ΥΣ13 - Computer Security

Network Security

Κώστας Χατζηκοκολάκης
Computers connected in a network
  - but also: smartphones, fridges, IoT devices, ...

Each device has an IP address

Packets are routed via intermediate nodes

Still using IPv4 (almost 40 year old!)
  - very very hard to replace
Context

- **Attacker model**
  - Intercept packets
  - Modify packets
  - Inject packets
  - Control some routers
  - Participate in any protocol

- Useful to consider combinations of the above
Internet protocol suite

Four layers (7 in the OSI model)

- **Link**
  - Physical addresses
  - Physical aspects of communication

- **Internet**
  - Addressing (source/dest IP)
  - Routing
  - Time to live
Internet protocol suite

Four layers (7 in the OSI model)

• **Transport**
  - Source/dest ports
  - Ordering of packets (Sequence numbers)
  - ACKs, checksums

• **Application**
  - The “real data”, application-dependent
Internet protocol suite

Protocols

• Link
  - Ethernet
  - WiFi
  - DSL
  - ...

• Internet
  - IP
  - ICMP
  - ...

Internet protocol suite

Protocols

• **Transport**
  - TCP
  - UDP
  - ...

• **Application**
  - HTTP / HTTPS
  - SSH
  - SMTP
  - ...
Internet protocol suite

Packet example

**Physical Layer:** ethernet
the 2 MAC addresses
+ IP indication

**Network Layer:** IP
IP addresses, TTL,
checksum, fragmentation

**Transport Layer:** TCP
Ports, Seq Ack numbers,
checksum, timestamps

**Application Layer:** HTTP
Request: GET
Request URI
Referrer
User-agent info
Connection info
• Connectionless communication
  - using only source/dest IP addresses

• Routing
  - communication across network boundaries
  - routing tables kept by routers
  - no authentication

• Fragmentation & reassembly
  - No reliability
TCP

- Connection-based communication
  - identified by source/dest IP + port (multiplexing)
- Server process “listens” to a port
  - Often determined by the application protocol (HTTP, SMTP, etc)
- Client process connects to dest IP+port
  - Source port selection usually random
- Connection established by handshake
- Reliability
UDP

- Connectionless communication over IP
- Fast alternative to TCP
  - Only 8 bytes overhead, no handshakes
  - Stateless
- Some higher-level features
  - Addressing based on IP+port (multiplexing)
  - Checksums
- But many missing
  - No ACKs (unreliable)
  - No ordering
- Often used for “streaming”-like applications
Traceroute to google.com (216.58.215.46), 30 hops max, 60 byte packets
1 _gateway (195.134.67.1) 0.715 ms 0.789 ms 0.884 ms
2 uoa-ilisia-1-gw.kolettir.access-link.grnet.gr (62.217.96.172) 0.763 ms 0.796 ms 0.835 ms
3 grnet-ias-geant-gw.mx1 ath2.gr.geant.net (83.97.88.65) 1.574 ms 1.630 ms 1.620 ms
4 ae0.mx2.ath.gr.geant.net (62.40.98.140) 31.556 ms 31.650 ms 31.547 ms
5 ae2.mx1.mil2.it.geant.net (62.40.98.150) 25.654 ms 27.861 ms 27.793 ms
6 72.14.203.32 (72.14.203.32) 25.593 ms 25.766 ms 25.500 ms
7 108.170.245.73 (108.170.245.73) 64.548 ms 108.170.245.89 (108.170.245.89) 73.238 ms
8 209.85.142.221 (209.85.142.221) 72.001 ms 72.14.238.21 (72.14.238.21) 71.999 ms
9 216.239.35.201 (216.239.35.201) 78.302 ms 78.299 ms 78.277 ms
10 209.85.251.217 (209.85.251.217) 54.466 ms 72.14.238.54 (72.14.238.54) 54.472 ms
11 108.170.245.1 (108.170.245.1) 52.509 ms 52.443 ms 50.669 ms
12 108.170.235.15 (108.170.235.15) 54.116 ms 51.975 ms 51.967 ms
13 par21s17-in-f14.1e100.net (216.58.215.46) 51.943 ms 54.241 ms 54.202 ms
Traceroute

- Time to live (TTL)
  - IP header
  - Decreased at every hop
  - If 0 the router discards and notifies the originator (ICMP time exceeded)

- Traceroute: repeatedly send packets (ICMP echo request)
  - with TTL = 1, 2, ...
  - 3 packets for every value
  - Until we reach the host (or a threshold)
  - Routers might not respond
TCP 3-way handshake

- Connection identified by source/dest address/port
- Sequence numbers (SN) in every message
- Handshake
  - SYN(SNc)
  - SYN(SNs)-ACK(SNc)
  - ACK(SNs)
  - Data-exchange (bidirect.)
TCP 3-way handshake

• Connection identified by source/dest address/port
• Sequence numbers (SN) in every message
• Handshake
  - SYN(SNc)
  - SYN(SNs)-ACK(SNc)
  - ACK(SNs)
  - Data-exchange (bidirect.)
• What can go wrong here?
SYN flood

- Flood the server with SYNs
- But no ACK!
- Connections stay “half-open” on the server until they timeout
  - Keeping state consumes resources
  - Can lead to Denial of Service (DoS)
SYN flood

- Flood the server with SYNs
- But no ACK!
- Connections stay “half-open” on the server until they timeout
  - Keeping state consumes resources
  - Can lead to Denial of Service (DoS)
- Can the server limit the number of SYNs from the same host?
  - No! the attacker can easily “spoof” the sender IP
IP spoofing

- Can we impersonate a client?

![Three-Way Handshake Diagram]

- (A(C) S : SYN(SNa)
- S ! C : SYN(SNs)-ACK(SNa)
- A(C) ! S : ACK(SNs)
IP spoofing

- Can we impersonate a client?
  - Trivial if we control an intermediate router!
  - If we don’t?
IP spoofing

• Can we **impersonate** a client?
  - Trivial if we control an intermediate router!
  - If we don’t?

• We can still **send** packets with a spoofed IP, without access to the replies
IP spoofing

- Can we impersonate a client?
  - Trivial if we control an intermediate router!
  - If we don’t?

- We can still send packets with a spoofed IP, without access to the replies
  - It’s sufficient to guess SNs for the ACK!
    - A(C) → S : SYN(SNa)
    - S ↦ C : SYN(SNs)-ACK(SNa)
    - A(C) ↦ S : ACK(SNs)
IP spoofing

• Can we guess the server’s SN?

• Initial Sequence Number
  - Counter incremented over time and for every new connection
  - Predictable!
IP spoofing

Why is it bad?
IP spoofing

Why is it bad?

• Bypass IP-based authorization
  - Still **widely-used** today
  - SMTP, web-services, firewall IP white/black-listing, etc
Why is it bad?

- Bypass IP-based authorization
  - Still *widely-used* today
  - SMTP, web-services, firewall IP white/black-listing, etc

- Inject data to existing connection
  - DNS response (UDP, no SN at all!)
IP spoofing

Why is it bad?

• Bypass IP-based authorization
  - Still **widely-used** today
  - SMTP, web-services, firewall IP white/black-listing, etc

• Inject data to existing connection
  - **DNS response** (UDP, no SN at all!)

• Reset existing connections (RST)
  - SNc is needed, but only approximately
  - Denial of service, or exploit to break some other protocol
IP spoofing

Solution?

• Routers expect ISN to be increasing
  - *Protocol* bugs are hard to fix (compared to implementation bugs)
IP spoofing

Solution?

- Routers expect ISN to be increasing
  - *Protocol* bugs are hard to fix (compared to implementation bugs)
- Different ISN for each client! How?
Solution?

- Routers expect ISN to be increasing
  - Protocol bugs are hard to fix (compared to implementation bugs)
- Different ISN for each client! How?

- RFC 6528:
  ISN = Timer + H(localip, localport, remoteip, remoteport, secretkey)
  - Why we include secretkey?
IP spoofing

Solution?

- Routers expect ISN to be increasing
  - Protocol bugs are hard to fix (compared to implementation bugs)

- Different ISN for each client! How?

- RFC 6528:
  ISN = Timer + H(localip, localport, remoteip, remoteport, secretkey)
  - Why we include secretkey?

- But still not perfect
  - 32bit space possible to guess
  - Still trivial if we control routers

- **Conclusion**: For serious security we need to build on top of TCP
Denial of Service

• Remotely consume a resource of the server
  - Bandwidth,
  - CPU
  - Memory
  - ...

• Until the resource is depleted
  - no more clients can connect
Denial of Service

- Typically involves some sort of **flooding**
- SYN flooding
Denial of Service

- Typically involves some sort of flooding
- SYN flooding
- Ping flooding
  - ICMP echo request
    ~$ ping google.com
    PING google.com (216.58.215.46) 56(84) bytes of data.
    64 bytes from par21s17-in-f14.1e100.net (216.58.215.46): icmp_seq=1 ttl=47 time=52 ms
  - Low-level protocol (no use of TCP), can send packets fast
  - The server sends a reply back
• Typically involves some sort of **flooding**

• **SYN flooding**

• **Ping flooding**
  - **ICMP echo request**
    ~$ ping google.com
    PING google.com (216.58.215.46) 56(84) bytes of data.
    64 bytes from par21s17-in-f14.1e100.net (216.58.215.46): icmp_seq=1 ttl=47 time=52.1 ms
  - Low-level protocol (no use of TCP), can send packets fast
  - The server sends a reply back

• **We need more resources than the server**
  - Use many senders at once
Smurf attack

- Send an Echo to a broadcast address
- **Spoof** the sender IP with the sender’s
- All machines flood the victim
Distributed Denial of Service (DDoS)

- Compromise hosts via virus, worm, etc
- Coordinate the attack
- Hard to distinguish from legitimate users
Fork Bomb

- Another kind of DoS
- Fork, and keep forking in the children
  - exponential growth
- Consumes OS resources for process management
- Try this in your own machine!

```
~$ :(){ :|:& };:

# some other terminal
~$ ls
bash: fork: retry: Resource temporarily unavailable
```
Preventing SYN floods

Crucial properties

- Packets need to arrive from multiple source IPs
  - otherwise trivial to filter
- The adversary spoofs the sender IP
  - but does not get replies!
- The server needs to keep state for all fake clients
Preventing SYN floods

Crucial properties

- Packets need to arrive from multiple source IPs - otherwise trivial to filter
- The adversary spoofs the sender IP - but does not get replies!
- The server needs to keep state for all fake clients

Idea

- Make the client store the state!
- Only store state in the server for clients that have proven to get our replies
Preventing SYN floods

SYN cookies

Encode the state in the SNs sent to the client.
Then forget about the connection (no state!)

Check the SNs contained in the client’s ACK.
Store state only if ok.

Spoofing the source is useless.
The adversary needs to control the users or the network.

One approach:
SNs = H(ports, ips, key, time) || time
Preventing SYN floods

**SYN cookies**

- Encode the state in the SNs send to client
  - Then **forget** about the connection (no state!)

- Check the SNs contained in the client’s ACK
  - Store state only if ok

- Spoofing the source is useless
  - The adversary needs to control the users or the network
Preventing SYN floods

**SYN cookies**

- Encode the state in the SNs send to client
  - Then *forget* about the connection (no state!)
- Check the SNs contained in the client’s ACK
  - Store state only if ok
- Spoofing the source is useless
  - The adversary needs to control the users or the network
- One approach
  - $\text{SNs} = H(\text{ports}, \text{ips}, \text{key}, \text{time}) \parallel \text{time}$
Preventing SYN floods

SYN cookies

• One approach
  - SNs = H(ports, ips, key, time) || time

• Protocol compliant

• Problem 1: when ACK is lost
  - The server is supposed to resend SYN-ACK
  - But it won’t, it does not store connections!

• Problem 2: options sent in SYN are lost

• Solution: only use under attack
Preventing DoS

• Client needs to **solve a puzzle** to connect
  - Eg: brute-force a hash (within a controlled range of values)

• Generic solution, also used to prevent spam

• But requires changes to both client and server
Achieving secure communication

- TCP is an inherently insecure protocol
  - no security against an adversary who controls the network
  - limited security against an adversary who simply participates

- Solution
  - Use crypto to build a secure connection over an insecure network

- Most widely used: TLS
  - Also: IPSec, SSH, …

- We can also tunnel the traffic of an entire network
  - Secure VPN
• Widely used in web-browsers

• Crucial use of crypto:
  - Assymetric-crypt: exchange keys
  - Symmetric crypto: encrypt the main traffic
  - Digital signatures: authentication
TLS handshake

Client

ClientHello

ServerHello
Certificate*
ServerKeyExchange*
CertificateRequest*
ServerHelloDone

Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
Finished

[ChangeCipherSpec]
Finished

Application Data

Server

* = προαιρετικά
TLS handshake

Client

version, random1, TLS RSA WITH AES 128 CBC_SHA

ClientHello

version, random2, session id, cipher, PK_{RSA}, Sign(SK_{CA}, PK)

ServerHello

Certificate

E(PK_{RSA}, premaster_key)

ready

ready

Extract master_key from random1 + random2 + premaster_key

Server
• Make sure to stay up to date
  - SSL 2.0, 1995, Deprecated in 2011
  - SSL 3.0, 1996, Deprecated in 2015
  - TLS 1.0, 1999, Deprecated in 2020
  - TLS 1.1, 2006, Deprecated in 2020
  - TLS 1.2, 2008
  - TLS 1.3, 2018

• But it’s hard to do while maintaining compatibility
• POODLE
  - Man in the middle
  - Block the connection until the client tries SSLv3
  - The server will happily accept it
  - TLS_FALLBACK_SCSV: tell the server we are downgrading
References

• Ross Anderson, Security Engineering, Chapter 21
• A look back at ”security problems in the TCP/IP protocol suite
• SYN cookies
• Bypassing domain control verification with DNS response spoofing