NETWORK SECURITY

DNS LABORATORY REPORT

Alessandro Fontana, Davide Pedrotti, Lorenzo Masè

May 06, 2025

Contents

1 Introduction and Laboratory Setup

The following laboratory shows how to implement and execute both DNS cache poisoning and Kaminsky attack, which are two attacks on Domain Name System (DNS). Additionally, it covers DNSSEC, a security extension of DNS that applies cryptography to implement authentication and integrity during communication. We decided to use docker containers to set up the laboratory as they provide isolation and are easy to deploy. The main project directory includes a script named (**build_images.sh**) for building the Docker images, a **solutions/** folder with the solutions of the lab exercises, as well as dedicated folders for each attack and the DNSSEC configuration.

2 DNS

Domain Name System (DNS) is a hierarchical and distributed name service that provides a naming system for computers, services and other resources on the Internet. It translates human-readable domain names (e.g. www.google.com) into machine-readable IP addresses (e.g. 216.58.204.228). For this reason, it is often called "the phonebook of the Internet".

2.1 Terminology

As previously mentioned, DNS is organized as a hierarchical tree structure, and each zone represents a portion of that hierarchy. So, a zone can be defined as a part of the DNS namespace that contains all the DNS records for a certain domain.

During a name resolution for a given domain name, many nameservers are involved, following an ordering based on the hierarchy and they can be divided into two categories: authoritative nameservers and recursive nameservers. An authoritative nameserver is a server that maintains updated all the hostname and IP address associations for a certain zone. It is the master of that zone and it doesn't rely on any other nameservers regarding that zone. Instead, a recursive nameserver is a server that, on behalf of its clients, queries the Internet to resolve domain names for which it is not authoritative. After it receives the answer for a given domain name, it stores the IP address for a certain time inside the cache, defined by the Time To Live (TTL) that it is set by the zone administrator. When it expires, the record is deleted and the nameserver must perform a name resolution again to get the IP. The use of caching is very important, as it allows the recursive nameserver to directly return the response to the client without contacting any other nameservers, improving performance and reducing unnecessary DNS queries.

Finally, the resolver is the client part of the DNS protocol that only asks questions about hostnames. It is very simple, since it is only a file on Linux (`/etc/resolv.conf`), and it knows just enough to contact a nameserver for a name resolution, usually your ISP.

Table 1: DNS record types

Type	**Description**
A	This is an IP address record, and is the most common type of data supported by DNS
NS	This describes a nameserver record responsible for the domain asked about
SOA	The start of authorities record describes data about the zone as defined by the administrator

Moreover, as defined in the table above, there are different types of DNS records, each of which contains different kinds of information necessary for the resolution and management of domain names. Just a clarification: an A record refers to an IPv4 address, while an AAAA record refers to an IPv6 address. Furthermore, regarding NS record, in addition to the name of the nameserver, there is always present an additional section, called "glue", that contains the IP address of the given nameserver.

Note that there are other record types, but we will not see them during the laboratory, so it is pointless to explain them.

2.2 Name resolution

A typical name resolution scenario starts from the client, which generates a DNS query to the recursive nameserver whose IP address is saved in the resolver [1]. If the recursive nameserver doesn't have the IP address of the hostname of the query in the cache, it will traverse through the DNS hierarchy, querying other nameservers starting from the root. From the root, it will receive a list of TLD nameservers that may have the IP address that is searching for, and so on until it contacts an authoritative nameserver of the domain, which answers back with the IP address.

Once the recursive nameserver receives a response, it must verify that the response corresponds to the pending query. This verification process ensures the integrity of the name resolution. To confirm the validity of the response, the recursive nameserver checks that the response arrives on the same UDP port from which the query was sent, that the question section (which is duplicated in the reply) matches the question in the pending query, that the query ID of the reply corresponds to the query ID of pending query and that the authority and additional sections represent names within the same domain as the original question.

2.3 DNS packet

A DNS packet is composed of several fields organized into different sections: the header, the question section, the answer section, the authority section and the additional section.

The header contains information about metadata, like the QID, flags and counters for the next sections. The question section holds the domain name being queried with the type and class of the query. The answer section contains the result of the query and finally, the authority and additional sections include information about authoritative name servers and additional helpful data, such as the IP addresses of those servers.

Note that a DNS packet it is encapsulated into a UDP datagram to improve performance and scalability [2].

3 BIND

Berkeley Internet Name Domain (BIND) is a suite of software for interacting with the DNS. The main components of BIND are [3]: the named.conf file, which is the main configuration file where options are defined, the zone files, which contain information about domains and subdomains, such as A and NS records and the logging configuration file, which defines where and when logs are written. BIND is highly configurable, allowing full control over DNS behavior, including access control, forwarders, source port numbers and more. It also offers full support for DNSSEC, enabling zone signing, key management and validation of responses.

In our project we provide two different DNS configurations, one simulating a topology for a DNS cache poisoning attack and the other regarding a Kaminsky attack. Moreover, there is also present a secure configuration of BIND that apply DNSSEC to mitigate the Kaminsky attack.

4 DNS Cache Poisoning Attack

4.1 Idea

The idea behind this attack is to trick a recursive nameserver into caching a fake record (e.g., `www.google.com` is at 10.0.0.20). If the attack succeeds, a user asking for `www.google.com` will be redirected to the attacker's server instead of the legitimate one (10.0.0.30). This behaviour will last until the TTL of the poisoned record expires, since after the record is deleted, the recursive nameserver has to perform a name resolution again if it receives a query for `www.google.com`. To succeed, the attacker needs to win a race condition against the authoritative nameserver by sending 2^{16} spoofed responses, each with a different query ID, before the authoritative nameserver replies to the recursive nameserver with the legitimate response.

4.2 Attack flow

Figure 1 shows the attack flow of a DNS cache poisoning in our topology. The victim asks for `www.google.com`, but since the recursive nameserver doesn't have the IP address in the cache, it needs to contact the authoritative nameserver to obtain it. Immediately after the recursive nameserver sends the query, the attacker starts flooding all the spoofed responses that contain the IP address of the attacker's server for every possible query ID. If the spoofed response with the correct query ID arrives before the legitimate response, the IP of the attacker's server is saved in the cache. As a result, the victim and all subsequent users that contact the recursive nameserver for `www.google.com` before the TTL expires will be redirected to the attacker's server instead of the legitimate one.

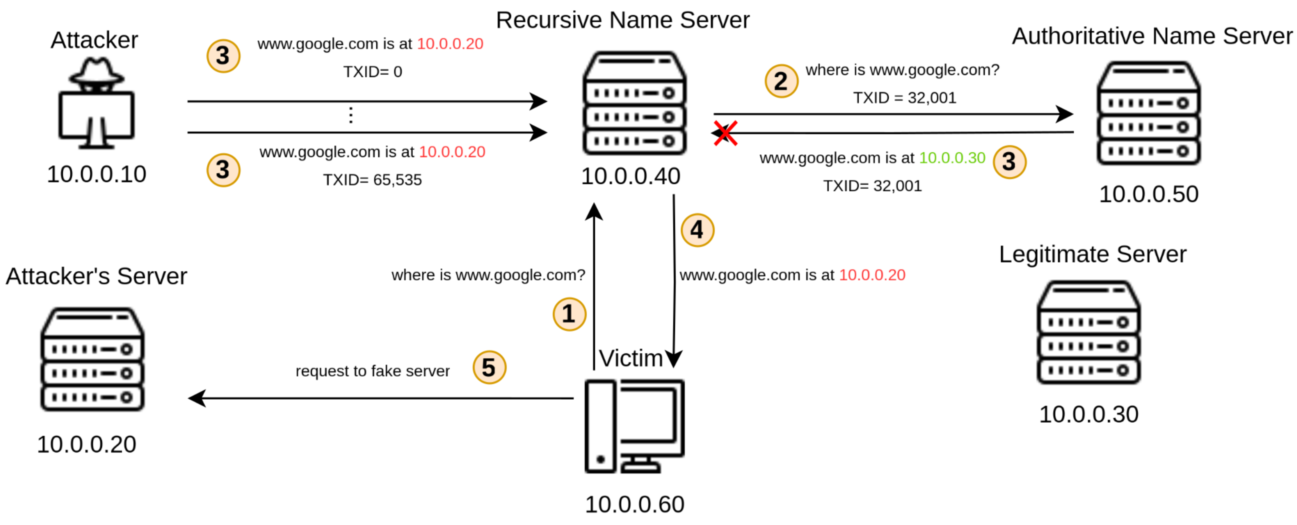
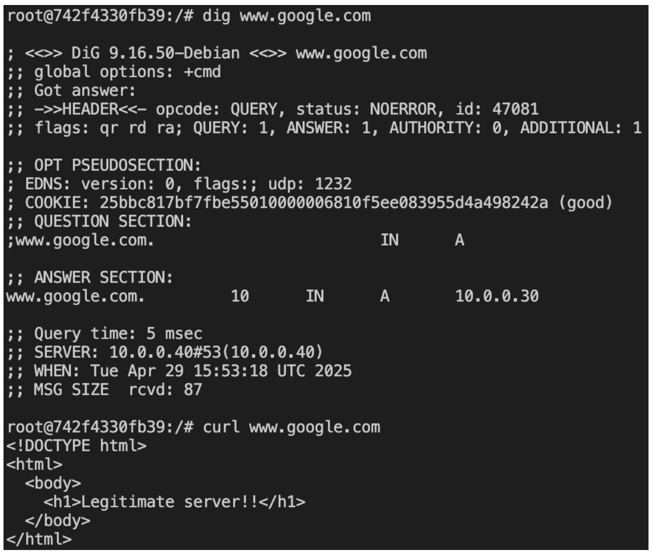

Attacker
10.0.0.10
3 www.google.com is at 10.0.0.20
TXID= 0
3 www.google.com is at 10.0.0.20
TXID= 65,535
Recursive Name Server
10.0.0.40
2 where is www.google.com?
TXID = 32,001
www.google.com is at 10.0.0.30 3
TXID= 32,001
Authoritative Name Server
10.0.0.50
Legitimate Server
10.0.0.30
4 www.google.com is at 10.0.0.20
where is www.google.com? 1
Attacker's Server
10.0.0.20
request to fake server 5
Victim
10.0.0.60

Figure 1: DNS cache poisoning attack

4.3 Laboratory

To successfully perform a DNS cache poisoning, we need to set up the laboratory. First, navigate inside the **cache-poisoning/** directory and run **docker compose up**, which will start the containers. After that, open three terminals and execute **docker exec -it [attacker|victim|authoritative_ns] bash**. To gain a deeper understanding of the exchanged packets, we will use wireshark, so just open another terminal, type **sudo wireshark** (the password is **password**) and inside wireshark GUI select the interface **br-...** and apply the **dns** filter to display only DNS packets exchanged between the containers.



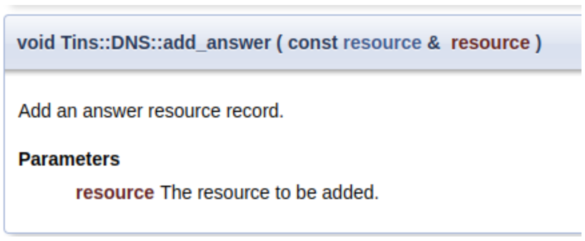
```
root@742f4330fb39:/# dig www.google.com

; <<>> DiG 9.16.50-Debian <<>> www.google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 47081
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; COOKIE: 25bbc817bf7fbe55010000006810f5ee083955d4a498242a (good)
;; QUESTION SECTION:
;www.google.com.                        IN      A

;; ANSWER SECTION:
www.google.com.         10      IN      A       10.0.0.30

;; Query time: 5 msec
;; SERVER: 10.0.0.40#53(10.0.0.40)
;; WHEN: Tue Apr 29 15:53:18 UTC 2025
;; MSG SIZE  rcvd: 87

root@742f4330fb39:/# curl www.google.com
<!DOCTYPE html>
<html>
  <body>
    <h1>Legitimate server!!</h1>
  </body>
</html>
```

Figure 2: Name resolution

The first thing we want to verify is whether the DNS configuration works correctly. To test that, inside the **victim** container, run **dig www.google.com**. `dig` is a command-line tool for querying DNS. As shown in figure 2, we receive an A record in the response that contains the IP address **10.0.0.30**, which corresponds to the legitimate server. We can also verify this using **curl www.google.com** to ensure we are redirected to the legitimate server.

Before launching the attack, we introduce a delay in the responses from the authoritative nameserver. This gives the attacker enough time to flood all the spoofed packets before the authoritative nameserver replies with the legitimate answer. To do this, inside the **authoritative_ns** container, run **tc qdisc add dev eth0 root netem delay 2000ms;**.

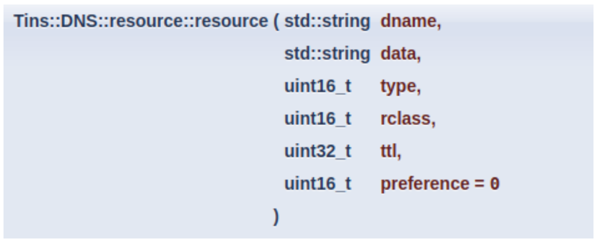
Now we can move into the **attacker** container, where there is a file named **cache_poisoning.cpp**. To correctly perform the attack, you first need to complete the TODOs in the code. Below you find the documentation that you need to solve the exercise. A reference solution can be found at **solutions/cache_poisoning.cpp** if you wish to compare it with your implementation. To launch the attack, run **g++ cache_poisoning.cpp -o cache_poisoning -ltins** followed by **./cache_poisoning**.



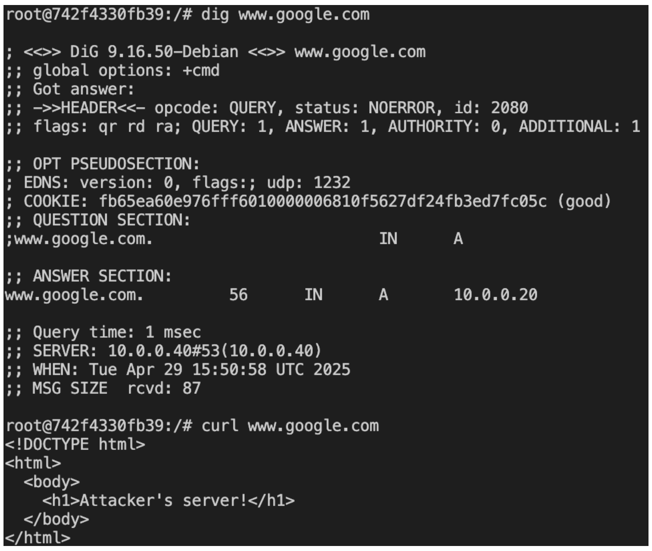
```
void Tins::DNS::add_answer ( const resource & resource )
```

Add an answer resource record.

Parameters
    resource The resource to be added.



```
Tins::DNS::resource::resource ( std::string dname,
                                std::string data,
                                uint16_t    type,
                                uint16_t    rclass,
                                uint32_t    ttl,
                                uint16_t    preference = 0
                              )
```

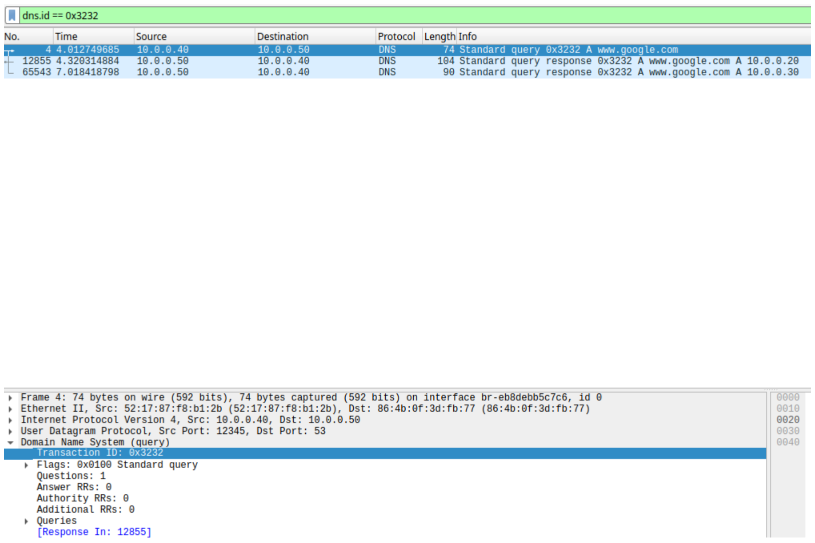


```
root@742f4330fb39:/# dig www.google.com

; <<>> DiG 9.16.50-Debian <<>> www.google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 2080
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; COOKIE: fb65ea60e976fff6010000006810f5627df24fb3ed7fc05c (good)
;; QUESTION SECTION:
;www.google.com.                        IN      A

;; ANSWER SECTION:
www.google.com.         56      IN      A       10.0.0.20

;; Query time: 1 msec
;; SERVER: 10.0.0.40#53(10.0.0.40)
;; WHEN: Tue Apr 29 15:50:58 UTC 2025
;; MSG SIZE  rcvd: 87

root@742f4330fb39:/# curl www.google.com
<!DOCTYPE html>
<html>
  <body>
    <h1>Attacker's server!</h1>
  </body>
</html>
```

Figure 3: Name resolution after attack

Now we can verify if the attack succeeds by moving again into the **victim** container and running **dig www.google.com**. You should receive **10.0.0.20**, as shown in figure 3, which is the IP address of the attacker's server. This means that we successfully poisoned a record inside the cache of the recursive nameserver.

We can also inspect the traffic using Wireshark (remember to clear the captured packets before starting the attack). The second packet displayed should be a query from `10.0.0.40` (the recursive nameserver) to `10.0.0.50` (the authoritative nameserver) asking for **www.google.com**. Click on the packet, open the **Domain Name System** section at the bottom of the screen, right-click on the **Transaction ID**, and select **Apply as Filter -> Selected**. By doing so, we should obtain an output similar to figure 4, which contains three packets. The first one is the query, and the remaining two are responses. If the attack succeeds, we can clearly see that the spoofed response from the attacker arrives before the legitimate response from the authoritative nameserver.



```
dns.id == 0x3232

No.    Time         Source      Destination  Protocol Length Info
    4  4.012749685  10.0.0.40   10.0.0.50    DNS      74  Standard query 0x3232 A www.google.com
12855  4.320314884  10.0.0.50   10.0.0.40    DNS      104 Standard query response 0x3232 A www.google.com A 10.0.0.20
65543  7.018418798  10.0.0.50   10.0.0.40    DNS      90  Standard query response 0x3232 A www.google.com A 10.0.0.30
```

```
Frame 4: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface br-eb8debb5c7c6, id 0
Ethernet II, Src: 52:17:87:f8:b1:2b (52:17:87:f8:b1:2b), Dst: 86:4b:0f:3d:fb:77 (86:4b:0f:3d:fb:77)
Internet Protocol Version 4, Src: 10.0.0.40, Dst: 10.0.0.50
User Datagram Protocol, Src Port: 12345, Dst Port: 53
Domain Name System (query)
    Transaction ID: 0x3232
    Flags: 0x0100 Standard query
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 0
    Queries
    [Response In: 12855]
```

Figure 4: Wireshark's output after filtering

4.4 Mitigations

The main issue in DNS cache poisoning is that the attacker only needs to guess a 16-bit Query ID (QID), which means sending at most **65,536** spoofed responses. This is because although multiple fields must match in a valid DNS response, the other are predictable or can be easily inferred by the attacker. The simplest and most effective mitigation is to randomize the **UDP source port** for each query, which increases the entropy from 16 to almost 32 bits, making brute-force attacks almost infeasible. A stronger and more complete solution is to use **DNSSEC**, which provides cryptographic authentication of DNS data.

5 Kaminsky Attack

5.1 Idea

The idea of Kaminsky was to poison an NS record instead of an A record, by pretending to be an authoritative name server [4]. The impact of such an attack is much bigger than that of poisoning an A record since the attacker will be able to poison an entire domain. This attack works by impersonating an authoritative name server; in this way, all requests related to that specific domain will be forwarded to the attacker's name server until the TTL expires. To succeed, the attacker must win a race condition with the real nameserver, sending 2^{16} spoofed replies, each with a different transaction ID (TXID). The reason why the replies are 2^{16} is because we're assuming the recursive nameserver's port is fixed and known by the attacker.

5.2 Attack Flow

Fig. 5 shows a slight difference with respect to Fig. 1. In this case, the attacker will send forged NS records (step 3) that associate the target domain (**google.com**) with a malicious nameserver (**attacker.google.com**). As a result, the recursive nameserver will query the attacker's nameserver for **www.google.com** and will obtain the attacker's server IP as shown in step 5.

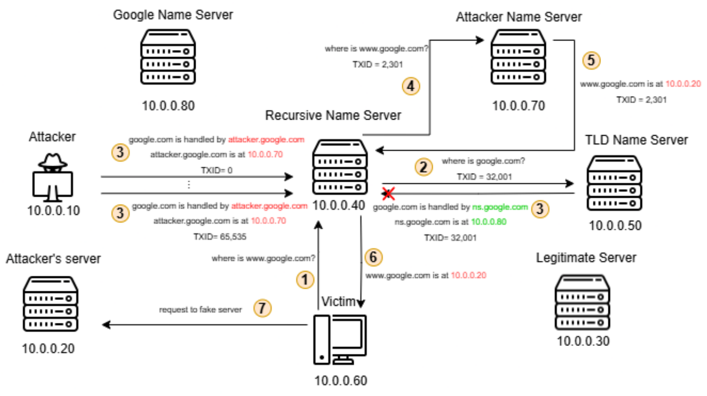
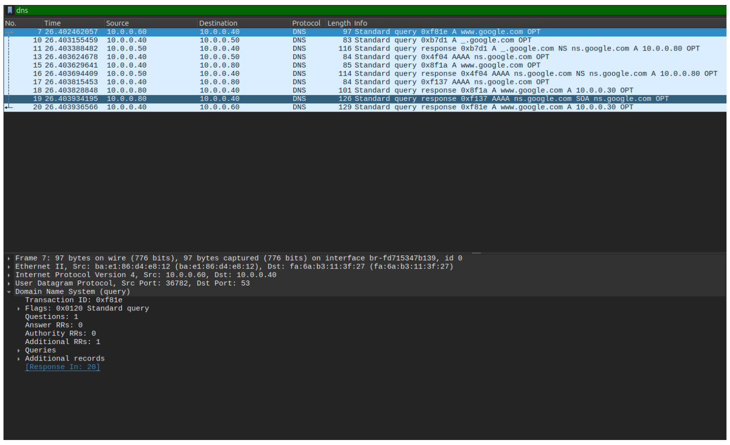


Figure 5: Kaminsky Attack

5.3 Laboratory

To begin this laboratory, move to the **kaminsky/** folder and run **docker system prune -f**, followed by **docker compose up**. Once the containers are running, open three terminals and execute **docker exec -it [attacker|victim|tld_ns] bash**.

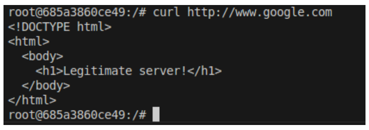
We will start by analyzing the regular packet flow between nameservers. In a new terminal, launch Wireshark with **sudo wireshark** (the password is **password**), then look for the network interface that begins with **br-...**, double-click on it, then apply the **dns** filter. From the victim's container send **dig www.google.com**. You should see DNS packets similar to those shown in Fig. 6. To check the webpage,



```
dns
No.   Time           Source      Destination  Protocol Length Info
    7 26.402462057  10.0.0.60   10.0.0.40    DNS       97 Standard query 0xf81e A www.google.com OPT
   10 26.403155459  10.0.0.40   10.0.0.50    DNS       83 Standard query 0xb7d1 A _.google.com OPT
   11 26.403388482  10.0.0.50   10.0.0.40    DNS      116 Standard query response 0xb7d1 A _.google.com NS ns.google.com A 10.0.0.80 OPT
   13 26.403624678  10.0.0.40   10.0.0.50    DNS       84 Standard query 0x4f04 AAAA ns.google.com OPT
   15 26.403629641  10.0.0.40   10.0.0.80    DNS       85 Standard query 0x8f1a A www.google.com OPT
   16 26.403694409  10.0.0.50   10.0.0.40    DNS      114 Standard query response 0x4f04 AAAA ns.google.com NS ns.google.com A 10.0.0.80 OPT
   17 26.403815453  10.0.0.40   10.0.0.80    DNS       84 Standard query 0xf137 AAAA ns.google.com OPT
   18 26.403828848  10.0.0.80   10.0.0.40    DNS      101 Standard query response 0x8f1a A www.google.com A 10.0.0.30 OPT
   19 26.403934195  10.0.0.80   10.0.0.40    DNS      126 Standard query response 0xf137 AAAA ns.google.com SOA ns.google.com OPT
   20 26.403936566  10.0.0.40   10.0.0.60    DNS      129 Standard query response 0xf81e A www.google.com A 10.0.0.30 OPT

› Frame 7: 97 bytes on wire (776 bits), 97 bytes captured (776 bits) on interface br-fd715347b139, id 0
› Ethernet II, Src: ba:e1:86:d4:e8:12 (ba:e1:86:d4:e8:12), Dst: fa:6a:b3:11:3f:27 (fa:6a:b3:11:3f:27)
› Internet Protocol Version 4, Src: 10.0.0.60, Dst: 10.0.0.40
› User Datagram Protocol, Src Port: 36782, Dst Port: 53
⌄ Domain Name System (query)
    Transaction ID: 0xf81e
  › Flags: 0x0120 Standard query
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 1
  › Queries
  › Additional records
    [Response In: 20]
```

Figure 6: dig to www.google.com

execute **curl HTTP://www.google.com** inside the victim's container. It should return the legitimate Google homepage, as shown in Fig. 7.

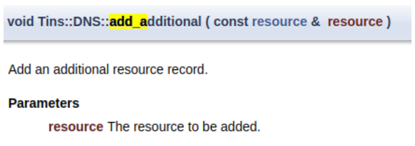


```
root@685a3860ce49:/# curl http://www.google.com
<!DOCTYPE html>
<html>
  <body>
    <h1>Legitimate server!</h1>
  </body>
</html>
root@685a3860ce49:/# 
```

Figure 7: Google's server

To perform the attack, edit the **kaminsky.cpp** file located in **/kaminsky/attacker** and complete all the TODOs. Once you've done that, compile the code within the attacker's container using **g++ kaminsky.cpp -o kaminsky -ltins**.

To help complete the TODOs, refer to the documentation shown in Figures 8, 9 and 10.

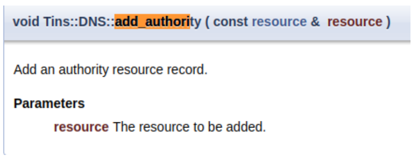


void Tins::DNS::add_additional (const resource & resource)

Add an additional resource record.

Parameters
resource The resource to be added.

Figure 8: add_additional

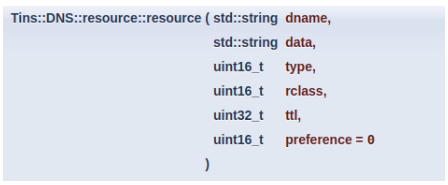


void Tins::DNS::add_authority (const resource & resource)

Add an authority resource record.

Parameters
resource The resource to be added.

Figure 9: add_authority

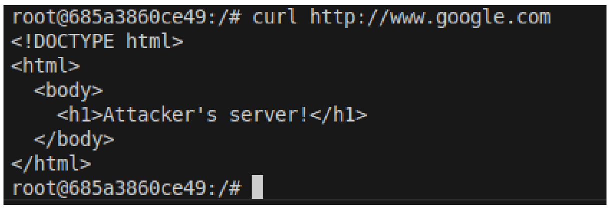


```
Tins::DNS::resource::resource ( std::string  dname,
                                std::string  data,
                                uint16_t     type,
                                uint16_t     rclass,
                                uint32_t     ttl,
                                uint16_t     preference = 0
                              )
```

Figure 10: resource

You can compare your solution with the solution provided at **/solutions/kaminsky.cpp**. Since the attack will take roughly 1s to send all the packets, we will add a delay to the TLD nameserver to increase the success rate of the attack. While being inside the **tld_ns** container, run **tc qdisc add dev eth0 root netem delay 2000ms**. Then, open Wireshark with the **dns** filter and run **./kaminsky** in the attacker's container.

While being inside the victim's container, check if the attack worked by running **dig www.google.com** (the IP should now be **10.0.0.20**) and curl **http://www.google.com**. If you used curl, you should see the result in Fig. 11.



```
root@685a3860ce49:/# curl http://www.google.com
<!DOCTYPE html>
<html>
  <body>
    <h1>Attacker's server!</h1>
  </body>
</html>
root@685a3860ce49:/# 
```

Figure 11: Attacker's server

5.4 Mitigations

The mitigations for this type of attack are the same as listed previously for DNS cache poisoning: source port randomization and DNSSEC.

6 DNSSEC

In this section, the theory of DNSSEC is explained, required to understand the laboratory. Then, two guided exercises are proposed in which DNSSEC is implemented in the networks explained in Fig. 1 and Fig. 5. Furthermore, it is shown how with the implementation of DNSSEC, the attacks proposed previously do not have any impact on the legitimate users.

6.1 What is DNSSEC

DNSSEC (DNS Security Extensions) is a functionality of DNS that applies cryptography to implement authentication and integrity during communication. This is required as DNS does not provide any security by itself.

As it is demonstrated in the previous sections, it is not very complicated to inject packets to mislead a legitimate user into connecting to a malicious server.

DNSSEC is based on two main concepts:

- **Signatures**, which are used to ensure that the records are coming from a specific source and have not been manipulated.
- **Chain of trust**, which ensures that the resolver is always communicating with a trusted source, this requires a *trust-anchor*, usually being the root ns.

By implementing these concepts, DNSSEC defends a system from DNS cache-poisoning, MitM, Kaminsky, and other related attacks.

6.2 Main components

To implement the concepts explained before, DNSSEC requires new types of records, other than the ones already part of DNS.

- **Resource Record Signature records (RRSIG):** Contain digital signatures for an RRset, used to verify the authenticity of DNS records.
- **DNSKEY Records:** Contain the public keys that resolvers use to validate the RRSIG signatures.
- **Delegation Signer records (DS):** Published by the parent zone; contain a hash of the child's DNSKEY to link parent and child zones securely.
- **Resource Record Sets (RRsets):** Groups of DNS records with the same name, class, and type, signed together as a single unit.
- **Next Secure records (NSEC/NSEC3):** Prove the non-existence of a domain name or record. NSEC3 adds protection against zone walking by hashing names.
- **Zone Signing Key (ZSK):** A key used to sign all the DNS records in a zone; its public part is published in a DNSKEY record.
- **Key Signing Key (KSK):** A key used specifically to sign the zone's DNSKEY records; its hash is published in the parent zone's DS record.

6.3 Validation process

DNSSEC implements authentication and integrity by adding a **validation process** after the name has been resolved [5]. As shown in Fig. 12, once a client sends a request to the resolver, the usual DNS name resolving process is executed, and after the resolver executes the checks over the keys and the signatures from child to parent.

This is how the validation process works in detail. If all the steps described have a positive result, the A record is sent back to the client, otherwise, it is discarded and a **SERVFAIL** packet is sent to the client:

- **isc.org:** The resolver firstly asks for the **A record**, and sets its **DNSSEC bit** to 1, requiring validation over the A record. isc.org sends back the A record together with the **signature**, then the resolver queries for the **cryptographic keys** to ensure that the signature corresponds. After the response, the resolver, to be sure that isc.org is a legitimate server, has to query the **parent NS**.
- **org:** The resolver queries for the **information of the child (DS)** contained in org, so that it can **authenticate** isc.org. After the response, the resolver has to validate the responses from the org NS, so it asks for its **signature and cryptographic keys**, and the org NS responds. Now the resolver has to **check the keys and signatures** of the org, so it queries its parent (root).
- **root:** The resolver asks root for the **information related to its child**, after the response, it **validates** the information of .org, now **both isc.org and .org are validated**. But what about root? The resolver asks the root for its **keys**, and to check for its authentication, the resolver

has already **hard-coded the keys of the legitimate root**, called *trust-anchor*; this way, by just checking if the response from the root and the *trust-anchor* contained in the trusted-keys match, the resolver knows that the **root is trusted**.

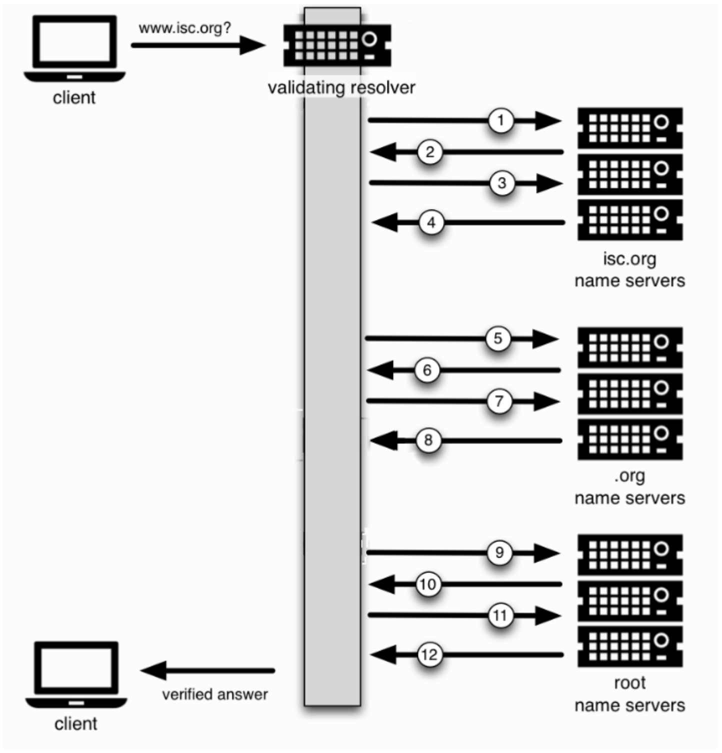


Figure 12: DNSSEC validaiton process

By following this validation process, it is shown how the **chain of trust** is built, by requiring authentication between parent and child with a secure link.

6.4 Laboratory

In this section, it is explained how to set up DNSSEC in the same network used for the Kaminsky attack. This laboratory is mainly related to the correct setup of configuration files inside the servers.

As shown in Fig. 13, the result of this exercise is that after the spoofed responses hit, the resolver will validate them, concluding with a **SERVFAIL, Broken trust chain** response to the client, which will not be affected by the attack.

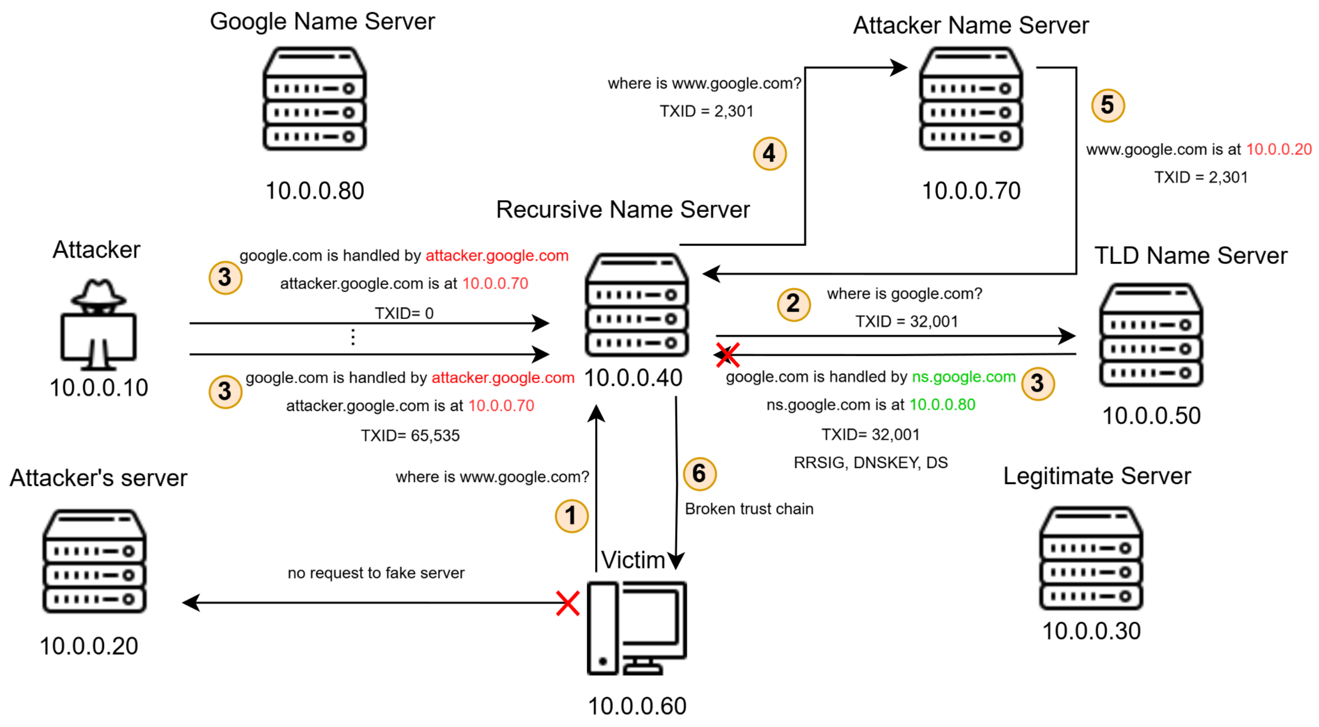


Figure 13: Kaminsky attack flow with DNSSEC

6.4.1 Setup

To set up the laboratory, it is required to:

- If you haven't done it yet, run **docker system prune -f**.
- Move to the **/DNNSSEC_kaminsky** folder.
- For this exercise, it will be required to implement DNSSEC over three different name servers **google_ns, tld_ns, and recursive_ns**.
- Two zones will have to be signed, so **8 keys**(4 public, 4 private) will have to be generated, and the **trust chain** will be implemented with the **DS record** of the Google NS.

6.4.2 Google NS

Firstly, let's start with the Google NS, which is the authoritative NS. The first step is to create cryptographic keys, which will be used to generate the new zone file, which contains the records of the zone together with the signatures. All the commands required to run can be found in the **/DNS-Cache-Poisoning-Kaminsky-Lab/README.md** file.

By executing the following commands with the **correct zone**, a private and a public key will be generated for both the ZSK and KSK.

- **dnssec-keygen -a NSEC3RSASHA1 -b 2048 -n ZONE TODO**
- **dnssec-keygen -f KSK -a NSEC3RSASHA1 -b 4096 -n ZONE TODO**

Solution, in this case, the correct zone is google.com, so the solution is:

- **dnssec-keygen -a NSEC3RSASHA1 -b 2048 -n ZONE google.com**
- **dnssec-keygen -f KSK -a NSEC3RSASHA1 -b 4096 -n ZONE google.com**

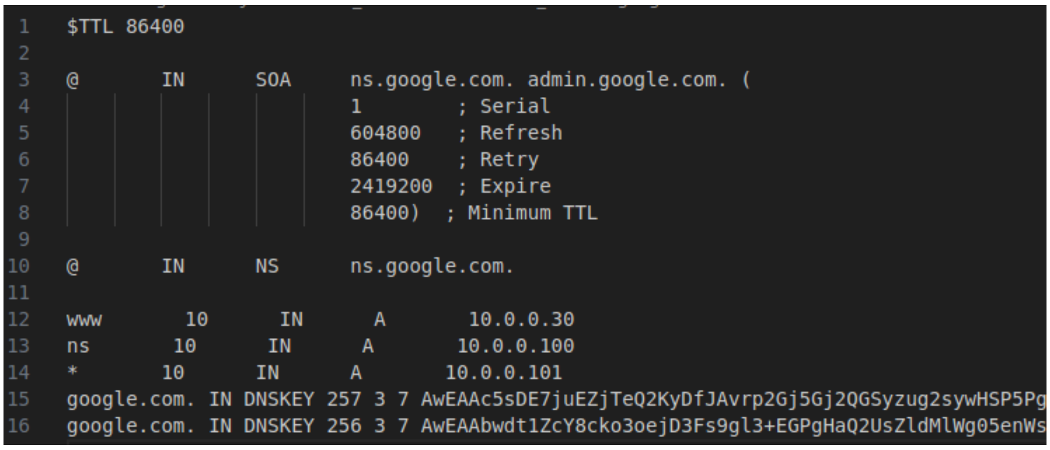
It is **mandatory** to use the correct zone, otherwise, it will not be possible to sign the zone file.

The correct keys will be generated; the algorithm used is specific to implement NSEC3 instead of NSEC, as the latter is exploitable for **zone walking attacks**.

The next step is to sign the zone file, so the public keys are **required** to be copied inside the zone file **google.com.zone**, then it is possible to run the command:

- **dnssec-signzone -3 $(head -c 1000 /dev/random | sha1sum | cut -b 1-16) -A -N INCREMENT -o google.com -t TODO**.

Solution, the public keys need to be copied as shown in Fig 14, then it is required to understand what is missing from this command.



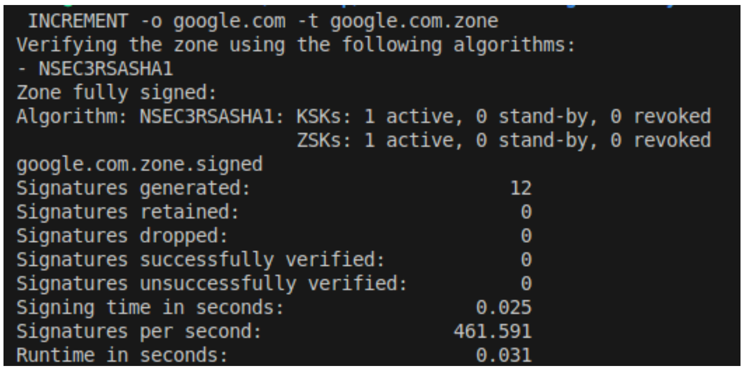
```
$TTL 86400

@       IN      SOA     ns.google.com. admin.google.com. (
                        1         ; Serial
                        604800    ; Refresh
                        86400     ; Retry
                        2419200   ; Expire
                        86400)   ; Minimum TTL

@       IN      NS      ns.google.com.

www       10      IN      A       10.0.0.30
ns       10      IN      A       10.0.0.100
*       10      IN      A       10.0.0.101
google.com. IN DNSKEY 257 3 7 AwEAAc5sDE7juEZjTeQ2KyDfJAvrp2Gj5Gj2QGSyzug2sywHSP5Pg
google.com. IN DNSKEY 256 3 7 AwEAAbwdt1ZcY8cko3oejD3Fs9gl3+EGPgHaQ2UsZldMlWg05enWs
```

Figure 14: How the zone file should look like before the sign

The zone file was missing:

- **dnssec-signzone -3 $(head -c 1000 /dev/random | sha1sum | cut -b 1-16) -A -N INCREMENT -o google.com -t google.com.zone**

If the command is correctly executed and the keys are correctly generated, the output of the command should be similar to the one shown in Fig 15.



```
 INCREMENT -o google.com -t google.com.zone
Verifying the zone using the following algorithms:
- NSEC3RSASHA1
Zone fully signed:
Algorithm: NSEC3RSASHA1: KSKs: 1 active, 0 stand-by, 0 revoked
                         ZSKs: 1 active, 0 stand-by, 0 revoked
google.com.zone.signed
Signatures generated:                       12
Signatures retained:                         0
Signatures dropped:                          0
Signatures successfully verified:            0
Signatures unsuccessfully verified:          0
Signing time in seconds:                 0.025
Signatures per second:                 461.591
Runtime in seconds:                      0.031
```

Figure 15: Correct output of the sign command

The sign command will generate two different files:

- **google.com.zone.signed**: Which contains the new records together with the new records.
- **dsset file**: Which contains the DS record of the Google NS that will be used later.

Now, the flags and options that DNSSEC requires to be implemented are already taken care of, these can be found in the **named.conf** and **named.conf.options** files of every NS.

This makes it so that the Google NS is already well implemented with DNSSEC. Now, it is required to implement DNSSEC in the parent zone (TLD NS) by creating a link to the Google NS as well.

6.4.3 TLD NS

To set up DNSSEC it is required:

- To move to the **/DNSSEC_kaminsky/tld_ns** directory.
- To generate another pair of public and private keys.

The command that is required to use is the same as before, but for the zone:

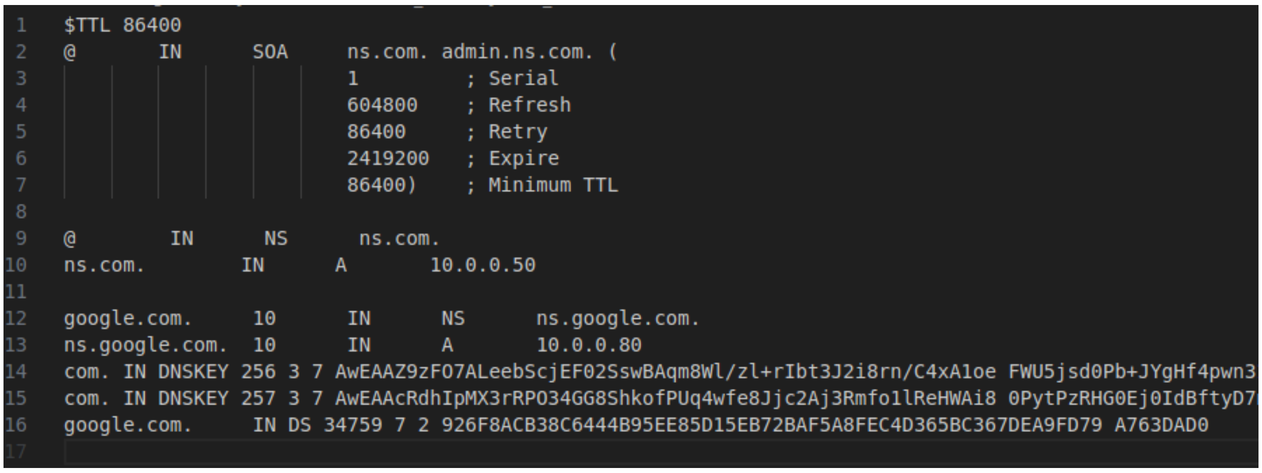
- **dnssec-keygen -a NSEC3RSASHA1 -b 2048 -n ZONE TODO**
- **dnssec-keygen -f KSK -a NSEC3RSASHA1 -b 4096 -n ZONE TODO**

Solution, in this case, the zone managed by the TLD is **com**, so the solution for these commands is:

- **dnssec-keygen -a NSEC3RSASHA1 -b 2048 -n ZONE com**
- **dnssec-keygen -f KSK -a NSEC3RSASHA1 -b 4096 -n ZONE com**

Now that we have generated the keys, it is required to sign the zone, solve the TODOs inside the **default.zone** file and then execute **dnssec-signzone -3 $(head -c 1000 /dev/random | sha1sum | cut -b 1-16) -A -N INCREMENT -o TODO -t TODO**.

Solution, before signing the zone file **default.zone** should look like the one showed in Fig. 16, it is required to copy the **public keys** of the TLD NS and the **DS record** of Google NS that was generated previously, by doing this the resolver will be able to authenticate the Google NS.



```
$TTL 86400
@       IN      SOA     ns.com. admin.ns.com. (
                        1          ; Serial
                        604800     ; Refresh
                        86400      ; Retry
                        2419200    ; Expire
                        86400)     ; Minimum TTL

@        IN      NS      ns.com.
ns.com.         IN      A       10.0.0.50

google.com.     10      IN      NS      ns.google.com.
ns.google.com.  10      IN      A       10.0.0.80
com. IN DNSKEY 256 3 7 AwEAAZ9zFO7ALeebScjEF02SswBAqm8Wl/zl+rIbt3J2i8rn/C4xA1oe FWU5jsd0Pb+JYgHf4pwn3
com. IN DNSKEY 257 3 7 AwEAAcRdhIpMX3rRPO34GG8ShkofPUq4wfe8Jjc2Aj3Rmfo1lReHWAi8 0PytPzRHG0Ej0IdBftyD7
google.com.     IN DS 34759 7 2 926F8ACB38C6444B95EE85D15EB72BAF5A8FEC4D365BC367DEA9FD79 A763DAD0
```

Figure 16: TLD zone file before sign

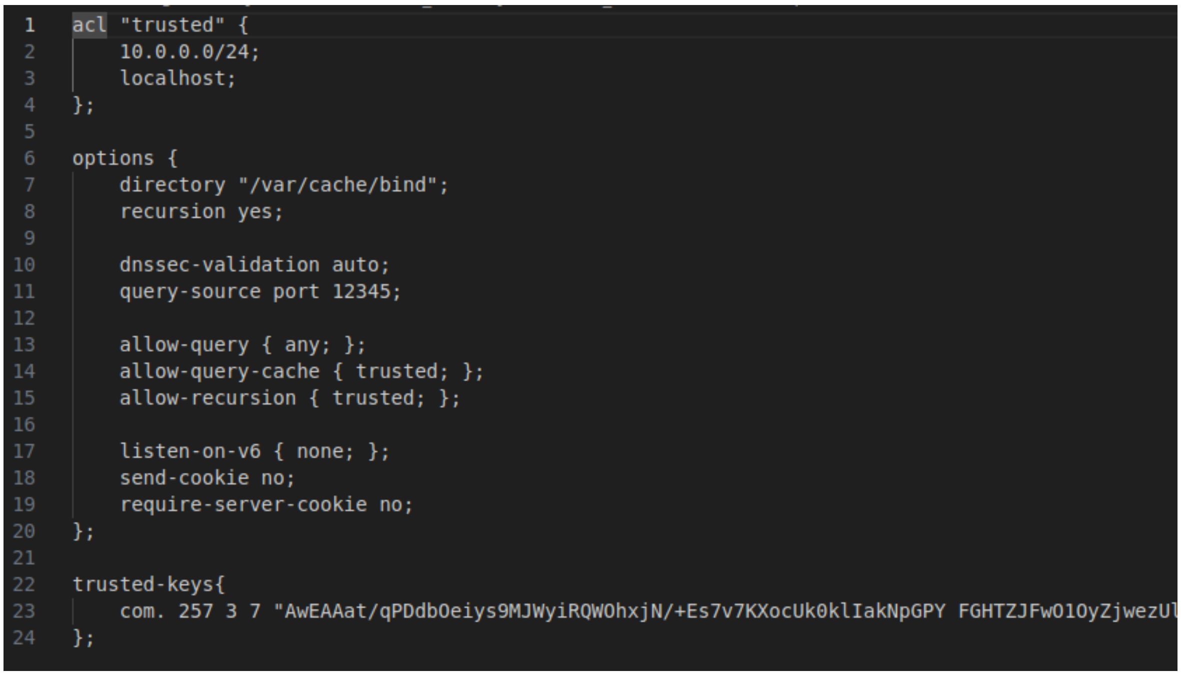
Then, it is possible to execute the command with the correct zone and zonefile: **dnssec-signzone -3 $(head -c 1000 /dev/random | sha1sum | cut -b 1-16) -A -N INCREMENT -o com -t default.zone**

The output of this command should be similar to that in Fig. 15, again the flags and options in the **named.conf** and **named.conf.options** are already taken care of, so DNSSEC is already well implemented in the TLD NS. The last step is to configure the resolver so that it makes the checks for the signatures and keys.

6.4.4 Recursive NS

The resolver configuration is different from the previous ones, as it only requires some flags and options to be implemented, such as the dnssec-validation flag, the *trust-anchor*, and others. To configure the resolver, it is required to complete the TODOs in the **/DNSSEC_kaminsky/recursive_ns/named.conf.options** file to implement the *trust-anchor*.

The solution, as shown in Fig. 17, is to insert the public KSK of the TLD NS as a trusted key; by doing this, the resolver will be able to authenticate the TLD as a trusted source.



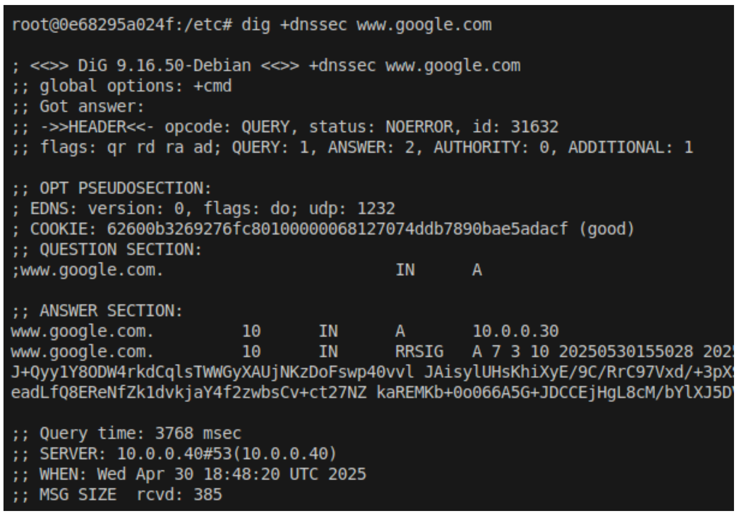
```
acl "trusted" {
    10.0.0.0/24;
    localhost;
};

options {
    directory "/var/cache/bind";
    recursion yes;

    dnssec-validation auto;
    query-source port 12345;

    allow-query { any; };
    allow-query-cache { trusted; };
    allow-recursion { trusted; };

    listen-on-v6 { none; };
    send-cookie no;
    require-server-cookie no;
};

trusted-keys{
    com. 257 3 7 "AwEAAat/qPDdbOeiys9MJWyiRQWOhxjN/+Es7v7KXocUk0klIakNpGPY FGHTZJFwO1OyZjwezUl
};
```

Figure 17: Configuration of the resolver

6.4.5 Testing

By following the previous sections, DNSSEC should be successfully implemented. It is now possible to test the implementation.

Move back to the **/DNSSEC_kaminsky** directory and run **docker compose up**. Then, inside the **victim** container run **dig +dnssec www.google.com**, the output of the command should be similar to the one shown in Fig. 18, the most important part of the output is the **ad** flag in the header, as it stands for **authenticated data**, meaning that the resolver has done all the checks that are explained in Section 6.3 and gave a positive response.

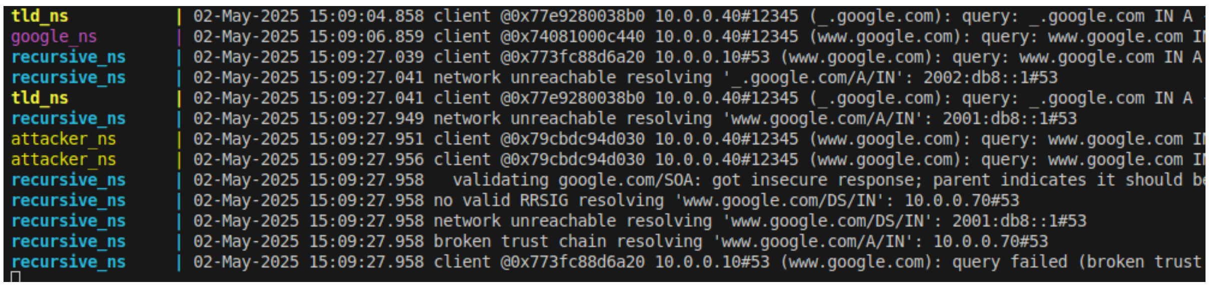


```
root@0e68295a024f:/etc# dig +dnssec www.google.com

; <<>> DiG 9.16.50-Debian <<>> +dnssec www.google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 31632
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 1232
; COOKIE: 62600b3269276fc80100000068127074ddb7890bae5adacf (good)
;; QUESTION SECTION:
;www.google.com.                        IN      A

;; ANSWER SECTION:
www.google.com.         10      IN      A       10.0.0.30
www.google.com.         10      IN      RRSIG   A 7 3 10 20250530155028 202
J+Qyy1Y8ODW4rkdCqlsTWWGyXAUjNKzDoFswp40vvl JAisylUHsKhiXyE/9C/RrC97Vxd/+3pX
eadLfQ8EReNfZk1dvkjaY4f2zwbsCv+ct27NZ kaREMKb+0o066A5G+JDCCEjHgL8cM/bYlXJ5D

;; Query time: 3768 msec
;; SERVER: 10.0.0.40#53(10.0.0.40)
;; WHEN: Wed Apr 30 18:48:20 UTC 2025
;; MSG SIZE  rcvd: 385
```

Figure 18: Correct output of dig command

To conclude the testing, it is possible to execute the Kaminsky attack seen previously to see if the result obtained is the same as in the introduction of Section 6.4.

Open two terminals, one for the **attacker** and one for the **tld_ns**, then in the TLD container run **tc qdisc add dev eth0 root netem delay 2000ms;** to set up the delay over the legitimate NS. Run in the attacker container **g++ kaminsky.cpp -o kaminsky -ltins** to compile the c++ file and then execute the attack with **./kaminsky**. After the attack concluded, check for the **docker compose logs** in the terminal in which docker compose up was run. If the configuration of DNSSEC is correct, the logs should be similar to the ones in Fig. 19. What is shown is exactly what was expected, as the broken trust chain log is related to the response given from the NS server of the attacker to the resolver. The resolver then sends back a **SERVFAIL** to the client that made the DNS request.



```
tld_ns             | 02-May-2025 15:09:04.858 client @0x77e9280038b0 10.0.0.40#12345 (_.google.com): query: _.google.com IN A
google_ns          | 02-May-2025 15:09:06.859 client @0x74081000c440 10.0.0.40#12345 (www.google.com): query: www.google.com I
recursive_ns       | 02-May-2025 15:09:27.039 client @0x773fc88d6a20 10.0.0.10#53 (www.google.com): query: www.google.com IN A
recursive_ns       | 02-May-2025 15:09:27.041 network unreachable resolving '_.google.com/A/IN': 2002:db8::1#53
tld_ns             | 02-May-2025 15:09:27.041 client @0x77e9280038b0 10.0.0.40#12345 (_.google.com): query: _.google.com IN A
recursive_ns       | 02-May-2025 15:09:27.949 network unreachable resolving 'www.google.com/A/IN': 2001:db8::1#53
attacker_ns        | 02-May-2025 15:09:27.951 client @0x79cbdc94d030 10.0.0.40#12345 (www.google.com): query: www.google.com I
attacker_ns        | 02-May-2025 15:09:27.956 client @0x79cbdc94d030 10.0.0.40#12345 (www.google.com): query: www.google.com I
recursive_ns       | 02-May-2025 15:09:27.958   validating google.com/SOA: got insecure response; parent indicates it should b
recursive_ns       | 02-May-2025 15:09:27.958 no valid RRSIG resolving 'www.google.com/DS/IN': 10.0.0.70#53
recursive_ns       | 02-May-2025 15:09:27.958 network unreachable resolving 'www.google.com/DS/IN': 2001:db8::1#53
recursive_ns       | 02-May-2025 15:09:27.958 broken trust chain resolving 'www.google.com/A/IN': 10.0.0.70#53
recursive_ns       | 02-May-2025 15:09:27.958 client @0x773fc88d6a20 10.0.0.10#53 (www.google.com): query failed (broken trust
```

Figure 19: Correct docker compose logs

References

[1] GeeksforGeeks. *Address Resolution in DNS (Domain Name Server)*. https://www.geeksforgeeks.org/address-resolution-in-dns-domain-name-server/.

[2] Steve Friedl. *An Illustrated Guide to the Kaminsky DNS Vulnerability*. http://unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html.

[3] ClouDNS. *BIND Explained: A Powerful Tool for DNS Management*. https://www.cloudns.net/blog/bind-explained-a-powerful-tool-for-dns-management/#What_is_BIND. 2024. (Visited on 04/27/2025).

[4] Steve Friedl. *An Illustrated Guide to the Kaminsky DNS Vulnerability*. http://unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html.

[5] Bind9. *DNSSEC Guide*. https://bind9.readthedocs.io/en/latest/dnssec-guide.html. 2023. (Visited on 04/29/2025).