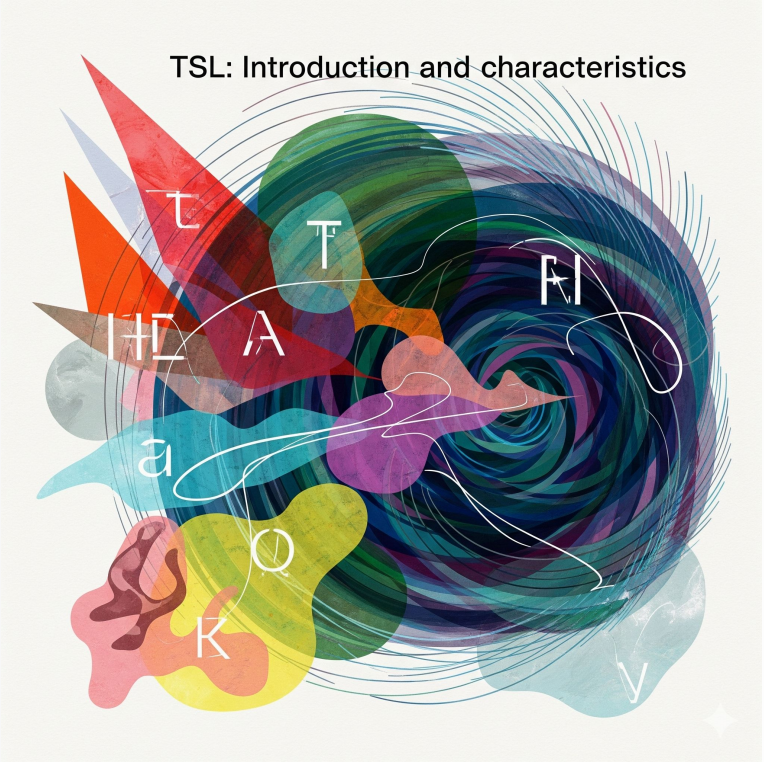
TLS

Introduction e characteristics



The TLS protocol

- Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL), are cryptographic protocols that secure communications over computer networks
- TLS (and formerly SSL) provide end-to-end confidentiality, integrity, and authentication for data exchanged between applications
- TLS typically runs on top of TCP

Main threats addressed

- **Eavesdropping** = interception of communications to gain unauthorized access to data
- **Tampering** = unauthorized modification of transmitted data (integrity violation)
- **Spoofing (impersonation)** = forging messages to mislead the recipient about the sender’s identity
- **Replay attacks** = reusing previously transmitted data to trick the recipient into accepting it as fresh

TLS in the web

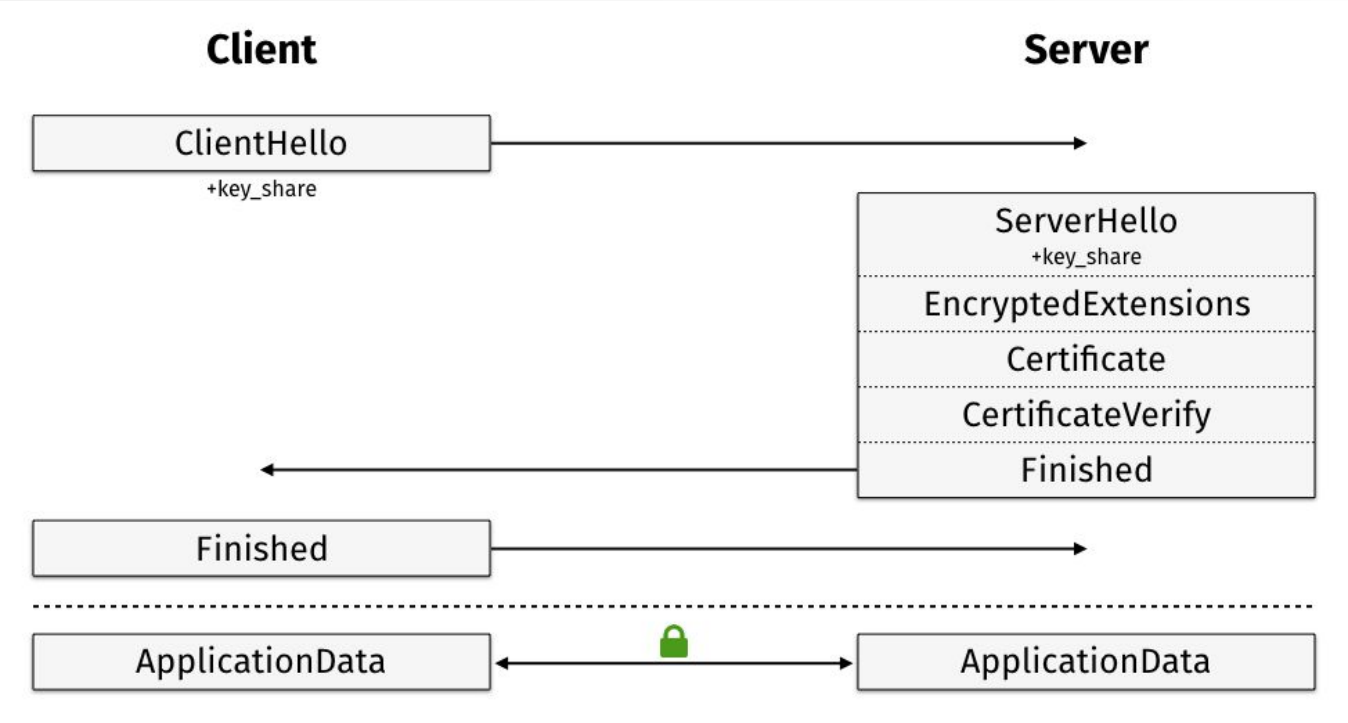
- In typical web browsing, authentication is unilateral: only the server is authenticated via its certificate (the client remains anonymous at the TLS layer)
- TLS also supports mutual authentication, commonly called mTLS: in this mode, both client and server present X.509 certificates
- mTLS is rare on the public web, but widely used in enterprise environments

Phases

TLS involves three main phases

1. Negotiation – peers agree on supported algorithms (cipher suites)
2. Authentication & Key Exchange – server (and optionally client) authentication, establishment of shared keys
3. Secure Communication – symmetric encryption and message authentication for data transfer

TLS 1.3 architecture



ClientHello

1. Protocol Version – indicates the TLS versions supported by the client
2. Random – 32 bytes of randomness used for key derivation
3. Cipher Suites – supported cryptographic algorithms (AEAD ciphers + hash function in TLS 1.3)
4. Extensions, e.g.
    - Supported Versions
    - Key Share (ECDHE public parameters)
    - SNI (Server Name Indication)
    - Signature Algorithms
    - Supported Groups (elliptic curves, DH groups)
    - ALPN (Application-Layer Protocol Negotiation)
    - ... (other optional extensions)

ClientHello example

1. Protocol Version: TLS 1.3
2. Random: 32 bytes
3. Session ID: 0 (unused in TLS 1.3, kept for compatibility)
4. Cipher Suites: [TLS_AES_128_GCM_SHA256, TLS_AES_256_GCM_SHA384, ...]
5. Extensions
    - Supported Versions: [TLS 1.3, TLS 1.2]
    - Key Share: [x25519: key data]
    - SNI: www.example.com
    - Signature Algorithms: [rsa_pss_rsae_sha256, ecdsa_secp256r1_sha256, ...]
    - Supported Groups: [x25519, secp256r1, ...]
    - ALPN: [http/2, http/1.1]

ServerHello

1. Protocol Version – negotiated TLS version
2. Random – 32 bytes of randomness generated by the server
3. Cipher Suite – single cipher suite selected from the client's list
4. Extensions, e.g.:
   - Supported Versions – confirms that TLS 1.3 is the negotiated version
   - Key Share – contains the server's ephemeral public key for (EC)DHE; combined with the client's key share to derive the shared secret
   - ... (other optional extensions)

ServerHello example

1. Protocol Version: TLS 1.2 (with extension indicating TLS 1.3)
2. Random: [32-byte random value]
3. Cipher Suite: TLS_AES_128_GCM_SHA256
4. Extensions
    - Supported Versions: TLS 1.3
    - Key Share: [server's ephemeral public key]
    - Pre-Shared Key (optional): [identifier for PSK]

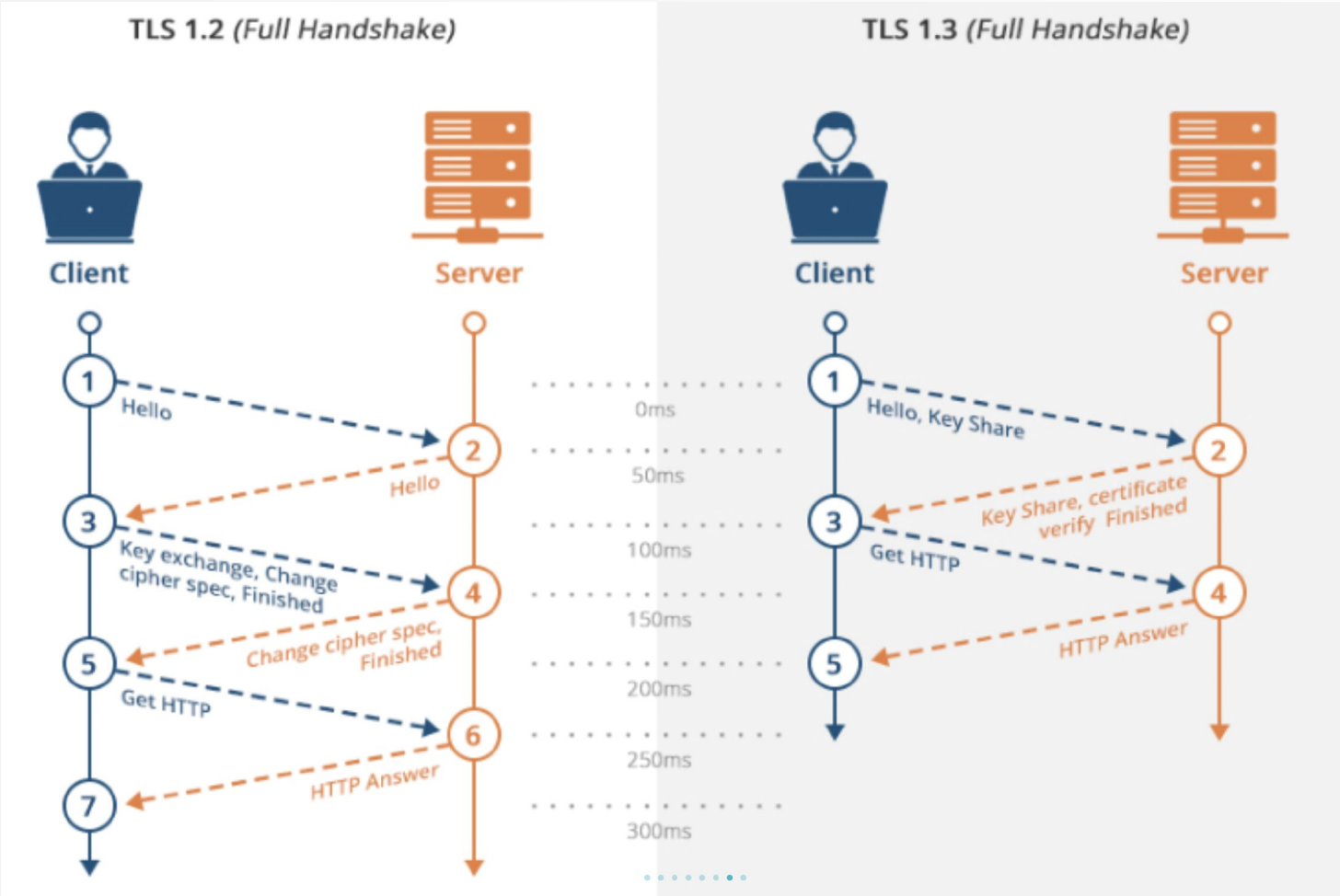
TLS 1.3 Handshake Flows

TLS

1. ClientHello (Client → Server)
2. ServerHello (Server → Client)
3. EncryptedExtensions (Server → Client)
4. Certificate (Server) (Server → Client)
5. CertificateVerify (Server) (Server → Client)
6. Finished (Server) (Server → Client)
7. Finished (Client) (Client → Server)

mTLS

1. ClientHello (Client → Server)
2. ServerHello (Server → Client)
3. EncryptedExtensions (Server → Client)
4. CertificateRequest (Server → Client)
5. Certificate (Server) (Server → Client)
6. CertificateVerify (Server) (Server → Client)
7. Finished (Server) (Server → Client)
8. Certificate (Client) (Client → Server)
9. CertificateVerify (Client) (Client → Server)
10. Finished (Client) (Client → Server)


TLS 1.2 (Full Handshake)
Client
Server
1
Hello
2
Hello
3
Key exchange, Change cipher spec, Finished
4
Change cipher spec, Finished
5
Get HTTP
6
HTTP Answer
7
0ms
50ms
100ms
150ms
200ms
250ms
300ms
TLS 1.3 (Full Handshake)
Client
Server
1
Hello, Key Share
2
Key Share, certificate verify Finished
3
Get HTTP
4
HTTP Answer
5

Core Properties of TLS 1.3

- Forward secrecy. Mandatory (via ephemeral Diffie-Hellman)
- 1-RTT handshake. Faster connection setup
- Simplified ciphersuites: AEAD-only (AES-GCM, ChaCha20-Poly1305), legacy modes removed
- HKDF-based key derivation (standardized, robust KDF)
- Replay protection: handshake protected; 0-RTT data limited/vulnerable

HKDF = HMAC-based KDF

- Global inputs
  - IKM (initial secret, from ECDHE)
  - Salt (optional)
- Step 1: Extract. Compute PRK = HMAC(salt, IKM)
- Step 2: Expand (per key)
  - For each key i, choose $info_i$ (context string identifying the key's purpose) and length L_i
  - Derive OKM_i = HKDF-Expand(PRK, $info_i$, L_i)
- Outputs
  - Independent keys OKM_1, OKM_2, ..., each of length L_i
  - Secure separation ensured by different $info_i$

High-Level Handshake Phases

1. **Parameter negotiation** – agree on version, ciphersuite, extensions
2. **Key exchange** (ECDHE) – derive a shared secret
3. **Authentication** – server certificate; optional client certificate
4. **Key derivation** (HKDF) – compute handshake secrets, session keys, IVs
5. **Replay protection** – ensured by unique nonces and limited 0-RTT

ClientHello

Category	Name	Purpose	Status
Field	CipherSuites	AEAD + hash algorithms	Mandatory
Extension	SupportedVersions	TLS versions offered by client	Mandatory
Extension	KeyShare	Client's ECDHE parameters	Mandatory
Extension	SupportedGroups	Supported EC/ DH groups	Mandatory
Extension	SignatureAlgorithms	Accepted signature for certs	Mandatory
Extension	SNI	Indicates target hostname	Optional
Extension	ALPN	Negotiates application protocol (HTTP/2, HTTP/3)	Optional
Extension	status_request	OCSP stapling	Optional
Extension	early_data	Enables 0-RTT data	Optional
Extension	post_handshake_auth	Allows client auth after handshake	Optional

Extensions vs fields

Extensions

- Defined in the TLS specification and registered at IANA
- Can be added over time (extensible design)
- May be mandatory (TLS 1.3 core) or optional (e.g., ALPN, SNI)
- Ignored by peers that do not understand them

Fields

- Built-in components of the TLS record/handshake structure
- Always mandatory when present in the spec
- Part of the historical “fixed” design of TLS (e.g., CipherSuites list)

ServerHello

Category	Name	Purpose	Status
Field	CipherSuite	Selected AEAD + hash algorithm	Mandatory
Extension	SupportedVersions	TLS version chosen by the server	Mandatory
Extension	KeyShare	Server's ECDHE parameters for key exchange	Mandatory
Extension	SupportedGroups	Confirms group choice (if **HelloRetryRequest** used)	Conditional
Extension	ALPN	Confirms application protocol (e.g., HTTP/2, HTTP/3)	Optional
Extension	SNI	Rarely echoed; mostly server-side only	Optional
Extension	status_request	OCSP stapling response (in Certificate message later)	— (not here)
Extension	early_data	Accepted/rejected indication for 0-RTT	Optional
Extension	post_handshake_auth	Indicates server support for later client auth	Optional

Server Response to ClientHello

- **ClientHello**
  - Includes: CipherSuites, SupportedVersions, KeyShare, Extensions
- Server replies with one of
  - **ServerHello** (normal case)
    - Chooses version, cipher suite, key share
    - Handshake continues directly
  - **HelloRetryRequest** (HRR)
    - Special variant of ServerHello
    - Used if no acceptable KeyShare is found (e.g., unsupported group)
    - Instructs client to resend ClientHello with a new KeyShare
    - Handshake restarts with corrected parameters

Transcript Hash & Binding

- All handshake messages are fed into a running transcript hash
- Each new message is absorbed into the ongoing hash state (incremental update)
- The transcript cryptographically binds negotiated parameters and random values
- Ensures handshake integrity

How the Transcript Hash is Used

- **Key derivation**
  - Input to HKDF final stages (HKDF-Expand)
  - Keys depend on the entire handshake sequence
- **Finished message**
  - Both peers send a MAC over the Transcript Hash
  - Proves handshake integrity and authenticity
- **Downgrade protection**
  - Transcript includes negotiated version and ciphersuite
  - Prevents an attacker from forcing TLS 1.2 or weaker algorithms
- **Replay protection**
  - Transcript binds client and server randoms
  - Ensures uniqueness of each handshake

When Transcript Is Verified

- Transcript Hash is never sent directly
- Verification happens implicitly in the Finished messages
- Server Finished
  - Server sends a MAC over the current Transcript Hash
  - Client recomputes the MAC locally
  - Matching values confirm transcript consistency
- Client Finished
  - Client sends a MAC over the updated Transcript Hash
  - Server recomputes locally
  - Matching values confirm integrity of the full handshake
- Any mismatch handshake fails immediately

Handshake Flow (Server Authentication Only)

- Client → Server
  - ClientHello (with SupportedVersions, CipherSuites, KeyShare, Extensions)
- Server → Client
  - ServerHello (selects version, cipher suite, key share)
  - EncryptedExtensions (negotiated extensions)
  - Certificate (server authentication)
  - CertificateVerify (server proves possession of private key)
  - Finished (MAC over transcript to authenticate handshake)
- Client → Server
  - Finished (client confirms transcript and handshake integrity)
- Result
  - Both sides share the same session keys
  - Client is assured of server's identity
  - Secure channel ready for application data

0-RTT Mode (Optional)

- How it works
  - Client can send early data immediately after ClientHello
  - Keys derived from a PSK (session ticket) from a previous connection
  - Handshake continues; new ephemeral keys are later established (1-RTT)
- Benefits
  - Faster startup, reduced latency (no need to wait for ServerHello)
- Limitations
  - Early data lacks forward secrecy
  - Vulnerable to replay attacks → only safe for idempotent requests (e.g., HTTP GET)
  - Many deployments disable or restrict 0-RTT

Session ticket

- Issued by server in NewSessionTicket (after handshake)
- Opaque to client; identifies a PSK on server side
- Used in next connection via pre_shared_key extension
- Enable session resumption and optional 0-RTT
- Single-use and short-lived (anti-replay, security)

Structure of a NewSessionTicket

- **ticket_lifetime** (4 bytes)
  - Validity of the ticket in seconds
- **ticket_age_add** (4 bytes, random)
  - Random offset used by client to calculate ticket age
  - Mitigates timing-based replay
- **ticket_nonce** (variable length)
  - Unique nonce for deriving the PSK
- **ticket** (opaque blob)
  - Token opque to client, identifies or encodes PSK
  - Can be stateful (ID) or stateless (encrypted PSK)
- **extensions** (optional)
  - e.g., *early_data* with max_early_data_size

Replay

- **1-RTT Handshake**
  - Replay impossible: fresh ECDHE + transcript binding → new keys every session
- **0-RTT Early Data**
  - Replay possible: based only on PSK, no server contribution yet
- **Server-side mitigations**
  - Track PSK ticket use (stateful anti-replay)
  - Apply time-based expiry for early data
  - Optionally enforce single-use tickets
- **Best practice**
  - Restrict 0-RTT to idempotent, safe operations (e.g., HTTP GET)
  - Many deployments disable 0-RTT by default due to replay risk

IPsec

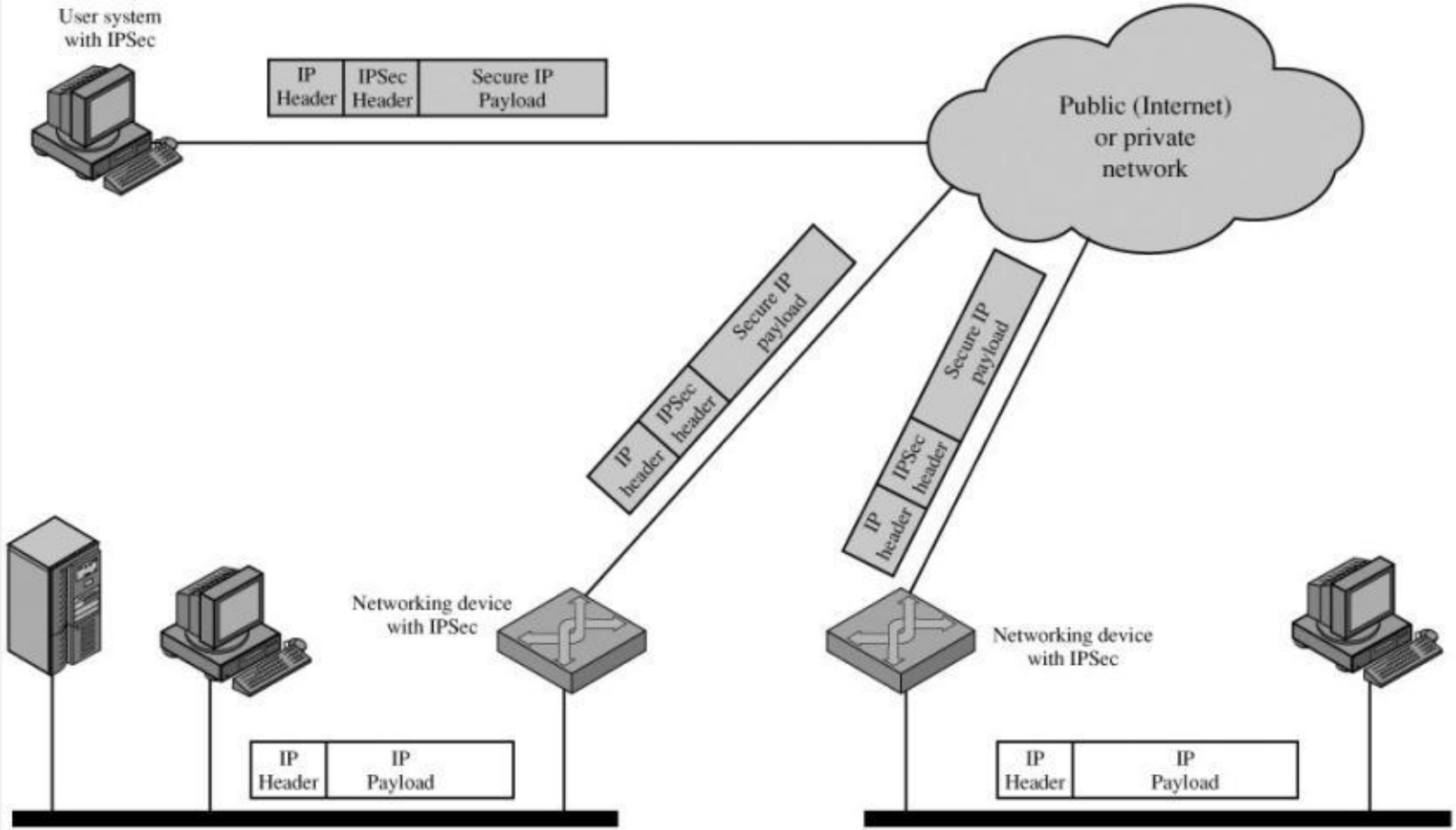
Introduction & architecture

Security on Datagrams

- Goal: Protect IP packets while in transit across networks
- Approach 1: Hop-by-hop encryption (not adopted)
  - Each router decrypts and re-encrypts at every hop
  - Good: even headers could be hidden
  - Bad: complex key management, high router workload
- **Approach 2: IPsec (adopted)**
  - End-to-end or tunnel mode encryption
  - Routers just forward ciphertext, no decryption at each hop
  - Provides confidentiality, integrity, authentication, anti-replay

IPsec

- Provides
  - **Authentication**
  - **Confidentiality**
  - **Integrity**
  - **Key management**
- Can be used over LANs, across public & private WANs, and on the Internet
- Specification is complex and spread across many RFCs
- Standardization status
  - Initially mandatory in IPv6
  - Since 2011 (RFC 6434), optional in IPv6
  - Always optional in IPv4


User system with IPSec
IP Header
IPSec Header
Secure IP Payload
Public (Internet) or private network
IP header
IPSec header
Secure IP payload
IP header
IPSec header
Secure IP payload
Networking device with IPSec
Networking device with IPSec
IP Header
IP Payload
IP Header
IP Payload

Benefits of IPSec

- Security gateways (firewalls/routers) can enforce strong protection on all traffic across the network perimeter
- Hard to bypass, since it operates at the network layer
- Operates below the transport layer, transparent to applications and end users
- Flexible: can also secure traffic for individual hosts when required

Drawbacks of IPsec

- Complex configuration and management
- NAT traversal issues
- Difficult troubleshooting and monitoring
  Traffic is encrypted at the IP level, making debugging and traffic inspection harder
- Protocol overhead and MTU issues
- Limited scalability in large or dynamic environments
- No application-level visibility or authentication
  IPsec protects the network layer but cannot distinguish between legitimate and malicious applications
- Less suited for modern cloud and microservice architectures
  Rigid, IP-based tunnels are less flexible than identity-based security (mTLS, Zero Trust)

Security features

Implemented as additional headers following the IP header

- Authentication Header (**AH**): provides integrity and authentication (no encryption)
- Encapsulating Security Payload (**ESP**): provides encryption, integrity, and authentication

Services provided by AH and ESP

- ESP can operate in two modes: **with** or **without** authentication

	AH	ESP (encryption only)	ESP (encryption plus authentication)
Access control	✔	✔	✔
Connectionless integrity	✔		✔
Data origin authentication	✔		✔
Rejection of replayed packets	✔	✔	✔
Confidentiality		✔	✔
Limited traffic flow confidentiality		✔	✔

Security Associations

- A Security Association (SA) is a unidirectional relationship between sender and receiver that provides security services for IP traffic flows
- An SA consists of a set of security parameters shared between peers
- All active SAs are stored in the Security Association Database (SAD)
- Each IPsec-enabled interface maintains separate inbound and outbound SADs, due to the directional nature of SAs
- Each SA is uniquely identified by three values
  - Security Parameters Index (SPI)
  - IP Destination Address
  - Security Protocol Identifier (AH or ESP)

SA Parameters (1/2)

- Sequence Number Counter
  - 32-bit counter incremented with each protected packet; value placed in the Sequence Number field of AH or ESP headers
- Sequence Counter Overflow
  - Flag indicating whether counter overflow should trigger an auditable event and block further packet transmission under this SA
- Anti-Replay Window
  - Sliding window mechanism used to detect and reject replayed AH or ESP packets

SA Parameters (2/2)

- Authentication algorithm & key (AH/ESP)
  - Specifies the algorithm and key used for data origin authentication and integrity
- Encryption algorithm & key (ESP only)
  - Specifies the algorithm and key used to ensure confidentiality of payload data
- Lifetime
  - Defines validity of the SA (time-based or volume-based)
- Mode
  - Transport or Tunnel
- Path MTU
  - Maximum Transmission Unit relevant for fragmentation handling

Security Policy Database (SPD) and SA Selectors

- The Security Policy Database (SPD) determines whether traffic should be
  - protected with IPsec
  - bypassed (sent in the clear)
- Each SPD entry is defined by a set of IP and upper-layer protocol field values, called **selectors**
- Each entry points to one or more Security Associations (SAs); a many-to-many mapping may exist
- **Selectors** are used to filter traffic and map packets to the correct SA Outbound processing typically follows this sequence
  1. Compare selector fields in the packet against the SPD to find matching entries (pointing to zero or more SAs)
  2. Select the appropriate SA (if any) and its Security Parameters Index (SPI)
  3. Apply the required IPsec processing (e.g., AH or ESP)

SA Selectors

- Destination IP Address
  - Single address, range/list, or wildcard (mask)
- Source IP Address
  - Single address, range/list, or wildcard (mask)
- User ID
  - Identifier from the OS; not an IP/protocol header field, but usable if IPsec runs in the same host as the user
- Data Sensitivity Level
  - For systems with information flow security (e.g., classified vs. unclassified)
- Transport Protocol
  - IPv4/IPv6 protocol number, list, or range
- Source and Destination Ports
  - TCP/UDP port values, list, or wildcard

SAD vs SPD

- Security Policy Database (**SPD**)
  - Defines the policies that determine how IP traffic should be handled
    - protected with IPsec
    - bypassed
- Security Association Database (**SAD**)
  - Stores the parameters of all active Security Associations (SAs), including keys, algorithms, sequence numbers, and lifetimes

IPsec protocols and modes

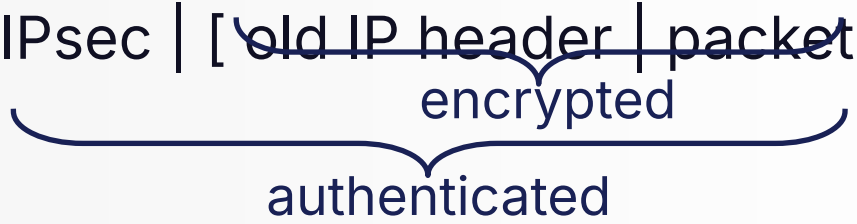
- Protocols used by IPsec
  - Authentication Header (AH): provides integrity and data origin authentication (no encryption)
  - Encapsulating Security Payload (ESP): provides confidentiality, and optionally integrity and authentication
- Modes of operation
  - Transport mode
  - Tunnel mode

Transport Mode – Summary

- **Transport mode**: original IP header not touched; IPsec information added between IP header and packet body
  - IP header | IPsec | [packet]
    protected
  - Typically used for end-to-end communication between two hosts

Tunnel Mode – Summary

- **Tunnel mode**: keep original IP packet intact but protect it; add new header information outside
  - New IP header | IPsec | [~~old IP header | packet~~]



  - Can be used when IPsec is applied at intermediate point along path (e.g., for firewall-to-firewall traffic)
  - Treat the link as a secure tunnel

Transport & tunnel modes

Tunnel & Transport Mode Functionality

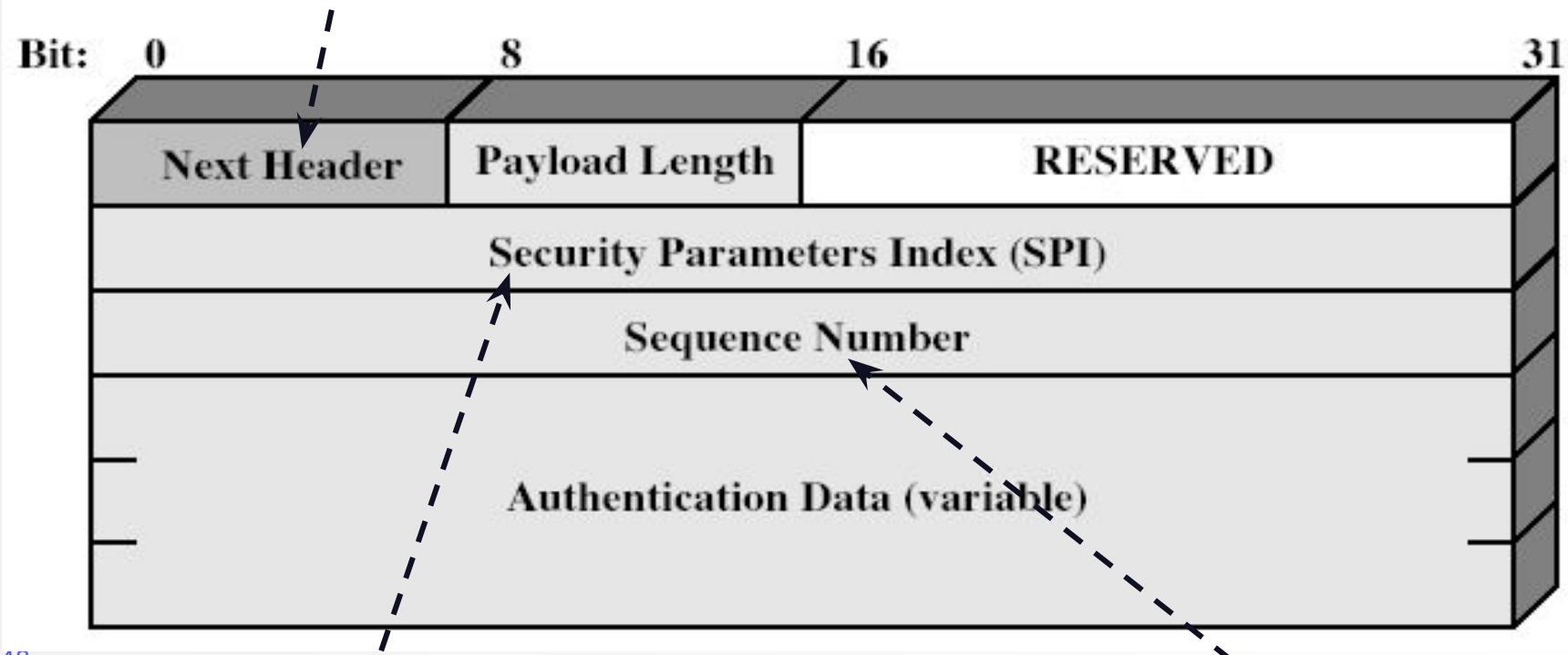
	Transport Mode SA	Tunnel Mode SA
AH	Authenticates IP payload and selected portions of IP header and IPv6 extension headers	Authenticates entire inner IP packet (inner header plus IP payload) plus selected portions of outer IP header and outer IPv6 extension headers
ESP	Encrypts IP payload and any IPv6 extension headers following the ESP header	Encrypts entire inner IP packet
ESP with Authentication	Encrypts IP payload and any IPv6 extension headers following the ESP header. Authenticates IP payload but not IP header	Encrypts entire inner IP packet. Authenticates inner IP packet

Authentication Header (AH)

- Ensures data integrity and authentication of IP packets
  - End system or router can authenticate the sender
  - Protects against **replay attacks** by using sequence numbers
- Does not provide confidentiality (no encryption)
- Relies on a MAC
- Requires that communicating parties share a secret key

Authentication Header (AH)

higher level protocol, e.g., TCP

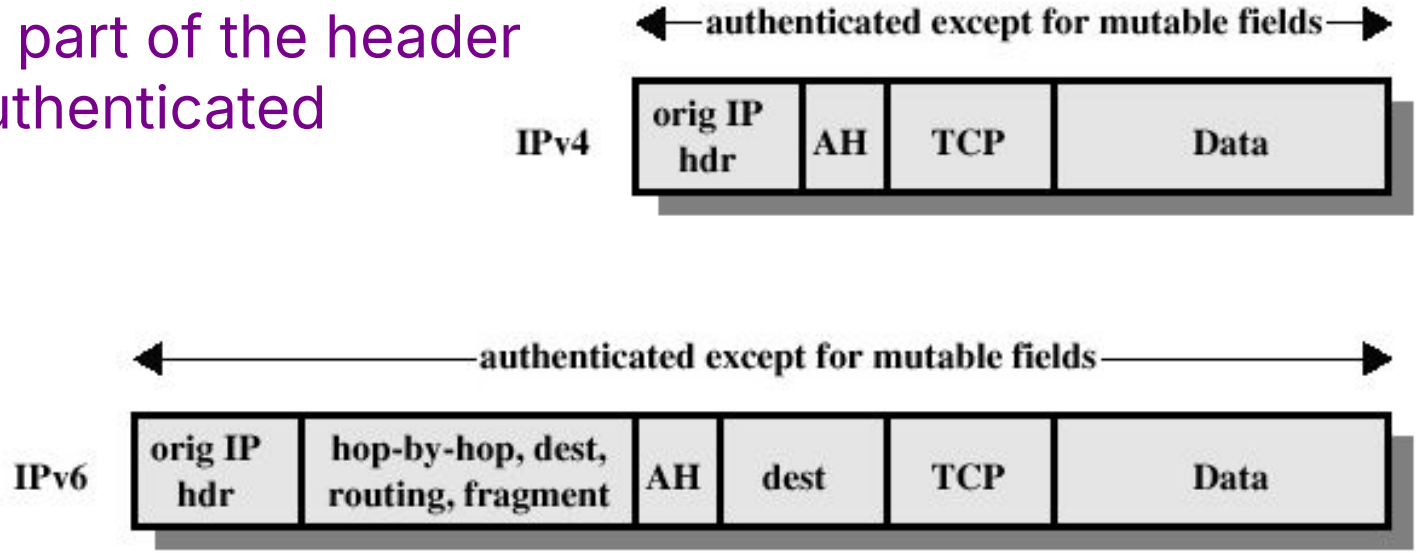


identifies the active SA

not restarting

AH in transport mode

only part of the header is authenticated



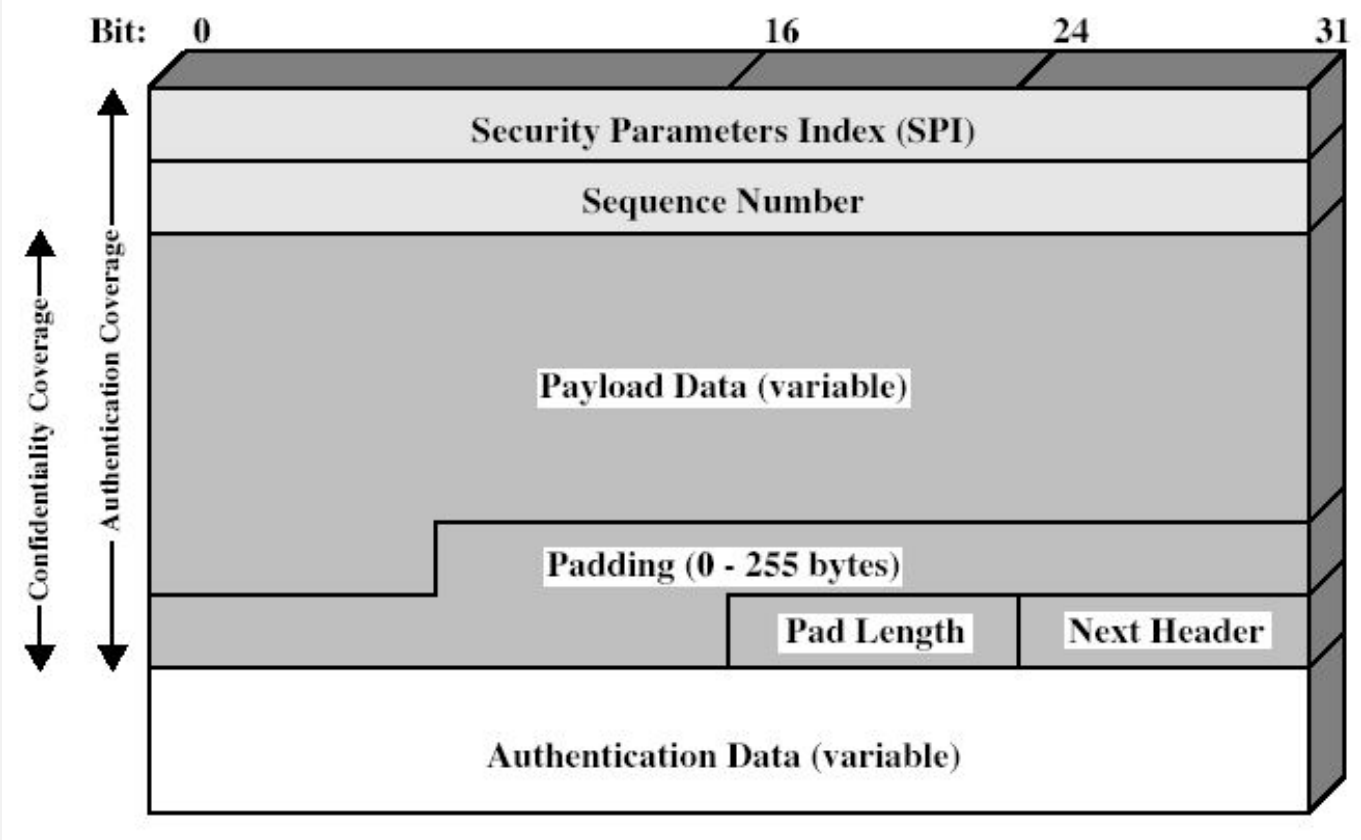
AH in tunnel mode

only part of the header
is authenticated

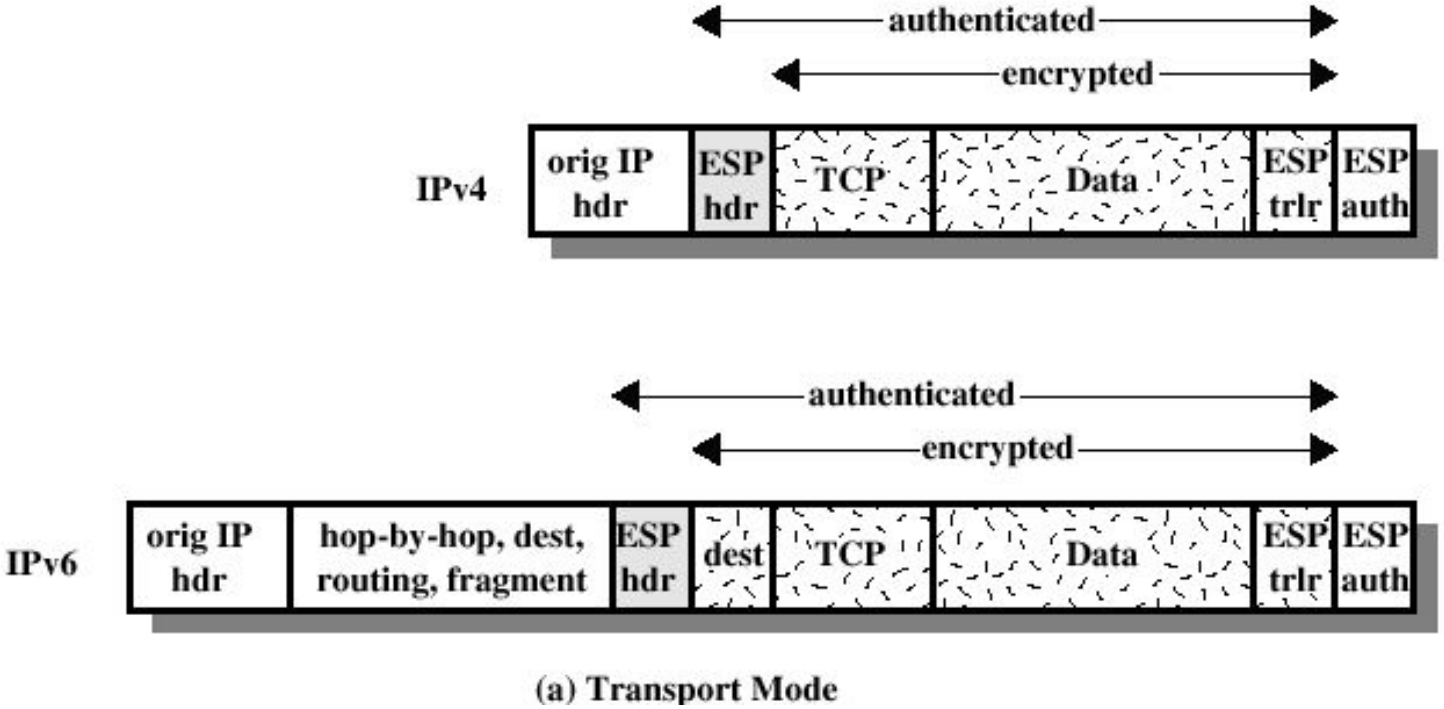
Encapsulating Security Payload (ESP)

- Provides confidentiality of message content and limited traffic flow confidentiality
- Can optionally provide authentication and integrity (same services as AH)
- Supports a variety of encryption algorithms and modes
  - AES, 3DES, Blowfish, etc.
  - CBC mode most common (historically), newer standards prefer AES-GCM (combined mode)
- Uses padding to align data to the block size of the cipher
- Integrity protection (if enabled) with HMAC, like AH

Encapsulating Security Payload

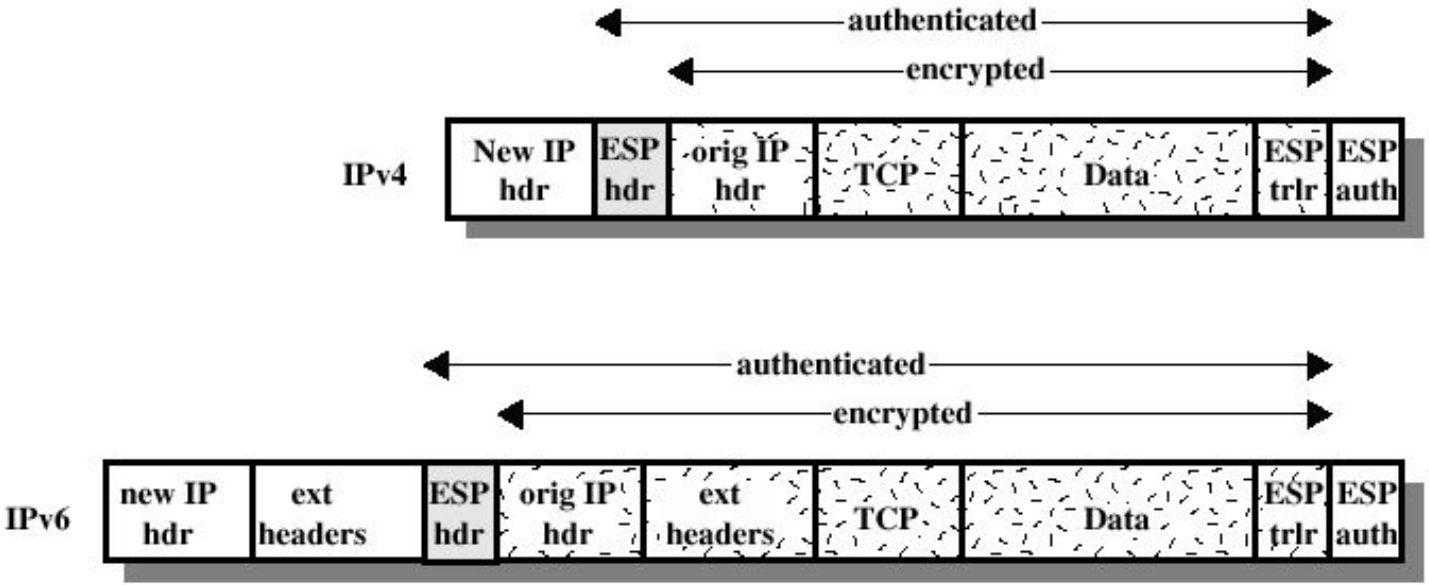


ESP in Transport Mode: Encryption and Authentication



(a) Transport Mode

ESP in Tunnel Mode: Encryption and Authentication



(b) Tunnel Mode

Transport vs Tunnel Mode ESP

- Transport Mode
  - Encrypts and optionally authenticates only the payload (TCP/UDP header + data)
  - IP header remains in clear (visible to adversary)
    - Vulnerable to traffic analysis
  - Best for host-to-host communication
- Tunnel Mode
  - Encrypts the entire original IP packet (header + payload)
  - Adds a new outer IP header for routing to the next hop
  - More overhead, slightly slower
  - Primary mode for VPN tunnel deployments

Key Management in IPsec: IKE

- IKE provides automated key management
- Negotiates cryptographic algorithms
- Establishes and refreshes Security Associations (SAs)
- Uses Diffie–Hellman for shared secret derivation
- Supports Perfect Forward Secrecy
- Handles rekeying and SA expiration

What is IKE

- Protocol framework working with IPsec to establish secure communication
- Authenticates peers using PSK, certificates, or other
- Establishes shared secret keys, typically via DH or ECDH to provide Perfect Forward Secrecy
- Can also rely on pre-shared keys without DH, but with weaker security
- Ensures confidentiality, integrity, and authenticity of IP traffic

Why IKE

- Automates Security Association (SA) creation, avoiding manual keying
- Provides secure negotiation of algorithms and keys between peers
- Enables strong peer authentication, integrated with IPsec
- Scales efficiently to large and diverse networks

IKEv2 structure

- The handshake between peers is composed of four exchanges
  - **IKE_SA_INIT**: negotiation of algorithms, Diffie–Hellman exchange, provisional IKE SA
  - **IKE_AUTH**: peer authentication, consolidation of the IKE SA, creation of the first Child SA
  - **CREATE_CHILD_SA**: additional Child SAs or rekeying of existing ones
  - **INFORMATIONAL**: error reporting, SA deletion, keep-alive messages
- Each exchange usually involves 2 messages (request/response), except EAP-based authentication which may require more

IKEv2 phases explained

- **IKE_SA_INIT**
  - Agreement on algorithms and exchange of nonces/DH values
- **IKE_AUTH**
  - Authentication of peers (PSK, certificates, or EAP)
  - Consolidation of the IKE SA and creation of the first Child SA
- **CREATE_CHILD_SA (optional)**
  - Multiple Child SAs can coexist under the same IKE SA
- Each Child SA is effectively bidirectional, implemented as a pair of unidirectional SAs (inbound and outbound)

IKE_SA_INIT Exchange

1. **Initiator → Responder**
   - HDR: IKE header with Initiator SPI, Responder SPI = 0, exchange type, flags
   - SAi1: Initiator's proposals (encryption, integrity, PRF, DH group)
   - KEi: Diffie–Hellman value from the Initiator
   - Ni: Initiator nonce (random value)
2. **Responder → Initiator**
   - HDR: IKE header with both SPIs set
   - SAr1: Responder's chosen algorithms from the proposals
   - KEr: Diffie–Hellman value from the Responder
   - Nr: Responder nonce (random value)
   - [CERTREQ] (optional): certificate request

- **Purpose**
  - Negotiate cryptographic algorithms
  - Exchange DH values and nonces
  - Derive the shared secret and create a provisional IKE SA

IKE_AUTH Exchange

1. Initiator → Responder
   - HDR: IKE header with both SPIs set
   - IDi: Initiator Identity (IP address, FQDN, or other identifier)
   - AUTH: Authentication payload (PSK, certificate-based signature, or EAP method)
   - [CERT, CERTREQ] (optional): certificates and certificate requests
   - SAi2: Proposal for the first Child SA (ESP or AH parameters)
   - TSi: Traffic Selectors (initiator's view of protected traffic)
   - TSr: Traffic Selectors (responder's view of protected traffic)
2. Responder → Initiator
   - HDR: IKE header with both SPIs
   - IDr: Responder Identity
   - AUTH: Authentication payload of the responder
   - [CERT] (optional): responder's certificate(s)
   - SAr2: Accepted Child SA proposal
   - TSi/TSr: Final traffic selectors agreed for the Child SA

- Purpose
  - Authenticate both peers (initiator and responder)
  - Consolidate the IKE SA (it becomes fully established)
  - Negotiate and create the first Child SA for IPsec traffic

CREATE_CHILD_SA Exchange

1. Initiator → Responder
   - HDR: IKE header with existing IKE SA SPIs
   - SA: Proposal for the new Child SA (algorithms, mode, lifetimes)
   - Ni: Nonce generated by the initiator
   - [KEi] (optional): Diffie–Hellman value, if a new DH exchange is required (for Perfect Forward Secrecy)
   - TSi: Traffic Selectors proposed by initiator
   - TSr: Traffic Selectors proposed for responder
2. Responder → Initiator
   - HDR: IKE header with same SPIs
   - SA: Accepted proposal for the Child SA
   - Nr: Nonce generated by the responder
   - [KEr] (optional): Responder's Diffie–Hellman value
   - TSi/TSr: Final traffic selectors chosen by responder

- Purpose
  - Create additional Child SAs under the same IKE SA
  - Perform rekeying of an existing Child SA or the IKE SA itself
  - Allow traffic with new parameters, selectors, or lifetimes without renegotiating the IKE SA

SAs through IKE

- IKE SA (Control Plane)
  - Secures IKE signaling and negotiation
- Child SA(s) (Data Plane)
  - Protect data traffic using ESP or AH
  - All Child SAs are created, updated, and deleted by a single IKE SA
- SAD (Security Association Database)
  - Stores IKE SA and all Child SAs
  - Contains
    - keys
    - algorithms
    - lifetimes
    - sequence numbers

IPsec with IKE

1. Complementary roles
   - IPsec (AH/ESP) provides packet security
   - IKE manages negotiation and keys
2. Transparency
   - Works below transport and application layers
3. Deployment flexibility
   - Host-to-host, gateway-to-gateway, host-to-gateway

Supported use cases