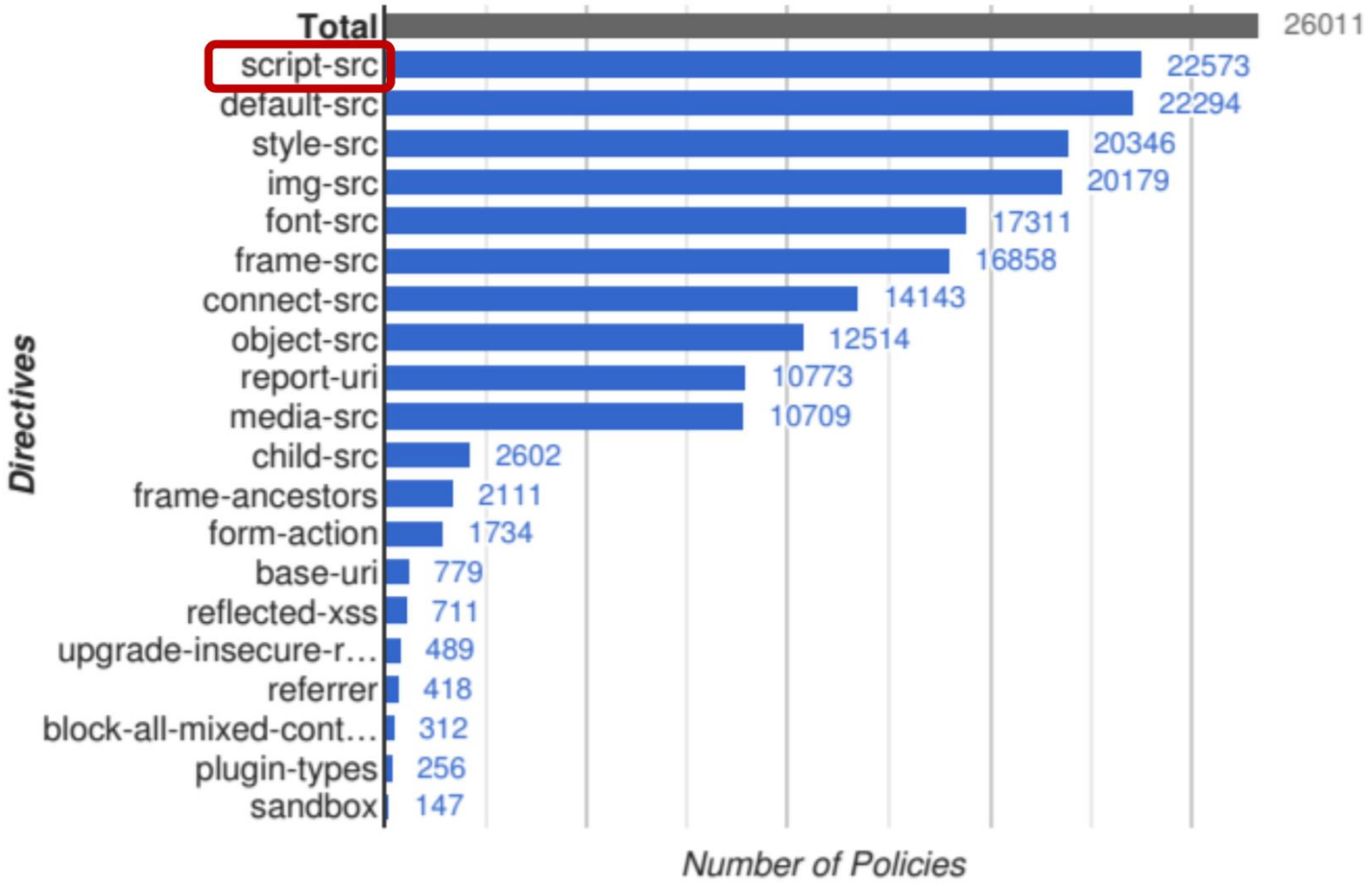
CSP Is Dead, Long Live CSP!, *CCS'16*

- Among 106 billion unique URLs from 1 billion hostnames and 175 million top private domains,
  - 3.9 billion URLs carried a CSP (3.7%)

- Collected 26,011 unique policies

CSP Is Dead, Long Live CSP!, *CCS'16*

- Among 106 million hosts, collected 26,011 unique policies
  - 87% were to prevent XSS



CSP Is Dead, Long Live CSP!, *CCS'16*

- Among 106 million hosts, collected 26,011 unique policies
  - 87% were to prevent XSS
  - 94.72% were bypassable!

CSP Is Dead, Long Live CSP!, *CCS'16*

Data Set	Total	Report Only	Bypassable				
			Unsafe Inline	Missing `object-src`	Wildcard in Whitelist	Unsafe Domain	Trivially Bypassable Total
Unique CSPs	26,011	2,591 9.96%	21,947 84.38%	3,131 12.04%	5,753 22.12%	19,719 75.81%	24,637 94.72%
XSS Policies	22,425	0 0%	19,652 87.63%	2,109 9.4%	4,816 21.48%	17,754 79.17%	21,232 94.68%
Strict XSS Policies	2,437	0 0%	0 0%	348 14.28%	0 0%	1,015 41.65%	1,244 51.05%

- **XSS Policies**: A policy that holds script-src or object-src directive
- **Strict XSS Policies**: A policy that does NOT hold unsafe directive values including `'unsafe-inline'` and * for whitelisting all hosts
- 94.72% is bypassible!

CSP – Bypasses



- Problem #1: User input at the trusted script
- Problem #2: Developer’s mistake/misconfiguration
- Problem #3: Browser bugs (CSP enforcement bugs)

Problem #3: Browser Bugs

- NOTE: CSP is enforced by the browser

Well-defined CSP →  → Execute attacker's JS code

Chrome

```
<?php
    header("HTTP/: 100");
    header("Content-Security-Policy: default-src 'self'")
?>
<script>alert(1)</script>
```

Expected behavior: Not execute JS code

 ✓ Execute JS code

Chrome

 X Not execute JS code

Safari

 X Not execute JS code

Firefox

Problem #3: Browser Bugs

CSP

```
script-src benign.com
```

HTML

```
<script>attack()</script>
```

Block!

CVE-2020-6519

```
<iframe src="javascript:attack()"></iframe>
```

Allow!

CVE-2021-30538

```
<iframe srcdoc="<script>attack()</script>"></iframe>
```

Allow!

Research Question:
How to find
CSP-related browser bugs?

DiffCSP, *NDSS'2023*

- The first testing framework that identifies browser bugs in CSP enforcement regarding JS execution

DiffCSP: Finding Browser Bugs in Content Security Policy Enforcement through Differential Testing

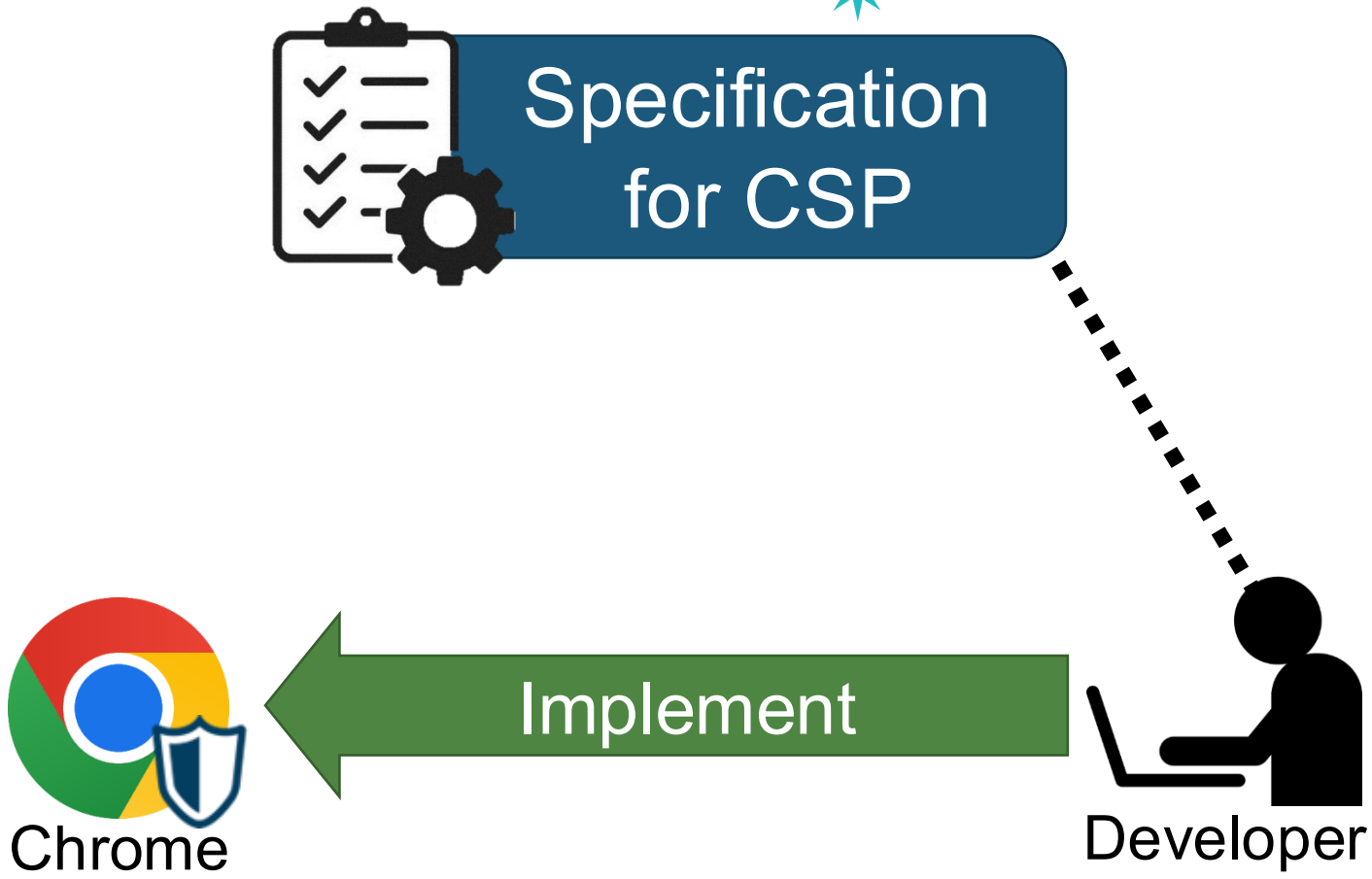
Seongil Wi*, Trung Tin Nguyen†‡, Jihwan Kim*, Ben Stock†, Sooel Son*

*School of Computing, KAIST
†CISPA Helmholtz Center for Information Security
‡Computer Science Graduate School, Saarland University

***Abstract*—The Content Security Policy (CSP) is one of the *de facto* security mechanisms that mitigate web threats. Many websites have been deploying CSPs mainly to mitigate cross-site scripting (XSS) attacks by instructing client browsers to**

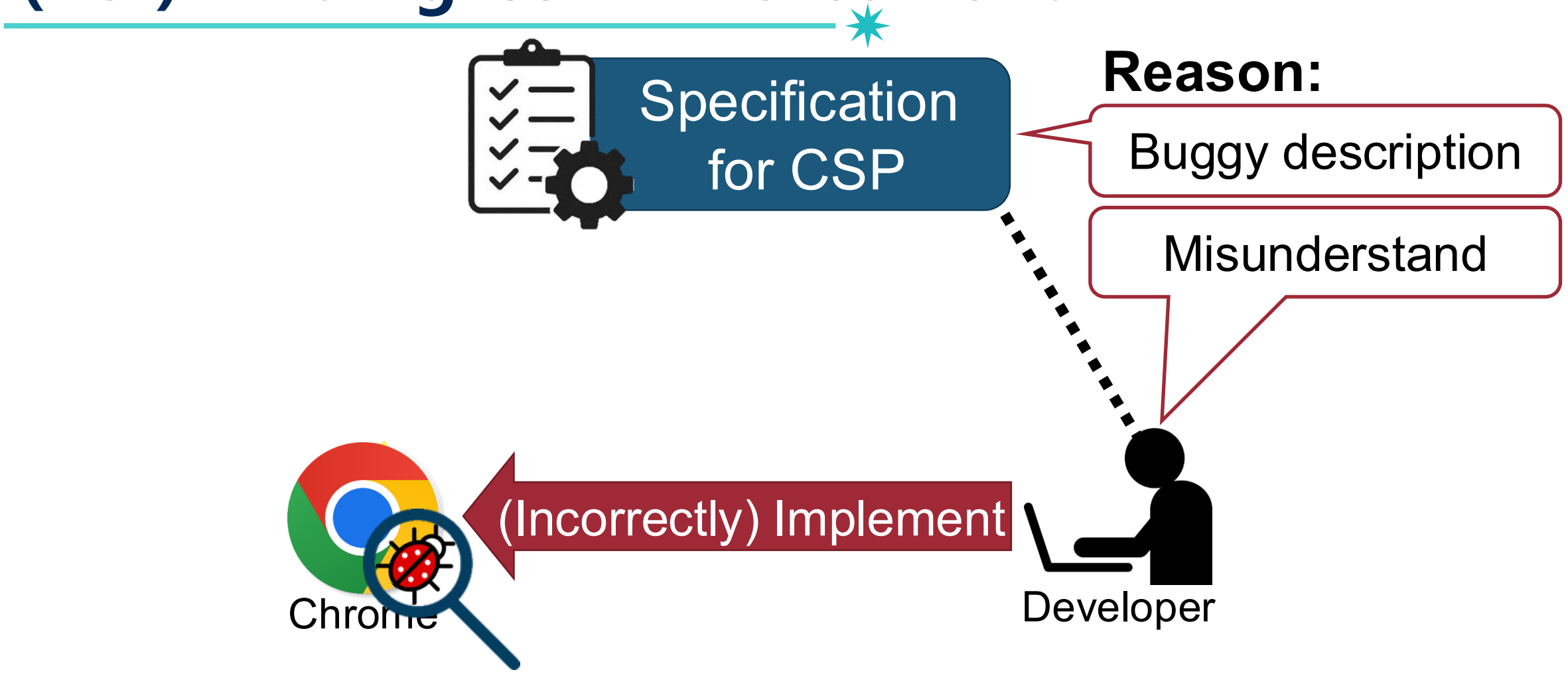
```
1  XSS attack payload:
2    http://[Target URL]/PoC.html#javascript:alert('XSS')
3  CSP: script-src-elem 'sha256-aHbTR...';
4  Target website:
5  <script>
```

(Ref) Finding CSP Enforcement

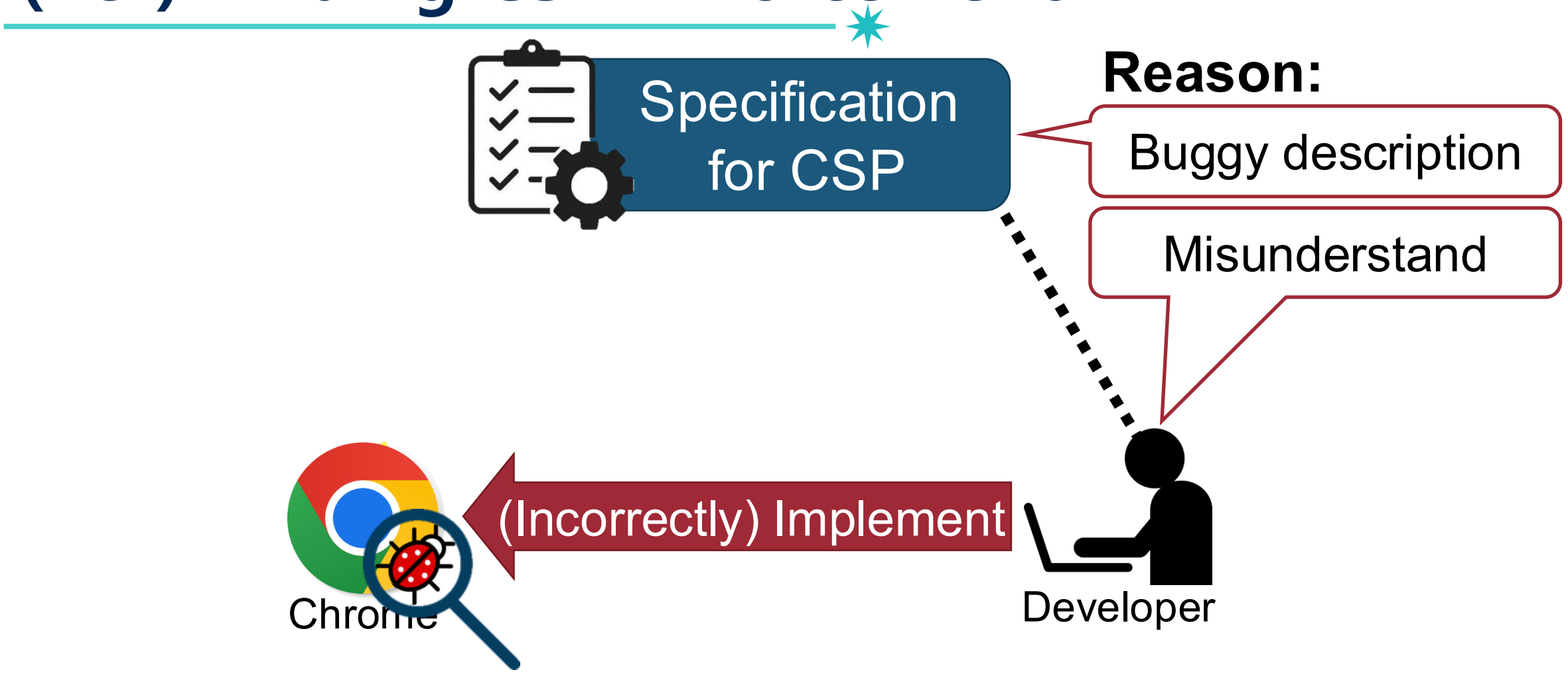


Specification for CSP: https://www.w3.org/TR/CSP3/

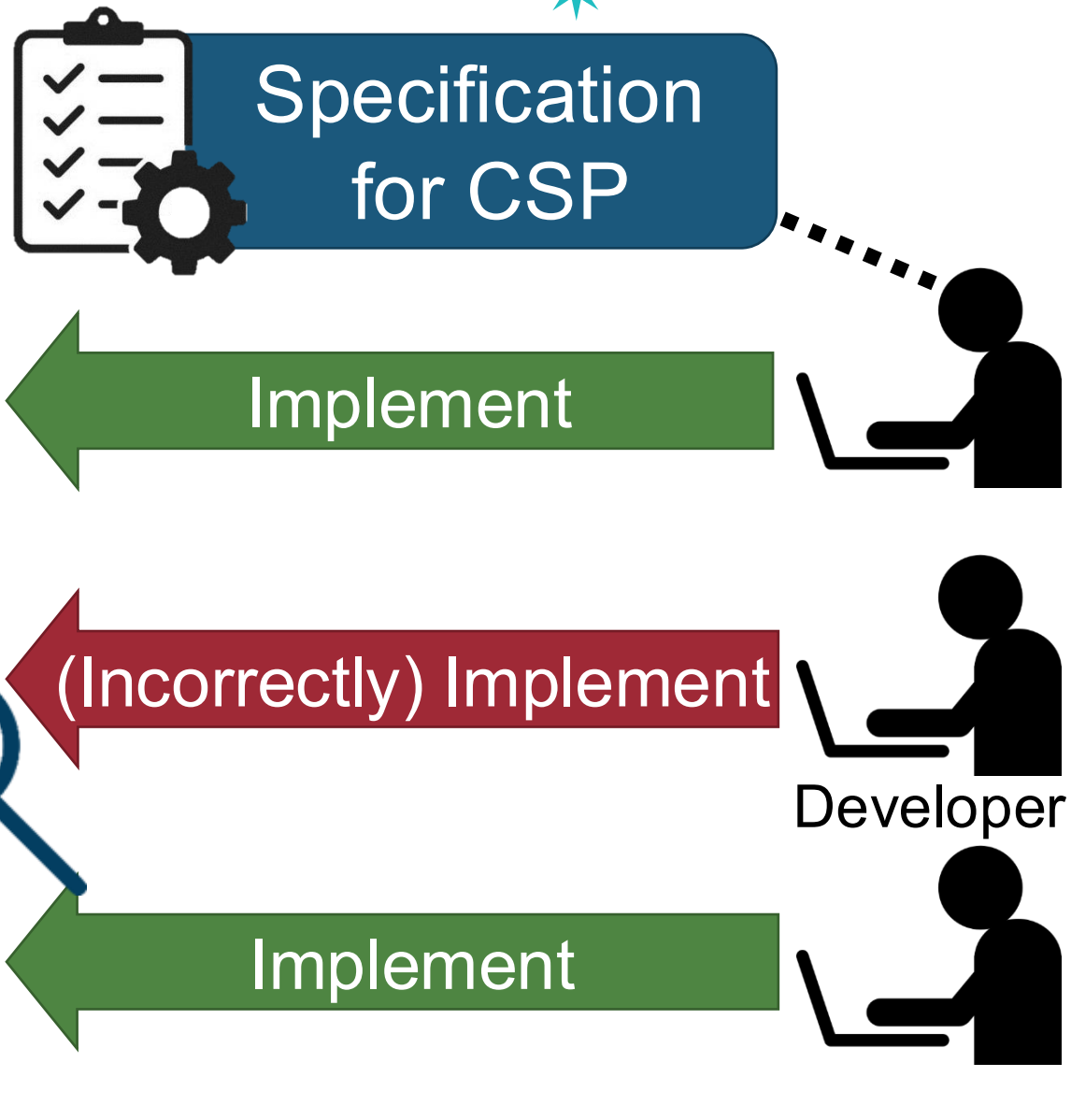
(Ref) Finding CSP Enforcement



(Ref) Finding CSP Enforcement

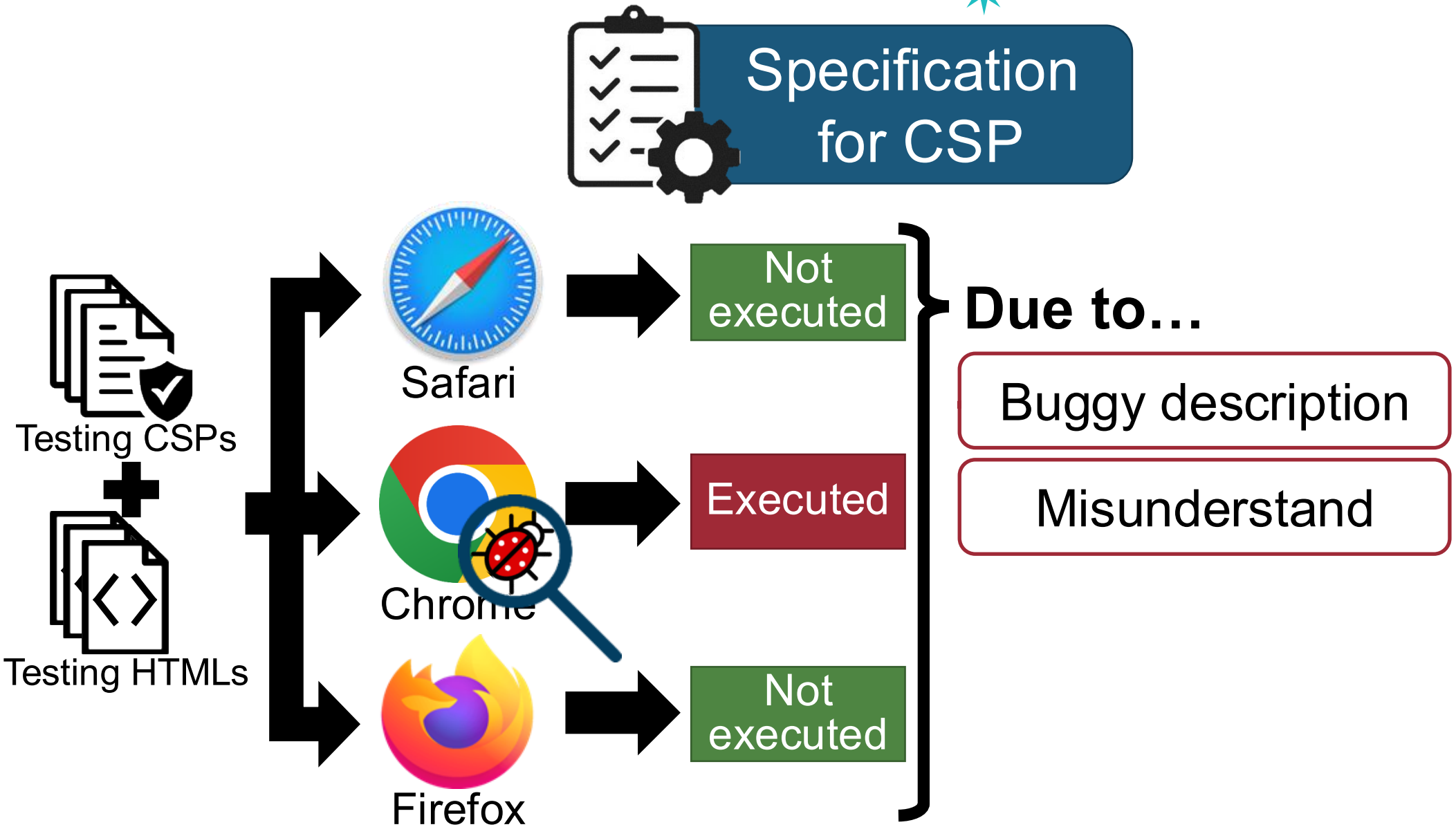


(Ref) Finding CSP Enforcement



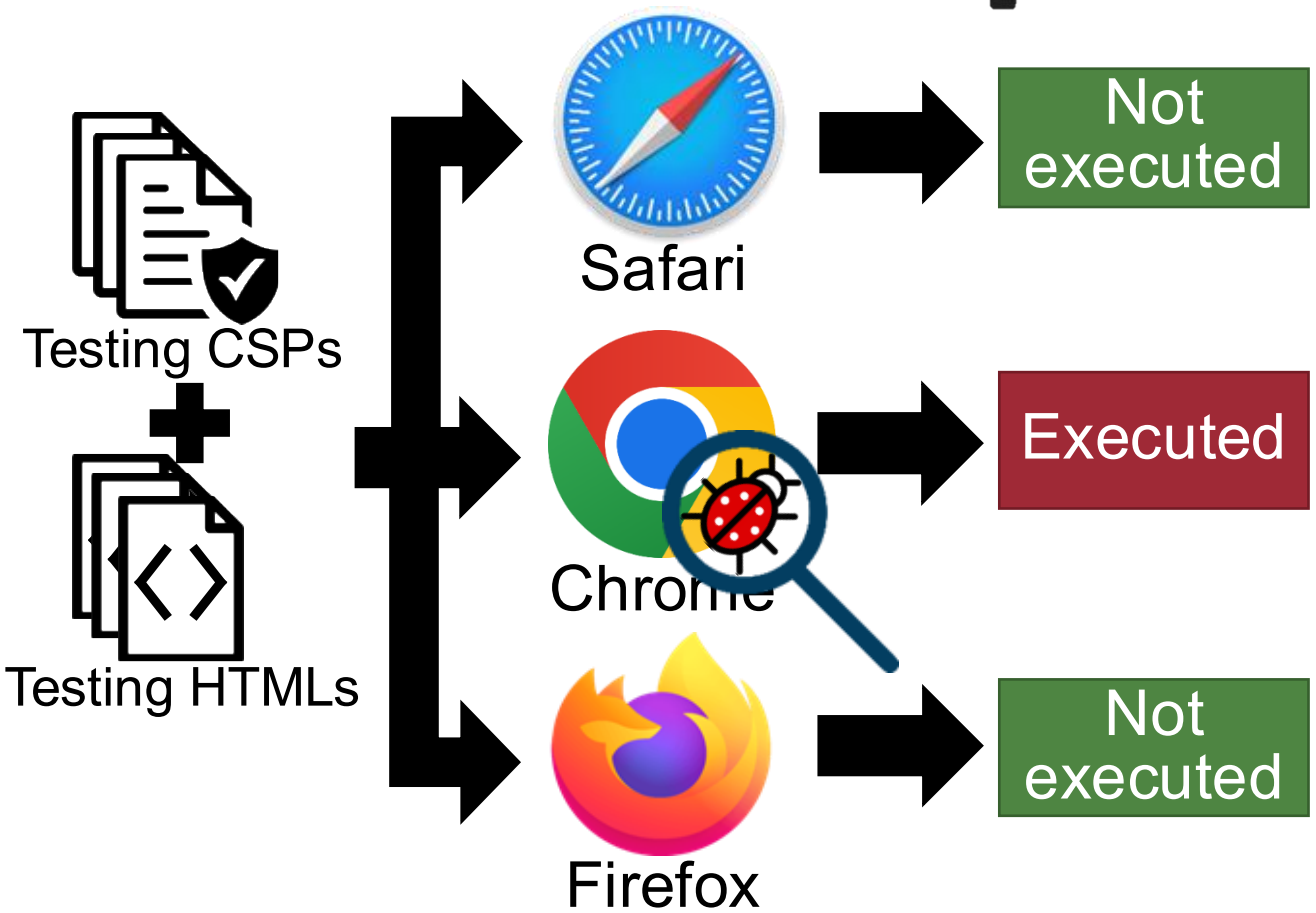
Our approach:
Differential testing!

(Ref) Finding CSP Enforcement



(Ref) Finding CSP Enforcement





Result

- Found **27 browser bugs** in Chrome, Safari, and FireFox
- …and **three description bugs**
- 23 bugs have been patched (5,000$ rewards)
- Published in *NDSS'23*

Recent Studies

- BUIzz, ***USENIX Sec '26***
- Head(er)s Up!, ***CCS '25***
- DiffCSP, ***NDSS '23***
- A Bug's Life, ***USENIX Sec '23***
- 12 angry developers, ***CCS '21***
- Complex security policy?, ***NDSS '20***
- CSP is dead, long live CSP!, ***CCS '16***

Conclusion

- Content Security Policy (CSP)
  - Allow resources which are trusted by the developer
- Many research on generating CSPs, deploying CSPs, and bypassing CSPs
- Even if CSP is deployed, very hard to get right
  - >90% of all policies in study of CSS 2016 easily bypassable
- **CSP is an improvement, but by no means of complete fix**

Question?