Outline

Security web intro (cont'd)
SQL injection
Web authentication failures
Announcements intermission
Cross-site scripting
More cross-site risks
Confidentiality and privacy
Even more risks

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

GET, POST, and cookies

- GET request loads a URL, may have parameters delimited with ?, &,
  =
  Standard: should not have side-effects
- POST request originally for forms
  Can be larger, more hidden, have side-effects
- Cookie: small token chosen by server, sent back on subsequent requests to same domain

User and attack models

- "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
Relational model and SQL
- Relational databases have tables with rows and single-typed columns
- Used in web sites (and elsewhere) to provide scalable persistent storage
- Allow complex queries in a declarative language SQL

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

SQL + injection (A1)
- Why is this named most critical web app. risk?
- Easy mistake to make systematically
- Can be easy to exploit
- Database often has high-impact contents
  - E.g., logins or credit cards on commerce site
- 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; --"
Using tautologies

- Tautology: formula that's always true
- Often convenient for attacker to see a whole table
- Classic: OR 1=1

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

Retain functionality: escape

- Sanitizing data is transforming it to prevent an attack
- Escaped data is encoded to match language rules for literal
  - E.g., \n and \n in C
- But many pitfalls for the unwary:
  - Differences in escape syntax between servers
  - Must use right escape for context: not everything's a string

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:
  - 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

Injection beyond SQL

- XPath/XQuery: queries on XML data
- LDAP: queries used for authentication
- Shell commands: example from Ex. 1
- More web examples to come

Outline

- Security web intro (cont'd)
- SQL injection
- Web authentication failures
- Announcements intermission
- Cross-site scripting
- More cross-site risks
- Confidentiality and privacy
- Even more risks

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
- If 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

Session management (A2)

- Create new session ID on each login
- Invalidate session on logout
- Invalidate after timeout
  - Usability / security tradeoff
  - Needed to protect users who fail to log out from public browsers

Account management

- Limitations on account creation
  - CAPTCHA? Outside email address?
- See previous discussion on hashed password storage
- Automated password recovery
  - Usually a weak spot
  - But, practically required for large system

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 (A4)

- 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 (A7)

- 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
Outline
Security web intro (cont’d)
SQL injection
Web authentication failures
Announcements intermission
Cross-site scripting
More cross-site risks
Confidentiality and privacy
Even more risks

John out of town this week
(At ACM CCS in Berlin)
Thursday and Friday office hours canceled
Best to include other staff on emails

Outline
Security web intro (cont’d)
SQL injection
Web authentication failures
Announcements intermission
Cross-site scripting
More cross-site risks
Confidentiality and privacy
Even more risks

XSS: HTML/JS injection (A3)
Note: CSS is “Cascading Style Sheets”
Another use of injection template
Attacker supplies HTML containing JavaScript (or occasionally CSS)
OWASP’s most prevalent weakness
A category unto itself
Easy to commit in any dynamic page construction

Why XSS is bad (and named that)
attacker.com can send you evil JS directly
But XSS allows access to bank.com data
Violates same-origin policy
Not all attacks actually involve multiple sites

Reflected XSS
Injected data used immediately in producing a page
Commonly supplied as query/form parameters
Classic attack is link from bad site to victim site
Persistent XSS
-Injected data used to produce page later
-For instance, might be stored in database
-Can be used by one site user to attack another user
  -E.g., to gain administrator privilege

DOM-based XSS
-Injected occurs in client-side page construction
-Flaw at least partially in code running on client
-Many attacks involve mashups and inter-site communication

No string-free solution
-For server-side XSS, no way to avoid string concatenation
-Web page will be sent as text in the end
  -Research topic: ways to change this?
-XSS especially hard kind of injection

Danger: complex language embedding
-JS and CSS are complex languages in their own
-Can appear in various places with HTML
  -But totally different parsing rules
-Example: "..." used for HTML attributes and JS strings
  -What happens when attribute contains JS?

Danger: forgiving parsers
-History: handwritten HTML, browser competition
-Many syntax mistakes given "likely" interpretations
-Handling of incorrect syntax was not standardized

Sanitization: plain text only
-Easiest case: no tags intended, insert at document text level
-Escape HTML special characters with entities like &lt; for <
-OWASP recommendation:
  & < > " ' /
Sanitization: context matters

- An OWASP document lists 5 places in a web page you might insert text
  - For the rest, “don’t do that”
  - Each one needs a very different kind of escaping

Sanitization: tag whitelisting

- In some applications, want to allow benign markup like `<b>`
- But, even benign tags can have JS attributes
- Handling well essentially requires an HTML parser
  - But with an adversarial-oriented design

Don’t blacklist

- Browser capabilities continue to evolve
- Attempts to list all bad constructs inevitably incomplete
- Even worse for XSS than other injection attacks

Filter failure: one pass delete

- Simple idea: remove all occurrences of `<script>`
- What happens to `<scr<script>ipt>`?

Filter failure: UTF-7

- You may have heard of UTF-8
  - Encode Unicode as 8-bit bytes
- UTF-7 is similar but uses only ASCII
- Encoding can be specified in a `<meta>` tag, or some browsers will guess
  - `+ADw-script+AD4-`

Filter failure: event handlers

- `<IMG onmouseover="alert('xss')">`
- Put this on something the user will be tempted to click on
- There are more than 100 handlers like this recognized by various browsers
Use good libraries

- Coding your own defenses will never work
- Take advantage of known good implementations
- Best case: already built into your framework
  - Disappointingly rare

Content Security Policy

- New HTTP header, W3C candidate recommendation
- Lets site opt-in to stricter treatment of embedded content, such as:
  - No inline JS, only loaded from separate URLs
  - Disable JS eval et al.
- Has an interesting violation-reporting mode

Outline

Security web intro (cont’d)
- SQL injection
- Web authentication failures
- Announcements intermission
- Cross-site scripting
- More cross-site risks
- Confidentiality and privacy
- Even more risks

HTTP header injection

- Untrusted data included in response headers
- Can include CRLF and new headers, or premature end to headers
- AKA “response splitting”

Content sniffing

- Browsers determine file type from headers, extension, and content-based guessing
  - Latter two for ~ 1% server errors
- Many sites host “untrusted” images and media
- Inconsistencies in guessing lead to kind of XSS
  - E.g., “chimera” PNG-HTML document

Cross-site request forgery (A8)

- Certain web form on bank.com used to wire money
- Link or script on evil.com loads it with certain parameters
  - Linking is exception to same-origin
- If I’m logged in, money sent automatically
- Confused deputy, cookies are ambient authority
CSRF prevention

- Give site's forms random-nonce tokens
  - E.g., in POST hidden fields
  - Not in a cookie, that's the whole point
- Reject requests without proper token
  - Or, ask user to re-authenticate
- XSS can be used to steal CSRF tokens

Open redirects (A10)

- Common for one page to redirect clients to another
- Target should be validated
  - With authentication check if appropriate
- **Open redirect**: target supplied in parameter with no checks
  - Doesn't directly hurt the hosting site
  - But reputation risk, say if used in phishing
  - We teach users to trust by site

Outline

Security web intro (cont'd)
SQL injection
Web authentication failures
Announcements intermission
Cross-site scripting
More cross-site risks
Confidentiality and privacy
Even more risks

Site perspective (A6)

- Protect confidentiality of authenticators
  - Passwords, session cookies, CSRF tokens
- Duty to protect some customer info
  - Personally identifying info ("identity theft")
  - Credit-card info (Payment Card Industry Data Security Standards)
  - Health care (HIPAA), education (FERPA)
  - Whatever customers reasonably expect

You need to use SSL

- Finally coming around to view that more sites need to support HTTPS
  - Special thanks to WiFi, NSA
- If you take credit cards (of course)
- If you ask users to log in
  - Must be protecting something, right?
  - Also important for users of Tor et al.

Server-side encryption

- Also consider encrypting data "at rest"
  - (Or, avoid storing it at all)
- Provides defense in depth
  - Reduce damage after another attack
- May be hard to truly separate keys
  - OWASP example: public key for website → backend credit card info
Adjusting client behavior

- HTTPS and password fields are basic hints
- Consider disabling autocomplete
  - Usability tradeoff, save users from themselves
  - Finally standardized in HTML5
- Consider disabling caching
  - Performance tradeoff
  - Better not to have this on user’s disk
  - Or proxy? You need SSL

User vs. site perspective

- User privacy goals can be opposed to site goals
- Such as in tracking for advertisements
- Browser makers can find themselves in the middle
  - Of course, differ in institutional pressures

Third party content / web bugs

- Much tracking involves sites other than the one in the URL bar
  - For fun, check where your cookies are coming from
- Various levels of cooperation
- Web bugs are typically 1x1 images used only for tracking

Cookies arms race

- Privacy-sensitive users like to block and/or delete cookies
- Sites have various reasons to retain identification
- Various workarounds:
  - Similar features in Flash and HTML5
  - Various channels related to the cache
  - Evercookie: store in n places, regenerate if subset are deleted

Browser fingerprinting

- Combine various server or JS-visible attributes passively
  - User agent string (10 bits)
  - Window/screen size (4.83 bits)
  - Available fonts (13.9 bits)
  - Plugin versions (15.4 bits)

(Data from panopticlick.eff.org, far from exhaustive)

History stealing

- History of what sites you’ve visited is not supposed to be JS-visible
- But, many side-channel attacks have been possible
  - Query link color
  - CSS style with external image for visited links
  - Slow-rendering timing channel
  - Harvesting bitmaps
  - User perception (e.g. fake CAPTCHA)
Browser and extension choices

- More aggressive privacy behavior lives in extensions
  - Disabling most JavaScript (NoScript)
  - HTTPS Everywhere (whitelist)
  - Tor Browser Bundle
- Default behavior is much more controversial
  - Concern not to kill advertising support as an economic model

Outline

Security web intro (cont'd)
SQL injection
Web authentication failures
Announcements intermission
Cross-site scripting
More cross-site risks
Confidentiality and privacy
Even more risks

Misconfiguration problems (A5)

- Default accounts
- Unneeded features
- Framework behaviors
  - Don't automatically create variables from query fields

Openness tradeoffs

- Error reporting
  - Few benign users want to see a stack backtrace
- Directory listings
  - Hallmark of the old days
- Readable source code of scripts
  - Doesn't have your DB password in it, does it?

Using vulnerable components (A9)

- Large web apps can use a lot of third-part code
- Convenient for attackers too
  - OWASP: two popular vulnerable components downloaded 22m times
- Hiding doesn't work if it's popular
- Stay up to date on security announcements

Clickjacking

- Fool users about what they're clicking on
  - Circumvent security confirmations
  - Fabricate ad interest
- Example techniques:
  - Frame embedding
  - Transparency
  - Spoof cursor
  - Temporal “bait and switch”
Crawling and scraping

- A lot of web content is free-of-charge, but proprietary
  - Yours in a certain context, if you view ads, etc.
- Sites don't want it downloaded automatically (web crawling)
- Or parsed and user for another purpose (screen scraping)
- High-rate or honest access detectable

Next time

- Firewalls, NATs, and network intrusion detection