Network Security 3
Spot the Zero Day: TPLink Miniature Wireless Router
Spot the Zero Day:
TPLink Miniature Wireless Router
DNS Resource Records and RRSETs

- DNS records (Resource Records) can be one of various types
  - Name TYPE Value
    - Also a “time to live” field: how long in seconds this entry can be cached for
- Addressing:
  - A: IPv4 addresses
  - AAAA: IPv6 addresses
  - CNAME: aliases, “Name X should be name Y”
  - MX: “the mailserver for this name is Y”
- DNS related:
  - NS: “The authority server you should contact is named Y”
  - SOA: “The operator of this domain is Y”
- Other:
  - text records, cryptographic information, etc.…

- Groups of records of the same type form RRSETs:
  - E.g. all the nameservers for a given domain.
The Many Moving Pieces
In a DNS Lookup of www.isc.org

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>TTL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Many Moving Pieces
In a DNS Lookup of www.isc.org

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>TTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.</td>
<td>NS</td>
<td>a0.afilias-nst.info</td>
<td>172800</td>
</tr>
<tr>
<td>a0.afilias-nst.info</td>
<td>A</td>
<td>199.19.56.1</td>
<td>172800</td>
</tr>
</tbody>
</table>

User’s ISP’s
Recursive Resolver

Authority Server

? A www.isc.org
Answers:
Authority:
isc.org. NS ns.isc.afilias-nst.info.
Additional:
sfba.sns-pb.isc.org. A 199.6.1.30
ns.isc.afilias-nst.info. A 199.254.63.254
The Many Moving Pieces
In a DNS Lookup of www.isc.org

User’s ISP’s  ? A www.isc.org
Recursive Resolver

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>TTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.</td>
<td>NS</td>
<td>a0.afilias-nst.info</td>
<td>172800</td>
</tr>
<tr>
<td>a0.afilias-nst.info</td>
<td>A</td>
<td>199.19.56.1</td>
<td>172800</td>
</tr>
<tr>
<td>isc.org.</td>
<td>NS</td>
<td>sfba.sns-pb.isc.org</td>
<td>86400</td>
</tr>
<tr>
<td>isc.org.</td>
<td>NS</td>
<td>ns.isc.afilias-nst.info</td>
<td>86400</td>
</tr>
<tr>
<td>sfbay.sns-pb.isc.org</td>
<td>A</td>
<td>199.6.1.30</td>
<td>86400</td>
</tr>
</tbody>
</table>

isc.org.
Authority Server

? A www.isc.org
Answers:
www.isc.org. A 149.20.64.42
isc.org. NS ns.isc.afilias-nst.info.
Additional:
sfba.sns-pb.isc.org. A 199.6.1.30
ns.isc.afilias-nst.info. A 199.254.63.254
The Many Moving Pieces
In a DNS Lookup of www.isc.org

User’s ISP’s Recursive Resolver

? A www.isc.org
Answers: www.isc.org A 149.20.64.42

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>TTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.</td>
<td>NS</td>
<td>a0.afilias-nst.info</td>
<td>172800</td>
</tr>
<tr>
<td>a0.afilias-nst.info</td>
<td>A</td>
<td>199.19.56.1</td>
<td>172800</td>
</tr>
<tr>
<td>isc.org.</td>
<td>NS</td>
<td>sfba.sns-pb.isc.org</td>
<td>86400</td>
</tr>
<tr>
<td>isc.org.</td>
<td>NS</td>
<td>ns.isc.afilias-net.info</td>
<td>86400</td>
</tr>
<tr>
<td>sfbay.sns-pb.isc.org</td>
<td>A</td>
<td>199.6.1.30</td>
<td>86400</td>
</tr>
<tr>
<td><a href="http://www.isc.org">www.isc.org</a></td>
<td>A</td>
<td>149.20.64.42</td>
<td>600</td>
</tr>
</tbody>
</table>
Stepping Through This With **dig**

- Some flags of note:
  - `+norecurse`: Ask directly like a recursive resolver does
  - `+trace`: Act like a recursive resolver without a cache

```shell
nweaver% dig +norecurse slashdot.org @a.root-servers.net

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 26444
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 6, ADDITIONAL: 12

;; QUESTION SECTION:
;slashdot.org.                  IN      A

;; AUTHORITY SECTION:
org.                    172800  IN      NS      a0.org.afilias-nst.info.
...

;; ADDITIONAL SECTION:
a0.org.afilias-nst.info. 172800 IN      A      199.19.56.1
```
So in `dig` parlance

- So if you want to recreate the lookups conducted by the recursive resolver:
  - `dig +norecurse www.isc.org @a.root-servers.net`
  - `dig +norecurse www.isc.org @199.19.56.1`
  - `dig +norecurse www.isc.org @199.6.1.30`
Security risk #1: malicious DNS server

- Of course, if *any* of the DNS servers queried are malicious, they can lie to us and fool us about the answer to our DNS query…
- and they used to be able to fool us about the answer to other queries, too, using *cache poisoning*. Now fixed (phew).
Security risk #2: on-path eavesdropper

- If attacker can eavesdrop on our traffic… we’re hosed.
- Why?
Security risk #2: on-path eavesdropper

• If attacker can eavesdrop on our traffic… we’re hosed.
• Why? They can see the query and the 16-bit transaction identifier, and race to send a spoofed response to our query.
• China does this operationally:
  • Note: You may need to use the IPv4 address of www.tsinghua.edu
• dig www.benign.com @www.tsinghua.edu
• dig www.facebook.com @www.tsinghua.edu
Security risk #3: off-path attacker

• If attacker can’t eavesdrop on our traffic, can he inject spoofed DNS responses?
• Answer: It used to be possible, via blind spoofing. We’ve since deployed mitigations that makes this harder (but not totally impossible).
Blind spoofing

- Say we look up mail.google.com; how can an off-path attacker feed us a bogus A answer before the legitimate server replies?

- How can such a remote attacker even know we are looking up mail.google.com?

Suppose, e.g., we visit a web page under their control:

```html
...<img src="http://mail.google.com" ...> ...
```
Blind spoofing

• Say we look up mail.google.com; how can an off-path attacker feed us a bogus A answer before the legitimate server replies?

• How can such an attacker even know we are looking up mail.google.com?

Suppose, e.g., we visit a web page under their control:

\[
\ldots <\text{img src="http://mail.google.com" } \ldots > \ldots
\]

This HTML snippet causes our browser to try to fetch an image from mail.google.com. To do that, our browser first has to look up the IP address associated with that name.
Blind spoofing

Once they know we’re looking it up, they just have to guess the Identification field and reply before legit server.

How hard is that?

Originally, identification field incremented by 1 for each request. How does attacker guess it?

<img src="http://badguy.com" ...> —— They observe ID k here
<img src="http://mail.google.com" ...> —— So this will be k+1
DNS Blind Spoofing, cont.

Once we **randomize** the Identification, attacker has a 1/65536 chance of guessing it correctly.

Are we pretty much safe?

Attacker can send lots of replies, not just one ...

**However**: once reply from legit server arrives (with correct Identification), it’s **cached** and no more opportunity to poison it. Victim is innoculated!

Unless attacker can send 1000s of replies before legit arrives, we’re likely safe – phew! ?
Enter Kaminski...

Glue Attacks

- Dan Kaminski noticed something strange, however...
- Most DNS servers would **cache** the in-bailiwick glue...
- And then **promote** the glue
- And will also **update** entries based on glue
- So if you first did this lookup...
  - And then went to `a0.org.afilias-nst.info`
  - there would be no other lookup!

nweaver% dig +norecurse slashdot.org @a.root-servers.net

```
; <<>> DiG 9.8.3-P1 <<>> +norecurse slashdot.org @a.root-servers.net
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 26444
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 6, ADDITIONAL: 12

;; QUESTION SECTION:
slashdot.org.                  IN      A

;; AUTHORITY SECTION:
org.                    172800  IN      NS      a0.org.afilias-nst.info.

;; ADDITIONAL SECTION:
a0.org.afilias-nst.info. 172800 IN      A       199.19.56.1

;; Query time: 128 msec
;; SERVER: 198.41.0.4#53(198.41.0.4)
;; WHEN: Tue Apr 16 09:48:32 2013
;; MSG SIZE  rcvd: 432
```
The Kaminski Attack
In Practice

• Rather than trying to poison www.google.com...

• Instead try to poison a.google.com...
  And state that "www.google.com" is an authority
  And state that "www.google.com A 133.7.133.7"
  • If you succeed, great!

• But if you fail, just try again with b.google.com!
  • Turns "Race once per timeout" to "race until win"

• So now the attacker may still have to send lots of packets
  • In the 10s of thousands

• The attacker can keep trying until success
Defending Against Kaminski: Up the Entropy

- Also randomize the UDP source port
  - Adds 16 bits of entropy
- Observe that most DNS servers just copy the request directly
  - Rather than create a new reply
- So caMeLcase the NamE ranDomly
  - Adds only a few bits of entropy however, but it does help
Defend Against
Kaminski: Validate Glue

• Don't blindly accept glue records...
  • Well, you *have* to accept them for the purposes of resolving a name
• But if you are going to cache the glue record...
• Either only use it for the context of a DNS lookup
  • No more promotion
• Or explicitly validate it with another fetch
• Unbound implemented this, bind did not
  • Largely a political decision: bind is heavily committed to DNSSEC (next week's topic)
Oh, and Profiting from Rogue DNS

- Suppose you take over a lot of home routers...
- How do you make money with it?
- Simple: Change their DNS server settings
- Make it point to yours instead of the ISPs
- Now redirect all advertising
- And instead serve up ads for "Vimax" pills...
Today:
The Internet

- How the Internet routes IP packets
  - Distributed trust through Autonomous Systems
- How TCP works
- Denial of Service Attacks
- (If time) the Firewall #1
## IP Packet Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit/Byte Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>4-bit</td>
</tr>
<tr>
<td>Header Length</td>
<td>4-bit</td>
</tr>
<tr>
<td>Type of Service (TOS)</td>
<td>8-bit</td>
</tr>
<tr>
<td>Total Length (Bytes)</td>
<td>16-bit</td>
</tr>
<tr>
<td>Identification</td>
<td>16-bit</td>
</tr>
<tr>
<td>Flags</td>
<td>3-bit</td>
</tr>
<tr>
<td>Fragment Offset</td>
<td>13-bit</td>
</tr>
<tr>
<td>Time to Live (TTL)</td>
<td>8-bit</td>
</tr>
<tr>
<td>Protocol</td>
<td>8-bit</td>
</tr>
<tr>
<td>Header Checksum</td>
<td>16-bit</td>
</tr>
<tr>
<td>Source IP Address</td>
<td>32-bit</td>
</tr>
<tr>
<td>Destination IP Address</td>
<td>32-bit</td>
</tr>
<tr>
<td>Options (if any)</td>
<td></td>
</tr>
<tr>
<td>Payload</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit/Byte Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# IP Packet Structure

<table>
<thead>
<tr>
<th>4-bit Version</th>
<th>4-bit Header Length</th>
<th>8-bit Type of Service (TOS)</th>
<th>16-bit Total Length (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit Identification</td>
<td>3-bit Flags</td>
<td>13-bit Fragment Offset</td>
<td>8-bit Time to Live (TTL)</td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (if any)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Payload</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Specifies the length of the entire IP packet: bytes in this header plus bytes in the Payload.
## IP Packet Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>Specifies the version of the IP protocol (4 bits)</td>
</tr>
<tr>
<td>Header Length</td>
<td>Specifies the length of the header in 32-bit units (4 bits)</td>
</tr>
<tr>
<td>Type of Service (TOS)</td>
<td>Specifies the type of service (8 bits)</td>
</tr>
<tr>
<td>Total Length (Bytes)</td>
<td>Specifies the total length of the packet in bytes (16 bits)</td>
</tr>
<tr>
<td>Identification</td>
<td>Identifies the packet (16 bits)</td>
</tr>
<tr>
<td>Flags</td>
<td>Specifies the flags for the packet (3 bits)</td>
</tr>
<tr>
<td>Fragment Offset</td>
<td>Specifies the fragment offset (13 bits)</td>
</tr>
<tr>
<td>Time to Live (TTL)</td>
<td>Specifies the time to live (8 bits)</td>
</tr>
<tr>
<td>Protocol</td>
<td>Specifies the protocol (8 bits)</td>
</tr>
<tr>
<td>Source IP Address</td>
<td>Specifies the source IP address (32 bits)</td>
</tr>
<tr>
<td>Destination IP Address</td>
<td>Specifies the destination IP address (32 bits)</td>
</tr>
<tr>
<td>Options (if any)</td>
<td>Additional options for the packet (variable length)</td>
</tr>
<tr>
<td>Payload</td>
<td>The payload data for the transport protocol (variable length)</td>
</tr>
</tbody>
</table>

Specifies how to interpret the start of the Payload, which is the header of a *Transport Protocol* such as TCP or UDP.
## IP Packet Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>4-bit</td>
<td>4-bit</td>
</tr>
<tr>
<td>Header Length</td>
<td>4-bit</td>
<td>4-bit</td>
</tr>
<tr>
<td>Type of Service (TOS)</td>
<td>8-bit</td>
<td>8-bit</td>
</tr>
<tr>
<td>Total Length (Bytes)</td>
<td>16-bit</td>
<td>16-bit</td>
</tr>
<tr>
<td>Identification</td>
<td>16-bit</td>
<td>16-bit</td>
</tr>
<tr>
<td>Flags</td>
<td>3-bit</td>
<td>3-bit</td>
</tr>
<tr>
<td>Fragment Offset</td>
<td>13-bit</td>
<td>13-bit</td>
</tr>
<tr>
<td>Time to Live (TTL)</td>
<td>8-bit</td>
<td>8-bit</td>
</tr>
<tr>
<td>Protocol</td>
<td>8-bit</td>
<td>8-bit</td>
</tr>
<tr>
<td>Header Checksum</td>
<td>16-bit</td>
<td>16-bit</td>
</tr>
<tr>
<td>Source IP Address</td>
<td>32-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>Destination IP Address</td>
<td>32-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>Options (if any)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Payload</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# IP Packet Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>4</td>
</tr>
<tr>
<td>Header Length</td>
<td>4</td>
</tr>
<tr>
<td>Type of Service (TOS)</td>
<td>8</td>
</tr>
<tr>
<td>Total Length (Bytes)</td>
<td>16</td>
</tr>
<tr>
<td>Identification</td>
<td>16</td>
</tr>
<tr>
<td>Flags</td>
<td>3</td>
</tr>
<tr>
<td>Fragment Offset</td>
<td>13</td>
</tr>
<tr>
<td>Protocol</td>
<td>8</td>
</tr>
<tr>
<td>Header Checksum</td>
<td>16</td>
</tr>
<tr>
<td>Source IP Address</td>
<td>32</td>
</tr>
<tr>
<td>Destination IP Address</td>
<td>32</td>
</tr>
<tr>
<td>Options (if any)</td>
<td></td>
</tr>
<tr>
<td>Payload</td>
<td></td>
</tr>
</tbody>
</table>
### IP Packet Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>4-bit</td>
</tr>
<tr>
<td>Header Length</td>
<td>4-bit</td>
</tr>
<tr>
<td>Type of Service (TOS)</td>
<td>8-bit</td>
</tr>
<tr>
<td>Total Length (Bytes)</td>
<td>16-bit</td>
</tr>
<tr>
<td>Identification</td>
<td>16-bit</td>
</tr>
<tr>
<td>Flags</td>
<td>3-bit</td>
</tr>
<tr>
<td>Fragment Offset</td>
<td>13-bit</td>
</tr>
<tr>
<td>Time to Live (TTL)</td>
<td>8-bit</td>
</tr>
<tr>
<td>Protocol</td>
<td>8-bit</td>
</tr>
<tr>
<td>Header Checksum</td>
<td>16-bit</td>
</tr>
<tr>
<td>Source IP Address</td>
<td>32-bit</td>
</tr>
<tr>
<td>Destination IP Address</td>
<td>32-bit</td>
</tr>
<tr>
<td>Options (if any)</td>
<td></td>
</tr>
<tr>
<td>Payload</td>
<td></td>
</tr>
</tbody>
</table>
IP Packet Header (Continued)

- Two IP addresses
  - Source IP address (32 bits)
  - Destination IP address (32 bits)

- Destination address
  - Unique identifier/locator for the receiving host
  - Allows each node to make forwarding decisions

- Source address
  - Unique identifier/locator for the sending host
  - Recipient can decide whether to accept packet
  - Enables recipient to send a reply back to source
IP: “Best Effort” Packet Delivery

- Routers inspect destination address, locate “next hop” in forwarding table
  - Address = ~unique identifier/locator for the receiving host

- Only provides a “I’ll give it a try” delivery service:
  - Packets may be lost
  - Packets may be corrupted
  - Packets may be delivered out of order
IP Routing: Autonomous Systems

• Your system sends IP packets to the gateway...
  • But what happens after that?
• Within a given network its routed internally
• But the key is the Internet is a network-of-networks
  • Each "autonomous system" (AS) handles its own internal routing
  • The AS knows the next AS to forward a packet to
• Primary protocol for communicating in between ASs is BGP
Packet Routing on the Internet
Remarks

- This is a network of networks
  - It's designed with failures in mind:
    Links can go down and the system will recover
  - But it also generally trust-based
    - A system can lie about what networks it can route to!
- Each hop decrements the TTL
  - Prevents a "routing loop" from happening
- Routing can be asymmetric
  - Since in practice networks may (slightly) override BGP, and
IP Spoofing
And Autonomous Systems

• The edge-AS where a user connects should restrict packet spoofing
  • Sending a packet with a different sender IP address
• But about 25% of them don't...
  • So a system can simply lie and say it comes from someplace else
• This enables blind-spoofing attacks
  • Such as the Kaminski attack on DNS
• It also enables "reflected DOS attacks"
On-path Injection vs Off-path Spoofing

Host A communicates with Host D

On-path  Off-path

Host A  Router 1  Router 2  Router 5  Router 3  Router 7  Host D
Host B  Router 4
Host C

Off-path

Off-path
Lying in BGP
“Best Effort” is Lame! What to do?

- It’s the job of our Transport (layer 4) protocols to build data delivery services that our apps need out of IP’s modest layer-3 service.

- **#1 workhorse:** **TCP** (Transmission Control Protocol)

- **Service provided by TCP:**
  - **Connection oriented** (explicit set-up / tear-down)
    - End hosts (processes) can have multiple concurrent long-lived communication
  - **Reliable**, in-order, **byte-stream** delivery
    - Robust detection & retransmission of lost data
TCP “Bytestream” Service

- Process A on host H1
- Process B on host H2

Processes don’t ever see packet boundaries, lost or corrupted packets, retransmissions, etc.
Bidirectional communication:

Process B on host H2

There are two separate bytestreams, one in each direction

Process A
on host H1
TCP
TCP

These plus IP addresses define a given connection

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Sequence number</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Acknowledgment</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>HdrLen</th>
<th>Flags</th>
<th>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Options (variable)</th>
</tr>
</thead>
</table>

Data
Suppose our browser used port 23144 for our connection, and Google’s server used 443.

Then our connection will be fully specified by the single tuple

<216.97.19.132, 23144, 172.217.6.78, 443, TCP>
# TCP

TCP

Used to order data in the connection: client program receives data **in order**

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequence number</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Acknowledgment</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Flags</strong></td>
<td><strong>Advertised window</strong></td>
</tr>
<tr>
<td><strong>HdrLen</strong></td>
<td><strong>Checksum</strong></td>
</tr>
<tr>
<td><strong>Urgent pointer</strong></td>
<td><strong>Options (variable)</strong></td>
</tr>
</tbody>
</table>

**Sequence number** assigned to start of byte stream is picked when connection begins; **doesn’t** start at 0.
TCP

Used to say how much data has been received

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>0</td>
</tr>
<tr>
<td>Flags</td>
<td>Advertised window</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>

Data

Acknowledgment gives seq # just beyond highest seq. received in order.

If sender successfully sends \( N \) bytestream bytes starting at seq \( S \) then “ack” for that will be \( S+N \).
Sequence Numbers

Host A

ISN (initial sequence number)

Sequence number from A = 1st byte of data

TCP HDR

TCP Data

TCP HDR

TCP Data

Host B

ACK sequence number from B = next expected byte
TCP

Flags have different meaning:

SYN: Synchronize, used to initiate a connection

ACK: Acknowledge, used to indicate acknowledgement of data

FIN: Finish, used to indicate no more data will be sent (but can still receive and acknowledge data)

RST: Reset, used to terminate the connection completely

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Acknowledgment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HdrLen</th>
<th>Flags</th>
<th>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Options (variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Data
TCP Conn. Setup & Data Exchange

Client (initiator)
IP address 1.2.1.2, port 3344

Server
IP address 9.8.7.6, port 80

1. Client (initiator) sends an SYN packet to the server with Source Address 1.2.1.2, Source Port 3344, Destination Address 9.8.7.6, and Destination Port 80. The Sequence Number is x.

2. The server responds with an SYN+ACK packet. Source Address 9.8.7.6, Source Port 80, Destination Address 1.2.1.2, and Destination Port 3344. The Sequence Number is y, and the acknowledgment number is x+1.

3. The client sends an ACK packet. Source Address 1.2.1.2, Source Port 3344, Destination Address 9.8.7.6, and Destination Port 80. The Sequence Number is x+1, and the acknowledgment number is y+1.

4. The server sends a Data packet, which contains a request to the server (e.g., "GET /login.html"). Source Address 9.8.7.6, Source Port 80, Destination Address 1.2.1.2, and Destination Port 3344. The Sequence Number is y+1, and the Acknowledgment Number is x+16.

This completes the three-way handshake and establishes the TCP connection. After the connection is established, data exchange begins.
A abrupt Termination

- A sends a TCP packet with RESET (RST) flag to B
  - E.g., because app. process on A crashed
  - (Could instead be that B sends a RST to A)
- Assuming that the sequence numbers in the RST fit with what B expects, That’s It:
  - B’s user-level process receives: ECONNRESET
  - No further communication on connection is possible
Disruption

- Normally, TCP finishes ("closes") a connection by each side sending a FIN control message
  - Reliably delivered, since other side must ack

- But: if a TCP endpoint finds unable to continue (process dies; info from other "peer" is inconsistent), it abruptly terminates by sending a RST control message
  - Unilateral
  - Takes effect immediately (no ack needed)
  - Only accepted by peer if has correct* sequence number
TCP Threat: Data Injection

- If attacker knows ports & sequence numbers (e.g., on-path attacker), attacker can inject data into any TCP connection
  - Receiver B is *none the wiser*!
- Termed TCP connection hijacking (or “session hijacking”)
  - A general means to take over an already-established connection!
- We are toast if an attacker can see our TCP traffic!
  - Because then they immediately know the port & sequence numbers
TCP Data Injection

Client (initiator)
IP address 1.2.1.2, port 3344

Server
IP address 9.8.7.6, port 80

Attacker (AirPwn, QUANTUM, etc)
IP address 6.6.6.6, port N/A

Client dutifully processes as server’s response

SrcA=1.2.1.2, SrcP=3344, DstA=9.8.7.6, DstP=80,
ACK, Seq=x+1, Ack = y+1, Data="GET /login.html"

SrcA=9.8.7.6, SrcP=80,
DstA=1.2.1.2, DstP=3344,
ACK, Seq = y+1, Ack = x+16
Data="200 OK … <poison> ...

Client dutifully processes as server’s response
TCP Data Injection

Client (initiator)
IP address 1.2.1.2, port 3344

Server
IP address 9.8.7.6, port 80

Attacker
IP address 6.6.6.6, port N/A

Client ignores since already processed that part of bytestream: the network can duplicate packets so only pay attention to the first version in sequence.

\[ \text{SrcA}=1.2.1.2, \text{SrcP}=3344, \text{DstA}=9.8.7.6, \text{DstP}=80, \]
\[ \text{ACK}, \text{Seq}=x+1, \text{Ack}=y+1, \text{Data}="\text{GET /login.html}" \]

\[ \text{SrcA}=9.8.7.6, \text{SrcP}=80, \]
\[ \text{DstA}=1.2.1.2, \text{DstP}=3344, \]
\[ \text{ACK}, \text{Seq}=y+1, \text{Ack}=x+16, \text{Data}="\text{200 OK \ldots <poison> \ldots}" \]

\[ \text{SrcA}=9.8.7.6, \text{SrcP}=80, \text{DstA}=1.2.1.2, \text{DstP}=3344, \]
\[ \text{ACK}, \text{Seq}=y+1, \text{Ack}=x+16, \text{Data}="\text{200 OK \ldots <html> \ldots}" \]
TCP Threat: Disruption aka RST injection

- The attacker can also inject RST packets instead of payloads
  - TCP clients must respect RST packets and stop all communication
    - Because it’s a real world error recovery mechanism
    - So "just ignore RSTs don't work"

- Who uses this?
  - China: The Great Firewall does this to TCP requests
  - A long time ago: Comcast, to block BitTorrent uploads
  - Some intrusion detection systems: To hopefully mitigate an attack in progress
TCP Threat: Blind Hijacking

- Is it possible for an off-path attacker to inject into a TCP connection even if they can’t see our traffic?
- YES: if somehow they can infer or guess the port and sequence numbers
TCP Threat: Blind Spoofing

• Is it possible for an off-path attacker to create a fake TCP connection, even if they can’t see responses?
  • YES: if somehow they can infer or guess the TCP initial sequence numbers

• Why would an attacker want to do this?
  • Perhaps to leverage a server’s trust of a given client as identified by its IP address
  • Perhaps to frame a given client so the attacker’s actions during the connections can’t be traced back to the attacker
Blind Spoofing on TCP Handshake

Alleged Client (not actual)
IP address 1.2.1.2, port N/A

Server
IP address 9.8.7.6, port 80

Blind Attacker
SrcA=1.2.1.2, SrcP=5566, DstA=9.8.7.6, DstP=80, SYN, Seq = z

Server
SrcA=9.8.7.6, SrcP=80, DstA=1.2.1.2, DstP=5566, SYN+ACK, Seq = y, Ack = z+1

Attacker’s goal:
SrcA=1.2.1.2, SrcP=5566, DstA=9.8.7.6, DstP=80, ACK, Seq = z+1, ACK = y+1

SrcA=1.2.1.2, SrcP=5566, DstA=9.8.7.6, DstP=80, ACK, Seq = z+1, ACK = y+1, Data = “GET /transfer-money.html"
Blind Spoofing on TCP Handshake

**Alleged Client (not actual)**
- IP address 1.2.1.2, port NA

**Server**
- IP address 9.8.7.6, port 80

**Blind Attacker**
- 

```
SrcA=1.2.1.2, SrcP=5566, DstA=9.8.7.6, DstP=80, SYN, Seq = z
```

```
SrcA=9.8.7.6, SrcP=80, DstA=1.2.1.2, DstP=5566, SYN+ACK, Seq = y, Ack = x+1
```

**Small Note #1:** if alleged client receives this, will be confused ⇒ send a RST back to server … … So attacker may need to hurry!

But firewalls may inadvertently stop this reply to the alleged client so it never sends the RST 😐
Blind Spoofing on TCP Handshake

**Alleged Client (not actual)**
IP address 1.2.1.2, port NA

**Server**
IP address 9.8.7.6, port 80

Blind Attacker

\[
\begin{align*}
\text{SrcA}=1.2.1.2, & \text{ SrcP}=5566, \text{ DstA}=9.8.7.6, \\
& \text{ DstP}=80, \text{ SYN}, \text{ Seq } = z
\end{align*}
\]

\[
\begin{align*}
\text{SrcA}=9.8.7.6, & \text{ SrcP}=80, \\
& \text{ DstA}=1.2.1.2, \text{ DstP}=5566, \text{ SYN+ACK}, \text{ Seq } = y, \text{ Ack } = z+1
\end{align*}
\]

**Big Note #2:** attacker doesn’t get to see this packet!
Blind Spoofing on TCP Handshake

Alleged Client (not actual)
IP address 1.2.1.2, port N/A

Server
IP address 9.8.7.6, port 80

Blind Attacker
SrcA=1.2.1.2, SrcP=5566, DstA=9.8.7.6, DstP=80, SYN, Seq = z

SrcA=9.8.7.6, SrcP=80, DstA=1.2.1.2, DstP=5566, SYN+ACK, Seq = y, Ack = z+1

So how can the attacker figure out what value of y to use for their ACK?

SrcA=1.2.1.2, SrcP=5566, DstA=9.8.7.6, DstP=80, ACK, Seq = z+1, ACK = y+1

SrcA=1.2.1.2, SrcP=5566, DstA=9.8.7.6, DstP=80, ACK, Seq = z+1, ACK = y+1, Data = “GET /transfer-money.html”
Reminder: Establishing a TCP Connection

Each host tells its *Initial Sequence Number* (ISN) to the other host.

(Spec says to pick based on local clock)

Hmm, any way for the attacker to know this?

Sure – make a non-spoofed connection *first*, and see what server used for ISN y then!

How Do We Fix This?

Use a (Pseudo)-Random ISN
Summary of TCP Security Issues

- An attacker who can observe your TCP connection can manipulate it:
  - Forcefully terminate by forging a RST packet
  - Inject (spoof) data into either direction by forging data packets
  - Works because they can include in their spoofed traffic the correct sequence numbers (both directions) and TCP ports
  - Remains a major threat today
Summary of TCP Security Issues

- An attacker who can observe your TCP connection can manipulate it:
  - Forcefully terminate by forging a RST packet
  - Inject (spoof) data into either direction by forging data packets
  - Works because they can include in their spoofed traffic the correct sequence numbers (both directions) and TCP ports
  - Remains a major threat today

- If attacker could predict the ISN chosen by a server, could “blind spoof” a connection to the server
  - Makes it appear that host ABC has connected, and has sent data of the attacker’s choosing, when in fact it hasn’t
  - Undermines any security based on trusting ABC’s IP address
  - Allows attacker to “frame” ABC or otherwise avoid detection
  - Fixed (mostly) today by choosing random ISNs
But wasn't fixed completely...

- CVE-2016-5696
  - "Off-Path TCP Exploits: Global Rate Limit Considered Dangerous" Usenix Security 2016
  - https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/cao

- Key idea:
  - RFC 5961 added some global rate limits that acted as an information leak:
    - Could determine if two clients were communicating on a given port
    - Could determine if you could correctly guess the sequence #s for this communication
      - Required a third host to probe this and at the same time spoof packets
    - Once you get the sequence #s, you can then inject arbitrary content into the TCP stream (d'oh)
The Bane of the Internet: The (distributed) Denial of Service Attack

• Lets say you've run afoul of a bad guy...
  • And he don't like your web page
  • He hires some other bad guy to launch a "Denial of Service" attack

• This other bad guys controls a lot of machines on the Internet
  • These days a million systems is not unheard of

• The bad guy just instructs those machines to make a lot of requests to your server...
  • Blowing it off the network with traffic
And the Firewall...

- Attackers can't attack what they can't talk to!
  - If you don't accept *any* communication from an attacker, you can't be exploited
- The firewall is a network device (or software filter on the end host) that restricts communication
  - Primarily just by IP/Port or network/Port
- Default deny:
  - By default, disallow any contact to this host on any port
- Default allow:
  - By default, allow any contact to this host on any port
- More when we discuss Intrusion Detection next week