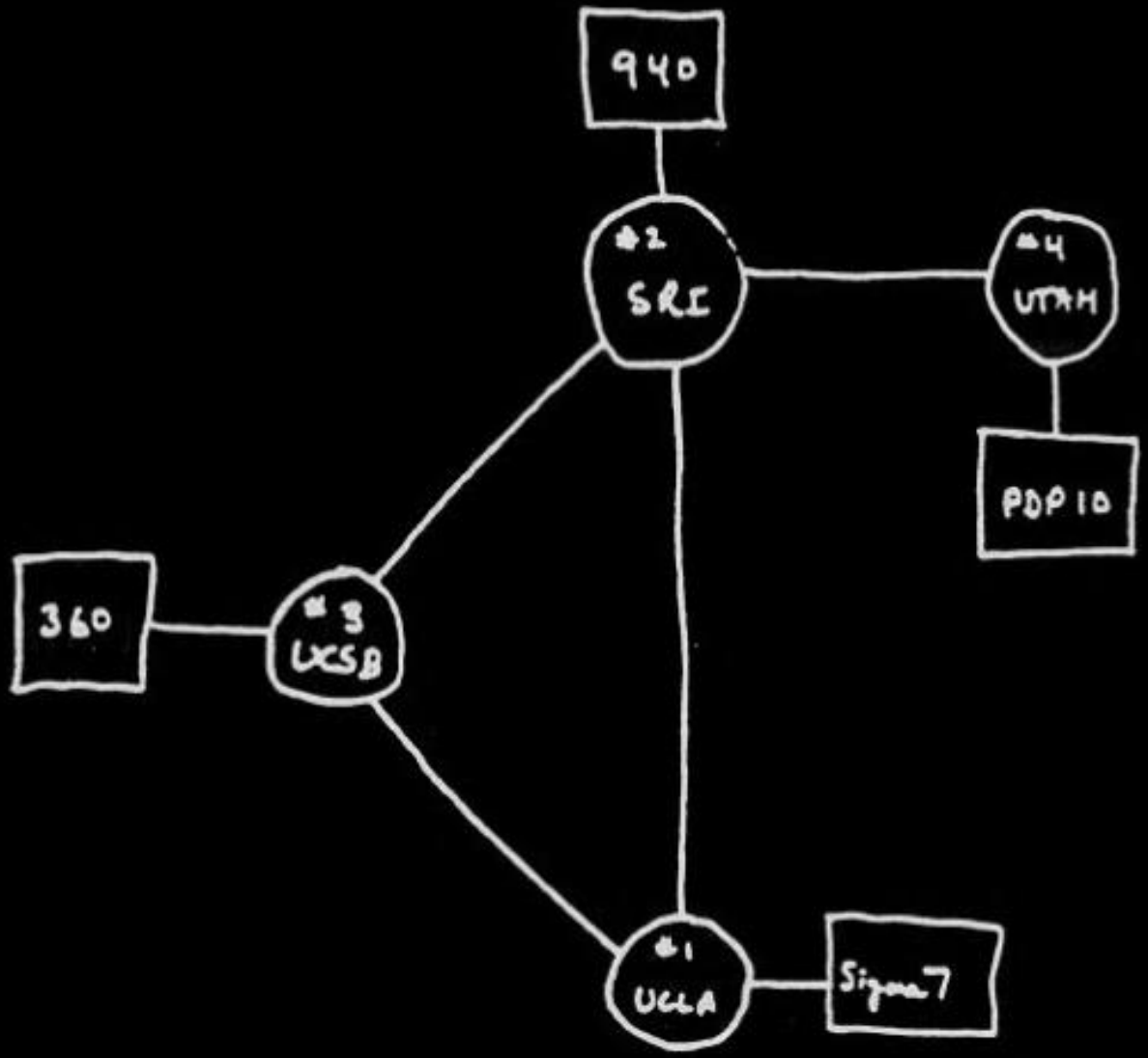
We Fight For The User's... Session

SaintCon 2024

Mark Hoopes


940
#2
SRI
#4
UTAH
PDP 10
360
#3
UCSB
#1
UCLA
Sigma 7

HTTP


N
A
X Wind
intosh

Session Defenses… Why?

I'm tired attribute
and s dings.

What do you want to sell?

Cookies and logins

[discussion about this not being the marketplace]

Sell away!

The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol which can be used for many tasks, such as name servers and distributed object management systems…

RFC 1945

Cookie RFC Section 8: Security Considerations

8.3: Unexpected Cookie Sharing

- Setting cross-domain cookies
  - Browsers blocked this one effectively

8.2: Cookie Spoofing

- Abuse of cookies from subdomains
  - Rare, but a thing. Liv Matan “My Terrible Roommates” BSidesLV 2024

8.1: Clear Text

- HTTP connections are not encrypted on the wire
  - Let’s start here

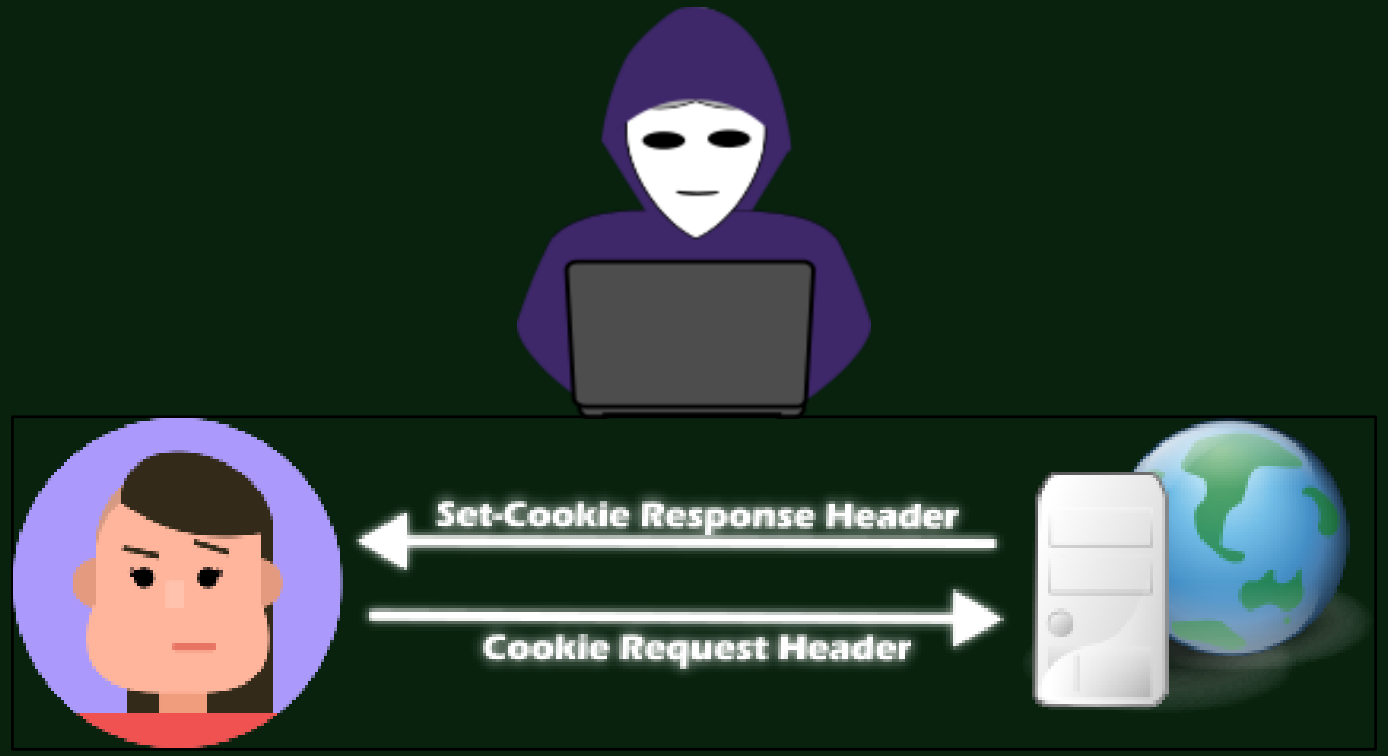
RFC 2109

Session Protection
“On-the-wire”

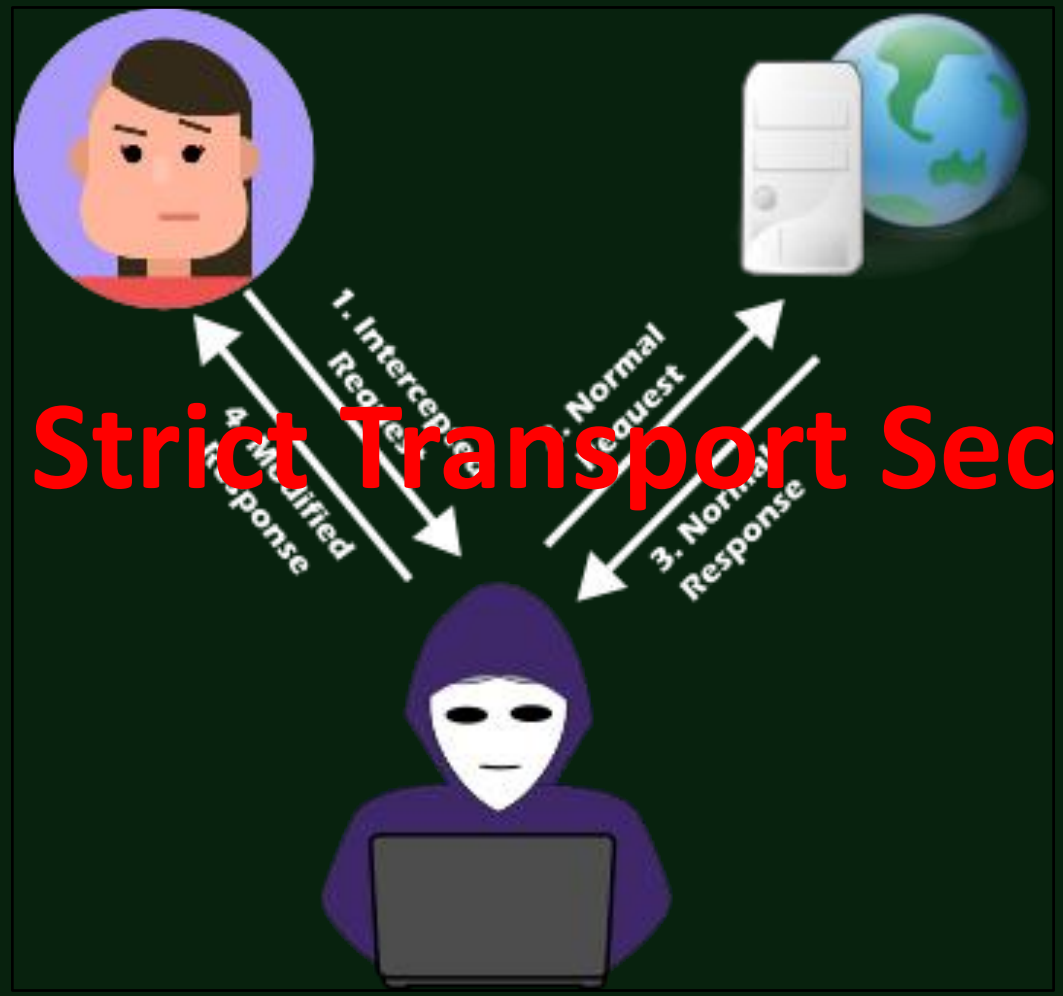
RFC 2109: Section 4.2.2

Secure

Optional. The Secure attribute (with no value) directs the user agent to use only (unspecified) secure means to contact the origin server whenever it sends back this cookie.

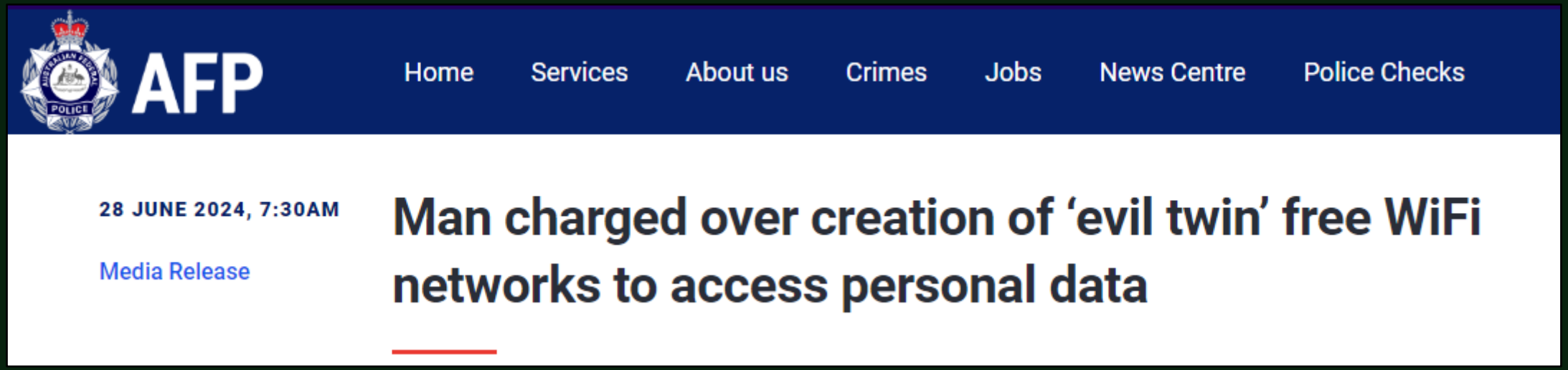

Set-Cookie Response Header
Cookie Request Header

HTTP Strict Transport Security



https://hstspreload.org/

But...Network Segmentation...

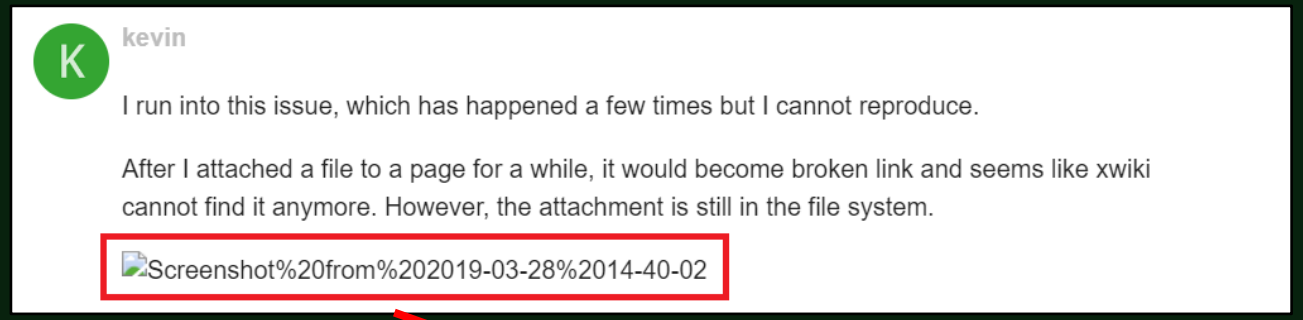


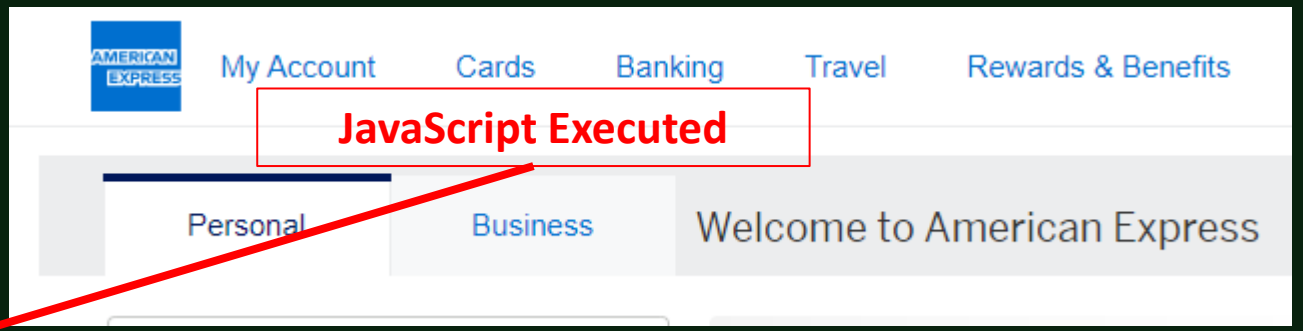
“On-the-wire” Defenses

- ❑ Use HTTPS
- ❑ Set the “secure” attribute on all sensitive cookies
- ❑ Set the “Same-site: Strict” attribute on all sensitive cookies
- ❑ Return the “Strict-Transport-Security” response header
- ❑ List your site on the HSTS Preload List (https://hstspreload.org/)
- ❑ Disable HTTP completely

JavaScript Injection
(a.k.a. Cross-site Scripting)

Origins of “Cross-site Scripting”





JavaScript Injection (JSI) Defenses

4.1.2.6. HttpOnly

Syntax Servers MUST NOT include a value.

Semantics The user agent SHOULD protect confidentiality of cookies with the HttpOnly attribute by not revealing their contents via "non-HTTP" APIs. (Note that this document does not define which APIs are "non-HTTP".)

```
Set-Cookie:
session=03HMPAPjvLt8UISBQBnDrSSAAAAWVFR9gg41o;
domain=.example.com; secure; httponly; SameSite=None
```

```
x=new XMLHttpRequest();
u='https://example.com/api/cookietest’;
x.open('GET', u, true); x.send();
```

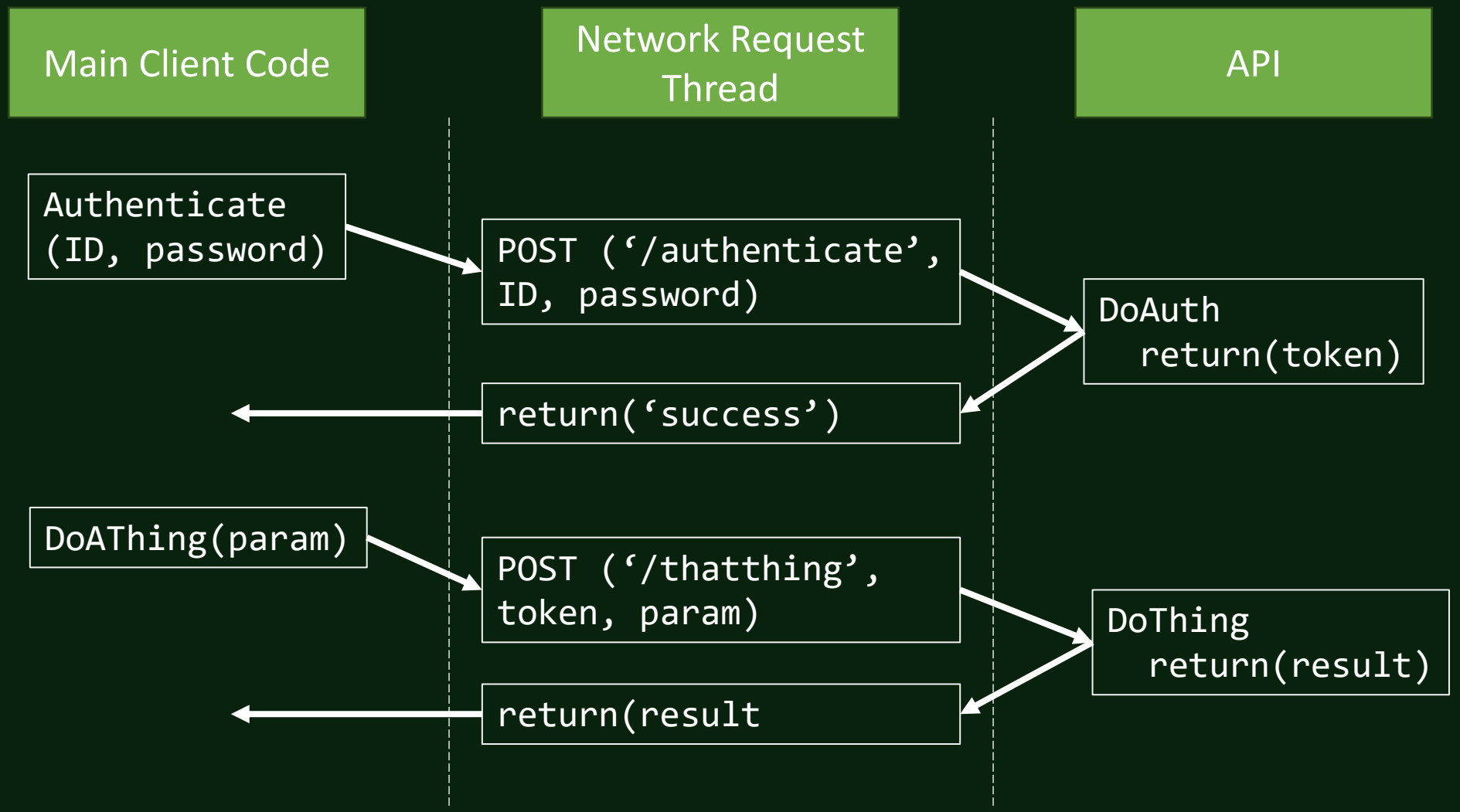
```
undefined
```

```
GET /api/cookietest HTTP/2
Host: www.example.com
Cookie: session=03HMPAPjvLt8UISBQBnDrSSAAAAWVFR9gg41o
```

Not All Session Tokens are Cookies

```
Authorization: BEARER 03HMPAPjvLt8UISBQBnDrSSAAAAWVFR9gg
```

- Authorization headers are not handled automatically by the browser
  - Except for “basic” authentication (bad for different reasons)
- Client-side JavaScript MUST be able to manage the token
  - Best-practice: hide the token inside a worker thread

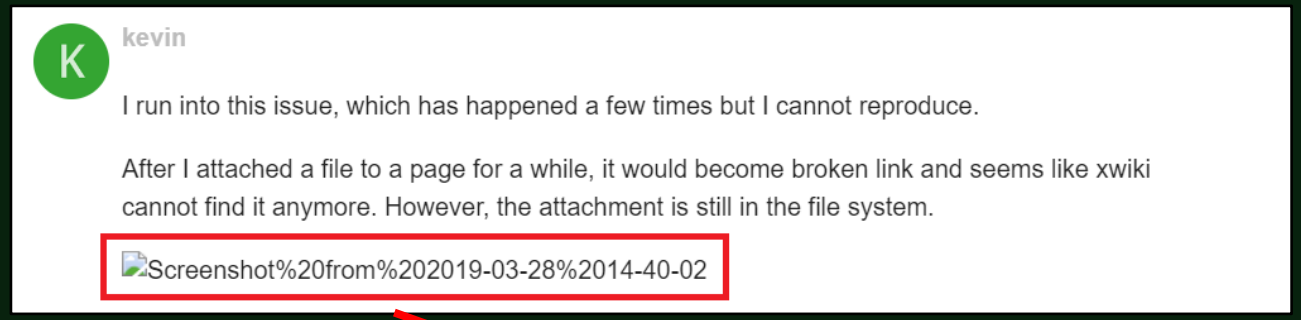

Main Client Code
Network Request Thread
API
Authenticate (ID, password)
POST ('/authenticate', ID, password)
DoAuth return(token)
return('success')
DoAThing(param)
POST ('/thatthing', token, param)
DoThing return(result)
return(result

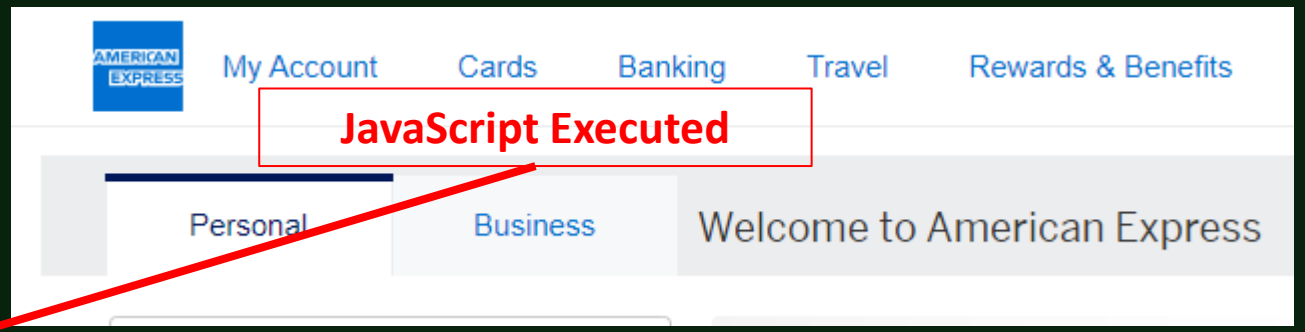
JavaScript Injection Defenses

- Set the “httponly” attribute on all sensitive cookies
- Isolate header token values in worker threads
- Don’t be vulnerable to JavaScript Injection
  - Classic Input Validation and Output Encoding
  - Use a tailored Content Security Policy

Blind Session Attacks

No token. No problem?





/admin/addEditor?docId=21&userId=42

CSRF Requirements

Condition	Defense
The target site must use cookies for authentication	Authorization Header **Anti-CSRF Token in Header** **Same-Site Attribute on session cookie**
The endpoint must use the GET or POST methods	Using UPDATE or DELETE methods
The endpoint must use only predictable parameters	**Require an anti-CSRF token as a parameter**
The results must be state-changing	Above defenses can in theory be applied only to requests that change state.

The JavaScript Injection Loophole

If you have a JavaScript Injection vulnerability on your site…

ALL IS LOST

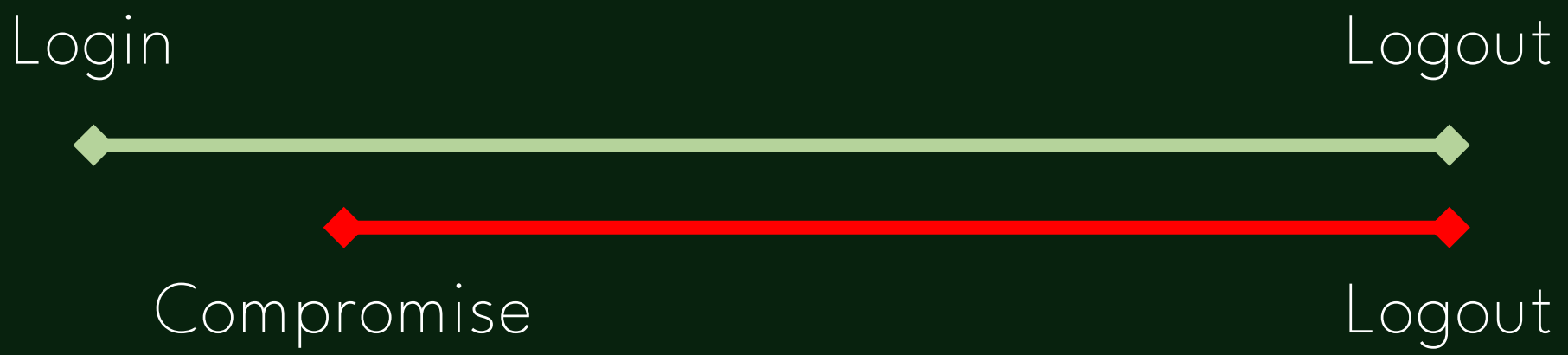
- Since the request comes from the same site
  - Cookies will always be sent
  - JavaScript is allowed to read page contents to retrieve CSRF tokens
  - JavaScript is allowed to add custom request headers (authentication and anti-CSRF)
  - JavaScript can call worker threads to submit protected tokens

Clickjacking

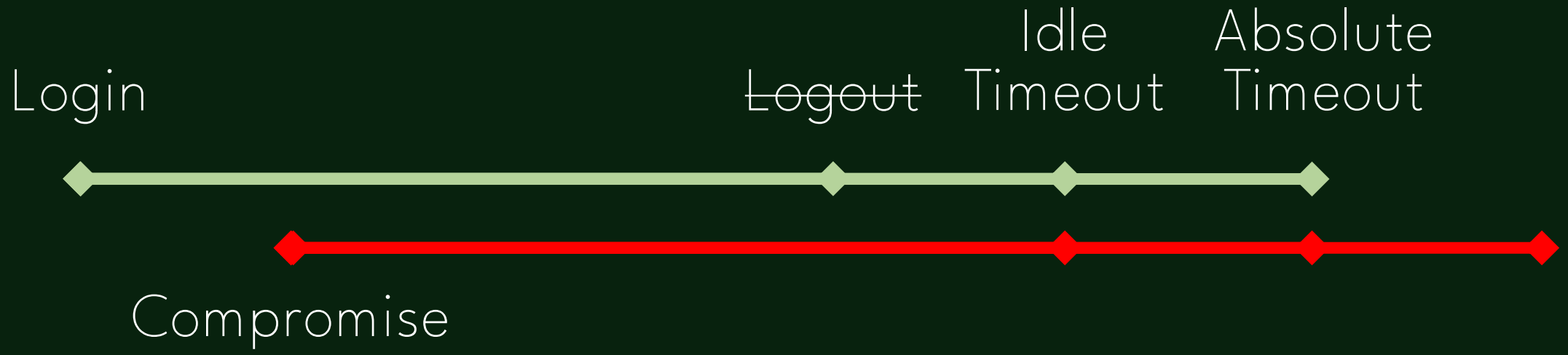
- Technically a risk
  - Facebook got hit with a number of attacks around 20xx
- Very rarely seen in web apps
- More of an issue in mobile apps, but sessions are different inside an application
- Simple to prevent
  - X-Frame Options Header
  - Framing-Ancestor CSP Directive

Post-Compromise Defenses

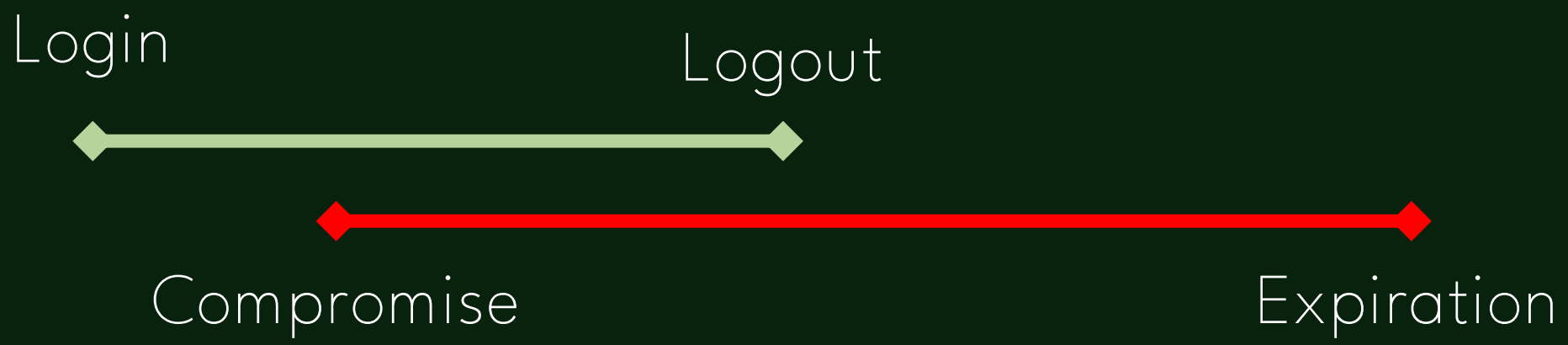
Lifetime of a Session Token



Session Lifetime: User doesn't log out



Why JWTs are ~~BAD BAD BAD~~ Slightly Worse



How many sessions does one user need?

- Allowing a only a single active session
  - Becomes another termination point for a stolen session
  - Can be a detection opportunity if a message is displayed
- If multiple sessions are allowed
  - Provide a way for a user to review and terminate sessions

Additional Defenses

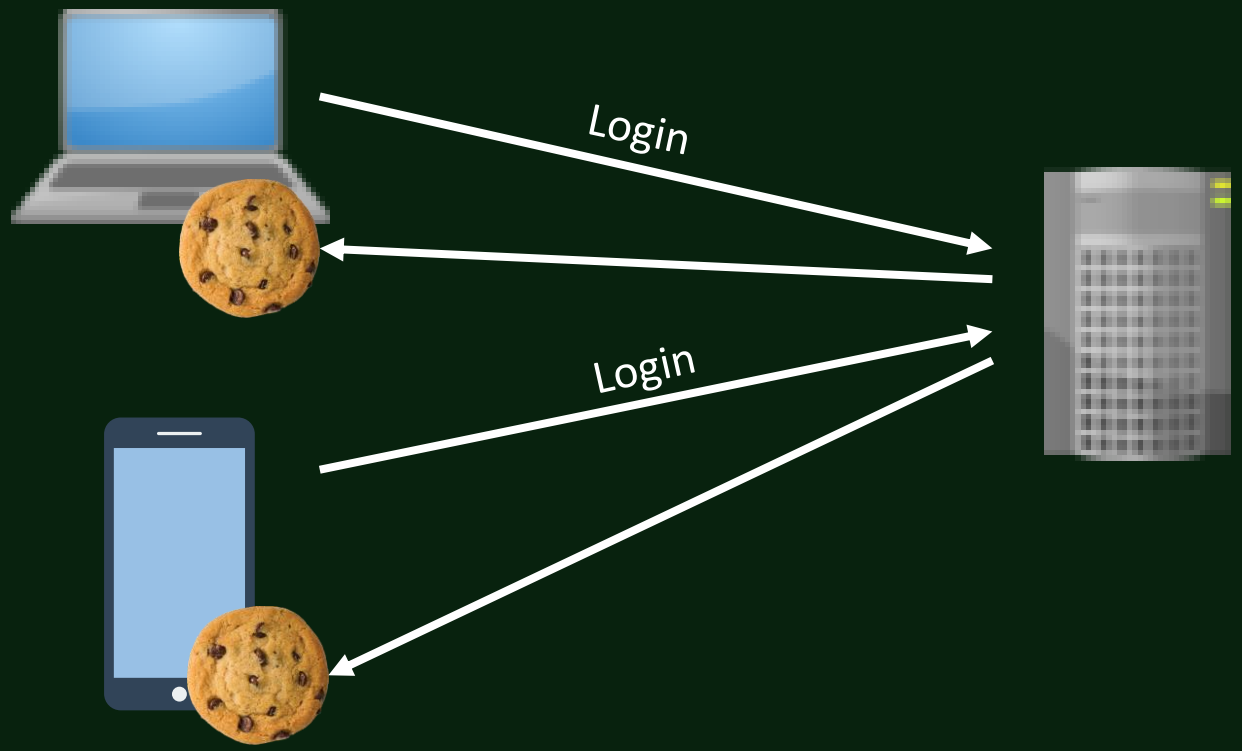
- “Last Login” messages also provide a detection opportunity
- Reauthenticating for sensitive transactions (password changes, money transfers, final approvals, etc.) also limits blast radius

Session Length Summary

- Make your idle and absolute session timeouts as short as the business will allow
- Allow multiple sessions only if the use case requires it
- If multiple sessions are allowed, show active sessions to the user somewhere
- Implement a “Last Login” message
- Require reauthentication before doing the most sensitive tasks

State of the Art

Cross-device tokens aren't a thing



Token Binding (Microsoft 2016)

Pub/Priv Key Pair | HSM/TPM | TLS Extension | HTTP Header

1. Client initiates the TLS connection with the “token binding” extension set to active and the server responds that it is supported.
2. Client generates a public/private key pair for this specific server/session and stores the private key in the TPM.
3. Client sends the public key in an HTTP Request header during the authentication process.
4. If authentication is successful, the server stores the public key with the session.
5. On future requests, the client takes random constants exchanged during TLS initiation, encrypts them with the private key, and sends the encrypted values to the server.
6. The server uses the client’s public key to decrypt the values and compares them with the random constants it also knows to verify the session.

Device Bound Session Credentials (Google 2024)

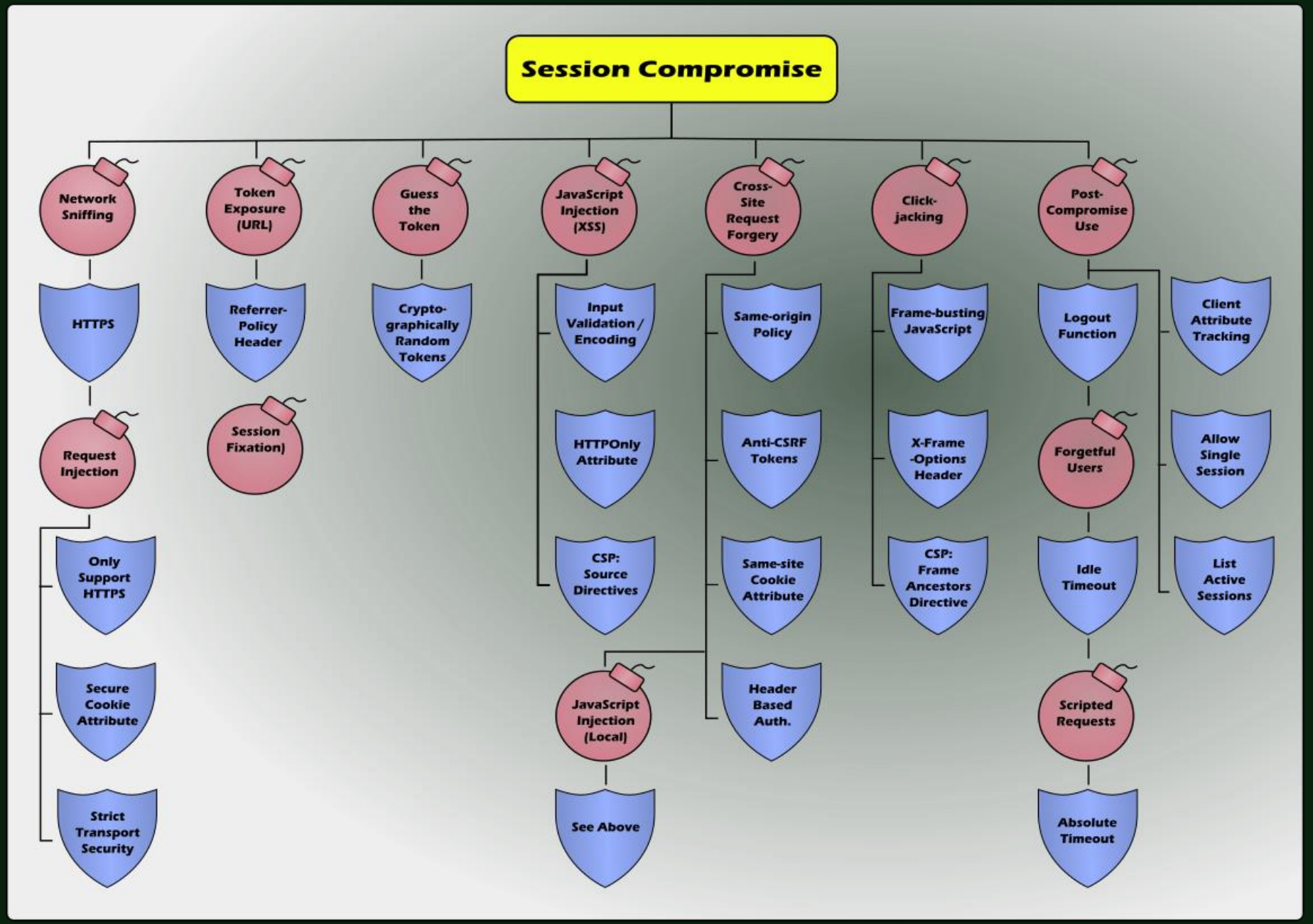
1. The user signs in to the website.
2. The server returns an HTTP header with a challenge, and token management endpoints.
3. The browser generates a public-private key pair with the private key stored in the TPM.
4. The browser POSTs the signed challenge and public key to a “startsession” endpoint.
5. The server returns a session token with a “short” lifetime.
6. When the token expires, the browser requests a new challenge from the “refresh” endpoint.
7. The challenge is signed and resubmitted to get a fresh token.

Challenges for DBSC

- Not all devices have a TPM
  - Google has developed an “emulated” TPM that will not be as good as a real TPM, but will do it’s best and will at least be better than a cookie file.
- ~~They’re using a JWT~~
- Technically, there is still a token that can be used on another device
  - Its lifetime “should” be 10 minutes or less
- Adoption!
  - Other browsers are supposedly implementing the standard, but so far it is only in Chrome 126+
  - Other browsers will have to also provide an emulated TPM

Conclusion

- Session attacks have matured, but often rely on the same flaws older attacks used
- In the opinion of a pentester, developers treat defenses as “old fashioned” and leave them out
- Defenses are “usually” trivial to implement which “should” make the return-on-investment case an easy one
- Device Bound Session Credentials are a game changer


Session Compromise
Network Sniffing
HTTPS
Request Injection
Only Support HTTPS
Secure Cookie Attribute
Strict Transport Security
Token Exposure (URL)
Referrer-Policy Header
Session Fixation)
Guess the Token
Crypto-graphically Random Tokens
JavaScript Injection (XSS)
Input Validation / Encoding
HTTPOnly Attribute
CSP: Source Directives
JavaScript Injection (Local)
See Above
Cross-Site Request Forgery
Same-origin Policy
Anti-CSRF Tokens
Same-site Cookie Attribute
Header Based Auth.
Click-jacking
Frame-busting JavaScript
X-Frame-Options Header
CSP: Frame Ancestors Directive
Post-Compromise Use
Logout Function
Forgetful Users
Idle Timeout
Scripted Requests
Absolute Timeout
Client Attribute Tracking
Allow Single Session
List Active Sessions

Thank You!

Slides and
Articles Suitable for Sharing with Developers
https://www.merisec.com/blog

Mark Hoopes
mark@meristeminfosec.com
https://www.linkedin.com/in/markhoopes/

Where are the Cookies Really?

Browser	Location (Windows)
Chrome	%LOCALAPPDATA%\Google\Chrome\User Data\Default\Network\Cookies
Firefox	%APPDATA%\Mozilla\Firefox\Profiles\[random ID]\cookies.sqlite
Edge	%LOCALAPPDATA%\Microsoft\Edge\User Data\Default\Network\Cookies
Opera	%AppData%\Opera Software\Opera Stable\Network\Cookies

Browser	Location (Mac)
Safari	~/Library/Containers/com.apple.Safari/Data/Library/Cookies/*.binarycookies