CSE551: Advanced Computer Security

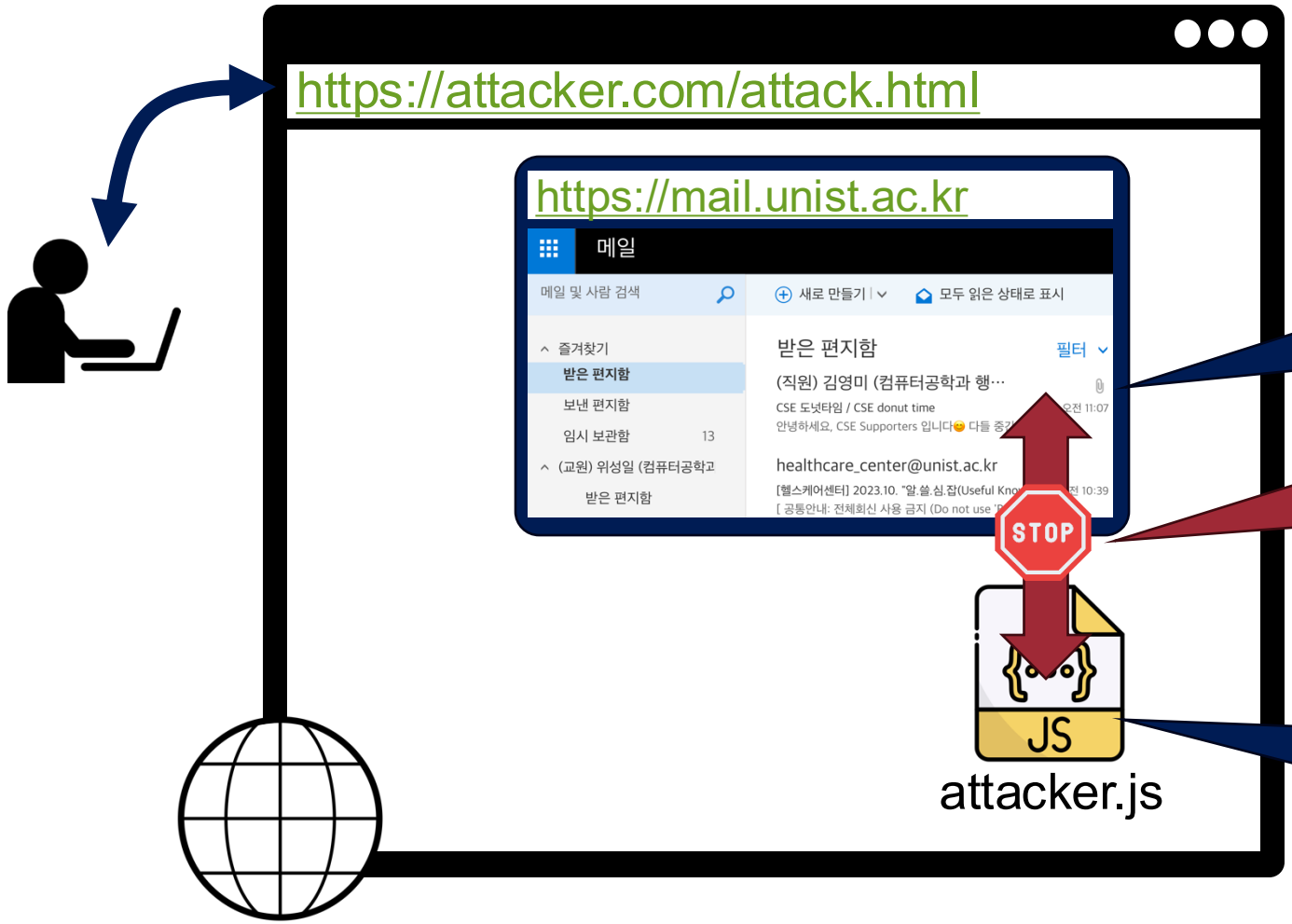
15. Content Security Policy

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**

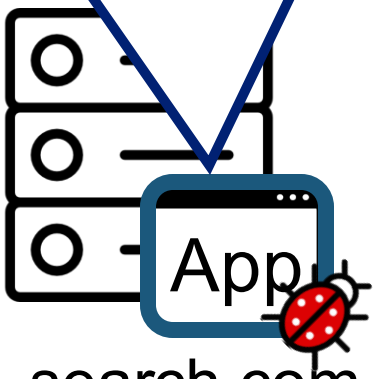
Recap: Cross-Site Scripting (XSS)

```
<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!

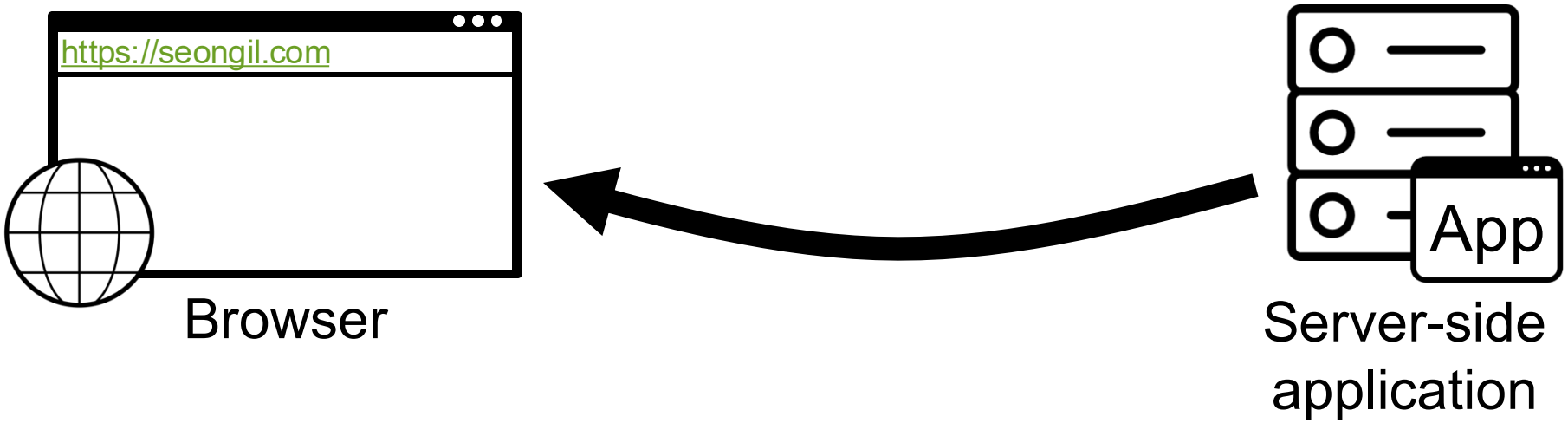
Content Security Policy (CSP)

Content Security Policy (CSP)

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

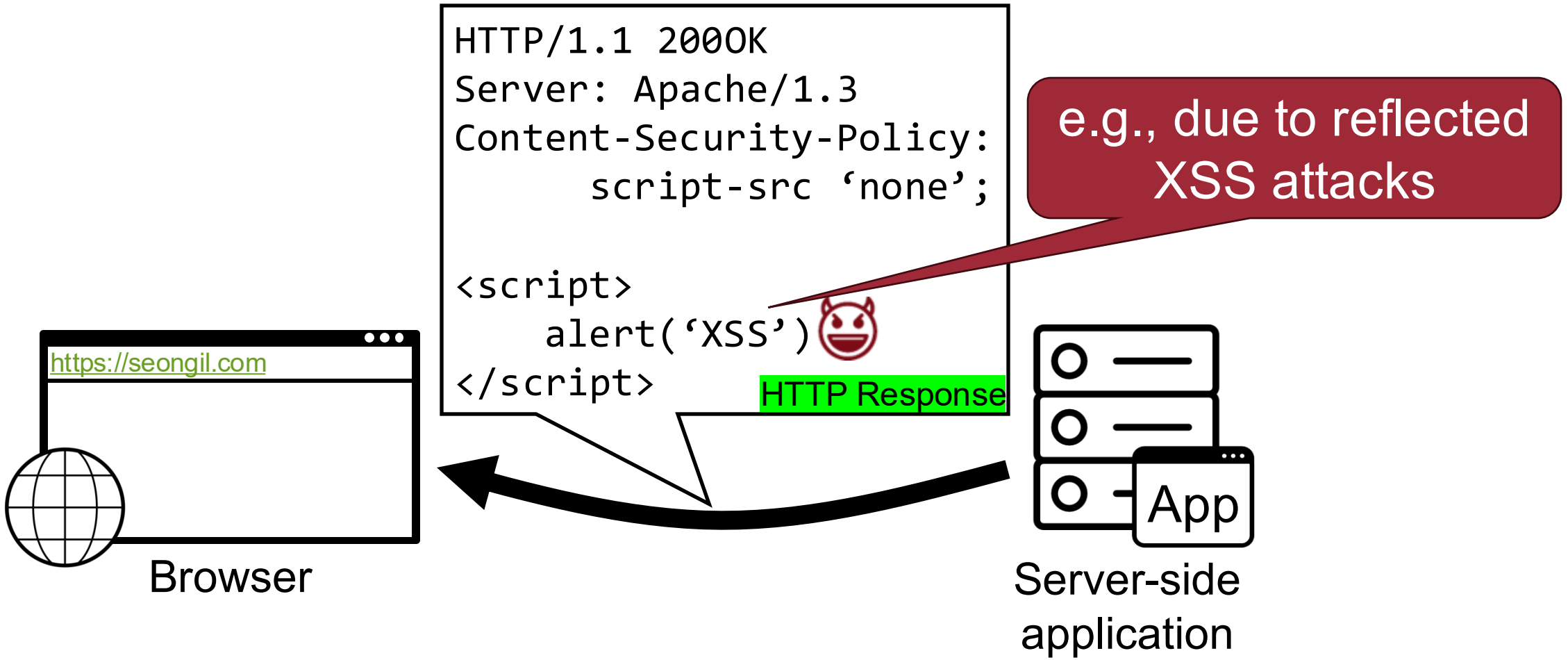
CSP Workflow

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



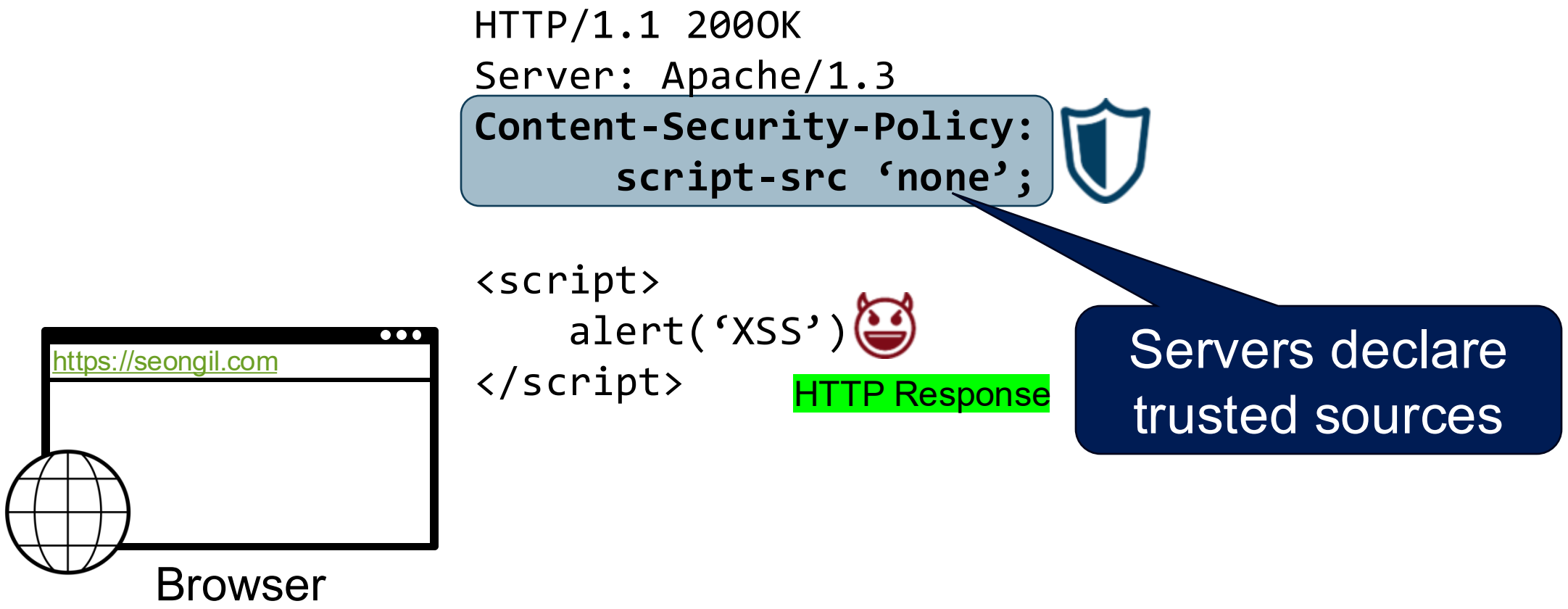
CSP Workflow

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



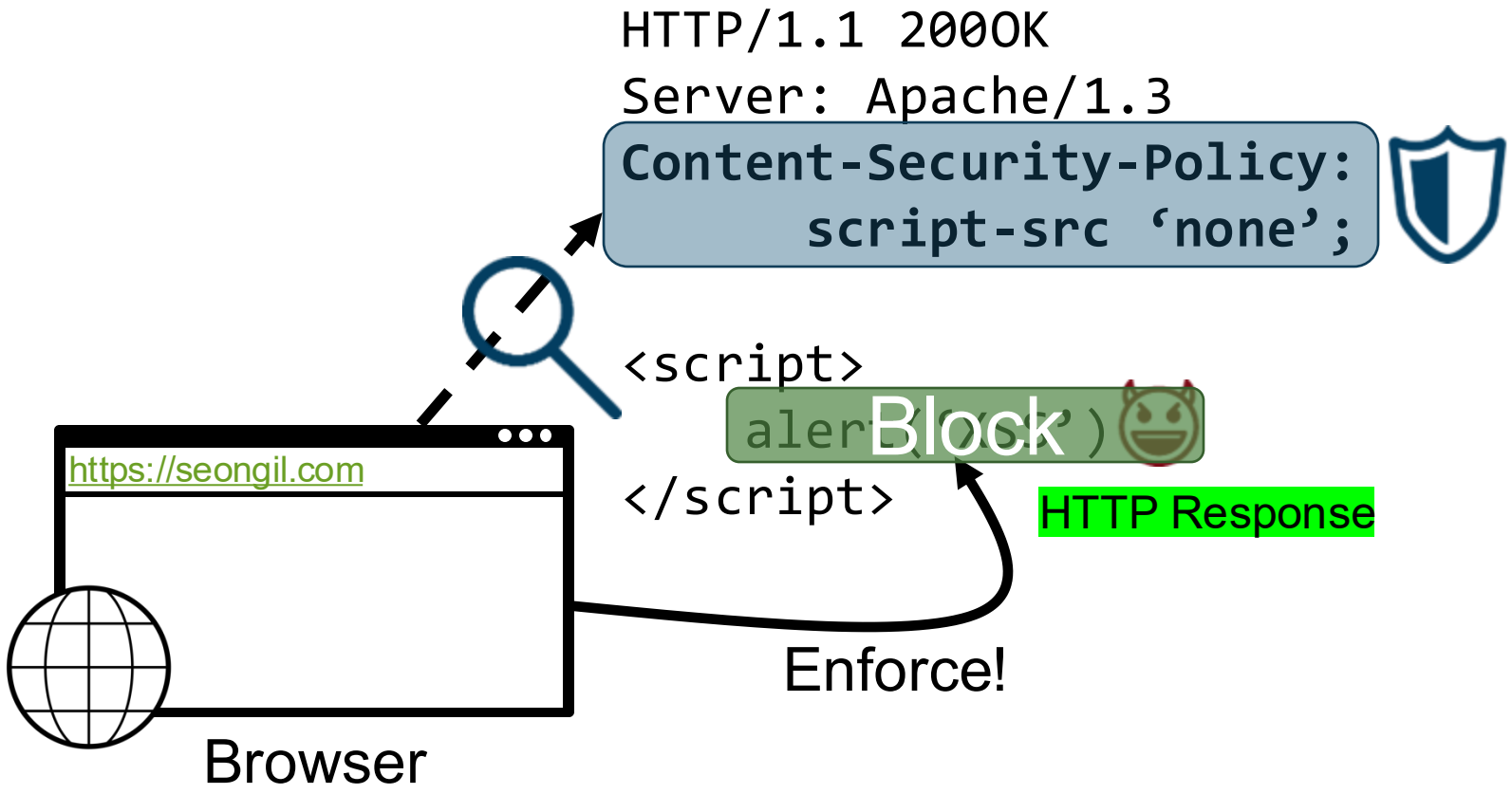
CSP Workflow

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



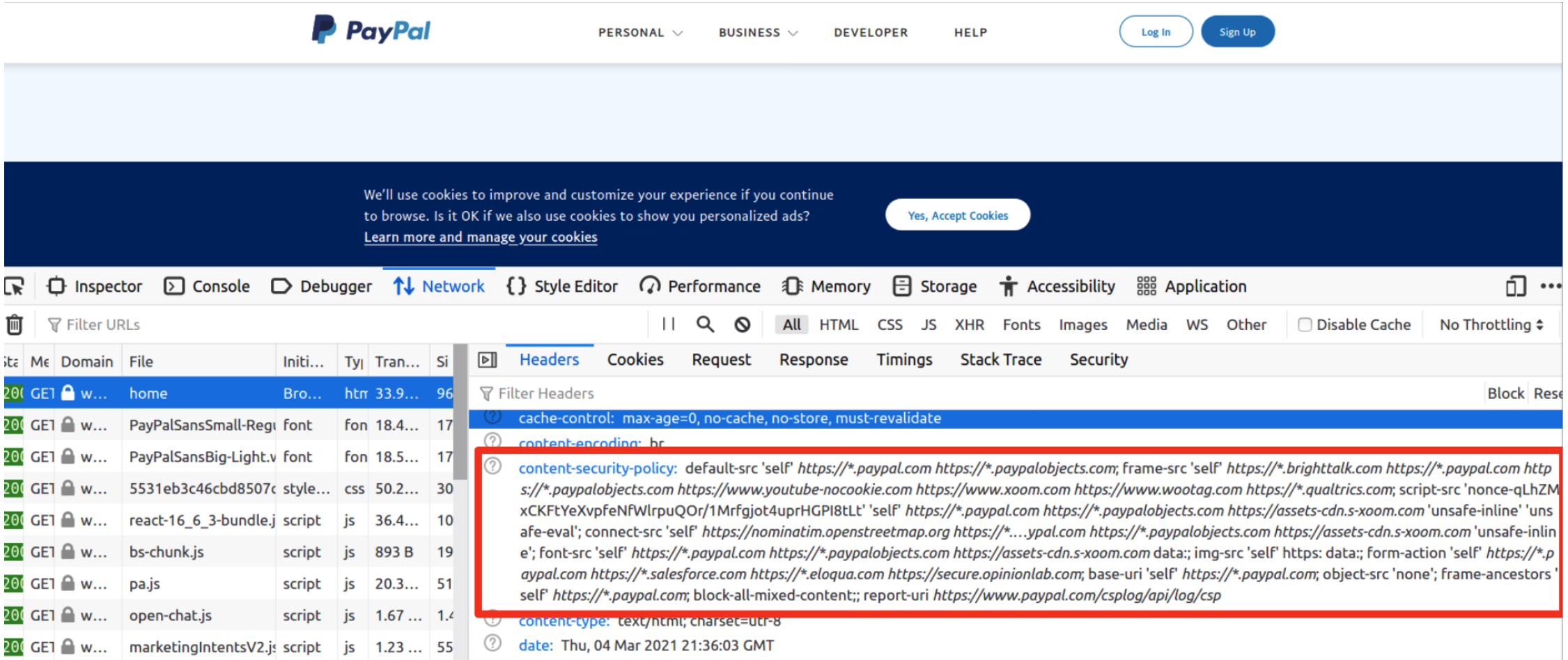
CSP Workflow

- 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

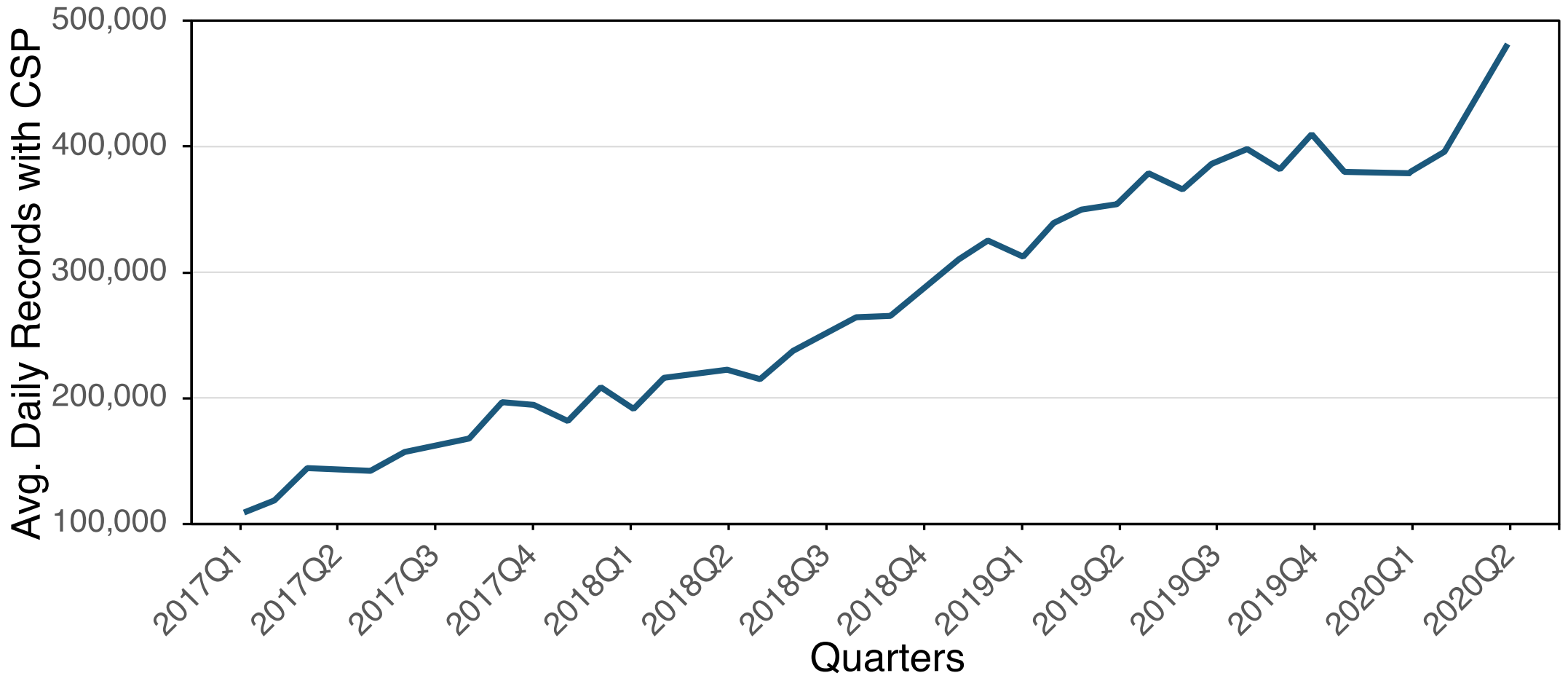
Firefox

`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 – Example

Executes scripts only from the same origin and https://unist.ac.kr

CSP for website https://example.com:

```
script-src ‘self’ https://unist.ac.kr;
```

HTML	Execution Allowed?
`<script src = “https://unist.ac.kr/myscript.js”></script>`	✓
`<script src = “https://example.com/stuff.js”></script>`	✓
`<script>alert(1)</script>  // inline script`	✗
`<script src = “https://ad.com/someads.js”></script>`	✗

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 – Exercise

CSP for website https://example.com:

```
default-src https://unist.ac.kr; script-src ‘unsafe-inline’;
                                  img-src ‘self’
```

HTML	Allowed?
`<script>alert(1)</script>  // inline script`	
`<img src=“https://example.com/logo.png”></img>`	
`<iframe src=“youtube.com/video1”></script>`	
`<script src = “https://unist.ac.kr/stuff.js”></script>`	

CSP Level 1 – Example and Limitations

CSP for website https://example.com:

```
<script src = “https://ad.com/someads.js”></script>
// ad.com will add stuff from company.com


<script>
// … some required inline script
</script>
```

```
Content-Security-Policy: script-src ‘self’
```

- will block any scripts added here

CSP Level 1 – Example and Limitations

CSP for website https://example.com:

```
<script src = “https://ad.com/someads.js”></script>
// ad.com will add stuff from company.com


<script>
// … some required inline script
</script>
```

JS from company.com will be rejected

```
Content-Security-Policy: script-src ‘self’ https://ad.com
```

- will block inline script
- … and script which was added by ad.com

CSP Level 1 – Example and Limitations

CSP for website https://example.com:

```
<script src = “https://ad.com/someads.js”></script>
// ad.com will add stuff from company.com


<script>
// … some required inline script
</script>
```

```
Content-Security-Policy: script-src ‘self’ https://ad.com
                         https://company.com
```

- will block inline script

CSP Level 1 – Example and Limitations

CSP for website https://example.com:

```
<script src = “https://ad.com/someads.js”></script>
// ad.com will add stuff from company.com


<script>
// … some required inline script
</script>
```

```
Content-Security-Policy: script-src ‘self’ https://ad.com
                         https://company.com ‘unsafe-inline’
```

- will allow inline script

CSP Level 1 – Example and Limitations

CSP for website https://example.com:

```
<script src = “https://ad.com/someads.js”></script>
// ad.com will add stuff from company.com
<script>// XSS attack!</script>
<script>
// … some required inline script
</script>
```

```
Content-Security-Policy: script-src ‘self’ https://ad.com
https://company.com ‘unsafe-inline’
```

- will allow inline script
- **… but allow XSS injection**

We need to avoid the use of ‘unsafe-inline’

CSP Level 1 – Example and Limitations

CSP for website https://example.com:

```
<script src = “https://ad.com/someads.js”></script>
// ad.com will add stuff from company.com
<script src =
  “https://example.com/myinlinescript.js”></script>
```

```
Content-Security-Policy: script-src ‘self’ https://ad.com
https://company.com
```

- requires removing inline script and converting it into an external script

CSP Level 1 – Example and Limitations

CSP for website https://example.com:

```
<script src = “https://ad.com/someads.js”></script>
// ad.com will add stuff from company.com
<script src =
  “https://example.com/myinlinescript.js”></script>
<button onclick=“meaningful()”>Click me</button>
```

```
Content-Security-Policy: script-src ‘self’ https://ad.com
https://company.com
```

- removing onclick handler is painful…

CSP Level 1 – Example and Limitations

CSP for website https://example.com:

```
<script src = “https://ad.com/someads.js”></script>
// ad.com will add stuff from company.com
<script src =
  “https://example.com/myinlinescript.js”></script>
<button id=meaningful()>Click me</button>
<script src =
  “https://example.com/eventhandler.js”></script>
```

```
var button = document.getElementById(“meaningful”);
button.onclick = meaningful;
```

```
Content-Security-Policy: script-src ‘self’ https://ad.com
https://company.com
```

- finally!

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

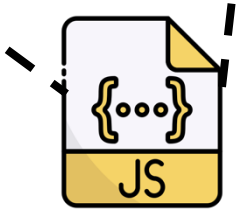
For each inline script…

```
<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

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>
```

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

appendChild is not “parser-inserted”

document.write is “parser-inserted”

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”
- Disables list of allowed hosts (such as `‘self’` and `‘unsafe-inline’`)

CSP Level 3 – Additional Changes

- `frame-src` undeprecated
  - Worker-src added to control workers specifically
  - Both fall back to child-src if absent (which falls back to default-src)

- manifest-src
  - Controls from where AppCache manifests can be loaded

CSP Level 3 – Backwards Compatibility


```
script-src https://ad.com ‘unsafe-inline’
‘nonce-d90e0153c074f6c3fcf53’ ‘strict-dynamic'
```

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

CSP Level 3 – Backwards Compatibility


```
script-src ~~https://ad.com 'unsafe-inline'~~
'nonce-d90e0153c074f6c3fcf53' 'strict-dynamic'
```

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

Modern browser
(CSP Level 3)

Ignores `unsafe-inline` and allowed hosts

CSP Level 3 – Backwards Compatibility


```
script-src https://ad.com ‘unsafe-inline’
‘nonce-d90e0153c074f6c3fcf53’ ‘strict-dynamic'
```

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

Modern browser (CSP Level 3)

Ignores `unsafe-inline` and allowed hosts

Old browser (CSP Level 1)

Ignores `strict-dynamic` and nonce, executes script through unsafe-inline and allowed hosts

CSP – Composition

- Browser always enforces all observed CSPs
  - Hence, CSP can never be relaxed, only tightened

- Useful for combatting XSS and restricting hosts at the same time
  - Idea: send two CSP headers, both will have to applied
    - Policy 1: `script-src ‘nonce-random’`
    - Policy 2: `script-src ‘self’ https://cdn.com`
  - Only nonced scripts can be executed (policy 1)
  - Only scripts from own origin and CDN can be executed (policy 2)
  - Result: only scripts that carry a nonce and are hosted on origin/CDN are allowed

CSP – Reporting Functionality

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

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

CSP – Report Only Mode

- Implementation of CSP is a tedious process
  - Removal of all inline scripts and usage of eval
  - Tricky when depending on third-party providers
    - E.g., advertisement includes random script (due to real-time bidding)

- Restrictive policy might break functionality
  - Remember: client-side enforcement
  - Need for (non-breaking) feedback channel to developers

- Content-Security-Policy-Report-Only
  - `default-src ….; report=uri /violations.php`
  - Allows to field-test without breaking functionality (reports current URL and causes for fail)
  - Does not work in meta element

Important!

- CSP does not stop XSS, tries to mitigate its effects
  - Similar to, e.g., the NX bit for stacks on x86/x64

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

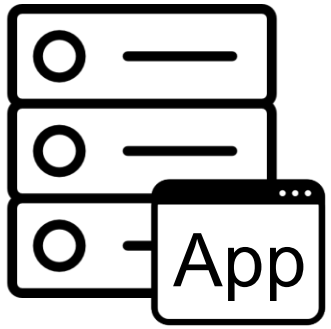
- Any allowed site with JSONP endpoint is potentially dangerous
  - E.g., `https://allowed.com/jsonp?callback="my malicious code here"//`

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 – 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-

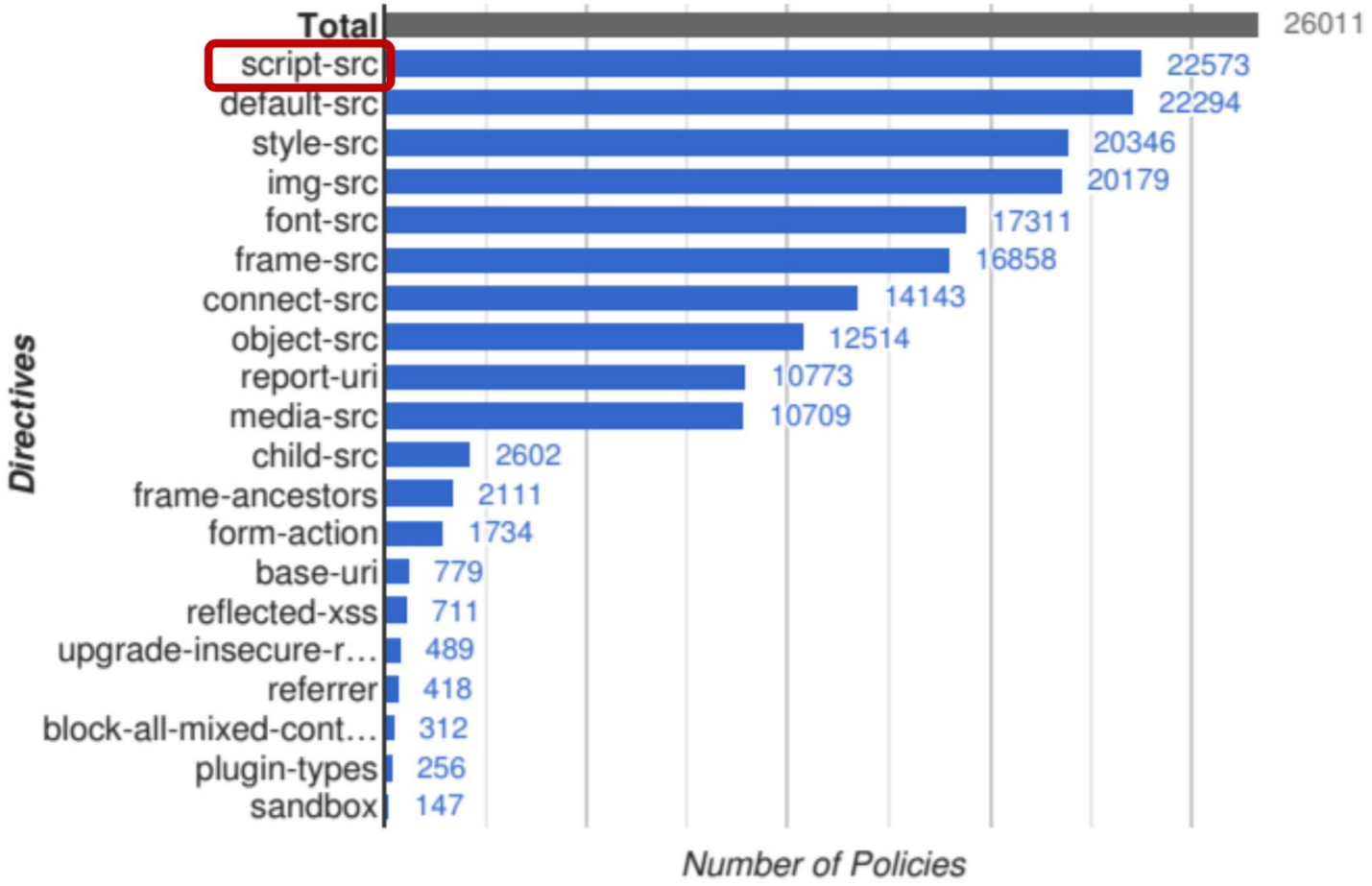
CSP Is Dead, Long Live CSP!, *CCS'16*

- 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 1.6 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

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

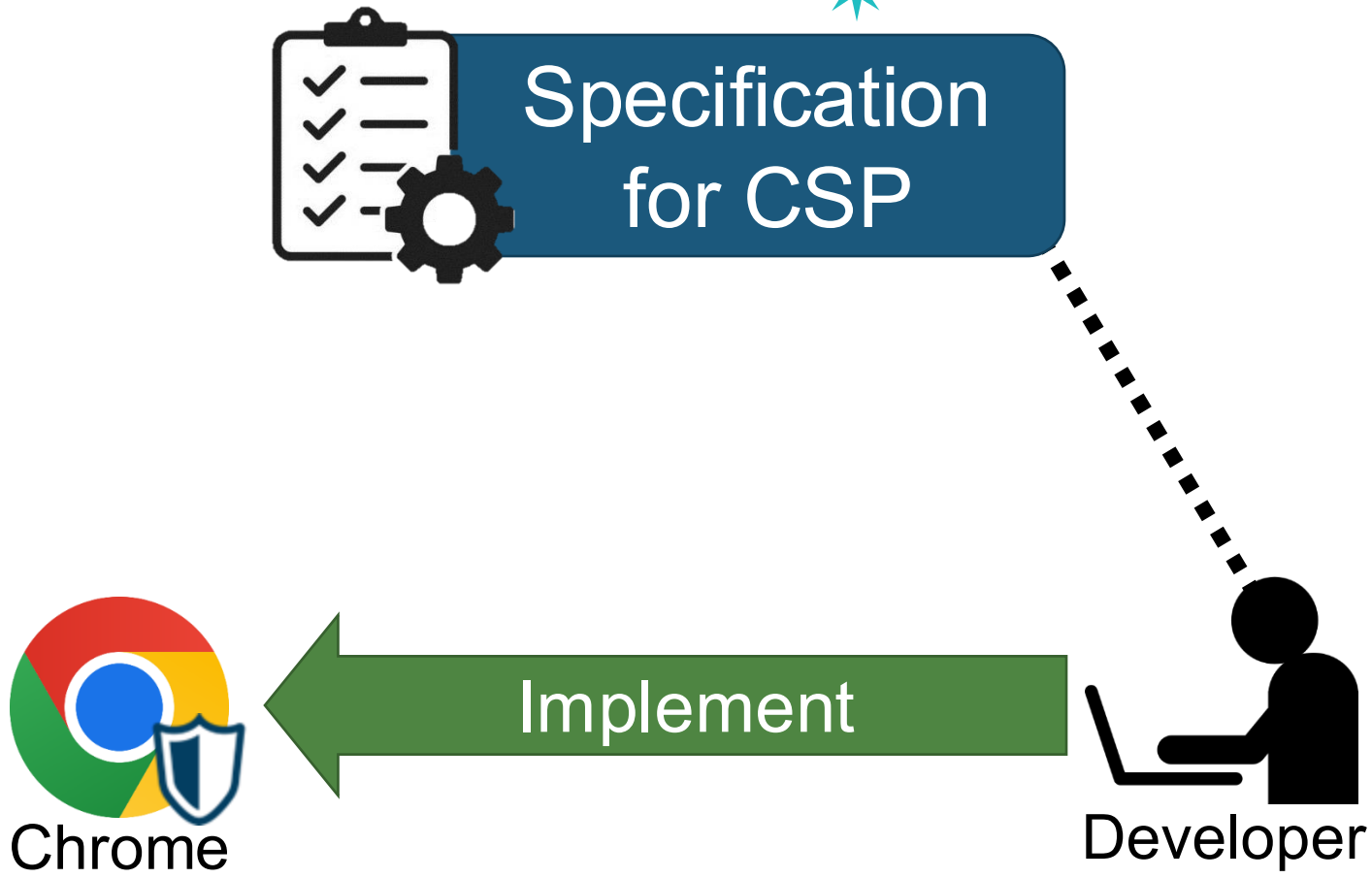
Seongil Wi*, Trung Tin Nguyen†‡, Jihwan Kim*, Ben Stock†, Sooel Son*

*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**

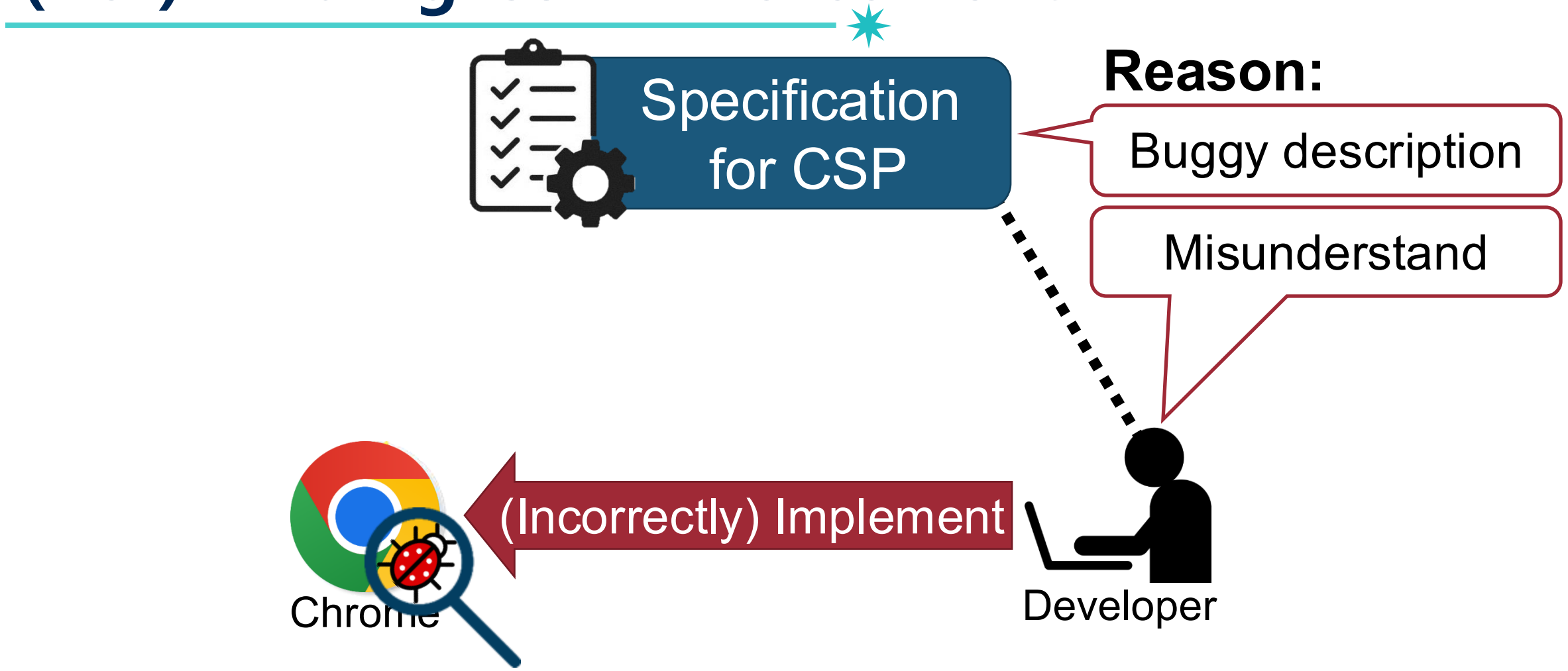
```
1  XSS attack payload:
2    http://[Target URL]/PoC.html#javascript:alert('XSS')
3  CSP: script-src-elem 'sha256-aHbTR...';
4  Target website:
5  <script>
```

(Ref) Finding CSP Enforcement

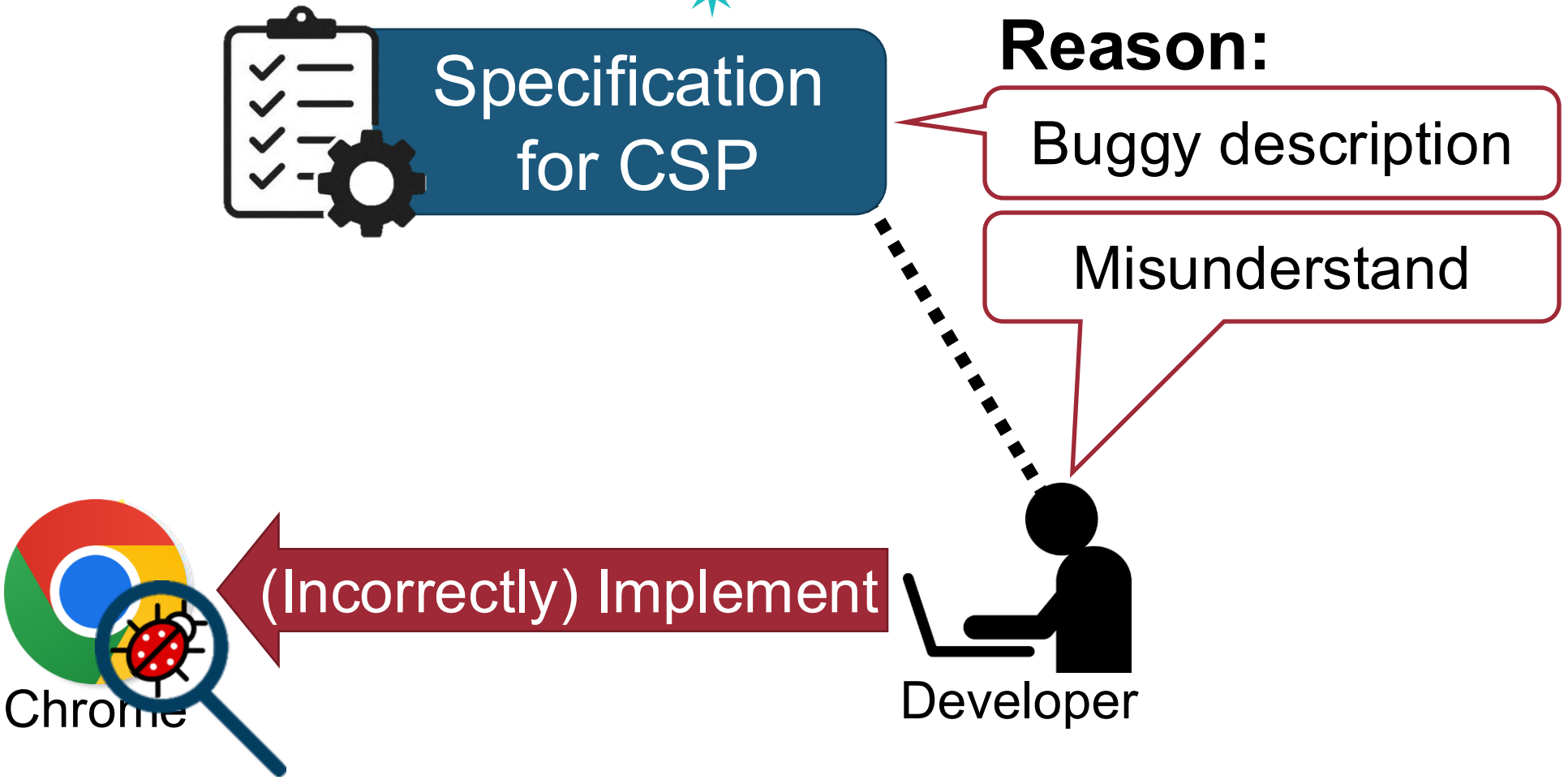


Specification for CSP: https://www.w3.org/TR/CSP3/

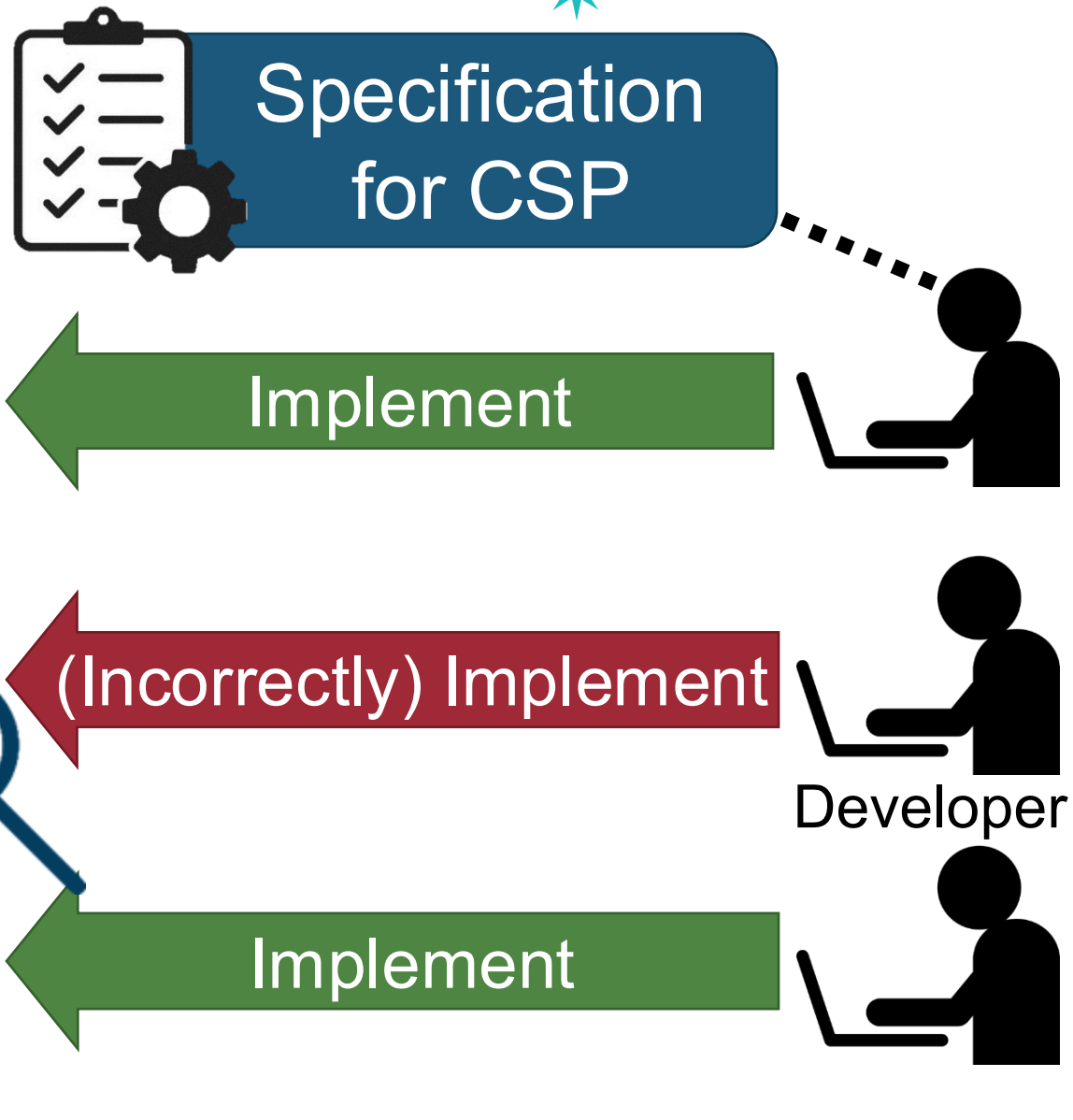
(Ref) Finding CSP Enforcement



(Ref) Finding CSP Enforcement

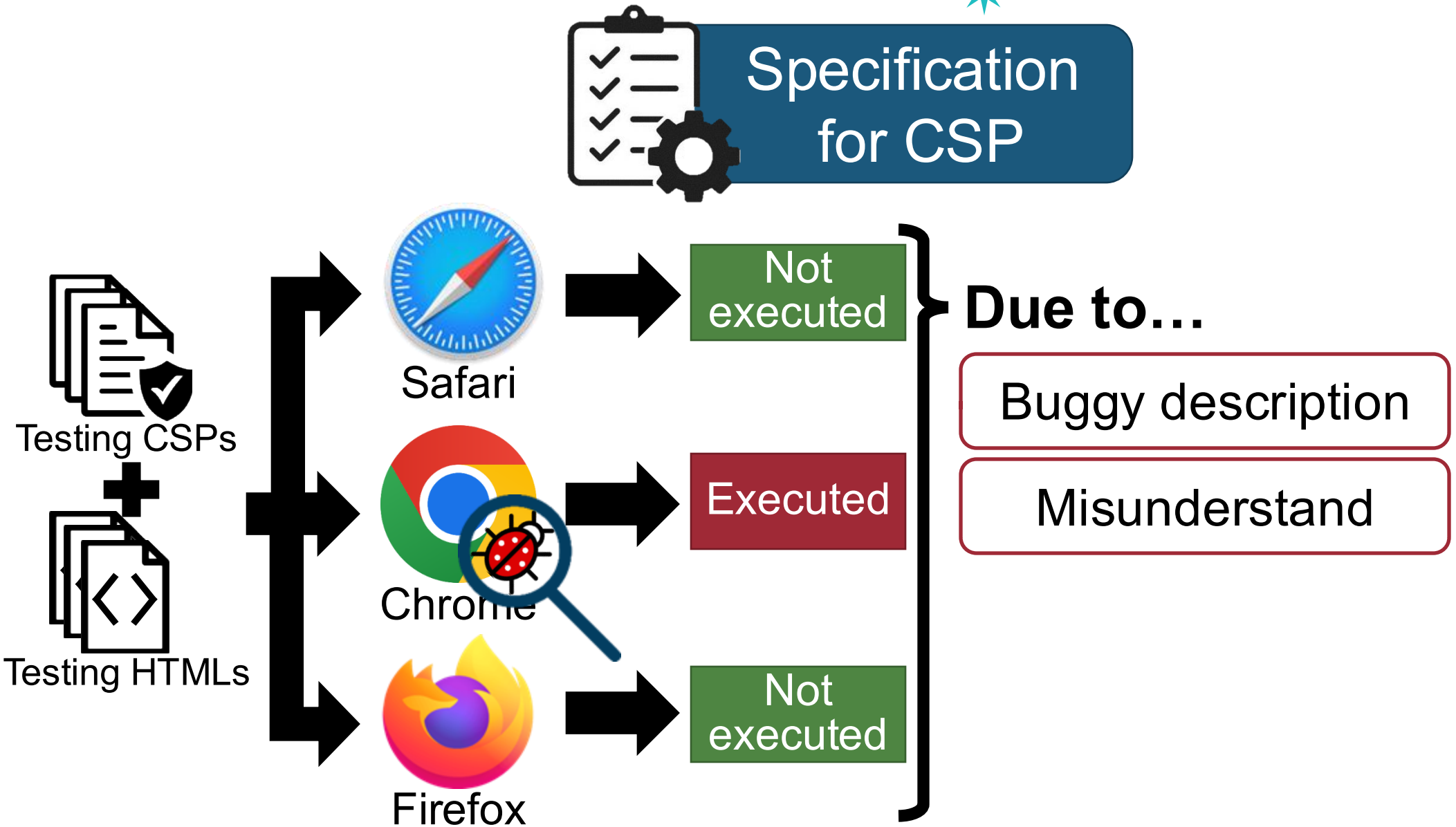


(Ref) Finding CSP Enforcement



Our approach:
Differential testing!

(Ref) Finding CSP Enforcement



Executing JS Snippets in Diverse Ways

Our intuition: we can find bugs by executing JS snippets in diverse ways

CVE-2020-6519

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

Allow!

CVE-2021-30538

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

Allow!

Grammar-based Input Generation

- Known CSP bugs
- XSS payloads
- ECMAScript spec
- HTML cheat sheet

Diverse ways of executing JS snippets

Grammar-based Input Generation

Grammar

[JS]

Derive all known forms of executing JS codes

Known CSP bugs

```
document.body.innerHTML+=
  “<script>
     eval(‘attack()’)
   </script>”
```

HTML cheat sheet

```
<iframe onload=
  attack()>
</iframe>
```

Diverse ways of executing JS snippets

Testing HTMLs

Grammar-based Input Generation

Known CSP bugs

```
document.body.innerHTML+=
  “[HTML]”
```

HTML cheat sheet

```
<iframe onload=
  attack()>
</iframe>
```

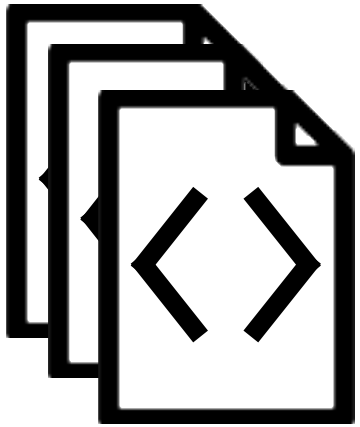
Grammar

[JS]

```
document.body.innerHTML+=
  “[HTML]”
```

[HTML]

Derive all known forms of executing JS codes



Testing HTMLs

Grammar-based Input Generation

Known CSP bugs

```
document.body.innerHTML+=
  “<script>
     eval(‘[JS]’)()’)
  </script>”
```

HTML cheat sheet

```
<iframe onload=
  attack()>
</iframe>
```

Grammar

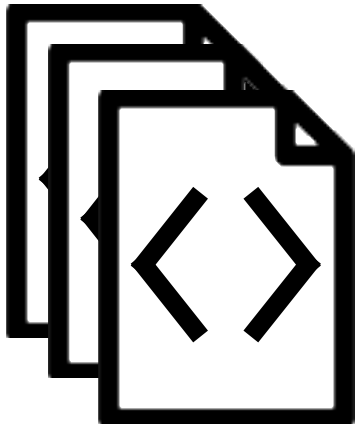
[JS]

```
document.body.innerHTML+=
  “[HTML]”
```

```
eval(‘[JS]’)
```

[HTML]

Derive all known forms of executing JS codes



Testing HTMLs

Grammar-based Input Generation

Known CSP bugs

```
document.body.innerHTML+=
  “<script>
     eval(‘attack()’)
   </script>”
```

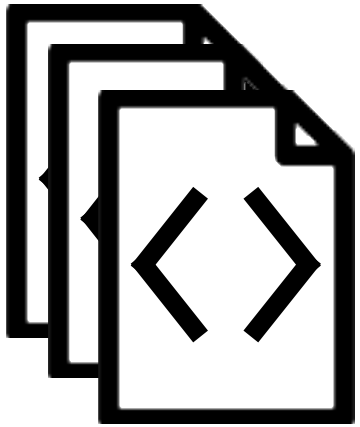
HTML cheat sheet

```
<iframe onload=
  attack()>
</iframe>
```

Grammar

[JS]
`document.body.innerHTML+= “[HTML]”`
`eval(‘[JS]’)`
`attack()`
[HTML]

Derive all known forms of executing JS codes



Testing HTMLs

Grammar-based Input Generation

Known CSP bugs

```
document.body.innerHTML+=
  “<script>
     eval(‘attack()’)
   </script>”
```

HTML cheat sheet

```
<iframe onload=
  [JS]>
</iframe>
```

Grammar

[JS]

```
document.body.innerHTML+=
  “[HTML]”
```

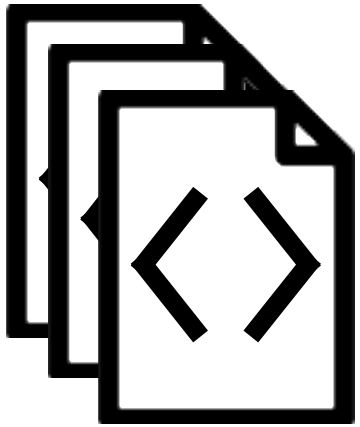
```
eval(‘[JS]’)
```

```
attack()
```

[HTML]

```
<iframe onload=
  [JS]>
</iframe>
```

Derive all known forms of executing JS codes



Testing HTMLs

Grammar-based Input Generation

Known CSP bugs

```
document.body.innerHTML+=
  "<script>
     [JS]('attack()')
   </script>"
```

HTML cheat sheet

```
<iframe onload=
  attack()>
</iframe>
```

Grammar

[JS]

```
document.body.innerHTML+=
  "[HTML]"
```

```
eval('[JS]')
```

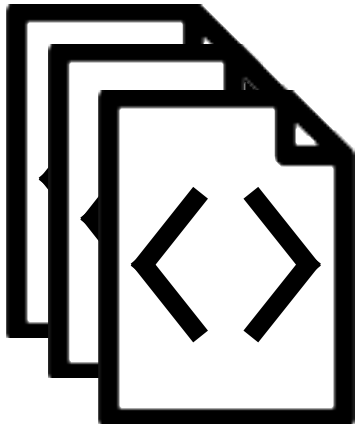
```
attack()
```

[HTML]

```
<iframe onload=
  [JS]>
</iframe>
```

```
<script>
  [JS]
</script>
```

Derive all known forms of executing JS codes



Testing HTMLs

Grammar-based Input Generation

Known CSP bugs

```
document.body.innerHTML+=
  “<script>
     eval(‘attack()’)
   </script>”
```

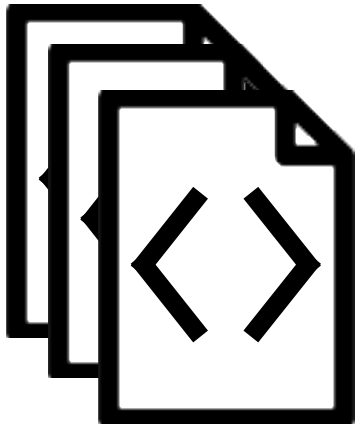
HTML cheat sheet

```
<iframe onload=
  attack()>
</iframe>
```

Grammar

[JS]
`document.body.innerHTML+= “[HTML]”`
`eval(‘[JS]’)`
`attack()`
[HTML]
`<iframe onload= [JS]> </iframe>`
`<script> [JS] </script>`

Derive all known forms of executing JS codes



Testing HTMLs

Grammar-based Input Generation

Known CSP bugs

```
document.body.innerHTML+=
  “<script>
     eval(‘attack()’)
   </script>”
```

HTML cheat sheet

```
<iframe onload=
  attack()>
</iframe>
```

Grammar

[JS]

```
document.body.innerHTML+=
  “[HTML]”
```

```
eval(‘[JS]’)
```

```
attack()
```

[HTML]

```
<iframe onload=
  [JS]>
</iframe>
```

```
<script>
  document.body.innerHTML+=
    “<iframe onload=
       eval(‘attack()’)>
     </iframe>”
</script>
```

Testing HTML #1

```
<iframe onload=
  eval(‘document.body.innerHTML+=
    “<script>
       attack()
     </script>”’)>
</iframe>
```

Testing HTML #2

Generate **25,880 HTML** instances

CSP Generation

CSP

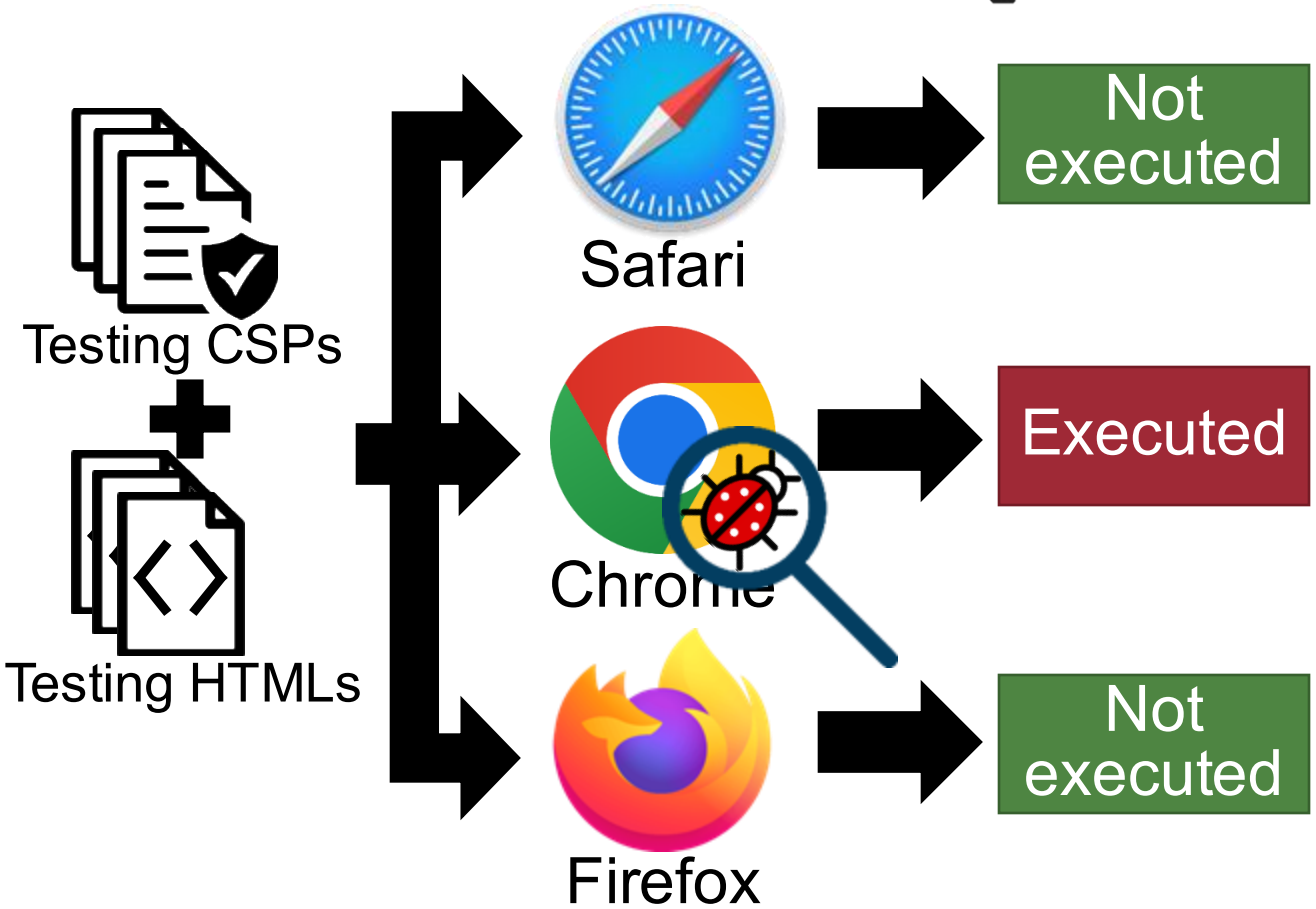
```
script-src  benign.com;
```

```
default-src,
script-src,
script-src-elem,
script-src-attr
```

Keyword	none, unsafe-inline, unsafe-eval, self, strict-dynamic, unsafe-hashes
Host-source	Self URL, Allowed URL, *
Schemes	data:, blob:, http:, https:
Nonce-source	nonce-123
Hash-source	sha256-[HASH]

Generate **1,006 policies**

(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*

Let's Split HTMP and Code

- CSP is valuable and effective to mitigate XSS attacks
- Deploying CSP to legacy web applications is a painful and difficult task

Why don't we make a tool to
automatically refactor the code?
=> deDacota, ***CCS '2013***

deDacota, CCS '2013

- Secure legacy web applications by automatically and statically rewriting an application so that the **code and data are separated**

deDacota: Toward Preventing Server-Side XSS via Automatic Code and Data Separation

Adam Doupé
UC Santa Barbara
adoupe@cs.ucsb.edu

Weidong Cui
Microsoft Research
wdcui@microsoft.com

Mariusz H. Jakubowski
Microsoft Research
mariuszj@microsoft.com

Marcus Peinado
Microsoft Research
marcuspe@microsoft.com

Christopher Kruegel
UC Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
UC Santa Barbara
vigna@cs.ucsb.edu

ABSTRACT

Web applications are constantly under attack. They are popular, typically accessible from anywhere on the Internet, and they can be abused as malware delivery systems.

Cross-site scripting flaws are one of the most common types of vulnerabilities that are leveraged to compromise a web application and its users. A large set of cross-site script-

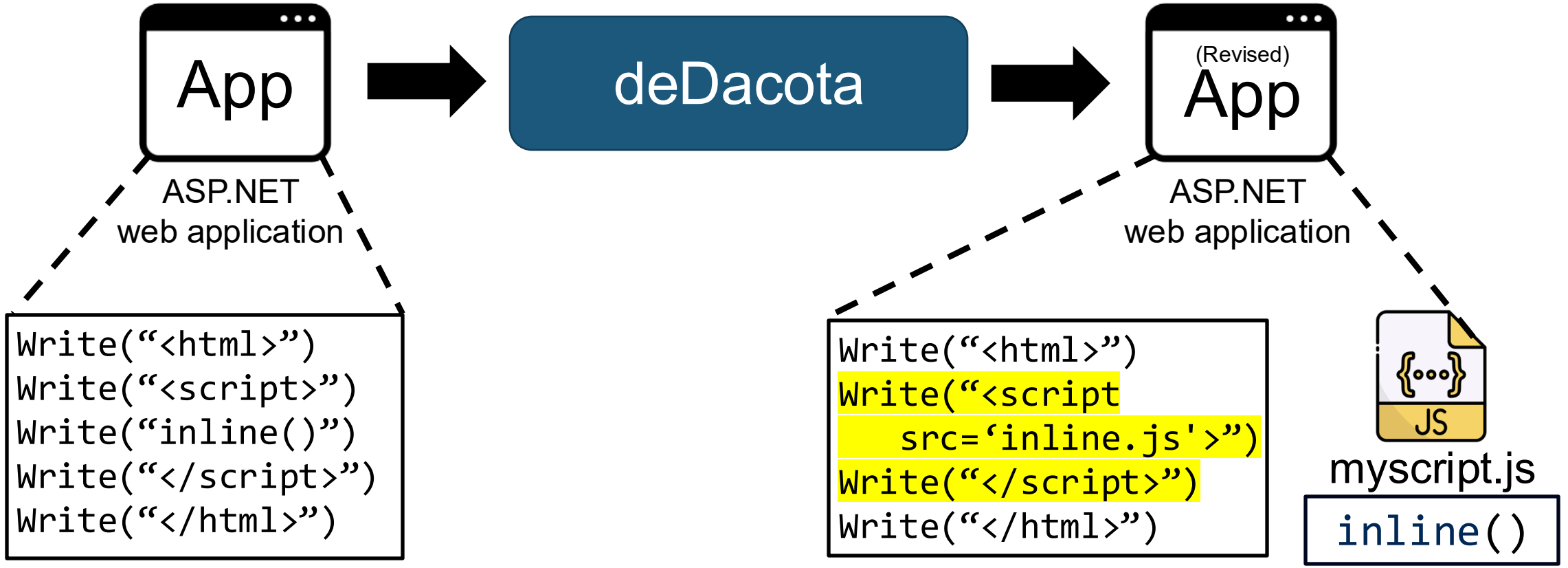
1. INTRODUCTION

Web applications are prevalent and critical in today's computing world, making them a popular attack target. Looking at types of vulnerabilities reported in the Common Vulnerabilities and Exposures (CVE) database [11], web application flaws are by far the leading class.

Modern web applications have evolved into complex pro-

deDacota, CCS '2013

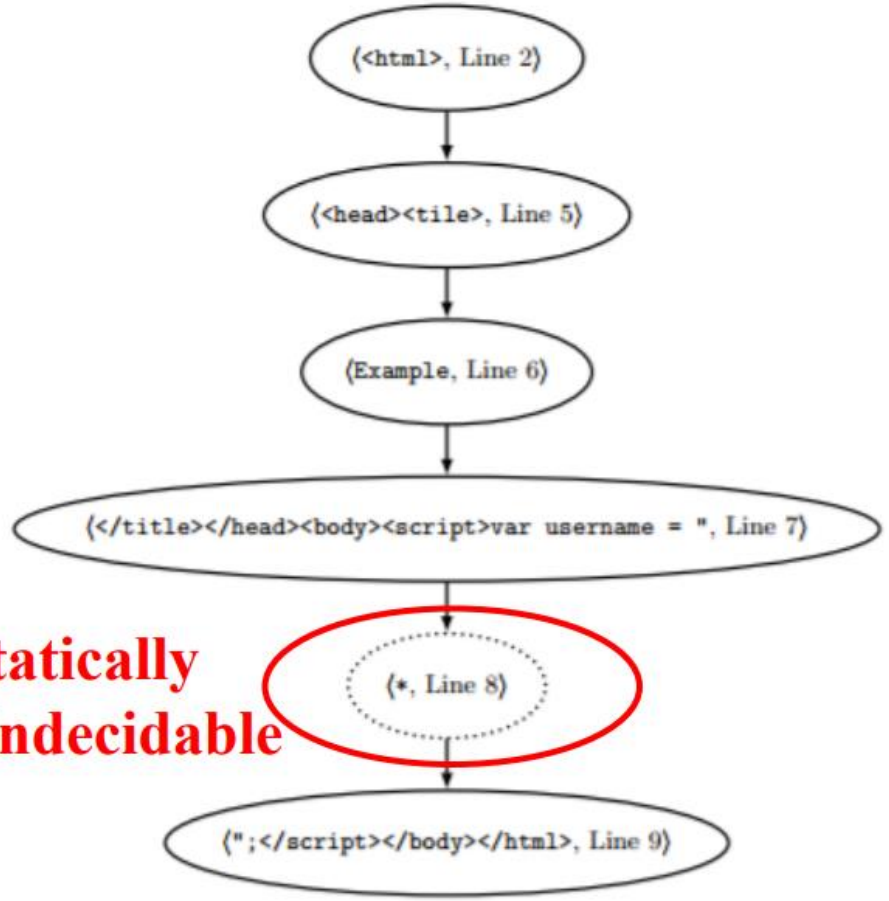
- Input: ASP.NET web application
- Output: Transformed web application that writes inlined JavaScript snippet to a file system



deDacota – Build an Graph

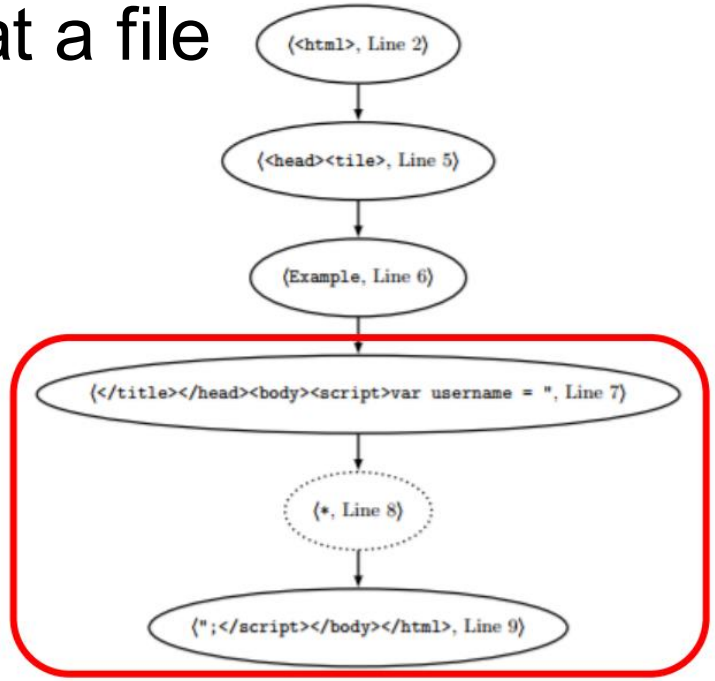
1. Identify all Write function and its constant string arguments
2. Identify the execution order of Write functions
3. Build an approximation graph

```
1  void Render(TextWriter w) {
2    w.Write("<html>\n  ");
3    this.Title = "Example";
4    this.Username = Request.Params["name"];
5    w.Write("\n  <head><tile>");
6    w.Write(this.Title);
7    w.Write("</title></head>\n  <body>\n
         <script>\n      var username = \"");
8    w.Write(this.Username);
9    w.Write("\";\n    </script>\n  </body>\n
         </html>");
10 }
```



deDacota – Convert Web Application

1. From all possible execution path in a graph, finds lines that emits `<script>` and `</script>`
2. Rewrites a program so that a string between `<script>` and `</script>` can be stored in the session buffer
3. At the closing </script>, the stored string at the session is written at a file




```
1  w.Write("</title></head>\n  <body>\n   ");
2
3  Session["7"] = "\n    var username = \"");
4  Session["7"] += this.Username;
5  Session["7"] += "\";\n    ";
6
7  var hashName = Hash(Session["7"]) + ".js";
8  WriteToFile(hashName, Session["7"]);
9
10 w.Write("<script src=\"" + hashName + "
       \"></script>");
11
12 w.Write("\n  </body>\n</html>");
```

deDacota – Limitations

- deDacota changes the server application logic
  - Auditing is hard whether the revised one is correct
  - Execution overhead to write inlined JS code into files

- deDacota is unable to block XSS on the script snippets that are dynamically changed by user inputs
  - Separated files will contain injected payloads

- How about `$script_starter = db_read(); write($script_starter)`? Instead of `write("<script>")`
  - False Negative

CSP – Other Use Cases

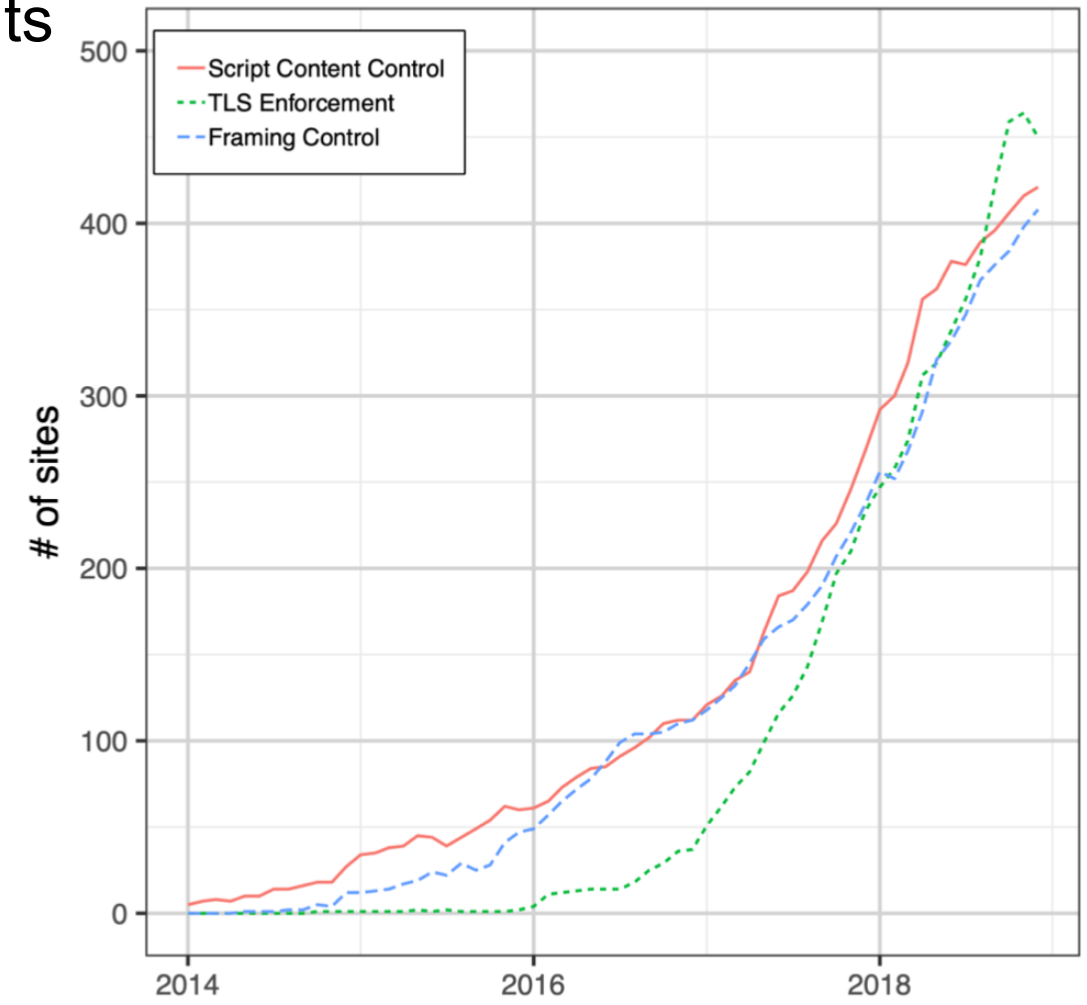
- Complex Security Policy?, ***NDSS '20***
  - Document the evolution of CSP and its use cases over time, showing its gradual move away from content restriction to other security goals

Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies

Sebastian Roth*, Timothy Barron†, Stefano Calzavara‡, Nick Nikiforakis†, and Ben Stock*
*CISPA Helmholtz Center for Information Security: {sebastian.roth,stock}@cispa.saarland
† Stony Brook University: {tbarron,nick}@cs.stonybrook.edu
‡ Università Ca' Foscari Venezia: calzavara@dais.unive.it

***Abstract*—The Content Security Policy (CSP) mechanism was developed as a mitigation against script injection attacks in 2010. In this paper, we leverage the unique vantage point of the Internet Archive to conduct a historical and longitudinal analysis of how CSP deployment has evolved for a set of 10,000 highly ranked domains. In doing so, we document the long-term struggle site operators face when trying to roll out CSP for content restriction and highlight that even seemingly secure whitelists can be bypassed through expired or typo domains. Next to these new insights, we also shed light on the usage of CSP for other use cases, in particular, TLS enforcement and framing control. Here, we find that CSP can be easily deployed to fit those security scenarios, but both lack wide-spread adoption.**

Though the (in)effectiveness of CSP has been analyzed and debated in several research papers [6, 8, 50, 51], CSP is still under active development and is routinely adopted by more and more Web sites: the most recent study [8] observed an increase of one order of magnitude in CSP deployment in the wild between 2014 and 2016. Notably though, virtually all papers have focused on CSP as a means to restrict content and have treated its newly added features (such as TLS enforcement and framing control) as side notes. To close this research gap and holistically analyze CSP it is important to take a critical look at how CSP deployment has evolved over time, so as to understand for which purposes developers use CSP and how



Recent Studies

- DiffCSP, ***NDSS '23***
- 12 angry developers, ***CCS '21***
- Complex security policy?, ***NDSS '20***
- CSP is dead, long live CSP!, ***CCS '16***
- Reining in the web with CSP, ***WWW '10***
- CCSP, ***USENIX Security '17***
- CSPAutoGen, ***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?