Security: How systems behave in the presence of an adversary

- "Defense in depth": Swiss cheese model, where multiple layers of defense cover potential "holes"
- Computer Fraud and Abuse Act (CFAA)

Cryptography: Communicate securely in presence of an adversary — adversary typically modelled as MITM — can see, modify, and forge messages between A, B

3 GOALS (CIA):

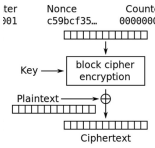
- Confidentiality: Mallory can't see what A, B are saying to each other.
- Integrity: Mallory can't modify msgs without being detected.
- Authenticity: Alice can confirm that a message really came from Bob, and vice-versa. May be a modified msg, but really from Bob.

One-time pads / Stream Ciphers

- achieve CONFIDENTIALITY
- OTP: Alice and Bob share secret, long string of bits **k**
  - → $c_i = p_i \oplus k_i \Rightarrow p_i = c_i \oplus k_i$
  - → information-theoretically secure
  - → CANNOT REUSE PAD — leak information $(a \oplus k) \oplus (b \oplus k) = a \oplus b$
  - → usually impractical — how to exchange k?
- Stream ciphers: Use PRG for OTP
  - → Alice and Bob choose PRG family G, w/ secret key k ($G_k()$)
  - → Need new key every time, never reuse PRG output bits — but provably secure!
  - → Can't prove secure PRGs exist :\
  - → Broken: RC4 ; Good: ChaCha20

Counter (CTR) Cipher Mode

- Turn a block cipher into stream
- Using a PRG, generate stream using k
- Have a counter (incremented per block)
- Have unique nonce per message
- Benefits: don't need padding, efficient parallelism / random access



For each block, encrypt Nonce || Ctr using key stream as key for block cipher

No chaining

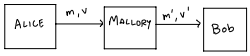
NEVER REUSE NONCE FOR SAME K

Diffie-Hellman Key Exchange

- Publicly known params:
  - → Large prime p (2048 bits)
  - → Generator g for p (usually small)
- Alice chooses secret key a, computes and sends $g^a \bmod p = A$
- Bob chooses b, sends $g^b \bmod p = B$
- $S = B^a \bmod p = A^b \bmod p = g^{ab} \bmod p$
- S is secret key!
- VULNERABILITY: MITM attack succeeds
  - → Alice and Bob each compute secret key s w/ Mallory — she has to intercept + transform every msg, but can read
- Defenses:
  - → Trust on first use, detect later (SSH)
  - → Communicate separately to verify same key
  - → rely on proximity to hope no MiTM
  - → digital signatures! HTTPS does this
- FORWARD SECRECY
  - if key stolen, don't want to crack previous messages
  - Use D-H to generate sesh keys
  - Authenticate session keys w/ separate long-term key
  - If long-term key leaked, past still secure

Message Integrity

- Idea: Have some verifier V for each message m. Verifier should be computable by Alice + Bob, but not Mallory (at least not easily)



Bob accepts iff $f(m') = v'$

⇒ ideally, this can only happen when $m = m'$, $v = v'$ as f unknown to Mallory

Want f to "look random"

- ideally, a random function (too inefft)
- so, we use PRF like HMAC-SHA256
- can't prove this is a PRF (would imply P = NP), but assumed
- For m, compute $v = \text{HMAC-SHA256}(m)$, with exact function in fn family determined by secret key k (known to Alice + Bob, but not Mallory

Block Ciphers

- Encrypt each n-bit block w/ some (reusable) key k

$E_k(p) \rightarrow c$; $D_k(c) = E_k^{-1}(c) = p$

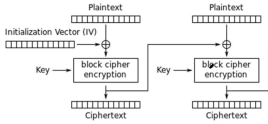
- k selects what function to use; there are $(2^n)!$ functions (2^n strings, $(2^n)!$ ordering, each ordering corresponds to a $2^n \leftrightarrow 2^n$ bijection)
- This is a Pseudorandom Permutation b/c it is bijective; a PRF compresses.

Cipher Modes

- Given black-box block cipher, how do we chain it to encrypt msg?
- ECB is flawed !!
  - encrypting each block independently with same key leaks info

Cipher Block Chaining (CBC):

- "Chain" ciphertexts to obscure
- Need initialization vector IV (random)
  - sent unencrypted
  - must be diff each msg



- Problems: can't encrypt in parallel; have to send IV w/ ciphertext each time

RSA (Rivest, Shamir, Adelman)

- Public-key cryptography — previous paradigms are 1-1 comm. What if you want to publish data + show integrity?
- Choose large primes p, q: $N = pq$
  - → Pick small e, s.t. relatively prime to $\phi(N) = (p-1)(q-1)$
  - → Compute d s.t. $ed \equiv 1 \bmod \phi(n)$
- Public key e, modulus N
- Anyone can encrypt: $c = m^e \bmod N$
- To decrypt: $m = c^d \bmod N$ $(m < N)$
- RSA Signatures:

  Sign m w/ $s = m^d \bmod n$

  Verify: $s^e \bmod N \stackrel{?}{=} m$

  Must be for FIXED msg
- Drawbacks: ABS 1000x faster; ∃ subtle mathematical attacks
- Bleichenbacher's Attack: If e is small (say 3), $m < N^{1/e}$, can just take cube root of c
- Key gen failures (take coprime of msg to find sig if $s < n$)
- Forge arbitrary msg signatures

Forging signatures on specific messages:
Suppose Bob will sign any message Mallory provides that he deems *non-sensitive*.
Mallory can trick Bob into signing *sensitive* **m'**:
Mallory computes: $\mathbf{m} := \mathbf{r^e m'}$ for random **r**
Bob signs **m**: $\mathbf{s} := \mathbf{m^d} \bmod \mathbf{N} = (\mathbf{r^e m'})^\mathbf{d} \bmod \mathbf{N}$
Then, Mallory finds signature $\mathbf{s'} = (\mathbf{m'})^\mathbf{d} \bmod \mathbf{N}$ by computing $\mathbf{s}(\mathbf{r^{-1}}) \bmod \mathbf{N} = (\mathbf{r^e m'})^\mathbf{d}(\mathbf{r^{-1}}) \bmod \mathbf{N}$

Hash Functions

- PRF: Mallory should not be able to distinguish PRF from known function family of size 2^n from truly random function in polynomial time
  - → "random": deterministic but rand
  - → n: # of bits in key
  - → could always do in exp time by checking against each possible $k \in [2^n]$
- Hash functions act as PRFs
  - no keys, just a fixed function
  - idea: prepend key to msg and hash to get v — this fails due to length-extension attack
  - MD-5, SHA-1 broken
  - SHA-256, SHA-512, SHA-3 secure

HMAC: MAC is method authentication code

$v = \text{HMAC}_k(m) = H(k \oplus c_1 \,||\, H(k \oplus c_2 \,||\, m))$

H: any hash function ⊕: XOR

c_1, c_2: publicly known constants

AES Block Cipher

- 2001, believed to be PRP, efficient
- 128-bit blocks; keys 128, 192, 256 bits
  - → 10, 12, 14 rounds depending on key
  - → generate r + 1 round subkeys
- Each round: 128-bit input, 128-bit subkey, 128-bit output
  - put input in 4x4 grid of bytes
  - ① Nonlinear function (lookup table)
  - ② Shift rows (i^{th} row by i)
  - ③ Multiply by invertible matrix
  - ④ XOR w/ subkey

Padding: How to make blocks a fixed size to use block ciphers?

- Needs to be unambiguous
- PKCS7: Add k bytes of value k each; always add at least 1 padding byte to remove ambiguity
- Add to end of msg to make n-multiple

Ciphers - Only Do Not Achieve Integrity

- Eg: In both CBC, CTR mode, flipping a bit in one block will have limited effect on how whole message is decoded
  - → in CBC, flipping ciphertext bit j in block i will:
  - → Corrupt block i of plaintext
  - → Flip bit j in block i+1 of plaintext
  - → NOTHING ELSE!!
- Elliptic Curve Crypto - more common now than DH, RSA

Web Platform: Basics

- Collection of technologies based on open standards
- Many different threat models:
  - → Malicious Website
  - → Website loading compromised resource
  - → Network attacker
- HTML — standard language for documents displayed in browser
  - escape certain chars — < > & " '
- JavaScript: runs in browser — can read + modify page content
  - DOM: Document Object Model API to allow JS to modify HTML
  - Pages can include scripts / load them from external resources

URLs

scheme : // host : port / path ? query # fragment

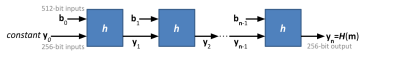
- scheme: http, https, etc.
- host: IP addr / domain name
- port: TCP port (default 443 for https, 80 http)
- path: which resource on server?
- query: parameter string passed to server
  - often key-value pairs, could be arbitrary
- fragment: not visible to client (what section of page?)

Length-Extension Attacks

- Many hash functions, like SHA-256, use Merkle-Damgard construction to hash arbitrary-length inputs to 256 bits.
- If $v = f(m) = \text{SHA-256}(k \,||\, m)$, this is vulnerable!
- Given $v = f(m) = \text{SHA-256}(k \,||\, m)$, for unknown m, attacker can calculate $v' = f(m')$, where $m' = m \,||\, \text{padding} \,||\, \text{arbitrary}$
  - use digest v as y_n for rest of message — as key already prepended to m, every thing else (sans h, y_0, padding scheme) is publicly known.
  - Final m" that hashes to v" will be:

    key || original msg || orig padding || new msg || new padding

- M-D construction:
  - → Have some compression func h
    - → Input: b (512 bit), y (256 bit)
    - → Output: 256 bits
  - → Split m into $b_0, b_1, \ldots, b_{n-1}$ with padding as required — ALWAYS add some padding
  - → Have some publicly known y_0 (random 256-bit)



Generating Randomness

- Need to select a truly random key - true randomness is hard; we use PRG
- PRG: Function g that isn't distinguishable from truly random function in poly-time — generates a STREAM of bits
- Given PRF f w/ key k, $g() = f_k(0) \,||\, f_k(1) \,||\, f_k(2) \ldots$ is PRG (random stream of bits)
- RNG is a good attack target — knowing PRG is VERY POWERFUL
- Eg: NSA backdoored Dual-EC DBRG (2006)

Authenticated Encryption

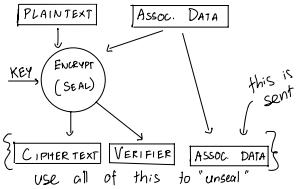
- Approach 1: Compose encryption + MAC
- Approach 2: Build primitive to do both
- Basically, decrypt fn can return "fail" now

How to Compose MAC + Encrypt

- Cryptographic Doom: If you perform any cryptographic operations on a message without verifying MAC FIRST, you're screwed
- Encrypt-and-MAC: E(P) || MAC(P)
- MAC-then-encrypt: E(P || MAC(P))
  - → Vulnerable to padding oracle attack
- Encrypt-then-MAC: E(P) || MAC(E(P))
  - → THIS IS SAFEST

Authenticated Encryption w/ Assoc Data

- AEAD — use unencrypted assoc data



Use all of this to "unseal"

- associated data is VERIFIED, so can check MAC first
- AES-GCM (Galois Counter Mode)
- ChaCha-Poly1305, common on mobile
- AES-GCM: Seal function consists of CTR encryption for plaintext, and Galois Field MAC of ciphertext + assoc data
  - → Same key used for both (non-generic composition)
  - → NEVER REUSE NONCE

HTTP Protocol

- Sequence of requests and responses for retrieving data
- Client sends: Method (GET / POST), path + query, headers
- Server: response code, headers, data
- LOADING A PAGE ⑥ Call JS functions in response to events
  - ① Load content @ URL
  - ② Parse HTML, run in-line JS
  - ③ Recursively fetch subresources

Cookies

- Cookies use a different scope than DOM — this leads to problems
- Scoped on ([scheme], domain*)
- Secure attribute: prevent cookies being sent over http
- HttpOnly attr: prevent cookies from being read by JS running in origin
- Site can SET cookie for own domain or any parent domain as long as parent domain is not public suffix (eg .com, .gov, .edu)
- You don't know who set a cookie when you receive it
- Site can READ cookie for own domain, parent domains. Scripts can read cookie for any path on domain — but diff path cookies not sent by default

Historical Confidentiality

- Caesar Cipher: shift everything by some offset (1) easy to brute-force (2) letter-freq attack!
- ETAOIN SHRDLU CMWFGY PBVKXJ QZ
- Vigènere Cipher: repeated Caesar cipher, except offsets determined by some key word (so n diff offsets applied sequentially)
  - → easy to break if n is known — solve each Caesar cipher individually
  - → Find n by looking at repetion-often kn apart

Padding Oracle Attack

- Suppose something uses MAC-then-encrypt
- Then, decryption works like this:
  1) m || v || pad = Decrypt (c)
  2) Check for PadError, check for MACerror
- If these error types distinguishable, you leak info
- Suppose attacker has oracle that returns (1) "valid" (2) PadError (3) MACerror on any ciphertext
- Then, iterate thru all 256 possibilities for j^{th} bit of $(i-1)^{th}$ block — each of these correspond to one value of j^{th} bit of i^{th} block
  - exactly one value will create MACerror/valid — rest will have padError
  - you know padding scheme, so you can invert the XOR to get original byte j in i.
  - Then, you can keep decoding blocks back to front
  - chop off block after you're done

Sizes for Safety

- 128 bit keys "classically" safe
- 256 bit keys quantum-safe
- Need 2n bits of output for collision resistance of 'n bits"
  - → SHA-256 has 128 bits (classically safe) collision resistance

Estimating what's feasible to compute?

$2^{64} \approx 10^{19}$ $2^{128} \approx 10^{38}$ 1 year = 3×10^7 s

CPU mining: $\approx 10^8$ SHA-256/s

GPU mining: $\approx 10^{11}$ SHA-256/s

ASIC mining: $\approx 10^{14}$ SHA-256/s

Bitcoin miners globally: 10^{20} SHA-256 blocks/s

Same Origin Policy (SOP)

- Separates content into origins; restricts data flows between them. Eg: attacker.com can't load Gmail in an iFrame and read your messages
- Origin: scheme :// domain : port
- What's isolated? Cookies, DOM storage, JS namespace, local hardware perms
- ALLOWED: cross-origin writes (links, redirects, form submissions), cross-origin embedding (images, videos, iFrames, etc.)
- CORS: server can allow cross-origin requests from specified origins

CSRF Attacks

- Core idea: Cookies being sent are determined by domain of resource being requested
  - → if a page on attacker.com requests image from bank.com, bank cookies sent (unless samesite attr. set, etc.)
- Malicious site can send requests to other sites — auth cookies will be sent w/ it
  - → eg. send POST req to bank.com to transfer money; login to site under attacker acct to monitor activity
- Cookie-Based Authentication Alone Bad

CSRF Defenses

- Referer Validation: HTTP request contains URL of page making request (however, not always sent! Doesn't always work)
- Secret Token Validation: Pages on actual site include secret token (csrf-token) in all requests — helps block cross-site requests in general (session-dependent token). Attacker cannot retrieve due to SOP
- Same Site Cookies: SameSite attribute prevents cookie from being sent cross-site
  - → Strict: not sent in ANY context, even when following regular link
  - → Lax: Sent on cross-site links, but not subrequests (usually want this)

XSS Defenses

- Validation + Escaping: Check all user input robustly to prevent any scripts
  - → also encode all special characters in output
  - → adopt positive security policy — specify what IS allowed, rather than what isn't (less likely to have loopholes)
- Content Security Policy:
  - → http header, tightly specifies what scripts allowed to execute (what domains scripts must come from) — inline scripts banned. More modern approach

PROJECT 2: For SQLi, terminate string w ' and then do OR 1=1 # to comment out

For XSS, can insert between <script> tags or onerror of img tags, etc. Insert into URL s.t. it will be put into HTML somehow (echo user inputs). Wait for document to load. this is vulnerable:

For CSRF, can have hidden code on attacker's website that logs in/ submits other req to other websites. Use POST reqs w/ FORMS, redirect to iframes. If there are XSS vulnerabilities, can bypass CSRF tokens by setting/getting cookies.

PROJECT 1: Vulnerability exists when using pure hash functions as MACs. Can do length extension to hash some longer message, even if key unknown.

bytes.fromhex(---) Sha256(m).hexdigest

```
$ openssl dgst -sha256 file1
$ fastcoll -o file1 file2 -p prefix
```

→ generates MD5 collisions w/ same prefix and diff ending

→ Hashes will be same if same Suffix added (Merkle-Damgard)

BLEICHENBACHER: padding not checked, so less FFs — this makes $m^3 < n$, so ciphertext is easy to decode! Can forge RSA signatures.

Physical Layer (L1): "Encrypt @ higher"

- Easy to tap, jam, intercept. Historically poor encryption — WPA, 2G, 3G, 4G all broken

Link Layer (L2): How do I get to next hop?

- Assume: some "physical" link b/w nodes
- Ethernet / WiFi: Send packets (frames) to hosts on local network (LAN)
- Devices ship w/ globally-unique 48-bit MAC address (but can change arbitrarily)
- EtherType: L3 protocol spec
- Ethernet Switches: each port connects to host / other switch. LEARNS MACs on each port eventually to route - broadcast to all if unknown
- These do not scale! We need IP for L3
- Plaintext; packets could be corrupted, lost, etc.

SQL Injections

- Website does not sanitize inputs to SQL statements — can have data interpreted as code (DANGEROUS)
- Typical SQL statement: SELECT * FROM users WHERE username = '$user' AND password = '$pwd'
- What if $pwd ends SQL stmt and does other stuff? Ex: pwd = ' OR 1=1 -- (selects all rows from DB)
- Defense: Don't build SQL cmds yourself — use parametrized SQL statements, where query and arguments are passed separately

XSS (Cross-site scripting)

- Type of injection — data interpreted as code by insecure site
- Client-side attack

Reflected XSS

- Vulnerability when site echoes inputs back to user w/o cleaning
- Could put code in query of malicious URL — site could accidentally put it in HTML (thinking it's text), and it gets executed
- Danger: Even one page having vulnerability exposes whole site, as script not blocked by SOP any longer

Stored XSS

- Vulnerability: Site stores + displays unsanitized user content to other people
- Attacker can upload a script as part of some content. This content could be accessed by other users, at which point script is executed on their browsers!
- Samy Worm: 2005. Uploaded script to MySpace that replicated in user profiles when they viewed his profile. Propagated across site.

Network Layer (L3): Packets to final dest

- IP (v4, v6). Routing + Fragmentation - no security; plus packets could be lost
- IP header on every packet: source addr (unverified), dest. addr, other stuff
- v4 addr: 32 bits, exhausted
- v6 addr: 128 bits, AA:BB::XX:YY
  - → :: means all 0s
- IP addr has network prefix, host suffix
  - → CIDR notation: how many bits in prefix
  - → 144.212. ... (24) OR a mask
- Same prefix = same LAN — just go over ethernet. Else, send to router (network gateway)
- Private IPv4s:
  - 10.0.0.0 → 10.255.255.255
  - 172.16.0.0 → 172.16.255.255
  - 192.168.0.0 → 192.168.0.255
- Residential ISPs only get 1 addr - Home routers assign each device address from private range; rewrite packets in both directions
- localhost: 127.0.0.1

Threat Model:

- Off-path: talk to hosts, but can't see packets (most attackers @ first)
- On-path: See + add packets, but can't modify or block - ethernet passive tap, for instance
- In-path: MITM, full power

Packet Capture: Hosts can log all packets arriving on network interface (pcap format, tools like Wireshark)

- mtr, traceroute to see how your packets travel

TLS (Transport Layer Security)

- Protocol stacked on top of TCP, which is a plaintext communication protocol.
  - → TCP provides "phone-call" semantics
- HTTP → TLS → TCP → Internet
- Threat model: malicious network
  - Govt. censorship, data harvesting, compromised routers, route hijacking, etc.
- Provides: Confidentiality + Integrity for data while in transit. Client can also authenticate server's identity
- Does not Provide: Malware resistance, tracking, metadata analysis (ie which sites you visited), DDOS, etc.
- TLS 1.3 (2018): Major redesign
- Confidentiality + Integrity thru AEAD ciphers
- Authentication w/ certificates + public key crypto

TLS Certificates

- Server signs hash of handshake w/ private key and provides public key. How do we know public key is legit??
- Server provides a certificate w/ identity + public key. This is signed by some Certificate Authority (CA)
  - public keys of CAs hardcoded into browsers (root CAs)
  - TLS uses X.509 certificates
- Often a chain of certificates leading back to some root CA
  - root CA has self-signed certificate w/ public key — this means that even if root key stolen, existing certs still legit b/c signed with public key who's private key not known
- Obtaining a certificate:
  - Server proves identity thru email, http, DNS
  - ACME: open protocol that automates this
- Public Key Infrastructure (PKI)
  - → Maintained by CAs (100s exist), browser developers, CA/Browser forum
- Revocation lists exist if CA revokes
- Browser shows warning if cert expired
- HTTPS now ubiquitous — negative indicators now for http

Address Resolution Protocol (ARP)

- map IP addresses to MAC addresses (LAN)
- Host that needs MAC for IP broadcasts ARP packet to LAN — whoever has it replies. Host caches this mapping

ARP Spoofing: Anyone on LAN can claim to have that MAC to both user and router — become MITM. Typically no prevention for this — must analyze @ network-level

Router: Device w/ MULTIPLE link layers, each on diff network. Forwards packets to neighbor to get it closer to destination.

Border Gateway Protocol (BGP): ISPs use to announce network prefixes, connectivity. Routers compute route tables from this.

- no authentication! ISP can announce someone else's network. BGP Hijacking is common!
- RPKI: sign records — slow adoption

Transport Layer (L4): Add ports, encryption

- Application on server ↔ on top of bare packets identified by 16-bit port # — allow concurrent connections. 0-1023 usually require superuser priv.
- 32768-65535 EPHEMERAL — use for source ports
- HTTP: 80 • HTTPS: 443 • SMTP: 25 • SSH: 22

User Datagram Protocol (UDP): Lightweight Wrapper on IP (L4)

- Unreliable data transfer, but very fast!
- Used by real-time communications apps, DNS, QUIC
- Vulnerable to impersonation by off-path attacker

Transmission Control Protocol (TCP):

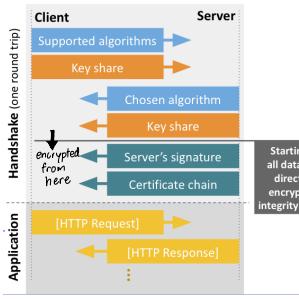
- Connection-oriented, reliable byte stream, congestion control. NO STRONG SECURITY
- sequence #s, acknowledgement #s to reassemble

TLS Handshake

① Negotiate Crypto algorithms
- find mutually-supported ones
- → Key Exchange Algo (eg. DHE)
- → Signature Algo (eg. RSA w/ SHA-256)
- → Symmetric Crypto Algo (AEAD like AES-GCM)

② Establish Shared Secret
- Diffie-Hellman for forward secrecy

③ Authenticate Server
- Server signs hash of all previous messages w/ private key
- sends certificate chain to authenticate public key — client checks
  - → if public key not verified, anyone can sign message and give "correct" public key.



- Client guesses which key-exchange algo server will pick — easy if comm. before. Reduces expected # of round-trips.

Fooling Users

- Homographs: Attackers can buy look-alike domain names
  - IDN homographs: different unicode chars look the same; can get perfect looking match for domain name in browser
  - → only heuristics for defense
- Stripping: Some browsers default to http. (SSL Strip)
  - MITM connects w/ server using https, but sends to user over http — now can read user data
  - Mitigation: HSTS. Server sends header on first visit to only use https for x seconds. HSTS preload lists shipped w/ browser for first visit protection
  - → hard-fail if https not available
- Phishing: https cannot prevent
  - browsers use ML to detect

Bugs in TLS

- TLS complicated, hard to code
  - Eg: Apple accidentally skipped cert checking for a year
  - Null prefix attack: diff in Pascal/C-style strings
  - Memory leaks leak random data
  - Mitigate using formal verification
    - miTLS is formally-verified lib

Server Vulnerabilities

- TLS servers must protect private keys — always have isolation, access control. Contact CA for cert revocation if keys leaked.

TCP Handshake: TCP carries data in segments

- Session starts w/ 3-way handshake — client and server pick init. sequence number, acknowledge each others
- On-path TCP attacks are trivial
- Off-path attacker can't do handshake without guessing server's seq number — $1/2^{32}$, so some security props.

Application Layer (L5):

- Eg: SSH, DNS, SMTP, HTTP, etc.
- DNS: Delegatable, hierarchical database used to resolve hostnames to IP addresses
  - control over regions of namespace (unrelated) delegated to AUTHORITATIVE servers (DNS Zones)
  - 13 root authoritative servers exist; these delegate to servers for .com, .edu, etc.
  - Cache responses for TTL (time to live)
- DNS Hijacking: Attacker hacks router so it points requests to attacker's DNS server; now, attacker can give whatever IP address for any domain name. TLS certs protect.
- Hosts file Hijacking: OS has DNS override file — can try and change this.
- DNS monitoring/blocking: Plaintext DNS allows network to see what websites you are visiting (and potentially stop you).

Defenses: DNSSEC adds authentication to DNS responses using TLS-like PKI. Issue: Slow adoption.

Site Design Problems

- Mixed content: resources loaded over http inside https page
- Secure attribute on cookies not set
- TLS sends domain name in plaintext
  - censorship, privacy concerns
  - users can use Tor / VPNs
  - Encrypted SNI standard emerging

CA Weaknesses

- Fooling Validation: Attacker could falsely get certificate for someone else's domain name
  - → Mitigation: Domains can limit which CAs can issue certs in CAA DNS record.
  - → Can prohibit common admin emails
  - → Multi-perspective validation
- CAs can get hacked
  - Small, intermediate CAs especially
  - Mitigation: Certificate Revocation List (CRL) public, crawled often
  - → Certificate Transparency (CT) logs

TLS Protocol Vulnerabilities

- Versions < 1.3 vulnerable
- Compression Attacks: TLS has compression but exposes len of plaintext — this can enable decoding (repeated strings compressed)
- Export-grade crypto: US law limited strength of overseas crypto until 2000
  - MITM attacker can tell server that only export-grade supported by "client".
  - client's buggy browser accepts export-grade, even if not asked for — now, weak crypto
  - FREAK Attack, patched in TLS 1.3
- Mitigate: Tools like SSL Labs

- PKCS pads hash of true message to create what you actually want to sign (structured form). - if not checked correctly, can have arbitrary bytes at end - so, can make PKCS be a perfect cube, and then set cube root to be signature. When checked with S^3, will get PKCS m.
  - ★ need $m < n$

Off Path DNS Cache Poisoning (2008):

- Victim visits attacker's site, which executes some JS that sends both DNS reqs to [???].bank.com
- Attacker can spoof responses of queries by guessing 16-bit query ID, being close to victim. Can temporarily redirect domains to whatever IP addr.
- Can increase randomization by randomizing source IP addr, randomizing capitalization in query

DOS Attacks: Overwhelm host/network w/ traffic

- Assymmetry: when small requests have large response
- TCP-SYN flooding: Space allocated before handshake complete. Fixed by making server seq. num the MAC of client's, so no space req.
- Amplification: attacker queries server's w/ victim's IP as the source, inundating victim w/ responses.
- Botnets: Many devices controlled by 1 person.
- ISPs should drop packets w/ spoofed IPs to prevent.
- CDNs like cloudflare can help.
- DNS-over-TLS, DNS-over-HTTPS operate DNS over encrypted channel (TLS instead of plaintext UDP)