The Net Part 4: DNS, IP, TCP...

"I don’t think we fear machines in physical. We fear the beings they will become. We fear the logic they will bring to bear. Detached from humanity, with fresh eyes on Earth and our imprint upon it fresh, will they judge us unworthy of life?

We shouldn’t fear they will be wrong, ending our reign without justification.

We should fear they will be right to do it."

- Taylor Swift
Spot the Zero Day: TPLink Miniature Wireless Router
Spot the Zero Forever Day:
TPLink Miniature Wireless Router

![Image of TP-Link Miniature Wireless Router with circled MAC address and SSID/Password]
DNS Threats

- DNS: path-critical for just about everything we do
  - Maps hostnames ⇔ IP addresses
  - Design only **scales** if we can minimize lookup traffic
    - #1 way to do so: caching
    - #2 way to do so: return not only answers to queries, but additional info that will likely be needed shortly
      - The "glue records"

- What if attacker eavesdrops on our DNS queries?
  - Then similar to DHCP, ARP, AirPwn etc, can spoof responses
  - Consider attackers who *can’t* eavesdrop - but still aim to manipulate us via *how the protocol functions*

- Directly interacting w/ DNS: **dig** program on Unix
  - Allows querying of DNS system
  - Dumps each field in DNS responses
Use Unix “dig” utility to look up IP address (“A”) for hostname eecs.mit.edu via DNS

```
dig eecs.mit.edu A

; ; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
; ; global options: +cmd
; ; Got answer:
; ; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
; ; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
eeecs.mit.edu.           IN      A

;; ANSWER SECTION:
eeecs.mit.edu.           21600   IN      A       18.62.1.6

;; AUTHORITY SECTION:
mit.edu.                11088   IN      NS      BITSY.mit.edu.
mit.edu.                11088   IN      NS      W20NS.mit.edu.
mit.edu.                11088   IN      NS      STRAWB.mit.edu.

;; ADDITIONAL SECTION:
STRAWB.mit.edu.         126738  IN      A       18.71.0.151
BITSY.mit.edu.          166408  IN      A       18.72.0.3
W20NS.mit.edu.          126738  IN      A       18.70.0.160
```
The question we asked the server

dig eecs.mit.edu A

;; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
@eecs.mit.edu.                  IN      A

;; ANSWER SECTION:
eecs.mit.edu.           21600   IN      A       18.62.1.6

;; AUTHORITY SECTION:
mit.edu.                11088   IN      NS      BITSY.mit.edu.
mit.edu.                11088   IN      NS      W20NS.mit.edu.
mit.edu.                11088   IN      NS      STRAWB.mit.edu.

;; ADDITIONAL SE:
STRAWB.mit.edu.         126738  IN      A       18.71.0.151
BITSY.mit.edu.          166408  IN      A       18.72.0.3
W20NS.mit.edu.          126738  IN      A       18.70.0.160
**dig eecs.mit.edu A**

`; ; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
; ; global options: +cmd
; ; Got answer:
; ; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
; ; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
;eecs.mit.edu.                  IN      A

;; ANSWER SECTION:
eecs.mit.edu.           21600   IN      A       18.62.1.6

;; AUTHORITY SECTION:
mit.edu.                11088   IN      NS      BITSY.mit.edu.
mit.edu.                11088   IN      NS      W20NS.mit.edu.
mit.edu.                11088   IN      NS      STRAWB.mit.edu.

;; ADDITIONAL SECTION:
STRAWB.mit.edu.         126738  IN      A       18.71.0.151
BITSY.mit.edu.          166408  IN      A       18.72.0.3
W20NS.mit.edu.          126738  IN      A       18.70.0.160

A 16-bit **transaction identifier** that enables the DNS client (**dig**, in this case) to match up the reply with its original request.
```
dig eecs.mit.edu A

;; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
;eecs.mit.edu.                  IN      A

;; ANSWER SECTION:
;eecs.mit.edu.           21600   IN      A       18.62.1.6

;; AUTHORITY SECTION:
mit.edu.                11088   IN      NS      BITSY.mit.edu.
mit.edu.                11088   IN      NS      W20NS.mit.edu.
mit.edu.                11088   IN      NS      STRAWB.mit.edu.

;; ADDITIONAL SECTION:
STRAWB.mit.edu.         126738  IN      A       18.71.0.151
BITSY.mit.edu.          166408  IN      A       18.72.0.3
W20NS.mit.edu.          126738  IN      A       18.70.0.160
```

"Answer" tells us the IP address associated with eecs.mit.edu is 18.62.1.6 and we can cache the result for 21,600 seconds
In general, a single Resource Record (RR) like this includes, left-to-right, a DNS name, a time-to-live, a family (IN for our purposes - ignore), a type (A here), and an associated value.
dig eecs.mit.edu A

; ; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
; ; global options: +cmd
; ; Got answer:
; ; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
; ; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3
; ; QUESTION SECTION:
eecs.mit.edu.                  IN      A
; ; ANSWER SECTION:
eecs.mit.edu.           21600   IN      A       18.62.1.6
; ; AUTHORITY SECTION:
mit.edu.                11088   IN      NS      BITSY.mit.edu.
mit.edu.                11088   IN      NS      W20NS.mit.edu.
mit.edu.                11088   IN      NS      STRAWB.mit.edu.
; ; ADDITIONAL SECTION:
STRAWB.mit.edu.         126738  IN      A       18.71.0.151
BITSY.mit.edu.          166408  IN      A       18.72.0.3
W20NS.mit.edu.          126738  IN      A       18.70.0.160

“Authority” tells us the name servers responsible for the answer. Each RR gives the hostname of a different name server (“NS”) for names in mit.edu. We should cache each record for 11,088 seconds.

If the “Answer” had been empty, then the resolver’s next step would be to send the original query to one of these name servers.
`dig eecs.mit.edu A`

```
; ; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
; ; global options: +cmd
; ; Got answer:
; ; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
; ; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

; ; QUESTION SECTION:
; ; QUESTION SECTION:
; eecs.mit.edu.

; ; ANSWER SECTION:
; eecs.mit.edu.

; ; AUTHORITY SECTION:
mit.edu. 11088 IN NS BITSY.mit.edu.
mit.edu. 11088 IN NS W20NS.mit.edu.
mit.edu. 11088 IN NS STRAWB.mit.edu.

; ; ADDITIONAL SECTION:
STRAWB.mit.edu. 126738 IN A 18.71.0.151
BITSY.mit.edu. 166408 IN A 18.72.0.3
W20NS.mit.edu. 126738 IN A 18.70.0.160
```

"Additional" provides extra information to save us from making separate lookups for it, or helps with bootstrapping.

Here, it tells us the IP addresses for the hostnames of the name servers. We add these to our cache.
DNS Protocol

Lightweight exchange of *query* and *reply* messages, both with *same* message format

Primarily uses UDP for its transport protocol, which is what we’ll assume

Servers are on port 53 always

Frequently, clients used to use port 53 but can use any port
Message header:

- **Identification**: 16 bit # for query, reply to query uses same #
- Along with repeating the Question and providing Answer(s), replies can include “**Authority**” (name server responsible for answer) and “**Additional**” (info client is likely to look up soon anyway)
- Each Resource Record has a **Time To Live** (in seconds) for **caching** (not shown)
dig eecs.mit.edu A

;; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
eecs.mit.edu.                  IN      A

;; ANSWER SECTION:
eecs.mit.edu.           21600   IN      A       18.62.1.6

;; AUTHORITY SECTION:
mit.edu.                11088   IN      NS      BITSY.mit.edu.
mit.edu.                11088   IN      NS      W20NS.mit.edu.
mit.edu.                11088   IN      NS      STRAWB.mit.edu.

;; ADDITIONAL SECTION:
STRAWB.mit.edu.         126738  IN      A       18.71.0.151
BITSY.mit.edu.          166408  IN      A       18.72.0.3
W20NS.mit.edu.          126738  IN      A       18.70.0.160

What if the mit.edu server is untrustworthy? Could its operator steal, say, all of our web surfing to berkeley.edu’s main web server?
dig eecs.mit.edu A

;; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
;eecs.mit.edu.

;; ANSWER SECTION:
eecs.mit.edu. 21600 IN A 18.62.1.6

;; AUTHORITY SECTION:
mit.edu.        11088 IN  NS  BITSY.mit.edu.
mit.edu.        11088 IN  NS  W20NS.mit.edu.
mit.edu.        11088 IN  NS  STRAWB.mit.edu.

;; ADDITIONAL SECTION:
STRAWB.mit.edu. 126738 IN  A  18.71.0.151
BITSY.mit.edu.  166408 IN  A  18.72.0.3
W20NS.mit.edu.  126738 IN  A  18.70.0.160

Let’s look at a flaw in the original DNS design (since fixed)
dig eecs.mit.edu A

;; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
eecs.mit.edu.                  IN      A

;; ANSWER SECTION:
eecs.mit.edu.           21600   IN      A       18.62.1.6

;; AUTHORITY SECTION:
mit.edu.                11088   IN      NS      BITSY.mit.edu.
mit.edu.                11088   IN      NS      W20NS.mit.edu.
mit.edu.                11088   IN      NS      www.berkeley.edu.

;; ADDITIONAL SECTION:
www.berkeley.edu.       100000  IN      A       18.6.6.6
BITSY.mit.edu.          166408  IN      A       18.72.0.3
W20NS.mit.edu.          126738  IN      A       18.70.0.160

What could happen if the mit.edu server returns the following to us instead?
We’d dutifully store in our cache a mapping of `www.berkeley.edu` to an IP address under MIT’s control. (It could have been any IP address they wanted, not just one of theirs.)
dig eecs.mit.edu A

;; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
;eecs.mit.edu.                  IN      A

;; ANSWER SECTION:
eecs.mit.edu.           21600   IN      A       18.62.1.6

;; AUTHORITY SECTION:
mit.edu.                11088   IN      NS      BITSY.mit.edu.
mit.edu.                11088   IN      NS      W20NS.mit.edu.
mit.edu.                11088   IN      NS      www.berkeley.edu.

;; ADDITIONAL SECTION:
www.berkeley.edu.       100000  IN      A       18.6.6.6
BITSY.mit.edu.          166408  IN      A       18.72.0.3
W20NS.mit.edu.          126738  IN      A       18.70.0.160

In this case they chose to make the mapping last a long time. They could just as easily make it for just a couple of seconds.
dig eecs.mit.edu A

;; <<>> DiG 9.6.0-APPLE-P2 <<>> eecs.mit.edu a
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19901
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
eecs.mit.edu.                  IN      A

;; ANSWER SECTION:
eecs.mit.edu.           21600   IN      A       18.62.1.6

;; AUTHORITY SECTION:
mit.edu.                11088   IN      NS      BITSY.mit.edu.
mit.edu.                11088   IN      NS      W20NS.mit.edu.
mit.edu.                30      IN      NS      www.berkeley.edu.

;; ADDITIONAL SECTION:
www.berkeley.edu.       30      IN      A       18.6.6.6
BITSY.mit.edu.          166408  IN      A       18.72.0.3
W20NS.mit.edu.          126738  IN      A       18.70.0.160

How do we fix such cache poisoning?
Don’t accept **Additional** records unless they’re for the domain we’re looking up

E.g., looking up eecs.mit.edu ⇒ only accept additional records from *.mit.edu

No extra risk in accepting these since server could return them to us directly in an **Answer** anyway.

This is called "**Bailiwick** checking"
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>

User’s ISP’s

Authority Server
(the “root”)

Recursive Resolver

? A www.isc.org
Answers:
Authority:
org. NS a0.afilias-nst.info
Additional:
a0.afilias-nst.info A 199.19.56.1
## The Many Moving Pieces

In a DNS Lookup of **www.isc.org**

### User’s ISP’s

<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>

### Recursive Resolver

### Authority Server

- 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

<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

<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

```
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
```
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:
  • dig www.benign.com @www.tsinghua.edu.cn
  • dig www.facebook.com @www.tsinghua.edu.en
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:

...<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:

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

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
<br>
<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 inoculated!

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 \textit{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 \textit{political} decision:
    bind's developers are 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...
## IP Packet Structure

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-bit Version</td>
<td>Specifies the length of the entire IP packet: bytes in this header plus bytes in the Payload</td>
</tr>
<tr>
<td>4-bit</td>
<td>Header Length</td>
</tr>
<tr>
<td>16-bit Total Length (Bytes)</td>
<td>3-bit Flags 16-bit Identification 13-bit Fragment Offset 8-bit Protocol</td>
</tr>
<tr>
<td>8-bit Time to Live (TTL)</td>
<td>16-bit Identification 8-bit Protocol 32-bit Source IP Address</td>
</tr>
<tr>
<td>8-bit Protocol</td>
<td>32-bit Destination IP Address</td>
</tr>
<tr>
<td>Options (if any)</td>
<td>Payload</td>
</tr>
</tbody>
</table>
### IP Packet Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>8-bit</th>
<th>16-bit</th>
<th>32-bit</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>4-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Header Length</td>
<td>4-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type of Service (TOS)</td>
<td>8-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Length (Bytes)</td>
<td>16-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identification</td>
<td>16-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flags</td>
<td>3-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fragment Offset</td>
<td>13-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time to Live (TTL)</td>
<td>8-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Protocol</td>
<td>8-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source IP Address</td>
<td>32-bit</td>
<td></td>
<td></td>
<td>Options (if any)</td>
</tr>
<tr>
<td>Destination IP Address</td>
<td>32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (if any)</td>
<td></td>
<td></td>
<td></td>
<td></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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-bit Version</td>
<td>Version of the IP protocol.</td>
</tr>
<tr>
<td>4-bit Header Length</td>
<td>Length of the header.</td>
</tr>
<tr>
<td>8-bit Type of Service (TOS)</td>
<td>Type of Service field.</td>
</tr>
<tr>
<td>16-bit Total Length (Bytes)</td>
<td>Total length of the packet in bytes.</td>
</tr>
<tr>
<td>16-bit Identification</td>
<td>Identification number for fragments.</td>
</tr>
<tr>
<td>3-bit Flags</td>
<td>Flags for fragment handling.</td>
</tr>
<tr>
<td>13-bit Fragment Offset</td>
<td>Offset for fragmented packets.</td>
</tr>
<tr>
<td>8-bit Time to Live (TTL)</td>
<td>Time to Live field.</td>
</tr>
<tr>
<td>8-bit Protocol</td>
<td>Protocol number of the upper-layer protocol.</td>
</tr>
<tr>
<td>16-bit Header Checksum</td>
<td>Checksum of the header.</td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td>Source Internet Protocol (IP) address.</td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td>Destination IP address.</td>
</tr>
<tr>
<td>Options (if any)</td>
<td>Options for the packet, if any.</td>
</tr>
<tr>
<td>Payload</td>
<td>Payload of the packet.</td>
</tr>
</tbody>
</table>
## IP Packet Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>4-bit</td>
<td>Version number of the IP protocol version.</td>
</tr>
<tr>
<td>Header Length</td>
<td>4-bit</td>
<td>Header length in 32-bit units.</td>
</tr>
<tr>
<td>Type of Service (TOS)</td>
<td>8-bit</td>
<td>Type of Service field (TOS).</td>
</tr>
<tr>
<td>Total Length (Bytes)</td>
<td>16-bit</td>
<td>Total packet length in bytes.</td>
</tr>
<tr>
<td>Identification</td>
<td>16-bit</td>
<td>Identification number of the packet.</td>
</tr>
<tr>
<td>Flags</td>
<td>3-bit</td>
<td>Flags indicating fragment status.</td>
</tr>
<tr>
<td>Fragment Offset</td>
<td>13-bit</td>
<td>Fragment offset in units of 8 bytes.</td>
</tr>
<tr>
<td>Protocol</td>
<td>8-bit</td>
<td>Protocol number of the upper-layer protocol.</td>
</tr>
<tr>
<td>Header Checksum</td>
<td>16-bit</td>
<td>Header checksum of the IP packet.</td>
</tr>
<tr>
<td>Source IP Address</td>
<td>32-bit</td>
<td>Source IP address of the packet.</td>
</tr>
<tr>
<td>Destination IP Address</td>
<td>32-bit</td>
<td>Destination IP address of the packet.</td>
</tr>
<tr>
<td>Options (if any)</td>
<td></td>
<td>Options (if any)</td>
</tr>
<tr>
<td>Payload</td>
<td></td>
<td>Payload of the packet.</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>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

- Checksum is arithmetic, not CRC...
  - To allow easily modification of the packet by the network
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 (but that is 'assume drop' based on layer 2 error detection)
  - 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:
  - Each router announces what networks it can provide and the path onward
  - *Most precise* route with the shortest path and no loops preferred
Packet Routing on the Internet: Border Gateway Protocol & Routing Tables

Sender

AS 1

AS 2

AS 3

AS 4

AS 5

AS 6

Recipient

AS4->AS6->Recipient

AS5->AS6->Recipient

AS6->Recipient

AS6->Recipient

AS6->Recipient
Remarks

• This is a network of networks
  • Its 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 other such considerations
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

Host A

Router 1

Router 2

Router 3

Router 4

Router 5

Router 6

Router 7

Host B

Host C

Host D

Host E
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

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
<th>Byte 4</th>
<th>Byte 5</th>
<th>Byte 6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Process B

on host H2

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
<th>Byte 4</th>
<th>Byte 5</th>
<th>Byte 6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bidirectional communication:

There are two separate byte streams, one in each direction.
TCP

<table>
<thead>
<tr>
<th>Layer</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Application</td>
</tr>
<tr>
<td>4</td>
<td>Transport</td>
</tr>
<tr>
<td>3</td>
<td>(Inter)Network</td>
</tr>
<tr>
<td>2</td>
<td>Link</td>
</tr>
<tr>
<td>1</td>
<td>Physical</td>
</tr>
</tbody>
</table>

TCP Header Fields:
- Source port
- Destination port
- Sequence number
- Acknowledgment
- HdrLen
- Flags
- Advertised window
- Checksum
- Urgent pointer
- Options (variable)
- Data
TCP

These plus IP addresses define a given connection

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

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></td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

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

Host B

- TCP Data
- ACK sequence number from B = next expected byte
**TCP**

<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>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>HdrLen</th>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Options (variable) | Data

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
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: 
   
   SrcA=1.2.1.2, SrcP=3344,
   DstA=9.8.7.6, DstP=80, SYN, Seq = x

2. Server: 
   
   SrcA=9.8.7.6, SrcP=80,
   DstA=1.2.1.2, DstP=3344, SYN+ACK, Seq = y, Ack = x+1

3. Client: 
   
   SrcA=1.2.1.2, SrcP=3344,
   DstA=9.8.7.6, DstP=80, ACK, Seq = x+1, Ack = y+1

4. Server: 
   
   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

5. Client: 
   
   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

6. Server: 
   
   SrcA=9.8.7.6, SrcP=80, DstA=1.2.1.2, DstP=3344,
   ACK, Seq = y+1, Ack = x+16, Data = “200 OK ... <html> ...”
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

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.
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**

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

**Attacker’s goal:**

1. SrcA=1.2.1.2, SrcP=5566, DstA=9.8.7.6, DstP=80, ACK, Seq = z+1, ACK = y+1
2. 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

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
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

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)