CSci 5271 Introduction to Computer Security Web security, part 1

Stephen McCamant University of Minnesota, Computer Science & Engineering

Outline

The web from a security perspective

SQL injection

Announcements intermission

Web authentication failures

Once upon a time: the static web

- HTTP: stateless file download protocol TCP, usually using port 80
- HTML: markup language for text with formatting and links
- All pages public, so no need for authentication or encryption

Web applications

- The modern web depends heavily on active software
- Static pages have ads, paywalls, or "Edit" buttons
- Many web sites are primarily forms or storefronts
- Web hosted versions of desktop apps like word processing

Server programs

- Could be anything that outputs HTML
- In practice, heavy use of databases and frameworks
- Wide variety of commercial, open-source, and custom-written
- Flexible scripting languages for ease of development PHP, Ruby, Perl, etc.

Client-side programming

- Java: nice language, mostly moved to other uses
- ActiveX: Windows-only binaries, no sandboxing Glad to see it on the way out
- Flash and Silverlight: most important use is DRM-ed video
- 🖲 Core language: JavaScript

JavaScript and the DOM

- JavaScript (JS) is a dynamically-typed prototype-OO language
 - No real similarity with Java
- Document Object Model (DOM): lets JS interact with pages and the browser
- Extensive security checks for untrusted-code model

Same-origin policy

- Origin is a tuple (scheme, host, port) E.g., (http, www.umn.edu, 80)
- Basic JS rule: interaction is allowed only with the same origin
- Different sites are (mostly) isolated applications





- "Web attacker" owns their own site (www.attacker.com)

 And users sometimes visit it
 Realistic reasons: ads, SEO

 "Network attacker" can view and sniff unencrypted
 - data
 - Unprotected coffee shop WiFi



Example SQL queries

 SELECT name, grade FROM Students WHERE grade < 60 ORDER BY name;
 UPDATE Votes SET count = count + 1 WHERE candidate = 'John';

Template: injection attacks

- Your program interacts with an interpreted language
- Untrusted data can be passed to the interpreter
- Attack data can break parsing assumptions and execute arbitrary commands



Strings do not respect syntax

- Key problem: assembling commands as strings
- 🗐 "WHERE name = '\$name';"
- Looks like \$name is a string
- 🖲 Try \$name = "me' OR grade > 80; --"



Non-string interfaces

- Best fix: avoid constructing queries as strings
- SQL mechanism: prepared statement Original motivation was performance
- Web languages/frameworks often provide other syntax



Lazy sanitization: whitelisting

- Allow only things you know to be safe/intended
- 🖲 Error or delete anything else
- Short whitelist is easy and relatively easy to secure
- E.g., digits only for non-negative integer
- But, tends to break benign functionality

Poor idea: blacklisting

- Space of possible attacks is endless, don't try to think of them all
- Want to guess how many more comment formats SQL has?
- Particularly silly: blacklisting 1=1

Attacking without the program

- Often web attacks don't get to see the program
 Not even binary, it's on the server
 Surmountable obstacle:
- Surmountable obstacle
 - Guess natural names for columns
 - Harvest information from error messages

Blind SQL injection

- Attacking with almost no feedback
- Common: only "error" or "no error"
- One bit channel you can make yourself: if (x) delay 10 seconds
- Trick to remember: go one character at a time



Outline

The web from a security perspective

SQL injection

Announcements intermission

Web authentication failures

Note to early readers

- This is the section of the slides most likely to change in the final version
- If class has already happened, make sure you have the latest slides for announcements

Outline

The web from a security perspective

SQL injection

Announcements intermission

Web authentication failures

Per-website authentication

Many web sites implement their own login systems

- + If users pick unique passwords, little systemic risk
- Inconvenient, many will reuse passwords
- Lots of functionality each site must implement correctly
- Without enough framework support, many possible pitfalls

Building a session

- HTTP was originally stateless, but many sites want stateful login sessions
- Building by tying requests together with a shared session ID
- Must protect confidentiality and integrity

Session ID: what

- 🖲 Must not be predictable
- Not a sequential counter
- Should ensure freshness
 - E.g., limited validity window
- 🖲 lf encoding data in ID, must be unforgeable
 - E.g., data with properly used MAC
 - Negative example: crypt(username || server secret)

Session ID: where Session IDs in URLs are prone to leaking Including via user cut-and-paste Usual choice: non-persistent cookie Against network attacker, must send only under HTTPS Because of CSRF (next time), should also have a non-cookie unique ID





Client and server checks

For usability, interface should show what's possible

- But must not rely on client to perform checks
- Attackers can read/modify anything on the client side
- Easy example: item price in hidden field

Direct object references

- Seems convenient: query parameter names resource directly
 E.g., database key, filename (path traversal)
- Easy to forget to validate on each use
- Alternative: indirect reference like per-session table
 - Not fundamentally more secure, but harder to forget check

Function-level access control

- E.g. pages accessed by URLs or interface buttons
 Must check each time that user is authorized

 Attack: find URL when authorized, reuse when logged off
- Helped by consistent structure in code