

CSci 5271  
Introduction to Computer Security  
Protocols and web combined slides

Stephen McCamant  
University of Minnesota, Computer Science & Engineering

## Outline

- SSH
- SSL/TLS
- DNSSEC
- Announcements intermission
- The web from a security perspective
- SQL injection
- Web authentication failures
- Cross-site scripting

## Authentication methods

- Password, encrypted over channel
- .shosts: like .rhosts, but using client host key
- User-specific keypair
  - Public half on server, private on client
- Plugins for Kerberos, PAM modules, etc.

## Old crypto vulnerabilities

- 1.x had only CRC for integrity
  - Worst case: when used with RC4
- Injection attacks still possible with CBC
  - CRC compensation attack
- For least-insecure 1.x-compatibility, attack detector
- Alas, detector had integer overflow worse than original attack

## Newer crypto vulnerabilities

- IV chaining: IV based on last message ciphertext
  - Allows chosen plaintext attacks
  - Better proposal: separate, random IVs
- Some tricky attacks still left
  - Send byte-by-byte, watch for errors
  - Of arguable exploitability due to abort
- Now migrating to CTR mode

## SSH over SSH

- SSH to machine 1, from there to machine 2
  - Common in these days of NATs
- Better: have machine 1 forward an encrypted connection (cf. HAI)
  1. No need to trust 1 for secrecy
  2. Timing attacks against password typing

## SSH (non-)PKI

- When you connect to a host freshly, a mild note
- When the host key has changed, a large warning

```
*****
@  WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!  @
*****
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now
(man-in-the-middle attack)!
It is also possible that a host key has just been changed.
```

## Outline

- SSH
- SSL/TLS
- DNSSEC
- Announcements intermission
- The web from a security perspective
- SQL injection
- Web authentication failures
- Cross-site scripting

## SSL/TLS

- Developed at Netscape in early days of the public web
  - Usable with other protocols too, e.g. IMAP
- SSL 1.0 pre-public, 2.0 lasted only one year, 3.0 much better
- Renamed to TLS with RFC process
  - TLS 1.0 improves SSL 3.0
- TLS 1.1 and 1.2 in 2006 and 2008, only gradual adoption

## IV chaining vulnerability

- TLS 1.0 uses previous ciphertext for CBC IV
- But, easier to attack in TLS (vs. SSH):
  - More opportunities to control plaintext
  - Can automatically repeat connection
- "BEAST" automated attack in 2011: TLS 1.1 wakeup call

## Compression oracle vuln.

- $\text{Compr}(S \parallel A)$ , where  $S$  should be secret and  $A$  is attacker-controlled
- Attacker observes ciphertext length
- If  $A$  is similar to  $S$ , combination compresses better
- Compression exists separately in HTTP and TLS

## But wait, there's more!

- Too many vulnerabilities to mention them all in lecture
- Kaloper-Meršinjak et al. have longer list
  - "Lessons learned" are variable, though
- Meta-message: don't try this at home

## HTTPS hierarchical PKI

- Browser has order of 100 root certs
  - Not same set in every browser
  - Standards for selection not always clear
- Many of these in turn have sub-CAs
- Also, "wildcard" certs for individual domains

## Hierarchical trust?

- No. Any CA can sign a cert for any domain
- A couple of CA compromises recently
- Most major governments, and many companies you've never heard of, could probably make a `google.com` cert
- Still working on: make browser more picky, compare notes

## CA vs. leaf checking bug

- Certs have a bit that says if they're a CA
- All but last entry in chain should have it set
- Browser authors repeatedly fail to check this bit
- Allows any cert to sign any other cert

## MD5 certificate collisions

- MD5 collisions allow forging CA certs
- Create innocuous cert and CA cert with same hash
  - Requires some guessing what CA will do, like sequential serial numbers
  - Also 200 PS3s
- Oh, should we stop using that hash function?

## CA validation standards

- CA's job to check if the buyer really is `foo.com`
- Race to the bottom problem:
  - CA has minimal liability for bad certs
  - Many people want cheap certs
  - Cost of validation cuts out of profit
- "Extended validation" (green bar) certs attempt to fix

## HTTPS and usability

- Many HTTPS security challenges tied with user decisions
- Is this really my bank?
- Seems to be a quite tricky problem
  - Security warnings often ignored, etc.
  - We'll return to this as a major example later

## Outline

SSH  
SSL/TLS  
DNSSEC  
Announcements intermission  
The web from a security perspective  
SQL injection  
Web authentication failures  
Cross-site scripting

## DNS: trusted but vulnerable

- Almost every higher-level service interacts with DNS
- UDP protocol with no authentication or crypto
  - Lots of attacks possible
- Problems known for a long time, but challenge to fix compatibly

## DNSSEC goals and non-goals

- + Authenticity of positive replies
- + Authenticity of negative replies
- + Integrity
- Confidentiality
- Availability

## First cut: signatures and certificates

- Each resource record gets an RRSIG signature
  - E.g., A record for one name→address mapping
  - Observe: signature often larger than data
- Signature validation keys in DNSKEY RRs
- Recursive chain up to the root (or other "anchor")

## Add more indirection

- DNS needs to scale to very large flat domains like `.com`
- Facilitated by having single DS RR in parent indicating delegation
- Chain to root now includes DSes as well

## Negative answers

- Also don't want attackers to spoof non-existence
  - Gratuitous denial of service, force fallback, etc.
- But don't want to sign "x does not exist" for all x
- Solution 1, NSEC: "there is no name between `acacia` and `baobab`"

## Preventing zone enumeration

- Many domains would not like people enumerating all their entries
- DNS is public, but “not that public”
- Unfortunately NSEC makes this trivial
- Compromise: NSEC3 uses password-like salt and repeated hash, allows opt-out

## DANE: linking TLS to DNSSEC

- “DNS-based Authentication of Named Entities”
- DNS contains hash of TLS cert, don't need CAs
- How is DNSSEC's tree of certs better than TLS's?

## Signing the root

- Political problem: many already distrust US-centered nature of DNS infrastructure
- Practical problem: must be very secure with no single point of failure
- Finally accomplished in 2010
  - Solution involves ‘key ceremonies’, international committees, smart cards, safe deposit boxes, etc.

## Deployment

- Standard deployment problem: all cost and no benefit to being first mover
- Servers working on it, mostly top-down
- Clients: still less than 20%
- Will probably be common for a while: insecure connection to secure resolver

## What about privacy?

- Users increasingly want privacy for their DNS queries as well
- Older DNSCurve and DNSCrypt protocols were not standardized
- More recent “DNS over TLS” and “DNS over HTTPS” are RFCs
- DNS over HTTPS in major browsers might have serious centralization effects

## Outline

SSH  
SSL/TLS  
DNSSEC  
Announcements intermission  
The web from a security perspective  
SQL injection  
Web authentication failures  
Cross-site scripting

## Hands-on assignment 2 up

- If same group as HA1, host and group number are the same
  - Otherwise, contact Travis to change
- Instructions and VMs now available
- Due Friday, November 22nd

## Outline

SSH  
SSL/TLS  
DNSSEC  
Announcements intermission  
The web from a security perspective  
SQL injection  
Web authentication failures  
Cross-site scripting

## 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

## 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

## Outline

SSH  
SSL/TLS  
DNSSEC  
Announcements intermission  
The web from a security perspective  
SQL injection  
Web authentication failures  
Cross-site scripting

## 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

- 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., `\"` 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

SSH  
SSL/TLS  
DNSSEC  
Announcements intermission  
The web from a security perspective  
SQL injection  
Web authentication failures  
Cross-site scripting

## 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
- Built 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

- 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

- 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

## Outline

SSH  
SSL/TLS  
DNSSEC  
Announcements intermission  
The web from a security perspective  
SQL injection  
Web authentication failures  
Cross-site scripting

## XSS: HTML/JS injection

- 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 evil 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 right
- 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