EECS 388 Mid CTPP Zhusheng Li

Message Integrity: ensure attackers cannot modify message without being detected

Alice —(m, v)→ Mallory —(m', v')→ Bob  Machine-in-the-middle (MITM)
(Verifier: v' = f(m'))  f(): Random? : Too large! (k is secret n-bit key)

Pseudorandom functions (PRF): $f_0() \dots f_{(2^n-1)}()$ known to Mallory. $v() = f_k()$  Coin b = 0 RFC() / 1 f_k()

Cryptographic Hash Functions: HC()
Input: arbitrary length data
Output: fixed size digest (n bits)
- preimage resistance: Given h, hard to find m s.t. h=H(m)
- Second preimage resistance: Given $m_1$, find $m_2$ s.t. $H(m_1)=H(m_2)$
- collision resistance: hard to find $m_1, m_2$ s.t. $H(m_1)=H(m_2)$

SHA-256 (Output 256 bit)
Input: arbitrary length data
Output: 256-bit digest

Built from a compression function h:
Inputs: (256 bits, 512 bits), Output: 256 bits
"compresses" 768 bits to 256 bits using a complicated function (details out of scope for 388)

1. Pad input m to the next multiple of 512 bits (adds at least 1 bit, uses fixed algorithm (why?)) and split into 512-bit blocks: $b_0, b_1, \dots b_{n-1}$
2. $y_0$ := <256-bit constant>; $y_1 := h(y_0, b_0) \dots y_i := h(y_{i-1}, b_{i-1})$
3. Return $y_n$ which is defined to be SHA-256(m)


Pseudorandom Generator (PRG): $g_k: \perp \to \{0,1\}^n$ for n = poly(|k|)  (k is a truly random seed)
Build PRG from PRF: Given random key k, PRF $f_k()$, $g_k := f_k(0) \,//\, f_k(1) \,//\, f_k(2) \dots$
Randomness is an input to your program (Provided by OS, via special APIs)
Confidentiality: keep the content of message p secret from an eavesdropper (窃听)  k = {k=0, k=1, k=2}
Alice [c := $E_k(p)$] —send c / Eve→ Bob [p := $D_k(c)$]
Caesar Cipher: $(p_i + k) \bmod 26$, $(c_i - k) \bmod 26$  Vigenère: $(p_i + k_{i \bmod n}) \bmod 26$, $(c_i - k_{i \bmod n}) \bmod 26$
One-Time Pad (OTP): share a secret, long string of random bits k  $c_i = p_i \oplus k$  $p_i = c_i \oplus k$ (Impractical)
Stream Cipher (use PRG): share k, s := $g_k()$  $c_i = p_i \oplus s_i$
Block Cipher: function that encrypts fixed-size(n) blocks with reusable k: $E_k(p): \{0,1\}^{|k|} \times \{0,1\}^n \to \{0,1\}^n$
AES (Advanced Encryption Standard) Block Cipher: (block size 128, key size: 128, 192, 256)
Padding: add bytes to end of message to make it a multiple of block size (PKCS7: M M M M M 03 03 03)
Cipher modes: 1. Encrypted codebook (ECB) mode  $c_i = E_K(p_i)$ (preserve pattern, replay, no error propagation)

Length Extension Attack
Given $y = H(x)$ for unknown x
Attacker can calculate $z = H(x \,//\, \text{padding} \,//\, s)$ for any s
Solution: Message Authentication Code
$HMAC_k(m) = H(k \oplus c_1 // H(k \oplus c_2 // m))$  (36bb… 5c5c…)
HMAC-SHA-256 (as a PRF)

Diffie-Hellman Key Change: Alice —A := $g^a \bmod p$→ Bob; Bob —B := $g^b \bmod p$→ Alice
$B^a \bmod p = A^b \bmod p$  (p: large prime, g: primitive root modulo (small))

**Confidentiality: Public-key encryption**
Alice encrypts m using Bob's public key to get c
Bob decrypts c using Bob's private key to get m

**Integrity/sender authenticity: Digital signatures**
Alice signs m using Alice's private key to get s
Bob verifies m,s using Alice's public key

**Both properties:** Use both, with two key pairs
Alice encrypts m using Bob's public key to get c, then Alice signs c using Alice's private key to get s
Bob verifies c,s using Alice's public key, then Bob decrypts c using Bob's private key to get m

**Safely encrypting with RSA**
Encrypt a message m to RSA public key (e,N):
Use RSA to encrypt a random x < N, use a KDF to derive a key from x, then encrypt message using a symmetric cipher and key k.
1. Generate random x < N
2. $c_1 := x^e \bmod N$
3. k := KDF(x)
4. $c_2 := \text{AES-GCM}_k(m)$

This is called "hybrid" encryption. Advantages:
- Identical messages yield different ciphertexts
- Don't have to worry about RSA padding
- Don't have to worry about message length

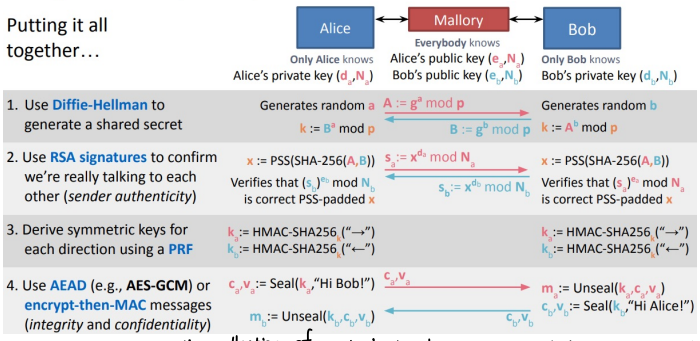
**Safely signing with RSA**
Sign a message m with RSA private key (d,N):
Use RSA to sign a *carefully padded* version of the digest of v. Many gotchas! A good padding scheme is "Probabilistic Signature Scheme" (PSS).
1. v := SHA-256(m)
2. x := PSS(v)
3. $s := x^d \bmod N$

Verifier checks padding:
1. v' := SHA-256(m')
2. $x' := s^e \bmod N$
3. Verify that x' is correct PSS-padded v'

Caution: If you don't correctly verify the padding, attacker can forge signatures w/ Bleichenbacher attack. You'll exploit in Project 1!

In practice, almost always should use

Putting it all together…




The web platform is the collection of tech developed as open standards that powers web sites/apps
Goal: protect users from malicious sites and networks; isolate sites with each other within browser
- integrity: attest session
- confidentiality: steal data

scheme://host:port/path?query#fragment
HTTP: client send: method, path & query, headers; server return: response code, headers, content data
Cookie: maintain session state, personalization, tracking; clients can read/change/erase cookie
Same-Origin Policy (SOP): origin is defined by scheme://domain:port
Protect: cookies, DOM storage & tree, JS namespace, Permission model

**scheme**: protocol used to access resource
- https (HTTP with end-to-end encryption)
- http (plaintext HTTP — unsafe!)

**host**: server's domain name or IP address
- eecs388.org (DNS name)
- 141.212.118.72 (IPv4 address)
- [2607:f018:600:8::20] (IPv6 address)

**port**: TCP port (default 443 for HTTPS and 80 for HTTP)

**path**: identifies resource to server
- /papers/index.html (format up to server)

**query**: parameter string passed to server
- ?key1=value1&key2=value2 (key-value pairs)
- ??7265de6a2c4c8068ed3e75 (arbitrary string)

**fragment**: parameter string visible only to client
- #Section4 (tells browser to scroll to named location)

**Cookie Scope**
Sending cookies only to the right websites is really important!
Frequent source of problems:
Cookies use a different scope than DOM
DOM: Scoped based on (scheme, host, port)
Cookies: Scoped based on ([scheme], domain*)
* Complicated rules shown in later slides

**Cookie Security Attributes**
**Weakness:** By default, cookies set over HTTPS will also be sent on requests over HTTP (and visible to network eavesdroppers).
**Solution:** Server can set "Secure" attribute to limit cookie to HTTPS requests.
`Set-Cookie: id=a3fWa; Secure;`
**Weakness:** By default, cookies can be read by any JavaScript running in the origin.
E.g., if **bank.com** includes Google Analytics script, Google can read authentication cookies.
**Solution:** Server can set "HttpOnly" attribute to prevent cookie from being accessed by DOM
`Set-Cookie: id=a3fWa; Secure; HttpOnly;`

Cross-Site Request Forgery (CSRF): cause the user's browser to perform unwanted actions on a different site on user's behalf
Defenses: referer validation; secret token validation; Samesite cookies
Set-Cookie: AuthToken = X; Secure; Samesite = Lax
Injection: SQL: change meaning of SQL (' OR 1=1 --); Cross-Site Scripting: change HTML and JS
Solution: parameterized SQL (query & args) Separate
XSS: reflected: echoes script back to the same user in the context of the site; stored: store malicious code in a resource managed by server (e.g DB), where it can target users

**Input validation:** Checks all headers, cookies, query strings, form fields, and hidden fields (i.e., all user-controlled parameters that might appear in output) against a rigorous specification of what should be allowed.

**Output escaping:** Encodes all special characters in output to prevent interpretation as code.

**Content Security Policy (CSP)** is a more modern approach that allows sites to eliminate XSS by tightly specifying what scripts are allowed to execute.

parameterized SQL
sql = 'SELECT * FROM users WHERE username = ?'
cursor.execute(sql, [zhli])

TCP: Plaintext transport protocol, Provide semantics: dial, send/receive data, hang up
Doesn't Provide: confidentiality, integrity, authenticity
TLS (Transport Layer Security) is a cryptographic protocol that is layered above TCP to provide a secure channel. HTTP over TLS ⇒ HTTPS
Network Adversaries: Passive: only eavesdrops; Active: can see, inject, modify, or block
→ provide confidentiality & integrity; client can auth server's identity

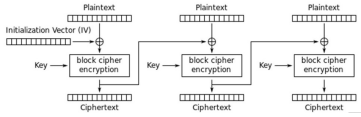
## Cipher-block chaining (CBC) mode

"Chains" ciphertexts to obscure later ones

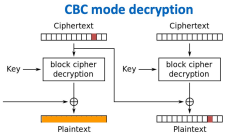
Choose a random **initialization vector IV**

Encrypt: $c_0 := IV$; $c_i := E_k(p_i \oplus c_{i-1})$
Decrypt: $p_i := D_k(c_i) \oplus c_{i-1}$,
[Why do we need the IV?]

Have to send IV with ciphertext
Can't encrypt blocks in parallel or out of order




CBC mode decryption




Malleable: can transform ciphertext into another that decrypts to related plaintext without knowing plaintext.

## Counter (CTR) mode

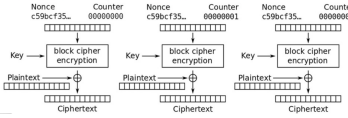
Turns a block cipher into a stream cipher

Generate **keystream** s for k and unique **nonce**:
$s := E_k(\text{nonce} \,//\, 0) \,//\, E_k(\text{nonce} \,//\, 1) \,//\, E_k(\text{nonce} \,//\, 2) \,//\, \dots$

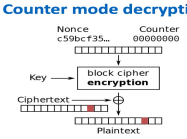
Encrypt: $c := p \oplus s$  Decrypt: $p := c \oplus s$

Benefits: Doesn't require padding
Efficient parallelism/random access

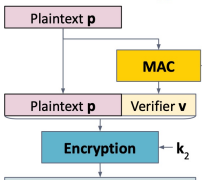
Never Reuse nounce for same k



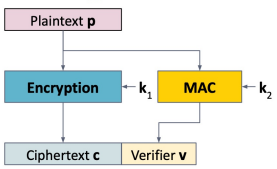

Counter mode decryption



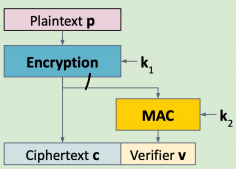

## MAC-then-Encrypt




## Encrypt-and-MAC




## Encrypt-then-MAC




Safest approach

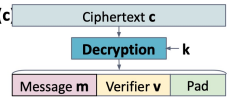
Note: **Use separate keys** for different purposes. (Can derive from a single key using PRG.) [Why?]

[Which approach is safest?] Our encryption methods (so far) only secure against passive eavesdroppers. Only EtM can ensure ciphertext isn't tampered with before decryption.

Common flaw when using **MAC-then-Encrypt**

Suppose an implementation uses **CBC mode**.
Decryption involves the following steps:

1. $m \,//\, v \,//\, pad := D_k(c)$
2. Check that **pad** is valid **PKCS7**, else raise PadError
3. Check that $v = \text{MAC}_k(m)$, else raise MacError



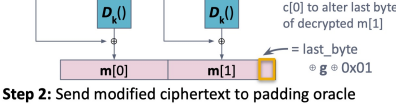

This is how TLS 1.0 worked. Seems reasonable?

*Any method* to distinguish these two error types (even tiny timing differences) **leaks the plaintext!**

**Padding oracle:** attacker submits any ciphertext and learns if last bytes of plaintext are a valid pad

Suppose attacker intercepts **c**, wants to learn **m**.

**Step 1:** Let **g** be a guess for last byte of block **m**[1]:




CBC is malleable: Change last byte of c[0] to alter last byte of decrypted m[1]

**Step 2:** Send modified ciphertext to padding oracle
If g = last_byte: $g \oplus \text{last\_byte} \oplus 0x01 = 0x01$.
Modified plaintext ends in 0x01, so padding's valid; oracle returns MacError
else: Padding is invalid*, oracle returns PadError
(*Except for edge cases: e.g., what if m[1] ends in 0x02 0x01 and g = 0x02?)

**Step 3:** Repeat with g = 0,1, … 255 to learn last_byte. Then use a 0x02, 0x02 pad to learn next byte, etc.

Preferred modern approach:

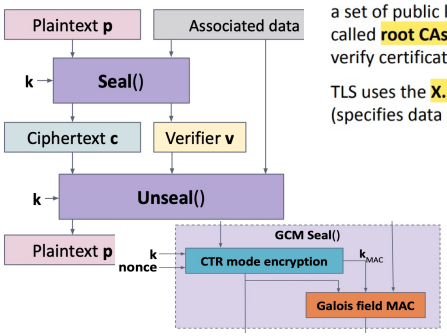
## Authenticated encryption with associated data (AEAD)

AES-GCM  ChaCha20-Poly1305

Integrity and encryption in a single primitive:

**c, v := Seal(k, p, [associated_data])**
encrypts plaintext **p** and returns ciphertext **c** and a verifier **v** (called a "**tag**")

**p, err := Unseal(k, c, v, [associated_data])**
returns **p** or an error if **v** does not match the supplied **c** and **associated_data**




**How does client obtain server's public key?**

Server presents a **certificate**: a message asserting the server's identity and its public key, signed by a **certificate authority (CA)**.

A CA is an entity trusted by clients to **verify server identities** and issue certificates.

Each major platform and browser includes a set of public keys for the CAs that it *trusts*, called **root CAs**. Clients use these keys to verify certificates.

TLS uses the **X.509 certificate format** (specifies data structure, encoding, etc.)

The Web's **public key infrastructure** is operated by a community that includes:

- **Certificate authorities**
  Verify identities, issue certificates, manage revocation
- **Browser/platform developers**
  Implement cert. validation and UI; trust or distrust CAs
- **CA/Browser Forum**
  Consortium that sets industry-wide issuance policies

HTTPS Attacks and Defenses

1. Fooling Users: Homographs: visually similar domains Sol: use punycode to block
Fooling users: Stripping: (HTTP) SSLStrip attack Sol: HTTP Strict Transport Security
Fooling users: Phishing: 钓鱼 Sol: Google Safe Browsing/Phishing-resistant authentication

**Mixed Content** (resources loaded over HTTP from inside an HTTPS page) can be modified or leak information.

Browsers block HTTP scripts but may* allow images to be loaded over HTTP.

**Mitigation:** Use HTTPS for all resources.

By default, **cookies** set over HTTPS will be **sent unencrypted** if the browser later requests an HTTP URL from the domain.

**Mitigation:** Set the **Secure attribute** on cookies, so they will only be sent via HTTPS.

```
Set-Cookie: session=<TOKEN>; Secure
```

**HTTPS reveals certain information about a site.**

TLS: sends certificate chain in plaintext, and domain (but not URL) in plaintext **Server Name Indication (SNI)** header.

Privacy and censorship concerns.

**Mitigation:** Clients can use Tor or VPNs for privacy.

**Mitigation:** An emerging standard for **Encrypted Client Hello** can let servers hide domain.

CA Weakness: fooling validation

**CA identity validation isn't foolproof.**

If attacker can **falsely convince a CA** that they control a domain, they can obtain a certificate and fool browsers.

Example: Getting a cert for `umich.edu`

Attack 1: (Email validation)
CA emails confirmation code to `administrator@umich.edu`.

Attack 2: (HTTP validation)
Attacker becomes MITM between CA and U-M (e.g., by route hijacking) and redirects CA's GET request to own site.

**Mitigation:**
Domains can **limit which CAs** can issue certs for them by creating a DNS record called **Certification Authority Authorization (CAA)**:

```
eecs388.org. IN CAA 0 issue "letsencrypt.org"
```

**Mitigation:**
Sites must prohibit users creating these email addresses: administrator, admin, webmaster, hostmaster, postmaster.

**Mitigation:**
CAs such as Let's Encrypt are introducing **multi-perspective validation**, which makes MITM attacks on the CA more difficult.

Web PKI uses a distributed architecture. Thousands of intermediate CAs can issue certificates for any domain.

Attack on CAs

**Occasionally CAs get hacked.**

In 2011, a cert for `*.google.com` was issued to an attacker who broke into **DigiNotar**, a small Dutch CA.

Cert was used for MITM attacks in Iran.

Nobody noticed the attack until somebody found the certificate in the wild…

Google, Microsoft, Apple and Mozilla distrusted all DigiNotar certs, and the CA ceased operation.

**Mitigation:**
CAs can **revoke** (i.e., cancel) certs they issue by adding them to a **Certificate Revocation List (CRL)** that they sign and publish.

Browser companies crawl CRLs and push some* updates to users.

**Mitigation:**
**Certificate Transparency** helps ensure that compromises will be discovered quickly.

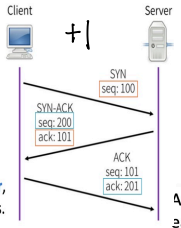
Bugs in TLS

**Mitigation:**

**Formal verification techniques** can be used to check software correctness using a proof.

**miTLS** is a formally verified TLS library.

Every TCP session begins with a **three-way handshake**.




Special packets with **flag bits** set:
1. SYN
2. SYN+ACK
3. ACK

Client and server each pick random 32-bit **initial sequence number**, acknowledge the other's.

**DNS (typically) uses UDP packets.**

**Four sections: questions, answers, authority, additional records**

**Identification:** 16-bit random value (QID) that links response to query

SOP allow scripts from other origins <script> allow sending request but no reading response

Internet Protocol (IP) delivers packets to internet destinations

Packets are in plaintext. May be reordered, lost, corrupted, duplicated or attacked.

***Issue:*** How does a host know what MAC address to use to reach a given IP address?

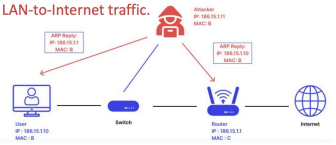
**Address Resolution Protocol (ARP)** is a layer-3 protocol for mapping IP addresses to MAC addresses:

1. Host that needs MAC address M corresponding to IP address A ***broadcasts*** an **ARP packet** to entire LAN asking, "*who has IP address A?*"
2. Host that has IP address A will reply, "*IP address A is at MAC address M.*"
3. Host H caches <IP A: MAC M>

**Since any host on the LAN can send ARP requests and replies, any host can claim to be another host on the local network!**

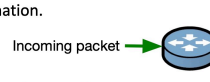
**ARP Spoofing:** Host X can force IP traffic between hosts A and B to flow through X:
- Claim $IP_A$ is at attacker's MAC address $M_X$.
- Claim $IP_B$ is at attacker's MAC address $M_X$.
- Re-send $IP_A$'s traffic to $M_A$, and vice versa.

By **spoofing the gateway**, attacker can MITM all LAN-to-Internet traffic.



A **router** is a device with multiple link layers, each part of a different network.

Examines incoming packet's destination IP and **forwards it** out a link that will get it closer to the destination.




***Issue:*** How does each router know where to send the packet next?

ISPs use **Border Gateway Protocol (BGP)** to **announce** their network prefixes and connectivity to neighbors. Routers compute **route tables** from these announcements.

**BGP has no authentication: Compromised ISP can *announce someone else's network*!**

**BGP hijacking** is common, usually due to operator error, but sometime malicious. Packets to hijacked network are routed to attacker, who can eavesdrop or MiTM.


**Defense: RPKI**
Cryptographic method of signing records that associate a BGP route announcement with the correct originating ISP. *Slow adoption…*

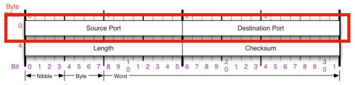
Host can record all packets arriving ("packet sniffing") data format: pcap Wireshark software

Send data to particular app: each app is identified by 16-bit port number (1~65535)

By convention, HTTP:80, HTTPS:443, SMTP:25, DHCP:67, SSH:22, Telnet:23 Root (privilege): <1024

**User Datagram Protocol (UDP)** is a transport-layer protocol that is essentially just a wrapper around IP that adds ports to demultiplex traffic by application.

**UDP header:**




Applications that use UDP:
- DNS
- QUIC
- Many multiplayer games and real-time communications apps

**UDP properties:**
- Connectionless.
- Unreliable data transfer.
- No sequencing or flow control.
- No congestion control.
- **No security.**

UDP is highly vulnerable to **impersonation**: If **off-path** attacker knows what ports a client and server are using, it can inject packets (with spoofed source IP addresses) that appear to be part of the communication.

Using UDP allows/forces applications to provide missing properties at a higher level as needed, which sometimes allows better performance.

**Transmission Control Protocol (TCP)** is a transport-layer protocol providing:
- **Connection-oriented semantics**
- **A reliable, bi-directional *byte stream***
- **Congestion control**

TCP provides these despite IP packets being dropped, re-ordered, and duplicated.

Each segment has 32-bit **sequence number** (where data sent fits in the stream) and **acknowledgement number** (what data is expected next).

Receiver **reassembles** segments as output stream. Sender **retransmits** unacknowledged segments.

**TCP header:**

**TCP does not provide: strong security.**

Data carried in **segments**, each one IP packet.

**DNS compromise allows MITM attacks.**

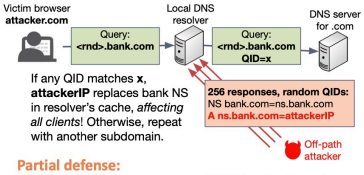
An attacker who subverts DNS can return malicious responses that direct clients to visit attacker's server in place of real server.

Examples of DNS attacks:
- **DNS hijacking**: Attacker hacks into home router, changes DHCP settings to point clients to attacker-controller DNS server…
- **Hosts file hijacking**: Malware edits OS `hosts` file to redirect target sites to malicious server.
- **DNS monitoring** and **DNS blocking**: ISP monitors DNS requests to track users. Injects fake responses to block censored sites.

**Off-path DNS cache poisoning**

Attack discovered by Dan Kaminsky in 2008.

DNS authenticates responses with 16-bit QueryID. Valid responses may include names not requested, if same DNS server controls them ("in bailiwick")…




**Partial defense:**
Randomize UDP source port (16 bits of entropy).
Randomize capitalization in query (1 bit per letter).

Encrypted DNS: DNSSEC (no confidentiality), DNS-over-TLS, DNS-over-HTTPS (doesn't guarantee results authenticated)

Slow adoption by domain & resolver May fall back to UDP (allow downgrade)

The internet: global network that deliver packets between connected hosts

Packet { header: metadata used by network; payload: data to be transported } (Any packet might be dropped)

Each host has a unique identifier (IP address) Routing: A series of routers

Network protocols define how hosts communicate (open standards specified in RFCs)

Protocols are used together via encapsulation HTTPS: HTTP(RFC7231) | TLS(5246) | TCP (793)

Protocol layers

| | |
|---|---|
| Layer 5 **Application** | Defines how individual applications communicate (e.g., HTTP, SSH, DNS) |
| Layer 4 **Transport** | Adds features (ports, connections, encryption) on top of bare packets (e.g., UDP, TCP, TLS, QUIC) |
| Layer 3 **Network** | Gets packets to the final destination, over arbitrarily many hops (e.g., IP) |
| Layer 2 **Link** | Provides a point-to-point link to get packets to next hop (e.g., Ethernet) |
| Layer 1 **Physical** | How bits get translated into electrical, optical, or radio signals |

**IPv4 address**: 32 bits
Written as 4 bytes in form `A.B.C.D` where A,…,D are 8-bit integers in decimal (called "dotted quad")
e.g., 192.168.1.1

**IPv6 address**: 128 bits
Written as 16 bytes in form `AA:BB::XX:YY:ZZ` where AA,…,ZZ are 16-bit integers in hexadecimal and :: implies zero bytes
e.g., 2620:0:e00:b::53 = 2620:0:e00:b:0:0:0:53

**IP packet Spoofing**
If a host sets a fake source address, the **"spoofed" packets** will reach destinations, but responses will be routed to the real hosts.

**Common protocol vulnerabilities:**

**Plaintext transmission**
Passive attackers can eavesdrop on unencrypted communication.

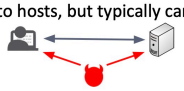
**No source authentication**
The source address on packets you receive can be spoofed, can't trust it.

**No cryptographic integrity**
Protocols don't prevent data modification. (Checksums stop data corruption, not attacks.)

**No built-in bandwidth control**
Attacks can deny service by flooding hosts or the network itself with traffic.

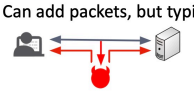
**Network threat models:**

**Off-path attacker. Network participant.** Can talk to hosts, but typically can't see victim's packets.



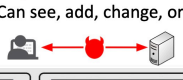
Most attackers, initially

**On-path attacker. Sees *copy* of victim's packets.** Can add packets, but typically **can't** change/block.

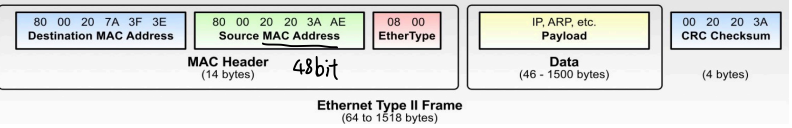


WiFi sniffing or Ethernet passive tap

**In-path attacker. MITM.**
Can see, add, change, or block victim's packets.



Obtain via ARP spoofing, BGP hijacking, DNS attacks




**Denial of service (DoS) attacks** overwhelm a host or network with traffic, such that it can't process legitimate requests

DoS stems from *asymmetry*: especially vulnerable if attacker's cost to make requests is low, victim's cost to process them is high

**DoS Bug:** Design flaw that allows a malicious client to easily disrupt service (e.g., if a malformed request can crash the server)

**DoS Flood:** Attacker sends a large number of messages to exhaust finite resources

**DoS opportunities exist at every layer:**
- **Physical**: e.g., **radio jamming** interfere with transmission medium
- **Link**/**Network**: e.g., **packet flooding** exceed capacity of on-path switches/routers
- **Transport**: e.g., **TCP SYN flooding** open many simultaneous connections, each requiring the server to maintain state
- **Application**: e.g., **HTTP request flooding** cause servers to perform expensive queries or cryptographic operations

**Botnets** are collections of compromised machines (**bots**) under the unified control of an attacker (**botmaster**). Used for DDoS

Passive Monitoring

**Intrusion Detection Systems (IDSes)** are software/devices to monitor network traffic for attacks or policy violations

**Network Telescopes** passively monitor traffic coming into unallocated address space
Used to monitor for Internet-wide scanning and

A **virtual private network (VPN)** creates an encrypted channel that "tunnels" IP packets to a distant network location.

**SYN flood attack:** attacker sends SYN packets from many spoofed source IPs

**SYN cookies** is a defense against SYN flooding that *encodes* the initial connection state in the SYN-ACK packet itself. Invented by D. J. Benstein

**DDoS Defenses:**

**Egress filtering:** To prevent IP source address spoofing, ISPs should drop packets when source IP is outside the local CIDR block
- *Disadvantage*: Requires global coordination. All ISPs need to filter for this to be effective, but little incentive for an ISP to implement.

**Content Delivery Networks (CDNs):** Large CDNs have enough bandwidth to absorb very large DDoS attacks. Security teams work with ISPs to filter DoS traffic upstream

Network Defense Add Encryption
1. **Integrate into apps** (e.g., SSH)
2. **At transport layer** (e.g., TLS)
3. **Below network** (e.g., VPNs)
4. **In the link layer** (e.g. 5G, WPA3)