Web Application Security

CS 3320 Lecture 13.2

Overview

- Security Rules
- Common Attack Vectors
- The Single-Origin Policy
- Designing for Security
- Securing Yourself

Security Rules

- Rule 1
  - The client is full of lies
- Rule 2
  - The adversary will take any advantage
- Rule 3
  - If you give users a way to communicate, they will spam

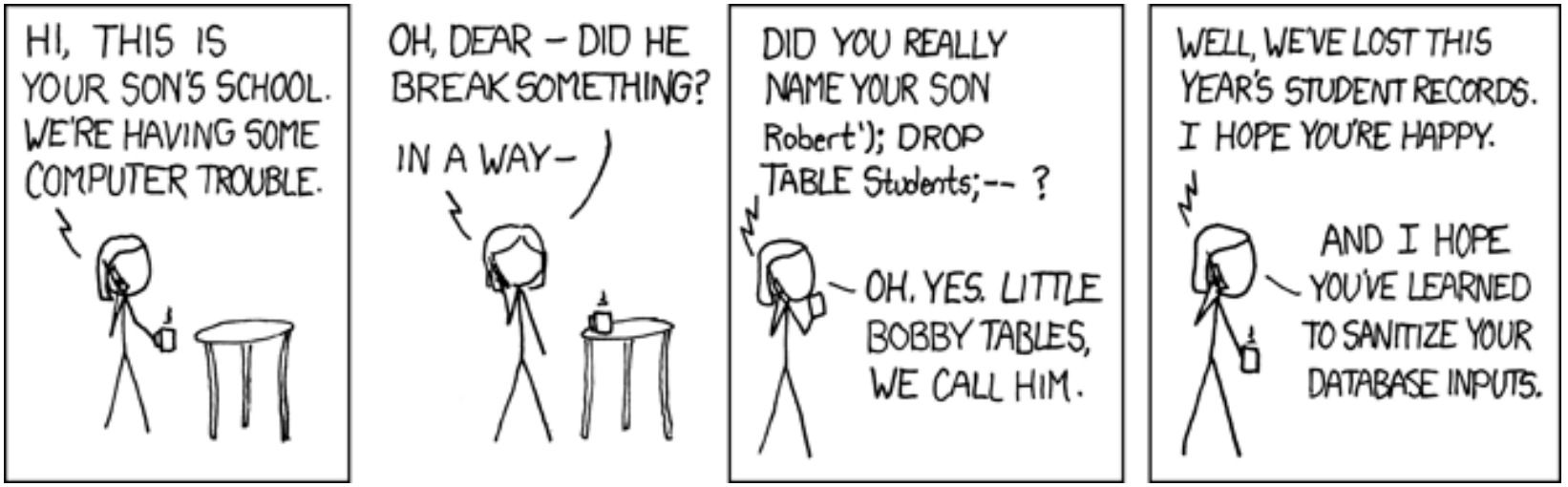
Attack: SQL Injection

What it is:

Client provides SQL code that runs against the database.

How it happens:

Client-contributed input is put into an SQL query.



XKCD #327 by Randall Monroe https://xkcd.com/327/

Attack: SQL Injection

What it is:

Client provides SQL code that runs against the database.

How it happens:

Client-contributed input is put into an SQL query.

How to prevent it:

Use SQLAlchemy

Use *parameterized queries*

Never: build SQL from user-provided strings

mysql_real_escape_string is evil preventing worse evil

Related Attack: Shell Injection

- Like SQL injection, but leaks to shell
- Arises when running external commands with user-provided input
- To prevent: *never* use system() or equivalents that call shell
  - Use replacements like Python subprocess that take an executable and a list of arguments
  - Make sure they do *not* call the shell with the arguments
- Otherwise: a tweet of ‘; rm –rf /’ causes many problems

Attack: Cross-Site Scripting (XSS)

What it is:

Attacker injects arbitrary HTML or JavaScript into other users' browsers, doing many evil things.

How it arises:

Including user-provided content in HTML without escaping or sanitizing.

Example: if ppl can have HTML in microblog posts

How to prevent:

Use Jinja2 (it escapes by default)

When users can provide HTML, sanitize it first

Sanitizing HTML

If HTML is **not allowed**, escape user data

- replace ‘<‘ with ‘`<`’, ‘&’ with ‘`&`’ at least

If HTML is allowed, sanitize it

- Forbid `script`, `style`, `link`, `iframe`, `object`
- Restrict `img` (only safe image types)
- **This is list is certainly not exhaustive**

Attack: Cross-Site Request Forgery (CSRF)

What it is:

Attacker's site performs action on yours as browser's logged-in user

How it arises:

Form handlers that can be called without verifying that the form calling them was created by the handling server

How to prevent:

Include a cross-site request forgery token in session and forms, verify they match

Can also use non-reusable *nonce* as CSRF token, but it's harder

Single-Origin Policy

The backbone of client-side web security!

What it means:

- JavaScript (and similar things) can only connect to page's **origin** (roughly: same domain name)
- HTML can load JS, CSS from other origins
- But resulting JS runs in the page's origin

This keeps browser JS from being attack platform against other sites

SOP Nuances

- JS can add script or link tags to load from other origins
  - But cannot directly access contents!
  - Can only launch GET requests
  - Loaded script can call a *callback* to make a change (JSONP)
- JS can submit forms to other sites
  - But that triggers a whole page reload, so it can't run afterwards
  - Cannot submit via $.ajax!

CSRF Tokens

- CSRF token is included in session
  - Only available to host server, host-provided JS/HTML
  - SOP means attacker JS cannot get user's CSRF token (without an XSS vulnerability)
  - Attacker server will not have session
- CSRF token is included in form
  - Again, SOP keeps attacker JS from getting it
  - Attacker server would receive different CSRF
- Result: attacker cannot get the right CSRF token for user, so they cannot submit a valid form

Designing for Security

- **Always** escape outputs, unless you **know** that code is needed and has been sanitized
  - Flask turns on Jinja escaping by default
- **Always** use SQLAlchemy or parameterized SQL queries
- **Do not** impose needless restrictions on passwords
  - Rejecting ‘invalid’ characters is unnecessary if you have clean data paths through code, to database
- **Always** store passwords securely (PB-KDF2, bcrypt, or scrypt if possible)
- **Layer** software and deployment to compartmentalize failure
- **Always** ask ‘how could this be abused?’

Protecting Yourself

Note: I'm going to get opinionated here.

- Use a password manager
  - LastPass, 1Password
- Use a strong, unique password for every site
  - Easy with a password manager
  - Compartmentalizes failure
- Use a very strong password for PW manager
  - I use Diceware
  - Length is more important than complexity

Protecting Yourself

- Run adblock (e.g. uBlock Origin)
  - I hate to recommend it, but it's necessary for safety
  - Ads are a very common malware delivery vector
  - Forbes ads have been used to deliver malware
- Do not use Flash or Java
  - Google Chrome has a (more) secure Flash, if you must
- Keep browser & OS up to date
  - Security patches are crucial
  - Software as complex as browsers **will** have vulnerabilities

Protecting Users

- Allow long passwords
- Do not interfere with password manager
  - Some financial orgs have lawyers who believe regulation requires trying to block the password manager
  - This is **very bad**, but there may be nothing you can do about it besides protest loudly.
- Do not use Flash or client-side Java; avoid other plug-in tech
- Minimize third-party resources
  - Privacy – each one can track your users
  - Security – any one can be hacked to attack your users
- Support HTTPS
- Don’t block Tor

Security is hard

- Very hard.
- Consult experts.
- I’m trying to give rules that are
  - Easy to follow
  - Conservative (following them won’t cause vulnerabilities, but might prevent useful features)