WEB SECURITY

- Prevent fake logins
- Protect user data
- protect access to the system
- protect information on the system

XSS

Cross-Site scripting (XSS)

(Why "X"? English is weird)

Running a client-side script that is NOT yours

SIMPLE XSS DEMO

Consider

```
const express = require('express');
const app = express();
const PORT = 3000;

app.get('/', (req, res) => {
  const name = req.query.username;
  res.send(`<p>Hello ${name}</p>`);
});

app.listen(PORT, () => console.log(`http://localhost:${PORT}`));
```

- What if we send HTML as `username`?
- What if that HTML is a script tag?

Modern browsers try to block, so it's hard to demonstrate a SIMPLE example

- But XSS isn't hard

WHY IS XSS BAD?

- They can inject ads (incl. popups)
- They can scrape any data off the page and send it elsewhere
  - Including passwords
- They can alter any data on the page

HOW TO DEFEND AGAINST XSS

Never use data from the user without filtering it

- Ideally, "whitelist" data that is allowed and block anything else
- Whitelisting isn't always practical, but should always be the first choice

Attempts to "Blacklist" bad data eventually fail

Examples:

- (whitelist) Phone number is only 0-9, parens, dot, and hyphen.
- (blacklist) Phone number can't contain star/asterix or angle brackets

XSRF/CSRF

Cross-Site Request Forgery

(Why sometimes a "C" and sometimes an "X"? English is weird)

Similar to XSS, but it SENDS requests (service requests/page requests) AS THE USER (using any credentials that are in the browser).

The site doesn't know it isn't the user. Anything the user is allowed to do it will do.

WHY IS CSRF BAD?

- They can send server calls as the user (updating/modifying/deleting data)

Examples:

- Send commands to your bank
- Send messages to your friends on Facebook
- Buy things on Amazon, ship them to other places
- Vote things up/down on social media

DEFENSE AGAINST CSRF

Similar to XSS.

SQL INJECTION

Just like XSS is inserting javascript into your HTML, SQL Injection is inserting SQL commands into your SQL.

Consider:

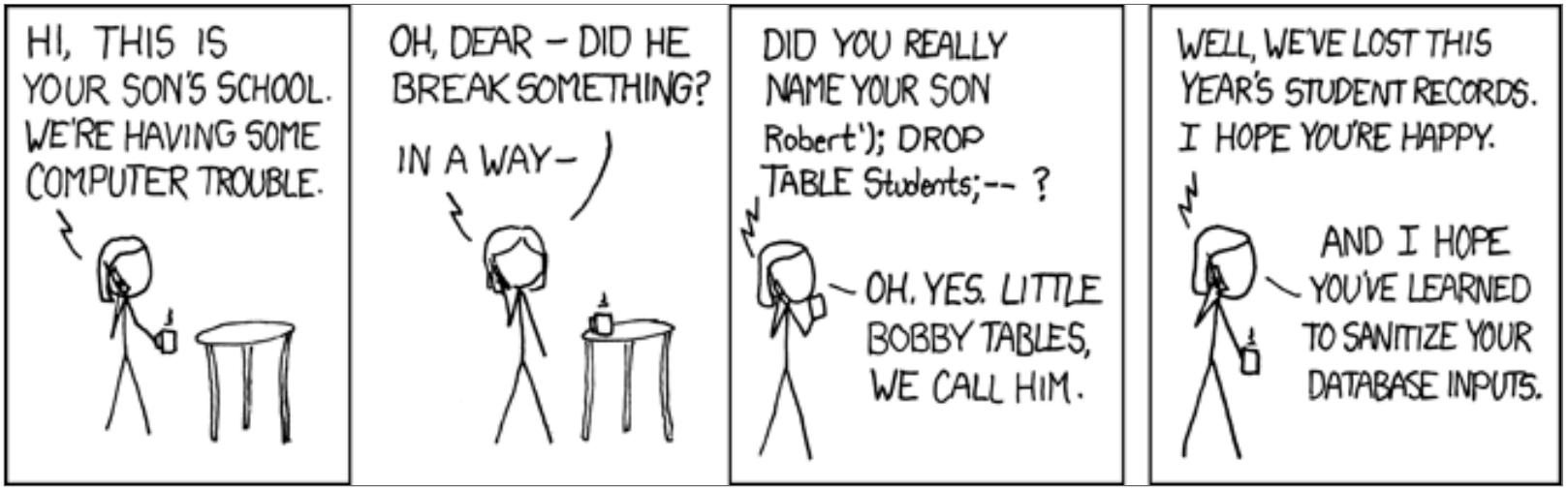
- `SELECT age FROM people WHERE name = "${name}"`

What happens is all based on what `name` is:

- Send `Bao` and it works fine.
- Send `Bao"; DELETE * FROM people WHERE "1" = "1` and you just deleted everything

LITTLE BOBBY TABLES

XKCD (**http://xkcd.com**)



WHY IS SQL INJECTION BAD?

- They inject ads
- They inject scripts for XSS/XSRF
- They can delete your data
- They can copy your data
- They can encrypt your data (ransomware)
- They can alter your data (incl. theft)

DEFENSE AGAINST SQL INJECTION

- Never craft your SQL from user input
- Always use "bound" variables when possible
- If not possible to use bound, use the escaping libraries from your vendor
  - AND whitelist your data

POOR PASSWORD SECURITY

If someone can read your DB...

- Malicious employee
- Security Hole

...it is much better if they can't actually get passwords

HASHING

You never need to store the password directly.

You don't care about the password itself, all you need is proof the user KNOWS it.

When given a password, run it through a one-way hashing algorithm, store resulting hash.

To later confirm, when they give you the password you can again hash it and compare to the stored hash.

You never store the password.

SALTING

But hashing algorithms can be used to pre-calculate options for a given result (a "rainbow table")

- And a lot of passwords are painfully common
- So add a "salt" to it. Store the salt with the hash.

Example:

- user "bob" has password "123456"
- Hashed that might end up as "ip9awlnfhiorwijeqds"
- But let's pick a random salt of "ih7g57r"
- So we hash "ih7g57r-123456"
- That gives us "hhncdhluxhluxhlu3xl2"
- So we store "ih7g57r-hhncdhluxhluxhlu3xl2"
- Next time "bob" logs in, he gives us "123456"
- We hash the salt(from stored value)+password
- We compare the salt+hash to the stored value

DON'T DO LOGINS

My advice: Don't try to do logins

- Do you know cryptographically secure hashing algorithms?
- Do you know how get a large enough salt?
- Do you know how these will remain secure as technology advances?

Your stuff may not be a big deal, the user's password IS

- They reuse it somewhere else more important

Use an alternative

- External Password provider (Google, Facebook, Github, etc)
- OIDC (Okta, Autho, etc)

BROWSER DEFENSES

- Same Origin Policy (SOP)
- Cross Origin Resource Sharing (CORS)

SAME ORIGIN POLICY

In the beginning, a browser could load ANYTHING from ANYWHERE

- Powerful
- Terrible security

The Same-Origin Policy (SOP) says that a browser can only load:

- JS
- CSS
- images

from domains that are not the same as the current page. Anything else is denied.

This reduced the amount of bad things.

CORS

But SOP didn't allow for us to make service calls to other domains!

Including different subdomains (e.g. **www.google.com** and api.google.com)

For that we need CORS (Cross-Origin Resource Sharing)

CORS says that the *site* has to say you're allowed to talk to it.

This is entirely enforced on the browser, not by HTTP.

A non-browser can talk to a site **without** caring about CORS.

CORS ERRORS

If you see a browser console error message that suggests using **'no-cors'**...

It is useless and confusing. `no-cors` is almost NEVER useful (you can't get any data back from a `no-cors` request.

CAN I BYPASS CORS?

Don't try. It'd be pretty lousy security if you could chose to not follow it, so you'll just waste time.

Make sure the server is sending the CORS headers.

If you can't change the server, you could use a proxy server that you CAN talk to. It can talk to the problematic server. With no browser in between them, there is no CORS issue.

But most places either have a way to set the CORS headers, or a good reason not to accept CORS.

SUMMARY

- Never trust input
  - don't store it
  - don't display it
  - don't use it in string commands
- Service calls are all visible to the user
  - and points in-between
- You will not be smarter than the bad people
  - You win by not giving them the chance to try