Secure Software Design

Andey Robins

Spring 23 - Week 13

Web Security

Outline

- Web basics
- Web security
- Web vulnerabilities
- DevOps

Prepare yourself for acronym hell.

Web Basics

- HTML/CSS/JS/JSON
- Client/server model
- Cookies, Certs, and Authorities
- DOM
- Frameworks
- HTTP/REST/RPC

Foundational Technologies

- HTML
- CSS
- Javascript (and TypeScript)
- JSON (or XML and others)

HTML

Hyper Text Markup Language: an encoding schema for expressing the content on a web page as hyper text.

```
<html>
  <body>
    <p>I must not fear, fear is the mind killer...</p>
  </body>
</html>
```

CSS

Cascading Style Sheets: a means to express the styling of individual objects or classes of objects in the DOM.

```
app {
  background-color: #000;
  color: #fff;
}

h1 {
    font-size: 2em;
    font-family: 'courier';
}
```

JavaScript

Javascript: is a lightweight, interpreted, JIT compiled, functional programming language. It is used on the server side of 98% of websites today.

```
let fooElement = document.getElementById('foo');

fooElement.textContent = 'New text content';
```

Aside: The History of JS

Back in the Netscape days of the internet, Brendan Eich was brought in to embed Scheme into the next Netscape browser. Instead, they believed it was the correct move to develop a new language just for the web. Brendan then took 10 days, and the early versions of JS are the result.

TypeScript

TypeScript: A superset of JS which adds type-checking, interfaces, and classes to vanilla JS. Developed and released by Microsoft and Anders Hejlsberg, it's used very often in frontend JS applications making use of libraries.

```
let fooElement: Element = document.getElementById('foo');

let fooElement.textContent: string = 'New text content';
```

JSON

Javscript Object Notation: A common means of representing and exchanging information across the internet. It fits very nicely into javascript due to adopting the JS style of modeling data.

```
{
    name: "Andey Robins",
    age: 22,
    job: {
        title: "Graduate Research Assistant",
    }
}
```

Client Server Models

The **Client/Server Model** of design designates two locations for logic and a many-to-one* relationship between them. Computation can be performed at either point in the system, but a strict trust boundary exists between the two.

Static Pages

The simplest setup is a static site (i.e. every user receives the same content). Here the server is said to “serve” the content to the client. A complete HTML document (or whatever) is handed off through HTTP.

Dynamic Pages

Dynamic pages are resources that look different for each visitor. There are two main approaches to delivering dynamic content, placing the burden of rendering on either the client or server.

CSR

"Client-side rendering(CSR) involves rendering pages directly in the browser using JavaScript. All logic, data retrieval, templating, and routing are handled on the client, not on the server." - Scythe Studio

CSR Page

```
<html>
  <head>
    <title> CSR App </title>
  </head>

  <body>
    <div id="app"> </div>
    <script src="../src/index.js"> </script>
  </body>
</html>
```

SSR

"Server-side rendering (SSR) is an application's ability to convert HTML files on the server into a fully rendered HTML page for the client. The web browser submits a request for information from the server, which instantly responds by sending a fully rendered page to the client. Whenever the client navigates to a different page on the website, the server will do the work once more." - Scythe Studio

SSR Page

```
<!DOCTYPE html>
<html>
 <head>
   <title> Web Page Rendered on Server Side </title>
 </head>
<body>
  <h1> This is a Heading   </h1>
  <div>
    <p> This is a form </p>
      <form>
        <label for="fname">First name:</label><br>
        <input type="text" id="fname" name="fname"><br>
        <label for="lname">Last name:</label><br>
        <input type="text" id="lname" name="lname">
      </form>
  <div>
</body>
</html>
```

CSR vs SSR

Server Load -> **CSR**

SEO -> **SSR**

Interactivity -> **CSR**

Intial Load Time -> **SSR**

SPA/SaaS Style App -> **CSR**

Content Heavy -> **SSR**

CSR and SSR Security

- Who is making the requests?
- Where is the data located?
- What is an expected UX?

The DOM

The **Document Object Model (DOM)** is *the* datastructure for the web. It represents a page as a collection of objects, which can be created, modified, or deleted programatically by the documents which make up the page and the JS runtime.

Top Frameworks

- Next.js/Sveltekit
- Nuxt.js/Svelte
- Express
- React/Angular/Vue

Batteries Included

- These have everything you need in one system
- Often opinionated
- Minimal transferability

Ex: Next.js, Sveltekit, Gatsby, etc.

Figure 1: Nuxt, Next, SK, Marko, Qwik, Gatsby, VuePress, Astro, Elder

Batteries Not Included

- Just components
- Do one thing and do it well
- Can often be embedded in other applications

Ex: React, Svelte, Vue, Angular, etc.

Aside: Runtimes

As for backend runtimes, there are several in the JS world right now. They are ranked in order of importance/prevalence.

1. Node
2. Deno
3. Bun

Communication Protocols

These are the potential ways to send and request information over the web.

HTTP

- Basic
- Verb driven
- Infinitely configurable
- No model baked in

REST

Representational State Transfer: is an approach to API design which removes all need for maintaining state from the web-server.

A primary benefit is that it is client agnostic, allowing for arbitrary client stacks (i.e. Web and Mobile and Desktop through the same API)

HTTP vs REST

HTTP may or may not have some concept of state. If it does, it isn't RESTful. If it doesn't, it's a REST API.

CRUD

- **C**reate, **R**ead, **U**pdate, **D**elete
- Maps to HTTP Verbs (POST, GET, PUT, DELETE)
- Good for data-first applications
- Built on top of HTTP

RPC

Remote Procedure Call is a computational methodology where almost all of the processing is done on the server. Modern versions use Protobuffs and HTTP 2 as well to make them incredibly quick with throughput.

Web Security

Security Objects

While we'll talk more about security in the web, it's important to provide some discussion surrounding the common, security primitives that we use in the web.

- Cookies
- Certificates
- CAs

Cookies

Cookies are small pieces of data which a server asks the client to store and provide on subsequent requests made to the server.

An Analogy

I'm a baker, and I want to keep track of how many times someone visits my bakery. If it's their first time I give them a slip of paper with the number 1 on it. Every time they come back in, I give them a new slip with their number plus 1.

It's then easy to see how often a customer visits by checking their slip of paper (their cookie).

Session Cookies

Session Cookies are a special type of cookie which establish a secure session between a client and the server. It allows an individual client to be identified by the server and is usually backed by some cryptographic assurance of identity.

Good Cookies

- Preferences
- Configurations/Settings
- “Front End Stuff”
- Identification
- Session Management

Bad Cookies

- Trackers*
- Price
- Security Information

In other words, treat a cookie as a form of user input which is subject to all of the validation any user input is subject to.

Cookie Interaction

A `<script>` has access to cookies by default unless they set the `httponly` attribute. Therefore, we can assume that cookies by default can be accessed by a malicious script, so why do we not just have `httponly` set by default?

Cookie Interaction

A `<script>` has access to cookies by default unless they set the `httponly` attribute. Therefore, we can assume that cookies by default can be accessed by a malicious script, so why do we not just have `httponly` set by default?

Backwards Compatibility

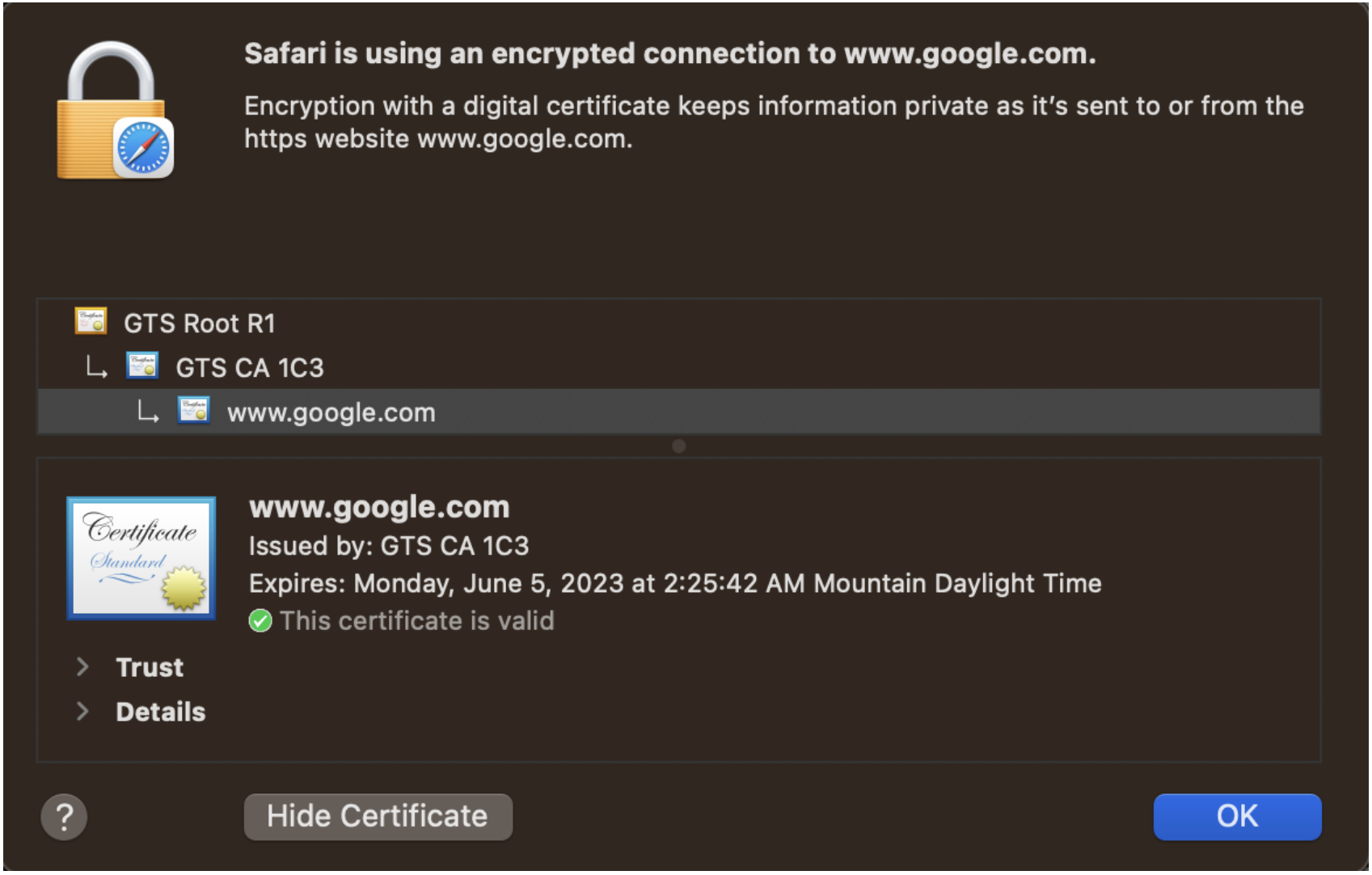
Certificate Authorities

Certificate Authorities are groups which are largely granted trust and who we trust to sign the certificates and authorize/authenticate other principles in the net.

Ex: Digicert, Let's Encrypt, Google Trust Services, etc.

Certificates

Certificates are the digital artifacts produced by Certificate Authorities. It's basically just like any form of ID, but backed by a cryptographic identity.



HTTP vs HTTPS

HTTPS is just HTTP over TLS/SSL. It addresses threats by:

1. Encrypting traffic in flight
2. Prevent modification in the middle
3. Verify communicator identities

Adoption

- Developed in 1994
- Facebook used HTTP up until 2013
- General hardware good enough as of 2015
- 40% of the web in HTTPS in 2016
- 85% as of 2020

Why the Delay?

1. Hardware speeds
2. Crypto bugs
3. Lack of CAs (price wise)
4. No consumer demand

Same Origin Policy

You don't want a single website to be able to open up and modifying your accounts on other pages [citation needed].

The "Same Origin Policy" is the idea that other pages (or origins) should not be able to *reach in* to a page's content.

CORS

Cross-Origin Resource Sharing is the mechanism to circumvent the Same Origin Policy. Established through headers and enforced by the browser, this places restrictions on which domains are allowed to send information to other domains. This is a common problem to run into while developing web apps locally, and slightly beyond the scope of what we have time to talk about.

Visible Information: Cookies

Can the webpage served by the host see the cookies set for other hosts?

Host	ex.com	dog.ex.com	cat.ex.com	ex.org
ex.com	Yes	No	No	No
dog.ex.com	Yes	Yes	No	No
cat.ex.com	Yes	No	Yes	No
ex.org	No	No	No	Yes

Visible Information: HTTP

Can an attacker...	HTTP	HTTPS
see web traffic between endpoints?	Yes	Yes
see both IP addresses?	Yes	Yes
deduce the web server's identity?	Yes	Sometimes
see what page within the site is requested?	Yes	No
see the page content and the body of POSTs?	Yes	No
see the headers and URL?	Yes	No
tamper with the URL, headers, or content?	Yes	No

Web Vulnerabilities

XSS

Cross Site Scripting Attacks (XSS) are a class of injection attacks which alters the behavior of a website to run an unauthorized script.

Example: XSS

We have a website with different colors which are encoding in the URL.

`https://www.example.com/page?color=green`

Which then has the color inserted into the page:

```
<h1 style="color:green">This is colorful text</h1>
```

Example: XSS

The server side code would look something like this:

```
query_params = urllib.parse.parse_qs(self.parts.query)
color = query_params.get('color', ['black'])[0]
h = '<h1 style="color:%s">This is colorful text.</h1>' % co
```

Example: XSS

The attack then looks like:

```
https://www.example.com/page?color=orange"><script>alert(
  "Gotcha!")</script><span%20id="dummy
```

Example: XSS

Which makes the page render as:

```
<h1 style="color:orange">
  <script>alert("Gotcha!")</script>
  <span id="dummy">This is colorful text.
</h1>
```

XSS

What might an attack look like:

1. An attacker finds a website which allows for XSS injection
2. The attacker crafts a payload which exfiltrates session tokens
3. The attacker embeds this payload in something sent to general users
4. The users open up the page, triggering the payload
5. The attacker gets the user's cookies

Forms of XSS

Reflected - What we just walked through

Stored - Getting the payload to live in storage somewhere

DOM-Based XSS - Getting the payload to somehow lie in the underlying DOM rendering system

Mitigation

1. Use a good library
2. Validate user inputs
3. Don't manually construct strings of HTML

XSRF

Cross Site Request Forgery is an attack on the limitations of the Same Orogin Policy.

We will walk through the example fromm the textbook to see this in action.

Invariants

1. Any website can GET resources from any other
2. Any website can POST to any other
   2.1 This request changes the state of the other system
3. Website should be able to pull in content from other pages
4. SOP means a website pulling in content can't see that content

XSRF Scenario

Social website Y has many users. The site is running a poll and every user gets one vote. The website creates a one-time use cookie which is dropped by the voting page.

XSRF Potential

A comment on the voting page of site Y has a link and says “Check this out before you vote!”

This page is hosted on website X

XSRF Attack

In the background, site X submits a false vote using the browser cookie jar to present the valid one-time cookie from site Y. The votes are thus manipulated in favor of site X.

Common Mitigation

Above all else, **use a strong framework**. They will prevent all of these.

- Don't let attackers inject untrusted input
- Specify MIME types properly to leverage the browser's type security
- Don't allow arbitrary redirects
- Only embed trusted websites in an `<iframe>`
- Beware of XML external entity attacks
- Remember the CSS `:visited` selector can leak user history

OWASP Top 10

1. Broken Access Control
2. Crypto Failures
3. Injection
4. Insecure Design
5. Security Misconfiguration
6. Vulnerable and Outdated Components
7. Identification and Authentication Failures
8. Software and Data Integrity Failures
9. Security Logging and Monitoring Failures
10. Server-Side-Request Forgery

OWASP Top 10

Turns out, we've covered and either mitigated or talked about all of these already!

1. Broken Access Control (Midterm program)
2. Crypto Failures (Week 5)
3. Injection (Input sanitization)
4. Insecure Design (This whole course)
5. Security Misconfiguration (Week 4)
6. Vulnerable and Outdated Components (Week 8)
7. Identification and Authentication Failures (DD)
8. Software and Data Integerity Failures (DD)
9. Security Logging and Monitoring Failures (DD)
10. Server-Side-Request Forgery (Earlier Today)

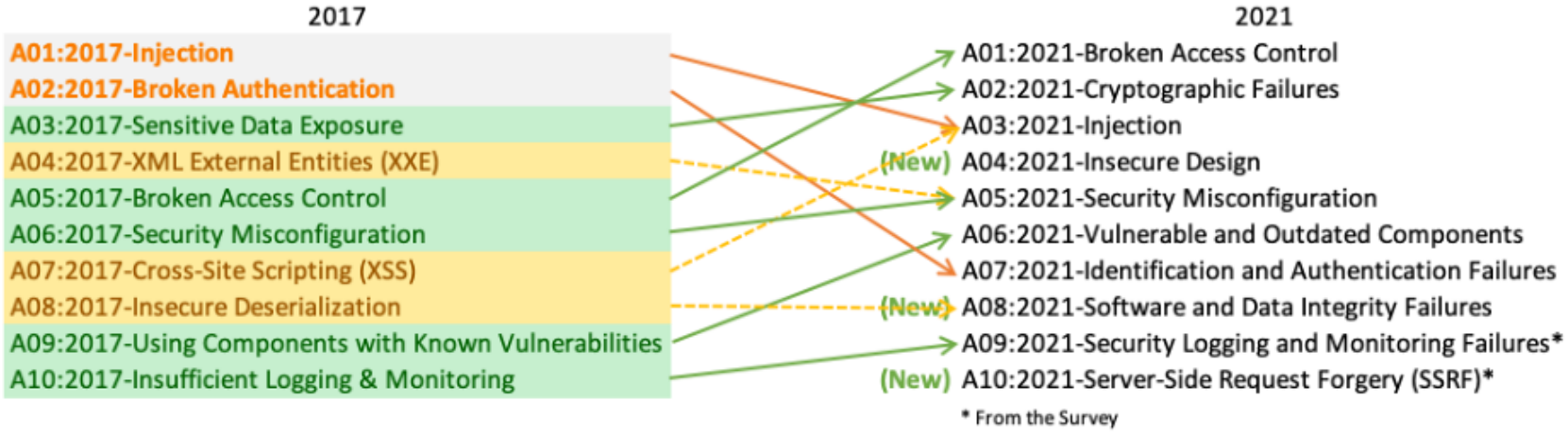


Figure 3: The mapping of 2017 OWASP to 2021 OWASP

DevOps and the Web

Test, Stage, Prod

Put your work in three places of increasing importance.

- **Test** can have whatever the current build is, bugs and all. It's okay if it's broken.
- **Stage** is the internal version of the next release. This should be relatively stable but where you go to test things work. Don't use real data.
- **Prod** is where you want to be sure nothing can be broken. This is where you play for keeps.

AWS and CSPs

All CSP companies provide some version of security tools. Learn how to use them and set them up properly. If everything is done right, you offload 95% of your security to your CSP.

OAuth and JWTs

OAuth is the defacto standard for authentication. It's the protocol behind all the "login with google/facebook/amazon/etc." buttons.

JWTs are the crypto backed identity for most of the web.

Questions

Next Time

- Security Testing
- Project Time!!