Computer Security

CMPS122

Winter 2019

DNS, TLS, and DNSSEC
Notices

• **Lab 3 out**: Three phases:
  - Phase 1 due 23:59 Friday March 1
  - Phase 2 due 23:59 Friday March 8
  - Phase 3 due 23:59 Friday March 15

• **Homework #4** cancelled (free points!)
TCP Vulnerabilities: Recap

• An attacker who **observes** 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

• An attacker who can **predict the ISN** chosen by a server can “**blind spoof**” a connection to that server
  – Makes it appear that **friendly host** has connected, and has sent data of the attacker’s choosing, when in fact it hasn’t
  – Undermines any security based on trusting **friendly host**’s IP address
  – Allows attacker to **cast suspicion** on **friendly host** and/or avoid detection
  – Fixed (mostly) today by choosing random ISNs
DNS Operation & Vulnerabilities
3. Finding the address of google.com

google.com’s IP address is 172.217.11.174
Host Names vs. IP Addresses

• Host Names
  – Examples: www.cnn.com and bbc.co.uk
  – Mnemonic name understood by humans
  – Variable length, full alphabet of characters
  – Provide little (if any) information about location

• IP Addresses
  – Examples: 64.236.16.20 and 212.58.224.131
  – Numerical address understood by routers
  – Fixed length, binary number
  – Hierarchical, related to host location
Mapping Names to Addresses

- **Domain Name System** (DNS)
  - Hierarchical name space divided into sub-trees ("zones")
    - E.g. .edu .ucsc.edu .soe.ucsc.edu
  - Zones distributed over collection of DNS name servers

- Hierarchy of DNS servers
  - **Root** (hardwired into other servers)
  - **Top-level Domain** (TLD) servers
    - E.g. .com .org .uk .biz .tv
  - **Authoritative** servers (e.g. for facebook.com)

- End systems (Alice’s Laptop) configured with IP address of a local DNS **resolver** to contact for their lookups
DNS Lookup via a Resolver

1. User queries the local DNS server with the name "nogbad.soe.ucsc.edu".
2. The local DNS server forwards the query to the Root DNS server ".".
3. The Root DNS server forwards the query to the Top Level Domain DNS server "edu".
4. The Top Level Domain DNS server forwards the query to the Authoritative DNS server "ucsc.edu, soe.ucsc.edu".
5. The Authoritative DNS server returns the IP address 192.168.56.101.
6. The Top Level Domain DNS server forwards the query back to the local DNS server with the IP address 192.168.56.254.
7. The local DNS server forwards the query to the local DNS server "Resolver".
8. The local DNS server returns the IP address 192.168.56.101 to the user.
DNS Threats

- DNS is on the **critical path** for just about everything we do
  - Maps hostnames ⇔ IP addresses
  - Design only scales if we can **minimize lookup traffic**
    - No. 1 traffic reduction mechanism: **CACHING**
    - No. 2 traffic reduction mechanism: return not only answers to queries, but **ADDITIONAL INFO** that will almost certainly be needed shortly

- What if attacker eavesdrops on our DNS queries?
  - Simple to then redirect us with spoofed (mis)information

- Consider attackers who **can’t eavesdrop** - but still aim to manipulate us via the mechanics of the protocol (how it functions)

- Directly interacting with DNS: **dig** utility on Unix / Unix-Like
  - Allows querying of DNS system
  - Dumps fields found in DNS responses
$ dig nogbad.soe.ucsc.edu A

; <<>> DiG 9.8.3-P1 <<>> nogbad.soe.ucsc.edu A
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 46385
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 2, ADDITIONAL: 4

;; QUESTION SECTION:
;nogbad.soe.ucsc.edu. IN A

;; ANSWER SECTION:
nogbad.SOE.UCSC.EDU. 28800 IN A 128.114.59.29

;; AUTHORITY SECTION:
SOE.UCSC.EDU. 43200 IN NS adns1.ucsc.edu.
SOE.UCSC.EDU. 43200 IN NS adns2.ucsc.edu.

;; ADDITIONAL SECTION:
adns1.ucsc.edu. 86400 IN A 128.114.100.100
adns2.ucsc.edu. 86400 IN A 128.114.100.200
adns1.ucsc.edu. 86400 IN AAAA 2607:f5f0:2::100
adns2.ucsc.edu. 86400 IN AAAA 2607:f5f0:2::200

;; Query time: 1 msec
;; SERVER: 128.114.142.6#53(128.114.142.6)
;; WHEN: Wed Feb 14 13:27:17 2018
;; MSG SIZE  rcvd: 200
Use Unix “dig” utility to look up IP address (“A”) for hostname eecs.mit.edu via DNS

; ; <<>> 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 W2ONS.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
W2ONS.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 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

This is dig identifying its version and the query it is attempting to look up
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 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

Status values returned from the remote name server queried by dig
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. 2160 IN A

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

Including 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:
mits.edu. 11088 IN NS BITSY.mits.edu.
mits.edu. 11088 IN NS ONS.mits.edu.
mits.edu. 11088 IN NS RAWB.mits.edu.

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

The name server echoes back the question that it is answering as the first part of its reply.
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: 13208
;; 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

"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.
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.
mit.edu.
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

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, which stands for “Address”), and an associated value
dig eecs.mit.edu A

"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:
;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 Packets

Ethernet Header (Physical/Link Layers)

IP Header (Network Layer)

- Source Port
- Destination Port
- Length
- Checksum
- Identification
- Flags
- No. of Question RRs
- No. of Answer RRs
- No. of Authority RRs
- No. of Additional RRs

Questions
Answers
Authorities
Additional Informations
**DNS Protocol**

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

Primarily uses the **UDP** transport protocol, which is what we’ll assume from now on

Typically uses **port 53** for both clients and servers

<table>
<thead>
<tr>
<th>Ethernet Header (Physical/Link Layers)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IP Header (Network Layer)</strong></td>
</tr>
<tr>
<td>Source Port</td>
</tr>
<tr>
<td>Length</td>
</tr>
<tr>
<td>Identification</td>
</tr>
<tr>
<td>No. of Question RRs</td>
</tr>
<tr>
<td>No. of Authority RRs</td>
</tr>
<tr>
<td>Questions</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
### DNS Protocol

#### Ethernet Header (Physical/Link Layers)

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

#### IP Header (Network Layer)

<table>
<thead>
<tr>
<th>Length</th>
<th>Checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Identification</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No. of Question RRs</th>
<th>No. of Answer RRs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No. of Authority RRs</th>
<th>No. of Additional RRs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Questions

- 16 bit No. for query (reply to query uses same No.)

- 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** (seconds) for caching (not shown)
Let's look at a flaw in the original DNS design that's since been fixed.

What if the *mit.edu* server is untrustworthy? Could it’s operator steal or eavesdrop on all our *facebook.com* traffic?

```
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. 21088 IN A 18.71.0.151

;; AUTHORITY SECTION:
mit.edu. 11088 IN NS W2ONS.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
W2ONS.mit.edu. 126738 IN A 18.70.0.160
```
What would happen if the mit.edu server is returns the following to us instead of the legitimate information?
We’d dutifully store in our cache a mapping of www.facebook.com to an IP address under MIT’s control. (It could have been any IP address they wanted, not just one of theirs.)
In this case they chose to make the mapping disappear after 30 seconds. They could have made it persist for weeks, or disappear even quicker.
Next time one of our clients wants to connect to www.facebook.com, it will ask our resolver for the corresponding IP address. The resolver will find the answer in its cache and return 18.6.6.6 😞
`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. 11088 IN NS BITSY.mit.edu.
mit.edu. 11088 IN NS W20NS.mit.edu.

;; AUTHORITY SECTION:
www.facebook.com 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
```
dig eecs.mit.edu A

Don’t accept Additional records unless they’re for the domain of the name server we queried

E.g. contacting a name server for 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.
Don’t accept Additional records unless they’re for the domain of the name server we queried.

E.g. contacting a name server for `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.

---

**bailiwick**

/nəˈwɪk/ (noun)

1. One’s sphere of operations or particular area of interest. 
   "you never give the presentations—that’s my bailiwick"
DNS Protocol Recap

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

Primarily uses the **UDP** on **port 53** for both clients and servers.

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** (seconds) for caching (not shown).

**Threat:** Cache Poisoning

**Defense:** Bailiwick Checking
DNS Threats: Blind Spoofing

- Say we look up `mail.google.com` - how can an off-path attacker feed us a **bogus Answer** before the legitimate server replies?
- How can such a remote attacker even know we are looking up `mail.google.com`?
- Suppose we visit a web page under the attacker’s control that includes this snippet of HTML:
  ```html
  <img src="http://mail.google.com/…">
  ```
  Browser will try to fetch an image from `mail.google.com`
- To do that, the browser first has to look up the IP address associated with `mail.google.com`
DNS Threats : Blind Spoofing

- Once the attackers know the browser is looking up mail.google.com, they just have to guess the **Identification** field, and reply before the legitimate server
- **How hard is that?**
- Originally, identification field incremented by 1 for each request. **How would attacker guess it?**
  - <img src="http://www.attacker.com/...">  
  
  Observe returned genuine DNS ID K
  
  <img src="http://mail.google.com/...">  
  
  Then fake this one with K+1
- **Defense?**
  - **Randomisation** of the DNS ID

<table>
<thead>
<tr>
<th>Ethernet Header (Physical/Link Layers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP Header (Network Layer)</td>
</tr>
<tr>
<td>Source Port</td>
</tr>
<tr>
<td>Destination Port</td>
</tr>
<tr>
<td>Length</td>
</tr>
<tr>
<td>Checksum</td>
</tr>
<tr>
<td>Identification</td>
</tr>
<tr>
<td>Flags</td>
</tr>
<tr>
<td>No. of Question RRs</td>
</tr>
<tr>
<td>No. of Answer RRs</td>
</tr>
<tr>
<td>No. of Authority RRs</td>
</tr>
<tr>
<td>No. of Additional RRs</td>
</tr>
<tr>
<td>Questions</td>
</tr>
<tr>
<td>Answers</td>
</tr>
<tr>
<td>Authorities</td>
</tr>
<tr>
<td>Additionals</td>
</tr>
</tbody>
</table>
DNS Threats: Blind Spoofing

- Once we randomize the Identification, attacker has a 1/65,536 chance of guessing it correctly.
- **Are we safe now?**
- **Remember:** attacker can send *lots* of replies, not just one!
- **However:** once a reply from a legitimate server arrives (with correct Identification), it’s cached and the opportunity to poison it is lost.
- **Victim is inoculated!**
- Unless attacker can send 1,000s of replies before legitimate reply arrives, we’re safe!
- **Or are we...?**

---

<table>
<thead>
<tr>
<th>Ethernet Header (Physical/Link Layers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP Header (Network Layer)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Source Port</td>
</tr>
<tr>
<td>Length</td>
</tr>
<tr>
<td>Identification</td>
</tr>
<tr>
<td>No. of Question RRs</td>
</tr>
<tr>
<td>No. of Authority RRs</td>
</tr>
<tr>
<td>Questions</td>
</tr>
<tr>
<td>Answers</td>
</tr>
<tr>
<td>Authorities</td>
</tr>
<tr>
<td>Additionals</td>
</tr>
</tbody>
</table>

**16 bit Random**
No. for query
Kaminsky Blind Spoofing

• Two key ideas:
  – Spoof uses **Additional** field (rather than **Answer**)
  – Attacker can get around caching of legitimate replies by generating a **series** of **different** name lookups:

  `<img src="http://random1.google.com" ...>`
  `<img src="http://random2.google.com" ...>`
  `<img src="http://random3.google.com" ...>`
  ...
  `<img src="http://randomN.google.com" ...>`
Kaminsky Blind Spoofing

For each lookup of randomk.google.com, attacker spoofs a series of records like this, each with a different Identifier.

Once attacker wins the race, the record for mail.google.com is comprehensively poisoned... but so is the cached DNS record for google.com's name server - so any future xxx.google.com lookups go through the attacker's machine!
Blind Spoofing Defenses

- **Central problem**: All that tells a client they should accept a response is that it matches the Identification field.

- With only 16 bits, it **lacks sufficient entropy**: even if truly random, the search space an attacker must brute force is far too small.

- **Where can we get more entropy?**

  (without requiring a protocol change)
Blind Spoofing Defenses

- For requestor to receive DNS reply, it needs correct Identification and correct ports
- On request, **Destination port** = 53
- **Source port** usually also 53 - but not fundamental, just convenient
Blind Spoofing Defenses

- For requestor to receive DNS reply, needs correct Identification and correct ports
- On request, Destination port = 53
- Source port usually also 53 - but not fundamental, just convenient
- **Solution:** Client uses random source port
- Now blind attacker doesn’t know correct destination port to use in reply
- **32 bits of entropy** makes it orders of magnitude harder for attacker to guess all the necessary fields and dupe victim into accepting spoof response 😊
- This "secures" DNS against blind spoofing...
  
  *But not all resolvers have implemented random source ports 😞*
DNS Security Issues: Summary

- Attackers can **strike opportunistically** rather than eavesdropping
  - Cache poisoning originally only required victim to look up some name under the attackers’ control – this has now been fixed

- Attackers can **manipulate victims into vulnerable activity**
  - e.g., `<img src...>` in web page to force DNS lookups

- **Crucial** for identifiers to have **sufficient entropy**
  - i.e. many bits of unpredictability

- **Attacks only get better**: threats that appear technically challenging can quickly become practical due to unexpected attacker “smarts”
Channel vs. Object Security

- **Channel Security** = securing a means of communication
- **Object Security** = securing data values (message contents)
- **CIA** (confidentiality, integrity, authentication) applies to both
- **Transport Layer Security** (TLS) provides channel security
- **Secure DNS** (DNSSEC) provides object security for DNS results
Building Secure End-to-End Channels: SSL / TLS

• SSL = **Secure Sockets Layer** (predecessor)
• TLS = **Transport Layer Security** (standard)
• Note: Terms used interchangeably
• Motivation: Provide a means to secure *any* TCP based application
  – Secure = encryption/confidentiality + integrity + authentication
• API similar to “socket” interface for insecure network programming
  – Relatively easy to convert an application to be secure
SSL/TLS in Network Layering

Insecure Stack

Secure Stack
NOTE: All of this is in the clear, i.e. plain text

- Browser (client) connects via TCP to Amazon's HTTPS server
- Client picks 256-bit random number $R_B$ and sends over list of cryptography protocols (cipher suites) it supports
- Server picks 256-bit random number $R_S$ and selects a cipher suite to use for this session
- Server sends over its certificate
- Client validates server certificate
HTTPS Connections via SSL/TLS (2 of 2)

- For RSA, browser constructs 368-bit “Premaster Secret” PS
- Browser sends PS encrypted using Amazon's public RSA key $K_{Amazon}$
- Using PS, $R_B$, and $R_s$, browser & server derive symmetric cipher keys ($C_B$, $C_S$) & MAC integrity keys ($I_B$, $I_S$)
  - One pair to use in each direction
- Browser & server exchange MACs computed over entire dialog so far
- If a good MAC, Browser displays...
- All subsequent communication encrypted with symmetric cipher (e.g. AES128) cipher keys, MACs
  - Messages numbered to thwart replay attacks

Here’s my cert
Approx. 2-3 KB

Browser

| PS | $R_B$ | $R_s$ | $K_{Amazon}$ |

Amazon

| PS | $C_S$ | $I_S$ |

{$PS$}$_{K_{Amazon}}$

$MAC$(dialog,$I_B$)

$MAC$(dialog,$I_S$)

{$M_1$, $MAC$(M$_1$,I$_B$)}$_{C_B}$

{$M_2$, $MAC$(M$_2$,I$_S$)}$_{C_S}$
HTTPS Alternative Key Exchange

- For **Diffie-Hellman**, server generates random $a$, sends public parameters $g$ and $p$, and $g^a \mod p$ all **signed** with server’s public key.
- Browser verifies signature.
- Browser generates random $b$, computes $g^b \mod p$.
- Browser sends $g^b \mod p$ to server.
- Browser and server independently compute a shared secret $PS = g^{ab} \mod p$.
- Remainder is as before: from $PS$, $R_B$, and $R_S$, browser & server derive symmetric cipher keys ($C_B$, $C_S$) and MAC integrity keys ($I_B$, $I_S$) etc…
Secure Browsing
The CA is Symantec

Cipher suite used for the connection
### Certificate Viewer: “www.amazon.com”

**This certificate has been verified for the following uses:**

- **SSL Client Certificate**
- **SSL Server Certificate**

#### Issued To
- **Common Name (CN)**: www.amazon.com
- **Organization (O)**: Amazon.com, Inc.
- **Organizational Unit (OU)**: <Not Part Of Certificate>

#### Issued By
- **Common Name (CN)**: Symantec Class 3 Secure Server CA - G4
- **Organization (O)**: Symantec Corporation
- **Organizational Unit (OU)**: Symantec Trust Network

#### Period of Validity
- **Begins On**: 19 September 2017
- **Expires On**: 21 September 2018

#### Fingerprints
Who CERT was issued to
PKCS #1 = “Standard RSA encryption/signing” algorithms
Nice long 2048-bit public key
This cert is valid for associating with **any** of these DNS names

Browser will only honor this cert if the URL we’re accessing is served by one of these domains.
The key can be used for both encryption and digital signatures.

If the browser doesn’t understand this “Certificate Key Usage” extension, it must reject the cert.
Where to download the CA’s certificate revocation list from

Why is it ok to download via HTTP?

Because the CRL is signed using the CA’s public key, which we trust

Where to download the CA’s certificate revocation list from
Here is where to access the CA’s Online Certificate Status Protocol server to check for revocations.
The CA has signed a SHA-256 hash of this cert using RSA
Here’s the actual signature, which the browser needs to validate against an SHA256 hash computed over the cert.
Certificates

- Cert = signed statement about someone’s public key
  - Note that a cert does not say anything about the identity of who gives you the cert
  - It simply states a given public key $K_{Bob}$ belongs to Bob ...
    - ... and backs up this statement with a digital signature made using a different public/private key pair, say from Verisign
- Bob then can prove his identity to you by you sending him something encrypted with $K_{Bob}$ ...
  - ... which he then demonstrates he can read
- ... or by signing something he demonstrably uses
- Works provided you trust that you have a valid copy of Verisign’s public key ...
  - ... and you trust Verisign to use prudence when she signs other people’s keys
Validating Amazon’s Identity

• Browser compares domain name in cert with URL
  – This provides an end-to-end property
    • As opposed to a cert associated with an IP address

• Browser accesses separate cert belonging to issuer
  – These are hardwired into the browser; i.e. they are trusted
  – There could be a chain of these ...

• Browser applies issuer’s public key to verify signature, obtaining hash of whatever the issuer signed
  – Compares with its own SHA-256 hash of Amazon’s cert

• If hashes match, browser has high confidence it’s talking to Amazon
  – Assumes signatory is trustworthy, i.e.
    • Assumes they didn’t lose private key
    • Assumes they didn’t sign thoughtlessly
End-to-End Protection?

- Attacker runs a sniffer to capture our WiFi session?
- DNS cache poisoning? (client goes to wrong server)
- Attacker hijacks our connection and injects new traffic?
- DHCP spoofing? (client goes to wrong server)
- Attacker manipulates routing to run us by an eavesdropper or take us to the wrong server?
- Attacker slips in as a Man In The Middle?
End-to-End Protection

• Attacker runs a sniffer to capture our WiFi session?
  – Encrypted communication is unreadable 😇

• DNS cache poisoning? (client goes to wrong server)
  – Impersonation detected since attacker lacks a valid cert 😇

• Attacker hijacks our connection and injects new traffic?
  – Receiver rejects it due to failed integrity check 😇

• DHCP spoofing? (client goes to wrong server)
  – We detect impersonation since attacker lacks a valid cert 😇

• Attacker manipulates routing to run us by an eavesdropper or take us to the wrong server?
  – They can’t read and we can detect impersonation 😇

• Attacker slips in as a Man In The Middle?
  – They can’t read, they can’t inject 😇
  – They can’t replay previous encrypted traffic 😇
Validating Amazon’s Identity

• Browser accesses separate cert belonging to issuer
  – These are **hardwired** into the browser – i.e. **implicitly trusted**

• **What if browser can’t find a cert for the issuer?**
Your connection is not secure

The owner of untrusted-root.badssl.com has configured their website improperly. To protect your information from being stolen, Firefox has not connected to this website.

Learn more...

Go Back

Advanced

Report errors like this to help Mozilla identify and block malicious sites

Safari can't verify the identity of the website “untrusted-root.badssl.com”.

The certificate for this website is invalid. You might be connecting to a website that is pretending to be “untrusted-root.badssl.com”, which could put your confidential information at risk. Would you like to connect to the website anyway?

Show Certificate
Cancel Continue
Validating Amazon’s Identity

• Browser accesses separate cert belonging to issuer
  – These are hardwired into the browser - trusted!

• What if browser can’t find a cert for the issuer?

• If browser can’t find the cert, it warns the user that site has not been verified and shouldn’t be trusted
  – User can still proceed, just without authentication (bad, bad user 😞)

• **Which end-to-end security properties do we lose if we incorrectly trust that the site is whom we think it is?**

• **All of them!**
  – Goodbye confidentiality, integrity, authentication
  – Attacker can read everything, modify anything, impersonate anyone
SSL/TLS Limitations

• Properly used, SSL / TLS provides powerful end-to-end protections

• **So why not use it for everything?**
  – Computational cost of public-key crypto
    • Requires non-trivial CPU processing (but today a relatively minor issue)
    • Symmetric key crypto on most modern hardware is non-issue
  – Hassle of buying/maintaining certs (minor, but still annoying)
  – DoS amplification
    • Client can force server to undertake expensive public key operations
      – Requires established TCP connection, and given that, there are often juicier targets like inappropriately exposed back-end databases to have a go at
  – Integrating with other sites that don’t use HTTPS
  – Latency: extra round trips ⇒ pages take longer to load
SSL/TLS Limitations Cont.

- **Problems that SSL / TLS does not take care of?**
  - TCP-level denial of service (or any other DoS)
    - SYN flooding
    - RST injection (but does protect against data injection)
  - SQL Injection / XSS / server-side bugs
  - Browser bugs
  - User flaws
    - Weak passwords
    - Phishing
  - Vulnerabilities introduced by HTTP compatibility
Add Alexa to your speakers

echo dot $49.99
SSL Strip Attack

This is sent \textit{in the clear}, using HTTP rather than HTTPS. A MITM can connect to Amazon using HTTPS, but relay the content to user using HTTP, altering whatever they wish 😞. Attacker rewrites any embedded HTTPS URLs to HTTP.
HTTP Strict Transport Security (HSTS)

• To defend against SSL Strip attacks, a web server can return (during HTTPS connection) header directives such as:

  Strict-Transport-Security: max-age=31536000 includeSubDomains

• Directs browser to:
  – Only connect to that site using HTTPS (expires in 1yr)
  – Promote any HTTP links in pages to HTTPS
  – Don’t allow connections with cert errors to proceed

• Similar to TOFU, requires a safe initial connection
  – Otherwise, MITM attacker could strip out the header

• Many browsers today use a predefined list of HSTS sites
  – https://hstspreload.org/
TLS/SSL Trust Issues

• “Commercial certificate authorities protect you from anyone from whom they are unwilling to take money.”
  – Matt Blaze, circa 2001

• So how many CAs do we have to worry about, anyway?
<table>
<thead>
<tr>
<th>Name</th>
<th>Kind</th>
<th>Expires</th>
<th>Keychain</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA Certificate Services</td>
<td>certificate</td>
<td>Dec 31, 2028, 3:59:59 PM</td>
<td>System Roots</td>
</tr>
<tr>
<td>Actalis Authentication Root CA</td>
<td>certificate</td>
<td>Sep 22, 2030, 4:22:02 AM</td>
<td>System Roots</td>
</tr>
<tr>
<td>AddTrust Class 1 CA Root</td>
<td>certificate</td>
<td>May 30, 2020, 3:38:31 AM</td>
<td>System Roots</td>
</tr>
<tr>
<td>AddTrust External CA Root</td>
<td>certificate</td>
<td>May 30, 2020, 3:48:38 AM</td>
<td>System Roots</td>
</tr>
<tr>
<td>AddTrust Public CA Root</td>
<td>certificate</td>
<td>May 30, 2020, 3:41:50 AM</td>
<td>System Roots</td>
</tr>
<tr>
<td>AddTrust Qualified CA Root</td>
<td>certificate</td>
<td>May 30, 2020, 3:44:50 AM</td>
<td>System Roots</td>
</tr>
<tr>
<td>Admin-Root-CA</td>
<td>certificate</td>
<td>Nov 9, 2021, 11:51:07 PM</td>
<td>System Roots</td>
</tr>
<tr>
<td>AffirmTrust Commercial</td>
<td>certificate</td>
<td>Dec 31, 2030, 6:06:08 AM</td>
<td>System Roots</td>
</tr>
<tr>
<td>AffirmTrust Networking</td>
<td>certificate</td>
<td>Dec 31, 2030, 6:08:24 AM</td>
<td>System Roots</td>
</tr>
<tr>
<td>AffirmTrust Premium</td>
<td>certificate</td>
<td>Dec 31, 2040, 6:10:36 AM</td>
<td>System Roots</td>
</tr>
<tr>
<td>AffirmTrust Premium ECC</td>
<td>certificate</td>
<td>Dec 31, 2040, 6:20:24 AM</td>
<td>System Roots</td>
</tr>
<tr>
<td>ANF Global Root CA</td>
<td>certificate</td>
<td>Jun 5, 2033, 10:45:38 AM</td>
<td>System Roots</td>
</tr>
<tr>
<td>Apple Root CA</td>
<td>certificate</td>
<td>Feb 9, 2035, 1:40:36 PM</td>
<td>System Roots</td>
</tr>
<tr>
<td>Apple Root CA - G2</td>
<td>certificate</td>
<td>Apr 30, 2039, 11:10:09 AM</td>
<td>System Roots</td>
</tr>
<tr>
<td>Apple Root CA - G3</td>
<td>certificate</td>
<td>Apr 30, 2039, 11:19:08 AM</td>
<td>System Roots</td>
</tr>
<tr>
<td>Apple Root Certificate Authority</td>
<td>certificate</td>
<td>Feb 9, 2025, 4:18:14 PM</td>
<td>System Roots</td>
</tr>
<tr>
<td>ApplicationCA</td>
<td>certificate</td>
<td>Dec 12, 2017, 7:00:00 AM</td>
<td>System Roots</td>
</tr>
<tr>
<td>ApplicationCA2 Root</td>
<td>certificate</td>
<td>Mar 12, 2033, 7:00:00 AM</td>
<td>System Roots</td>
</tr>
<tr>
<td>Autoridad de...nal CIF A62634068</td>
<td>certificate</td>
<td>Dec 31, 2030, 12:38:15 AM</td>
<td>System Roots</td>
</tr>
<tr>
<td>Autoridad de...Estado Venezolano</td>
<td>certificate</td>
<td>Dec 17, 2017, 7:00:00 AM</td>
<td>System Roots</td>
</tr>
<tr>
<td>Baltimore CyberTrust Root</td>
<td>certificate</td>
<td>May 12, 2020, 3:48:38 AM</td>
<td>System Roots</td>
</tr>
<tr>
<td>Belgium Root CA2</td>
<td>certificate</td>
<td>Dec 15, 2039, 2:00:00 AM</td>
<td>System Roots</td>
</tr>
</tbody>
</table>

**175 Root CAs ?!?!**
TLS/SSL Trust Issues

• “Commercial certificate authorities protect you from anyone from whom they are unwilling to take money”
  – Matt Blaze, circa 2001

• So how many CAs do we have to worry about, anyway?

• Of course, it's not just their greed that matters ...

• And it's not just their diligence & security that matters ...

UCSC BSOE CMPS122 Winter 2018. Copyright © 2018 David C. Harrison. All Rights Reserved.
Fraudulent Google certificate points to Internet attack

Is Iran behind a fraudulent Google.com digital certificate? The situation is similar to one that happened in March in which spoofed certificates were traced back to Iran.

by Elinor Mills | August 29, 2011 1:22 PM PDT

A Dutch company appears to have issued a digital certificate for Google.com to someone other than Google, who may be using it to try to re-direct traffic of users based in Iran.

Yesterday, someone reported on a Google support site that when attempting to log in to Gmail the browser issued a warning for the digital certificate used as proof that the site is legitimate, according to this thread on a Google support forum site.
TLS/SSL Trust Issues

- “Commercial certificate authorities protect you from anyone from whom they are unwilling to take money.”  
  – Matt Blaze, circa 2001

- So how many CAs do we have to worry about, anyway?
- Of course, it's not just their greed that matters ...
- And it's not just their diligence & security that matters ...

- “A decade ago, I observed that commercial certificate authorities protect you from anyone from whom they are unwilling to take money. That turns out to be wrong; they don't even do that much.”  
  – Matt Blaze, circa 2010

- You also have to trust the developers of crypto libraries...
  – Both for clients when validating certs
  – And servers when generating certs
This is Apple SSL code to verify Diffie-Hellman parameters sent by the server have a valid signature w.r.t. the public key in the server’s certificate.

This computes the hash over the D-H parameters to then compare against the signature.
Securing DNS Lookups
Securing DNS Lookups

• How can we ensure when clients look up names with DNS, they can trust answers they receive?

• Idea 1: **Do DNS lookups over TLS**
  – Assumes either we run DNS over TCP, or we use “Datagram TLS” (out of scope)
  – Issues?
    • Performance: DNS is lightweight, TLS is not
    • Caching: crucial for DNS scaling, but how do we keep authentication assurances?
      – Object security vs. Channel security

• Idea 2: **Make DNS results like certs**
  – i.e. a verifiable signature that guarantees who generated a piece of data
    • Signing happens off-line
Operation of DNSSEC

- **DNSSEC** = Standardized DNS security extensions
  - Currently being deployed, not yet universally adopted

- As a resolver works its way from DNS root down to final name server for a name, at each level it gets a signed statement regarding the key(s) used by the next level
  - This builds up a **chain of trusted keys**
  - Resolver has **root’s key hardwired** into it

- The **final answer** the resolver receives is **signed by that level’s key**
  - Resolver can trust it’s the right key via **chain of support from higher levels**

- All keys and signed results are cacheable
Ordinary DNS

```
Client's Resolver  k.root-servers.net
com. NS a.gtld-servers.net
a.gtld-servers.net A 192.5.6.30
...

Client's Resolver  a.gtld-servers.net
www.google.com A ?
google.com. NS ns1.google.com
ns1.google.com A 216.239.32.10
...

Client's Resolver  ns1.google.com
www.google.com A ?
...
```
DNSSEC (Simplified)

Client’s Resolver

www.google.com A ?

k.root-servers.net

com. **NS** a.gtld-servers.net
a.gtld-servers.net **A** 192.5.6.30
...
com. **DS** hash-of-com’s-key
com. **RRSIG DS** signature-of-that-DS-record-using-root’s-key
DNSSEC (Simplified)

Client’s Resolver

www.google.com A ?

com. **NS** a.gtld-servers.net
a.gtld-servers.net A 192.5.6.30
...
com. **DS** hash-of-com’s-key
com. **RRSIG DS** signature-of-that-DS-record-using-root’s-key

k.root-servers.net

Note: there’s **no signature** over the **NS** or **A** information - If an attacker has fiddled with those, the resolver will ultimately find it has a record for which it can’t verify the signature

The process of retrieving .com’s public key is complex (involves multiple keys) so we’ll defer explanation...

This new RR ("**Delegation Signer**") lets us tell if we have a correct copy of .com’s public key (by comparing hash values)

Same as before

This new RR specifies a **signature** over another RR ... in this case, the signature covers the above **DS** record, and is made using the root’s private key

The resolver has the root’s public key **hardwired** into it - the client will only proceed with DNSSEC if it can validate the signature

---

UCSC BSOE CMPS122 Winter 2018. Copyright © 2018 David C. Harrison. All Rights Reserved.
DNSSEC (Simplified)

The resolver again proceeds to trying one of the name servers it’s learned about - nothing guarantees this is a legitimate name server for the query.

Back comes similar information as before: a way to securely identify google.com’s public key, signed by .com’s key (which the resolver trusts because the root signed information about it)

www.google.com A ?

Client’s Resolver

a.gtld-servers.net

google.com. NS ns1.google.com
ns1.google.com. A 216.239.32.10
...
google.com. DS hash-of-google.com’s-key
google.com. RRSIG DS signature-of-that-DS-record-using-com’s-key
DNSSEC (Simplified)

Client’s Resolver

...
www.google.com. RRSIG A
signature-of-the-A-records-using-
google.com's-key

ns1.google.com

Finally we’ve received the information we wanted (A records for www.google.com)… and a signature over those records 😊

Assuming the signature validates, then because we believe (due to the signature chain) it is indeed from google.com’s key, we can trust this is a correct set of A records … regardless of what name server returned them to us

The resolver tries one of the google.com name servers it’s learned about - again, nothing guarantees this is a legitimate name server for the query.
DNSSEC: Mallory Attacks!

Client’s Resolver → ns1.evil.com

www.google.com A 6.6.6.6

Resolver observes the reply didn’t include a signature, rejects it as insecure
DNSSEC: Mallory Attacks!

Client’s Resolver → www.google.com. A 6.6.6.6

If resolver **did not** receive a signature from .com for evil.com’s key, then it can’t validate this signature and ignores reply since it’s not properly signed


 If resolver **did** receive a signature from .com for evil.com’s key, it knows the key is for evil.com not google.com and ignores it

ns1.evil.com ← www.google.com. A 6.6.6.6
DNSSEC: Mallory Attacks!

If signature **actually** comes from google.com’s key, resolver will believe it ... but no such signature exists unless:

(a) google.com intended to sign the RR, or
(b) google.com’s private key was compromised
DNSSEC : Accessing Keys

To build up the keys needed for validation, client contacts each name server in the DNS hierarchy asking it for all of its associated keys. Here we ask the root for its keys - one of which we already know as our trust anchor.

We can ask for any other keys we need, such as .com’s and google.com’s, in parallel.

Very quickly we’ll have most of the keys we need in our cache.
DNSSEC: Accessing Keys

Each **DNSKEY** is a public key plus a "cryptogoop" description of the associated algorithms e.g. RSA+SHA256.

- **DNSKEY** cryptogoop for root's key-signing key (KSK)
- **DNSKEY** cryptogoop for root's zone-signing key (ZSK)
- **DNSKEY** cryptogoop for possibly other keys
  - ... 
  - **RRSIG DNSKEY** signature-of-those-DNSKEY-records-using-root's-KSK

**Client's Resolver**

The client has a hash of the root’s **KSK** hardwired into its config as a trust anchor.

The **KSK** is used to sign all of the **DNSKEY** entries in the zone.

The **ZSK** is used for signing all of the other **RRSIG** entries in the zone, including DS records for subzones: e.g. .com signs its DS record for google.com using .com’s **ZSK**.

For everything below the root (e.g. .com and google.com) we get a hash of the **KSK** via a DS record, as shown earlier, so we can tell if we get the right **KSK** in a **DNSKEY** entry.

Having separate **key-signing-keys** and **zone-signing-keys** allows a zone to change its ZSK without needing to get its parent to re-sign, since parent only signs the KSK ... enables frequent **key rollover**.
$ dig +dnssec ucsc.edu

; <<>> DiG 9.8.3-P1 <<>> +dnssec ucsc.edu
;;; global options: +cmd
;;; Got answer:
;;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 59731
;;; flags: qr aa rd ra
;;; QUERY SECTION:
;ucsc.edu. IN A

;;; ANSWER SECTION:
ucsc.edu.300 IN A 128.114.109.5
ucsc.edu. 300 IN RRSIG A 10 2 300 2018030225936 2018022620855 21353 ucsc.edu. zIjm5z6I2Rgv8sEIuINA6Ui6ljrCRBcN97uw75QEaNEMYtcE8jPaHbXo
g7k8Yv6u2UAGq8hYULt5k6WWX94+pD0Fb8rWPEOL3jKhlyem7V9D8zICdV8N46pGu6RMKT8z))?I2pC64vJ+q6c55rTGl8z6kYHnph5HFO+sl4aex lt4=

;;; AUTHORITY SECTION:
ucsc.edu. 86400 IN NS adns2.ucsc.edu.
ucsc.edu. 86400 IN NS ns.zocalo.net.
ucsc.edu. 86400 IN NS adns1.ucsc.edu.
ucsc.edu. 86400 IN NS sns-pb.isc.org.
ucsc.edu. 86400 IN NS dns.princeton.edu.
ucsc.edu. 86400 IN RRSIG NS 10 2 86400 20180302142719 20180226141840 21353 ucsc.edu. Uai4tXJ52AL8H3K2uvEWTZ8wDBoxBtwMUxpXAxeCCPtTFQAC4JdLq4Wt

;;; ADDITIONAL SECTION:
ns.zocalo.net. 68781 IN A 157.22.0.254
dns.princeton.edu. 33388 IN A 128.112.129.15
adns1.ucsc.edu. 86400 IN A 128.114.100.100
adns2.ucsc.edu. 86400 IN A 128.114.100.200
sns-pb.isc.org. 7185 IN AAAA 2001:500:2e::1
adns1.ucsc.edu. 86400 IN RRSIG A 10 3 86400 20180302035220 20180226033825 21353 ucsc.edu. TBEQG+/wF3IzVxUNOheQdPmHQaLyfq/e6saEtfDobndRnsxOHeW5UJp

; Query time: 1 msec
; SERVER: 128.114.124.68(128.114.124.68)
; WHEN: Mon Feb 26 18:46:34 2018
; MSG SIZE  rcvd: 2016
Issues with DNSSEC

- **Issue 1: Reponses are BIG**
  - e.g. “dig +dnssec ucsc.edu” returns many kilobytes
  - Potential for DoS amplification
  - Increased latency on low-capacity links
  - Headaches with older libraries that assume replies < 512 Bytes

- **Issue 2: Partial deployment**
  - What do you do with unsigned/unvalidated results?
  - If you trust them, weakens incentive to upgrade
  - If you don’t trust them, a whole lot of things break
Issues with DNSSEC Cont.

• Issue 3: **Management headaches**
  – What happens if you make a mistake updating your site’s keys?
  – Suddenly your entire site breaks 😞

• Issue 4: **Negative results** ("no such name")
  – What statement does the nameserver sign?
  – If "gabluph.google.com" doesn’t exist, then have to do expensive dynamic key-signing for all bogus requests
    • Implies a DoS vulnerability
  – Instead, sign (off-line) statements about the order of names
    • e.g., sign "gabby.google.com is followed by gabrunk.google.com"
    • Thus, client can see that gabluph.google.com can’t exist
  – But: now attacker can enumerate all names that do exist 😞
  – NSEC3 records return hash of next name, but not widely supported
Issues with DNSSEC Cont.

• Issue 5: **Who do you really trust?**
  - For your laptop (say), who does all the “grunt work” of fetching keys & validating DNSSEC signatures?

• Your laptop's local resolver?
  - The one you acquired via DHCP in the coffee shop?
    • i.e. potentially the most untrustworthy part of the DNS resolution process!

• **Alternatives?**
  - Your laptop needs to do all the validation work itself 😞
TLS & DNSSEC Summary

• **TLS**: Channel Security for communication over TCP
  – Confidentiality, Integrity, Authentication
  – Client & server agree on crypto, session keys
  – Underlying security dependent on trust in Certificate Authorities
    • Plus implementers (remember the Apple SSL bug)

• **DNSSEC**: Object Security for DNS results
  – Integrity and Authentication, no Confidentiality
  – No client/server setup “dialog”
  – Tailored to be caching-friendly
  – Underlying security heavily dependent on trust in Root Name Server’s key
    • Plus support provided by every level of DNS hierarchy from Root to final name server and local resolver