x86 Assembly / Call Stack:

eip is the instruction pointer, and it stores the address of the machine instruction currently being executed. In RISC-V, this register is called the PC (program counter).

ebp is the base pointer, and it stores the address of the top of the current stack frame. In RISC systems, this register is called the FP (frame pointer)

esp is the stack pointer, and it stores the address of the bottom of the current stack frame. In RISC-V, this register is called the SP (stack pointer).

x86 Function Calls:

1. Push arguments onto the stack. RISC-V passes arguments by storing them in registers, but x86 passes arguments by pushing them onto the stack. Note that esp is decremented as we push arguments onto the stack. Arguments are pushed onto the stack in reverse order.
2. Push the old eip (rip) on the stack. We are about to change the value in the eip register, so we need to save its current value on the stack before we overwrite it with a new value. When we push this value on the stack, it is called the old eip or the rip (return instruction pointer).
3. Move eip. Now that we've saved the old value of eip, we can safely change eip to point to the instructions for the callee function.
4. Push the old ebp (sfp) on the stack. We are about to change the value in the ebp register, so we need to save its current value on the stack before we overwrite it with a new value. When we push this value on the stack, it is called the old ebp or the sfp (saved frame pointer). Note that esp has been decremented because we pushed a new value on the stack.
5. Move ebp down. Now that we've saved the old value of ebp, we can safely change ebp to point to the top of the new stack frame. The top of the new stack frame is where esp is currently pointing, since we are about to allocate new space below esp for the new stack frame.
6. Move esp down. Now we can allocate new space for the new stack frame by decrementing esp. The compiler looks at the complexity of the function to determine how far esp should be decremented. For example, a function with only a few local variables doesn't require too much space on the stack, so esp will only be decremented by a few bytes. On the other hand, if a function declares a large array as a local variable, esp will need to be decremented by a lot to fit the array on the stack.
7. Execute the function. Local variables and any other necessary data can now be saved in the new stack frame. Additionally, since ebp is always pointing at the top of the stack frame, we can use it as a point of reference to find other variables on the stack. For example, the arguments will be located starting at the address stored in ebp, plus 8.
8. Move esp up. Once the function is ready to return, we increment esp to point to the top of the stack frame (ebp). This effectively erases the stack frame, since the stack frame is now located below esp. (Anything on the stack below esp is undefined.)
9. Restore the old ebp (sfp). The next value on the stack is the sfp, the old value of ebp before we started executing the function. We pop the sfp off the stack and store it back into the ebp register. This returns ebp to its old value before the function was called.
10. Restore the old eip (rip). The next value on the stack is the rip, the old value of eip before we started executing the function. We pop the rip off the stack and store it back into the eip register. This returns eip to its old value before the function was called.
11. Remove arguments from the stack. Since the function call is over, we don't need to store the arguments anymore. We can remove them by incrementing esp (recall that anything on the stack below esp is undefined).

C / Endianess: Code section: Machine code (raw bits) to be executed. Static section: static variable (variables declared outside of functions) Heap section: dynamically allocated memory (e.g. from malloc) Stack Section: local variables and stack frames

&buf2 3 2 1 0 7 6 5 4 11 10 9 8 15 14 13 12

0xffff1430: 0x00003500 0x6ca59820 0x0000abcd 0x74392bc2

Q2.4 TRUE or FALSE: `buf[5]` is `0x98`.

● (A) TRUE ○ (B) FALSE

Solution: True. `0x00003500` is the word that contains `buf[0]`, `buf[1]`, `buf[2]`, and `buf[3]`.

Then, in the word `0x6ca59820`, since the system is little-endian, `buf[4]` is the LSB of this word, `0x20`. Then, `buf[5]` is the second-least-significant byte of this word, `0x98`.

Memory Safety Vulnerabilities:

Buffer overflow vulnerabilities: Buffer overflow vulnerabilities arise from dangerous gets( ) calls allowing an attacker to overwrite any data located at addresses higher than the location of the buffer. Common exploits using buffer overflows involve overwriting the buffer, data between the buffer and the SFP of the function, and then overwriting the RIP of the function with shellcode (called stack smashing).

char* gets(char* str); Reads stdin into the char array pointed to by str until a newline character is found or EOF occurs. A null character is written immediately after the last character read into the array. The newline character is discarded but not stored in the buffer. **Doesn't care about the size of the buffer at str**

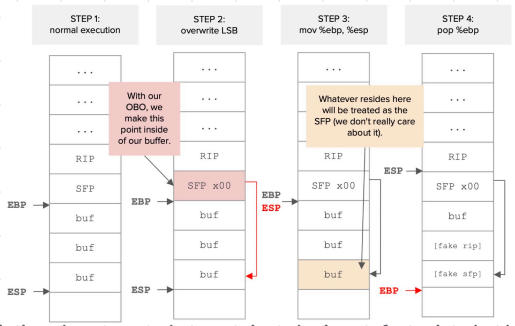
strcpy

strlen

Format string vulnerabilities: These type of vulnerabilities arise when the number of string format specifiers in a printf call are mismatched with the number of variables input as arguments into buf. Such a printf statement allows us to write to any location on the stack. Given a vulnerable printf call such as printf(buf), we can retrieve any value off the stack above the printf call. **%s** -> treat the argument as an address and print the string at that address up until the first null byte **%n** -> Treat the argument as an address and write the number of characters that have been printed so far to that address **%hn** -> Treat the argument as an address and write the number of character that have been printed so far into the lower half of that address **%c** -> Treat the argument as a value and print it out as a character **%x** -> Look at the stack and read the first variable after the format string **%[b]u** -> print out [b] bytes starting from the argument

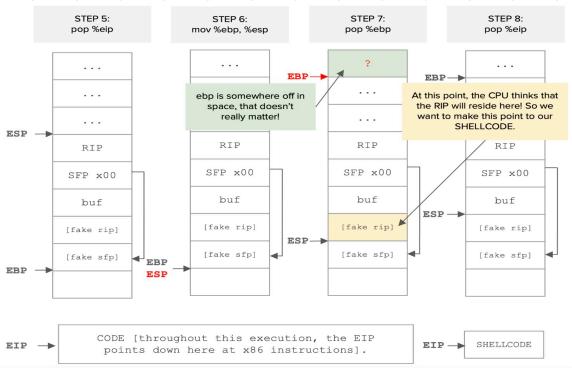
Integer conversion vulnerabilities: When converting a signed integer to an unsigned integer, this cause the potential for overflows. For example, a lot of C functions take in unsigned integer as opposed to signed integer. If the user inputs a negative number instead of a positive, the number will be casted to a positive number and, as a result, become very large, causing behavior that maybe advantageous to a malicious user. Commonly, these integer conversion vulnerabilities are used to bypass checks on read size or write size to read/write into uninitialized memory.

Off-by-one vulnerabilities: Off-by-one vulnerabilities are very common in programming (i.e. using <= rather than <, reading one extra byte on accident, starting the for loop counter at the wrong value). Consider a buffer whose bounds checks are off by one. This means that we write n + 1 bytes rather than n bytes. We can exploit such a trivial mistake as follows.



Step 1: This is what normal execution during a function looks like. Consider reviewing the x86 section of the notes if you'd like a refresher. The stack has the rip (saved eip), sfp (saved ebp), and the local variable `buff`. The esp register points to the bottom of the stack. The ebp register points to the sfp at the top of the stack. The sfp (saved ebp) points to the ebp of the previous function, which is higher up in memory. The rip (saved eip) points to somewhere in the code section. **Step 2**: We overwrite all of `buff`, plus the byte immediately after `buff`, which is the least significant byte of the sfp directly above `buff`. (Remember that x86 is little-endian, so the least significant byte is stored at the lowest address in memory. For example, if the sfp is `0x12345678`, we'd be overwriting the byte `0x78`.) We can change the last byte of sfp so that the sfp points to somewhere inside `buff`. The SFP label becomes FSP here to indicate that it is now a forged sfp with the last byte changed. **Step 3**: `mov %ebp, %esp`: esp now points where ebp is pointing, which is the forged sfp. **Step 4**: `pop %ebp`: Take the next value on the stack, the forged sfp, and place it in the ebp register. Now ebp is pointing inside the buffer. **Step 5**: `pop %eip`: Take the next value on the stack, the rip, and place it in the eip register. Since we didn't maliciously change the rip, the old eip is correctly restored.

Memory Safety Vulnerabilities (Cont.):



Step 6: `mov %ebp, %esp`: esp now points where ebp is pointing, which is inside the buffer. At this point in normal execution, both ebp and esp think that they are pointing at the sfp. **Step 7**: `pop %ebp`: Take the next value on the stack (which the program thinks is the sfp, but is actually some attacker-controlled value inside the buffer), and place it in the ebp register. The question mark here says that even though the attacker controls what gets placed in the ebp register, we don't care what the value actually is. **Step 8**: `pop %eip`: Take the next value on the stack (which the program thinks is the rip, but is actually some attacker-controlled value inside the buffer), and place it in the eip register. This is where you place the address of shellcode, since you control the values in `buff`, and the program is taking an address from `buff` and jumping there to execute instructions.

Memory Safety Vulnerability Mitigations:

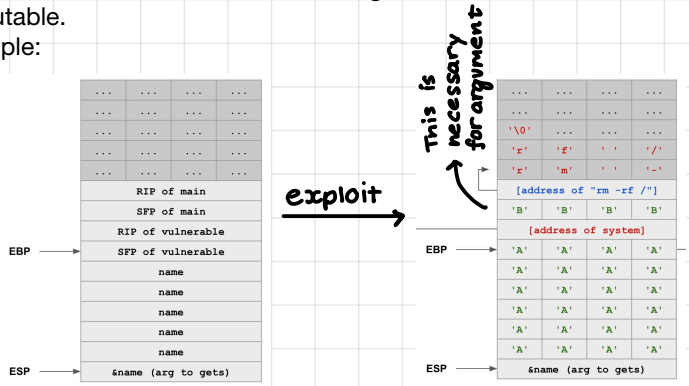
Just use a memory safe language lmao

Write memory-safe code: Use fgets instead of gets, strncpy or strlcpy instead of strcpy, and snprintf instead for printf.

Mitigations: Non-executable pages: If we make some pages of memory impossible to execute instructions on, we should be able to prevent many naive buffer overflow exploits such as stack smashing.

Subversion - Return to libc: Non-executable pages don't stop an attacker from executing code in memory. An attacker can exploit this by overwriting the rip with the address of a C library function. Ex: the execv function lets the attacker start executing the instructions of some other executable.

Example:



Subversion - Return-oriented programming: Essentially, we are creating a chain of return addresses starting at the RIP in order to execute a series of ROP gadgets which are equivalent to the desired malicious code. With ROP, you can execute your won code by simply executing different pieces of different code.

Mitigation: Stack canaries: Add a sacrificial value on the stack, and check if it has been changed. We place this value right below the SFP/RIP. Then in the function epilogue, check the value on the stack and compare it against the value in canary storage. This value is never used so if it changed, someone is probably attacking the system. This canary value is unique every time the program runs but the same for all function within a run. A canary value uses a NULL byte as the first byte to mitigate string-based attacks (i.e. printf vulnerabilities). The cost to add the stack canary is insignificant. Additionally, any vulnerability that leaks stack memory could be used to leak the canary's value. Furthermore, stack canaries stop attacks that write to increasing, consecutive addresses on the stack and canaries do not stop attacks that write to memory in other ways (i.e. writing around the canary using printf vulnerability, heap overflows). Or you could brute force the canary.

Mitigation: Pointer Authentication: Instead of placing the secret value below the pointer, store a value in the pointer itself. Replace the unused bits in an address with a pointer authentication code. Before we use a pointer in memory, we check if the PAC is still valid. Additionally, each address has its own PAC. Only someone who know the CPU' master secret can generate a PAC an address.

The only way we could subvert this is if we knew the master secret (by exploiting a vulnerability in OS), bruteforcing the PAC, or reusing a pointer that is already there.

Address Space Layout Randomization (ASLR): put each segment of memory in a different location each time the program is run. As a result, the attacker can't know where their shellcode will be because its address changes every time you run the program. ASLR can shuffle all four segments of memory

Subverting ASLR: We can leak the address of a pointer, whose address relative to our shellcode is known or we could guess the address of our shellcode using brute force.

Practice Exam Takeaways:

The best way to ensure integrity is to create a scheme where only authorized people have access.

When the ret instruction executes, the next value of the stack is treated as an address which causes the program to start executing at that address.

Arguments are always stored in reverse order on the stack.

Similarly, variables inside structs are also stored in reverse order

Padding is only need for schemes which pass the message through the block cipher, not for schemes that xor the message with the output of the block cipher since we can discard the bits we don't need.

Schemes that use MACs or Hashes as the underlying block cipher to produce ciphertext don't provide integrity since we can't actually verify the ciphertext. We must use another scheme alongside it to provide integrity as well.

Interactive schemes from a person to a server usually require the person to commit something (i.e. send something to the server).

Remember that the attacker can perform an offline attack (e.g. compute a possible pwd's and hash them)

Fast hashing is useful when designing integrity schemes and slow hashing is useful for passwords.

We must be able to detect tampering in order for a scheme to have integrity.

HMAC's leak information because of determinism.

If IV's are predictable, attackers can supply messages that cancel out with the IV making the block cipher deterministic.

Certificates: A signed endorsement of someone's public key.

Certificate Authorities: Delegated trust form a pool of multiple rooms CAs. Roots CAs can sign certificates for intermediate CAs. Certificates contain an expiration date. Revocation: CAs sign a list of revoked certificates as well.

Offline Attack: The attacker performs all the computation themselves

Online Attack: The attacker interacts with the service

One-Time Pads:

KeyGen( ) - Randomly generate an n-bit key, where n is the length of the message. For one-time pads, we always generate a new key for every message

If we encrypt A and B with the same key K, the eavesdropper observes K ^ A and K ^ B. Then the eavesdropper can compute (K ^ A) ^ (K ^ B) = A ^ B. Therefore, the eavesdropper has learned A ^ B (i.e. they know which bits of A match with which bits of B).

Takeaway: One-time pads are insecure if the key is reused

Enc(K, M) = K ^ M

Dec(K, C) = K ^ C

Block Ciphers:

A block cipher is a cryptographic scheme consisting of encode/decode algorithms for a fixed-sized block of bits

Enc(K, M) -> C: Takes in a k-bit key K and an n-bit plaintext M and outputs and n-bit ciphertext C

D(K, C) -> M: Takes in a k-bit key K and an n-bit ciphertext C and outputs an n-bit plaintext. This is the inverse of the encryption function

Properties:

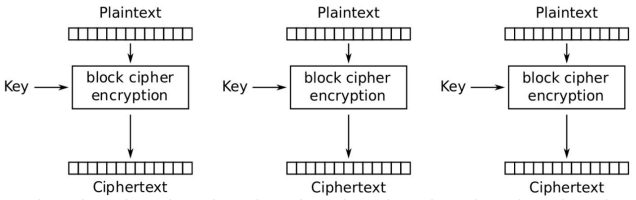
Correctness: Enc(K, M) is a permutation (bijective function) on n-bit strings (i.e. each input must correspond to exactly one unique output). If Enc(K, M) is not bijective then two inputs may correspond to the same output which means you cannot decode the ciphertext into the original message. Additionally, Dec(K, M) is the inverse.

Efficiency: Encoding/decoding must be fast

Security: A secure block cipher should behave like a randomly chosen permutation from the set of all permutations on n-bit strings. This property can be checked by a distinguishing game. Eve shouldn't be able to tell the encryption from another randomly chosen permutation. However, despite this, block ciphers are not IND-CPA secure since they are deterministic. In general, no deterministic (if the same input produces the same output) scheme can be IND-CPA secure.

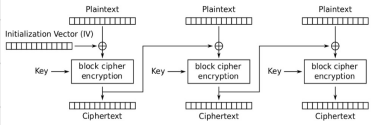
Block Cipher Modes of Operation:

Electronic Code Book (ECB) mode: Enc(K, M) = C1 || C2 || … || Cm

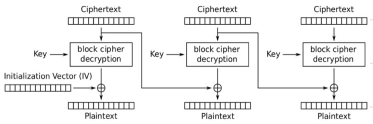


Cipher Block Chaining (CBC) mode: Ci = Enc(Mi ^ Ci-1); C0 = IV where IV is called the initialization vector which is different for every encryption.

Enc(K, M): Split M in m plaintext blocks P1, …, Pm each of size n. Then choose a random IV and compute and output (IV, C1, …, Cm) as the ciphertext.



Cipher Block Chaining (CBC) mode encryption

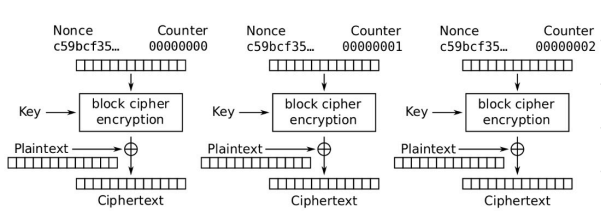


Cipher Block Chaining (CBC) mode decryption

If the message isn't a multiple of the block size, we pad the message until its a multiple of the block size. One padding scheme is that we append a 1 and then pad with zero. Another scheme is that we pad with the number of padding bytes. If we need 1 byte, pad with 01; if you need 3 bytes, pad with 03 03 03 etc. If you need 0 padding bytes, pad an entire dummy block.

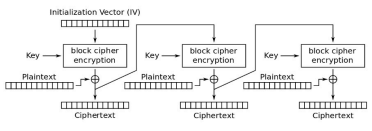
AES-CBC is IND-CPA secure with the assumption that that the IV is randomly generated and is never reused.

Counter (CTR) mode: We know that one-time pads are secure if we never reuse the key. Additionally, if a plaintext is encrypted with AES, the output looks random so can we use block ciphers to simulate a one-time pad? Yes, as follows:

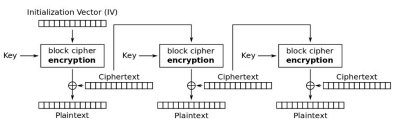


Decryption is identical as encryption but the block cipher output is xor'd with the ciphertext instead of the plaintext. In order to ensure IND-CPA, the nonce must never be reused and must be random.

Cipher Feedback (CFB) Mode:



Cipher Feedback (CFB) mode encryption



Cipher Feedback (CFB) mode decryption

Block ciphers are designed for confidentiality (IND-CPA). If an attacker tampers with the ciphers text, we are not guaranteed to detect it. No need to pad, just throw out the unneeded bits.

Block ciphers are only designed for confidentiality but Mallory can tamper with the ciphertext and we won't be able to detect it.

Cryptographic Hashes:

A hash function, H, takes in an input M and outputs a fixed length n-bit hash. The same input will always produce the same output, it is efficient to compute, is pre-image resistant [given an input y, it. Is infeasible to find any input x such that H(x) = y], collision resistant [x ≠ x' and H(x) = H(x'), we want to make it hard to find such collisions], and unpredictable. Finding a collision only n-bit output takes 2^(n/2) tries on average.

Length extension attack: Given H(x) and the length of x, but not x, an attacker can create H(x||m) for any m of the attacker's choosing. SHA-256 (256-bit version of SHA-2) is vulnerable. Hashing schemes usually don't provide integrity if Mallory can change the hash.

Message Authentication Codes (MACs):

We want to attach some piece of information to prove that someone with the key sent this message. Alice wants to send M to Bob, but doesn't want Mallory to tamper with it so Alice sends M and T = MAC(K, M). Bob receives M and T and Bob computes MAC(K, M) and checks whether it matches. If it matches then Bob can be confident that the message has not been tampered with. MAC generates a tag T for a message M using key K. They must be deterministic and EU-CPA (existentially unforgeable under chosen plaintext attack). A secure MAC that is EU-CPA means that without the key, an attacker cannot create a valid tag on a message (i.e. Mallory cannot generate MAC(K, M') without K and Mallory cannot find any M' ≠ M s.t MAC(K, M') = MAC(K, M). Even if Mallory can trick Alice into creating MACs for messages that Mallory chooses, Mallory cannot create a valid MAC on a message that she hasn't seen before already.

NMAC: NMAC(K1, K2, M) -> H(K1 || H(K2 || M)). NMAC is EU-CPA secure if the two keys are different. Using two hashes prevents a length extension attack otherwise an attacker who sees a tag for M could generate a tag for M || M'.

HMAC: We need two different same length keys for NMAC. Can we do it with only one key. HMAC(K, M) = H((K' ^ opad) || H((K' ^ ipad) || M)). The opad is the hardcoded byte 0x5c repeated and the ipad is the hard coded byte 0x36.

In general, MACs provide integrity since an attacker cannot tamper with the message without being detected. MAC can provide authentication if only two people have the secret key. If two people, then the message came from someone with the secret key but you can't narrow it down to one person. Additionally, MACs are deterministic, therefore, it is not IND-CPA secure.

Authenticated Encryption: A scheme that simultaneously guarantees confidentiality and integrity on a message. There are two ways to do this, combine schemes that provide confidentiality with scheme that provide integrity or use a scheme that is designed to provide confidentiality and integrity.

We can use an IND-CPA scheme with an unforgeable MAC scheme. If Alice sends Enc(K1, M) and MAC(K2, M), this scheme is not IND-CPA since MAC is not IND-CPA. What if we compute the MAC on the ciphertext instead of the plaintext.

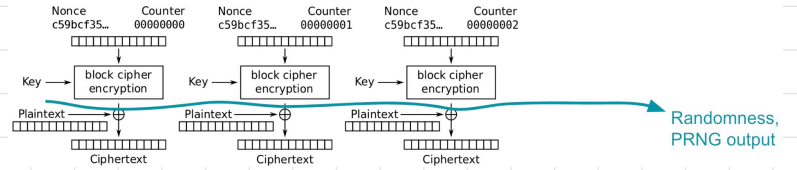
Encrypt-then-MAC: Enc(K1, M) and MAC(K2, Enc(K1, M))

MAC-then-Encrypt: Enc(K1, M || MAC(K2, M)).

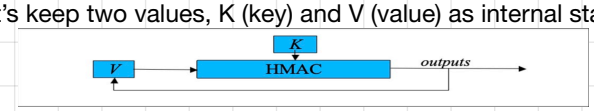
Key reuse: Using these same key in two different use cases. Reusing the key can cause the underlying algorithm to interfere with each other and affect security guarantees. Ex: if you use a block cipher MAC and block cipher chaining mode, the underlying block cipher may no longer be secure. Don't reuse keys! One key per use.

Authenticated encryption with additional data (AEAD): An algorithm that provides both confidentiality and integrity over the plaintext and integrity over additional data. This additional data is usually context (e.g. memory address), so you can't change context without changing MAC. If used incorrectly, could be catastrophic.

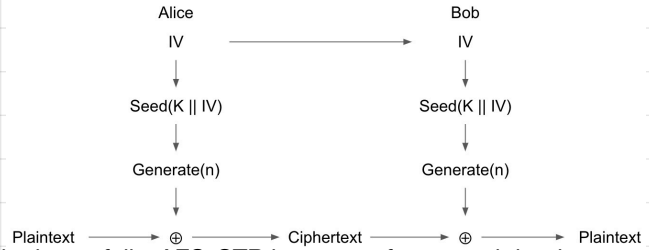
Pseudorandom Number Generators (PRNGs): Symmetric-key encryption schemes need randomness so how can we securely generate random numbers?
Entropy is a measure of uncertainty. In other words, a measure of how unpredictable the outcomes are and we want very high entropy. The uniform distribution has the highest entropy and is usually measured in bits (3 bits of entropy = uniform, random distribution over 8 values).
PRNGs: An algorithm that uses a little bit of true randomness to generate a lot of random-looking outputs. PRNGs are deterministic - output is generated according to a set algorithm, however, for an attacker who cannot see the internal state, the output is computationally indistinguishable from true randomness. A PRNGS has three functions:
PRNG.Seed(randomness): Initializes the internal state using the entropy.
PRNG.Reseed(Randomness): Updates internal state using existing state and entropy (should increase the entropy not reduce it).
PRNG.Generate(n): Generate n pseudorandom bits and updates the internal state as needed
A PRNG cannot be truly random. The output is deterministic given the initial seed. If the initial seed is s bits long, there are only 2^s possible output sequences. A secure PRNG is computationally indistinguishable from random to an attacker. The attacker should not be able to predict future output of the PRNG.
Rollback resistance: If the attacker learns the internal PRNG state, they cannot learn anything about previous states or outputs however this property is not required.
CTR-DRBG: PRNG.Seed(K || IV); Generate(M) = E(IV || 1) || … || E(IV || Ceil(m/n)); uses block cipher in CTR mode and uses the concatenation of all block cipher outputs as the result.



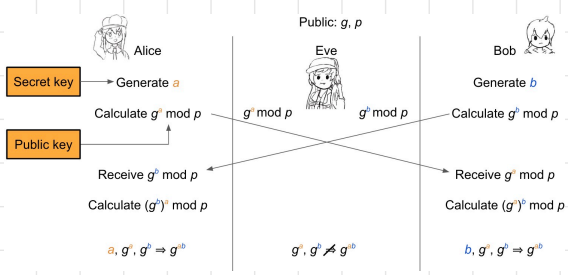
HMAC-DRBG: HMAC output looks unpredictable so why can't we use HMAC to build a PRNG? HMAC takes two arguments (key and message) so let's keep two values, K (key) and V (value) as internal state.



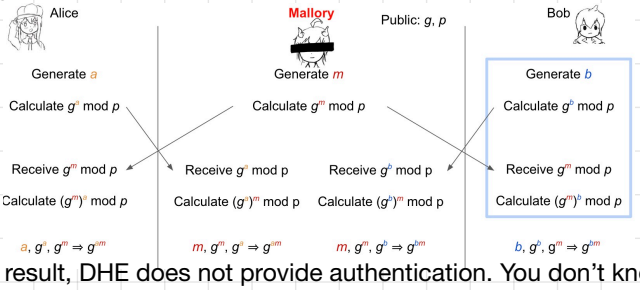
Assuming HMAC is secure, HMAC-DRMB is a secure rollback-resistant PRNG. If you can derive old outputs from the current state, you've found a way to reverse the hash function or HMAC. If you break HMAC-DRBG, you've either broken HMAC or the underlying hash function.
Stream Ciphers: A secure PRNG produces outputs that look indistinguishable from random so an attacker who cannot see the internal PRNG state cannot learn any outputs, so what if we used the PRNG output as the key to a one-time pad? A stream cipher is a symmetric encryption algorithm that uses pseudorandom bits as the key to a one-time pad.



If you look carefully, AES-CTR is a type of stream cipher because the output of the block ciphers is pseudorandom and used as a one-time pad. Stream ciphers are IND-CPA secure, assuming the pseudorandom output is secure. In some stream ciphers, security is compromised if too much plaintext this encrypted. Ex: in AES-CTR, if you encrypt so many block s.t the counter wraps around, you'll start to reuse keys. If the key is not bits long, you should stop after 2^(n/2) bits of outputs.
Diffie-Hellman Key Exchange: Given g, p, g^a mod p, and g^b mod p for random a, b, no polynomial time attacker can distinguish between a random value R and g^(ab) mod p. Discrete logarithm problem: Given g, p, g^a mod p for random a, it is computationally hard to find a.



Diffie-Hellman can be used ephemeral. Ephemeral: short term and temporary, not permanent. Alice and Bob discard a, b, and K = g^(ab) mod p when they're done. Since you need an and b to derive K, you can never derive K again. Such a K is called a session key because it's only used for an ephemeral session.
Forward Secrecy: Assume that Eve records everything sent over the insecure channel. Alice and Bob use DHE to agree on a key K = g^(ab) mod p and then use K as a symmetric key. After they're done, discard a, b, and K. If Eve steals all of Alice and Bob's secrets, Eve can't decrypt any messages she recorded since nobody saved a, b, or K, and her recording only has g^a mod p and g^b mod p.

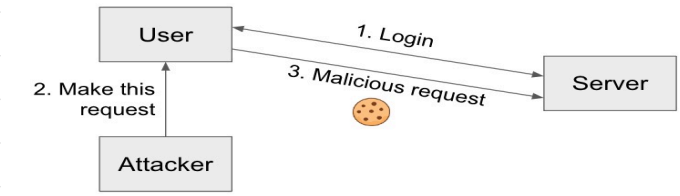


As a result, DHE does not provide authentication. You don't know who you exchanged keys with.
Public-Key Cryptography:
In public-key schemes, each person has two keys, a public key and a private key. Every public key corresponds to one private key. Enc(PK, M) -> C: Encrypt a plaintext M using public key PK to produce C. Dec(SK, C) -> M: Decrypt C using secret key SK.
ElGamal Encryption: KeyGen( ) - Bob generates private key B and public key B = g^b mod p. Enc(B, M) - Alice generates a random r and computes R = g^r mod p. Alice then computes M * B^(r) mod p. Alice sends C1 = R, C2 = M * B^(r) mod p. Dec(b, C1, C2) - Bob computes C2 * C1^(-b) = M * B^r * R^(-b) = M * g^(br) * g^(-br) mod p = M mod p. ElGamal isn't IND-CPA secure and the message is malleable because they can tamper with the message. They can send 2 * C2. RSA Encryption: KeyGen( ) - Randomly pick to large primes p and q. Compute N = pq. Chose e thats relatively prime to (p - 1)(q - 1). Compute D = e^(-1) mod (p - 1)(q - 1). PK = N and e; SK = d. Enc(e, N, M) = M^e mod N. Dec(d, C) = C^d mod N = (M^e)^d = M mod N. No way to find M if you only know N and C. Doing so would require factoring N fast. RSA is not IND-CPA.
Hybrid Encryption: encrypt data under a randomly generated key K using symmetric encryption and encrypt K using asymmetric encryption. Almost all cryptographic systems use this type of encryption.
Digital Signatures: KeyGen( ) -> PK, SK: generate a public/private keypair, where PK is the verify key and SK is the signing key. Sign(SK, M) -> sig: sign the message M using the signing key SK to produce a signature. Verify(PK, M, sig) -> {0, 1}: Verify sig on M using PK. Digital Signature schemes should be EU-CPA. RSA Signatures: Public key: N and E; Private key: d. Sign(d, M) = H(M)^d mod N. Verify(e, N, M, sig); H(M) = sig^e mod n?
Security Principles:
Know your threat model: Understand your attacker and their resources and motivation **Consider human factors:** If your system is unusable, it will be unused **Security is economics:** Balance the expected cost of security with the expected benefit **Detect if you can't prevent:** Security requires not just preventing attacks but detecting and responding to them **Defense in depth:** Layer multiple types of defenses **Least privilege:** Only grant privileges that are needed for correct functioning, and no more **Separation of responsibility:** Consider requiring multiple parties to work together to exercise a privilege **Ensure complete mediation:** All access must be monitored and protected, unbypassable **Shannon's maxim:** The enemy knows the system **Use fail-safe defaults:** Construct systems that fail in a safe state, balancing security and usability. **Design in security from the start:** Consider all of these security principles when designing a new system, rather than patching it afterwards

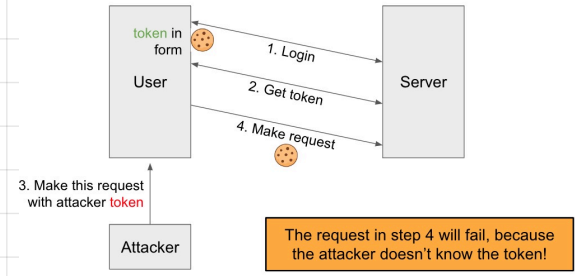
Introduction to Web: The website is a collection of data and services URLs: (Uniform Resource Locator) The way in which we uniquely identify one piece of data on the web (Scheme: https, http, ftp, etc - Domain: toon.cs161.org, google.com, etc - Port: number that directly follows the domain - Path: looks like /xorcist/avian.html, etc). We can use the ampersand (&) to separate arguments (in the query which is started after the question mark (?)) and use URL Escaping for special characters such as the ampersand, percent, etc.
HTTP (Hypertext Transfer Protocol): A protocol used to request and retrieve data from a web server) HTTPS: A secure version of HTTP which uses cryptography to secure data. HTTP is a request-response model. The web browser sends a request to the web server and the web server processes the request and sends a response to the web server.
An HTTP request consist of a URL path (possibly with query parameters) and a method. GET: Requests that don't change server-side state ("get" information from the server) POST: Request that update server-side state ("post" information to the server). Today, GET requests typically modify server-side state in some ways (e.g. analytics), but using GET instead of POST can have security implications. GET requests do not contain any data. POST requests can contain data.
Same-origin policy: A rule that prevents one website from tampering with other unrelated websites. This is enforced by the web browser and prevents a malicious website from tampering with behavior on other websites. Two web pages have the same origin if and only if the protocol, domain, and port of the URL all match exactly. Think string matching: The protocol, domain, and port strings must be equal Two websites with different origins cannot interact with each other
Exception: JavaScript runs with the origin of the page that loads it.
Exception: Websites can fetch and display images from other origins.
Exception: Websites can agree to allow some limited sharing.
Cookie: a piece of data used to maintain state across multiple requests. Creating cookies - The server can create a cookie by including a Set-Cookie header in its response, JavaScript in the browser can create a cookie, Users can manually create cookies in their browser. Storing cookies - Cookies are stored in the web browser (not the web server) The browser's cookie storage is sometimes called a cookie jar. Sending cookies - The browser automatically attaches relevant cookies in every request, The server uses received cookies to customize responses and connect related requests. If the Secure attribute is True, then the browser only sends the cookie if the request is made over HTTPS (not HTTP). If the HttpOnly attribute is True, then JavaScript in the browser is not allowed to access the cookie. **Cookie policy:** A set of rules enforced by the browser; cookie policy is not the same as same-origin policy. Server with domain X can set a cookie with domain attribute Y if the domain attribute is a domain suffix of the server's domain (X ends in Y, X is below or equal to Y on the hierarchy, X is more specific or equal to Y, The domain attribute Y is not a top-level domain (TLD). There are no restrictions for the Path attribute (the browser will accept any path). (Examples: mail.google.com can set cookies for Domain=google.com, google.com can set cookies for Domain=google.com, google.com cannot set cookies for Domain=com, because com is a top-level domain). Cookie Sending Policy: The browser sends the cookie if both of these are true: the domain attribute is a domain suffix of the server's domain and the path attribute is a prefix of the server's path

Session: A sequence of requests and responses associated with the same authenticated user. Session token: A secret value used to associate requests with an authenticated user. **What attributes should the server set for the session token?** Secure: Can set to True to so the cookie is only sent over secure HTTPS connections HttpOnly: Can set to True so JavaScript can't access session tokens

Idea: What if the attacker tricks the victim into making an unintended request? The victim's browser will automatically attach relevant cookies and the server will think the request came from the victim! This is called Cross-site request forgery (CSRF or XSRF): An attack that exploits cookie-based authentication to perform an action as the victim.
Strategy #1: Trick the victim into clicking a link. Note: Clicking a link in your browser makes a GET request, not a POST request, so the link cannot directly make the malicious POST request. Instead, the link can open an attacker's website, which contains some JavaScript that makes the actual malicious POST request. Strategy #2: Put some JavaScript on a website the victim will visit. Example: Pay for an advertisement on the website, and put JavaScript in the ad. Recall that JavaScript can make a POST request



CSRF Defenses CSRF Token: A secret value provided by the server to the user. The user must attach the same value in the request for the server to accept the request. CSRF tokens cannot be sent to the server in a cookie! Idea: In a CSRF attack, the victim usually makes the malicious request from a different website



Referer header: A header in an HTTP request that indicates which webpage made the request Checking the Referer header: Allow same-site requests: The Referer header matches an expected URL (Example: For a login request, expect it to come from https://bank.com/login) and disallow cross-site requests (i.e. the Referer header does not match an expected URL). If the server sees a cross-site request, reject it. Idea: Implement a flag on a cookie that makes it unexploitable by CSRF attacks. This flag must specify that cross-site requests will not contain the cookie. SameSite flag: A flag on a cookie that specifies it should be sent only when the domain of the cookie exactly matches the domain of the origin

Cross-site scripting (XSS): Injecting JavaScript into websites that are viewed by other users; Cross-site scripting subverts the same-origin policy

Stored XSS (persistent XSS): The attacker's JavaScript is stored on the legitimate server and sent to browsers. Stored XSS requires the victim to load the page with injected JavaScript

Reflected XSS: The attacker causes the victim to input JavaScript into a request, and the content is reflected (copied) in the response from the server. Reflected XSS requires the victim to make a request with injected JavaScript

How do we force the victim to make a request to the legitimate website with injected JavaScript? We should trick the victim into visiting the attacker's website, and include an embedded iframe that makes the request, trick the victim into clicking a link (e.g. posting on social media, sending a text, etc.), or trick the victim into visiting the attacker's website, which redirects to the reflected XSS link. Note that Reflected XSS and CSRF both require the victim to make a request to a link but they are different.

Defense - HTML sanitization: Replace special characters with ampersand and semicolon. Defense - Content Security Policy (CSP): idea: Instruct the browser to only use resources loaded from specific places. This uses additional headers to specify the policy. Standard approach: Disallow all inline scripts (JavaScript code directly in HTML), which prevents inline XSS and Only allow scripts from specified domains, which prevents XSS from linking to external scripts Clickjacking: Trick the victim into clicking on something from the attacker Phishing: Trick the victim into sending the attacker personal information,

SQL and Javascript:

```
SELECT [columns] FROM [table] WHERE [conditions]
INSERT INTO [table] VALUES (val1,val2,...), ...
UPDATE [table] SET [column] = [value] WHERE [condition]
DELETE FROM [table] WHERE [condition]
DROP [table]
CREATE TABLE [table]{ entry 1 TYPE, ...}
SELECT [columns] FROM [table] WHERE ... UNION SELECT...
-- (comments) for single line comments. Semicolons
seperate different statements
```

Introduction to Networking:

Network: A set of connected machines that can communicate with each other. **Internet**: A global network of computers. A **protocol** is an agreement on how to communicate that specifies syntax and semantics. **OSI model**: Open Systems Interconnection model, a layered model of Internet communication.

Layer 1 Physical Layer: Sends bits from one device to another. Encodes bits to send them over a physical link (patterns of voltage levels, RF modulation, etc.) **Layer 2 Link Layer**: Sends frames directly from one device to another. Relies on Layer 1. Encodes the messages into grouped of bits called frames. **Land Area Network (LAN)**: a set of computers on a shared network that can directly address on another. Frames must have 3 things, Source, Destination, and Data. **Ethernet**: a common layer 2 protocol that most endpoint devices use. **MAC Address**: A 6-bytes address that identifies a piece of network equipment. **Layer 3 Network Layer**: Sends packets from any device to any other device. Relies on Layer 3. Encodes messages into groups of bits called packets. Packets must contain 3 things, Source, Destination, and Data. Packets can be fragmented into smaller packets. **Reliability** ensures that packets are received correctly or, if random errors occur, not at all. This is implemented with a checksum. IP is unreliable and only provides a **best effort** service which means that packets may be lost, corrupted, or delivered out of order. Higher level protocols must ensure that the connection is reliable. **Layer 4 Transport Layer**: Provides transportation of variable length data from any point to any other point. Examples of this are TCP and UDP. **Layer 7 Application Layer**: Provides applications and services to users. Relies on Layer 4. Every online application is Layer 7.

Low Level Network Attacks:

	Can modify or delete packets	Can read packets
Man-in-the-middle attacker	✓	✓
On-path attacker		✓
Off-path attacker		

Address Resolution Protocol (ARP): Translates layer 3 IP addresses to layer 2 MAC addresses. Steps: Alice checks her cache to see if she already knows Bob's MAC address. If Bob's MAC address is not in the cache, Alice **broadcasts** to everyone on the LAN: "What is the MAC address of Bob's IP?" Bob responds by sending a message only to Alice "My IP is X and my MAC address is X". Alice caches Bob's MAC address. If Bob is outside of the LAN, the router will respond with its MAC address to be routed to other LANs that reach Bob. Alice can't verify the response of the ARP. An attacker can race Bob's response to provide his malicious MAC. Mallory must be in the same LAN as Alice in order to work. ARP spoofing allows Mallory to become a man-in-the-middle attacker. To defend against this, we can use **switches** to avoid broadcasts and ARP requests. When Alice wants to send a message to Bob, she sends the message to a switch on the LAN. The switch contains mappings of IP/MAC addresses. If Bob's MAC is in the cache, the switch sends the message to Bob, else the switch broadcasts the message.

Dynamic Host Configuration Protocol (DHCP):

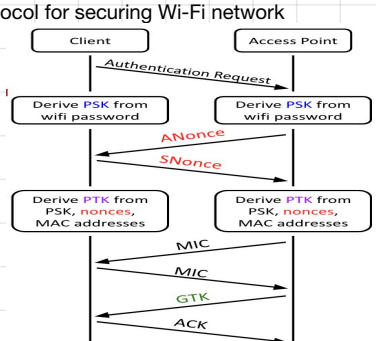
1. **Client Discover**: The client broadcasts a request for a configuration.
2. **DHCP Offer**: Any DHCP server can respond with a configuration offer. Usually only one DHCP server responds. The offer includes an IP address for the client, the DHS server's IP address, and the (gateway) router's IP address. The offer also has an expiration time (how long the user can use this configuration).
3. **Client Request**: The client broadcasts which configuration it has chosen. If multiple DHCP servers made offers, the ones that were not chosen discard their offer. The chosen DHCP sever gives the offer to the client.
4. **DHCP Acknowledgement**: The chosen server confirms that its configuration has been given to the client.

Alice has no way of verifying the DHCP response. Any attacker on the network can claim to have a configuration. Alice expects only one DHCP server to respond, so she will accept the first response. As long as the attacker responds faster, Alice will accept the attacker's response. DHCP attacks require that Mallory be in the same LAN as Alice. DHCP attacks allow Mallory to become a MITM attacker. Mallory claims the gateway and DNS server address is his own to intercept and modify packets before they are sent out. Hard to defend against since there is no root of trust. Therefore, we rely on defenses provided in higher layers.

Wi-Fi: A layer 2 protocol that wirelessly connects machines in a LAN. It has an **access point** which is a machine that will help you connect to the network and a **SSID** which is the name of the Wi-Fi network. May also have a password to secure Wi-Fi comms.

Wi-Fi Protected Access 2 (WPA2): A protocol for securing Wi-Fi network communications with cryptography.

1. The client sends an authentication request to the access point
2. Both use the password to derive the *PSK* (pre-shared key)
3. Both exchange random nonces
4. Both use the *PSK*, nonces, and MAC addresses to derive the *PTK* (pairwise transport keys)
5. Both exchange MICs (these are MACs from the crypto unit) to ensure no one has tampered with the nonces, and that the PTK was correctly derived
6. The access point encrypts and sends the *GTK* (group temporal key) to the client, used for broadcasts that anyone can decrypt
7. The client acknowledges receiving the GTK



Deny-allow: Allow all traffic but deny those on a specified deny list **Default-deny**: Deny all traffic, but allow those on the allow list Firewalls are often packet filters, which inspect packets and chooses what to do with them. **Stateless packet filters** have no history and all decisions must be made using only the information in the packet itself. If the packet uses TCP, we can allow incoming traffic is the ACK flag is set and deny traffic without it. For UDP, this is impossible to implement. **Stateful packet filters**: keep state in the implementation of the packet filter. Can also track the state of well known applications like HTTP and FTP. **Proxy Firewall**: instead of forwarding packets, form two TCP connections, one with the source and one with the destination. **Application proxy firewall**: certain protocol allow for proxy in at the application layer. **VPN:** A set of protocols that allows direct access to an internal network via an external connection. **Application Level DoS**: attack the high-level application **Network Level DoS**: Attackers the network of service **DDoS**: Use multiple systems to overwhelm the target system **Amplified DoS**: use an amplifies to overwhelm the target more effectively **SYN flooding**: A type of DoS that causes a server to allocate state for unfinished TCP connection, upon receiving a SYN packet. **Proof of Work**: force users to spend some resources to issue a request **Overprovisioning**: allocate a huge amount of resources

NIDS: a detector installed on the network, between the local network and the Internet. Cheap: single detector can cover a lot of systems. Easy to scale, simple management, smaller TCB, and end system are unaffected. Inconsistent interpretation drawback - all traffic is encrypted. **HIDS**: a detector installed on each end system. Expensive, works on encrypted messages, protects against non-network threats, scales better than NIDS. **Logging**: analyze log files generated by end systems. Cheap but attacks are only detected after the attack has happened. Attacker could also change the logs to erase evidence of an attack

Proxy: A third party that relays our Internet Traffic. Alice sends the message and the recipient to proxy and the proxy forwards the message to Bob. Bob receives the message from the proxy with no indication that it came from Alice. **Tor**: A network that uses multiple proxies (relays) to enable anonymous communication. Tor client for a circuit consisting of 3 relays. Step 1: Query the directory server for a list of relays. Step 2: Choose 3 relays to form a Tor circuit. Step 3: Connect to the first relay, forming an end-to-end TLS connection. Step 4: Connect to second relay through first relay, with TLS. Step 5: Connect third relay through second relay, with TLS. Step 6: Connect to web server. **Tor packet**: Wrap the packets via encryption (i.e. encrypt the message with the key of 3rd relay, then key of 2nd, then finally, the key of the 1st. Then as the packet is sent through the relays, unwrap the encryption. The 1st relay knows the message is from Alice and the last relay knows that message and that it is intended for Bob. All relays in the middle don't know contents, sender, nor recipient. The exit node is a MITM attacker. A network attacker with global view of network can learn that Alice and Bob are talking with timing attack. **Collusion**: multiple nodes working together and sharing information. Collusion is dishonest behaviors. If all nodes collude, anonymity is broken. If at least one node is honest, anonymity is preserved. Easy to form colluding nodes. **Guard nodes**: have a high reputation and have existed for a long time. The entry node is usually a guard node. Also Tor doesn't hide that you are using Tor. **Tor Bridges** defend against this by providing nodes not available on any public list as the entry node. We can also obfuscate traffic.

Malware: Attacker code running on victims computers

Self-replicating code: A code snippet that outputs a copy of itself

Virus: Code that requires user action to propagate

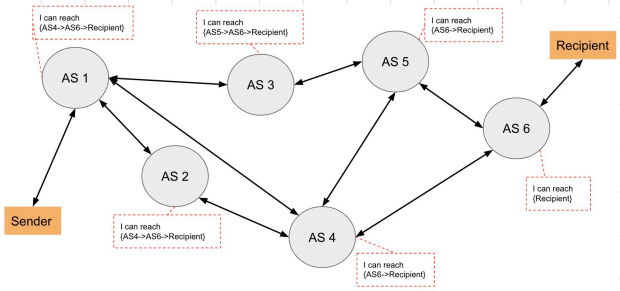
Worm: Code that does not require user action to propagate

Virus Detection Strategies: Signature based detection - viruses replicate by using copies of the same code. We can capture a virus on one system and look for bytes corresponding to the virus code on other systems

Polymorphic Code: Each time the virus propagates, it inserts an encrypted copy of the code along with the key it used. Encryption being used is obfuscation, not confidentiality. We can defend against this by adding a signature for detecting decryptor code or check if code performs decryption in a sandbox environment.

Metamorphic Code: Each time the virus propagates, it generates a semantically different version of the code. Renumber registers, change if-else order, replace algorithms, add useless code.

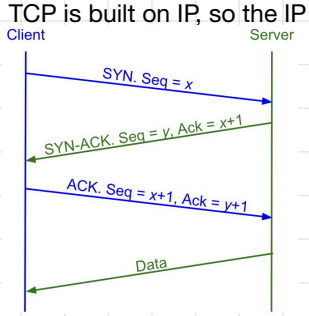
Border Gateway Protocol (BGP):
The **Internet Protocol (IP)** is the universal layer-3 protocol that all devices use to transmit data over the internet. An **IP address** is an address that identifies devices on the internet. IP routes by **subnets**, which are groups of addresses with a common prefix. To send a packet to a computer within the local network, we verify that the destination IP is in the same subnet, use ARP to get the destination MAC address, and send the packet directly to the destination using the destination MAC. To send a packet to a computer that is not within the local network, send that packet to the gateway. Past the gateway, the packet has to be routed through the internet. The Internet is a network of networks, comprised of many **autonomous systems (AS)** which are uniquely identified by its **autonomous system number (ASN)**. Each AS is comprised of one or more LANs and can forward packets to other ASes. The protocol for communication between AS is BGP.



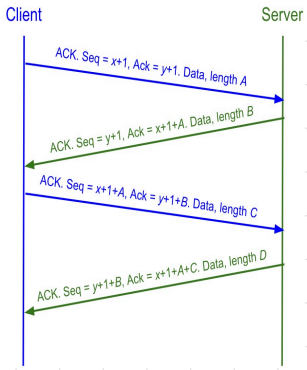
IP spoofing: Malicious clients can send IP packets with source IP values set to a spoofed value. Ideally edge ASes should check this
BGP hijacking: A malicious autonomous system can lie and claims itself to be responsible for a network which it isn't.

Transmission Control Protocol (TCP):
Provides a byte stream abstraction - bytes go in one end of the stream at the source and come out at the other end at the destination. TCP automatically breaks streams into segments. Provides ordering, reliability, and ports. **Ports** help us distinguish between different application on the same computer or server. TCP is built on IP, so the IP header is still present.

1. Client chooses an initial sequence number x and sends SYN packet to the server
2. Server chooses an initial sequence number y and responds with SYN-ACK
3. Client returns with an ACK
4. Once both have synchronized, the connection is established



If a packet is dropped, the recipient will not an ACK, so the sender will not receive the ACK. The sender repeatedly tries to send the packet. If a packet is received, but the ACK is dropped, the recipient will ignore the packet and wait for another one with the ACK.

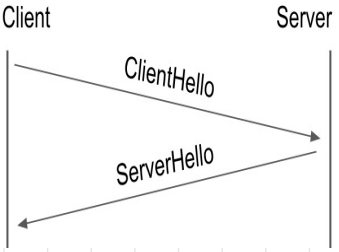


To end a connection, on side sends a packet with the FIN flag set which means that "I will no longer be sending any more packets, but I will still receive". To abort a connection, one side sends a packet with the RST flag set. This means that I will no longer be sending nor receiving packets.
TCP Hijacking: tampering with an existing session to modify or inject data. **Data injection**: spoofing packets to inject malicious data into a connection. **RST injection**: spoofing a RST packet to forcibly terminate
TCP spoofing: spoofing a TCP connection to appear to come from another source IP address. TCP provides no confidentiality or integrity.
User Datagram Protocol (UDP): provides Datagram abstraction and has a maximum size. No reliability or ordering guarantees, but uses ports. It is much faster than TCP since there is no 3-way handshake. We can easily inject data into a connection or spoof connection.

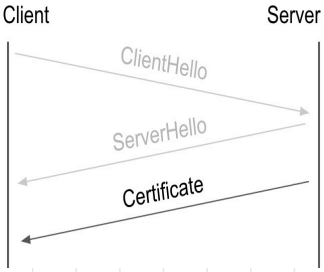
Signature Detection: Flag any activity that matches the structure of a known attack (e.g. blacklisting). Simple, good at detecting know attacks, easy to share signatures. Won't catch new attacks, variants of old attack. **Specification Detection:** Specify allowed behavior (whitelisting). Can detect new attacks, low FPR, takes a lot of time, may need to update. **Anomaly Detection:** Develop a model of what normal activity looks like. Similar to specification but learn a model. Can fail to detect known attacks, new attacks, but can detect attacks we haven't

TLS (Transport Layer Security): A protocol for creating a secure communication channel over the internet. This protocol is built on top of TCP. The goals of TLS are to provide confidentiality (ensure that attackers cannot read your traffic), integrity (ensure that attackers cannot tamper with your traffic), and authenticity (that you're talking to the legitimate server.

The client sends **ClientHello** with a 256-bit random number R_b and a list of algorithms to use.
The server sends **ServerHello** with a 256-bit random number R_s and a list of algorithms to use (from the client)
These random values prevent replay attacks. Ensures that no hand shake is ever the same.

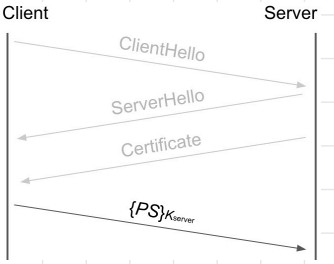


The server then sends its certificate.
The certificate is the server's identity and public key, signed by a trusted certificate authority.
The client validates the certificate (by verifying the signature in the certificate)
After this, the client now knows the server's public key. Importantly, at this point, the client still doesn't know if they are talking to the legitimate server since certificates can be provided by anyone



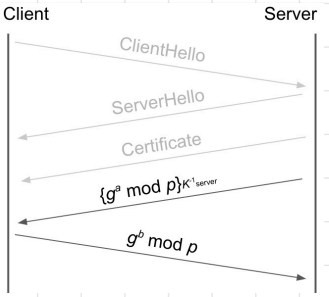
We will now generate the Premaster Secret which has two main purposes. It makes sure the client is talking to the legitimate server. This is because the server must prove that it owns the private key corresponding to the public key in the certificate and gives the client and the server a shared secret. This is achieved via RSA or Diffe-Hellman (DHE)

The client randomly generates a premaster secret (PS). The client encrypts PS with the server's public key and sends it to the server. (Recall that the client knows the server's PK).
The server decrypts the premaster key
The server and the client know share a secret.

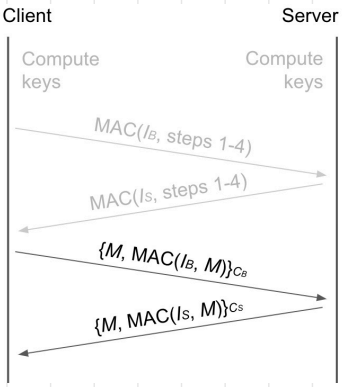


Or

The server generates a secret an and computes g^a mod p. The server signs g^a mop p with its private key and sends the message and signature. The client verifies the signature (which proves that the server owns the private key) and then the client generates a secret b and computes g^b mod p. The client and server now share a **premaster secret** (g^(ab) mod p).



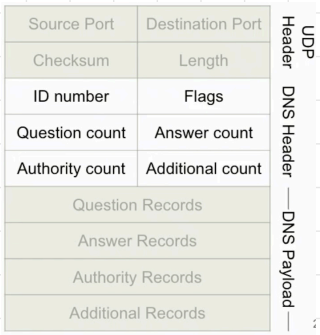
The server and the client derive symmetric keys from R_b, R_s and PS (usually done by seeding a PRNG with the three different values). Four symmetric keys are derived. C_b (for encrypting client-to-server messages), C_s (for encrypting server-to-client messages), I_b (for MACing client to server messages), and I_s (for MACing server to client messages. Server and client then exchange MACs on all message of the handshake so far to detect any tampering. After this check, messages can now be sent securely (Enc then MAC scheme).



Every handshake uses a different R_b and R_s so we cannot do any replay attacks. This is because the 4 shared keys will be different every time. Add **record numbers** in the encrypted TLS message to ensure that if the attacker replays a message in the from the current connection, the record numbers will repeat.

Computers only understand IP addresses, not human readable domain names like www.google.com. Therefore, we need someway to translate between human readable domains and IP addresses.
DNS (Domain Name System): An Internet protocol for translating human-readable domain names to IP addresses.
Name server: A server on the internet responsible for answering DNS requests. We send a DNS query to the name server and the name server sends a DNS response with the answer (e.g. "The IP address of www.google.com is …).
Name Server Hierarchy: If one name server doesn't know the answer to your query, the name server can direct you to another name server. We can do this by arranging the name servers in a tree hierarchy, with the intuition being that name servers will direct you down the tree until you receive the answer to your query. For example the .edu name server is responsible for queries like eecs.berkeley.edu, but not a query like mail.google.com. Each node in the tree represents a zone of domain and can delegate part of its zone to someone else.
DNS Lookup: All DNS queries start with a request to the root name server. We ask the root and the root says that it doesn't know but he has delegated authority to the .edu name server. Then you ask .edu the IP address of eecs.berkeley.edu. .edu doesn't know so it delegates authority to the berkeley.edu name server. Finally, the berkeley.edu name server responds with the IP address of eecs.berkeley.edu.
Stub Resolver: your computer tells another resolver to perform the query for you, and it only contacts the recursive solver and receives the answer. The Recursive Resolver is actually responsible to make the DNS queries and is usually run by ISPs or application providers. Remember that in DHCP, the user needs to fetch the IP address of the DNS server (a recursive resolver) for these queries. The stub resolver sends the query to the recursive resolver, the recursive resolver returns the final answer to the stub resolver. DNS uses UDP because we want DNS to be lightweight and fast.

ID number: Used to associate the queries with responses
Counts: The number of records of each type in the DNS payload
Resource Records: Question, answer, authority, and additional section
A type records: maps a domain name to IPv4
NS type record: designates another DNS server to handle a domain
Question section: What is being asked
Answer section: A direct response to the question
Authority section: A delegation of authority for the question - used to direct the resolver to the next name server and of type NS
Addition section: Additional information to help with the response - also called glue records. Provides non-authoritative records for domains

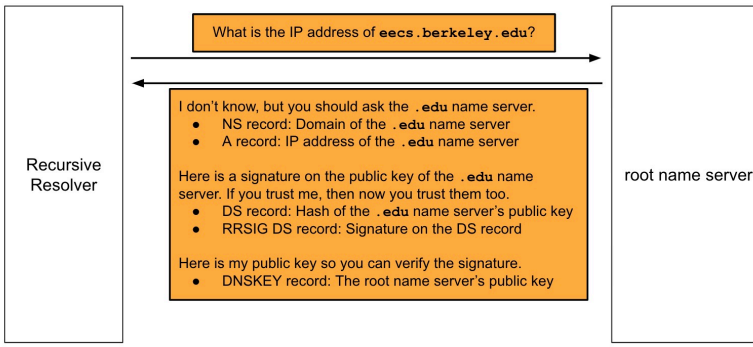


Cache Poisoning: Returning a malicious record to the client. Ex: supply a malicious A record mapping the attacker's IP address to a domain. Malicious name servers can lie and supply a malicious answer.
Defense - Bailiwick Checking: the resolver only accepts records if they are in the name server's zone.
MITM Attackers: they can poison the cache by adding, removing, or changing any record in the DNS response to their will. On-Path Attacker: They can attempt to race the actual name servers response to the recursive solver. Off-path attackers: they have to guess the ID field to spoof the response so they need to guess it. If the ID is randomly generated, the probability of guessing correctly is 1/2^16.
Kaminsky Attack: DNS clients would cache glue records as if they were authoritative answer. Use fake website with guessed ID fields and fake authority sections linking the search domain name with Mal's IP address

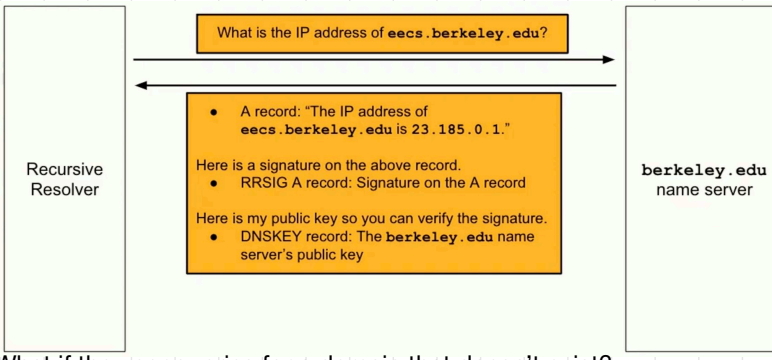
	Attacker **correctly** guesses the ID number	Attacker does **not** correctly guess the ID number
Attacker **beats** the race condition against the legitimate NS	The recursive resolver caches a mapping from legitimate domain names to the attacker's desired IP addresses.	The recursive resolver ignores the response because the ID does not match the request sent earlier. The recursive resolver caches something saying "This domain does not exist". Try again with another fake domain!
Attacker does **not** beat the race condition against the legitimate NS	The recursive resolver caches something saying "This domain does not exist". Try again with another fake domain!	The recursive resolver caches something saying "This domain does not exist". Try again with another fake domain!

Defense - Source Port Randomization: Randomize the source port of the DNS query to increase the number of guesses needed.
Defense - Glue Validation: Don't cache glue records as part of DNS lookups.
DNSSEC:
Public-Key Infrastructure: Name servers are arranged in a hierarchy, as in ordinary DNS. Parents can delegate trust to children. The parent signs the child's public key to delegate trust to the child - if you trust the parent name server, then now you trust the child name server. We implicitly trust the root name server. This method defeats malicious name servers.
A DNS record has a name, type, and value. A group of DNS records with the same name and type form a resource record set (RRSET). Used for simplifying signatures - instead of signing every record separately, we can sign an entire RRSET at once.
RRSIG (resource record signature): encodes signatures on records
DNSKEY: encodes the public key
DS (delegated signer): encodes the child's public key (used to delegate trust (the parent signs the child's public key to delegate trust to the child - DNSSEC delegates trust with two records: a DS type record with the hash of the signers name and the child's PK and an RRSIG type record with a signature on the DS record.

Query:



Final Answer:



What if the user queries for a domain that doesn't exist?
Option 1: Don't authenticate NXDOMAIN requests. Issue is that if NXDOMAIN responses don't have to be signed, attacker can spoof NXDOMAIN response for DoS
Option 2: Keep the private key in the name server itself, so it signs NXDOMAIN responses. Issues is that name servers have access to private key, which is an issue if they are hacked. Signing is also slow. Therefore, we need a way to prove that a domain doesn't exist ahead of time.
To prove no existence of a record type: sign a record stating that no record of a given type exists. To prove nonexistentence of a domain, provide two adjacent domains alphabetically, so that you know that no domain in the middle exists. Issue with this model (called NSEC) is that attacker can find every single subdomain of a domain by sequence of faulty DNS requests. Instead, we should store sequence of adjacent hashes. Still possible to brute-force all reasonable domain names. Only prevents attackers from learning long, random domain names, which would make brute-force difficult. NSEC3: given a nonexistent domain name, generate a signature on the fly that states that no record exists between H(name) - 1 and H(name) + 1. This requires online signature generation. Instead, we use offline signature generation. Offline signature: the application that computes signature is separate from the application that serves the signatures. This is efficient since records are signed ahead of time. Attacker must compromise the signature generation system. If the system is separate from the name server, we must compromise both.