① Security Principles

1) Know your threat model
2) consider human factors
3) security is economics
4) Detect if you can't prevent
5) Defense in depth, 6) least priviledge
7) Separation of responsibility
8) Ensure complete mediation (blockchain)
9) Shannon's maxim: don't depend on obscurity.
10) Use fail-safe defaults
11) Design in security from start.

② x86 Assembly and call stack

Stack layout:

- individual var: 1st at highest
- members of struct: 1st at lowest
- global var: 1st at lowest (args in func)

x86 Assembly:

opcode source, destination

xorl 4(%esi), %eax → EAX=EAX^*(ESI+4)

prologue	epilogue
pushl %ebp	movl %ebp, %esp
movl %esp, %ebp	popl %ebp
subl $4, %esp	ret

x86 Function call

caller	1. Push arguments on the stack
	2. Push old EIP (RIP) on the stack
	3. Move EIP
callee	4. Push old EBP (SFP) on the stack
	5. Move EBP
	6. Move ESP
	7. Execute the function
	8. Move ESP
	9. Pop (restore) old EBP (SFP)
	10. Pop (restore) old EIP (RIP)
caller	11. Remove arguments from stack

③ Safety Vulnerabilities

- stack smashing
- signed / unsigned integer

```
void func(int len, char *data) {
    char buf[64];
    if (len > 64)
        return;
    memcpy(buf, data, len);
}
```

This is a **signed** comparison, so `len > 64` will be false, but casting **-1** to an unsigned type yields `0xffffffff`: another buffer overflow!

`int` is a **signed** type, but `size_t` is an **unsigned** type. What happens if `len == -1`?

```
void *memcpy(void *dest, const void *src, size_t n);
```

- integer overflow

What happens if `len == 0xffffffff`?

```
void func(size_t len, char *data) {
    char *buf = malloc(len + 2);
    if (!buf)
        return;
    memcpy(buf, data, len);
    buf[len] = '\n';
    buf[len + 1] = '\0';
}
```

`len + 2 == 1`, enabling a heap overflow!

- Format strin

Specifier	Description
`%c`	Treats `args[i]` as a VALUE. Print it as a character. "%c" * ___ skip
`%__u`	Treats `args[i]` as a VALUE. Print a variable-length number of bytes starting from `args[i]` (set ___ to the desired length).
`%s`	Treats `args[i]` as a POINTER. Dereference the pointer and print the resulting value as a string. 8 bytes above RIP, print until NULL.
`%n`	Treats `args[i]` as a POINTER. Write the number of bytes that have been currently printed (as a four-byte number) to the memory address `args[i]`.
`%hn`	Treats `args[i]` as a POINTER. Write the number of bytes that have been currently printed (as a two-byte number) to the memory address `args[i]`.

%x → print in hex.

%d → print 4 bytes on stack, 8 bytes above RIP of printf.

④ Mitigating Memory Safety Vulnerabilities

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
   - Mitigation: Address-space layout randomization
3. Overwrite the RIP with the address of the shellcode
   - Mitigation: Stack canaries
   - Mitigation: Pointer authentication
4. Return from the function
5. Begin executing malicious shellcode
   - Mitigation: Non-executable pages

- **Non-executable pages**

  subversions: Return-to-libc, ROP

- **Stack canaries**

  right below SFP, if 1st byte is NULL, also prevent format string exploit

  subversions: leak, bypass, guess → remote server, timeout

  leak → format string

- **Pointer Authentication**
  - only on 64-bit systems, 22bit as PAC
  - only someone who knows CPU's master secret can generate PAC, cannot generate PAC using PAC of another address.
  - subversion: pointer reuse

- **ASLR**

subversion: leak address of a pointer, calculate relative address.

- combining mitigations → defense in depth

⑤ Intro to Cryptography

- Goals: confidentiality, integrity, authenticity
- **Kerckhoff's principle**: assume attacker knows entire system, except secret key.

	Symmetric-key	Asymmetric-key
Confidentiality	• One-time pads • Block ciphers with chaining modes (e.g. AES-CBC)	• RSA encryption • ElGamal encryption
Integrity, Authentication	• MACs (e.g. HMAC)	• Digital signatures (e.g. RSA signatures)

- Hash functions
- Pseudorandom number generators
- Public key exchange (e.g. Diffie-Hellman)
- Key management (certificates)
- Password management

36

- **IND-CPA**
  - can leak length. Eve is limited to polynomial time algorithms, must have non-negligible advantage to win.
  - Deterministic → not IND-CPA secure

- **One-Time Pads**
  - not IND-CPA secure (learns $M_0 \oplus M_1$)
  - $Enc(K, M) = K \oplus M$, $Dec(K, C) = K \oplus C$
  - not secure if key is reused.

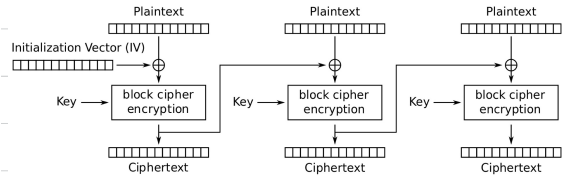
→ lack of integrity & authenticity

⑥ Block Ciphers, Modes of Operations

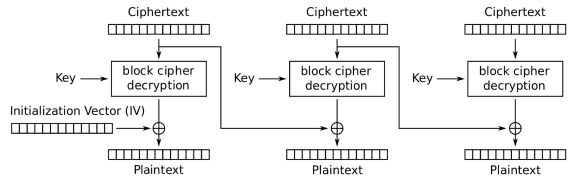
- **Block Ciphers**
  - fixed length n.
  - E_K random permutation (bijective)
  - not IND-CPA (deterministic)
- **ECB Mode** → solve fixed length
  - $Enc(K, M) = C_1 \| C_2 \| \cdots \| C$
  - not IND-CPA (deterministic)

* randomly generate IV for every encryption *

- **CBC Mode**
  - $C_0 = IV$, $C_i = E_K(M_i \oplus C_{i-1})$ ← not parallelized
  - $M_i = D_K(C_i) \oplus C_{i-1}$ ← parallelizable



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

- AES-CBC only defined for length multiple of block size → **padding** (PKCS #7: pad with the number of padding bytes) ex 3 bytes → 03 03 03
- **security**
  - IND-CPA only if IV random & never reuse, otherwise deterministic
  - **IV reuse** → reveal up to 1st diff plaintext block.

- **CTR Mode**

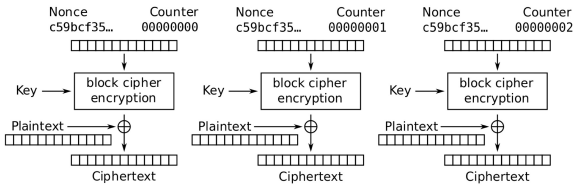
Enc • split M to $P_1 \ldots P_m$, output $(Nonce, C_1 \ldots C_m)$.

Dec • parse C to $(nonce, C_1 \ldots C_m)$. Compute P_i by XOR C_i with output of E_K, concatenate $M = P_1 \ldots P_m$.

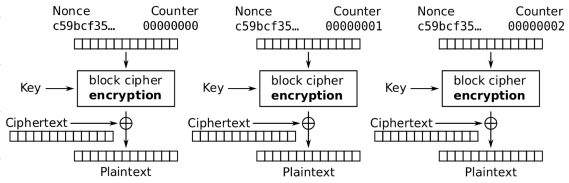
- only use block cipher enc, not dec.

* Both parallelizable

- no padding needed.



Counter (CTR) mode encryption



Counter (CTR) mode decryption

- **security**
- AES-CTR IND-CPA only if nonce random and never reused.
- **IV Reuse** → can XOR b/t two blocks of two messages on the same positions.

⑦ Cryptographic Hashes & MACs

Hash Functions

- fixed length, n bit hash. $\{0,1\}^* \to \{0,1\}^n$
- Deterministic → not IND-CPA ↗ Pigeonhole
- one-way-ness, collision resistance, unpredictable
  ↳ given y, can't find x that $H(x)=y$.
- Broke: MD5, SHA-1
- Current: SHA-2 (length-extension), SHA-3
- Have integrity if communicate hash securely. (cannot modify hash)

MACs (Message Authentication Codes)

- symmetric-key & integrity, authenticity

1) KeyGen() → K
2) MAC(K,M) → T

} send (M,T)

- correctness: determinism
- security: EU-CPA ↗ cannot generate MAC(K,M') without K, cannot find M'≠M that MAC(K,M') = MAC(K,M)

NMAC → prevent length extension (2 Hashes)

- KeyGen() → K1, K2 n bits fixed length

  NMAC(K1, K2, M) → H(K1 || H(K2 || M))

  Secure if K1, K2 different

HMAC

- HMAC(K,M) → H((K' ⊕ opad) || H((K' ⊕ ipad) || M))
  K' = version of K, arbitrary length (pad or cut)
- use K' derive opad (0x5C) and ipad (0x36)
- can't verify T if don't have K. (can't brute force M without knowing K).

- Do MACs provide integrity?
  - Yes. An attacker cannot tamper with the message without being detected
- Do MACs provide authenticity?
  - It depends on your threat model
  - If a message has a valid MAC, you can be sure it came from *someone with the secret key*, but you can't narrow it down to one person
  - If only two people have the secret key, MACs provide authenticity: it has a valid MAC, and it's not from me, so it must be from the other person
- Do MACs provide confidentiality?
  - MACs are deterministic ⇒ No IND-CPA security
  - MACs in general have no confidentiality guarantees; they can leak information about the message
    - HMAC doesn't leak information about the message, but it's still deterministic, so it's not IND-CPA secure

↗ confidentiality & integrity ✓

Authenticated Encryption

- MAC-then-Encrypt: Enc(K1, M || MAC(K2, M))
- ✓ $\boxed{\text{Enc-then-MAC}}$: MAC(K2, Enc(K1, M))
- Both IND-CPA, EU-CPA

AEAD (confd, integ, autn)

- additional Data.

⑨ Galois Counter Mode (GCM)

(DRBG)

⑧ PRNGs, Diffie-Hellman Key Exchange

- PRNG deterministic, but computationally indistinguishable from true randomness.
- Reseeding → always more entropy
- Rollback resistant (not required as secure)

HMAC-DRBG

↳ output looks random

- seed(s): K=0, V=0. K = HMAC(K, V || 0x00 || s)
  V = HMAC(K, V)
- Assume HMAC secure, HMAC-DRBG is secure, rollback resistant PRNG.
  - Intuition: If you can distinguish PRNG output from random, then you've distinguished HMAC from random
  - Intuition: If you can derive old output from the current state, then you've reversed the hash function

⑩ UUID

Stream Ciphers

- Symmetric enc, use PRNG as key to one-time pad.
  seed(K) → gen(n)
  ↓
  plaintext ⊕ → ciphertext
- encrypt multiple messages w/o key reuse
  IV → seed(K, IV) → gen(n)
  ↓
  plaintext ⊕ → ciphertext

- security: IND-CPA if PRNG secure (one-time pad IND-CPA if no key reuse)
- not secure if too much encryption → key reuse)
  AES-CTR (128) → stop after 2^{64} blocks output

↗ both need to be online (active)

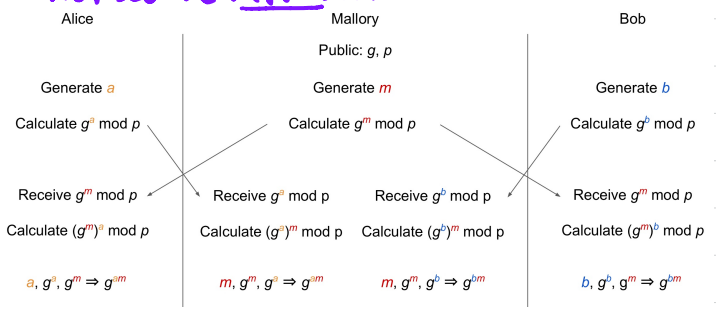
Diffie-Hellman (pub key exchange)

- Discrete log problem: given $g^a \bmod p$, comp. hard to find a.
- DH problem: given $g^a \bmod p$ and $g^b \bmod p$, comp. hard to find $g^{ab} \bmod p$. ($g^{a+b} \neq g^{ab} \bmod p$)

* DH → allow share secret over insecure channel, but can't encrypt Ms directly *

DHE (ephemerality)

- **Forward secrecy**: nobody save a, b, k, still cant decrypt a, b from $g^a \bmod p$ and $g^b \bmod p$
- **Not secure MITM attack**



- no authentication

- Elliptic-curve DH (ECDH) only sk sign, everyone verify

→ everyone can enc, only sk dec

⑨ Public Key Encryption, Digital Signatures

Nothing-Up-My-Sleeve-Numbers

- explain all choice of constants

Public Key Cryptography

- messages are numbers (not bitstring)
- Enc(PK, M), Dec(SK, C)
- **semantic security**

ElGamal Encryption → asym, confid.

- KeyGen(): $B = g^b \bmod p$ by Bob.

  Enc(g^b, M): Alice generate r, $g^r \bmod p$.

  Alice comp. $M \times (g^b)^r \bmod p$

  $\boxed{M = C_1^{-b} \cdot C_2}$ Alice send $C_1 = g^r \bmod p$, $C_2 = M \times (g^b)^r \bmod p$

  ↳ Dec(b, g^r, $M \cdot g^{br}$): Bob comp. $M \times g^{br} \times (g^r)^{-b} \bmod p$
- Eve can't derive g^{br}, so can't recover M.
- **Not IND-CPA** (send $M_0 = 0$, $M_1 \neq 0$)
- Semantically secure if $1 \le M \le p-1$, not if $M = 0$.

→ no integrity

ElGamal Malleability

- tamper: $C_1' = C_1$, $C_2' = 2 \cdot C_2 = 2 \cdot M \cdot (g^{SK})^r \Rightarrow 2 \cdot M$

$\boxed{M^{ed} \equiv M \bmod N}$

RSA Encryption → Asymmetric, confidentiality

- KeyGen(): $N = pq$ (p, q large primes)

  $e \to$ rel. prime to $(p-1)(q-1)$; $2 < e < (p-1)(q-1)$

  $d = e^{-1} \bmod (p-1)(q-1)$

  public key: N, e ; private: d
- Enc(e, N, M) → $M^e \bmod N$

  Dec(d, C) → $C^d \bmod n = (M^e)^d \bmod N$

- **RSA problem**: given n, $C = M^e \bmod N$, hard to find M

  ↳ factoring problem
- **Not IND-CPA** (deterministic, no randomness)
- Issues: side channels, send same M with diff PKs also leaks information.

  ↳ **OAEP** → RSA that has randomness
  - enc with fixed key, random input

Hybrid Encryption

- Symmetric enc → rand. gen. k encrypt data

  asymmetric enc → encrypt k
- encrypt large amounts of data quickly.

Digital Signatures → asym, integ. auth.

- only owner of SK can sign M with SK, everybody can verify signature with PK.
- Sign(SK, M) → sig → sign H(M) use hash for signing long messages.

  verify(PK, M, sig) → {0, 1}
- **Security**: EU-CPA
- integrity & auth → know M is endorsed by SK owner

RSA Signatures

- KeyGen → PK = N, e ; SK = d

  Sign(d, M) → $H(M)^d \bmod N$

  verify(e, N, M, sig) → $H(M) \equiv sig^e \bmod N$ verify
- **Factoring hardness assumption**: large N, hard to find p, q = N.
- **Discrete log hardness assumption**: large N, hash, $hash^d \bmod N$, hard to find d.

⑩ Certificates, Passwords

- Problem: Distribute Public Keys (MITM)

A	M	B
Receive PKM ←	send PKM ←	gen PKB, SKB, send PKB
send {M}PKM →	Decrept {M}PKM	
	send {M}PKB →	Decrypt {M}PKB

∴ need root of trust → trust anchor

- **trust-on-first-use**
- **certificate**: signed endorsement of some PK. contains at least: ① identity, ② key
- enc w/ PK: PK: {M}PK; sign w/ SK: SK: $\{M\}_{SK}^{-1}$

The Trust Directory (TD)

- PKTD, SKTD → $\{M\}_{SKTD}^{-1}$

  publish to everyone
- Assumptions: ① Receive TD's key correctly. ② TD won't sign w/o verify owner's identity.

- Problem: scalability, single point of failure

Hierarchical Trust multiple trust anchors

Revocation

1. Expiration Date
2. Periodically announce invalid certs.

Password Hashing

- Hash password, check if matches pwd hash in file.
- need determinism, one-way in hash
- Dictionary attack, rainbow tables

Salted Hashes

- H (pwd || salt) → unsalted DB O(M+N), salted O(MN)

Slow Hashes (PBKDF2)

- **Password-based key derivation function 2 (PBKDF2)**: A slow hash function
  - Setting: An underlying function that outputs random-looking bits (e.g. HMAC-SHA256)
  - Setting: The desired length of the output (*n*)
  - Setting: Iteration count (higher = hash is slower, lower = hash is faster)
  - Input: A password
  - Input: A salt
  - Output: A long, random-looking *n*-bit string derived from the password and salt
  - Implementation: Basically computing HMAC 10,000 times
- Benefits (assuming the user password is strong)
  - Derives an arbitrarily long string from the user's password
  - Output can be directly used as a symmetric key
  - Output can also be used to seed a PRNG or generate a public/private key pair
  - Algorithm is slow, but doesn't use a lot of memory (alternatives like Scrypt and Argon2 use more memory)

- **Offline attack**: The attacker performs all the computation themselves
  - Example: Mallory steals the password file, and then computes hashes herself to check for matches.
  - The attacker can try a huge number of passwords (e.g. use many GPUs in parallel)
  - Defenses: Salt passwords, use slow hashes
  - If an attacker can do an offline attack, you need a really strong password (e.g. 7 or more random words)
- **Online attack**: The attacker interacts with the service
  - Example: Mallory tries to log in to a website by trying every different password. Mallory is forcing the server to compute the hashes.
  - The attacker can usually only try a few times per second, with no parallelism
  - Defenses: Add a timeout to prevent the attacker from trying too many times