Web Security

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

Core Rules of Web Security

- Never trust user input
- You can never be more clever than all of them
- The Front End cannot add security
    - ...only convenience
- Your data IS of interest

XSS

Cross-Site scripting (XSS)

- Why "X"?
    - English is weird
    - CSS already has a meaning

Running client-side javascript that is NOT yours

Simple XSS Demo

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

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

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

Seems reasonable enough, right?

How to abuse with XSS

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

```
?name=%3Cimg+src=%27%27+onerror=%22alert(%27pwned%27)%22%3E
```

A user (not you!) can now run JS on your page

If we save that data and show it to others (like name), the attacker can run JS on the pages of OTHER USERS

Why is XSS Bad?

They can…

- inject ads (incl. popups)
- redirect page
- steal processor time
    - Bitcoin mining?
- scrape data off the page and send it elsewhere
    - Including private data/passwords
- alter any data on the page
- perform actions on the page
    - Enter data
    - Click buttons

How to defend against XSS

Rule #1: Never trust data from the user

- "allow" permitted data, block anything else
- Allowlisting isn't always practical, but should always be the first choice

Rule #2: Never assume the user isn't clever enough

- Attempts to "Denylist" bad data eventually fail

Examples:

- (allowlist) Phone is only 0-9, parens, dot, and '-'
- (denylist) Phone can't contain `*` or `<` or `>`

Never trust the front end

NEVER TRUST FRONT END JS TO ENFORCE SECURITY

- They can alter it, or even just not use a browser

Rule #3:

- **Security MUST be backend**
- Client-side JS provides convenience, not security

Rule #4:

- Your data IS of interest
    - Inject malware
    - Grab reused account info

SQL Injection

- XSS is inserting javascript into your HTML
- SQL Injection is sending SQL commands

Consider:

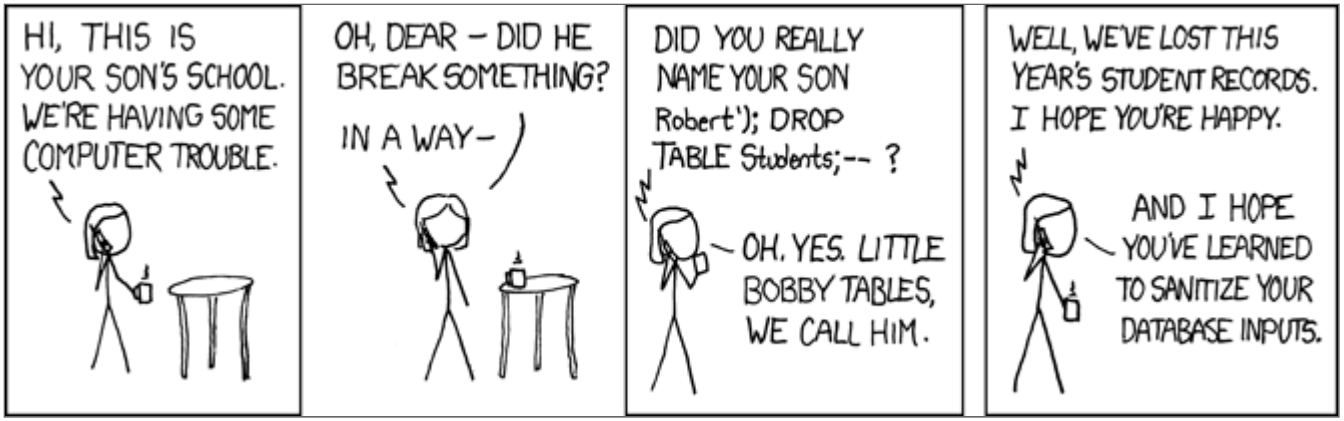
- `SELECT age FROM people WHERE name = "${name}"`

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/327/**)



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 allowlist 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

You never store the password

Hashing

When account is created:

- Use one-way hashing algorithm on password
- Store resulting hash

To later confirm a password:

- Hash the password they give you
- Compare to the stored hash

You never store the actual password

Rainbow tables

Hashing protects the actual password

- Attacker might get stored hash
- But won't know how to *generate* the given hash

Hashing is based on how difficult it is to reverse a hash

- But if they precalculate a big list
- They can find matches easily

a "rainbow table"

Salting

Make the pre-calculations more expensive

- A "salt" is a random value
- Store the salt with the hash
- Use the salt and password to generate the hash
- Users with same password have different hashes
- Checking a password can see stored salt to check

Salting - Create Account

- user "bob" has password "123456"
- We pick a random salt of "ih7g57r"
- We hash "ih7g57r-123456"
    - salt AND password
- Hash result is "hhncdhluxhluxhlu3xl2"
    - Different than hash result of just password
- We store "ih7g57r-hhncdhluxhluxhlu3xl2"
    - Both salt AND hash

Salting - Login

- Next time "bob" logs in, they give us "123456"
    - We see bob's salt and hash in our records
- We hash "ih7g57r-123456"
    - Salt from bob's stored record
    - Password given by user
- We compare the result to the stored hash
    - Hash from bob's stored record

Don't Do Logins

My advice: Don't try to do logins

- cryptographically secure hashing algorithms?
- how get a large enough salt?
- 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 external provider and OIDC

- Google, Facebook, Github, etc
- Okta, Auth0, etc

Secrecy is not Security

Do not assume a secret url is secure

- So much tracking users' browsing
- Brute-force attacks on urls
- Can't re-secret a secret

Storing Keys and Passwords

Often your program will need to access secured data

- no human involved in sending the passwords/keys/tokens

How do you keep this secure?

Frontend or Backend

Some tokens aren't "private"

- They are more like usernames for systems
    - not passwords for systems
- These aren't interesting to users
    - it is okay if users can "find" them
- Only these keys can be included in frontend code

Anything in frontend code can be seen by that user

Don't put backend keys in backend code

A key/password that only lives in backend code

- is secure from the user seeing it
- but NOT secure if you check it in your repo
    - repos are often accidentally made public
    - many bots scan github (et al) for such values

Using the Environment

One common solution:

- put password in operating system environment
    - program reads it from environment
- Ex: `KEY=supersekretpassword node server.js`
- password available to running code
    - but not in the written code

Further progress

Problem: all devs have to know/remember password

Solution: a file holds passwords, that file isn't in repo

- commonly a `.env` file, or sometimes many such files(`dev.env`, `prod.env`)
- code can read environment OR this file
    - many libraries make this easy
- the `.env` file is NOT in repo
- devs can exchange this file outside of the repo

Tip: Add `.env` to your .gitignore to prevent accidents

Common .env discussions and node tools

Problem is real:

- **https://www.zdnet.com/article/over-100000-github-repos-have-leaked-api-or-cryptographic-keys/**
- **https://dev.to/somedood/please-dont-commit-env-309h**

Node library to make .env easy:

- **https://github.com/motdotla/dotenv**
    - **https://github.com/dotenv-org/cli**

Summary

- Never trust input
    - don't store it
    - don't display it
    - don't use it in string commands
- Web requests/responses 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
- No site is too small to be a target
- use .env files outside repo for passwords/keys

Security Rules

- Rule #1: Never trust data from the user
- Rule #2: Always assume a user is clever enough
- Rule #3: Security MUST be server-side
- Rule #4: Your site WILL be a target