CSE467: Computer Security

16. Client-side Web Security (2)

Seongil Wi

Department of Computer Science and Engineering

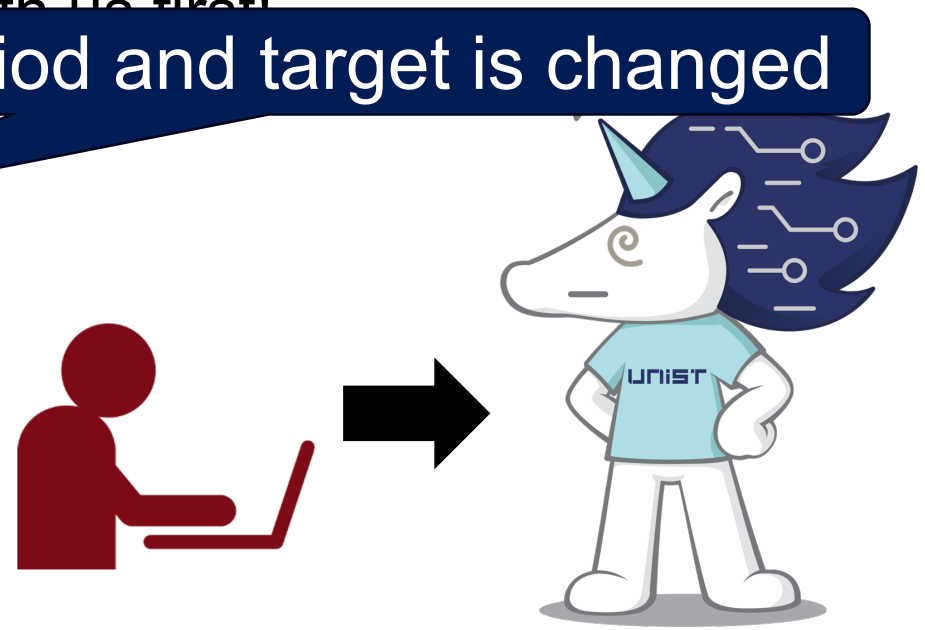
Activity #1: SaveUNIST

- **If you find unknown security problems on campus, report them to me!**

- Depending on the severity, bonus points will be given
  - E.g., A+ → 1 letter grade up → 10% score up → ... → 1 drink → ...

- **(IMPORTANT!) DO NOT** try anything illegal
  - If you cannot decide by yourself, discuss it with us first!

- **Period (tentative)**: Nov 20 ~ Dec 8
- **Target**: UNIST IST hompage
  (https://ist.unist.ac.kr/)



Today's Topic!

- **Network attacker**: resides somewhere in the communication link between client and server
  - Passive: evasdropping
  - Active: modification of messages, replay…

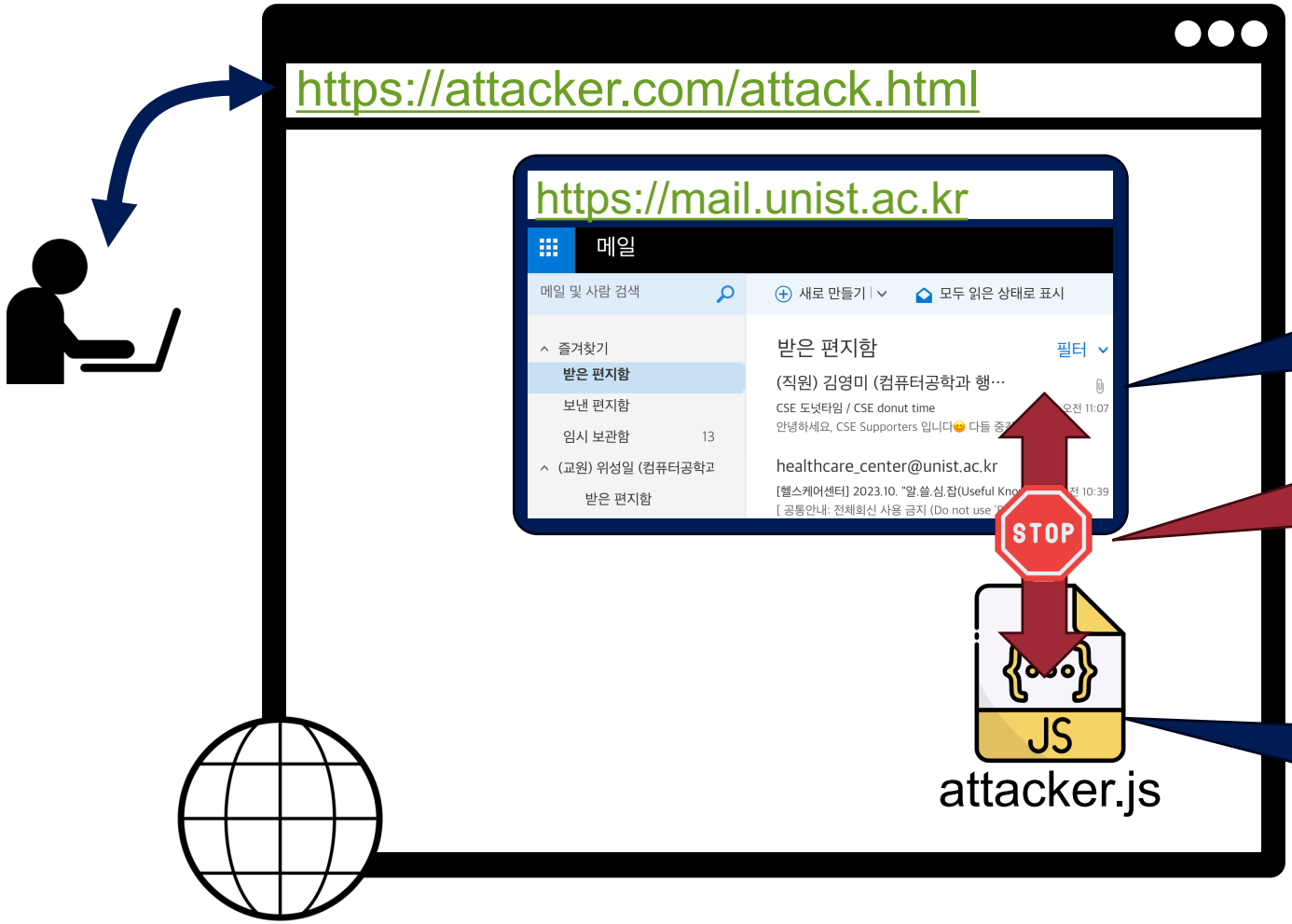
- **Remote attacker:** can co
  the network
  - Mostly targets the serv

Client-side web attack

- **Web attacker**: controls attacker.com
  - Can obtain SSL/TLS certificates for attacker.com
  - Users can visit attacker.com

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**

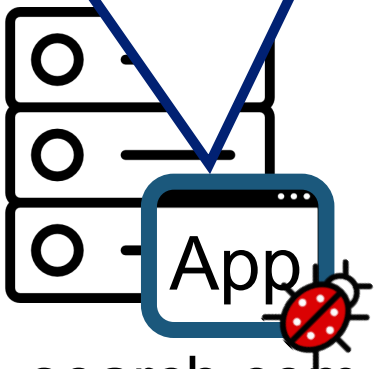
Recap: Cross-Site Scripting (XSS)

```
<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
- Next lecture!

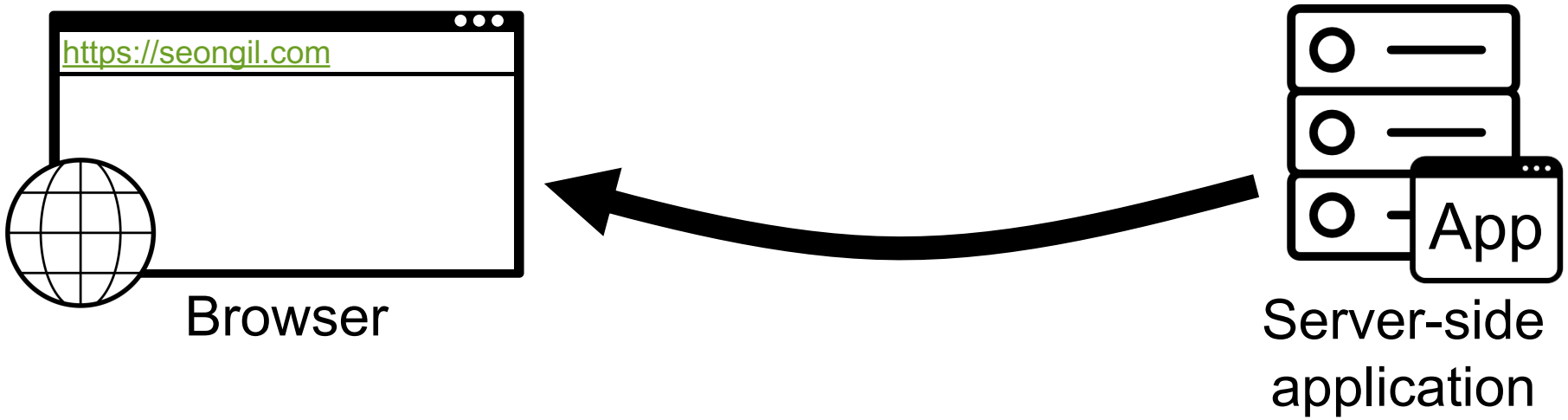
Content-Security Policy (CSP)

Content Security Policy (CSP)

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

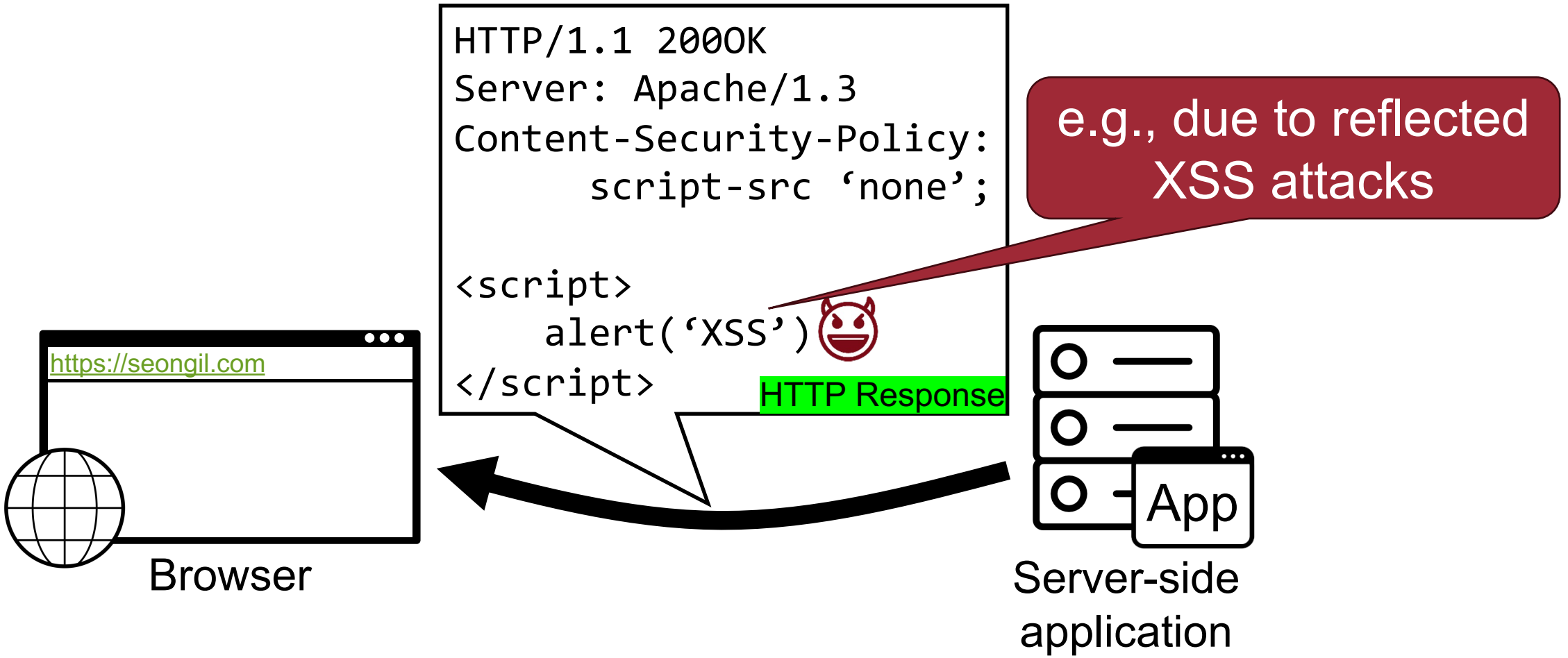
CSP Workflow

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



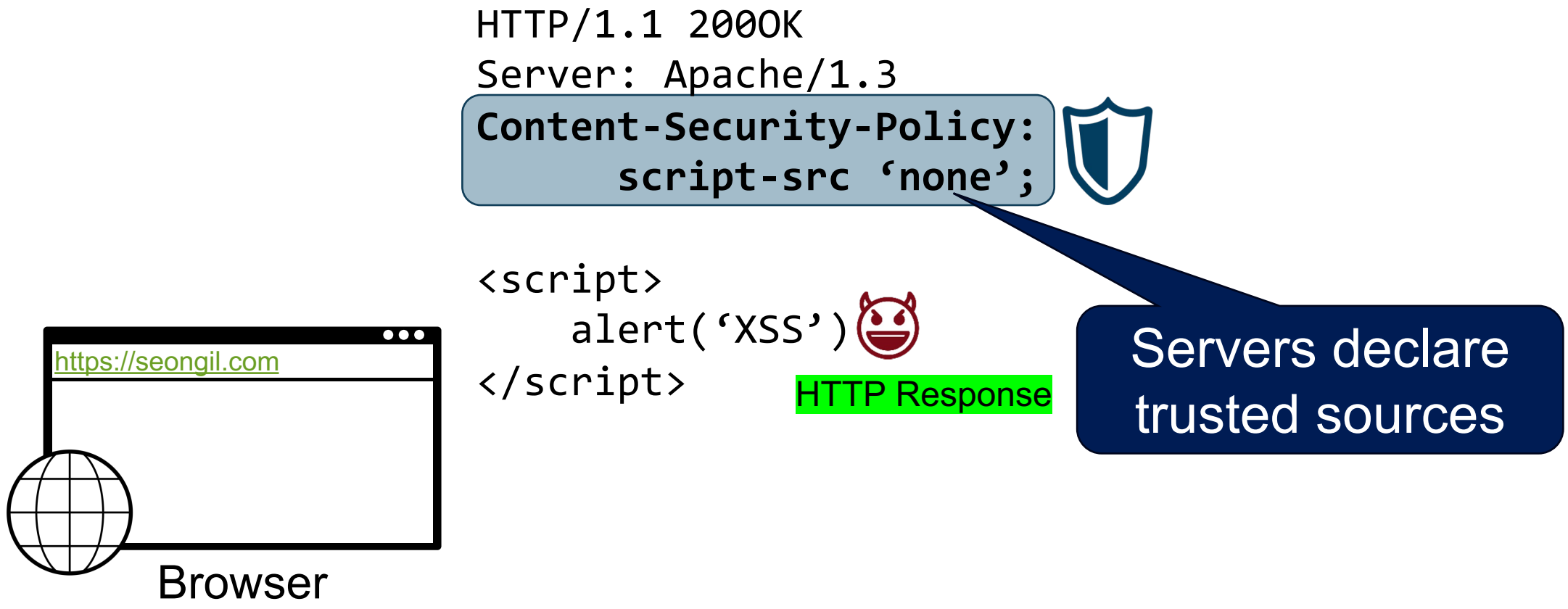
CSP Workflow

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



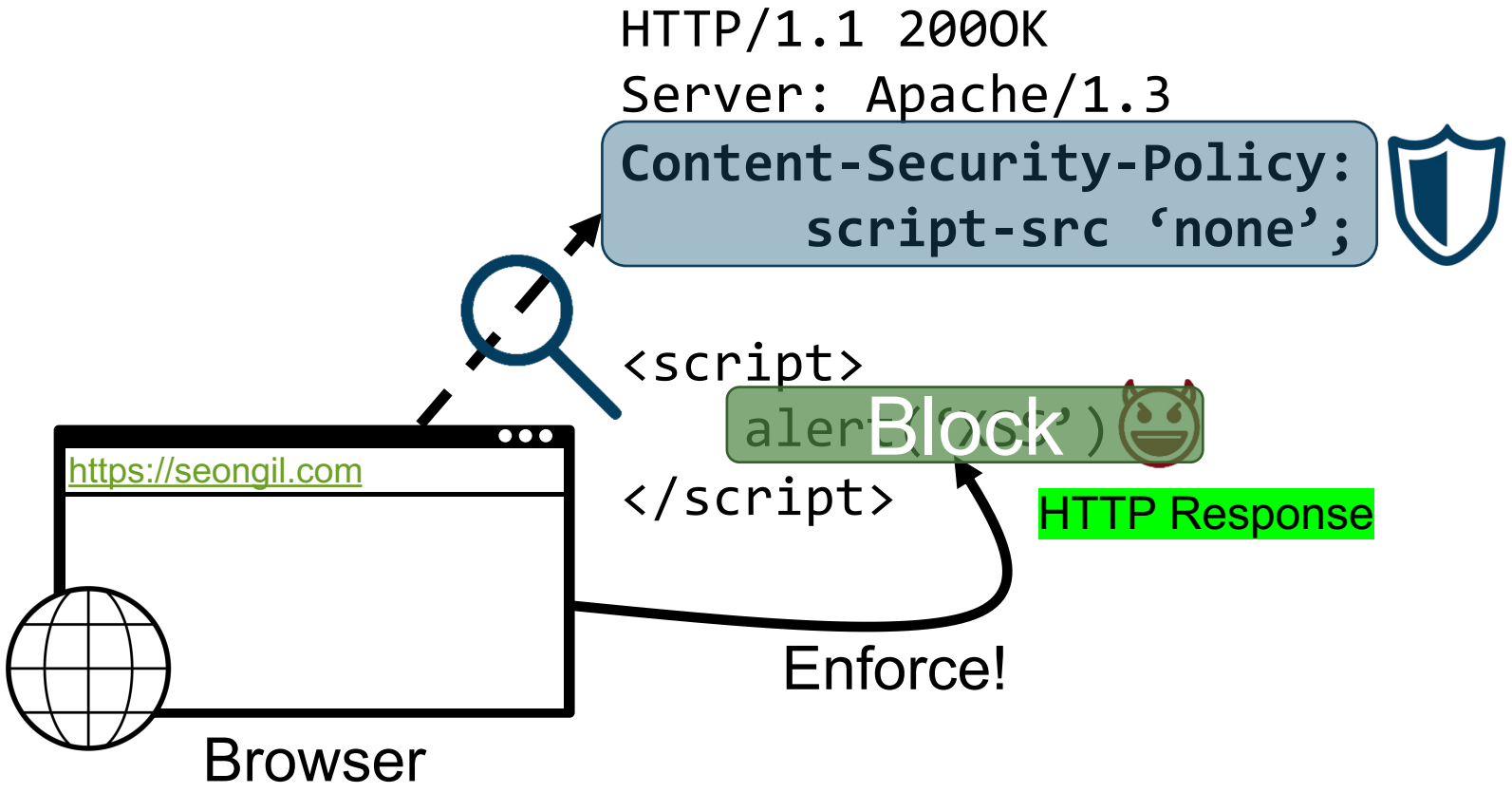
CSP Workflow

- 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

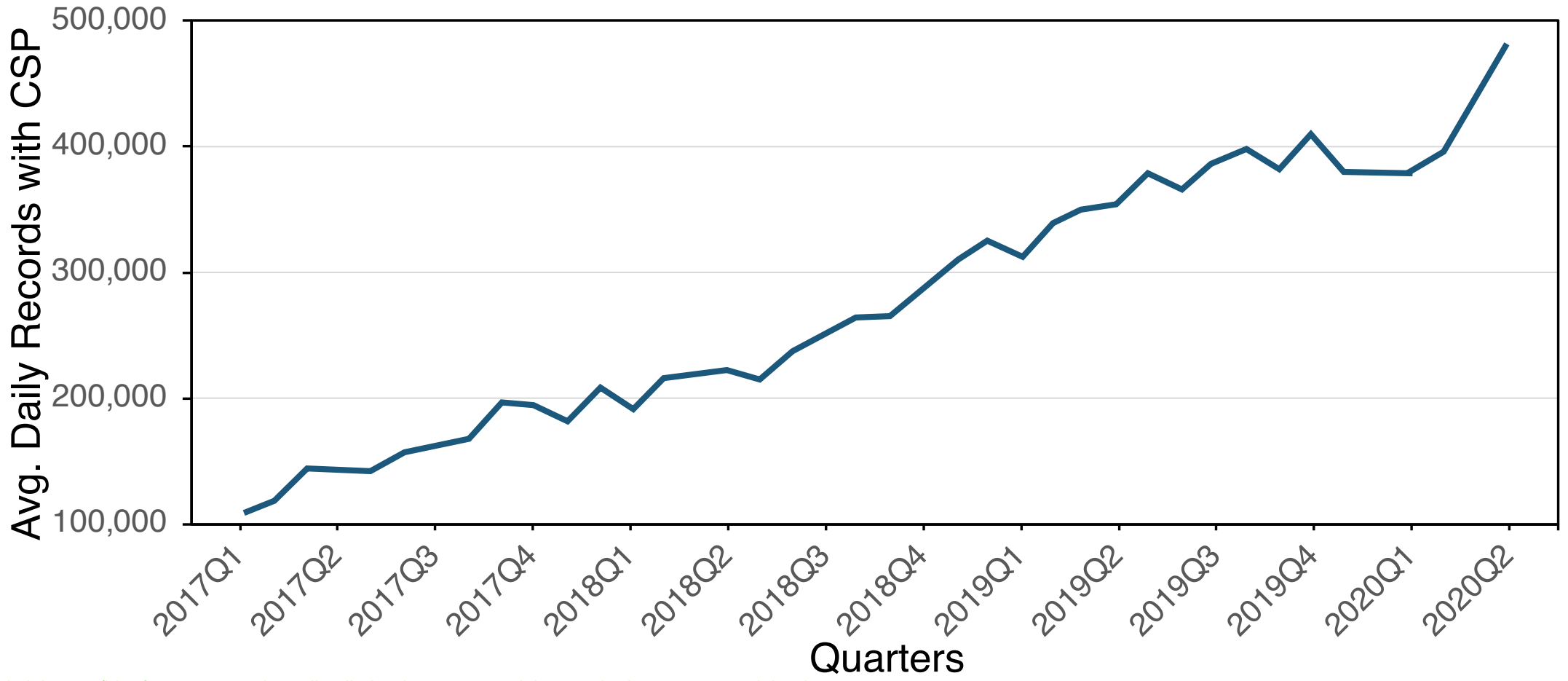
Firefox

- `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 – 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

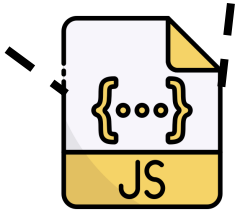
For each inline script…

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

converting

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

```
alert(1)
```



myscript.js

- Proposed improvement in CSP Level 2: **nonces and hashes**

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 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 – Limitations


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

```
<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!

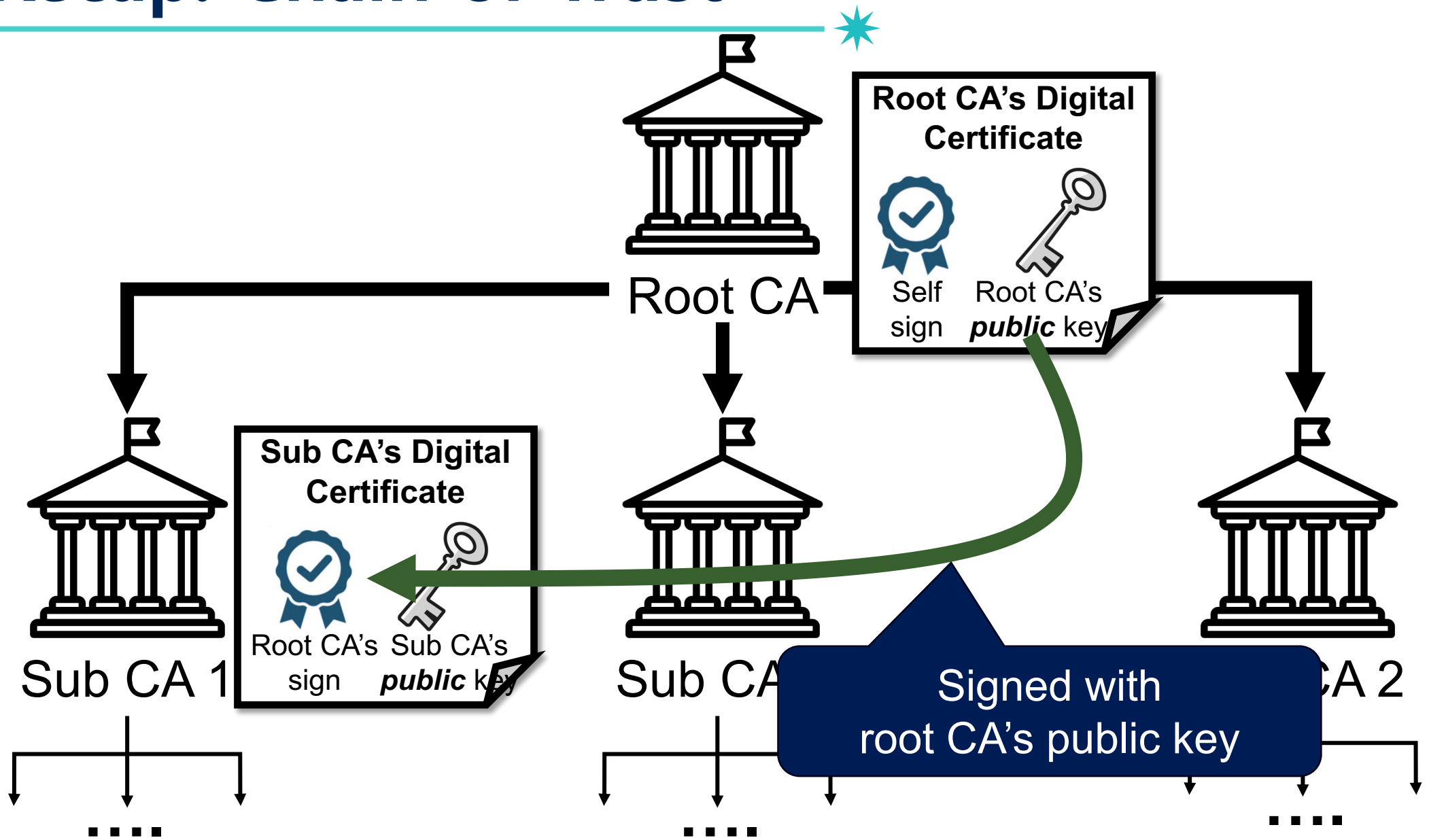
Changes from Level 2 to Level 3: Strict-dynamic

- 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

CSP – Bypasses

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

- 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 – Bypasses

- Does not stop XSS, tries to mitigate its effects
  - Similar to, e.g., the NX bit for stacks on x86
- 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 – Bypasses

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

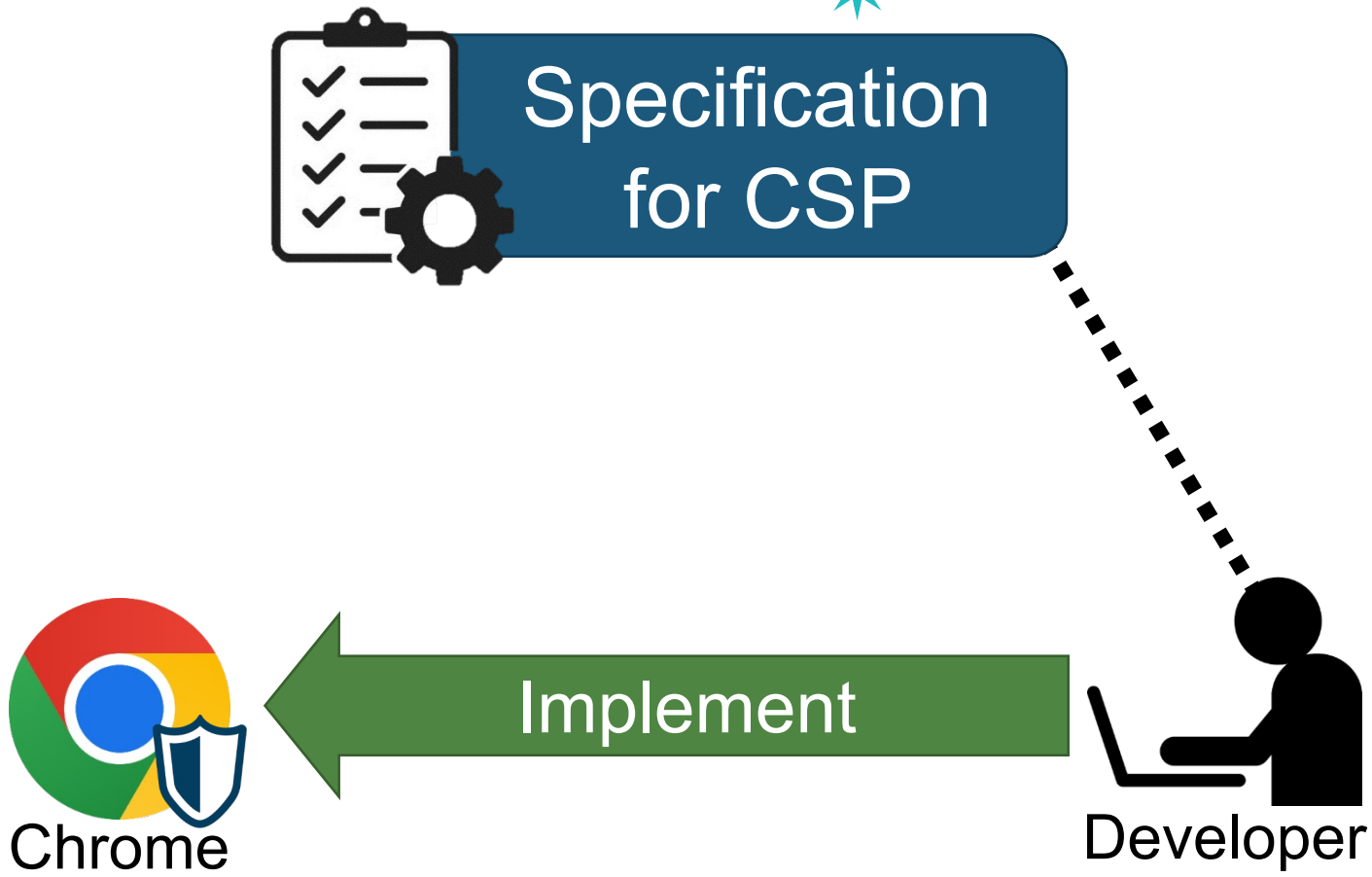
 X Not execute JS code

Safari

 X Not execute JS code

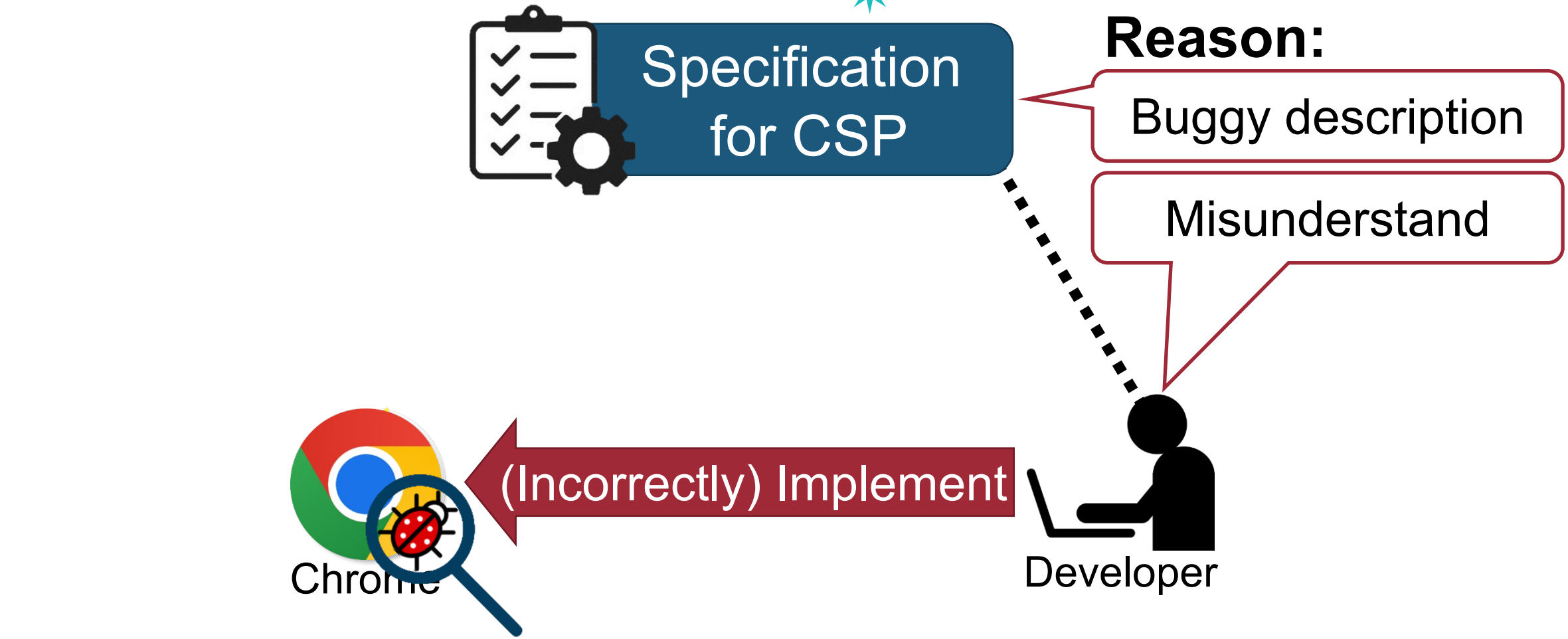
Firefox

(Ref) Finding CSP Enforcement

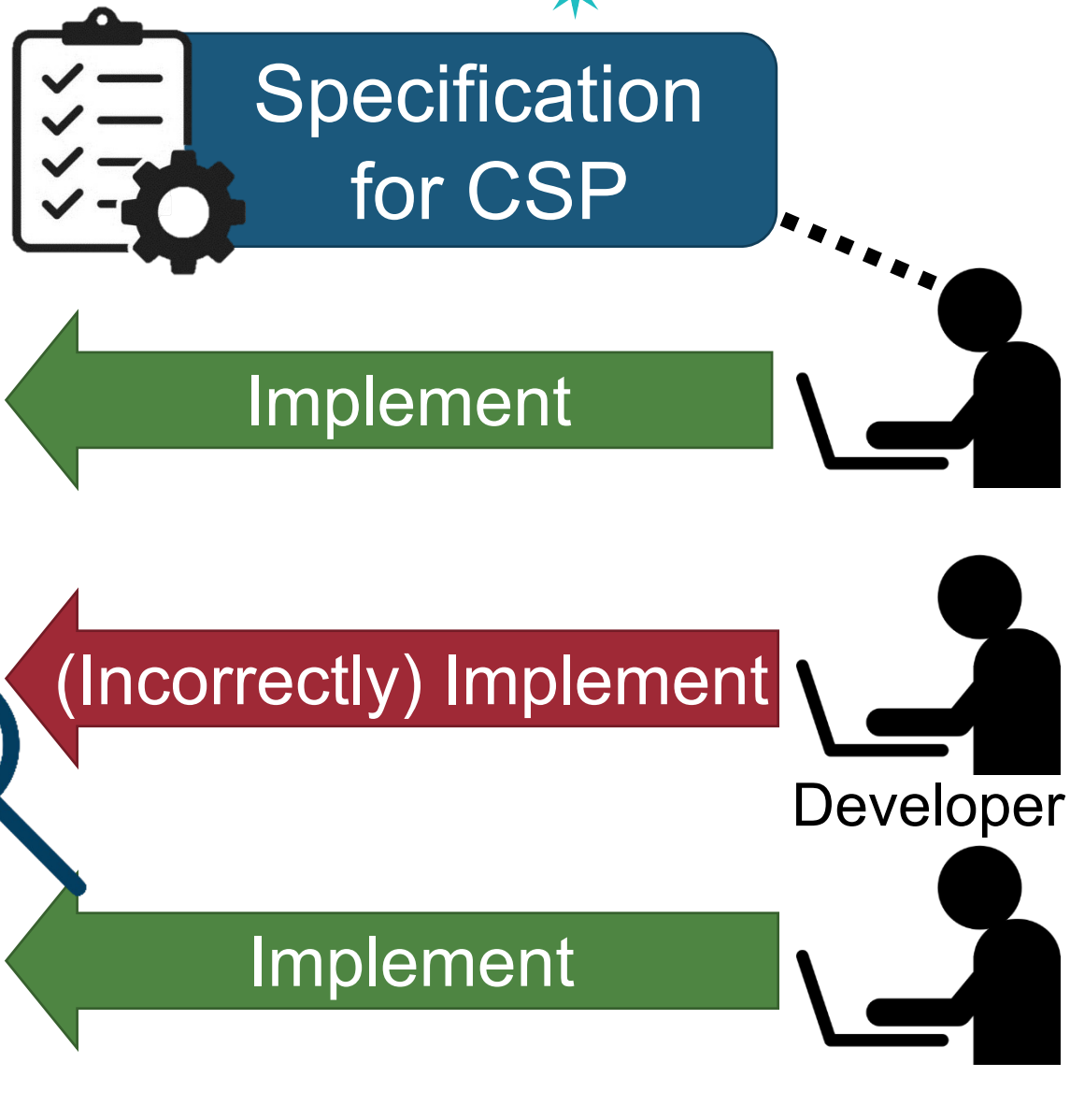


Specification for CSP: https://www.w3.org/TR/CSP3/

(Ref) Finding CSP Enforcement

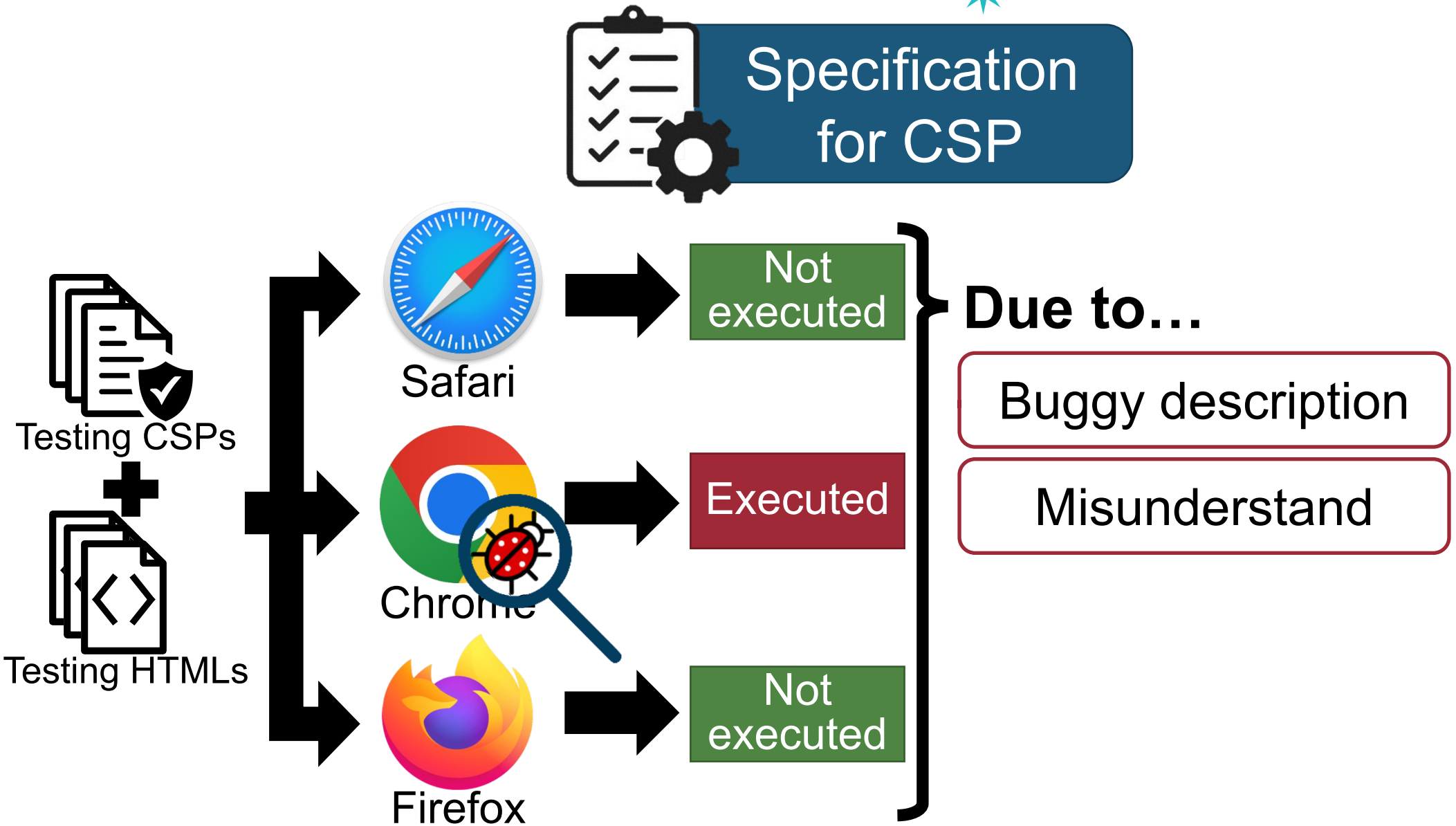


(Ref) Finding CSP Enforcement

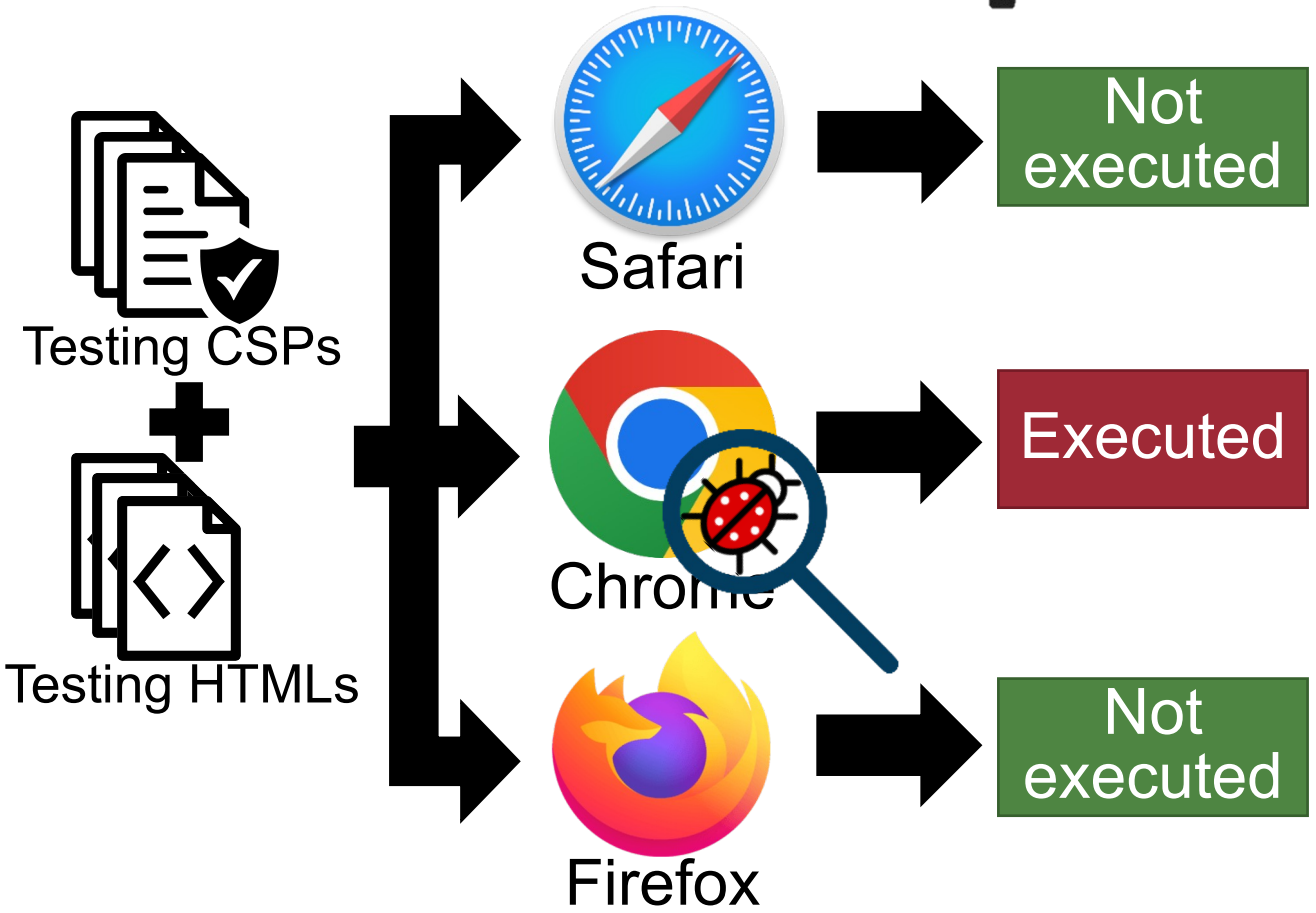


Our approach:
Differential testing!

(Ref) Finding CSP Enforcement



(Ref) Finding CSP Enforcement



Result

- First testing tool for finding CSP enforcement bugs
- Found **27 browser bugs** in Chrome, Safari, and FireFox
- …and **three description bugs**
- 23 bugs have been patched
- Published in *NDSS'23*

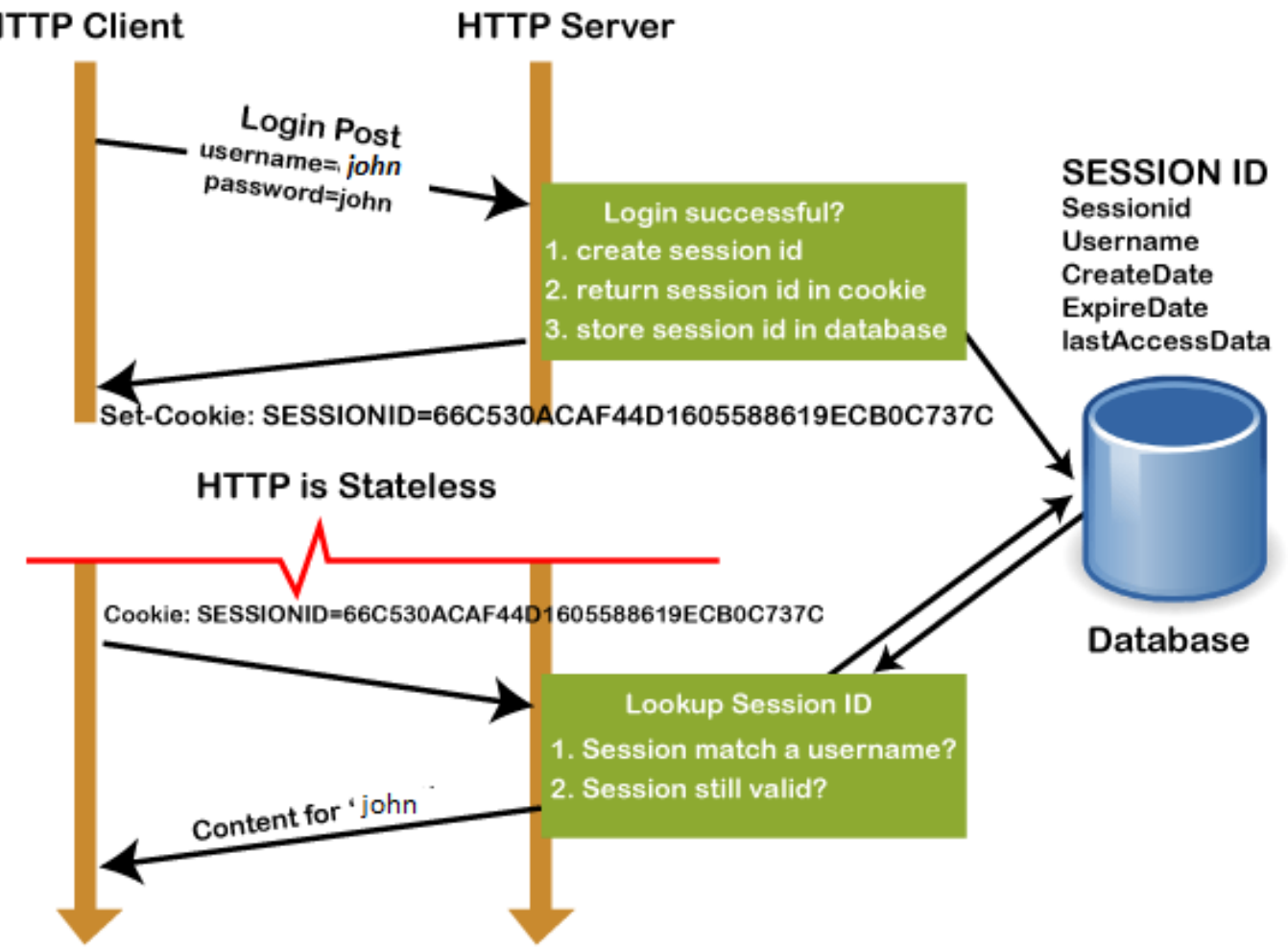
Recent Studies

- 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***

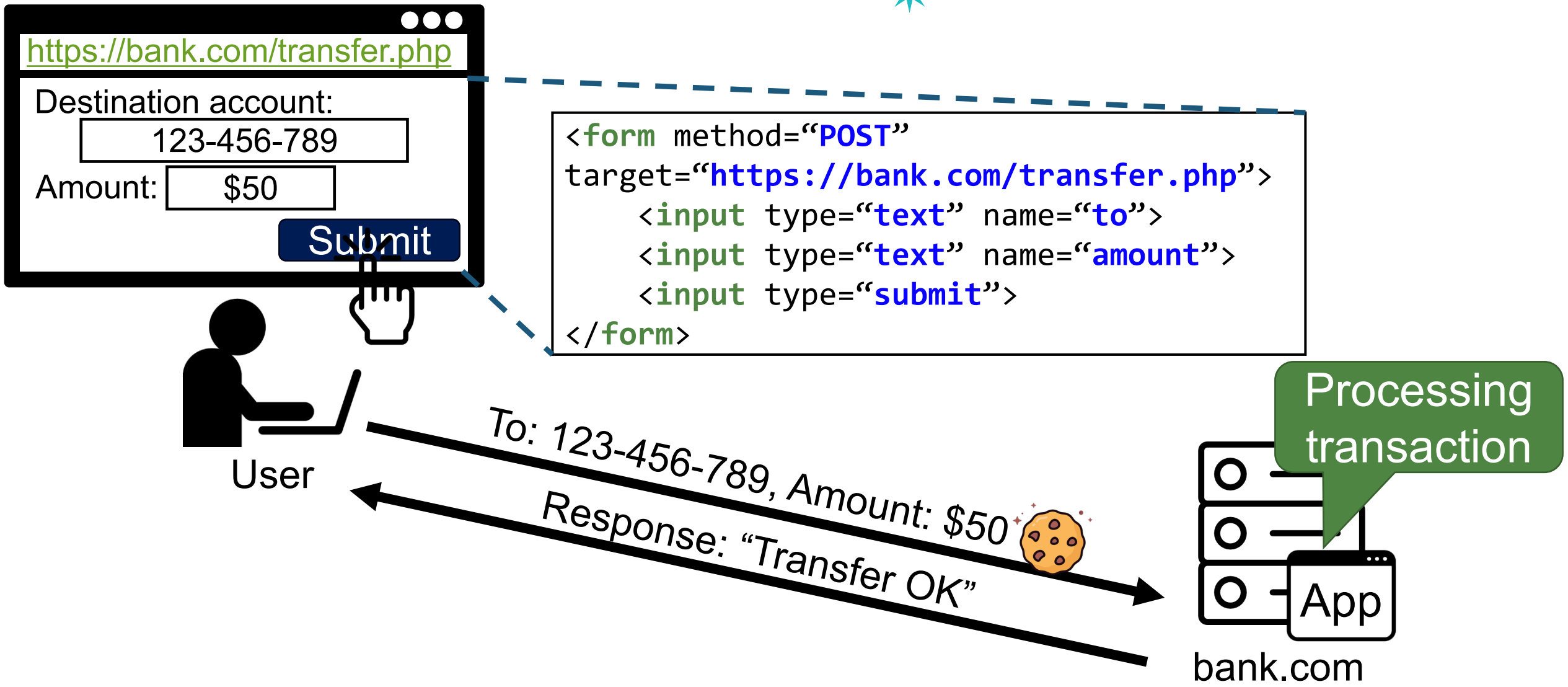
Cross-Site Request Forgery (CSRF)

Recall: Cookies 

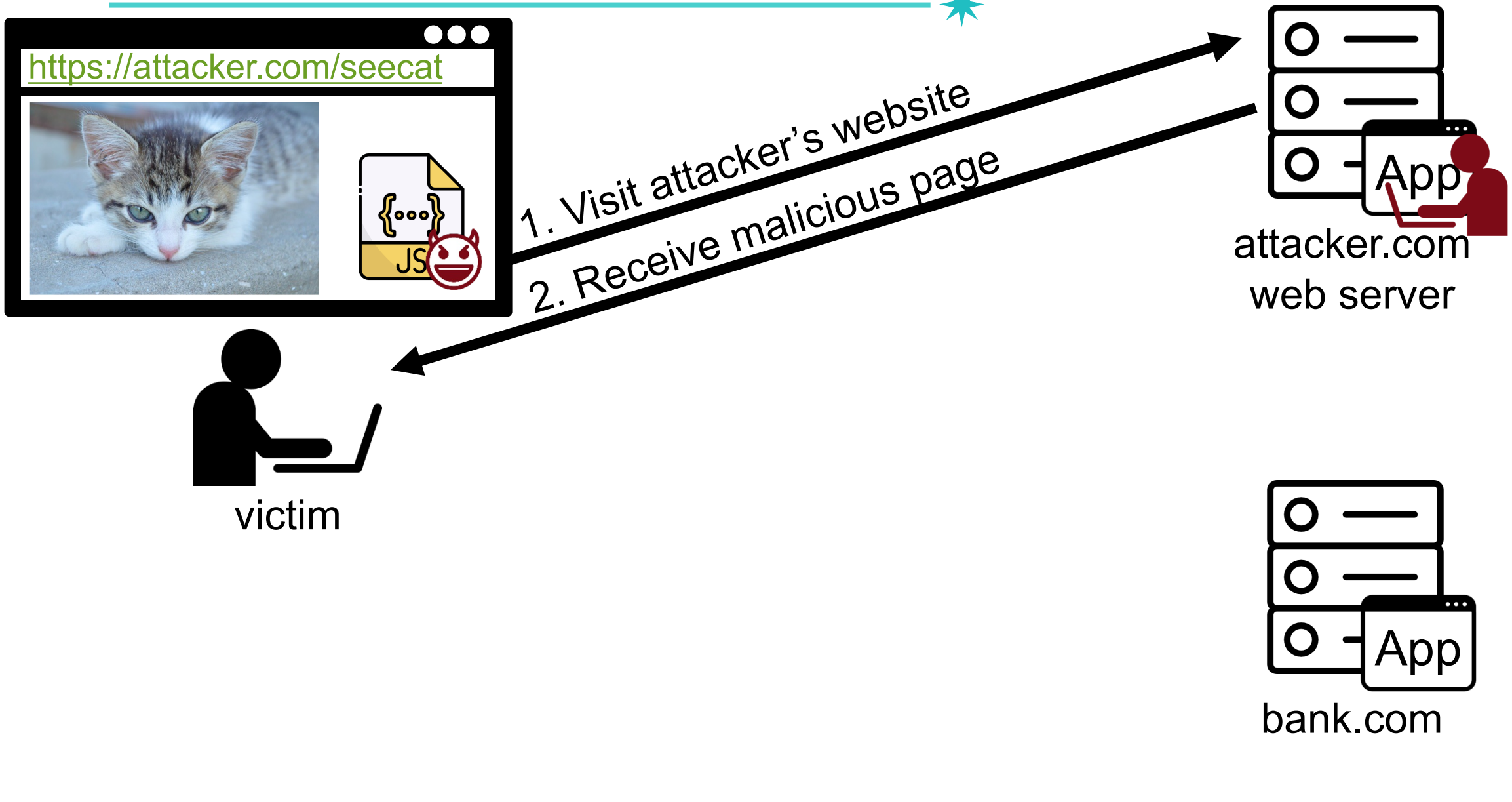
- A common usage: authentication
  - E.g., log into bank.com
- Once authenticated, subsequent request will be accepted
- What if an attacker **tricks the user** to do unwanted actions?
  - E.g., send money to the attacker



Regular Website Usage



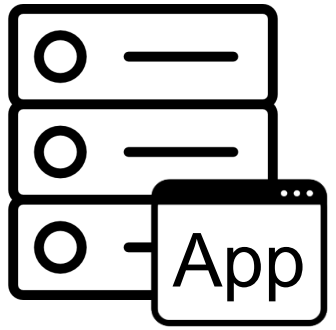
Cross-Site Request Forgery (CSRF)



Cross-Site Request Forgery (CSRF)

```
<form method="POST" id="transfer"
target="https://bank.com/transfer.php">
  <input type="hidden" name="to" value="attacker_account">
  <input type="hidden" name="amount" value="1000000">
</form>
<script>
   transfer.submit();
</script>
```

victim



bank.com

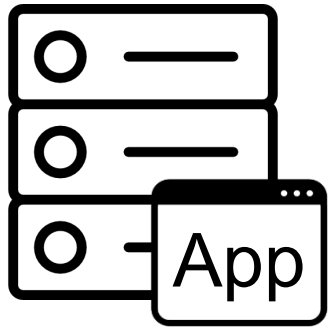
Cross-Site Request Forgery (CSRF)


```
<form method="POST" id="transfer"
target="https://bank.com/transfer.php">
  <input type="hidden" name="to" value="attacker_account">
  <input type="hidden" name="amount" value="1000000">
</form>
<script>
   transfer.submit();
</script>
```

Attacker can control these values

Send request!
(No user involvement required)

victim



bank.com

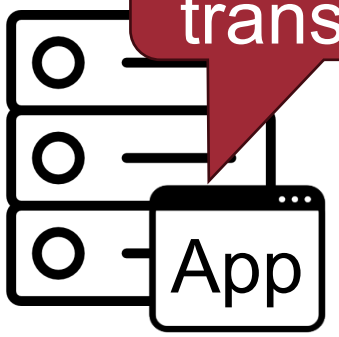
Cross-Site Request Forgery (CSRF)

```
<form method="POST" id="transfer"
target="https://bank.com/transfer.php">
  <input type="hidden" name="to" value="attacker_account">
  <input type="hidden" name="amount" value="1000000">
</form>
<script>
   transfer.submit();
</script>
```

victim

To: attacker_account, Amount: $1,000,000

Processing transaction



bank.com

Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF)

- Force a user to execute unwanted actions (e.g., changing state) on an authenticated web application
- Attack works for GET …
  - Invisible images, hidden iframes, css files, scripts, …

```
<img src="bank.com/transfer.php?to=attacker_account&amout=100" height="0" width="0">
</img>
```

Cross-Site Request Forgery (CSRF)

- Force a user to execute unwanted actions (e.g., changing state) on an authenticated web application
- Attack works for GET …
  - Invisible images, hidden iframes, css files, scripts, …

```
<img src="bank.com/transfer.php?to=attacker_account&amout=100" height="0" width="0">
</img>
```

Browser send request on behalf of the user (victim)

The image is not visible, but the request goes out

Cross-Site Request Forgery (CSRF)

- Force a user to execute unwanted actions (e.g., changing state) on an authenticated web application
- Attack works for GET …
  - Invisible images, hidden iframes, css files, scripts, …

```
<img src="bank.com/transfer.php?to=attacker_account&amout=100" height="0" width="0">
</img>
```

- and POST

```
<form method="POST" action="https://bank.com/transfer.php" id="transfer">
  <input type="hidden" name="act-to" value="attacker_account">
  <input type="hidden" name="amount" value="100000">
</form>
<script>
  transfer.submit()
</script>
```

Cross-Site Request Forgery (CSRF)

- Force a user to execute unwanted actions (e.g., changing state) on an authenticated web application

Very important point:
A web page can **send information to any site!**

- and POST

```
<form method="POST" action="https://bank.com/transfer.php" id="transfer">
  <input type="hidden" name="act-to" value="attacker_account">
  <input type="hidden" name="amount" value="100000">
</form>
<script>
  transfer.submit()
</script>
```

SOP Does Not Control Sending

- SOP violation? Nope!

- Same origin policy (SOP) controls access to DOM
- Active content (scripts) can send a request anywhere!
  - No user involvement required

CSRF on Netflix 2006

CSRF on Netflix 2006

- CSRF vulnerabilities at Netflix allowed the attacker to do:
  - Add movies to your rental queue
  - Add a movie to the top of your rental queue
  - Change the name and address of a victim's account
  - Change the email and password on a victim's account

```
<img
  src="http://www.netflix.com/changeinfo?email=seongil.wi@unist.ac.kr&password=hello"
  width="1"
  height="1"
  border="0">
</img>
```

How to Defense CSRF Attacks?

- Referrer checking: “where is this request coming from?”
  - Accept requests only if their referrer is the same as the server (e.g., *.bank.com)

Recap: Referrer Header


```
GET  /cse467.html  HTTP/1.1
Host: websec-lab.com
Accept-Language: en
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64;)
Referer: http://google.com
```

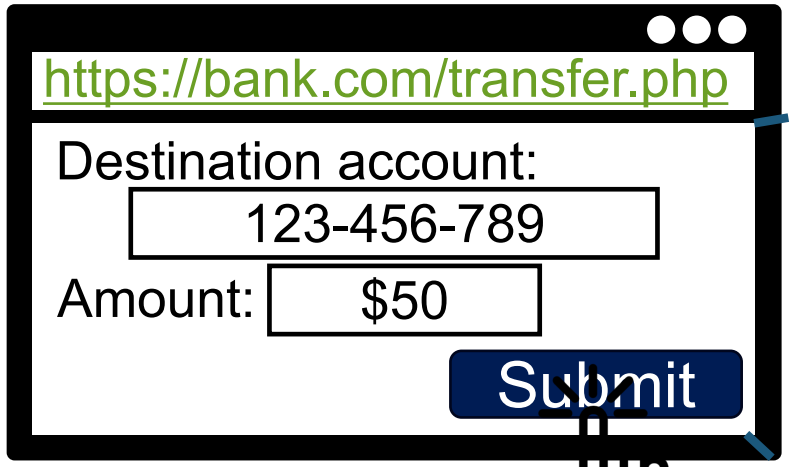
Contain the address from which a resource has been requested

How to Defense CSRF Attacks?

- Referrer checking: “where is this request coming from?”
  - Accept requests only if their referrer is the same as the server (e.g., *.bank.com)

- Secret validation token
  - For each session, a fresh secret token is generated by the server
  - Send requests with the token
  - Accept requests only if the token is valid

Secret Validation Token: Regular Usage




```
<form method="POST"
target="https://bank.com/transfer.php">
  <input type="text" name="to">
  <input type="text" name="amount">
  <input type="hidden" name="token" value="N73GN9IA">
  <input type="submit">
</form>
```

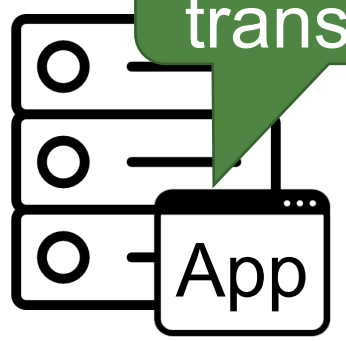
Random token

User

To: 123-456-789, Amount: $50

Token: N73GN9IA

Response: "Transfer OK"



bank.com

Processing transaction

Secret Validation Token: Preventing CSRF


```
<form method="POST" id="transfer"
target="https://bank.com/transfer.php">
  <input type="hidden" name="to" value="attacker_account">
  <input type="hidden" name="amount" value="1000000">
  <input type="hidden" name="token" value="noclue">
</form>
<script>
   transfer.submit();
</script>
```

victim

To: attacker_account, Amount: $1,000,000

Token: noclude

bank.com

Invalid token:
No transaction

How to Defense CSRF Attacks?

- Referrer checking: “where is this request coming from?”
  - Accept requests only if their referrer is the same as the server (e.g., *.bank.com)
- Secret validation token
  - For each session, a fresh secret token is generated by the server
  - Send requests with the token
  - Accept requests only if the token is valid
- SameSite Cookies

Same-Site Cookies

- Two modes
  - Strict: browser will NEVER send cookies with cross-origin request

```
Set-Cookie: session=0F8tgdOhi9ynR1M9wa3ODa; SameSite=Strict
```

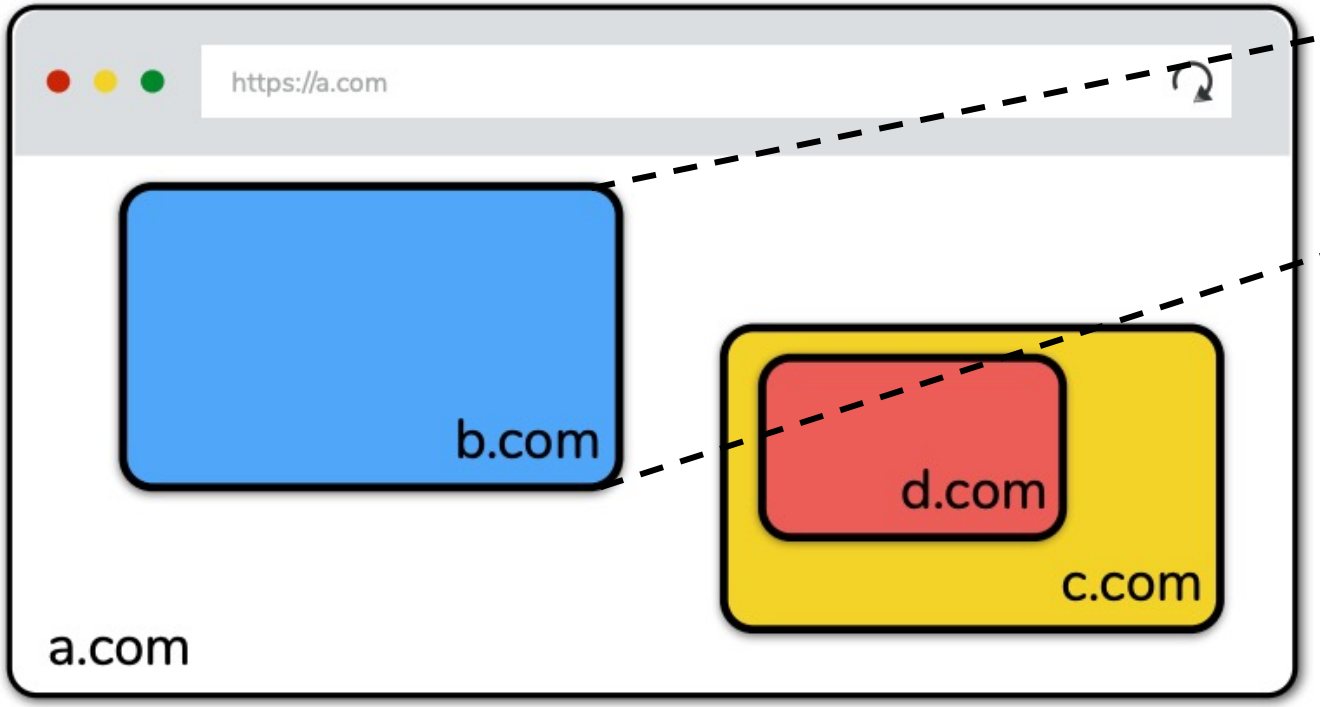
  - Lax: browser will send the cookie in cross-site requests, but only if both of the following conditions are met:
    - The request uses safe requests (e.g., GET)
    - The request resulted from a top-level navigation by the user, such as clicking on a link

- Until May 2018, only supported by Chrome and Opera
- Since Chrome 80, defaults to SameSite=lax

Clickjacking 

Recap: Browser Execution Model

- Windows may contain frames from different sources
  - **Frame**: rigid visible division
  - **iFrame**: floating inline frame




```
<iframe src="b.com">
</iframe>
```

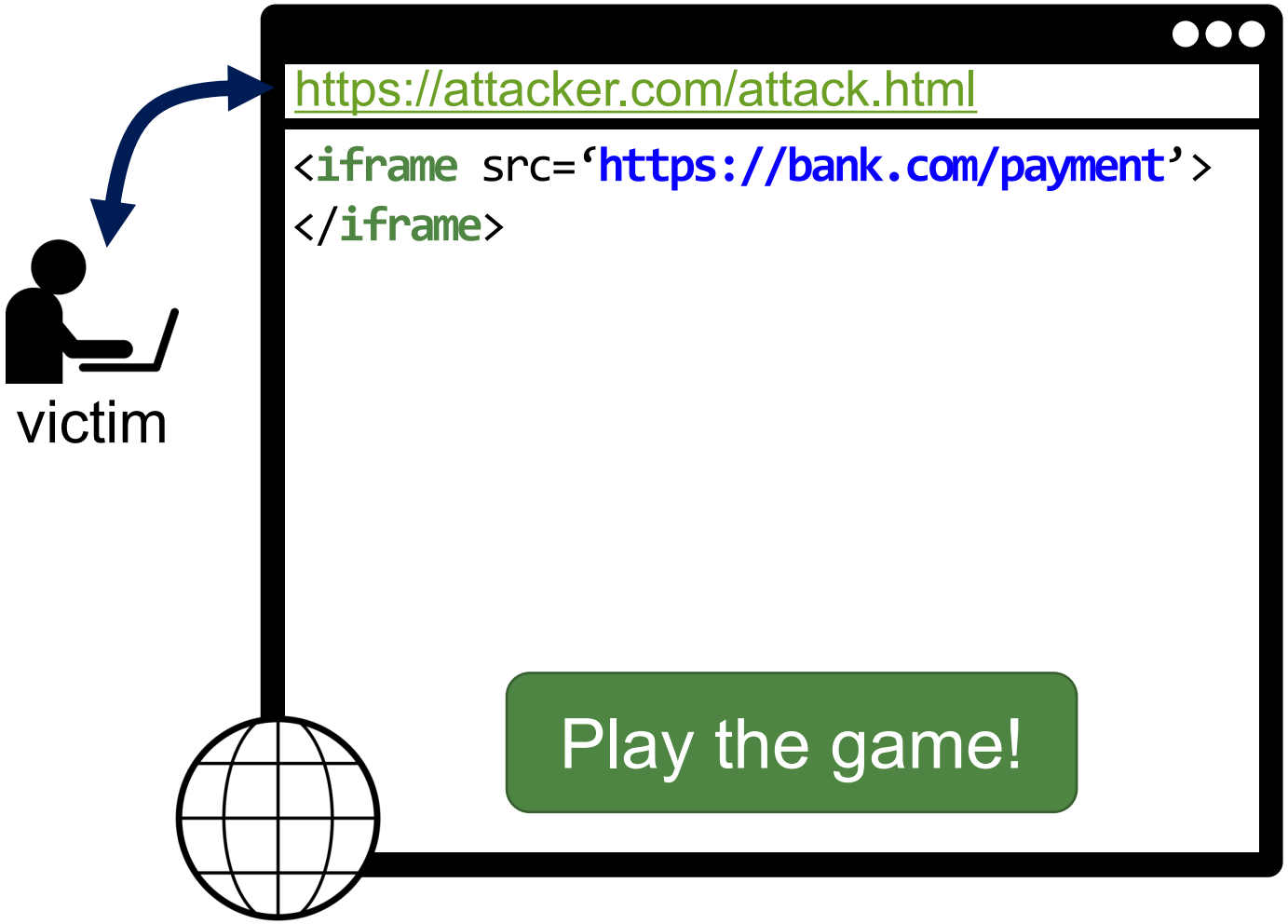
Framing other Websites

- HTML supports framing of other (cross-origin sites)
  - E.g., iframes
  - Very useful feature for advertisement, like buttons, …
- Embedding site controls most of the frame's properties
  - How large the frame should be
  - Where the frame is displayed
  - How opaque the frame should be….

What could go wrong?

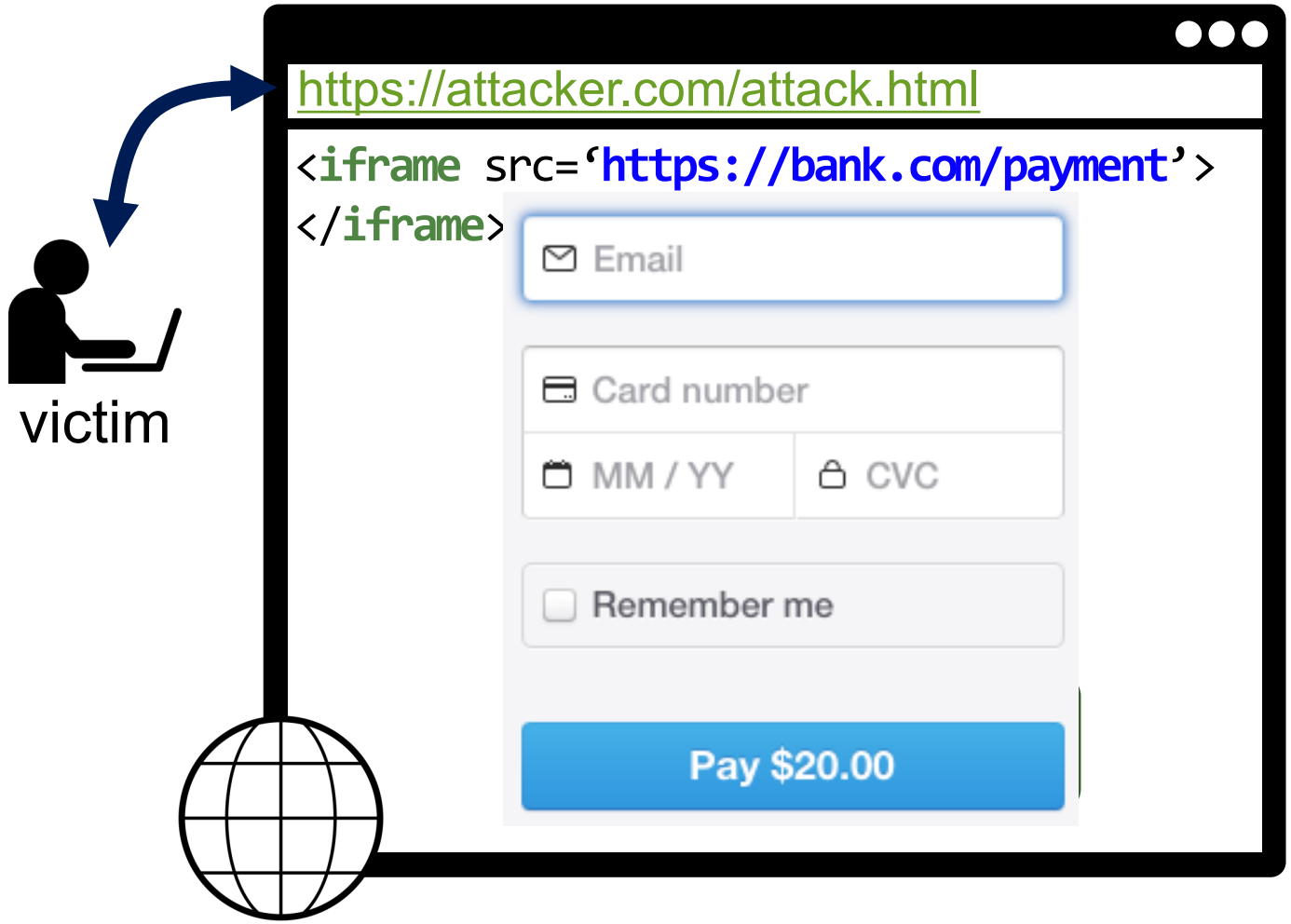
Clickjacking (UI Redressing)

- Attacker **overlays transparent or opaque frames** to trick a user into clicking on a button or link on another page



Clickjacking (UI Redressing)

- Attacker **overlays transparent or opaque frames** to trick a user into clicking on a button or link on another page

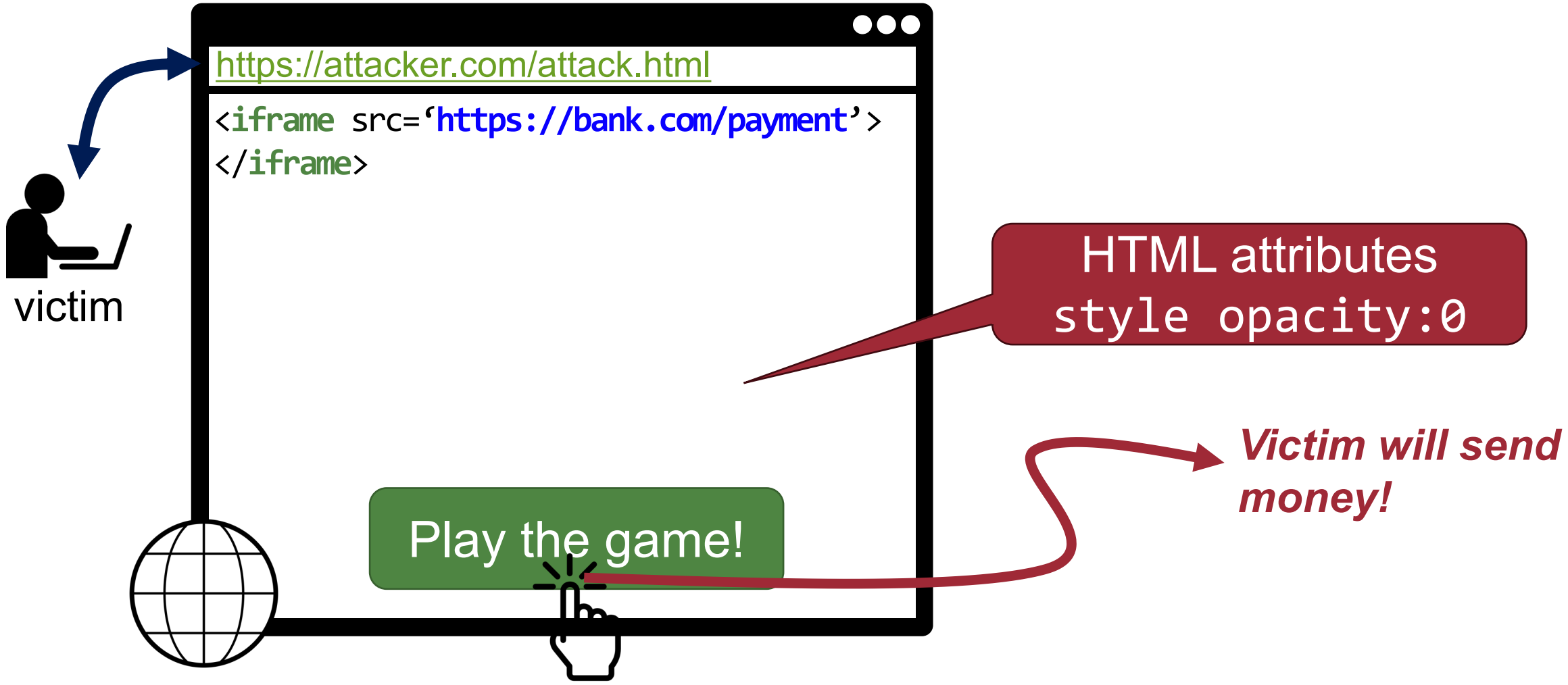


Clickjacking (UI Redressing)

- Attacker **overlays transparent or opaque frames** to trick a user into clicking on a button or link on another page

Clickjacking (UI Redressing)

- Attacker **overlays transparent or opaque frames** to trick a user into clicking on a button or link on another page

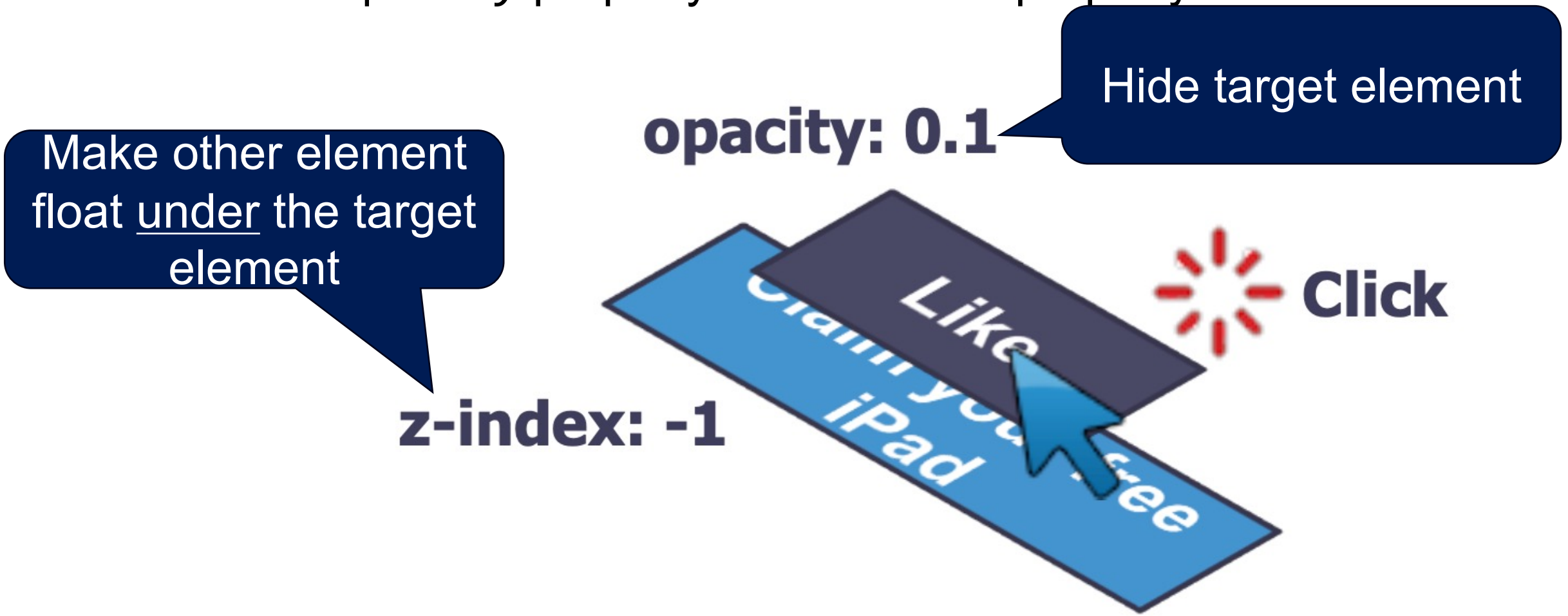


Clickjacking Demo

- Demo: https://websec-lab.github.io/courses/2023f-cse467/demo/clickjacking_demo1.html

Clickjacking – Hiding the Target Element

- Use CSS `opacity` property and `z-index` property



Clickjacking - Fake Cursors

- Use CSS `cursor` property and JavaScript to simulate a fake cursor icon on the screen

Real cursor icon

Fake cursor icon

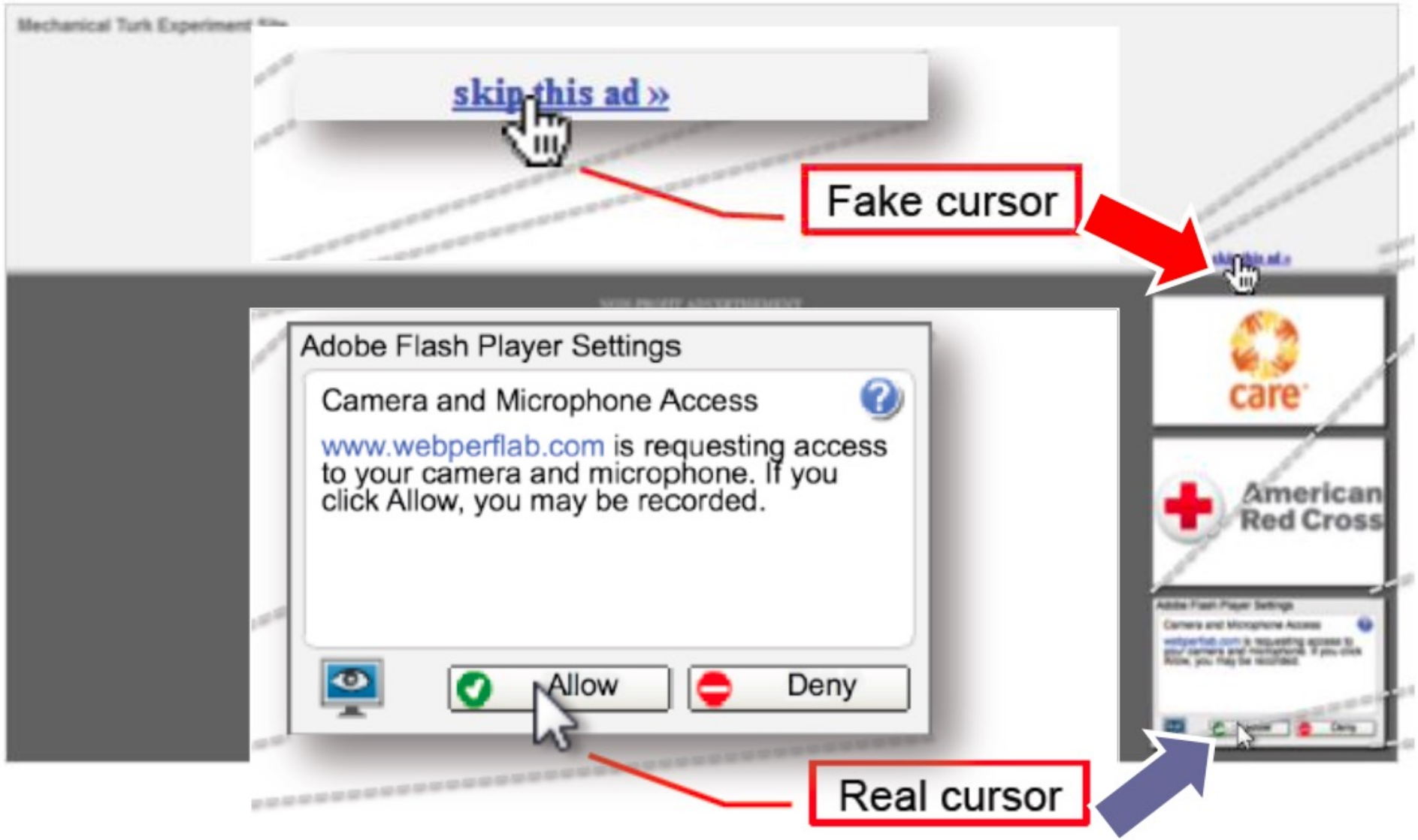
Clickjacking - Fake Cursors

- Use CSS `cursor` property and JavaScript to simulate a fake cursor icon on the screen

Hide real curson icon by using `cursor: none`

Fake cursor icon

Cursor Spoofing



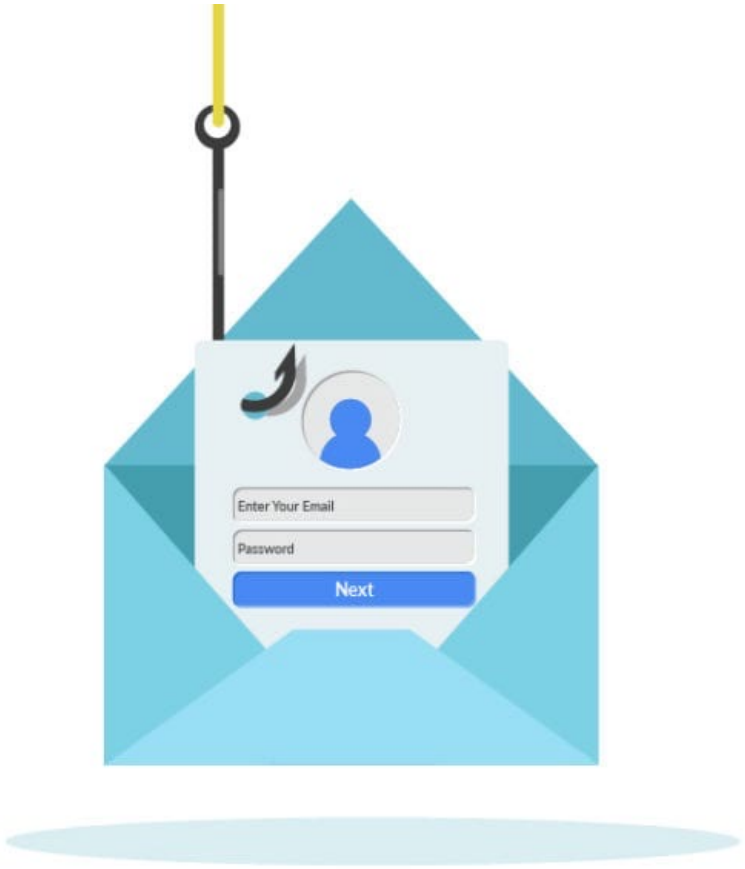
How to Prevent Clickjacking?

- Frame busting
  - Make sure that my website is not loaded in an enclosing frame

```
if (top != self)
  top.location = self.location
```
JS

- Use CSP's frame-ancestors
  - Determine whether my website may be embedded in another site
  - `'none'`: denies from any host
  - `'self'`: allows only from same origin
  - `http://example.org`: allows specific origin

Phishing

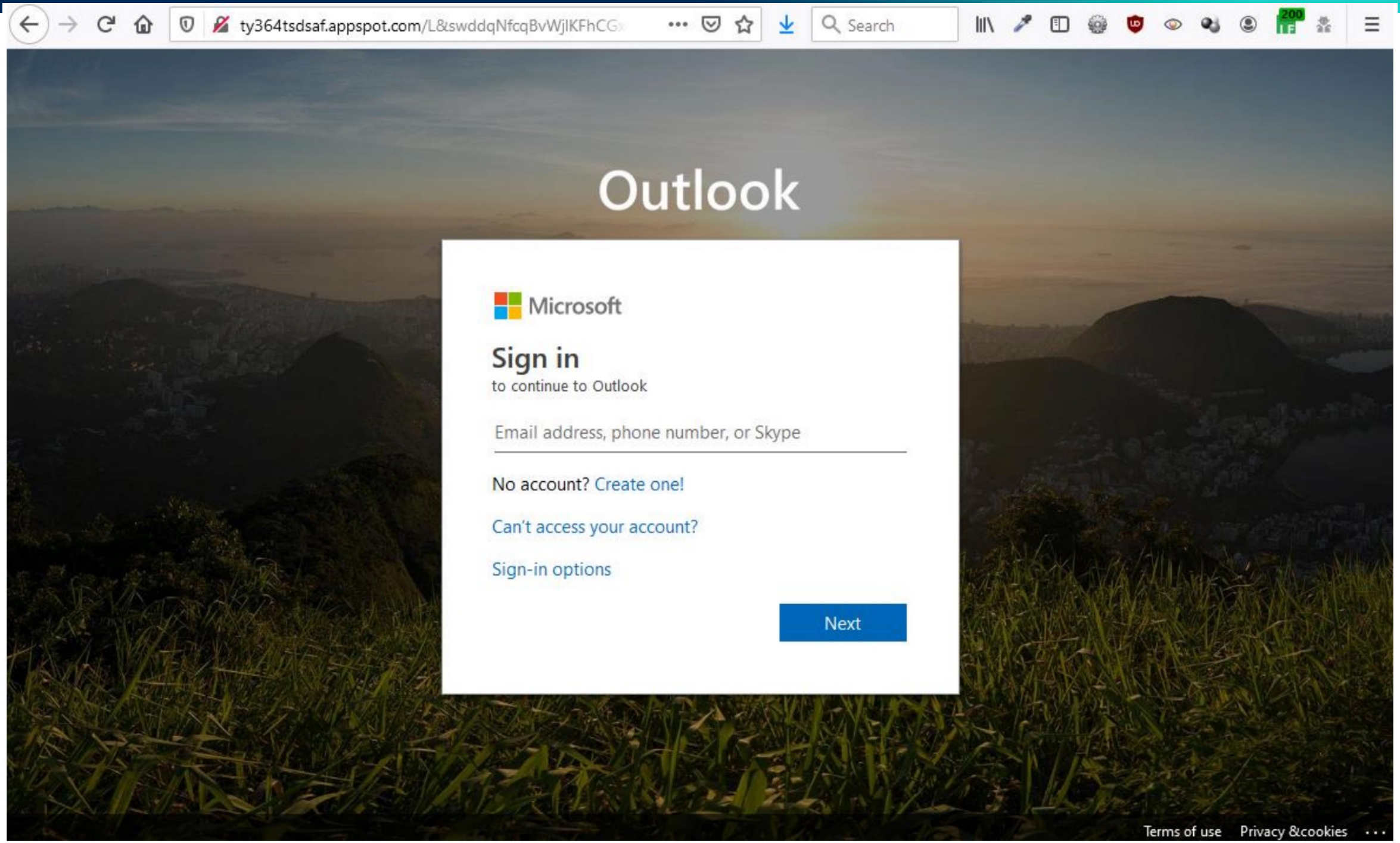


Phishing


1. Visit attacker's website
2. Receive malicious page
3. Send sensitive info. (e.g., credential info.)
App
attacker.com
web server
victim
https://attacker.com/login
로그인
계정생성
아이디찾기
비밀번호 초기화
ID
user_id
PW
my_password

Phishing

- Disguising as a trustworthy entity, and obtain private information
  - Login credentials
  - Financial records
- Links to phishing webpages dispatched to victims through email or SMS
- According to a report from the FBI, it received 800,944 reports of phishing, with losses exceeding $10.3 billion in 2022


ty364tsdsaf.appspot.com/L&swddqNfcqBvWjlKFhCG
Search
Outlook
Microsoft
Sign in
to continue to Outlook
Email address, phone number, or Skype
No account? Create one!
Can't access your account?
Sign-in options
Next
Terms of use
Privacy &cookies

Phishing

From: apple.Inc <Update.account.confirmed@altervista.org>
To:
Sent: Thursday, April 24, 2014 12:35 PM
Subject: Update your Account information !

Dear iTunes Customer!

Your itunes account has been frozen because we are unable to validate your account information.
Once you have updated your account records, we will try again to validate your information
and your account suspensionwill be lifted. This will help protect your account in the future.
This process does not take more than 3 minutes. To proceed to confirm your account details please click on the link below and follow the instructions.

Get Started

If you nee http://goo.gl/Gkx2HM our Help left by clicking the Help link located in the upper right-hand corner of any Apple page.

Sincerely,

Apple Inc

Please do not reply to this email. We are unable to respond to inquiries sent to this address. For immediate answers to your questions, visit our Help left by clicking "Help" at the top of any Apple page.

Typical Properties of Spoofed Sites

- Attackers manually copy/recreate web content from target website
  - Show logos found on the honest site
- Have suspicious URLs: mostly, being camouflaged as a URL that looks familiar to people
  - E.g., units.ac.kr
- Ask for user input
  - Debit card number, user name, password, …
- Phishing content served from attacker-owned web server
  - Or a compromised web server

Safe to Type Your Password?

Spear Phishing

- Phishing attempts directed at specific individuals
- This can increase the likelihood of success, as the sender appears more credible and informed

Spear Phishing

From: UDEL HR <hremployeepayroll@udel.edu>
Date: August 13, 2015 at 12:48:29 PM EDT
To: <[redacted]>
Subject: Your August 2015 Paycheck

UNIVERSITY of DELAWARE

Hello,

We assessed the 2015 payment structure as provided for under the terms of employment and discovered that you are due for a salary raise starting August 2015.

Your salary raise documents are enclosed below:

Access the documents here

Faithfully

Human Resources

University of Delaware

Spear Phishing

Mrs. Füsun Tümsavaş <jsc7339@gmail.com>

09-12 (화) , 오전 1:01

Seongil Wi <seongil.wi@unist.ac.kr>

전체 회신

지운 편지함

Hello Seongil Wi,

I am contacting you for the receipt of the sum of US$9,500,000.00 (Nine Million Five Hundred United State Dollars) only.

Please Let me know if you are interested,

Regards,
Mrs. Füsun Tümsavaş

How to Detect Phishing?

- Crowdsourcing, Blacklisting
  - lists reported phishing URLs
  - E.g., https://openphish.com/

Phishing URL	Targeted Brand	Time
https://diepost-zoll-ch.com/steps/	Generic/Spear Phishing	06:24:20
https://257.nhksf.com/	Tencent	06:18:49
https://ffspind7my.terbaru-2023.com/vhsfhqpdhdxih1	Garena	06:17:46
https://web.telegram.data-bees.cn/	Telegram	06:17:26
http://wantlengtime.com/	WhatsApp	06:16:08
http://manualmetarestore-39f.pages.dev/	Crypto/Wallet	06:15:36

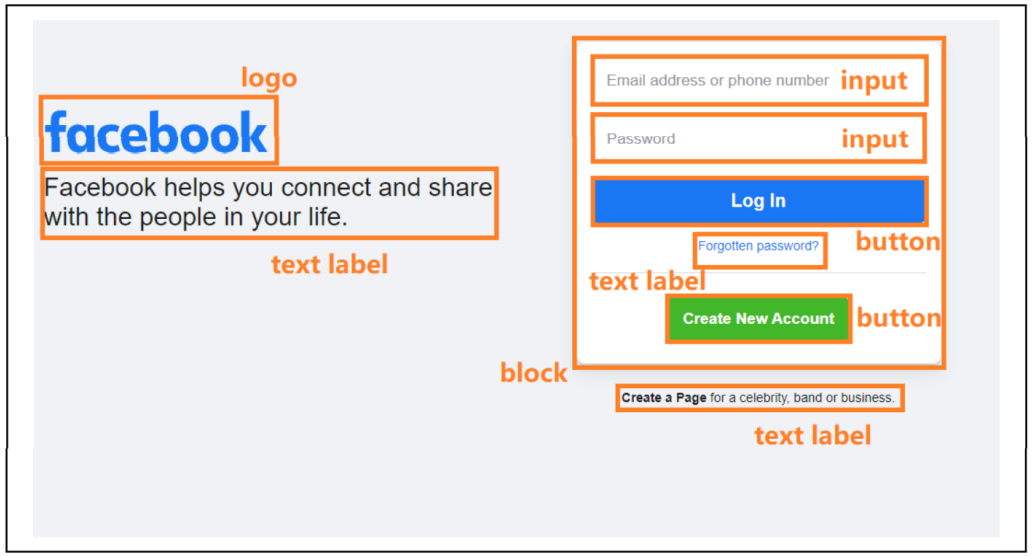
How to Detect Phishing?

- Crowdsourcing, Blacklisting
  - lists reported phishing URLs
  - E.g., https://openphish.com/

- URL-based pattern detection
  - E.g., A URL is phishy if its length ≥ 76
  - E.g., Brand name modification with '-'
    - youtube-x.com

- Content-based pattern detection

- Require human intervention and verification
- Phishers are starting to use one-time URLs



Conclusion

- Content Security Policy (CSP)
  - Allow resources which are trusted by the developer
- Cross-Site Request Forgery (CSRF)
  - Force a user to execute unwanted actions
- Clickjacking
  - Attackers overlays opaque frames to trick a user
- Phishing
  - Disguising as a trustworthy entity, and obtain private information

Question?