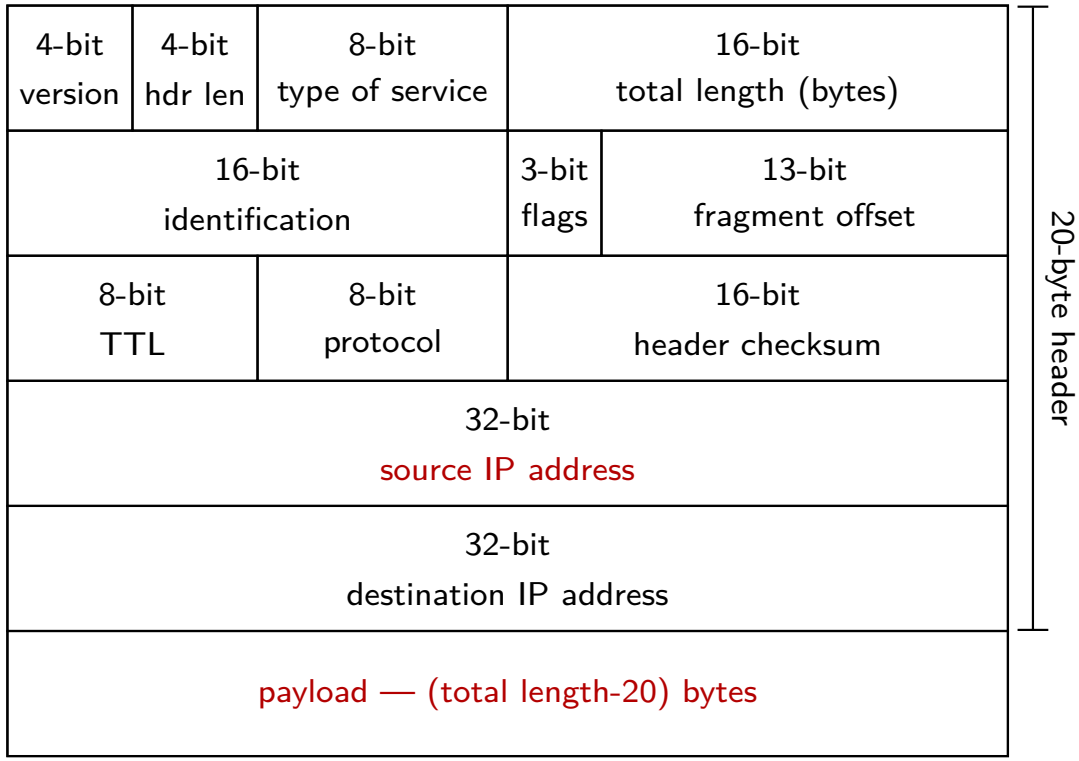
Internet Protocol and Transmission Control Protocol

CMSC 414

November 13, 2017

Internet Protcol

Recall:



No authentication

- Trusting sender to construct this correctly
- No defense against active/passive wiretappers

IP Source Spoofing

As with layer 2, attacker can set source IP address to *anything*

Enables some bad behavior:

- Hide the source of packet flood
- Get target to "respond" to a victim (reflection attacks)
- *Inject* traffic into an ongoing session

Egress Filtering

How can we protect against source spoofing?

Ingress When packets enter your network

Egress When packets leave your network

We can't enumerate all legitimate source for packet ingress
$\Rightarrow$ Packets can come from anywhere in the network

We *can* for packet egress
$\Rightarrow$ Packets coming from our network should be in a known range

Networks can implement **Egress Filtering** to prevent internal hosts from spoofing their source addresses

Requires *extra work* from network for *little/no benefit* to it
$\Rightarrow$ *Not widely implemented*

Eavesdropping

IP provides *no confidentiality*

Any router on path from src $\rightarrow$ dest can read all traffic

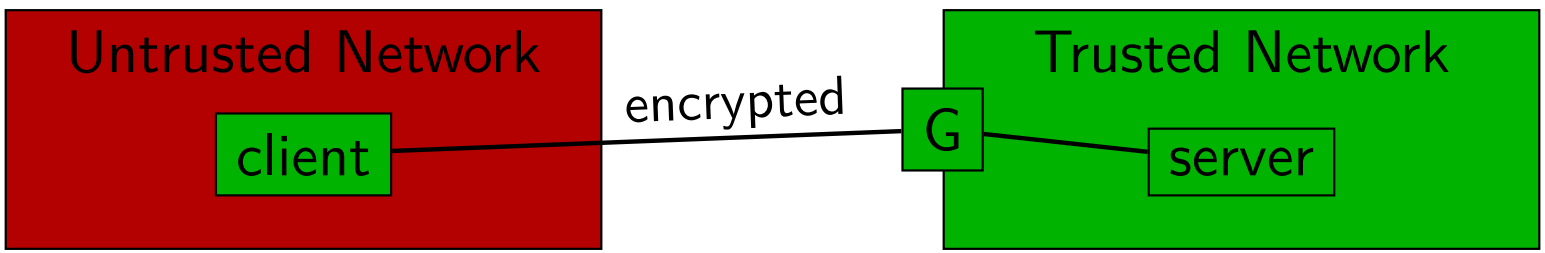
Other hosts on same subnet as src/dest/rtr might also see traffic

Want to prevent eavesdropping?
$\Rightarrow$ Secure *IP tunnels* over IP

Virtual Private Network

Create an *overlay* IP network

- Looks like Layer 3 to Layers 4 and above
- But...sits *on top of* Layer 3!
- Provides confidentiality/integrity



Client & Gateway create a *tunnel*

⇒ End-to-end encryption and authentication

⇒ Client appears to be *inside* trusted network

Lots of ways to do this, at different layers

⇒ Most popular is *IPsec*

IPsec

Two modes: *transport* and *tunnel*
$\Rightarrow$ We'll only worry about tunnel mode

Encapsulates entire IP packet in another IP packet

Consider two hosts, A and B, on different physical networks

- Want to create VPN that includes both
- Each has *two* IP addresses: IP_x^{pub} and IP_x^{priv}

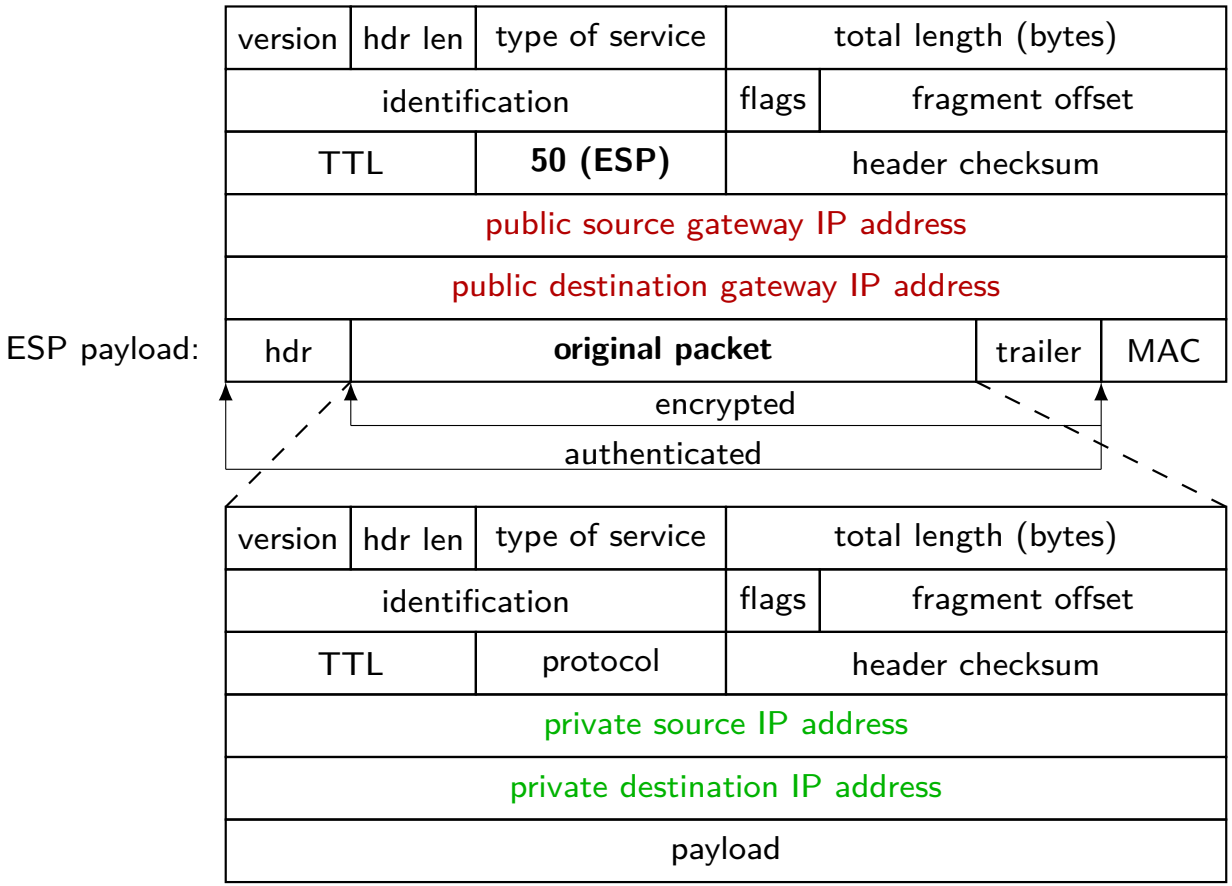
Public IP addresses go directly to Internet

Private IP addresses go through the VPN tunnel

Tunneled connections must go through a **gateway**
$\Rightarrow$ The gateway might be the host itself

IPsec Encapsulation

Encapsulation Security Payload (ESP) protocol



ESP header includes an identifier for a **Security Association**

Security Associations

What's in a **Security Association** (SA)?

- 32-bit identifier
- source interface (public gateway IP addr)
- destination interface (public gateway IP addr)
- encryption algorithm + mode
- encryption key
- integrity algorithm
- authentication key

An SA is **unidirectional** $\Rightarrow$ Need 2 for bidirectional comms

These can be hard-coded, or established dynamically with the **Internet Key Exchange** (IKE) protocol (over UDP)

Group Exercise 1

Use `get-assignment` to fork the `spoof` repository, and look at the README file. We're going to learn how to spoof source addresses!

If we're lucky we might even trip an intrusion detection system!

TCP Overview

Transmission Control Protocol provides *reliable* communications between **processes** on hosts, identified by **TCP ports**

Treats application data as a *byte stream*; tries to guarantee that

- All bytes are delivered to the destination
- Bytes are delivered *in order*
- Bytes are *unmodified*

Must detect *dropped* data, and *retransmit* it

Also provides

Flow Control Destination tells source not to send faster than it can read

Congestion Control Source determines current capacity of the network, so that it doesn't overload it

Without these, we get lots of dropped packets

Three-Way Handshake

How we establish a TCP connection

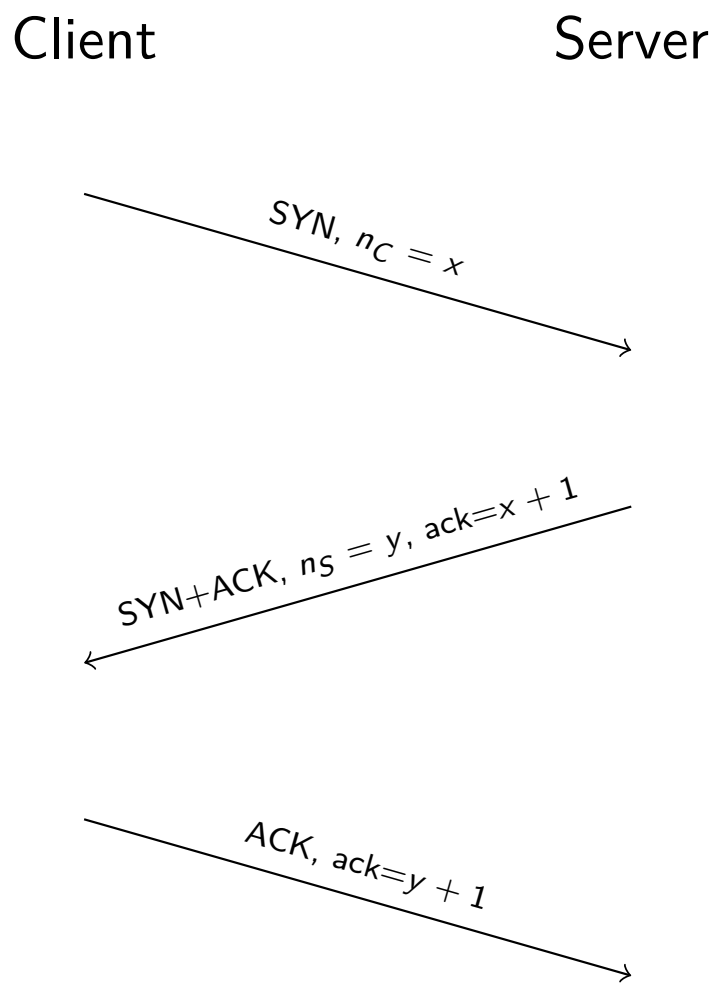
Client and server set their initial **sequence numbers**

SYN message specifies sequence number in one direction

ACK message signals acceptance of this number

Both are header options, and can be combined in a single packet

You've probably seen this in wireshark



SYN Flood

After receiving a SYN packet, receiver sets up *state* for the connection and sends SYN+ACK

No ACK from initiator $\Rightarrow$ re-transmit SYN+ACK

Attacker can initiate and never send final ACK

Do this a bunch of times

$\Rightarrow$ *Memory exhaustion* at server, prevents other connections

Hey, easy $\Rightarrow$ Block SYNs from source IP that isn't ACK'ing!

Attacker can *spoof* the source IP (as we've seen)

If there's a host at that IP, it might send a **TCP RST** message

$\Rightarrow$ Tears down the connection, frees state

$\Rightarrow$ Choose unused (or unresponsive) IP addrs

SYN Cookies

TCP connection state includes

- Remote IP addr
- Remote port
- *Maximum Segment Size* (MSS)

Use 32-bit sequence number as connection state, with

- 5-bit slow-moving timestamp (to prevent replays)
- 3-bit encoding of MSS
- 24-bit cryptographic hash of IP, port, and MSS

Prevents SYN floods, but adds its own issues

$\Rightarrow$ Trade-offs determine whether to bother using this

TCP Injection

TCP not authenticated

Any router on the path can see current sequence numbers

- Easy to predict next number
- Can inject packets of its choice into the stream

If not on the path, can still forge TCP packets

- Have to guess next number
- These used to be deterministic!

If we can get into the stream, we can:

- Send RST messages — kill the connection
- Inject data into a connection (aka *TCP veto*) — dest thinks it already has bytes, rejects as duplicates
- *Conduct one side of a conversation without seeing the other*

Mitnick Attack

SYN flood a trusted host that doesn't encrypt/authenticate traffic

Send SYN to target server, claiming to be trusted host
(source spoofing)

SYN+ACK sent to DoS'ed host, which won't respond

Attacker can predict seq num in SYN+ACK, and send its own ACK

Now able to send data with spoofed source IP

Requires predictable data exchanges

Might be able to install a backdoor!

The lesson: **Make initial sequence numbers unpredictable!**

Optimistic ACK

TCP uses ACKs to indicate that data bytes were received

Lack of ACK $\Rightarrow$ Packet loss, need to retransmit

Have to wait for ACKs before we can send more data (simplification)

Often leads to under-usage of available capacity
$\Rightarrow$ Send **optimistic ACKs**

If we think more capacity available than used, can ACK bytes we think are *in flight*, but not yet *delivered*
$\Rightarrow$ Sender will send more bytes sooner, and faster

When we see packet loss, we back off and let things equilibrate

Part of normal *congestion control* process

Opt-ACK Attack

Adversary can abuse optimistic ACKs

Continue ACK'ing more-and-more optimistically

Victim sends faster and faster

Eventually packets get dropped, but ACKs still coming!

Victim thinks the connection is great

Can consume all the capacity on victim's side

Amplification $\Rightarrow$ Attacker causes victim to send much more traffic than the attack requires

Shrew Attack

Congestion control looks for packet loss

$\Rightarrow$ Doubles time between sends until data gets through

$\Rightarrow$ After that, linear increase in send rate until next drop

For a single lost packet, not a big deal

Shrew attack sends *periodic bursts*

Each burst causes a small amount of loss

Can be timed so that next attempt by sender will coincide with another burst

As long as this continues, TCP connection will be blocked

$\Rightarrow$ Denial of service!

In practice, difficult to exploit effectively

Group Exercise 2

We've already seen how you can spoof source addresses using iptables. Let's look at what we might do with this.

Kevin Mitnick's attack was against a protocol rarely used now, but we can see how to construct the attack packets using our existing tools. We already have nc to generate arbitrary traffic (you can feed it an input file, or drive it by script).

Look through the iptables man page, and see what you can control. The goal is to be able to set the source address *and* port for a packet, as well as the *sequence number*. Is this possible with iptables? If not, how much can you accomplish?