CS 419: Computer Security

Week 11: Network Security
Securing Communication

Paul Krzyzanowski

Fundamental Layer 2 & 3 Problems

- **IP relies on routing via store-and-forward networking**
  - Network data passes through untrusted hosts
  - Packets can be sniffed (and new forged packets injected)
- **BGP Internet route advertisement protocols are not secure**
  - Routes may be altered to pass data through malicious hosts
- **Ethernet, IP, TCP & UDP**
  - All designed with no authentication or integrity mechanisms
  - No source authentication on IP packets – they might be forged
  - TCP session state can be examined or guessed … and then TCP sessions can be hijacked
- **ARP, DHCP, DNS protocols**
  - Can be spoofed to redirect traffic to malicious hosts
  - Man-in-the-middle attacks are possible

Transport Layer Conversation Isolation:
Transport Layer Security (TLS)

Communication with an insecure network

Cryptography gives us the tools we need to communicate securely

Privacy	Make data unreadable without the key	AES, ChaCha20
Authentication	Validate the endpoints	Public key cryptography
Integrity	Detect modifications	MACs, signatures
Key establishment	Securely agree on secret keys	Diffie-Hellman key exchange

Transport Layer Security

Goal: provide a *transport layer* security protocol

After setup, applications feel like they are using TCP sockets

SSL: Secure Socket Layer

Created with HTTP in mind

- Web sessions should be secure
  - Encrypted, tamperproof, resilient to man-in-the-middle attacks
- Mutual authentication is usually not needed
  - Client needs to identify the server, but the server isn't expected to know all clients
  - Rely on passwords or MFA to authenticate the client after the secure channel is set up

TLS vs. SSL – versions

SSL evolved to TLS (Transport Layer Security)

SSL 3.0 was the last version of SSL
… and is considered insecure

We now use TLS (but is often still called SSL)

- TLS 1.0 = SSL 3.1, TLS 1.1 = SSL 3.2, TLS 1.2 = SSL 3.3
- Latest version = TLS 1.3 = SSL 3.4

Retired versions

- As of the end of 2020, TLS 1.1 & 1.2 (and all older versions) were no longer supported

TLS Goals

Provide authentication (usually one-way), privacy, & data integrity between two applications

Principles

- **Authentication** — *Client should be convinced it is talking with the correct server*
  - Use public key cryptography & **X.509 certificates** for authentication
  - Server side is always authenticated; client optional
- **Data confidentiality** — *Prevent eavesdropping*
  - Use **symmetric cryptography** to encrypt data
  - **Key exchange**: initial keys generated uniquely at the start of each session
- **Data integrity** — *Prevent tampering and man-in-the-middle attacks*
  - Include a **MAC** with transmitted data to ensure message integrity

Most Recent Version: TLS 1.3 Goals

- **Remove support for older ciphers & hashes**
  - Reduce # of acceptable algorithms & parameters
  - Avoid security risk of **downgrade attacks**

Removed support for SHA-1 & MD5 hashes, DEC, 3DES, RC4, AES-CBC encryption, "export-grade" encryption (shorter keys).

- **Require Diffie-Hellman for key exchange**
  - No longer support RSA public keys; we want Perfect Forward Secrecy
- **Reduce handshake complexity**
  - Assume best-case common protocol options
  - Authenticate all data starting from the first response from the server
- **0-RTT: zero round-trip time – rapid connection restart via a pre-shared key**
  - Optionally, support near-instantaneous connection resumption
  - After “hello” phase, both sides generate a Resumption Master Key
  - If connection restarts, send a session ticket & data encrypted with Resumption Master Key
    - Session ticket = data about the session that the server sends to the client which the client returns at the restart
    - Resumption Master Key = derives session-specific keys for encrypting and authenticating the resumed session

TLS Protocol & Ciphers

Two sub-protocols

1. Handshake: authenticate & establish keys

- Authentication
  - X.509 certificates with RSA or Elliptic Curve Digital Signature Algorithm (or pre-shared key)
- Key exchange
  - Ephemeral Diffie-Hellman keys (keys generated for each session)

2. Record protocol: communication

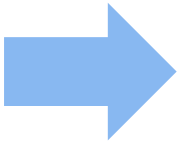
- Data encryption options – *symmetric cryptography*
  - AES-128-GCM, AES-256-GCM, ChaCha20-Poly1305
- Data integrity – *message authentication codes*
  - AEAD – Authenticated Encryption with Additional Data – MAC based on selected encryption
  - HMAC-SHA256, HMAC-SHA384 (not needed if AEAD-based algorithms used for encryption)

TLS 1.3 Basic Handshake

Goals:
1. Agree on a cipher suite
2. Establish trust
3. Agree on a master secret

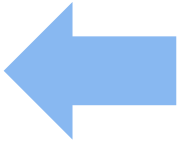
Client

- "Hello"
- Diffie-Hellman public key
- Algorithms/modes



Server

Client



- "Hello"
- Diffie-Hellman public key
- Certificate
- (optional certificate request)
- Proof of private key possession

Server

Both sides now know what algorithms to use & have a D-H common key
Both parties send an HMAC using derived keys to confirm handshake integrity

TLS 1.3 Key Derivation

- **Both sides have a common key after the handshake**
  - Use that to create all the keys we need – client and server can derive the same sets
- **HKDF - HMAC-based Extract-and-Expand Key Derivation Function (RFC5869)**
  - Specification to create any # of keys starting from one secret key
- **Key Derivation Function**
  - Extracts a fixed-length pseudorandom key, PRK, from the initial secret:

    $PRK = hash(non\text{-}secret\text{-}salt, key)$

  - Expands *K* into any number of additional keys

    $Key_0 = null$

    $Key_n = HMAC(PRK, Key_{n-1}, n)$

TLS 1.3 Communication: Confidentiality + Integrity

AEAD: Authenticated Encryption with Associated Data = encryption + MAC

- **Inputs:**
  - Message (record to be sent)
  - Secret key
  - Nonce - Initialization value (IV)
  - Additional Authenticated Data – metadata that's authenticated but not encrypted, like the record type, sequence #, or TLS version.
- **Use a new derived key for encrypting each message**

  Ciphertext, auth_tag = E_{Kn}(nonce, message, AAD)
- **HMAC not needed for AEAD encryption because generating an authentication tag is built into the cipher (AES-GCM, ChaCha2—Poly1305)**
  - This avoids the overhead of encrypting and then computing a MAC

Benefits & Downsides of TLS

Benefits

- Validates the authenticity of the server (if you trust the CA)
- Protects integrity of communications
- Protects the privacy of communications

Downsides

- Longer latency for session setup (only slightly with TLS 1.3)
- Older protocols had weaknesses
  (which is why TLS 1.3 doesn't allow downgrading to weak algorithms)
- Just because a session is over TLS doesn't mean its trustworthy
  - Do you trust the remote side's certificate & that the server hasn't been hacked?

Client authentication Problem

- **TLS supports mutual authentication**
  - Clients can authenticate servers & servers can authenticate clients
- **Client authentication is almost never used**
  - Generating keys & obtaining certificates is not an easy process for users
  - Any site can request the user's certificate – *User will be unaware their anonymity is lost*
  - Moving private keys around can be difficult
    - What about users on shared or public computers?
- **We usually rely on other authentication mechanisms**
  - Usually username and password
  - But there no danger of eavesdropping since the session is encrypted
  - Often use one-time passwords for two-factor authentication if worried about eavesdroppers at physical premises or credential theft (e.g., from the server or phishing attacks)

Some past attacks on TLS

- **Man-in-the-middle: BEAST attack in TLS 1.0**
  - Attacker was able to see Initialization Vector (IV) for CBC and deduce plaintext (because of known HTML headers & cookies)
    - An IV doesn't have to be secret – but it turned out this wasn't a good idea here
  - **Attacker was able to send chosen plaintext & get it encrypted with a known IV**
  - Fixed by using fresh IVs for each new 16K block
- **FREAK**
  - Tricks server into renegotiating a connection with weak RSA encryption keys
- **Man-in-the-middle: crypto renegotiation**
  - Attacker can renegotiate the handshake protocol during the session to disable encryption
  - Proposed fix: have client & server verify info about previous handshakes

Some past attacks on TLS

- **THC-SSL-DoS attack**
  - Attacker initiates a TLS handshake & requests a renegotiation of the encryption key – repeat over & over, using up server resources
- **Heartbleed: vulnerability in popular extension to OpenSSL library**
  - Extension was used to keep the connection alive
    - Client sends payload containing data & the size of the data
    - Server responds with the same message
  - If the client sent false data length, the server would respond with random data
    - That data was memory contents which could include the private key of the server

Network Layer Conversation Isolation:
Virtual Private Networks (VPNs)

Network vs. Transport Layer Secure Communication

- **TLS – Transport layer solution**
  - It allows two applications to communicate via a secure channel
  - The applications have to set up the connection

- **VPNs – Network layer solution**
  - Designed to connect networks together
  - Applications are unaware: all communication across all applications is secure

Solution: Use private networks

Connect multiple geographically-separated private subnetworks together

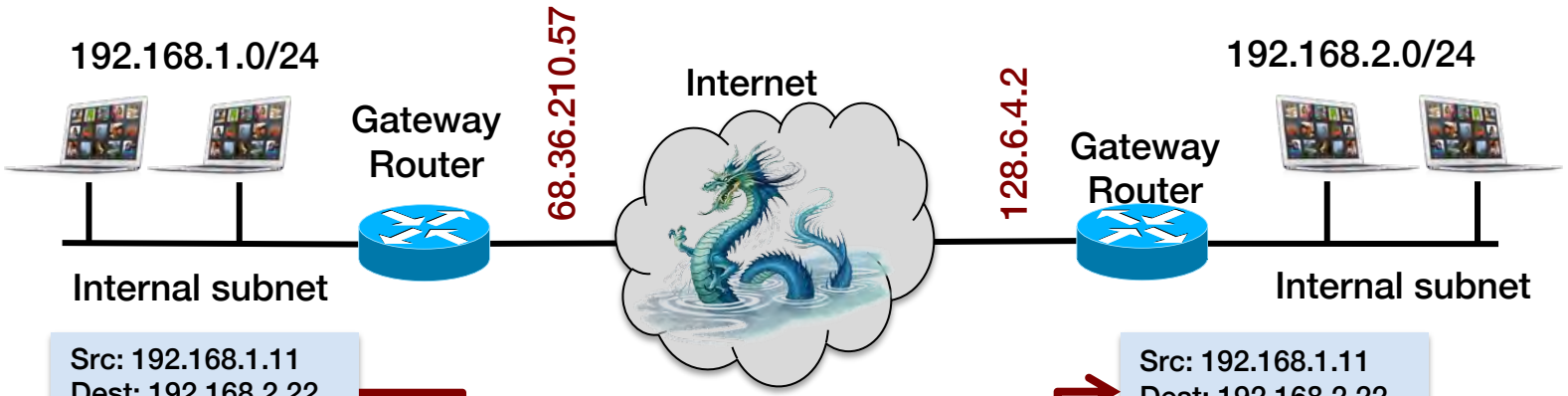
But this is expensive … and not feasible in most cases

(e.g., cost, bandwidth, use of cloud servers)

What's a tunnel?

Tunnel = Packet encapsulation

Treat an entire IP datagram as payload on the public network



Tunnel: IP packets to 192.168.2 are encapsulated as data in messages to the router on the public network at 128.6.4.2

Virtual Private Networks

Take the concept of tunneling

… and safeguard the encapsulated data

- **Add a MAC (message authentication code)**
  - Ensure that outsiders don't modify the data
- **Encrypt the contents**
  - Ensure that outsiders can't read the data

Virtual Private Networks

There are lots of VPN implementations

We'll look at just three popular ones

1. **OpenVPN**
   - Runs in user space leveraging TLS
   - Highly portable across nearly all platforms
2. **IPsec**
   - Implemented in the kernel at the network layer
   - Standardized, widely deployed, complex
3. **WireGuard**
   - Runs in kernel space but communicates via the transport layer (UDP)
   - High speed, low overhead, formally verified

OpenVPN

1st open-source VPN protocol

Step 1: Tunnel setup

- OpenVPN software runs in user space: creates tunnels over TCP or UDP
- A virtual network interface is created to intercept traffic for the VPN
  - Clients can get unique IP addresses
  - Most operating systems provide a TUN (network TUNnel) interface that allows passing IP packets from the kernel to a user process

OpenVPN

Step 2: Key exchange & authentication (Control channel)

- Supports TLS for key exchange and authentication (not transport)
- **Two authentication modes**
  - Pre-shared static keys
    - Four independent keys: HMAC send, HMAC receive, encrypt, decrypt
  - TLS + certificates (most common)
    - Bidirectional authentication: both sides present a certificate
    - Send list of supported ciphers
- Diffie-Hellman used to establish a shared session key

TLS Control channel:

- **Initial TLS handshake**
- **Kept active for the session**
- **Periodic renegotiation of session keys**
- **Keep-alive messages**
- **Termination messages**

OpenVPN

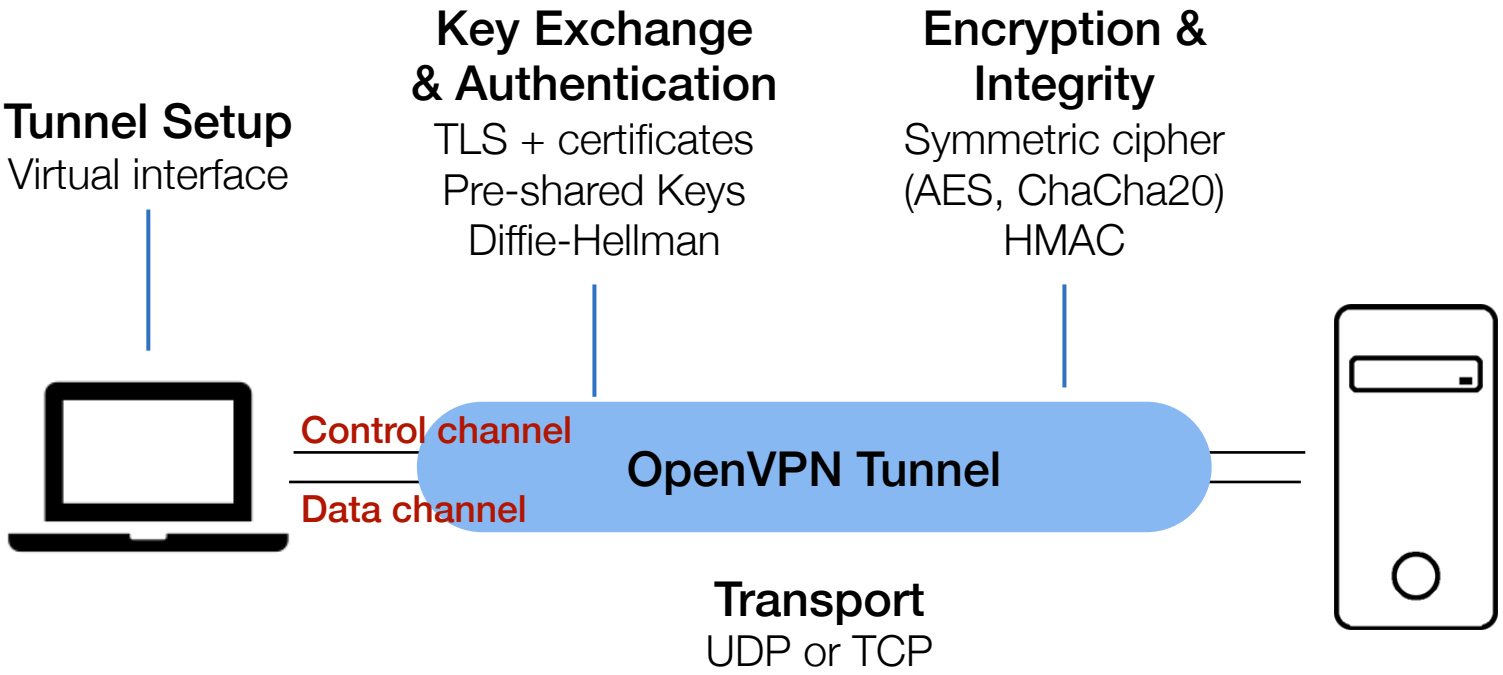
Step 3: Data encryption & Integrity

- Symmetric encryption: common algorithms are AES, ChaCha20
- HMAC for integrity: commonly HMAC-SHA256
- Forward secrecy achieved if using ephemeral keys (non-pre-shared)

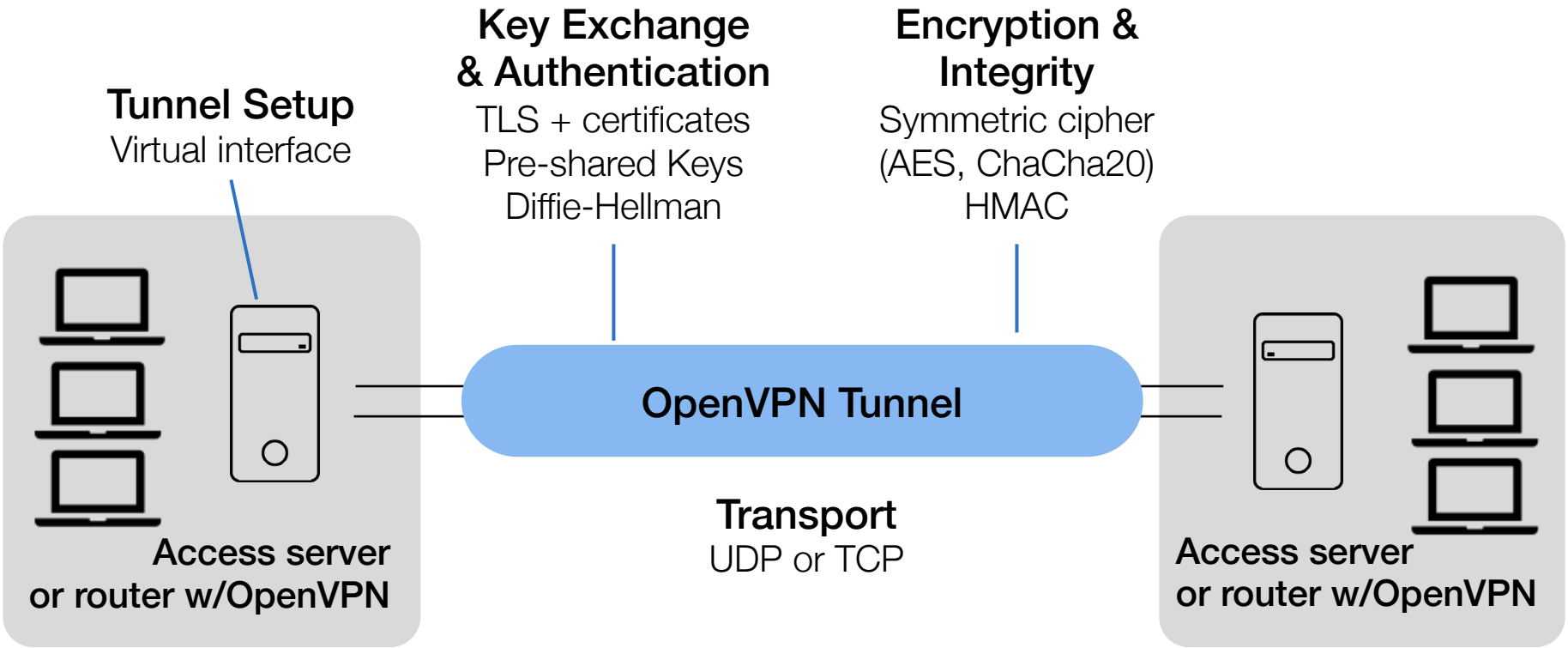
Transport options

- OpenVPN can run over TCP or UDP
  - UDP: great for performance
  - TCP: great for bypassing firewalls

OpenVPN


Tunnel Setup
Virtual interface
Key Exchange & Authentication
TLS + certificates
Pre-shared Keys
Diffie-Hellman
Encryption & Integrity
Symmetric cipher
(AES, ChaCha20)
HMAC
Control channel
Data channel
OpenVPN Tunnel
Transport
UDP or TCP

OpenVPN – Site-to-Site Communication



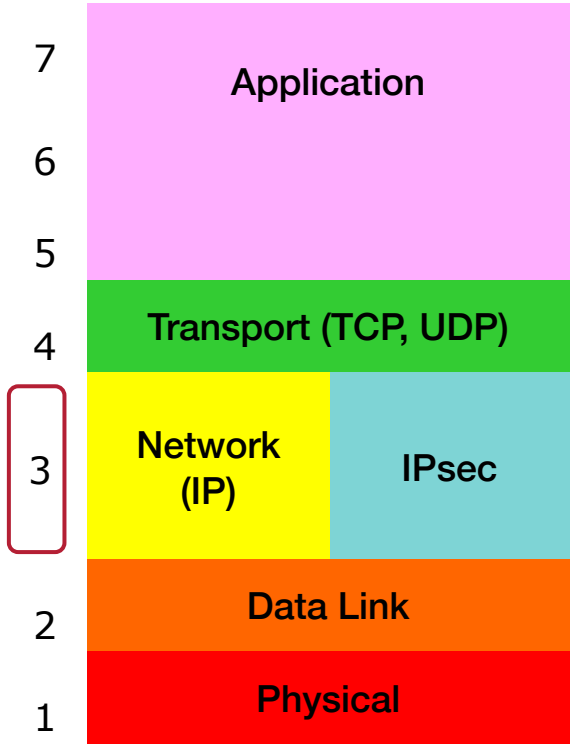
IPsec

Internet Protocol Security

End-to-end security at the IP layer

Two protocols:

- **IP Authentication Header Protocol (AH)**
  - Authentication & integrity of payload and header
  - *Provides integrity*
- **Encapsulating Security Payload (ESP)**
  - AH + encryption of payload
  - *Adds confidentiality*



IPsec is a separate protocol from UDP or TCP – protocols 50 (ESP) & 51 (AH) in the IP header. Layer 3 protocol – gateway routers are responsible for encapsulating/decapsulating

Tunnel mode vs. transport mode

IPsec Tunnel mode

- Communication between gateways: *network-to-network* or *host-to-network*
- The entire IP datagram is encapsulated
  - The system sends IP packets to various addresses on the subnet
  - A router (tunnel endpoint) on the remote side extracts the datagram and routes it on the internal network

IPsec Transport mode

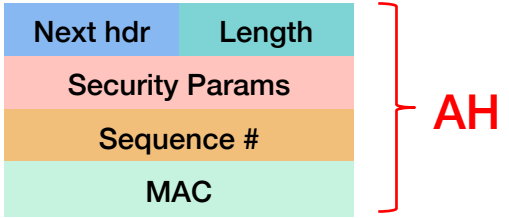
- Communication between hosts
- IP header is not modified
  - The system communicates directly with only one other system

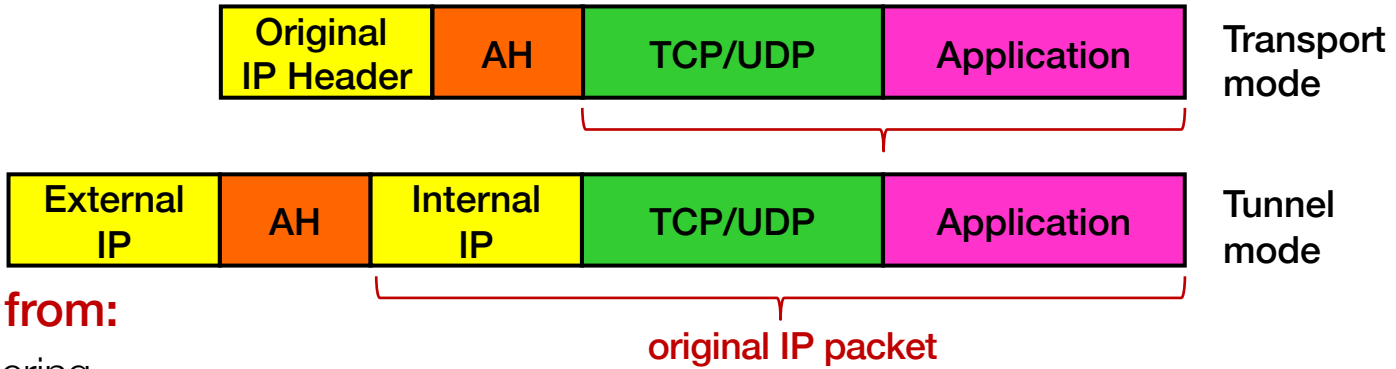
Note: this does not operate at the transport layer – it applies to all IP datagrams between systems or networks, not just a single application

IPsec Authentication Header (AH)

Guarantees integrity & authenticity of IP packets

- MAC for the contents of the entire IP packet
- Computed over unchangeable IP datagram fields (e.g., not TTL or fragmentation fields)





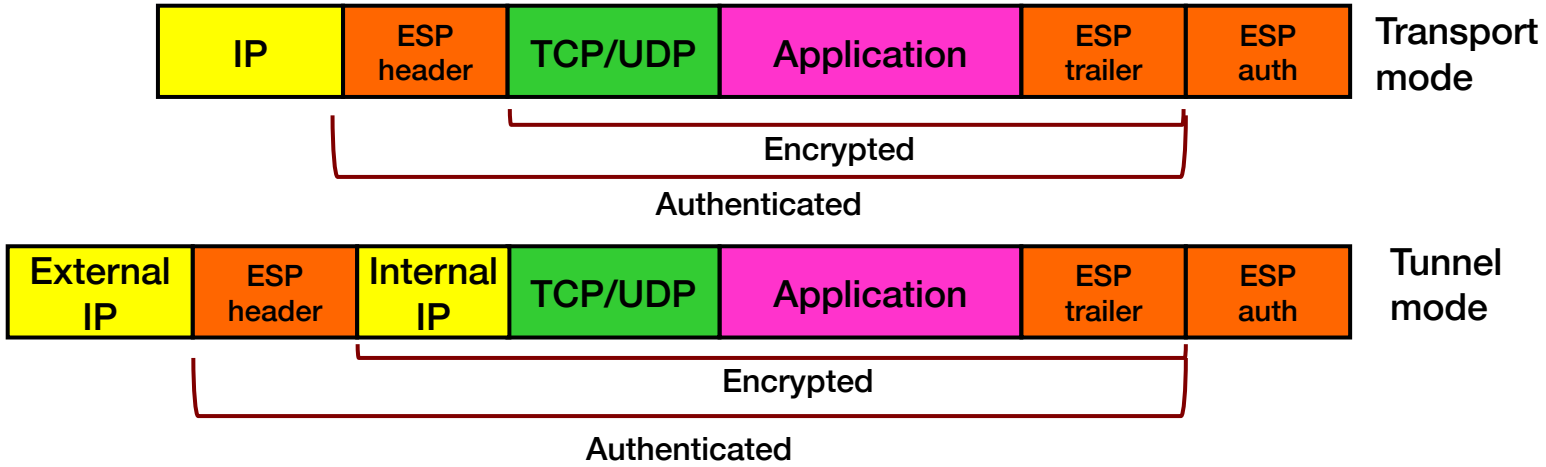
Protects from:

- Tampering
- Forging addresses
- Replay attacks (sequence number in MAC-protected AH)

IPsec Encapsulating Security Payload (ESP)

Encrypts entire payload

– Plus authentication of payload and IP header (everything AH does)
(may be optionally disabled – but you don't want to)



IPsec algorithms

- **Authentication: Certificates or pre-shared key authentication**
  - Public keys in certificates (RSA or ECC) used for authenticating users
    (authenticate by using your private key to decrypt data that was encrypted with the public key in your certificate)
  - Pre-shared keys = authenticate via a shared key that was set up ahead of time
- **Key exchange – Diffie-Hellman**
  - Diffie-Hellman to create a common key for key generation
  - Key lifetimes determine when new keys are regenerated
  - Random key generation ensures Forward Secrecy
- **Confidentiality – symmetric algorithm**
  - 3DES-CBC, AES-CBC, AES-CTR, …
- **Integrity protection & authenticity – MACs**
  - HMAC-SHA1, HMAC-SHA2

WireGuard

- **Simple design – focus on using only the latest algorithms & high performance**
  - Formally validated: codebase is only 4,000 lines of code
- **Setup**
  - Hosts share **public keys** with each other
  - Keys are associated with IP addresses that should be sent via the tunnel
- **Communication initialization (handshake)**
  - **Diffie-Hellman key exchange** to establish shared keys (Elliptic curve algorithm)
  - Re-established every minute to create new keys
- **Data transmission of packets**
  - Encryption: **ChaCha2** stream cipher
  - Message Authentication Code: **Poly1305** *hash*(message, secret)

The End