Midterm content: Lectures 1-14, projects 1 and 2
One double-sided cheat sheet allowed
There is a practice exam available

Cryptography

Kerckhoffs's Principle: "A cryptosystem should remain secure even if attackers know everything but the key."

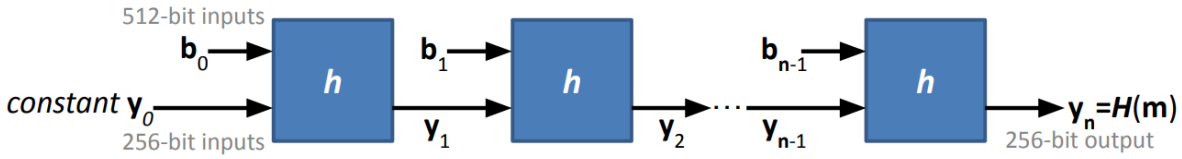
Security properties we want

- C- confidentiality; keep content of a message secret from an eavesdropper
  - Get by: Encrypting your message *m*, using a shared key only the sender and receiver know.
    - In practice: get shared key using diffie-hellman. Use ChaCha20 for the bit stream function to create one-time pads, XOR messages with those one-time pads to encrypt. Or
- I- message integrity; attacker cannot modify messages without being detected
  - Get by: send a verifier *v* with your message *m*. Verifier function *f* created using a shared key only the sender and receiver know. Verify *f*(*m*) == v.
    - In practice: get shared key using diffie-hellman, use HMAC-SHA-256 for the verifier function (as opposed to just SHA-256, to avoid length extension attack).
- A- sender authenticity; we can be sure the message came from the stated sender
  - Get by:

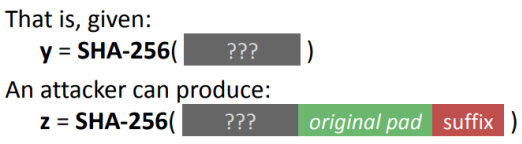
Properties of a strong hash function

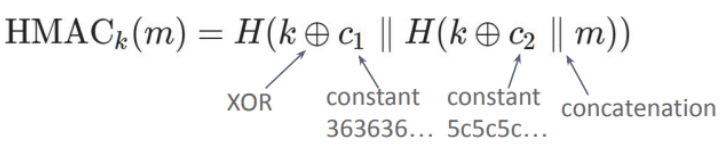
- Preimage resistance- hard to find input given output
- Second-preimage resistance- given an input, hard to find another input with the same output
- Collision resistance- hard to find any pair of inputs with the same output

Merkle-Damgard construction



Attack: length extension attack





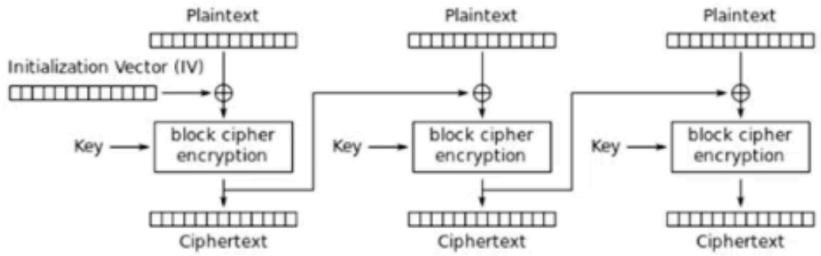
Defense: HMAC

Randomness

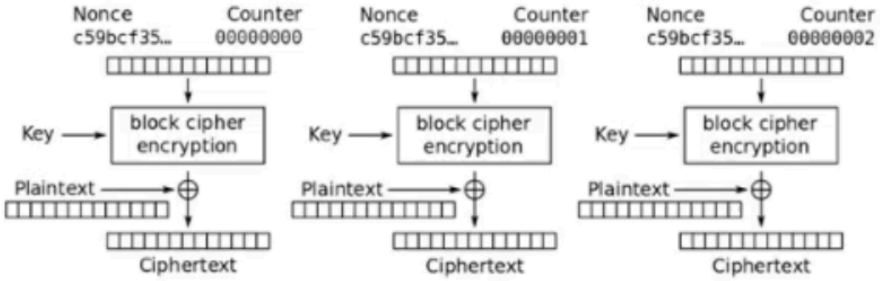
- Pseudorandom function (PRF): We choose a secret key *k* with *n* bits. We create a family of 2^*n* functions *f*. We let our verifier function be either a truly random function or the *k*th function in this family of functions *f_k*(). Must not be able to distinguish between using the PRF versus using a truly random function in polynomial time without knowing *k*.
    - Used for: verifier function that calculates verifier for a message
    - In practice: treat HMAC-SHA-256 as a PRF (though we can't prove it really is one)
- Pseudorandom generator (PRG): We choose a secret key *k* with *n* bits. We create a family of 2^*n* functions *g*. We let our bitstream function be either one that outputs a truly random stream of bits or the *k*th function in this family of functions *g_k*(). Must not be able to distinguish between using the PRG versus using a truly random function in polynomial time without knowing *k*.
    - Used for: shared key
    - If we have a secure PRF, we can construct a secure PRG (and vice versa)
    - In practice: treat ChaCha20 as a PRG
- Pseudorandom permutation (PRP): a function which produces outputs that cannot be distinguished from a truly random permutation in polynomial time without knowing *k*.
    - Used for: block cipher encryption
    - Like PRFs and PRGs, we don't know if they exist. Best we can do is design a complex function that is hopefully invertible only if you know *k*.
    - All PRPs are PRFs; the set of PRPs is a subset of the set of PRFs
    - In practice: treat AES as a PRP

Block ciphers

- Padding- bytes added to the end of a message to make it a multiple of a number of bytes
    - We use: PKCS7- n bytes of value n at the end of the message to communicate the length of the padding
- Malleability- a ciphertext is malleable when can you transform a it into another ciphertext that decrypts to a related plaintext, without knowing the plaintext. CBC and CTR modes are malleable.
- Cipher modes- algorithms for applying block ciphers to more than one blocks
    - Encrypted Codebook Mode (ECB)- encrypt each block independently. DON'T USE
    - Cipher Block Chaining Mode (CBC)- XOR block plaintext with previous block ciphertext. XOR first block of plaintext with initialization vector (IV).

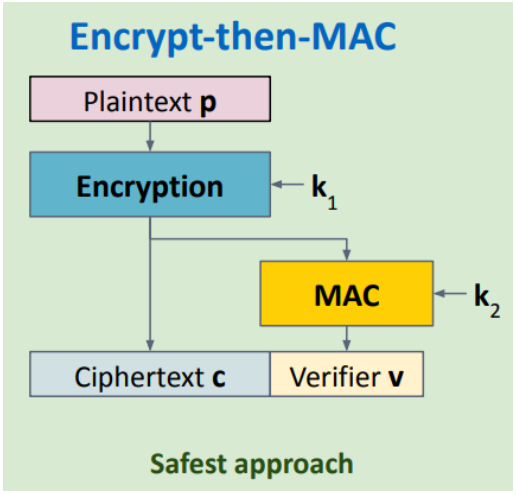


- Counter Mode (CTR)- for each block, concatenate a one-time use nonce with the block's number. Encrypt that, then XOR the plaintext with that

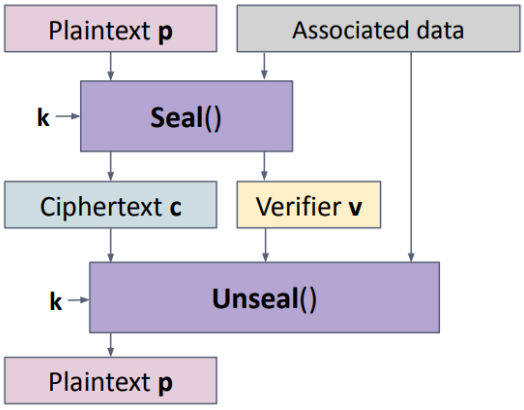


To combine confidentiality and integrity:

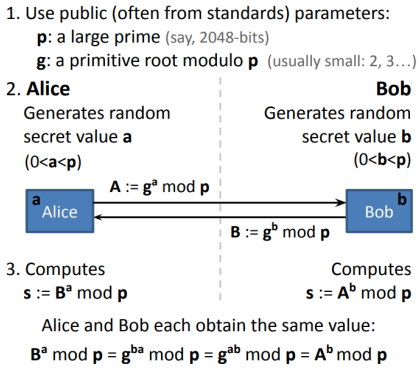
- Encrypt-then-MAC



- Authenticated Encryption with Associated Data (AEAD)



Diffie-Hellman exchange- p, g, A, and B are public. a and b are private.



It is too hard for the attacker to try and figure out a from A or b from B (discrete log problem). Alice and Bob can compute the shared key s.

Have both long term key and session keys to prevent retrospective decryption

Susceptible to MITM! Maybe use signatures…

"Textbook" RSA

Key generation (in secret):

1. Generate large random primes, **p** and **q** (say, 2048 bits each)
2. Compute "modulus" **N** := **pq** (say, 4096 bits)
3. Pick small "encryption exponent" **e** (say, 65537) Must be relatively prime to (**p** – 1)(**q** – 1)
4. Compute "decryption exponent" **d** such that **ed** ≡ 1 mod (**p** – 1)(**q** – 1)

Yields **RSA key pair**: **Public key**: (**e**, **N**)
Private key: (**d**, **N**)

Public-key encryption !!! **Digital signatures** !!!
Encrypt: $c := m^e \bmod N$ **Sign:** $s := m^d \bmod N$
Decrypt: $m := c^d \bmod N$ **Verify:** $m \stackrel{?}{=} s^e \bmod N$

RSA- e and N are public. p, q, and d are private. asymmetric keys. Slow.
Susceptible to attacks:

- Small *e* attack
- Stereotyped message attack
- Ciphertext malleability
- Key generation failures (both parties choose the same p)
- Forging signatures on random messages (pick random s, compute m that yields that s)
- Forging signatures on specific messages

Use better constructions.

A Secure Channel Protocol

Putting it all together…

	Alice	Mallory	Bob
	Only Alice knows Alice's private key (d_a,N_a)	**Everybody** knows Alice's public key (e_a,N_a) Bob's public key (e_b,N_b)	**Only Bob** knows Bob's private key (d_b,N_b)
1. Use **Diffie-Hellman** to generate a shared secret	Generates random a k := $B^a \bmod p$	$A := g^a \bmod p$ → ← $B := g^b \bmod p$	Generates random b k := $A^b \bmod p$
2. Use **RSA signatures** to confirm we're really talking to each other (*sender authenticity*)	x := PSS(SHA-256(A,B)) Verifies that $(s_b)^{e_b} \bmod N_b$ is correct PSS-padded x	$s_a := x^{d_a} \bmod N_a$ → ← $s_b := x^{d_b} \bmod N_b$	x := PSS(SHA-256(A,B)) Verifies that $(s_a)^{e_a} \bmod N_a$ is correct PSS-padded x
3. Derive symmetric keys for each direction using a **PRF**	k_a := HMAC-SHA256$_k$("→") k_b := HMAC-SHA256$_k$("←")		k_a := HMAC-SHA256$_k$("→") k_b := HMAC-SHA256$_k$("←")
4. Use **AEAD** (e.g., **AES-GCM**) or **encrypt-then-MAC** messages (*integrity* and *confidentiality*)	c_a,v_a := Seal(k_a,"Hi Bob!") m_b := Unseal(k_b,c_b,v_b)	c_a,v_a → ← c_b,v_b	m_a := Unseal(k_a,c_a,v_a) c_b,v_b := Seal(k_b,"Hi Alice!")

* Please don't actually code anything like this. Just use TLS!

Cryptography Attacks

- Length Extension- given H(m), can compute H(m || original padding || suffix)
    - Defense- HMAC
- Hash collision- hash collisions
    - Defense- use a hashing function that hasn't been broken (e.g SHA-256)
- Padding Oracle- server tells you (or it can be deduced) that a ciphertext was rejected, and you can distinguish whether the reason is because of a padding error or MAC error. Can leak plaintext
    - Defense- Encrypt then MAC (instead of MAC then Encrypt!)
- Bleichenbacher- did not check for correct number of FF bytes and last bytes ignored. Can forge valid signatures for short messages for e = 3
    - Defense- check signature correctly! Verify it has the right number of FF bytes and padding.

Web

Same-Origin Policy (SOP): a site cannot read responses to HTTP requests made to a different origin.
Cross-Origin Resource Sharing (CORS): allow sites of different origins to read HTTP requests made to your site under some circumstances. Do this by a pre-flight request made when attempting a cross-origin request.

Cookies- sent by server, stored in browser. So browser remembers you're logged in

Cookies can be set or read for its own domain or any parent domain (e.x login.site.com can set cookies for login.site.com and site.com)

Make sure to set the following cookie attributes:

- Secure- so cookies cannot be sent over HTTP where network eavesdroppers can see them in plaintext
- HttpOnly- so cookie cannot be accessed by the DOM
- SameSite=Lax or SameSite=Strict- so browser doesn't include cookies in cross-site requests

Web Attacks

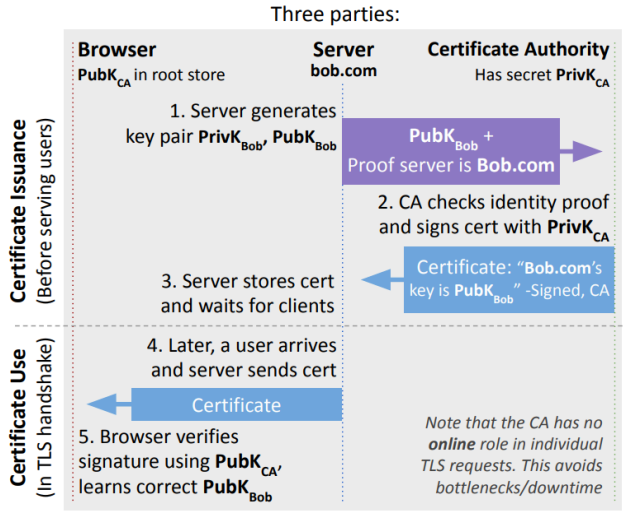
- SQL Injection- user can input data that is mistakenly interpreted as code
    - Defense- use prepared statements
- XSS- user can inject scripts into pages generated by a web application
    - Defense- input validation, output escaping, set Content-Security-Policy: default-src 'self' in HTTP header
- CSRF- causes user's browser to perform unwanted actions on a different site on the user's behalf

    - Defense- embed secret value in each request (session dependent), or set SameSite cookie attribute

TCP- protocol that lets devices over the internet communicate with each other in plaintext
TLS- encrypts communicate. HTTP over TLS = HTTPS. Provides CIA

TLS handshake

- Negotiate Crypto Algorithm- Server chooses most preferred that client supports
- Establish Shared Secret- Diffie-Hellman
- Authenticate the Server- server signs, and client verifies, a hash of entire transcript up to this point.



Client gets server's public key from a certificate signed by a certificate authority
A server can prove it's identity to a CA via Email, HTTP, DNS. Automated by ACME

CAs can give permission to other CAs to sign certificates

HTTP Attacks

- Homograph- website url looks like legit website to set up phishing attack
    - Defense- browsers block a lot of these, but users should be careful
- Stripping- attacker plays MITM when plaintext HTTP requests sent by client
    - Defense- use HTTP Strict Transport Security (HSTS) HTTP header to enforce only using HTTPS. Join HSTS preload list
- Phishing- trick user into inputting sensitive information into your website
    - Defense- Google Safe browsing; identifies dangerous websites, warns users
- Mixed Content- when resources on an HTTPS page are loaded over HTTP
    - Defense- use HTTPS for all resources
- Unencrypted cookies- default behavior on HTTPS sites
    - Defense- set Secure attribute on cookie so it's only sent over HTTPS
- Metadata visible to network- privacy/censorship concerns
    - Defense- Tor, VPN, encrypted Server Name Indication (SNI)
- Trick CA into thinking you own a domain you don't own

    - Defense- limit which CAs can issue certificates for your site, prohibit users from making certain email addresses, multi-perspective validation
- Hack CA- steal its private key
    - Defense- revoke certificates from hacked CA, certificate transparency to quickly discover compromises
- TLS buggy implementations- goTo fail, Mozilla BERserk, Null Prefix Attack, OpenSSL heartbleed
    - Defense- formal verification techniques, update server software
- TLS Protocol vulnerabilities- RC4, compression, CBC, Export related attacks
    - Defense- use TLS 1.3

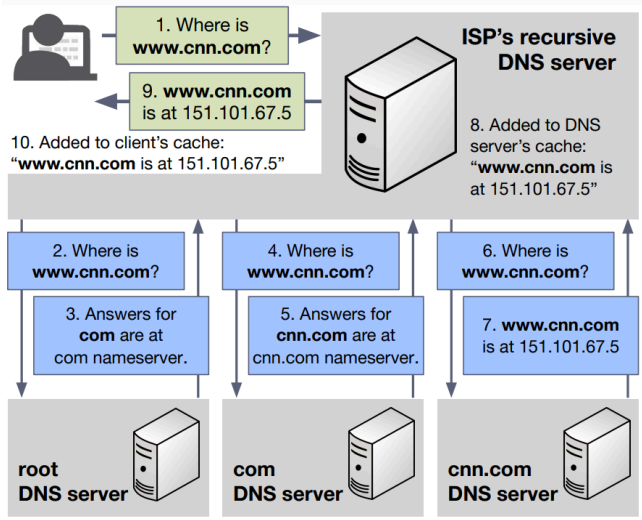
Networking

Network threat models:

- Off-path- talk to hosts, can't see packets
- On-path- can see packets, not modify
- In-path- can see and change packets

5 layers

- Physical- how bits get translated into electrical, optical, or radio signals
    - Examples- wired electrical links, Fiber optic links, Radio links
    - Attacks- Eavesdropping, Jamming, IMSI catchers, weak encryption
        - Defenses- physical security, encrypt at higher levels
- Link- provide a point-to-point link to get packets to next hop
    - Examples- Ethernet, Wifi, Cellular
- Internet Protocol (IP)- get packets to final destination over arbitrarily many hops
    - Examples- IPV4 (2^32), IPV6 (2^128)
    - Reserved for private use- 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16, localhost is 127.0.0.1
    - Attacks- packet sniffing, IP packet spoofing, Address Resolution Protocol (ARP) spoofing (MITM LAN traffic), Border Gateway Protocol (BGP) hijacking (MITM all traffic through compromised router)
        - Defenses- use HTTPS so plaintext packets cannot be sniffed, RPKI for BGP hijacking
- Transport- add features on top of bare packets
    - Examples- User Datagram Protocol (UDP), Transmission Control Protocol (TCP)
    - Attacks- UDP client impersonation (requires knowing source IP, source port and destination port)
- Application- defines how individual applications communicate
    - Examples- Domain Name System (DNS)



- Attacks- DNS hijacking (home router hacked), Hosts file hijacking (malware), DNS monitoring/blocking (ISP), Off-path DNS cache poisoning
    - Defenses- randomize UDP source port and capitlization to prevent cache poisoning

DNSSEC- sign DNS packets; provide authentication and integrity
DNS-over-TLS and DNS-over-HTTPS: encrypt DNS, provide confidentiality and integrity

Network Defenses

- Denial of Service (DoS) attacks- overwhelm network with traffic so it can't handle legitimate requests
    - Examples: radio jamming, packet flooding,
    - TCP SYN flooding
        - Defenses- avoid committing memory until TCP handshake completes or use TCP cookies to avoid SYN flooding
    - HTTP request flooding
        - Defense- rate limiting
    - Amplification (make small request with victim's IP, send big response to victim's IP)
    - Distributed Denial of Service (DDoS)- usually with botnets so it can be done without amplification
        - Defense- ingress filtering (drop packets if source IP outside a range), Content Delivery Networks (CDNs) (provide bandwidth to withstand), client puzzles (dynamically increase difficulty if server under heavy load)
- Encrypt layers; ex. Application -> SSH, Transport -> TLS, below Network -> VPN, Link -> 5G
- Don't build own network; use http2 + TLS 1.3 RPC, or REST on top of http2 + TLS 1.3
- Monitor network; Intrusion Detection Systems (IDSs) and network telescopes
- Port scanning; vertical (one host, Nmap) or horizontal (many hosts, Zmap)
- Firewalls; filter out packets

- Virtual Private Networks (VPNs);

Passwords and authentication

- Authenticate by something you know, have, or are
- Passwords are a common way to authenticate. But there are issues- specifically, people are bad at choosing strong passwords
    - Good practice for user: never re-use passwords, use 2 Factor Authentication (2FA), use a password manager, generate random passwords
    - Good practice for server: outsource sign-in, don't require too much complexity or rotation
- Attack- repeatedly guess password
    - Defense- lock account after *n* guesses, ratelimit, anomaly detection, require Completely Automated Public Turing tests to tell Computers and Humans Apart (CAPTCHA)
- To defend against password breaches, store random salt and hash(salt || password) in database instead of plaintext password
    - Don't re-use salts, and don't make short salts!
- People often forget passwords, so we need password reset mechanisms, but these are dangerous
    - Good option: use 2FA, where second factor is completely distinct from password, not just another password, and cannot be computed from password. Example: Duo, one-time password (but susceptible to phishing)
    - Another good option: use Universal 2nd Factor (U2F); performs challenge-response protocol with server (cannot be phished, but must be bought and limited support)
- Phishing attacks can steal passwords, typically done through email