x86

0x FFFF FFFF

stack ↓	function frames (RIP/SFP/local vars)
↑ heap	dynamically allocated variables (malloc)
static	static variables (const. strings, static mem. buffers)
code	x86 instructions where EIP & RIP usually point

after RIP SFP

0x 0000 0000

EIP	instruction
ESP	top of the stack (lower addr)
EBP	base of current frame (higher addr)

RIP : old EIP

SFP : prev. func.'s EBP

push puts reg value on stack & moves esp lower

pop puts value @ esp into reg

④ off by one vuln: write one byte past buf into LSB of SFP so it points into buf → shlcd

⑥ ret 2 ret

RIP main
SFP main
arg 2
arg 1
RIP foo
SFP foo
buf[5:8]
buf[0:4]

little-endian: least significant byte is stored at the lowest mem

0xDEADBEEF

EF BF AD DE

1. push args onto stack (reverse)
2. push old EIP (RIP) onto stack
3. update EIP
4. push old EBP (sfp) onto stack
5. move EBP down to SFP
6. move ESP down for new frame
7. run function.
8. move ESP up to EBP
9. restore old EBP by popping SFP.
10. restore EIP pop RIP
11. remove arguments from stack by moving ESP up.

MEMORY SAFETY ATTACKS

① buffer overflow

vuln: code uses unsafe gets, read, etc instead of fgets, fread can write to any region above buf (auth bool, *fnptr injection)

② stack smashing vuln: buffer overflow to overwrite RIP to point to shellcode. when func returns, exec will jump to RIP addr.

③ integer conversion vuln: checking len < 8, passing -1 (0xFF...) and it being interpretted as an unsigned int later (2s prev.)

⑤ format string vuln:

%c ingests one char of args,	VALUE
%k u prints as unsigned int & adds whitespace before to display k total characters	VALUE
%s derefs & prints val as string	PTR
%n writes # of bytes that have been printed as a 4-byte # to the mem addr in arg	PTR
%hn same but 2 byte word	PTR
%x prints words in hex	VALUES

defenses

① stack canary

RIP foo
SFP foo
CANARY
buf[4:8]
buf[0:4]

- can still heap overflow
- local vars can still be overwritten
- if atk happens before return, canary not checked
- can be leaked
- canary checked when return

③ ASLR randomize the start of each segment of memory

- relative addresses still preserved
- if one stack addr leaked, other addresses can be determined
- can be subverted with ROP: return-oriented programming which allows you to look for useful segments of code called gadgets that can allow you to perform specific attacks

confidentiality: can't read
integrity: can't change
authenticity: can verify sender

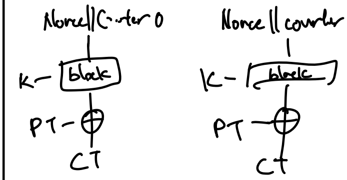
IND-CPA:
1. eve sends alice M_0, M_1
2. alice randomly encrypts & sends one
3. eve guesses 0.5

deterministic -> not IND-CPA secure

XOR
$1 \wedge 0 = 1$, $0 \wedge 0 = 1 \wedge 1 = 0$

commutative: $X \wedge Y = Y \wedge X$
associative: $X \wedge (Y \wedge Z) = (X \wedge Y) \wedge Z$
$X \wedge X = 0$
$0 \wedge Y = Y$

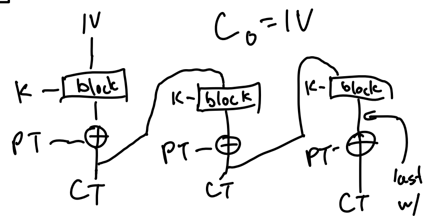
CTR (counter)
can be parallel



DEC swap PT & CT

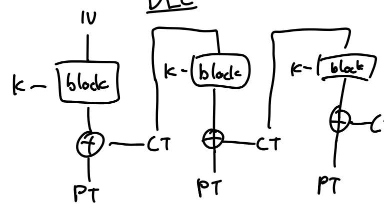
CFB mode:
$C_i = E_k(C_{i-1}) \oplus M_i$
$C_0 = IV$

ENC



last block, just XOR w/ the necessary # of bytes from encrypted ciphertext, thus no padding needed

DEC



$M_i = E_k(C_{i-1}) \oplus C_i$

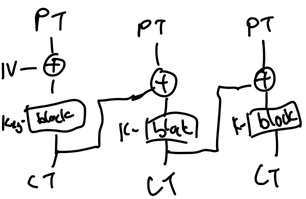
1-time pad:
gen random n-bit key
enc = dec = K ⊕

block ciphers by itself:
- not IND-CPA secure
- can't handle non-fixed size

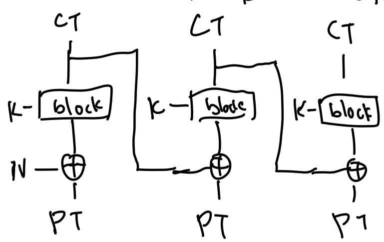
ECB mode:
encrypt block-sized chunks
still not secure

IV/nonce: random bits, public, not reuseable

CBC mode:
ENC $C_i = E_K(M_i \oplus C_{i-1})$, $C_0 = IV$



DEC $M_i = D_K(C_i) \oplus C_{i-1}$



hashes
- H(M): M is arbitrary length outputs fixed length n-bit hash
- looks "random" - fast
- one way: hard to find x given a y such that H(x)=y
- collision-resistant: hard to find $x \neq x'$ such that $H(x) = H(x')$

MAC
KeyGen() -> k
MAC(K, m): generates tag T (fixed len)

properties:
- correctness: deterministic
- eff
- security EU-CPA (atk cannot create a valid tag on M w/o k)

diffie hellman
Alice & Bob agree on large prime P and generator G, $1 < G < P-1$

Alice picks a, computes
$A = g^a \bmod p$
Bob picks $b \rightarrow B = g^b \bmod p$
announce A & B
Alice calculates $B^a = g^{ab} \bmod p$
Bob calculates $A^b = g^{ab} \bmod p$

relies on discrete log problem: given g & p & $g^a \bmod p$, can't find a

throw away ab & S when done so forward secrecy

non-exec pages
- local data manipulation still possible
- ROP may bypass non-exec pages

pointer authentication
unused bits of 64 bit system are set to authentication bits like a canary for an address

| 48 bit addr | 16 bit PAC |

- can trick CPU into gen PAC
- brute force able

public key cryptography

A & B don't need to share key

but much slower

---

el gamal

Bob announces $B = g^b \bmod p$

Alice sends $C_1 = R = g^r \bmod p$

and $C_2 = M \times B^r \bmod p$

Bob calculates $C_2 \times C_1^{-b} = M \times B^r \times R^{-b}$

$= M \times g^{br} \times g^{-br}$

$= M \bmod p$

RSA

Alice		Bob
$\boxed{M} \to$ $C = M^e \bmod N$	$\to \boxed{C} \to$	$C^d \bmod N = M^{ed} \bmod N = M$

key Gen()

- large primes p & q
- $N = pq$
- choose e that is relatively prime to $(p-1)(q-1)$
- compute $d = e^{-1} \bmod (p-1)(q-1)$ using EEA
- public key: (N, e)
- private key: d

WEB SECURITY

protocol domain port path query fragment

Uniform Resource Locators (URLs): https://www.example.com:443/index.html?query=helloworld#evanbot

Protocol: how to retrieve data over internet - HyperText Transfer Protocol (encrypted w/ TLS), File Transfer P, FILE, Git + SSH

Domain: which web server to contact; converted to an IP address using DNS; username @ domain (rare); 443/80 def
https http

Path: which resource on the web server requested; can be direct or parsed

Query: optional set of Key=value arguments; ?Key=value&Key=value

Fragment: optional, not sent to web server, tells browser to scroll there

* frame isolation: outer page cannot modify inner page, and vice versa

HTTP: request-response model; client init. connection w/ req.; 1.1 text based w/ info header & body, 2 is byte encoded
- first line: method, path, protocol version; meth = GET/POST; POST can modify server state, GET not supposed to
- POST - blank line between header & body, parameters in HTTP; GET - no body, parameters in URL

HTML: nestable <tag attr="value"> content </tag> | link: <a href=, pic: <img src=, js: <script> alert(1), embed: <iframe src= *

JavaScript: scripting lang that runs in browser; window.location = "url" moves browser's current page to "url", document.cookie returns a string w/ all cookies that have HttpOnly = F, fetch(url) executes GET, fetch(url, {method: "POST"})

Same-Origin Policy: isolates browser pages by origin (protocol + domain + port)
- Exceptions: JS runs w/ origin of page that loads it (not source page); images have origin of page they load from, diff from JS, load page only knows dimensions, frames have origin of URL where frame is retrieved from

Cookies: lets sites store info in browser; name/val pairs w/ attributes - domain, path, HttpOnly (whether to allow JS to access), Secure (whether to only send cookie when HTTPS enabled), SameSite (restricts when cookie is sent ('Strict'/'Lax'/'None'). strict/lax → only attach cookie when site making request is same site cookie was set for. Strict also checks the referrer (page too)), expires: time when cookie should be removed

Session Management: 1st time user + pwd → session cookie wristband

API + cookies: reqs = URL domain ends in cookie domain, URL path begin w/ cookie path

Cross Site Request Forgery: one site makes API call to other, relying on cookies to be attached
- defense #1: CSRF Token - bank.com has a CSRF token in its HTML when user loads page, corresponding to session
  - session token: saved as cookie val, persists across login | • CSRF token: saved in HTML, each load, one-time code
- defense #2: Referer/Origin Validation - requests can contain origin + referer (origin + page), which server checks
- defense #3: SameSite = Strict / Lax - Lax will attach for top level

Cross-Site Scripting (XSS): lets an attacker run JS within origin of trusted site; possible when query embedded directly into the webpage w/o sanitization; e.g. .com/?query=<script> alert(1) </script> OR from db
- defense #1: input sanitization - use lib to ensure text → text not HTML, ie replace < w/ <
- defense #2: content Secure policy (CSP) tell browser which resources site can load; header in server responses; only load scripts from curr url, disable eval(), no inline; can't prevent CRSF since CSP is for client-loaded scripts, CSRF for server-side

SQL Injection: SELECT FROM WHERE, UNION (combine multiple queries that have same # of cols / cols data types, DROP (remove a table from database) garbage' OR 1=1; --

- defense #1: escaping inputs - use library to detect SQL keywords like ' or ; and escape them (\', \;)
- defense #2: parameterized SQL (prepared statements) - special SQL query that compiles before user input

---

Networking

LAN/WAN: local area (eg apartment) + wide area (eg internet) networks. router connects multiple LANS

Layer 2 (ethernet): link layer (2) connects local machines on a LAN. MAC addresses uniquely identify machines in LAN

Layer 3 (IP): internetwork layer (3) connects many LANs. IP addresses uniquely identify machines globally

Layer 4 (TCP/UDP): transport layer (4) gives notion of connection between individual processes on machines, UDP is best effort transport layer protocol (no garuntee on order, less overhead), TCP is reliable, in-order, connection based

Layer 4.5 (TLS): TLS provides a secure connection (ie secure channel) between processes on machines

Layer 7 (HTTP): provides framework to build apps on top of lower level layers (eg HTTP GET/POST)

WPA (WiFi protected access): secure wireless communication protocol for in a LAN. WPA 2-PSK -> mult dev in LAN comm. sec.

DHCP (dynamic host config protocol): on Layer 2/3 enabling comm on LAN & Internet, allowing clients to get IP of self, DNS server, and router

BGP (border gateway): on layer 3, connects lots of local networks

DNS (domain name system & DNSSec): on layer 4, allows computers to resolve google.com to its IP addr using heirarchical system of name servers across internet. DNS queries are made over UDP.

FF:FF:...:FF (6 bytes) = special broadcast MAC addr all computers on network will accept

- if sending from LAN -> LAN, wrap layer-3 packet w/ layer 2 header [MAC source/dest (IP source/dest + content)]

Address Routing ARP: translates IP to MAC using layer 2 connection within LAN. If A knows B's IP addr, broadcast to everyone on LAN, what is MAC of 1.1.1.1, B replies only to A, my IP is 1.1.1.1 & MAC is xx:..., A cache IP->MAC

- ARP spoofing: Mallory races B to send her MAC after an ARP MAC addr request (as local on LAN)
- defense: switches - dev in LAN -> hub, ~~hub~~ switch w/ built in MAC cache to track IP -> MAC, expensive

DHCP: step 1: client config: Nick broadcasts I need a config, step 2: server offer, any server able to offer IP addr (any router in LAN) responds w/ your IP, subnet mask, gateway (router) IP, DNS server IP, step 3, cli. req. Nick broadcasts which config he's chosen, step 4: chosen server ack & saves nick's IP as a "used" addr (server acks)

- multiple computers can share global IP addr using NAT, gateway maps internal (source) -> remote (dest)
- DHCP spoofing: after step 1, Mallory races gateway w/ self as gateway IP so Nick sends packets through Mallory
- defense: higher-layer defense, eg TLS for end to end encryption

Mallory -> MITM

ACK = next byte expected (acknowledges received everything till then, SEQ = first byte im sending

TCP: Nick's PC, Slack

source port: 80
dest port: 1234
Seq #: A
Flags: SYN

seq #: B, s p: 80
Ack #: A+1, d p: 1234
Flags: SYN/ACK

Z -> S p: 1234
d p: 80
seq: A+1
ack #: B+1
flags: Ack

Client, Server

data, len X ->
<- data, len Y
data, len Z ->

seq	ack
A+1	B+1
B+1	A+1+X
A+1+X	B+1+Y
B+1+Y	A+1+X+Z

UDP: doesn't garuntee order or delivery, good for low-latency apps (video)

- headers: source + dest port, checksum

TCP: packets garunteed to be delivered & in correct order

- headers: source + dest port, seq + ack #, flags, check sum

processes: are assigned random 16-bit port # by OS, IP + port uniquely identify one process on one machine; local ports are arbitrary, server ports are defined & public

- TCP packet injection: spoof fake packet so recepient thinks it came from A
  - hard for off path bc guessing seq #
  - easy for on path since can see headers but need to race
  - very easy for MitM, block -> forge
- RST injection: attacker sends packet with RST flag & correct seq #, ends conn.
- defense: TLS, doesn't stop RST; use random #s

TLS: end to end encryption on top of TCP, asym encryption for integrity + confidentiality

- public keys use certificate authorities
- after handshake: ① ClientHello: contains random R_b, supported enc protocols
  ② Server Hello: random number R_s, the selected enc protocol, & server certificate (copy of servers pub key signed by certificate authority)
  ③ generate premaster secret (PS) known only to client & server
  - opt 1: RSA; client generates, sends through RSA
  - opt 2: diffie hellman: $\longleftarrow g, p, g^a \bmod p, \longrightarrow g^b \bmod p$, $PS = g^{ab} \bmod p$

  ④ derive enc/integrity keys for C/S: C_b I_b C_s I_s $ENC(C_b, M_1)$, $MAC(*, I_b)$

  ⑤ MAC (dialog, I_b) -> <- MAC (dialog, I_s) ⑥ comm w/ sym MAC + enc
- garuntees client talking to legit server, nobody tampered w/ handshake, secret sym keys unique
- DHE provides forward secrecy
- TLS protects against replay since unique nonces @ start

WiFi: wireless impl of Link Layer, same packet format + ARP, access points broadcast 'i am here' with network name (SSID), dev req join. WPA2-PSK used for sec comms

① 'nick office' SSID + 'gob ears' pwd chosen & PBKDF -> PSK derived from SSID & pwd

② computer derive PSK, AP sends nick Nonce, nick sends AP snonce

③ both derive a pairwise transport key (PTK)

④ both MIC exchange on dialogue

⑤ Group terminal key encrypted using PTK & sent to nick

GTK used for broadcast

PTK used for client <=> router

- Rogue Access Point: pretend to be AP & offer client a nonce, only works if atk knows PSK
- Offline Brute Force: guess wifi password since Nonces are unencrypted & Mac addr are public
- No forward secrecy: if atk learns A Nonce & SNonce & later pwd, can derive PTK
- defense: WPA-Enterprise: uses 1-time rand gen key by an auth server instead of PSK. user + pwd to server

DNS: got ans:

HEADER op code: Q , status: N , id: 26114 ← 16 random bits used to match requests & resp for UDP

QUES .edu

AUTH ← domains of children NS (maybe mult. per child) TTL cached

ADD NS → IP mappings

- Record spoofing / cache: return false IP-domain mapping, DNS is insecure for any on path attacker; off-path have to guess ID field when DNS returns $1/2^{16}$
- Kaminsky Attack: Query many non-existant domains to try to beat the race condition from the NS and poison the cache; nonexistant → when legit response arrives, nothing cached so no wait
- Bailiwick checking: only accept NS records in the NS's zone, doesn't stop Kaminsky
- UDP port randomization: 16-bit source port must be guessed; total = $1/2^{32}$
- DNSSec: can't have key for everyone → Digital Signatures / asymmetric crypto + trust delegations & anchors | NS gives domain+IP for .edu NS + a signature on that NS's public key & my public key
  - RRSIG (resource rec): encode signs on records, DNSKEY: encode pub keys, DS (delegated signer) encode child's public key (for trust). Hash of signer's name & c's pk
  - issue: Non existent domains: signing is slow so NSs pre compute signs on nonex doms
  - NSEC(3): two adj domains → middle domain doesn't exist
  - signing online whenever smth changes is slow: KSK - key signing keys KSK of root = hardcode ZSK - zone signing keys; parents sign children's ZSK

Intrusion Detection

Network Intrusion Detection System (NIDS): between router & internal network

- all requests to & from outside must pass through NIDS -ex: cheap + realtime
- Pros: single NIDS covers entire network
- cons: can't categorize packets (may be out of order / seperated), TCP reconstruction @ NIDS could diff from PC's (eg NIDS escapes, host doesn't), TLS encryption → can't analyze
- examples: all incoming unencrypted traffic to catch real time attacks on multi computer sys

Host based Intrusion Det (HIDS) • installed on end hosts (PCs)

- pros: directly check data & how data is parsed, can decrypt data too
- cons: very expensive, susceptible to path traversal still
- ex: obfuscation attack, need parse incoming data (enc), must be real time | insider threats in netw

logging: track requests made, files accessed, apps run

- pros: HIDS's pros + cheap • ex: no realtime, path traversal, offline, cheap
- cons: not real time

Detection Techniques:

- Anomaly Based: develop model of what normal activity looks like & flag anomalies
  - pros: can catch never before seen attacks
  - cons: need data for model, new secure activity -> flag -> many false +s
  - ex: novel, no time/resources
- Specification: same as ^ but manually define normal
  - pros: same + low false pos if well defined
  - cons: time consuming
  - ex: lot of resources, novel
- Behavioral detection: Look for evidence of compromise/result of exploit
  - pros: low FP rate, can catch new attacks
  - cons: after the fact, if known, attacker can change behavior while executing attack
  - ex: DoS -> honeypot server detection
- Signature based: look for activity that matches struct of known attks; blacklisting
  - pros: reliable for known attacks
  - cons: bad for new + variants
  - ex: buffer overflow w/ no evasion

Denial of Service: exploit prog flaws (buff overflow), resource exhaustion, (on bottleneck)

Anonymity: Tor - overlay network above internet; no individual node knows who you are
- attack: malicious nodes colluding | defend w 3 nodes
- correlation attack: brain use | defense, more nodes/users
- availability: easy to block Tor con w/ RST injection | defense: use obfuscate nodes that look like normal TLS traffic