WEB SECURITY

A Primer for Web Developers

Sept 12, 2017

Mike North

MIKE WORKS

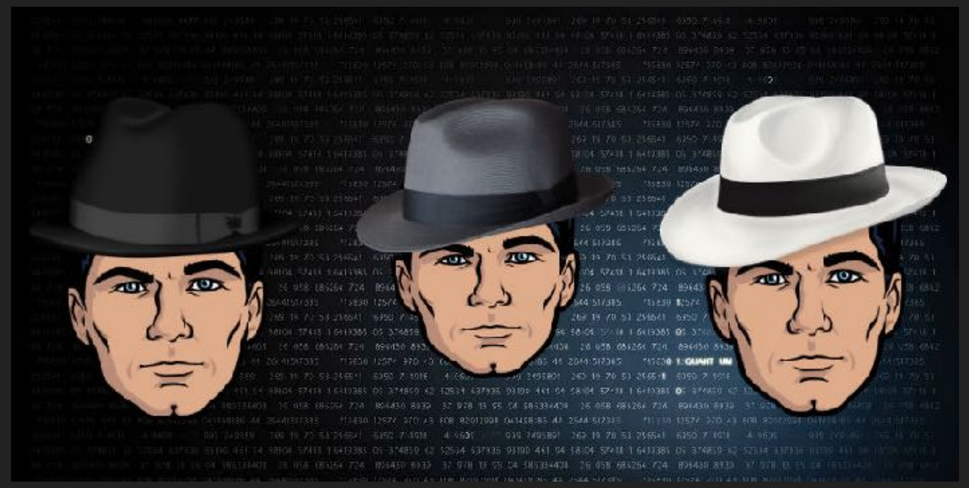
We have a BIG problem

- Features & Deadlines vs. Security
- Web developers have fallen behind
- Attacks are escalating in severity
- Barriers to staging an attack are lower than ever

Our Strawman

- Cookie-based "session" authentication
- List of accounts
- Ability to lookup other accounts & transfer funds
- IE9+

Types of Hackers

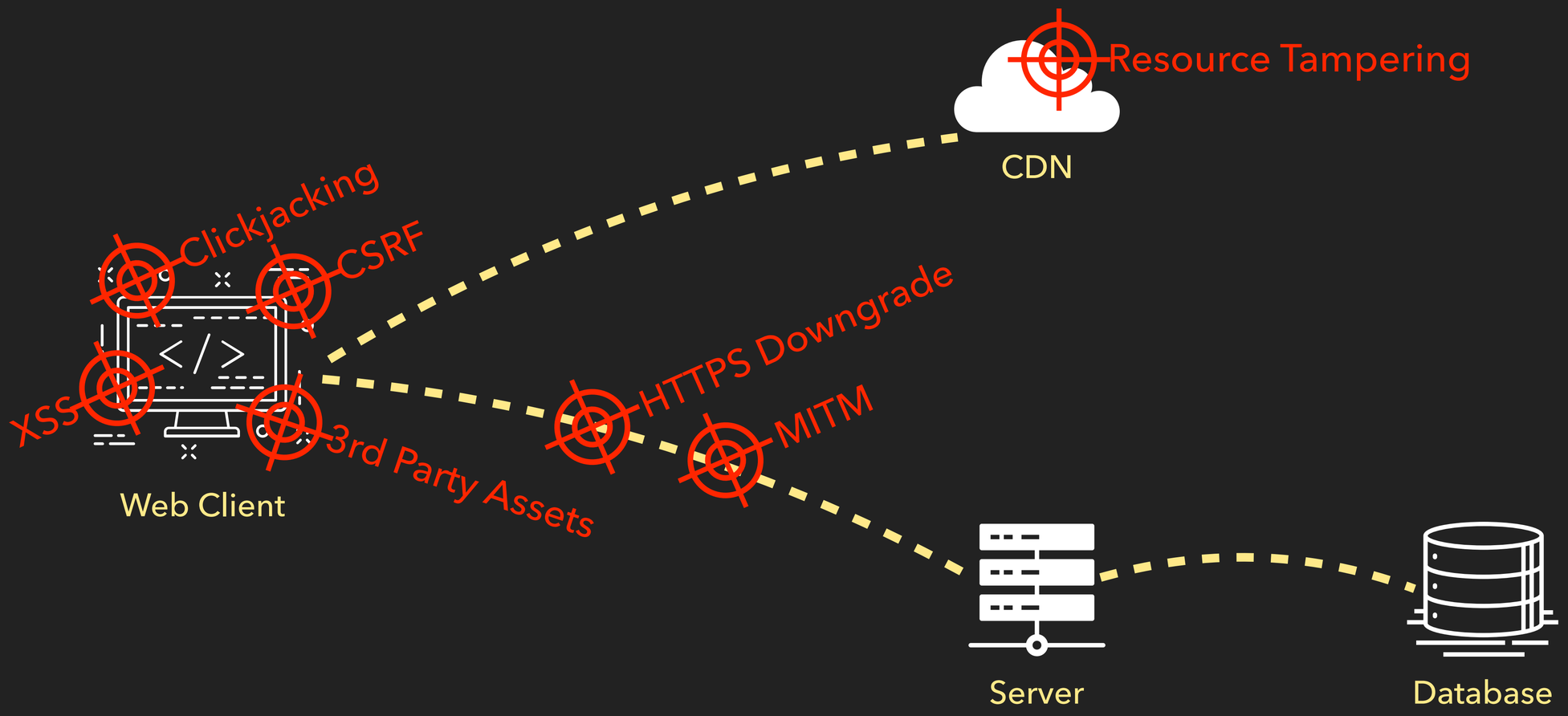


- **Black Hat** - Causes damage, hold data for ransom. Driven by personal gain, or desire to do as much damage as possible (because they can).
- **Grey Hat** - Break into systems, but usually cause no damage. Mostly driven by curiosity. Sometimes report vulnerabilities they find.
- **White Hat** - Break into systems with permission, responsibly disclose anything they find. Make money from "bug bounties" and consulting as penetration testers.

M.O. of an Attacker

- Gather information about your system
- Research vulnerabilities
- Get a foothold in the system
- Use that foothold to escalate to more serious attacks

Attack & Defend



- XSS
- ATTACHMENTS
- CSRF
- CLICKJACKING
- 3RD PARTY ASSETS

Client-Side Security

Cross-Site Scripting: XSS

- X? Because CSS already means something
- An injection attack
- Vulnerabilities are prevalent
- Allow attacker to read data, or perform operations on user's behalf

```
<h1>Welcome, <%- name %></h1>
```

```
name = "Mike";
```

```
<h1>Welcome, Mike</h1>
```

```
name = "Mike<script>terrible()</script>";
```

```
<h1>Welcome, Mike<script>terrible()</script></h1>
```

Cross-Site Scripting: XSS

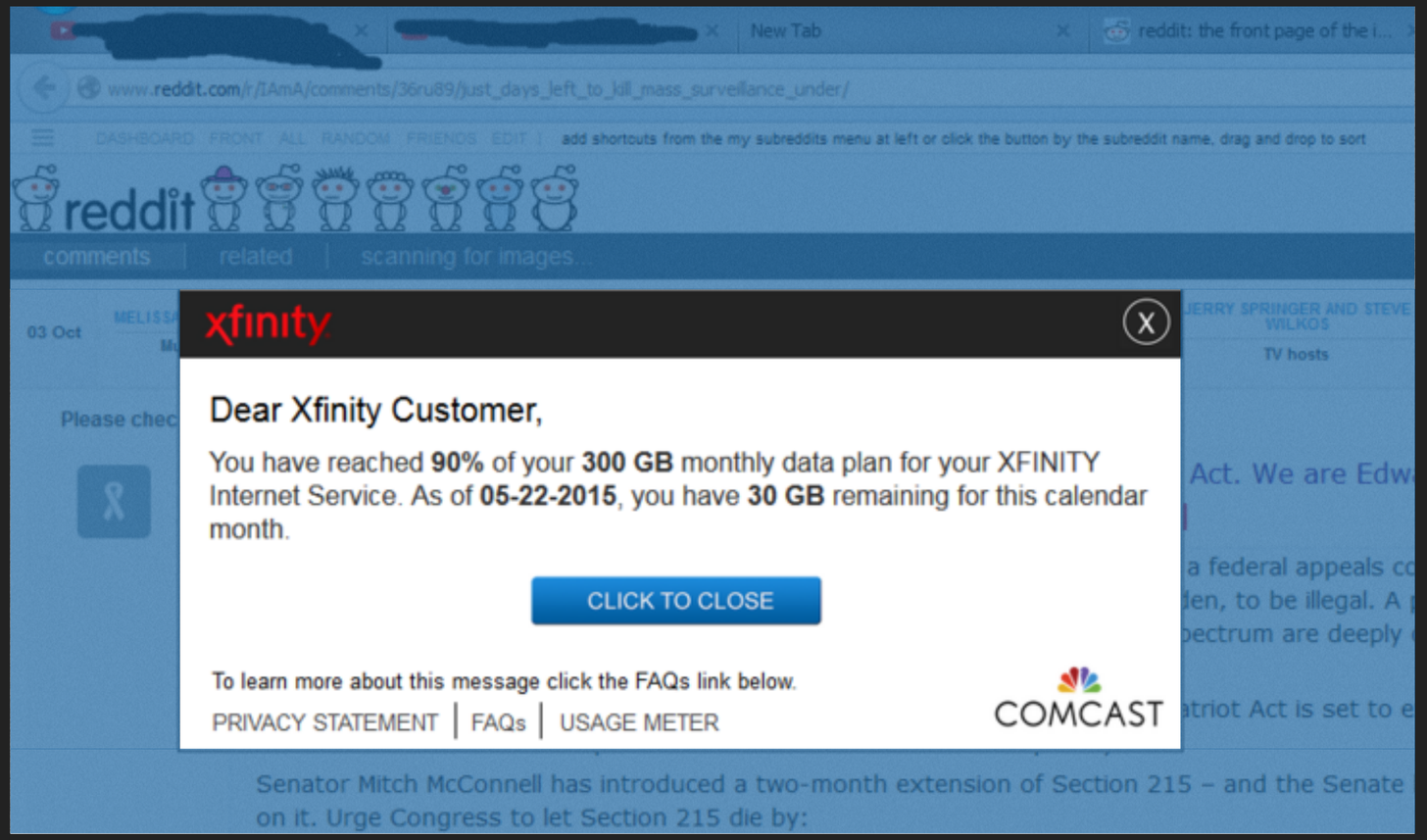
- **Stored XSS** - Code that executes attacker's script is persisted
- **Reflected XSS** - Transient response from server causes script to execute (i.e., a validation error)
- **DOM Based XSS** - No server involvement is required (i.e., pass code in via queryParams)
- **Blind XSS** - Exploits vulnerability in **another app** (i.e., log-reader), that attacker can't see or access under normal means

Cross-Site Scripting: XSS - Danger Zones

- User-generated rich text (i.e., WYSIWYG)
- Embedded content
- Anywhere users have control over a URL
- Anywhere user input is reflected back (i.e., "couldn't find __")
- Query Parameters rendered into DOM
- element.innerHTML = ?

XSS Demo

Cross-Site Scripting: XSS - As a feature??



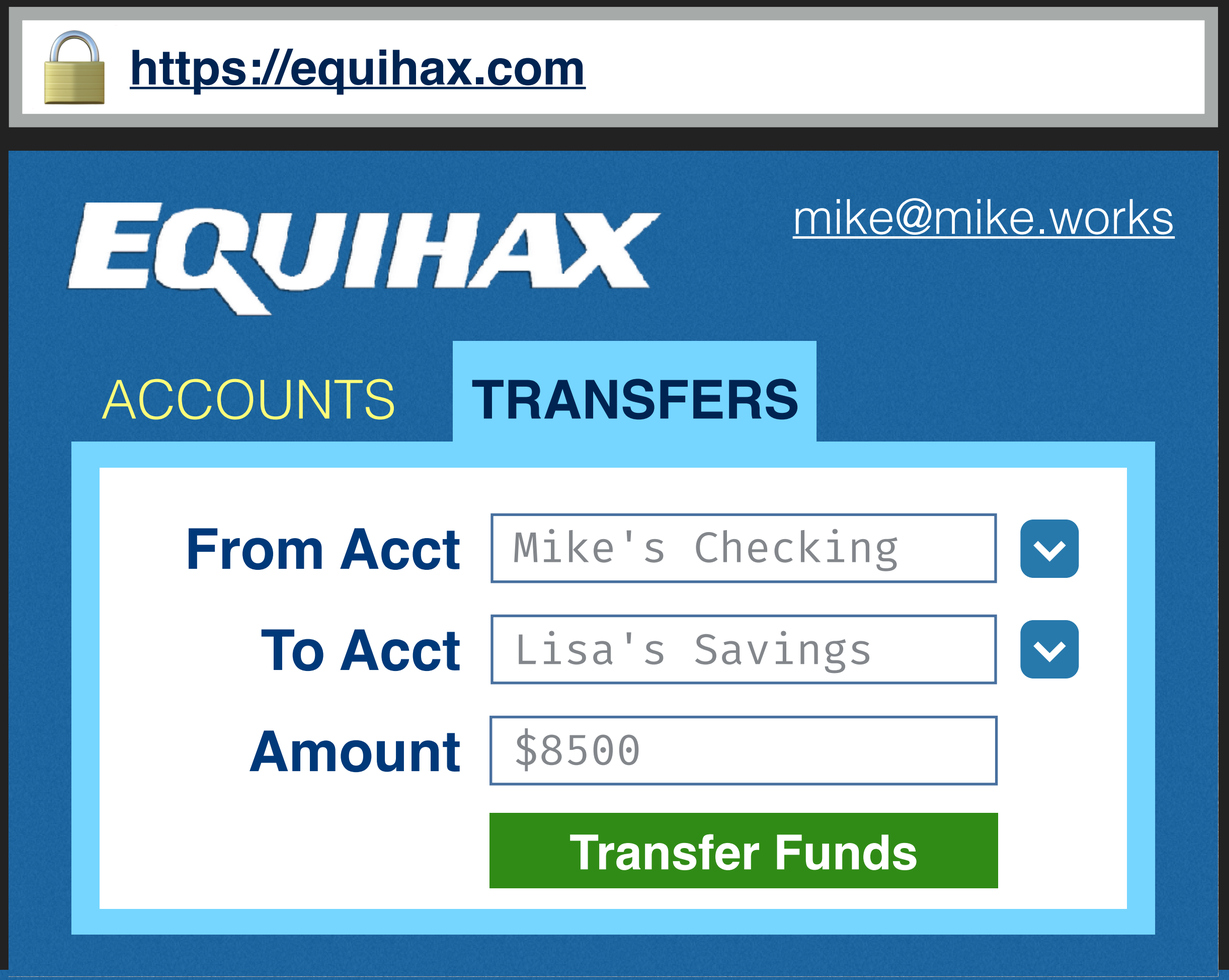
WARNING

Cross-Site Scripting (XSS): Questions for you

- How confident are you in the XSS protection of your OSS libraries?
- How carefully do people scrutinize browser plugins?
- If XSS happens, what's your exposure?
- In your apps, what could a successful XSS attack escalate to?

ATTACK

Cross-Site Scripting (XSS)




```
<select>
  <option value="1">
    Mike's Checking
  </option>
  <option value="2">
    Lisa's Savings
  </option>
  <option value="3">
    Elliot's Checking
    <script src="https:// ... totally-fine.js"></
  </option>
</select>
```

ATTACK: XSS

1

- There are at lease three Cross-Site Scripting vulnerabilities in Equihax
- Find them and cause as much trouble as you can 😈
- Simulate multiple users by using multiple browsers
- Try at least one modern browser and one legacy one
- Remember you can always host on *.lvh.me

XSS Defenses: NEVER put untrusted data in these places

- Directly in a script `<script> <%- userData %> </script>`
- In a HTML comment `<!-- <%- userData %> -->`
- In an attribute name `<iframe <%- userData %>="myValue" />`
- In a tag name `<<%- userData %> class="myElement">`
- Directly in a <style> block `<style> <%- userData %> </style>`

XSS Defenses: Escape data before putting it in HTML

```
<script>alert('hi')</script>
"%3Cscript%3Ealert('hi')%3C%2Fscript%3E"
```

- Just about every view library you've heard of does this automatically
- In the one for this class (ejs), it's the following

```
<%= "Escaped Expression" %>
<%- "Unescaped Expression" %>
<% "Non-Rendered Expression" %>
```

☠️ XSS Defenses: If you "unescape", sanitize data first ☠️

Ember, Vue	`{{{ "unescaped" }}}`
React	`return <div dangerouslySetInnerHTML={createMarkup()} />;`
Ejs	`<%- "Unescaped" %>`
Angular	`<div [innerHTML]="Unescaped"></div>`
Jade	`! This is #{"<b>unescaped</b>"}!`

XSS Defenses: Escape before putting user data in attributes

```
<div class="<%= 'UserValue' %>"> ... </div>
```

https://github.com/ESAPI/node-esapi

Be particularly careful when "templating" JS

```
<script>
  alert("Hello <%- userValue %>");
</script>
```

Content-Type: 'application/json'

XSS Defenses: Content Security Policy (CSP)

- Browsers can't tell the difference between scripts downloaded from your origin vs another. It is a single execution context.
- CSP allows us to tell modern browsers which sources they should trust, and for what types of resources
- This information comes via a HTTP response header or meta tag

```
Content-Security-Policy: script-src 'self' https://mike.works
```

XSS Defenses: Content Security Policy (CSP)

- Multiple directives are separated by semicolon
- Re-defining a directive with the same name has no effect
- By default, directives are permissive

```
Content-Security-Policy: script-src 'self' https://mike.works;
                         font-src: https://fonts.googleapis.com
```

XSS Defenses: Selection of Useful CSP Directives

- child-src - child execution contexts (frames, workers)
- connect-src - what you can connect to (fetch, WebSocket, EventSource)
- form-action - where you can <form> submit to
- img-src, media-src, object-src - where you can get images, media, flash from
- style-src - where external stylesheets can come from
- upgrade-insecure-requests - upgrades HTTP to HTTPS
- default-src - fallback, for when specific directive isn't provided

XSS Defenses: Selection of Useful CSP Directives

- There are a few keywords we can use along side sources
  - `'none'` - no sources allowed
  - `'self'` - current origin
  - `'unsafe-inline'` - allows inline JavaScript & CSS
  - `'unsafe-eval'` - allows `eval()`

XSS Defenses: CSP and 'unsafe-inline'

- Script tags embedded in HTML is the most common form of XSS. Banning it mitigates XSS risk considerably
- Sometimes we really want a little JS in our HTML. There's are a few solutions for this
- Cryptographic nonces must be generated per page load, and must change unpredictably

```
<script nonce=E9h3sdfn3f03nce9DNnIOefn3fa>
  alert('I have to do this inline for some reason');
</script>
```

```
Content-Security-Policy: script-src 'nonce-E9h3sdfn3f03nce9DNnIOefn3fa'
```

XSS Defenses: CSP and 'unsafe-inline'

- Script tags embedded in HTML is the most common form of XSS. Banning it mitigates XSS risk considerably
- Sometimes we really want a little JS in our HTML. There's are a few solutions for this
- Cryptographic nonces must be generated per page load, and must change unpredictably

```
<script>
  alert('I have to do this inline for some reason');
</script>
```

```
Content-Security-Policy: script-src 'sha256-qzn ... 1guq6ZZDob_Tng='
```

DEFEND: XSS

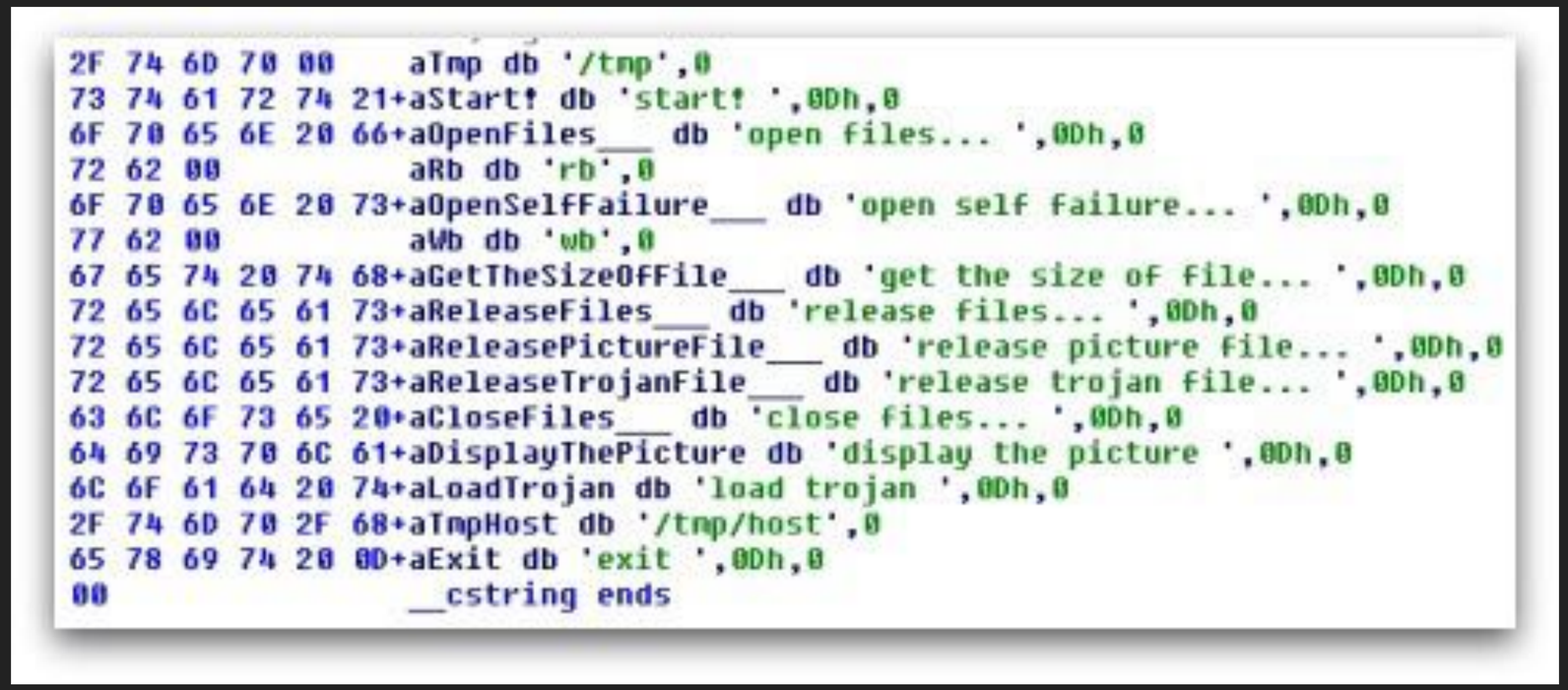
2

- Fix these three XSS bugs:
  - Notifications shown to user
  - Names of accounts, shown in the transfer window
  - User profile page - error for invalid user
- Add a reasonable CSP. **helmet-csp** is already installed
  https://github.com/helmetjs/csp
  This should be applied in `./server/index.js`

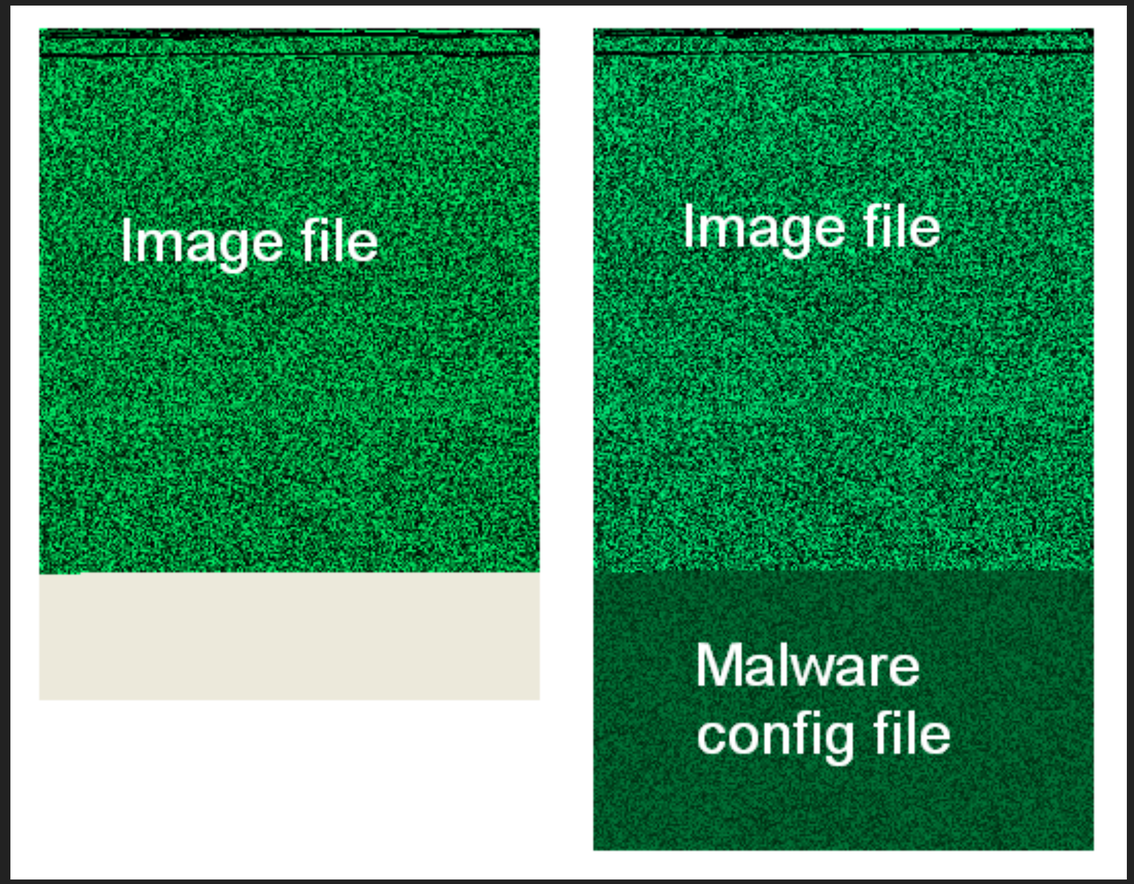
```
Content-Security-Policy:
        style-src 'self' 'unsafe-inline';
        font-src data:;
        default-src 'self'
```

ATTACK

Cross-Site Scripting (XSS): Malicious Attachments

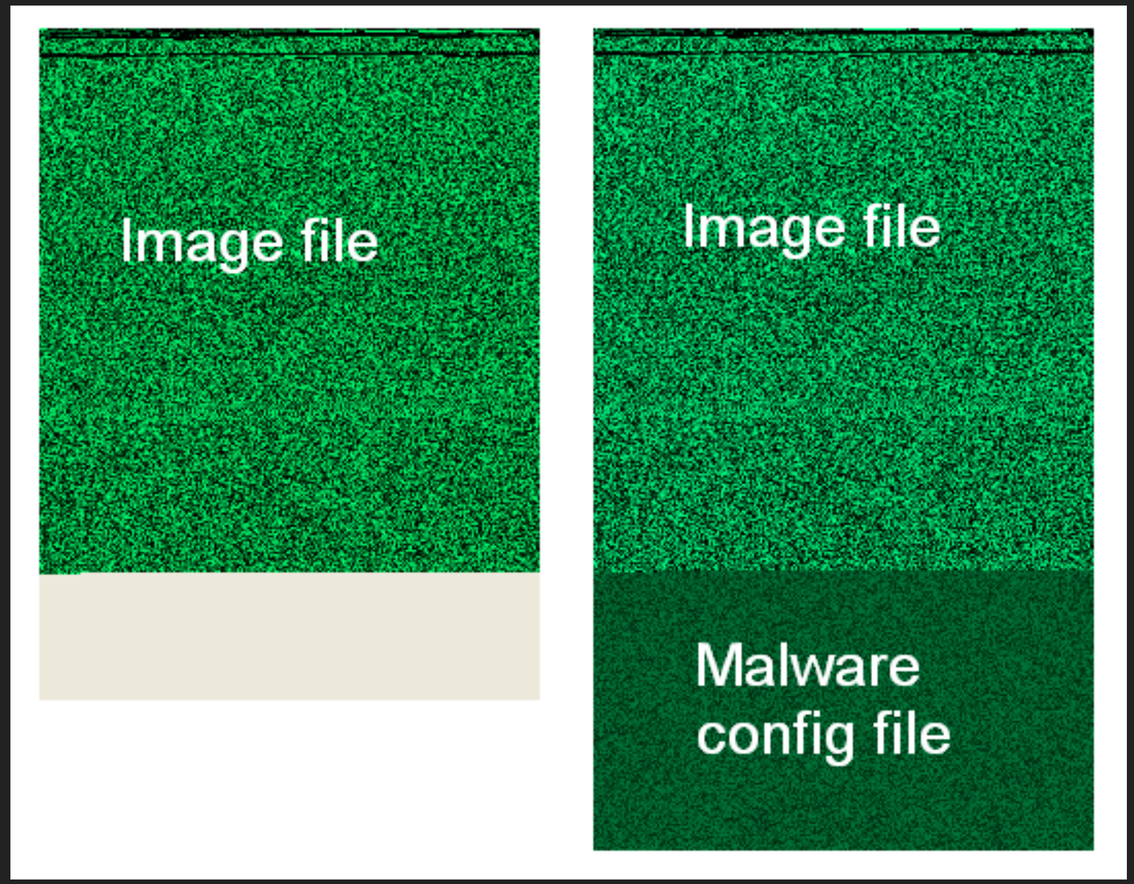


Embedded Malware



Cross-Site Scripting (XSS): Malicious Attachments

Embedded Malware



ATTACK: XSS Attachment

3

- Scenario: we allow image uploads, which are hosted on the same domain as our app. We check to ensure these are images upon users' upload http://equihax.lvh.me:3000/images/squirrel.jpg
- We give users the ability to rename their files http://equihax.lvh.me:3000/images/squirrel.html
- Check out what happens when you visit that URL
- See how much mischief you can create

ATTACK: XSS Attachment

3

- Read the EXIF comment comment field of an image, and pipe it onto your clipboard

```
node ./scripts/exif.js read ./examples/squirrel.jpg | pbcopy
```

- Replace the EXIF comment field of an image with the content of an HTML file

```
node ./scripts/exif.js write ./examples/squirrel.jpg scratch.html
```

  - NOTE: must collapse everything onto one line

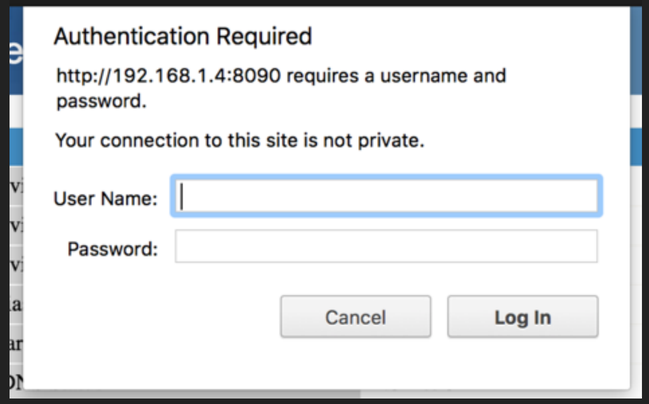
Cross-Site Scripting (XSS): Malicious Attachments

- The more restrictive you are on file upload types and ability to access those types, the less of a XSS vector your app becomes
- IMAGES: Generally, things that compress files drop non-visible data
- Before allowing other attachment types, research capabilities thoroughly
  - i.e., Did you know Acrobat PDFs could execute JavaScript?

ATTACK

Cross-Site Request Forgery

- Takes advantage of the fact that cookies (or Basic Authentication credentials) are passed along with requests.




```
<img src="http://localhost:3000/transfers/perform?
accountFrom=12&accountTo=21&amount=1" />
```

- One of several good reasons to align with REST conventions

ATTACK

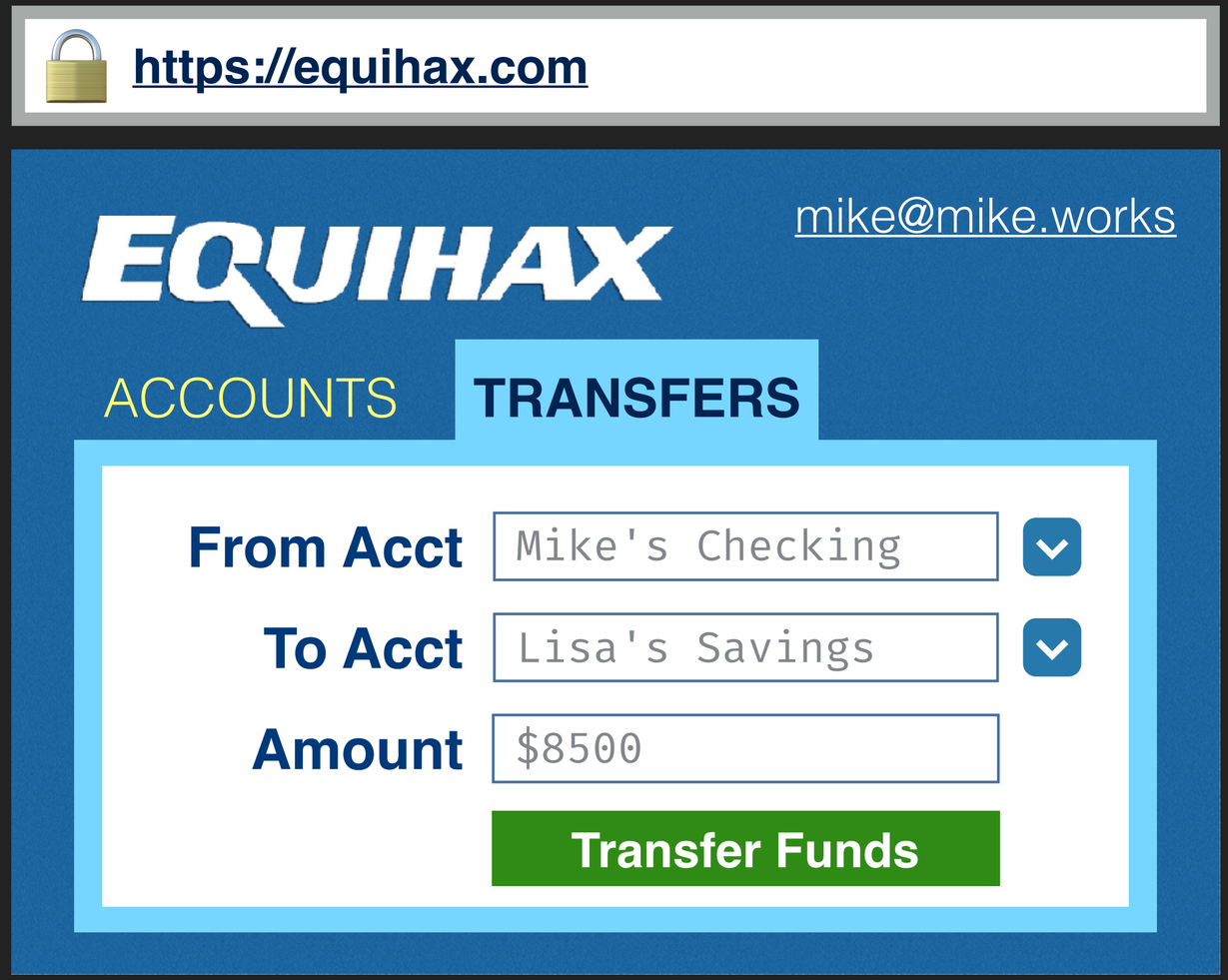
Cross-Site Request Forgery

```
<form name="badform" method="post"
  action="https://equihax.com/api/transfer">

  <input type="hidden"
    name="destination" value="2" />

  <input type="hidden"
    name="amount" value="8500" />
</form>

<script type="text/javascript">
  document.badform.submit();
</script>
```



ATTACK: CSRF

4

- See if you can make a page on jsbin... that cause bank transfers to happen for logged-in users
- Try this with both the GET request and POST request flavors of CSRF attack
- See how much damage you can do 😈

Cross-Site Request Forgery: Who's Vulnerable?

- Only `Basic` or cookie authentication schemes are vulnerable
  - Exception: "Client side cookie"
- KEY CONCEPT: **using** cookies doesn't require the ability to **read** cookies
- localStorage/sessionStorage - alternatives that don’t have this problem

Cross-Site Request Forgery: CSRF Token

- Changed with each request in an unpredictable way
- **KEY CONCEPT**: it's a value that's **not** in a cookie, which something with access to the frame (i.e., your JavaScript app) can see/use
  - Kind of like two-factor?
    1. something proving you're authenticated
    2. something proving you're sending a request from an appropriate place
- For server-rendered apps: meta tags are fine

Cross-Site Request Forgery: Validate Request Origin

- Modern browsers send an `Origin` header, which cannot be altered by client-side code, with each request (IE11- does not in some cases)
- In cases where there's no `Origin` header, there's almost always a `Referer` header.
  - YES it's misspelled

> Its like when I did the referer field. I got nothing but grief for my choice of spelling. I am now attempting to get the spelling corrected in the OED since my spelling is used several billion times a minute more than theirs.

- When behind a proxy, you can usually get some information from `Host` and `X-Forwarded-Host` headers

Cross-Site Request Forgery: CORS

```
// Preflight REQUEST to server
Origin: http://foo.example
Access-Control-Request-Method: POST
Access-Control-Request-Headers: Authorization, Content-Type

//                Preflight RESPONSE from server
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Methods: POST, PUT, GET, OPTIONS
Access-Control-Allow-Headers: Authorization, Content-Type
Access-Control-Max-Age: 86400

// Main REQUEST to server
Origin: http://foo.example
Access-Control-Request-Method: POST
Access-Control-Request-Headers: Authorization, Content-Type
Authorization: Bearer 12kfbdk1j2d1bj2dkjb12d12d
Content-Type: application/json
```

- **C**ross-**O**rigin **R**esource **S**haring
- A preflight OPTIONS request gives server a chance to indicate what's allowed
- Main request follows

DEFEND: CSRF

5

- Add a CSRF token to equihax, such that our previous hacks are now impossible
  - https://github.com/expressjs/csurf is already installed for you
    - Most of your work will be done in `./server/routes/transfers.js` and `./server/views/transfers.ejs`
- Experiment with IE9 if you have it setup

Clickjacking

ATTACK

Clickjacking

https://equihax.com?amount=8500&dest=185...

EQUIHAX

mike@mike.works

ACCOUNTS | TRANSFERS

From Acct: Mike's Checking

To Acct: Lisa's Savings

Amount: $8500

Transfer Funds

https://equihax.com-securebank.cc?tok=108...

You're eligible for Identity Theft protection

Proceed

ATTACK

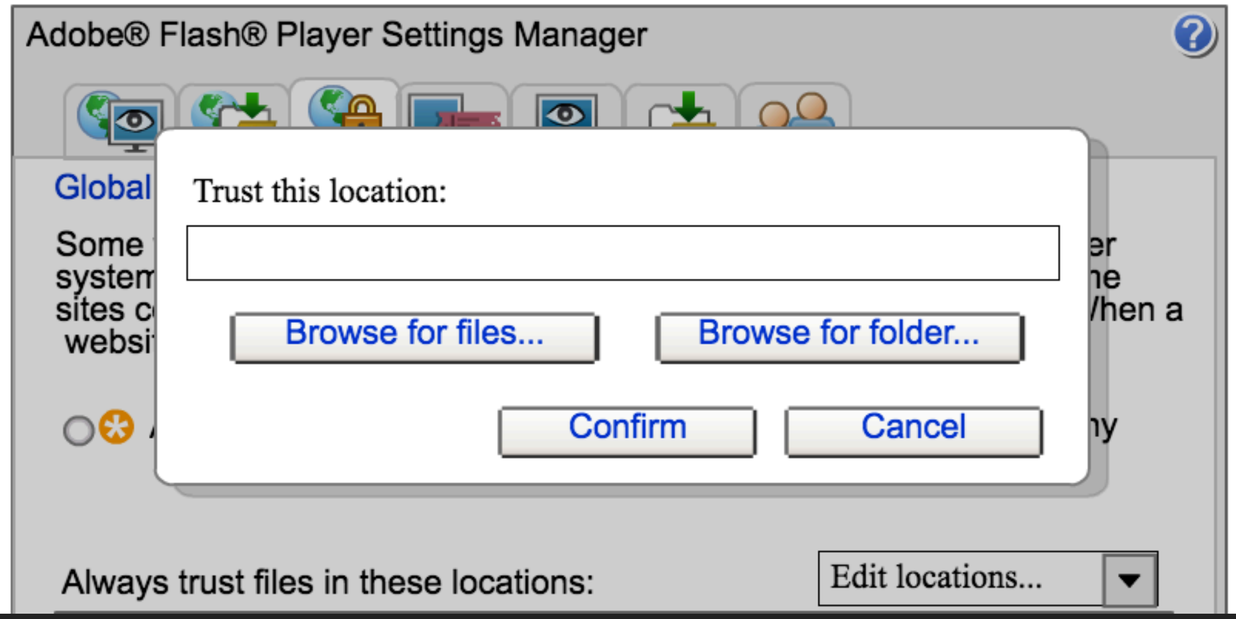
Clickjacking

Flash Player Help

Settings Manager
- Global Privacy Settings panel
- Global Storage Settings panel
- Global Security Settings panel
- Website Privacy Settings panel
- Website Storage Settings panel
- Protected Content Playback Settings panel
- Peer-Assisted Networking Panel

TABLE OF CONTENTS

Global Security Settings panel



Clickjacking

- A "UI redress attack"
- Can be used to capture keystrokes as well

ATTACK: Clickjacking

6

- Our "transfer funds" form can be pre-populated via queryParams http://localhost:3000/transfers?fromAccount=16&accountTo=21&amount=1400
- Using http://jsbin.com/, create a landing page that uses the clickjacking technique to trick a user into unintentionally performing an action
- Imagine how powerful this technique could be, when combined with an XSS attack
- Curl up into a fetal position
- Cry

Clickjacking Defenses: Modern Browsers

- `X-Frame-Options` HTTP response header

```
X-Frame-Options: DENY
X-Frame-Options: SAMEORIGIN
X-Frame-Options: ALLOW-FROM https://equihax.com/
```

- Chrome/Safari don't respect allow-from. Use `frame-ancestors` CSP directive instead
- This applies to the TOP LEVEL frame.

Clickjacking Defenses: Legacy Browsers

```
<style id="clickjack">
  body{
    display:none !important;
  }
</style>
```

```
if (self === top) {
  var cj = document.getElementById("clickjack");
  cj.parentNode.removeChild(clickjack);
} else {
  top.location = self.location;
}
```

DEFEND: Clickjacking

7

- Add modern and legacy browser defenses for clickjacking
- Verify that your attacks from the last exercise no longer work
- You may need to alter your CSP if any inline scripts are added

3rd Party Assets

Third Party Assets

```
<script src="http://cdnjs.cloudflare.com/ajax/libs/jquery/3.2.1/core.js">
```

```
"express": "^4.15.2",
```

```
<script>
(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
(i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new Date();a=s.createElement(o),
m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m
})(window,document,'script','https://www.google-analytics.com/
analytics.js','ga');
</script>
```

Third Party Assets: Version Changes

```
"express": "^4.15.2",
```

- The people who write your dependencies make mistakes
- RECOMMENDATIONS:
  - Reproducible builds, with a **lockfile**
  - Use LTS versions where you care less about bleeding edge features
  - Support **BUG BOUNTIES** in important open source projects
  - Tests that assert only expected requests are sent out

Third Party Assets: CDN Assets

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.2.1/core.js">
```

- If you must use a CDN like this, be aware you're trusting the owners a lot
- Use Subresource Integrity attributes on <script> and <link> tags

```
Failed to find a valid digest in the 'integrity' attribute for resource 'https://cdnjs.cloudflare.  lvh.me/:1
com/ajax/libs/jquery/3.2.1/core.js' with computed SHA-256 integrity
'BSsbXsDErniq/HpuhULFor8x1CpA2sPPwQLlEoEri+0='. The resource has been blocked.
```

Don't pass credentials →

```
<script integrity="sha256-oqVuAPzQho1wx4JwY8wC"
  crossorigin="anonymous"
  src="...../jquery/3.2.1/core.js"></script>
```

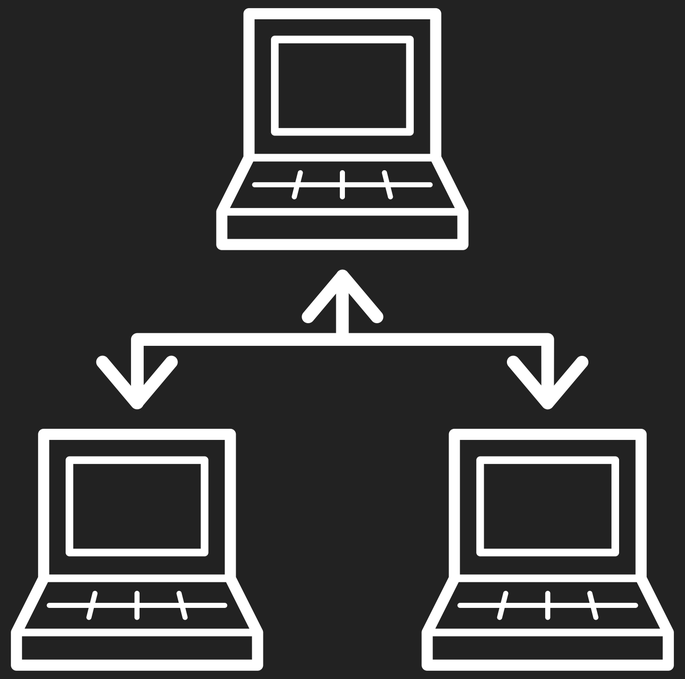
Third Party Assets: Vendor Tags

- These can be updated independently of your deploys!
- Definitely avoid adding scripts that add more scripts
- When "fail secure" is desired, add your own SRI to the script tags
- Ask that your vendors VERSION scripts, so you have control over when new code lands (and your SRI doesn't break)

DEFEND: Subresource Integrity

8

- Add SRI attributes to the script and style tags for our material design library
- Add some trivial code to either file
- Observe how the browser now refuses to fully load the resources



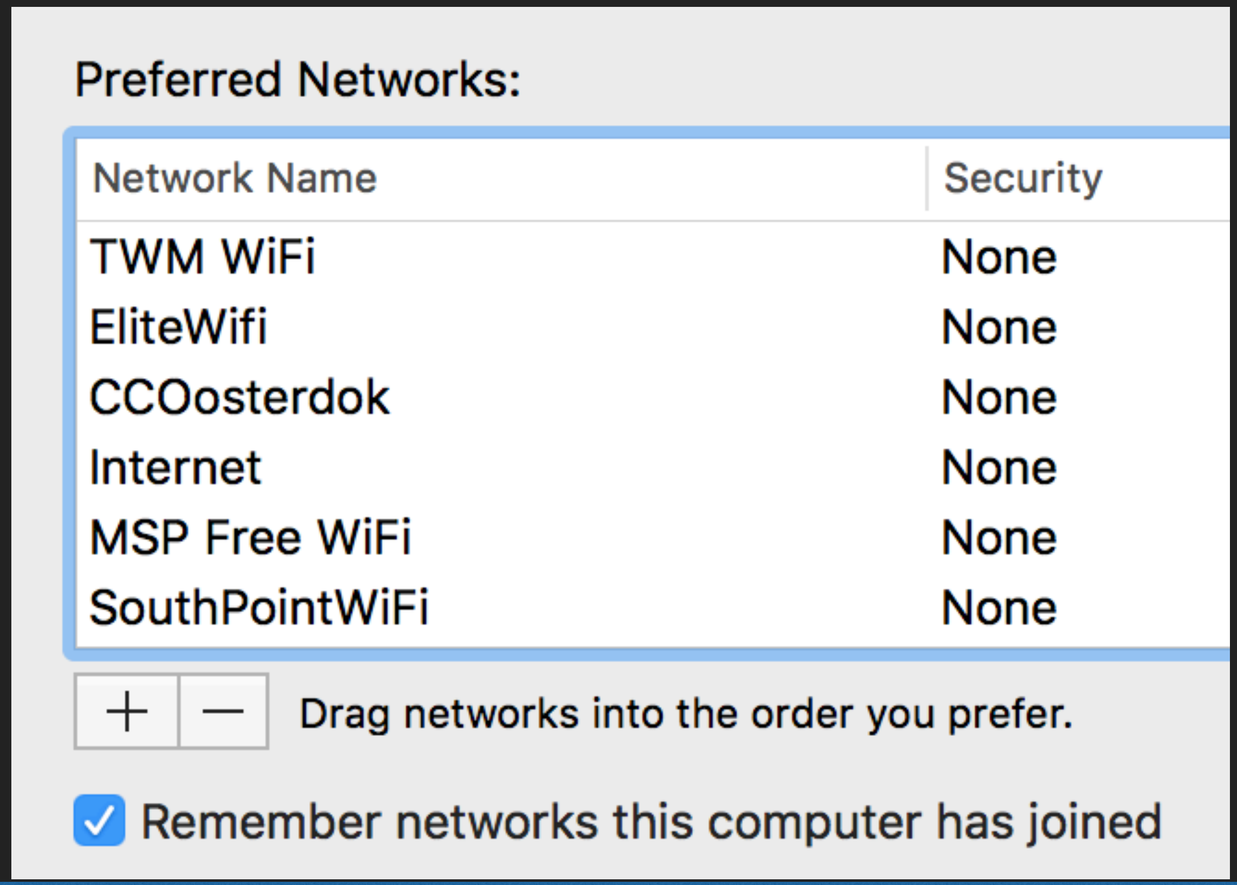
- MAN IN THE MIDDLE
- TLS (HTTPS)
- HTTPS DOWNGRADE

Network Security

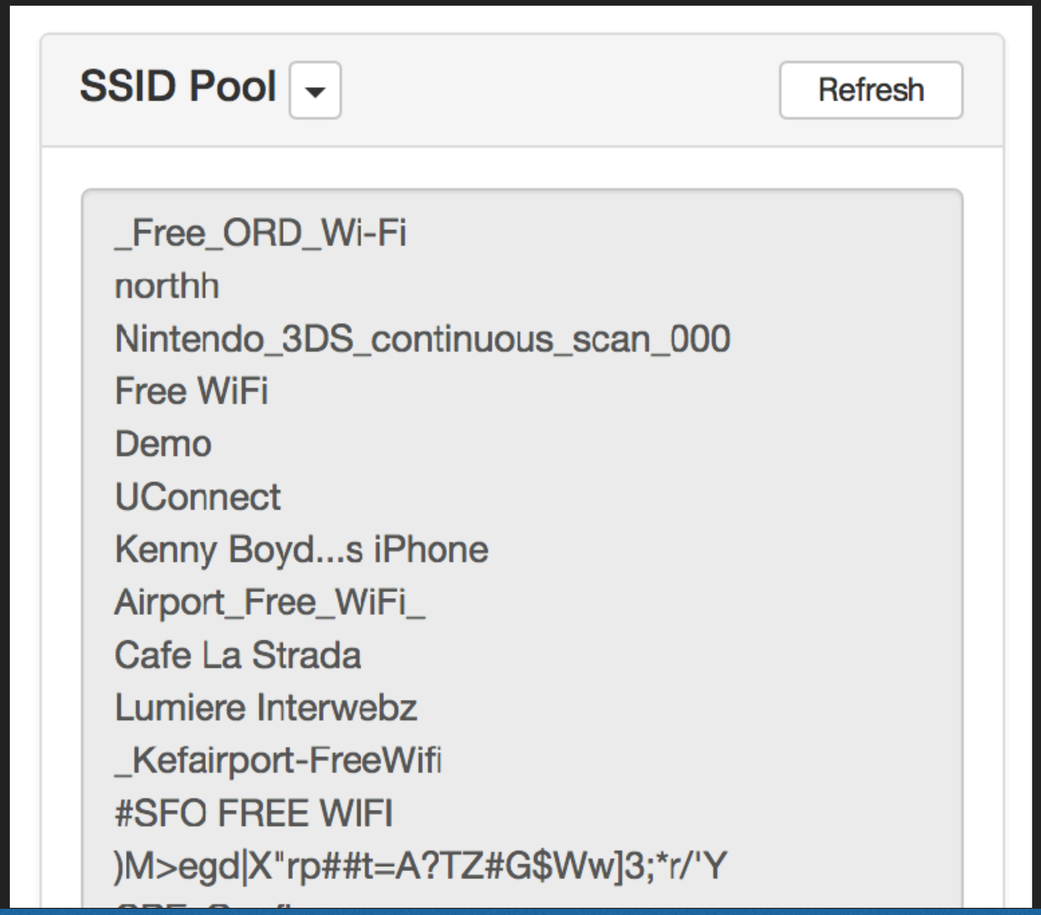
ATTACK

Man in the middle

Public WiFi: Trusted forever by default

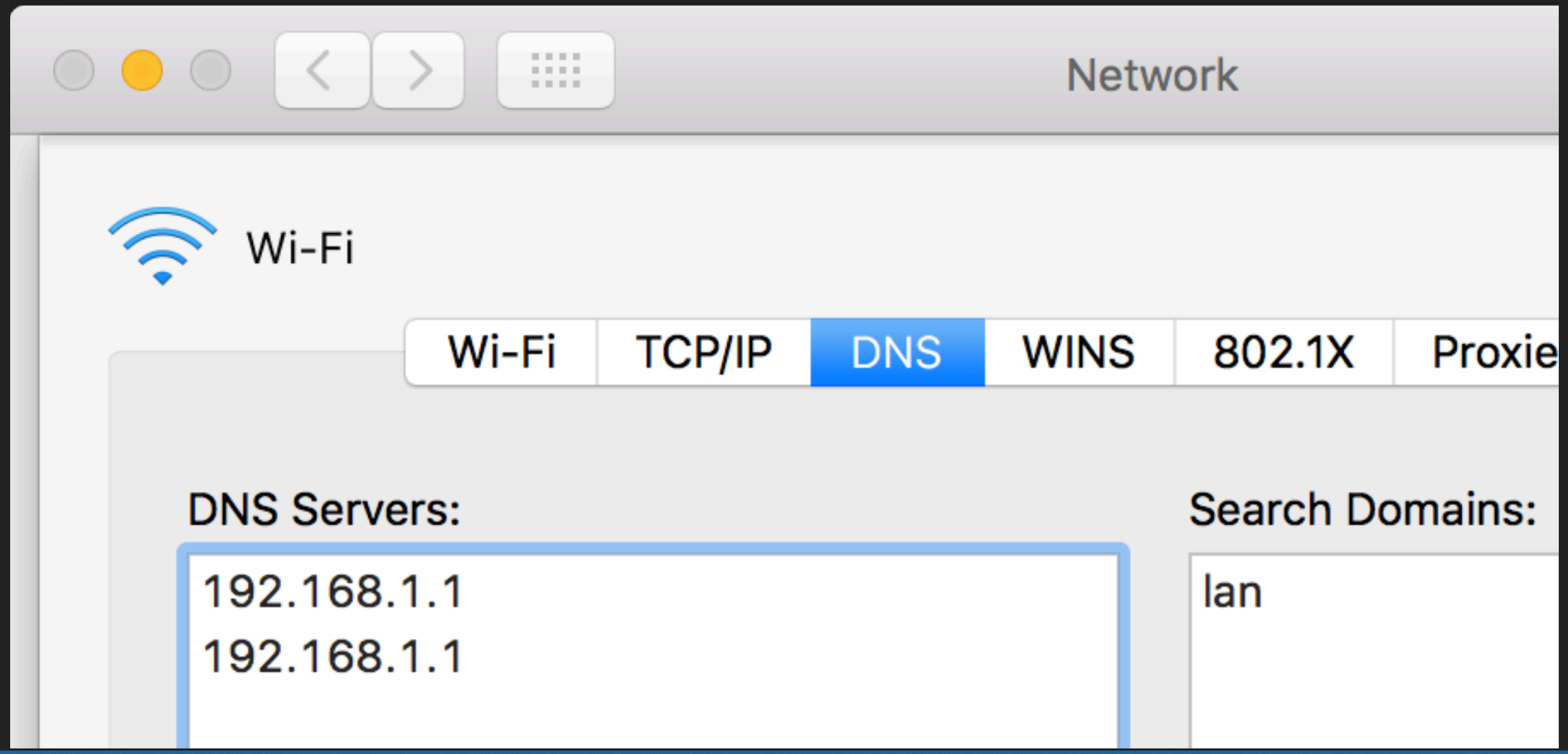


WiFi Devices broadcast what they're looking for



ATTACK

Router as DNS



ATTACK

DNS Hijacking (Poisoning)

ATTACK

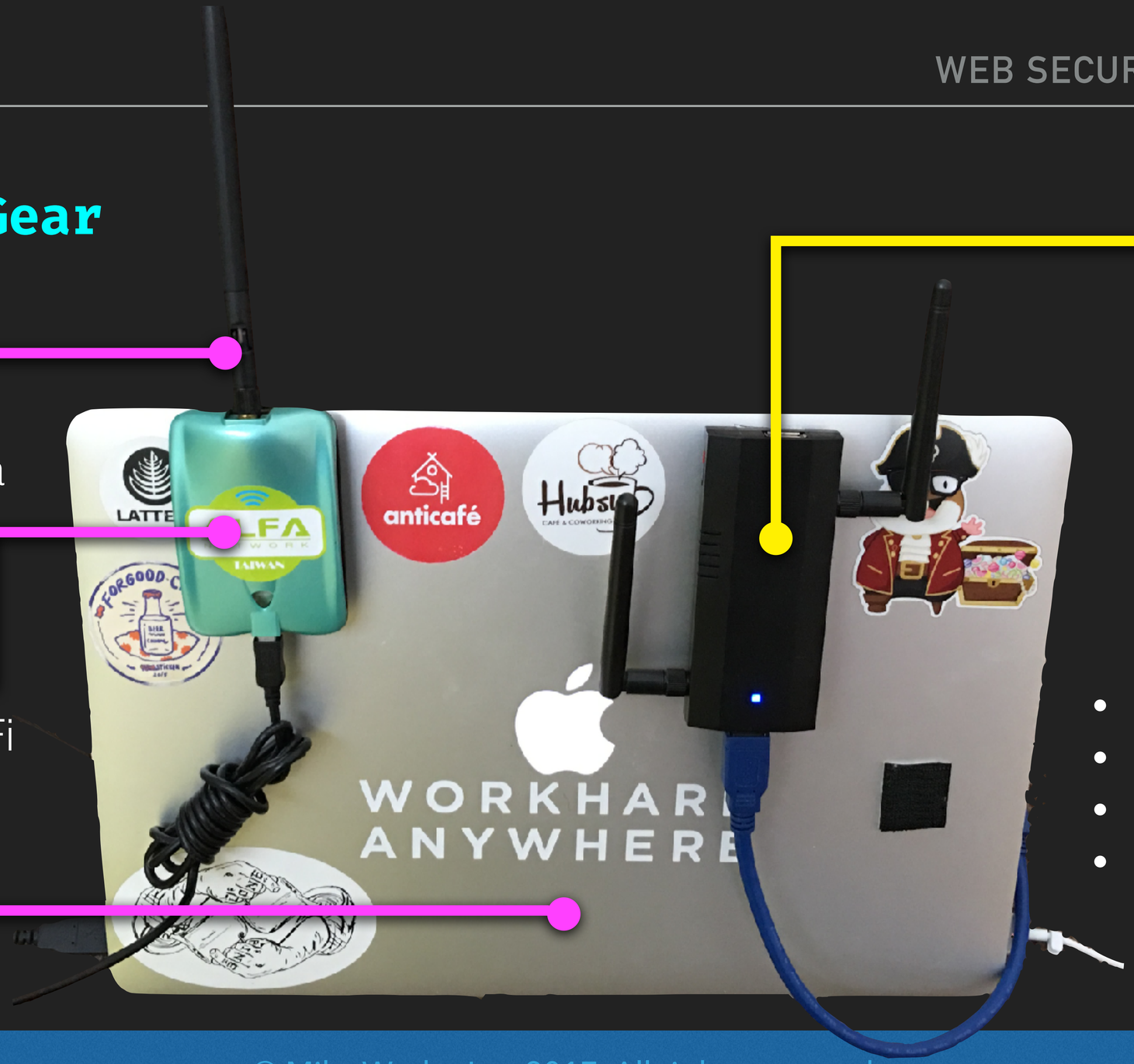
Man in the middle

What can an attacker do at this point?

- Eavesdrop on and tamper with communication between you and one or more servers
- XSS at will
- Capture your credentials and mess stuff up later
  - Try your credentials on other sites

ATTACK

Pentest Gear



WiFi Pineapple

- Linux
- 2x Wifi Cards
- High gain antennas
- "App store"

Pentest Gear

Pentest Gear

Ominous Box for WiFi Pineapple

$5.00 ~~$10.00~~

Let's say you've locked down 

ATTACK

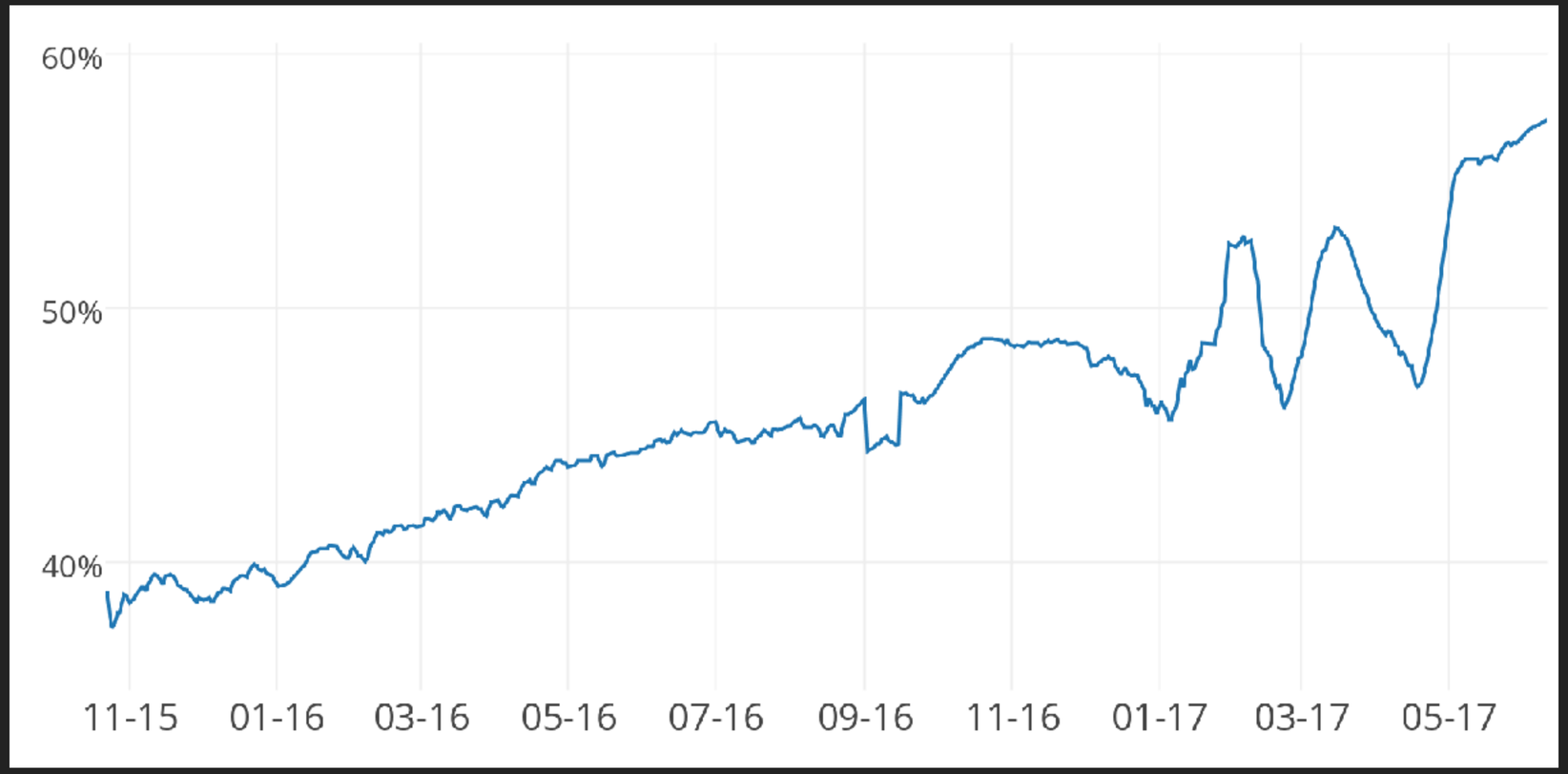
Femtocell

Man in the middle defense: Encrypt data in flight

- ~~SSL~~ TLS
- A secret key needed to read or alter request/response
- Certificates identify domains, and require "Domain Validation"
- "Enhanced Validation" often requires government ID, but that's just issuer policy

~56% of the web uses HTTPS

% of page loads over HTTPS



Time

HTTPS: how we got here

- 1990 - HTTP
- 1993 - Secure Mosaic implements Secure HTTP (S-HTTP)
- 1994 - Netscape invents and implements SSL, and HTTPS on top of it
- 1999 - IETF adopts SSL 3.1 known as Transport Layer Security 1.0
- 2008 - TLS 1.2 (SSL 3.3)

HTTPS: Cryptography

- Two types of encryption involved: Symmetric encryption and Public Key encryption
- Symmetric encryption is WAY faster, has no practical limit on size of content. If you've ever locked a file with a password, you've probably used this
- If an encryption key were generated on a per-connection basis, we'd almost have what we need, except...
- One catch: how do we safely share the key?

HTTPS: Cryptography

- We begin with public key encryption, just for the key exchange.
  - This is slow, but necessary in that a system can receive encrypted messages without divulging how to SEND them
- Server sends its public key 🔑 and certificate 📜 to the client
- Client and server then compare "cipher suites" 🍊🥃🍒
- Finally a session key 🔢 is generated by the client, encrypted with the server's public key 🔑 so only the server can read it
  - The session key 🔢 is what's used for encrypted data exchange.

HTTPS: Cryptography & Public Key Encryption

- Public key for WRITING messages
- Private key for READING messages
- RSA algorithm: product of two huge prime numbers
- Because exponential math is involved, practical limit on size of message
- Private keys can be used to SIGN messages

encrypt **17.47** ^ 2 = ?

decrypt √**175.5625** = ?

sign √**64**

verify **8** ^ 2 = 64

TLS Handshake

Client

Hi, I'd like to communicate securely. I understand ciphers 🍊, 🥃, and 🍒

Server

Hi! Let's use 🥃.
Here's my certificate 📜 to prove my identity, signed by someone you probably trust.
Also, here's my public key 🔑 so you can send me secret stuff

Your certificate 📜 checks out. It must really be you!
Here's a big random number 🔢, encrypted with your public key 🔑.
I'll be encrypting everything I say from now on with 🔢.

Fantastic. I'll be encrypting everything I say with 🔢 from now on

MITM Defense: OpenSSL

- Industry standard library for crypto
- DON'T implement your own algorithm, handshake, protocol, etc...
- OpenSSL is not user friendly
- You won't need it all that often

MITM Defense: OpenSSL

- Generate a private key

```
openssl genrsa -aes128 \
   -out my-private.key 2048
```

- Generate a public key from private key

```
openssl rsa -pubout \
   -in my-private.key \
   -out my-public.key
```

- Make a new Certificate Signing Request

```
openssl req -new \
   -key my-private.key \
   -out my-request.csr
```

- Sign the certificate with your private key

```
openssl x509 -req -days 3 \
   -in my-request.csr \
   -signkey my-private.key \
   -out my-certificate.crt
```

DEFEND: Man in the middle 9

- Generate a 2048 bit RSA private/public key pair, valid for 1 day
- Use the Node.js HTTPS module to serve our app over HTTPS

```
let server = https.createServer({
  cert: 'filename',
  key: 'filename',
  passphrase: 'key-passphrase'
}, app);
```

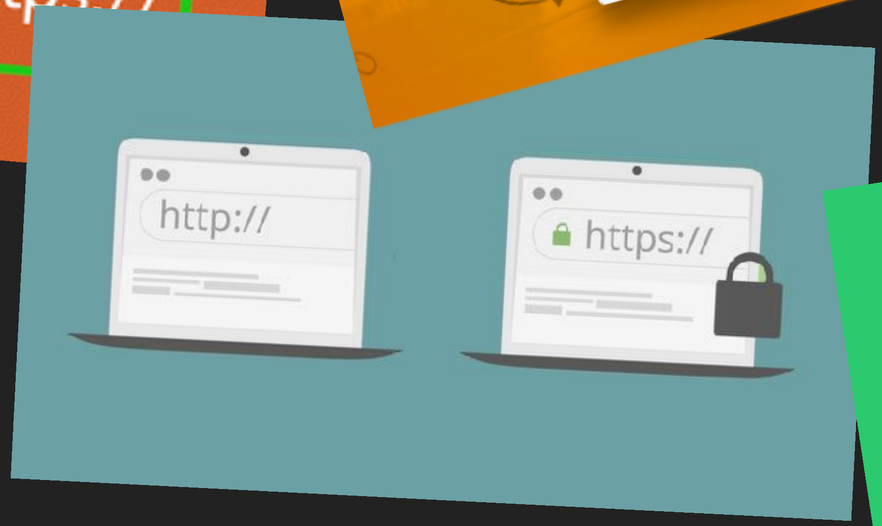
- Add the certificate to your operating system's trust store (Instructions for OS X, Windows 7, Windows 8/10)

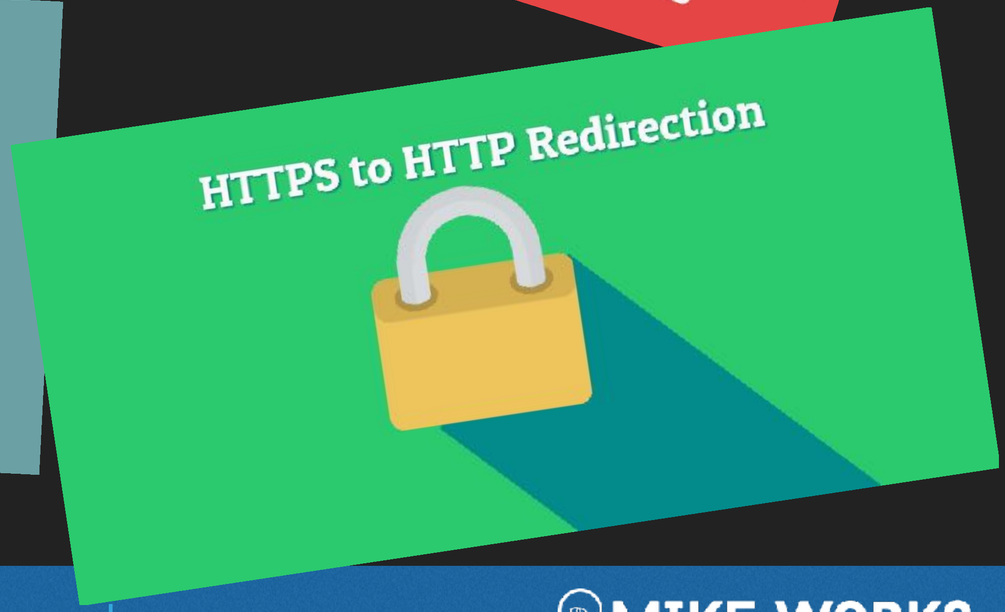
HTTPS downgrade

HTTPS Downgrading

- It's fantastic that the world is moving to HTTPS
- But – what do we do about incoming plain HTTP connections?
- Typical approach is....

HTTPS Downgrading





HTTPS Downgrading

- Seems good, but what happens if a user clicks on a plain http link?
- Initial request is over HTTP
- Server responds with a 301 Redirect to HTTPS

ATTACK

HTTPS Downgrading

- The initial request is still vulnerable to a man in the middle
- From server's perspective, connection is secure
- You're now depending on your user noticing the 🔒URL

HTTPS Downgrading

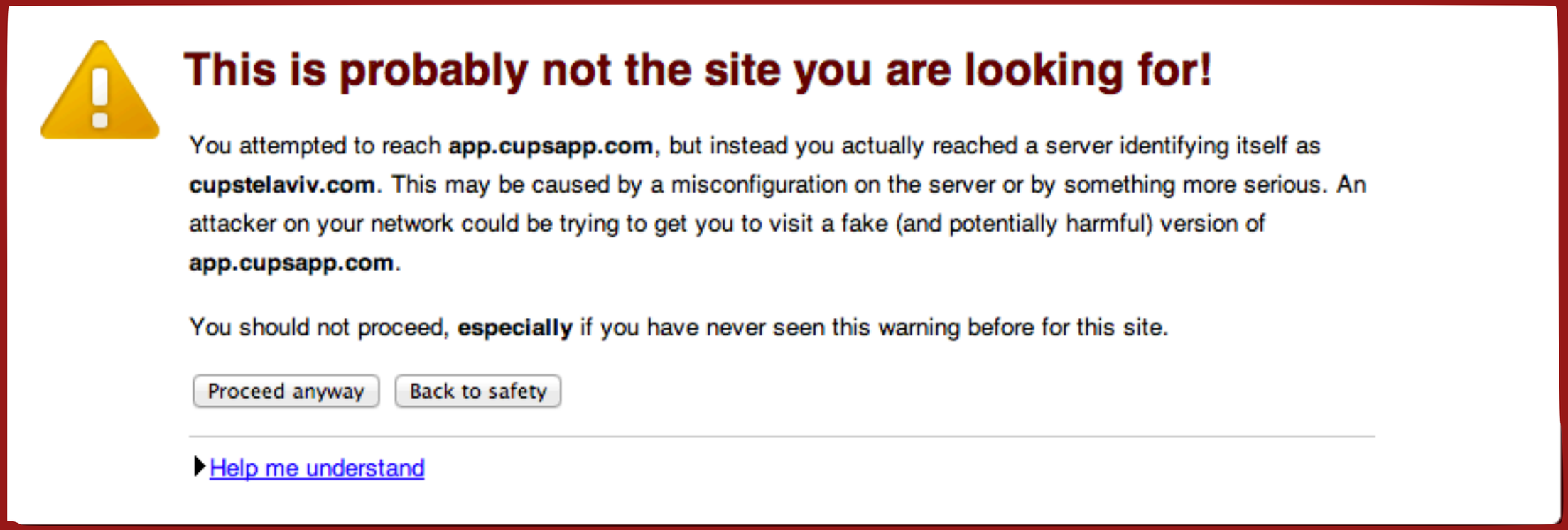
- Bookmarks, browser autocomplete remain HTTPS
- `Content-Security-Policy: upgrade-insecure-requests`
- Search engines now favor HTTPS.
- Browser plugins attempt a "secure upgrade" whenever possible

ATTACK

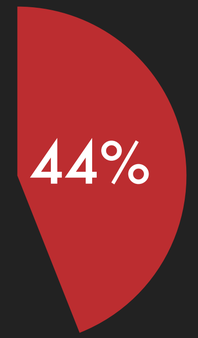
HTTPS Downgrading: With a bad certificate

- Even if a user takes these precautions, an attacker can still forge a certificate
- Server Name Indication (SNI) extension to TLS allows attackers to see the hostname a client would like to talk to
- Attackers still won't know anything else, but it's usually enough!

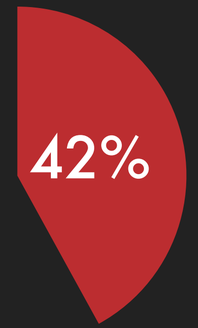
HTTPS Downgrading: With a bad certificate



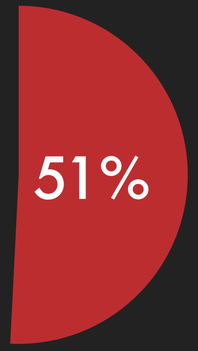
"...at least 44 percent of the top 382,860 SSL-enabled websites had certificates that would trigger warnings" [1]



After extensive data-driven improvement to Chrome warning messages, 42% of users ignore them instead of over 70% [2]



Over 50% of users don't understand eavesdropping vs. malware risk factors [3]



HTTPS Downgrading: With a bad certificate

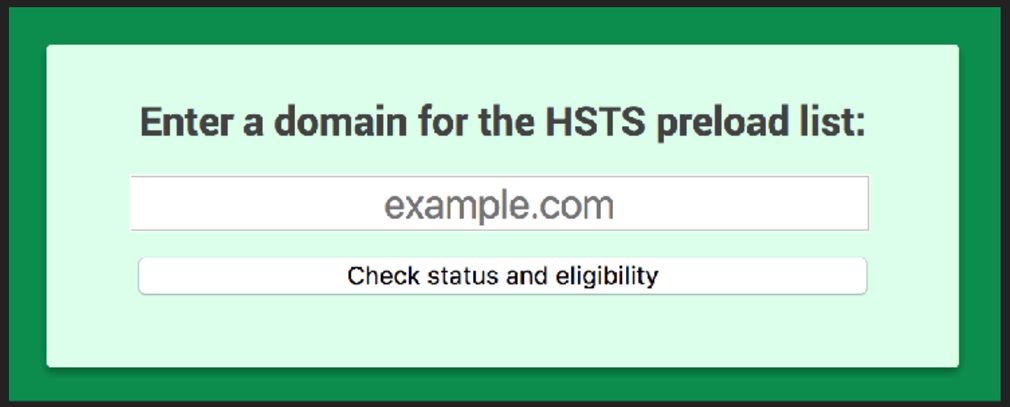
- There's a HTTP response header for that!

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

- For the specified time (in seconds) forbid the browser from making a plain HTTP connection to your domain
- Strongly recommend `includeSubDomains` option, to prevent a broad range of cookie manipulation attacks
- But what about before a user get the first response from your domain?

HTTPS Downgrading: With a bad certificate

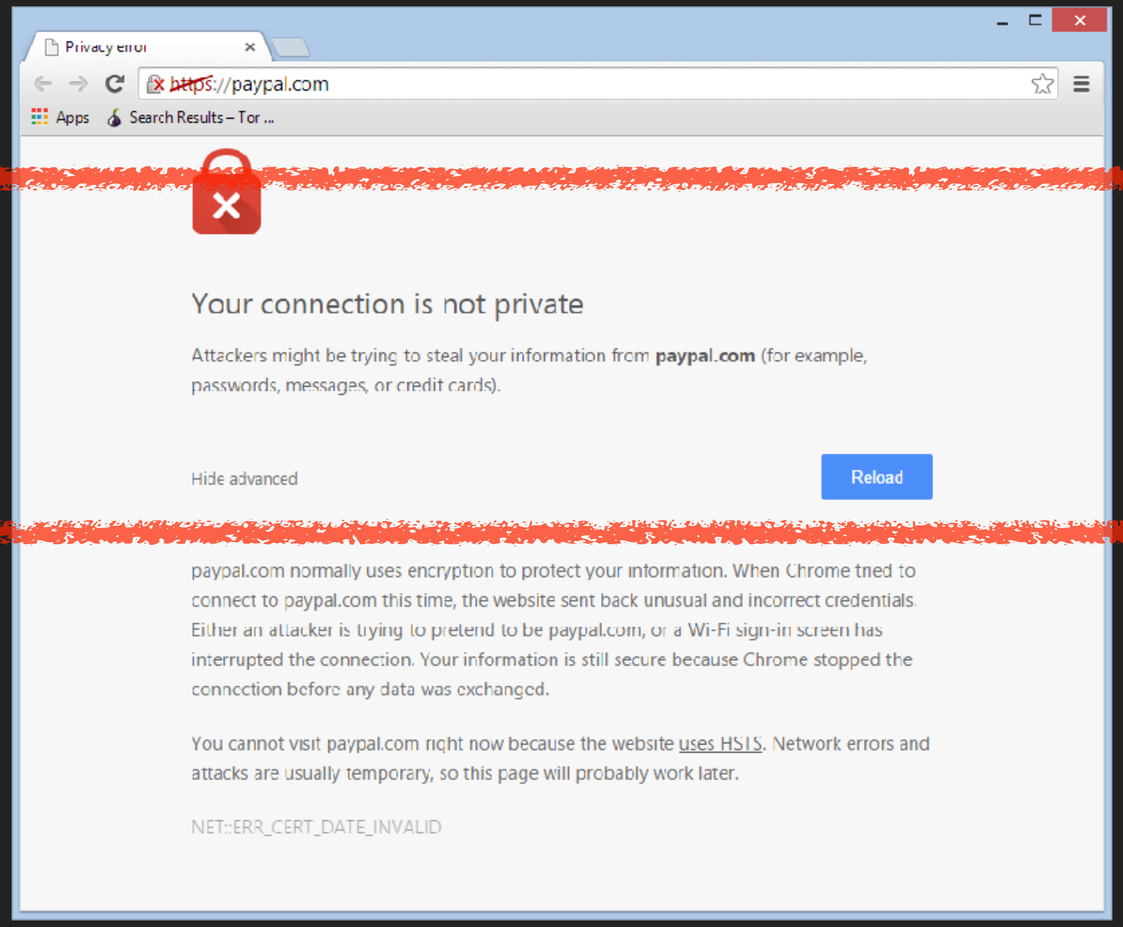
- Add your domain to the HSTS Preload list - https://hstspreload.org/
- Chrome, Firefox, Opera, Safari, IE 11, and Edge all add this list into their source code




```
{ "name": "www.paypal.com", "mode": "force-https" },
{ "name": "paypal.com", "mode": "force-https" },
{ "name": "www.elanex.biz", "mode": "force-https" },
{ "name": "jottit.com", "include_subdomains": true, "mode": "force-https" },
{ "name": "sunshinepress.org", "include_subdomains": true, "mode": "force-https" },
{ "name": "www.noisebridge.net", "mode": "force-https" },
{ "name": "neg9.org", "mode": "force-https" },
{ "name": "riseup.net", "include_subdomains": true, "mode": "force-https" },
{ "name": "factor.cc", "mode": "force-https" },
{ "name": "members.mayfirst.org", "include_subdomains": true, "mode": "force-https" },
{ "name": "support.mayfirst.org", "include_subdomains": true, "mode": "force-https" },
{ "name": "id.mayfirst.org", "include_subdomains": true, "mode": "force-https" },
{ "name": "lists.mayfirst.org", "include_subdomains": true, "mode": "force-https" },
{ "name": "webmail.mayfirst.org", "include_subdomains": true, "mode": "force-https" },
{ "name": "roundcube.mayfirst.org", "include_subdomains": true, "mode": "force-https" },
{ "name": "aladdinschools.appspot.com", "mode": "force-https" },
{ "name": "ottospora.nl", "include_subdomains": true, "mode": "force-https" },
{ "name": "www.paycheckrecords.com", "mode": "force-https" },
{ "name": "lastpass.com", "mode": "force-https" },
{ "name": "www.lastpass.com", "mode": "force-https" },
```

DEFENSE

HSTS WARNING



DEFEND: HTTPS Downgrade

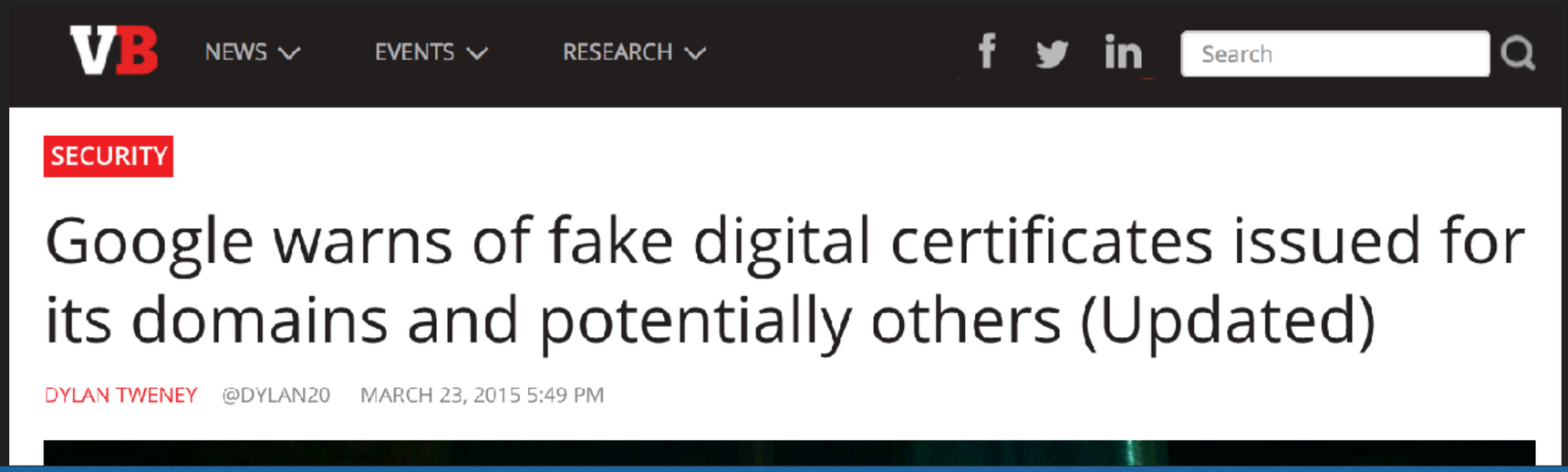
10

- Add HSTS headers to the Equihax app.
- Use a few lvh.me domains to see what happens on the first visit, and subsequent visits

ATTACK

Certificate Authority Compromise

- It's happened before, and will happen again
- Browser and OS vendors scramble to revoke the authority
- Users who trust this CA, may trust any bogus cert it generates, unless...



HTTP Public Key Pinning (HPKP)

- HTTP Response header informs browsers of what a public key should look like

```
Public-Key-Pins: pin-sha256="<pin-value>";
                 max-age=31536000;
                 includeSubDomains;
                 report-uri="<uri>"
```

- For the specified amount of time, browsers will continue to assert that the certificate for your domain matches the `pin-sha256` value
- The pin value is known as a Public Key Fingerprint.

HTTP Public Key Pinning (HPKP)

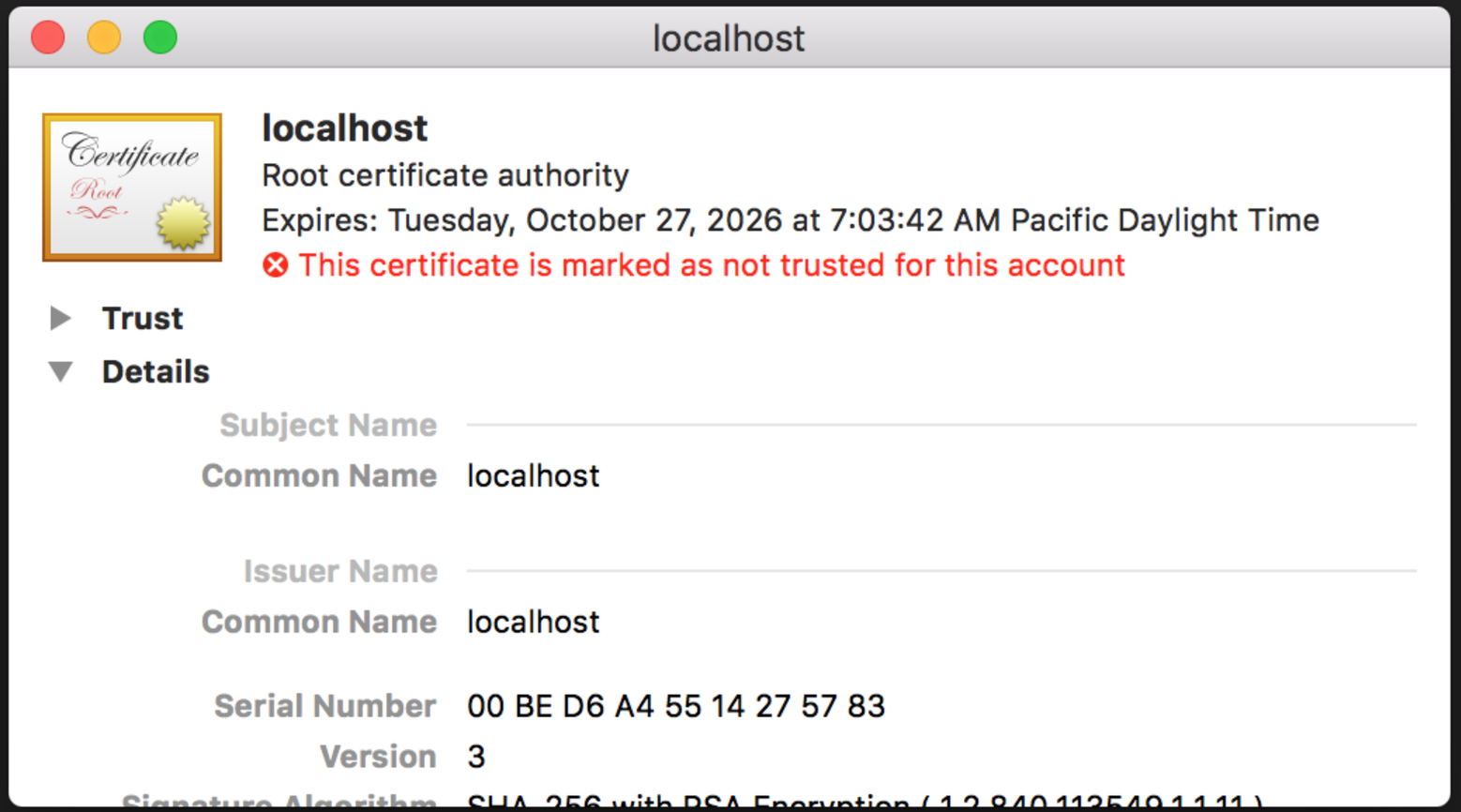
- Now, it's not a matter of A trusted certificate that matches the appropriate domain, but THE certificate you have
- 2016 survey: Less than 375 HPKP domains (+ 76 in "report only" mode) for the top 1M Alexa sites
- Enabling this is a good idea. Also go to great lengths to protect your private key.

WARNING

Treat Certificates With Care!

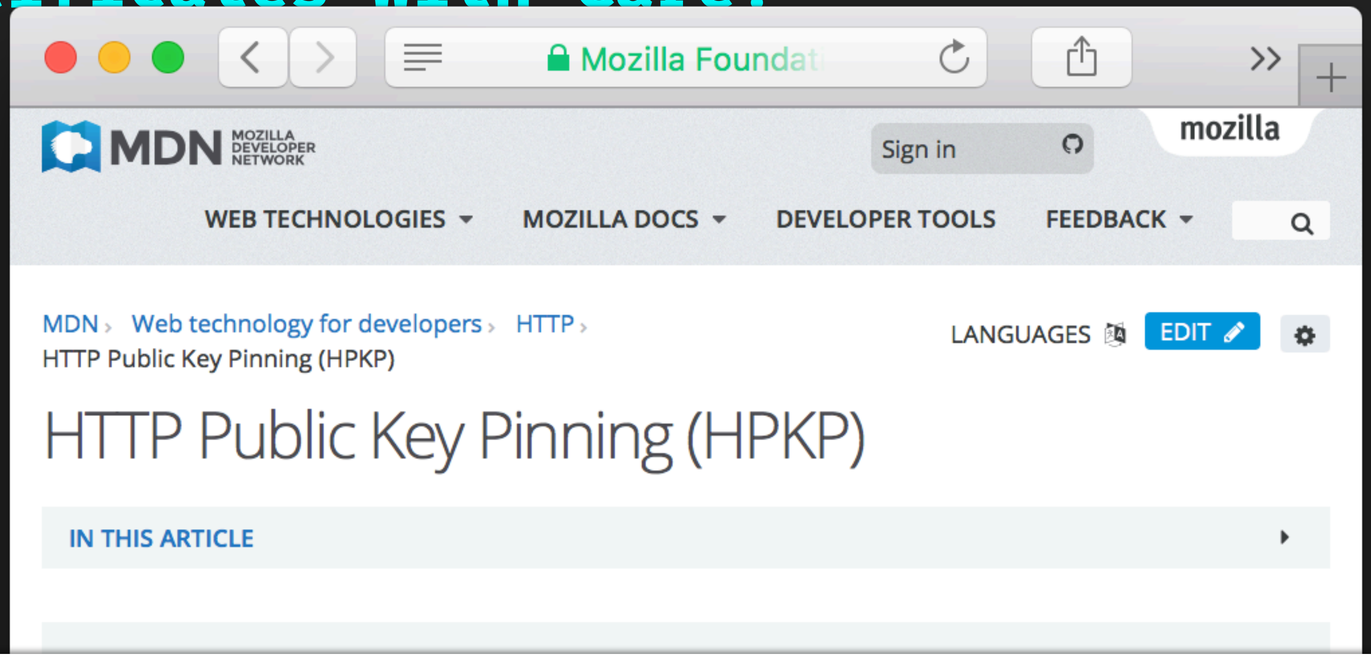
WARNING

Treat Certificates With Care!



WARNING

Treat Certificates With Care!



> Firefox and Chrome **disable pin validation** for pinned hosts whose validated certificate chain terminates at a **user-defined trust anchor** (rather than a built-in trust anchor). This means that for users who imported custom root certificates all pinning violations are ignored.

Attack & Defend