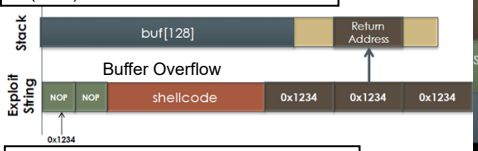
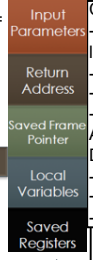
%esi = register value
(%esi) = pointer, dereference address %esi
0x8(%esi) = 0x8 + dereference value

Stack Bottom = High Mem





Common Vulnerabilities:
- Return into libc attacks (use code in libc like system(), inject on stack)
- Function pointer overwrite
- PLT/GOT overwrite
- Integer overflow, Bad bounds check, Argument overwrite

Defense:
- Stackguard stack smashing defense
- Address space layout randomization
- Non executable pages

Attack without overwriting return address:
- Function Pointer overwrite (closer than return address)
- Global offset table overwrite, dynamic link libraries
- libraries normally linked into the program at run time, at arbitrary locations, by dynamic linker
- need to map position independent function call to absolute location of function's code in the library (dynamic linker)
- used Procedure Linkage Table (PLT) and Global Offset Table (GOT)

GOT: table of pointers to functions. Contains absolute memory location of each dynamically loaded library function

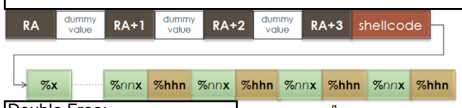
PLT: table of code entries. used to jump to locations in GOT
-Each code entry invokes corresponding function pointer in GOT
Ex. sprintf@PLT code may invoke " jmp GOT[k] where k corresponds to the sprint function index in GOT
-All calls to dynamic libraries jump to PLT
- first time function called, runtime linker invoked to load library
- runtime linker updates GOT entry, based on where library is loaded
- PLT/GOT contain commonly used library functions such as printf, fopen, fclose
-PLT/GOT always appear at a known location

Serialization: Process of translating objects into a format that can be stored or transmitted over a network

Deserialize: Reconstructs the object in memory. Attacker inject or man-in-the-middle to pass their serialized data which deserializes to much larger (expands) and cause a buffer overflow or overwrite some values

Format String Buffer Overflow:
sprintf(buf, "WARNING: %s", attacker_string);
Or printf(buf, values)

Format String Attack: snprintf(buf, len, attacker_string);
- info leak and format string
- front put address where think return address stored on stack
- Put enough "%" args so argument pointer points to front of format string
- Put %n at end to overwrite return address to point to shellcode in buffer



Return oriented programming (ROP):
- Place return address on stack pointing to gadgets (small instructions ending with returns located within memory)
- update the return address to point to one gadget, then code return will call next gadget
- Overflow the stack, placing a sequence of return addresses that correspond to the sequence of gadgets

info frame
p /x buf
p &buf
x/40x $esp or x/20x 0x2021fea8
frame <frame #>
x/s <addr> | x/20s & <var> or addrs // print strings
x/c | x/20c <addr> // for characters

Clock Glitching:
-introduce brief rapid clock pulse. Not enough time for instruction fetch/decode.
- Glitch duration:
Longer than the time to increment the Program Counter and shorter than the instruction fetch time
-Then can get instruction skipping or corruption
That is, program counter increments, but instruction is not fetched, so previous instruction gets executed.
This can then let us skip instructions, like when logins are checked (skip over return/jump)

Prevent return address overwriting:

Stackshield: Put return addresses on a separate stack with no other data buffers there

Stackguard: random canary value is placed just before the return address. Just before the function returns, the code checks the canary value and, if the value has changed, the program is halted

Address Space Layout Randomization (ASLR):
- OS maps stack of each process at randomly selected location with each invocation
- attacker will not be able to easily guess the target address
- ASLR also randomizes location of dynamically loaded libraries, making it harder to perform return into libc attacks or GOT overwrites

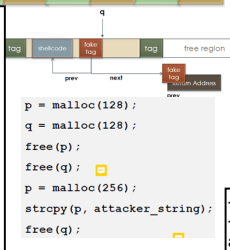
Non Executable Pages (NX):
- If stack is made non executable, then shellcode on the stack will not execute
- Page tables have NX protection bit. Requires support from OS

Self-signed X.509:
A certificate authority signs it's own X509 certificate so that when users pass their signed certificate by the CA to another, the recipient can validate the public key certificate of the CA that is to be used to decrypt the certificate of the sender.

Double Free:
-free(q) will try to use the chunk tag located just before the address pointed to by q, which is part of attacker_string

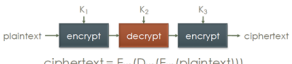
tag = q - sizeof(chunkTag);
tag->next->prev = tag->prev;
tag->prev->next = tag->next;

tag->next = return addr
tag->prev = shellcode



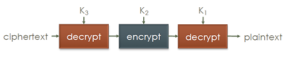

```
p = malloc(128);
q = malloc(128);
free(p);
free(q);
p = malloc(256);
strcpy(p, attacker_string);
free(q);
```

- bootloader like bios, controls startup of application
- powerup all the circuitry and then hand off to application
- processor go into provisioning mode, some random encryption keys
- processor blows fuses, so can't extract these keys unless you mechanically disassemble the processor
- Secure Hash (bootloader, key) = Digest
- everytime it boots, checks digest, recalculates hash and cmps digest match



ciphertext = $E_{K3}(D_{K2}(E_{K1}(\text{plaintext})))$

3DES



plaintext = $D_{K1}(E_{K2}(D_{K3}(\text{ciphertext})))$

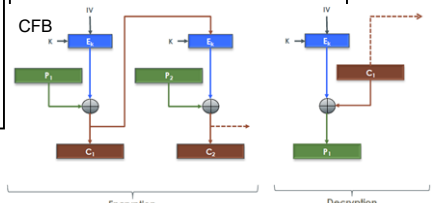
Stream Cipher:
- In general stream ciphers have better performance than block ciphers
- less rounds
- for synchrouns we can pre-compute the key streams
- Realtime, low latency, variable length mesgs
- As they operate a bit/byte at a time
- Don't have diffusion
- Like OTP
- Keystream: pseudo random sequence of bits generated from a much shorter key
- The stream of random bits is then used in place of the one time pad and XORed with the plain text
- Unlike block ciphers that encrypt a block at a time and then use different modes to improve security, stream ciphers have no modes because they can be used to encrypt arbitrarily long messages
- Are malleable
- synchronous stream ciphers, if a section of cipher text is lost, the ciphertext stream and keystream become out of sync and recovery is impossible unless we know exactly how much ciphertext is lost
- Self synchronizing stream ciphers will recover after n bits have passed
- Key scheduling algorithm initializes state S -> f()
- Pseudo Random Generation Algo, generates keystream => g()

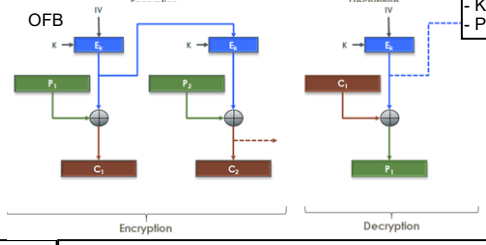
MDC vs HMAC?
- MDC uses hashes to provide integrity, by taking hash of a mesg and sending hash and mesg separately. Receiver can use it to detect if mesg was modified in transit. But the MDC value needs to be sent on a secure channel or encrypted
- MAC provides integrity and authentication as a key is used with mesg when generating hash value. Receiver knows who generated MAC must have known the key and therefore authenticating the source.

Cryptography Achieves:
- Confidentiality: data secrecy, provided by ciphers
- Integrity: trustworthiness of data, provided by hashes
- Authenticity: allows principle to prove identity or origin of data, provided by signatures message authentication codes (MACs)
- Non-Repudiation: prevents principle from denying they performed an action, achieved by trusted third party

Kerckhoff's Principle: The security of any given encryption system must depend only on the secrecy of the key and not the algorithm

CFB (Cipher Feedback) mode and OFB (Output Feedback) mode allow encryption and decryption in units of less than a full block at a time (i.e., they convert block ciphers into stream ciphers)
-slightly more efficient because no padding necessary
- CFB, pipelining is possible



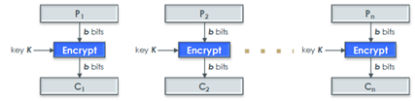


Stream vs Block:
-Stream ciphers are generally simple and very fast

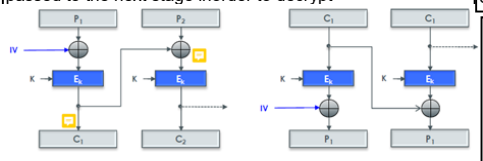
Block ciphers:
Confusion
- Obscures relationship between plaintext and ciphertext
- Makes statistical analysis difficult
- Non-linear => E_k(M1 + M2) != E_k(M1) + E_k(M2)
- Each character affected by entire key
- Confusion performs substitution and adds non-linearity. However, still allows attacker to know that affecting bit in ciphertext will affect same portion in plaintext.

Diffusion:
- Spreading the influence of individual plaintext characters over ciphertext
- hides patterns
- many input bit -> to output bit
- repetitive patterns in plaintext spread over ciphertext, thus hides statistical info about plaintext
- each plaintext character adds a bit to each ciphertext character, helps hide patterns
- Diffusion hides this issue by spreading affect of bit (hides substitution mapping)

ECB (Electronic codebook): simple mode
- message is broken into block-sized chunks
- Each chunk is encrypted independently
- Highly-parallelizable
- Poor security; adversary can add, delete or reorder blocks
- Frequency analysis as plaintext blocks encrypt to same ciphertext
- retransmit affected blocks



CBC (Cipher block chaining):
- hide patterns
- every blocks' input dependent on the cipher text of previous block
- IV does not have to be secret, but shouldn't be reused and random, so that plaintext doesn't encrypt to same
- Good security
- modification to a ciphertext block affects at most two blocks during decryption
- No parallelism for encryption (sequential)
- Decryption can be parallelized
- If you wanted to start in the middle of a video file, then you can start a bit before the middle due to the cyphertext passed to the next stage inorder to decrypt



Problem with MAC:
everyone knows the hash function used
- problem is if an attacker wants to extend the msg, then they can do that and just extend the hash calculation to calculate their own MAC from the extend version of the mesg

- good for parallelization because there's no dependency on the plaintext or cryptotext
- sender can pre-compute the blue part (encryption) chain. And the reciever can do the same at the same time
- quickly compute the xors, maybe even in parallel as the blue part is still being processed on reciever side, reciver can still XOR incoming data
- OFB, the key stream is independent of the plaintext: allows performing the cipher operations in advance (speeding up encryption and decryption), and easily supports error correction coding
- OFB mode good for fast decoding while hiding patterns with chaining, as it handles multiple blocks and can be parallelized for decryption, all state can be preprocessed and when mesg comes in it can just XOR with ciphertext.

IV rand generated value for first block/byte of encryption. should be different (unique) between messages to prevent frequency attacks and ensure the same mesg doesn't encrypt to the same ciphertext

-For block cipher: repeated IV values devolve encryption scheme, ECB pattern can be figured out.
- Stream ciphers encrypt plaintext to ciphertext by deriving a key stream from a given key and IV and computing C = P XOR K. Assume attacker observed messages C1 and C2 both encrypted with the same key and IV. Then knowledge of either P1 or P2 reveals the other plaintext since C1 XOR C2 = (P1 XOR K) XOR (P2 XOR K) = P1 XOR P2

Each f(Rn-1,Kn) contains
- Expansion permutation of 32-bit input (Rn-1) into 48 bits
- XOR with a 48-bit subkey (Kn)
- S-box substitution boxes that compress 48-bits into 32-bit output
- Only non-linear part
- Permutation of 32-bit output

HMAC = H[(K ⊕ opad) + H((K ⊕ ipad) + M)]

HMAC = H(key$_1$ + H(key$_2$ + message))

- The inner and outer padding are chosen to minimize number of common bits in key$_1$ and key$_2$

Cesur Cipher -> Shift Cipher -> Substitution Cipher
- Each plaintext letter is replaced with exactly one ciphertext letter
- The key = mapping between plaintext and ciphertext, lookup table
- Weak, easy to break. Static lookup, plaintext gets encrypted to the same ciphertext
- Do not hide frequency info as result
- Improve with polyalphabetic cipher

Polyalphabetic/period cipher:
- Instead of one mapping, have set of n mappings (n lookup), change mapping with every character (use a different lookup table per each character)
- For small number of mappings, only slightly hard to break than substitution cipher
- For large number of mappings, much more difficult
- Ex. enigma machine

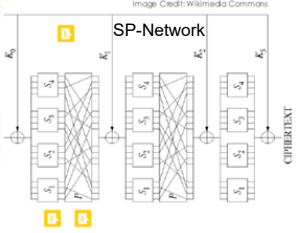
One-Time-Pad (OTP)/Vernam cipher:
- Special type of polyalphabetic cipher that never repeats
- A random substitution is used for every character (ie. infinite number of lookup tables)
- Key same length as msg; high overhead
- C = K XOR P
- plaintext bit is flipped when key bit is 1, same when 0
- Theoretically unbreakable
- Each key can be used only once
- Key needs to be secured and random
- Synchronization problem if messages are lost or reordered
- Is malleable, requires combining with integrity check to avoid tampering
- If only know ciphertext, secure
- If know plaintext, then CT XOR PT = key (hence, key one time)
- Only provide confidentiality, not integrity as no hash

Block/Stream Cipher:
- fixed length key < message
- Symmetric key ciphers use the same key

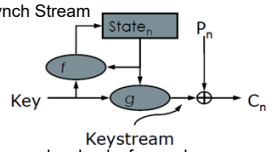
Stream ciphers
- Similar to OTP
- key is used to generate a pseudo random sequence of bits
- bits are then XORed with the plaintext, to encrypt a bit at a time
- hence useful for streaming applications
- suffer from synchronization problems, if any bits are lost, the entire stream may be corrupted

HMAC:
- first bit of the mesg creates the inner MAC
- then the inner MAC is put through a seperate calculation to calculate the HMAC
- know if an attacker wants to extend the msg their stuck, in order for them to calculate the correct value for the extension, they need to extend the calculation of the inner value, but they don't have the inner value anymore, all they have is the outer value/HMAC

- Hence, use HMAC to prevent attackers from extending our msg, and looks like a valid signature at the end of our mesg



Synch Stream

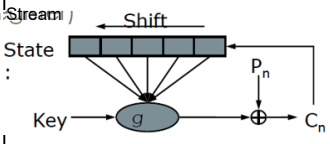


some drawbacks for random access, due to state
- ex. hard drive, have to decrypt from beginning till point needed
- efficient for short/traditional messages and not for large encrypted volumns

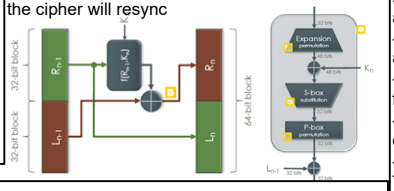
If just H(K + M), insecure
- may allow an adversary to add arbitrary information at the end of the message and compute a new, forged MAC

- each round, input is split into a left and right half
- two halves are switched, and some computation modifies half of the input bits
- Each round includes computation with a portion of the key (a subkey Kn)
- output of one round becomes the input for the next
- 56-bit key is put through a key schedule to create 16 subkeys (Kn), one for each round
- 56-bit key is split into two 28-bit halves
- Each half is shifted left by 1 or 2 bits
- 24-bits are selected from each of the 28-bit halves to make a 48-bit subkey (Kn)

Self-Synch Stream



Random access due to shift register
Ie. tries to solve the random access problems for stream ciphers
- For IV can insert random data at beginning
- With self synchronizing ciphers, an adversary ca replay previously sent ciphertext into a stream, an the cipher will resync



TOTP:
- Uses block cipher SHA1
- To handle clock skew, divides time by window
- Odds (%) of bad TOTP =[clock skew in seconds] * (100/[window length in seconds])
- When time drift is greater than window length, generated TOTP will be invalid 100% the time.
- HOTP = Truncate(HMAC-SHA-1(K,C))
- TOTP = HOTP(K, T), T=(T1-T0) / Time Window
- Server has key and passes it to user with QRCode, both calculate TOTP value, if match then login successful
- How to handle skew: time sync software check specific number of time intervals within a specific time window before the TOTPs are rejected and access is denied
- RECOMMEND validator be set with a specific limit to the number of time steps a prover can be "out of synch"
- validator could perform a validation against the current time and then two further validations for each backward step (for a total of 3 validations)
- Upon successful validation, the validation server can record the detected clock drift for the token in terms of the number of time steps.
- When a new TOTP is received after this step, the validator can validate the TOTP with the current timestamp adjusted with the recorded number of time-step clock drifts for the token

Block cipher made up by:
- iterated block cipher repeatedly applies these two ciphers in different combinations
Substitution cipher:
- replaces characters in the plaintext with other characters (mapping)
- confusion
Permutation cipher:
- transposes the plaintext characters
- diffusion

Key Exchange:
- deals with establishing a shared secret across an insecure channel
- For live conversation with a third party
- population of N people needs a total of N(N-1) ÷ 2 keys

Trusted Third Party:
- central key server delegates keys to everyone
- server T is trusted by every user
- Every user has a unique secret key
- The server knows users secret key
- Protocol:
A -> T : { A, B }
T -> A : { K_AB }K_A, { K_AB }K_B
A -> B : { K_AB }K_B
-Server generate random session key K_AB
Problems:
- B does not know with whom he's communicating
- third-party attacker could capture the {K_AB}K_B message, along with any subsequent messages and replay them later
- make B repeat a previous action
- B doesn't know who communicating with and if live conversation

Diffie-Hellman and Public Key Cryptography (RSA) allow establishing secure communications without a trusted server
- protocol for live key exchange, without trusted server
-for short mesg as very computational expensive
- establish a common (short) secret over an insecure link
- Not very efficient for long messages

Protocol:
- Alice selects n=(a large prime modulus), select number g=(a generator of the field n that lies between between 1 and (n-1))
g^x mod n = y
To establish secret key (can occur publicly)
- Alice selects a random integer x and computes: P = g^x mod n
- Alice sends P, g and n to Bob; x is kept secret
- Bob selects a random integer y and computes: Q = g^y mod n;
- Bob sends Q back to Alice; y is kept secret
- Alice computes Q^x mod n = g^xy mod n
- Bob also computes P^y mod n = g^xy mod n
- both share the secret g^xy mod n

Problems:
- diffe-hellman does not prevent man-in-the-middle attack, can't tell communicating with (no authentication)
- Man-in-the-middle can pretend to be the other person in the conversation and pass through the messages, and generate it's own shared key with each party

A->C {P,g,n}, x -> g^xy
C->A {y} -> g^xy
C->B {P, g, n}, g^mn
B->C {n}, g^mn
Hence, C has g^xy and g^mn

Public Key Certificates
-to ensure no man-in-the-middle attack, where attacker passes their key to be used
- to solve this problem, third party everyone trusts to certify peoples public key certificate
- PKI is a system where a trusted third party (a principal that everyone trusts) vouches for the identity of a key (i.e. that the key belongs to a principle)
Protocol:
- that Alice and Bob trust Trent
- that everyone knows Trent's public key
- Bob creates a public key and goes to Trent; Trent sees both Bob and his public key, creates a certificate that says This public key xxx belongs to Bob and signs it with his (Trent's) private key
- Bob sends Alice his own public key along with the certificate that bears Trent's signature
- Alice uses Trent's public key and the certificate to verify Bob's public key
- Helps prevent attacker pretending their key is someone elses
- Common standard format for certificates is X509
- PKI allows using a chain of certificates issued by a hierarchy of Cas
- diff from trusted server prev seen
- doesn't require third party server to be online all the time, all were getting from them is a signature you can walk away with (only go back when certificate expires)
- no longer a single point of failure on third party server
- it changes the trust level, were not giving the third party server my private key, their only seeing my public key and signing it, so they do not have any ability to decode the conversation taking place
- Trust level, availability, integrity
- When a browser connects to a secure website, the website sends the browser a certificate that you can verify by viewing the certificate.

Needham-Schroeder Protocol:
A -> T : { A, B, N_A }
- A picks a nonce (random number)
T -> A : { N_A, K_AB, B, {K_AB, A}K_B }K_A
- Nonce in reply from server assures A that this is not a replay
- Including B ensures no man-in-the-middle, who A wants to talk to
- Everything is encrypted with A's key, so safe
A -> B : { K_AB, A }K_B
- B will know who they are talking to as A is in it
B -> A : { N_B }K_AB
- Send a new nonce to allow B to make sure this is not a replay, and that their actually talking with A as they share K_AB
A -> B : { (N_B – 1) }K_AB
Problems:
- Server is trusted, and all communication goes through server, single-point-of-failure or bottleneck, and security concern if attacker gains access to server as all keys (session and user) are stored

Public-Key Cryptosystems:
To encrypt:
-sender encrypts the message with the intended recipient's public key
-Only the recipient should have the private key, so only the recipient can decrypt the message
- To provide authentication/signing/non-repudiation:
- Sender encrypts with their private key
- Anyone can decrypt with senders public key, proves sender was one who created mesg
Protocol:
- Alice randomly selects a key x and encrypts it with Bob's public key
- Bob receives the encrypted key and decrypts it with his private key
- Both Alice and Bob now share the same key x
- Like RSA or DSA

RSA:
-RSA is a public key algo
-Based on modular arithmetic
-RSA key generation starts by randomly picking two very large prime numbers: p and q
-n = p*q, n can be public, but p and q are private
- The size of n defines the key size: 4096bit
- we can publicly share the value n , because there is no known algorithm to efficiently factor it and recover p and q
-Euler's Theorem: phi = (p-1)(q-1)
- Pick a random value, e , that is coprime to phi.
- e will be our public key
- Someone can use public key (e) to encrypt plaintext (M), into a cyphertext C
C = M^e mod n
e = public key, n, C can be made public
p, q, phi must remain secret
- to decrypt need private key (d), which is a multiplicative inverse of public key (e*d) = 1 mod phi
M = C^d mod n => sub C = M^e mod n, and get M = M^ed mod n
- (e*d =1) mod phi
Problems:
- RSA has very poor resistance to spoofing because encryption uses exponentiation
- If someone signs messages adversary gives them, then can trick them into signing messages they have never seen

Three party diffe-hellman:
1) A,B,C generate random values: a, b, c
2) A,B,C calculate g^a, g^b, g^c
3) A->B: {g^a}, B->C: {g^b}, C->A: {g^c}
4) A: g^ac, B: g^ab, C: g^bc
5) A->B: {g^ac}, B->C: {g^ab}, C->A: {g^bc}
6) A: g^abc, B: g^abc, C: g^abc
Shared Key: g^abc

Modes hide patterns, protect against frequency/pattern attacks. Secure cross blocks and chaining prevent reordering and similar block encrypt to ciphertext