The other side of webapp security

Luka Kladaric
Sekura Collective
luka@sekura.io
@kll

What this talk is about

Securing users, their browsers and computers.

What this talk *isn't* about

Securing your servers or the data on them.

Quick poll

- Who here runs a website?
- Does it support HTTPS?
- Does it enforce HTTPS?
- Does it have HSTS headers set?

HTTPS / SSL / TLS

These mean subtly different things.

In the context of web development
A method of encrypting traffic to and from a web server using a server-provided certificate.

HTTPS (2)

If the certificate is signed by a **trusted party** it also gives us a degree of **confidence** that the server we're communicating with is actually **authorized** to speak on behalf of the website we're trying to visit.

Certificates can also be **self-signed**, or otherwise signed by an untrusted party. These are **mostly worthless**.

HTTPS (3)

Once you've established the other side possesses a **trusted** certificate, the chance you're being **MitM**[1]-ed or **spied on** drops by orders of magnitude.

[1] MitM - Man in the Middle attack

Forcing HTTPS and pitfalls

Forcing HTTPS means **redirecting any plaintext traffic** to HTTPS.

It’s good in that it redirects people to the secure site, and **you should definitely do that**.

But it’s bad in many ways.

Forcing HTTPS - gotchas (1)

Obscures/hides the fact that certain things still point to your plaintext address.

Like... **login form** or **oAuth callback** pointed at http

Users are successfully logged in, but the credentials are **transmitted unencrypted first**, before getting redirected to https.

Forcing HTTPS - gotchas (2)

Takes people to the right place which means they will never update their bookmarks.

Forcing HTTPS - gotchas (3)

Doesn’t usually work for non-human visitors - mobile apps and other clients, which generally construct each request from scratch rather than pick a link from the page they were served.

So, if it works at all it just duplicates traffic where every single request is initiated over HTTP first and then bounced to HTTPS.

Forcing HTTPS - gotchas (4)

Breaks POST requests which cannot be redirected.

Forcing HTTPS - gotchas (5)

The redirect itself is not secure.

An MitM attack could happen at this point, with the user continuing to browse the http site while the attacker proxies requests back and forth for the user over https with neither side being aware that anything is off.

Enter HSTS

The magic bullet?

HSTS?

HTTP Strict Transport Security

A response header that says

"*don't even attempt to contact me*
over plaintext HTTP for X seconds."

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

Include all subdomains

The header can optionally apply to all subdomains, too.

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

HSTS preload

HSTS + long timeout + includeSubDomains + preload = **win!**

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

HSTS gotchas (1)

If you can’t force HTTPS, you can’t enable HSTS.

HSTS gotchas (2)

If you have subdomains which don’t support HTTPS, or still need to receive plaintext traffic, you can’t enable “includeSubdomains”.

HSTS gotchas (3)

If you can’t include subdomains, you can’t enable preload.

Which means you still have that first request for users typing in your domain name or following an old link that is vulnerable.

HSTS is a must

Start rolling it out today.

It will hurt, but it will pay off.

Secure cookies

Cookies are used for various tracking and storage purposes, but the most critical use is to keep logged in state.

To leak these is to let other people impersonate a legitimate user in their interaction with your website / application.

Secure cookies

Absolutely make sure that all the cookies you are sending to your users have the “**Secure**” flag set on them, so that even if a plaintext HTTP request ever happens, they are not transmitted.

XSS

XSS is bad.

Don’t let XSS happen to you.

The end. j/k

CSP

Content Security Policy is an HTTP response header that reduces the risk of XSS in modern browsers by declaring which inline or external resources are allowed.

CSP

It also lets you **block all inline scripts** except the ones explicitly whitelisted by several available methods.

The **combination of these two things** makes XSS unusable on your website, despite any backend code flaws.

CSP - nonces

A nonce can be used to whitelist inline scripts which contain no dynamic content. A single nonce can be used for all static inline scripts.

```
Content-Security-Policy: default-src 'self'; script-src 'nonce-4AEemGb0xJptoIGFP3Nd'
<script type="text/javascript" nonce="4AEemGb0xJptoIGFP3Nd">
```

USE A DIFFERENT NONCE FOR EACH REQUEST!

CSP - hashes

For scripts that are static but trusted, we can use hashes.

```
Content-Security-Policy: script-src 'sha256-
cLuU6nVzrYJlo7rUa6TMmz3nylPFrPQrEUpOHllb5ic='
```

CSP - final

```
Content-Security-Policy:
    default-src 'none';
    script-src 'self' 'nonce-0gscAiOe9cvhDVnsp2jy'
            www.google-analytics.com
            cdnjs.cloudflare.com;
    object-src 'none';
    style-src 'self' 'unsafe-inline' cdnjs.cloudflare.com;
    img-src 'self' www.google-analytics.com
        stats.g.doubleclick.net;
    media-src 'none';
    font-src 'self' fonts.gstatic.com;
    connect-src 'self';
    base-uri 'self';
    child-src www.google.com;
    form-action 'self' accounts.google.com www.paypal.com;
    frame-ancestors 'none';
    upgrade-insecure-requests;
    report-uri https://yoursite.report-uri.com/r/d/csp/enforce
```

CSP Report-Only

Use **Content-Security-Policy-Report-Only** to log issues (to report-uri and console) without blocking any content while rolling out.

Questions?

Luka Kladaric
Sekura Collective
luka@sekura.io
@kll

Thank you!

Luka Kladaric
Sekura Collective
luka@sekura.io
@kll