# 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

# OWASP Top Ten

Tracking Web Security Issues for *decades*

**https://owasp.org/www-project-top-ten/**

History of OWASP Top 10:

- **https://medium.com/@dramkumar/history-of-all-owasp-top-10-over-the-years-9470c0adf43d**

# 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 `>`

# Allowlisting helps prevent attacks

**Allowlisting** sometimes called "whitelisting"

- You say characters are allowed
- NOT what characters are denied
- Good: Username is A-Z, a-z, 0-9, _ only
- Bad: Username cannot contain "<" or ">"

Opposite is **Denylisting** (blacklisting)

**Alway use Allowlist instead of Denylist when you can**

- Data like comments are difficult to allowlist

# 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:

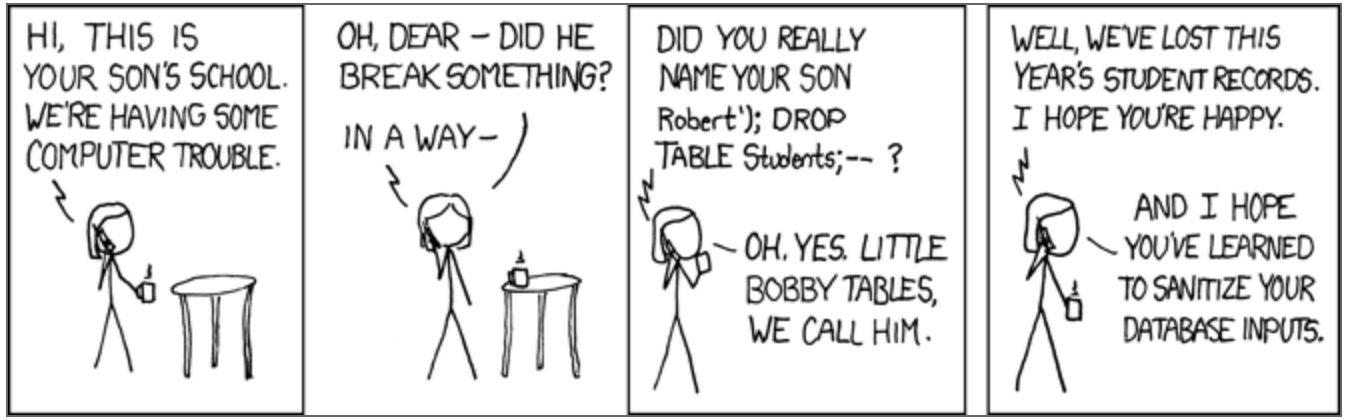
- `SELECT age FROM people WHERE name = "${name}"`

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 raw SQL from user input
- Always use "bound" variables when possible
    - SQL strings don't include values
    - Values passed in addition
    - Otherwise use vendor-provided escaping
- AND allowlist your data

Remember what **allowlisting** means:

- Say what characters allowed
- NOT saying what characters are disallowed

# 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 actual 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 can't reverse hash to know password

Hashing security based difficulty of reversing a hash

- But if they generate (not reverse) a big list
- They can find matches easily

Called a **rainbow table**

# Salting

Make Rainbow Tables more expensive

- A "**salt**" is a random value
- Generate salt when password created
- Use the salt and password to generate the hash
- Store the salt with the hash
- On Login attempt, lookup salt and hash for user
- Use salt and given password to generate a hash
- Compare this hash to stored hash
- Users with same password have different hashes
- Rainbow Table useless unless made w/same salt

# 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

Instead: Use an external provider and OIDC/Oauth

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

# Secrecy in use is not Security

Do not assume a secret url is secure

- Many sources track users' browsing
- Brute-force attacks on urls
- Can't re-secret a revealed 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
- NOT secure if you check it in your repo
    - Repos are often accidentally made public
    - Many bots scan github (et al) for such values
- Shared among many devs + machines
    - Each a vulnerability with poor tracking

# 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
    - Sometimes many 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: (Also experimentally built in to node)

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

# Summary

- Never trust input
    - Hesitate to 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
    - Advanced: "Vault" to hold credentials

# 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