UNIVERSITÀ DI TRENTO

NETWORK SECURITY

DNS Cache Poisoning
Kaminsky Attack

Beatrice Dall'Omo
Luca Righes
Lorenzo Cavada

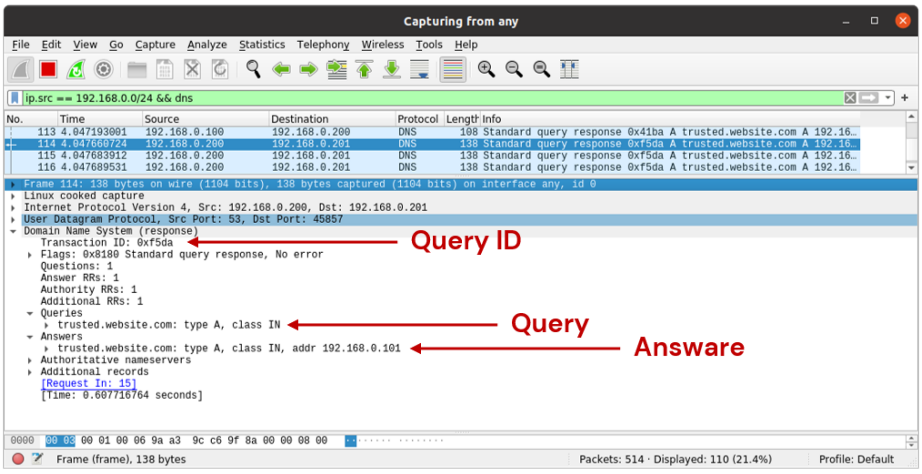
Contents

1. Introduction

In the 2008 the security researcher Dan Kaminsky presented at Black Hat (the information security conference), his discovery of the massively widespread and critical Domain Name System (DNS) vulnerability that allowed attackers to easily perform cache poisoning attacks on most name servers. This flaw sends users to malicious sites, hijack email and also made feasible a lot of attacks in most of Internet-based applications depending on DNS to locate their peers such as website impersonation, email interception and authentication bypass via the "Forgot My Password" feature on many popular websites. The exploit would allow attackers to impersonate any legitimate website and steal data. Indeed, the Kaminsky attack is the most dangerous type of DNS Cache poisoning since it amplifies the impact of the attack. In this report we are going to explain the difference between the two type of attack and try to replicate them using existing tools.

1.1 Packet structure

In the figure 1.1 it is possible to see how a DNS packet is actually made. In particular, it is shown the last message (*step 8*) sent by the *recursivedns* server to the *client* and it contains the actual answer to the query made by the client.


Capturing from any
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Query ID
Query
Answare

```
ip.src == 192.168.0.0/24 && dns

No.  Time         Source         Destination    Protocol Length Info
 113 4.047193001  192.168.0.100  192.168.0.200  DNS       108 Standard query response 0x41ba A trusted.website.com A 192.16…
 114 4.047660724  192.168.0.200  192.168.0.201  DNS       138 Standard query response 0xf5da A trusted.website.com A 192.16…
 115 4.047683912  192.168.0.200  192.168.0.201  DNS       138 Standard query response 0xf5da A trusted.website.com A 192.16…
 116 4.047689531  192.168.0.200  192.168.0.201  DNS       138 Standard query response 0xf5da A trusted.website.com A 192.16…

Frame 114: 138 bytes on wire (1104 bits), 138 bytes captured (1104 bits) on interface any, id 0
Linux cooked capture
Internet Protocol Version 4, Src: 192.168.0.200, Dst: 192.168.0.201
User Datagram Protocol, Src Port: 53, Dst Port: 45857
Domain Name System (response)
    Transaction ID: 0xf5da
    Flags: 0x8180 Standard query response, No error
    Questions: 1
    Answer RRs: 1
    Authority RRs: 1
    Additional RRs: 1
    Queries
        trusted.website.com: type A, class IN
    Answers
        trusted.website.com: type A, class IN, addr 192.168.0.101
    Authoritative nameservers
    Additional records
    [Request In: 15]
    [Time: 0.607716764 seconds]

0000  00 03 00 01 00 06 9a a3  9c c6 9f 8a 00 00 08 00   ........ ........

Frame (frame), 138 bytes                    Packets: 514 · Displayed: 110 (21.4%)      Profile: Default
```

Figure 1.1: Example of DNS message from step 8

The most interesting fields of this packet are the *Query ID* or *Transaction ID*, which will be crucial for the attacks, the *query field* where it can be seen what was the actual requested domain and the *answer field* where the association between the requested domain and the correlated IP can be found.

By default, the DNS protocol does not require any authentication, so in order to consider a response valid, it will take the first incoming messages with *Query ID* matching the one of the request and, in the latest version of BIND, the destination port equal to the source port of the request (in this laboratory this last security mechanism is not enforced due to the use of fixed source port). It is also important to distinguish the two main types of records that can come in a DNS response.
The first one is the *A type* (AAAA in case of IPv6) record contained in the message presented above. These records simply contain an association between a domain and an IP.
The other most common types of record that can come in a DNS response are the *NS type* records. These messages are sent between different DNS servers and have a management purpose allowing a query to scale down the hierarchy of DNS servers. Indeed, they specify that a DNS Zone, such as "website.com", is delegated to a specific Authoritative Name Server. Other than that, they also provide, in the *Additional Records* field, an A type (or AAAA in case of IPv6) record containing the actual IP of the Authoritative Name Server just presented.

1.2 DNS vulnerabilities

The main vulnerability that was discovered by Kaminsky was related to the sequential use of the Query Id for the DNS requests coming from the Recursive DNS and the predictability of the source port. In a DNS packet the length of these two fields is pointed out at 16 bits. This means that the attacker has 2^{16}=65.536 attempts to guess the right Query Id and this is a number small enough given enough opportunities. So, in previous DNS implementations, performing DNS cache poisoning attack was very easy. Indeed, in old name servers, the Query ID simply increments by one from the previous outgoing request and this means that it was straightforward for an attacker to guess what the next one will be (as long as the attacker can see a single query).

1.3 Kathará

Setting up a network infrastructure can be very expensive, even if it proves crucial for understanding computer networks. For this reason, to set up the network infrastructure, the Kathará framework has been employed. Kathará is a container-based framework that allows the users to deploy virtual networks featuring and traditional routing protocols. Therefore, it is a lightweight network emulation system based on Docker containers in which each virtual network device is emulated by a container and all these devices are interconnected by virtual L2 LANs (each container can potentially run a different Docker image). Kathará extremely simplifies the creation of complex networks using the concept of network scenario: a directory containing a file with the network topology, and, for each device, it maintains files and folders with the related configurations. It is an implementation of the notorious Netkit tool (Python based), hence it is cross-compatible, and inherits its language and features.
The main commands that the user should use in running the laboratory are:

- *kathara lstart*: search for a lab file and start the lab
- *kathara lclean*: stop the lab

- *kathara wipe*: delete all the kathará devices and links
- *kathara list*: show the instances

1.4 Topology of the laboratory

The entities that make up the environment for the laboratory are shown in the picture 1.2. All the machines are in the same subnet 192.168.0.0/24. On the left there are the malicious components of the infrastructure:

- *attacker*: a malicious server
- *bots*: three machines employed to perform the attacks by sending a lot of malicious DNS packets
- *dnsattacker*: a malicious name server that comes into play in the Kaminsky attack replacing the dnswebsite in order to bind trusted.website.com with 192.168.0.151 (malicious IP of the attacker)

The other entities that make up the network are the Recursive DNS (*dnsrecursive*) which receives all the requests that come from the client and, if it has not the necessary data in its cache, queries first the Root DNS (*dnsroot*) and then the Authoritative DNS (*dnsauthcom*) to solve the top-level domain and then the *dnswebsite* to solve the second-level domain.

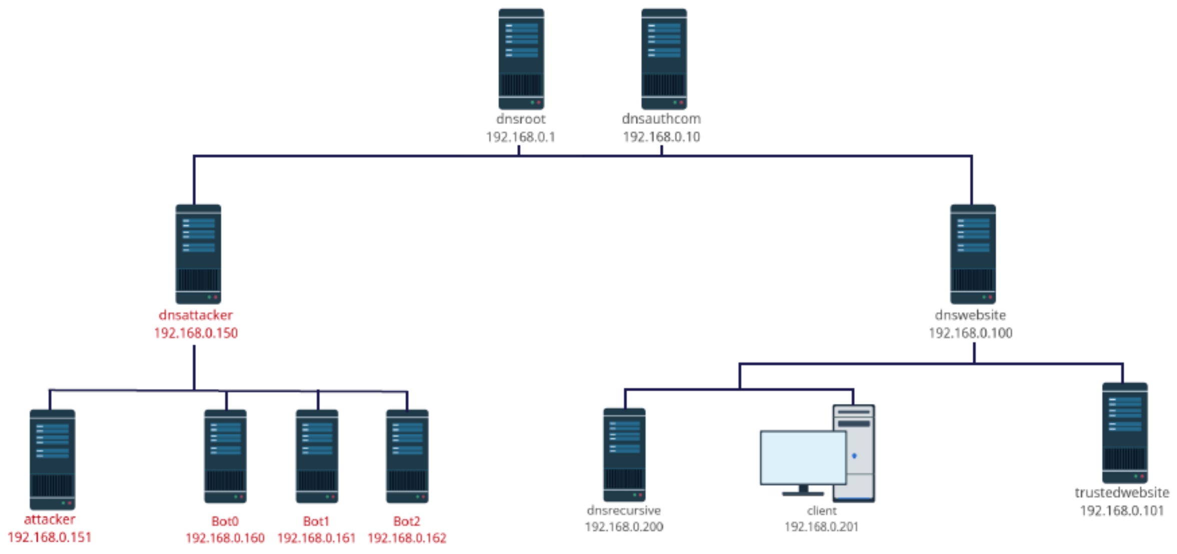


Figure 1.2: Topology of the laboratory

2. Configuration

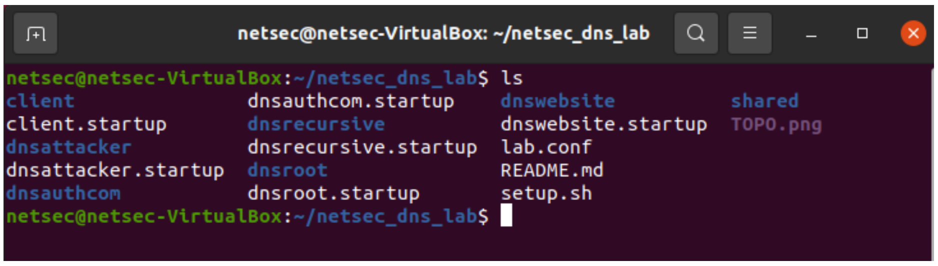
In this second chapter, it will be firstly shown the basis to start the virtual machine, then an overview of BIND and finally an explanation of the normal DNS flow. The first part aims to give the reader the knowledge to understand and adjust the laboratory as needed while, the second part, aims to give a practical introduction before the real exercises.

2.1 Starting the laboratory

After having downloaded the virtual machine from the shared folder, the user should import it into the Virtual Box (the repositories of the laboratory can be found on GitHub and on DockerHub). The machine should start automatically without requiring any installation and the credentials necessary to login are:

username: *netsec*
password: *netsec2021*

Once logged in, it is possible to open the laboratory folder (*netsec_dns_lab*) that can be easily found in the home directory. Inside the main folder it is possible to notice a series of connected folders and files (e.g. *client* folder and *client.startup* file). The *client* folder contains the configuration options of the docker image while the *client.startup* file contains the startup commands that will be run when the image is bootstrapped. The other relevant items are the *shared* folder (that allows to exchange files across the different containers) and the *lab.conf* (that is used to define the laboratory). At this point the reader should have the same view and be in the same location of what reported on the image 2.1.



```
netsec@netsec-VirtualBox:~/netsec_dns_lab$ ls
client              dnsauthcom.startup    dnswebsite           shared
client.startup      dnsrecursive          dnswebsite.startup   TOPO.png
dnsattacker         dnsrecursive.startup  lab.conf
dnsattacker.startup dnsroot               README.md
dnsauthcom          dnsroot.startup       setup.sh
netsec@netsec-VirtualBox:~/netsec_dns_lab$ 
```

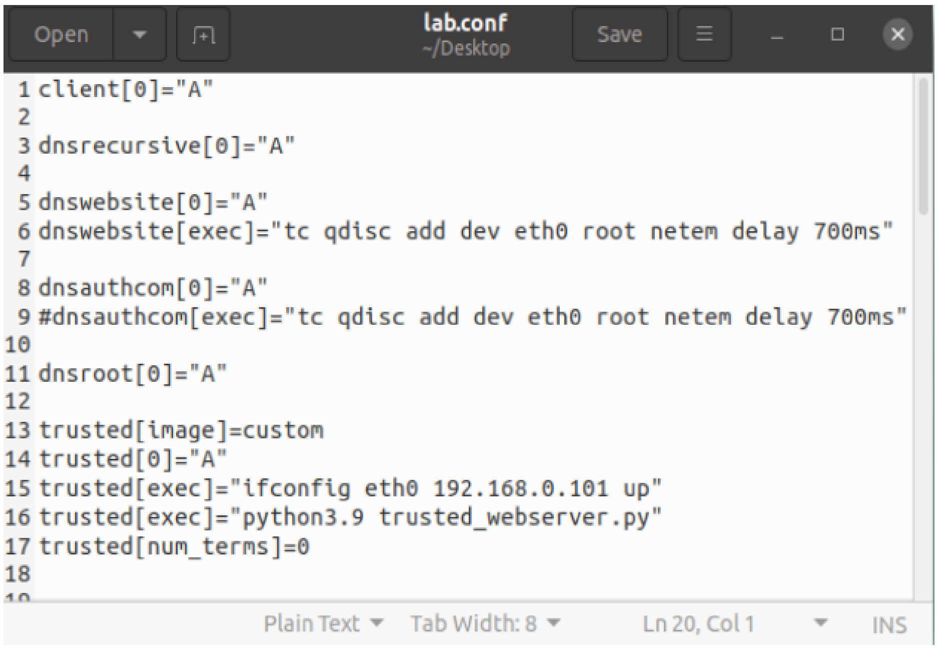
Figure 2.1: Main folder

2.2 lab.conf

The *lab.conf* is the main configuration file of the laboratory and it is essential to deepen it in order to understand and manage more easily the next steps. The reader can open the file by typing *gedit lab.conf* from the main folder (*netsec_dns_lab*). Inside, it is possible to see the name of the devices that will be started and the related configurations. The figure 2.2 reports partially the *lab.conf* file and each contained record follows the structure:

device [argument] = value

where *device* is the name of the device, *argument* can be either a number (e.g. 0 represents the network interface used to connect the device to the collision domain) or a keyword (e.g. *image* specifies that the Docker image for the device whenever it is different from the standard one or *exec* indicates a command that will be run) and the *value* states the configuration. For instance, at line 1, the command states that the client machine is going to use the network interface eth0 to connect to the collision domain A.



```
1 client[0]="A"
2
3 dnsrecursive[0]="A"
4
5 dnswebsite[0]="A"
6 dnswebsite[exec]="tc qdisc add dev eth0 root netem delay 700ms"
7
8 dnsauthcom[0]="A"
9 #dnsauthcom[exec]="tc qdisc add dev eth0 root netem delay 700ms"
10
11 dnsroot[0]="A"
12
13 trusted[image]=custom
14 trusted[0]="A"
15 trusted[exec]="ifconfig eth0 192.168.0.101 up"
16 trusted[exec]="python3.9 trusted_webserver.py"
17 trusted[num_terms]=0
18
```

Figure 2.2: lab.conf

2.3 Bind

A core part of the laboratory is the development of the DNS infrastructure. For this purpose, it has been employed BIND, the most popular DNS server in use today, open source and free to download. When the laboratory is going to be start, all the containers that require BIND automatically initiate it, according to the *device.startup* file. The configuration for each machine is reported in the *named.conf* file stored inside the path *device/etc/bind/*.

2.4 Normal message flow

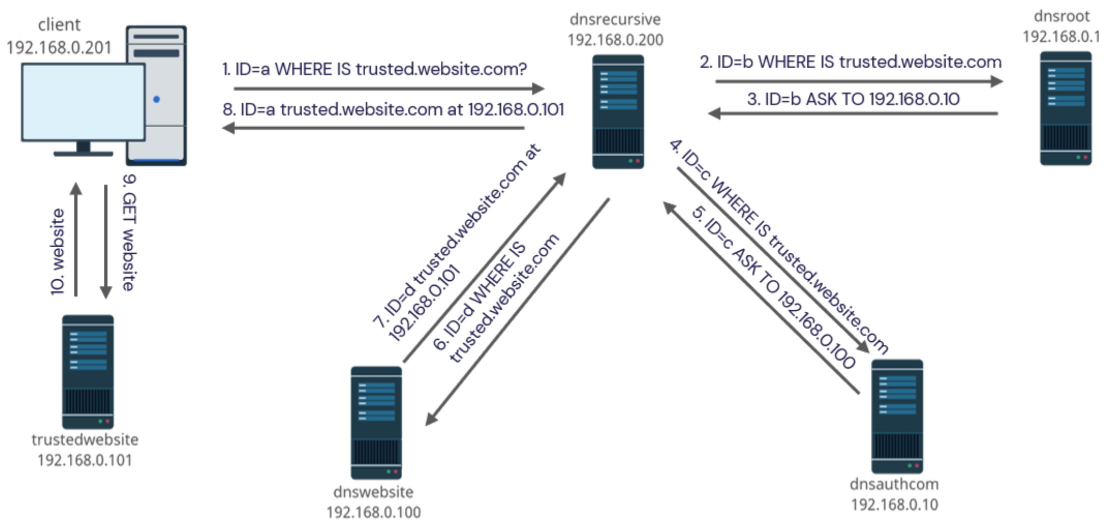

client
192.168.0.201
dnsrecursive
192.168.0.200
dnsroot
192.168.0.1
1. ID=a WHERE IS trusted.website.com?
8. ID=a trusted.website.com at 192.168.0.101
2. ID=b WHERE IS trusted.website.com
3. ID=b ASK TO 192.168.0.10
4. ID=c WHERE IS trusted.website.com
5. ID=c ASK TO 192.168.0.100
6. ID=d WHERE IS trusted.website.com
7. ID=d trusted.website.com at 192.168.0.101
9. GET website
10. website
trustedwebsite
192.168.0.101
dnswebsite
192.168.0.100
dnsauthcom
192.168.0.10

Figure 2.3: Normal flow of a DNS request

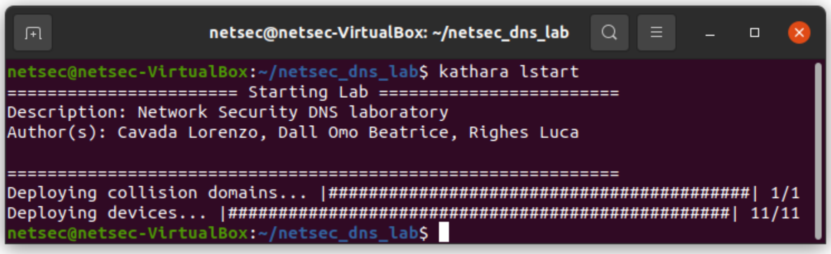
The figure 2.3 presents the normal flow followed by a typical DNS request (assuming the absence of cache). The following list explains the meaning of each packet exchanged during a normal DNS request:

0. The *client* wants to perform a request to *trusted.website.com* so it needs to know the actual associated IP
1. The *client* starts by asking to the *recursivedns* if it knows the IP associated to the requested domain, *trusted.website.com*
2. Assuming no cache, the *recursivedns* will start searching the answer for the query by asking to the root DNS server who is in charge for answering for the top level domain query (the *.com*). The IP of the *dnsroot* is hard-coded in the configuration of the *recursivedns*
3. The *rootdns* will check inside its database file and will redirect the *recursivedns* to the Authoritative Name Server in charge of handling the .com query, in this case the one with IP set to *192.168.0.10*. This answer will contain a NS type record followed by an A type record with the actual IP of the *authdns* server.
4. The *recursivedns* will now ask to the just received DNS server its query as it did in the step 2
5. The *authdns* server will not have a ready answer so it will response with an NS type record containing the DNS server in charge of the DNS zone *website.com*.
6. Again the *recursivedns* will perform the same request as before
7. Finally this DNS server will have inside its database file the association between the requested domain and the correlated IP. So it will be able to answer with an A type record containing the requested information.
8. The *recursivedns* will save the association inside its cache and will forward it to the client
9. The *client* will not be able to contact the *webserver*
10. The *webserver* will answer to the *client*

During all this process the *recursivedns* will store each association received by each DNS server in its cache decreasing the number of packets exchanged and increasing the speed of any other new incoming request. It is also important to notice how each couple of request - response has always a matching QID.

2.5 Replicate in the laboratory

The replication of what it has just been illustrated is pretty straightforward, so open the laboratory, open a terminal in the laboratory folder such as it is shown in figure 2.1. Here, type *kathara lstart* to start the laboratory. This will search inside the folder a *lab.conf* file and will start each machine using the settings specified in configuration file of each component. In case of an already running laboratory maybe it is needed to send a *kathara wipe* command to close any already existing instance.
After few seconds, it should be displayed a few command prompts and the output of the ubuntu shell should be the one in figure 2.4.

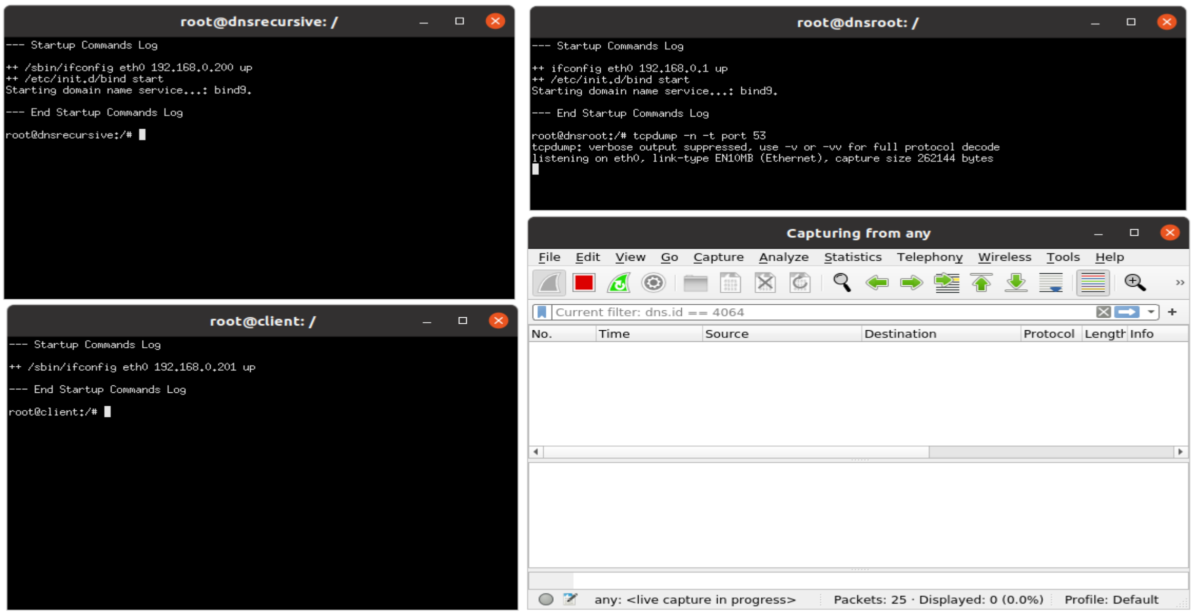


```
netsec@netsec-VirtualBox:~/netsec_dns_lab$ kathara lstart
======================= Starting Lab ========================
Description: Network Security DNS laboratory
Author(s): Cavada Lorenzo, Dall Omo Beatrice, Righes Luca

=============================================================
Deploying collision domains... |##########################################| 1/1
Deploying devices... |##################################################| 11/11
netsec@netsec-VirtualBox:~/netsec_dns_lab$
```

Figure 2.4: Expected output of: *kathara lstart*

For performing and analyzing a typical DNS request the only needed components are the *client*, the *recursivedns* and the *rootdns*. The other command prompts can be reduced to icon, remember not to close them otherwise the reader will have to restart the laboratory.
It is also needed to start *Wireshark* and start listening on the *any* interface. To do that, remember to use the root privilege, *sudo wireshark*, in order to be able to see all the exchanged traffic on the network.
Another useful component for sniffing the traffic on the network is the *tcpdump* command. In the *dnsroot* prompt type *tcpdump -n -t port 53* to start listening and intercepting all the traffic exchanged on the network that use the port 53. This will allow the reader to have an easier and quicker look on what it is happening on the network.

After this setup, the reader should have a situation like the one in figure 2.5.



```
root@dnsrecursive: /

--- Startup Commands Log

++ /sbin/ifconfig eth0 192.168.0.200 up
++ /etc/init.d/bind start
Starting domain name service...: bind9.

--- End Startup Commands Log

root@dnsrecursive:/# 
```

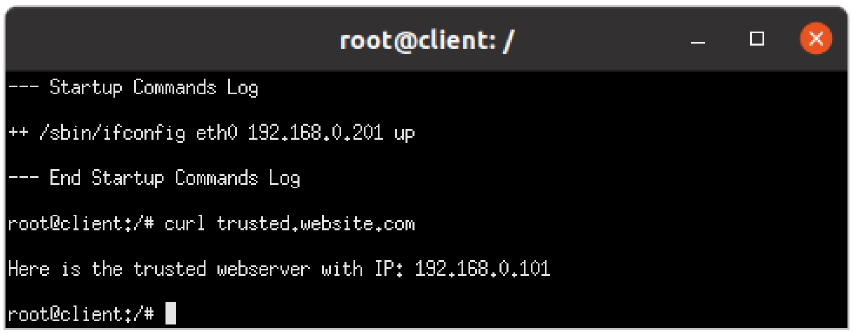
```
root@dnsroot: /

--- Startup Commands Log

++ ifconfig eth0 192.168.0.1 up
++ /etc/init.d/bind start
Starting domain name service...: bind9.

--- End Startup Commands Log

root@dnsroot:/# tcpdump -n -t port 53
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
```

```
root@client: /

--- Startup Commands Log

++ /sbin/ifconfig eth0 192.168.0.201 up

--- End Startup Commands Log

root@client:/# 
```

Capturing from any
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Current filter: dns.id == 4064
No. Time Source Destination Protocol Length Info
any: <live capture in progress>
Packets: 25 · Displayed: 0 (0.0%)
Profile: Default

Figure 2.5: Expected situation

It is now the time to generate some traffic so, in the *client* prompt, type *curl trusted.website.com*. This command will try to perform a simple get request to the specified domain. Of course the *client* at this moment does not know the related IP so, in the background, this request will trigger all the DNS request - response exchange seen before. After a few seconds should appear an answer on the *client* prompt with a simple string as shown in figure 2.6 which is the answer coming from the *webserver*.
It may happen that sometimes the request returns a server fail, in this case simply perform again the request until getting the expected answer.

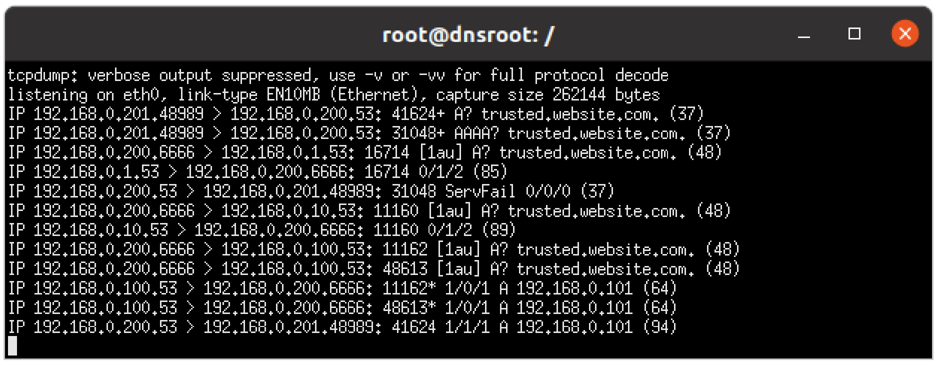


```
root@client: /

--- Startup Commands Log

++ /sbin/ifconfig eth0 192.168.0.201 up

--- End Startup Commands Log

root@client:/# curl trusted.website.com

Here is the trusted webserver with IP: 192.168.0.101

root@client:/# 
```

Figure 2.6: Expected answer

Now it is possible to see how all the traffic has been intercepted by the *tcpdump* in the *dnsroot* prompt as it can be seen in figure 2.7. The most interesting message is the last one where it can be seen that the *recursivedns* sent an answer to the *client* containing an A type record with the IP associated to the requested domain. It is also possible to see, just after the client IP:port, the actual QID of the request, in this case: 41624.

Using this QID, it is possible to intercept that specific package in *Wireshark* in order to see



```
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
IP 192.168.0.201.48989 > 192.168.0.200.53: 41624+ A? trusted.website.com. (37)
IP 192.168.0.201.48989 > 192.168.0.200.53: 31048+ AAAA? trusted.website.com. (37)
IP 192.168.0.200.6666 > 192.168.0.1.53: 16714 [1au] A? trusted.website.com. (48)
IP 192.168.0.1.53 > 192.168.0.200.6666: 16714 0/1/2 (85)
IP 192.168.0.200.53 > 192.168.0.201.48989: 31048 ServFail 0/0/0 (37)
IP 192.168.0.200.6666 > 192.168.0.10.53: 11160 [1au] A? trusted.website.com. (48)
IP 192.168.0.10.53 > 192.168.0.200.6666: 11160 0/1/2 (89)
IP 192.168.0.200.6666 > 192.168.0.100.53: 11162 [1au] A? trusted.website.com. (48)
IP 192.168.0.200.6666 > 192.168.0.100.53: 48613 [1au] A? trusted.website.com. (48)
IP 192.168.0.100.53 > 192.168.0.200.6666: 11162* 1/0/1 A 192.168.0.101 (64)
IP 192.168.0.100.53 > 192.168.0.200.6666: 48613* 1/0/1 A 192.168.0.101 (64)
IP 192.168.0.200.53 > 192.168.0.201.48989: 41624 1/1/1 A 192.168.0.101 (94)
```

Figure 2.7: Output of the *tcpdump* command

more information about its structure. So type in the *Wireshark* filter:

dns.id == 46124 && dns.flags.rcode == 0000

Wireshark should display 2 messages, the request coming from the *client* and the answer sent by the *recursivedns*. Opening the response the reader can see a package with the same structured as the one in figure 1.1.
The last thing that it is possible to check is the cache of the *recursivedns* which should now contain all the associations domain - IP that the server has been able to connect during the whole process. To see the content of the cache type in the *recursivedns* prompt the following two commands:

rndc dumpdb cache
cat varcachebindnamed_dump.db

The first command will generate a database file containing the dump of the cache then, using the cat command, it is possible to print in the screen the content of the just created file. What is interesting to check is the top of the output where it is possible to see all the associations made during the previous steps.

3. DNS Cache Poisoning

3.1 Overview

The first type of attack prepared for this laboratory is the DNS cache poisoning. As suggested by its name this attack basically consists in poisoning the cache of the recursive DNS server in order to add one or more malicious records that associate trusted domain to untrusted IP.
This attack is possible due to the lack of an authentication process in the DNS protocol. Indeed, without any additional settings, the exchange of requests and responses happens without any investigation on who is actually sending that messages.
From a very high level prospective this attack consists in simply replacing the DNS server which should response to a particular query sending to a target *dnsrecursive* server a fake response that appears to be valid but contains a malicious association between the requested domain and the corresponding IP.
This response will be cached by the *dnsrecursive* server effectively poisoning the cache. Each new request for the same domain will then be served based on the stored records making each client to be redirected to the malicious IP for all the duration of the *TTL* (*Time To Live*).
This process is well summarize in the figure 3.1 where it can be seen how the *bots* are trying to beat the *dnswebsite* by sending a malicious answer.

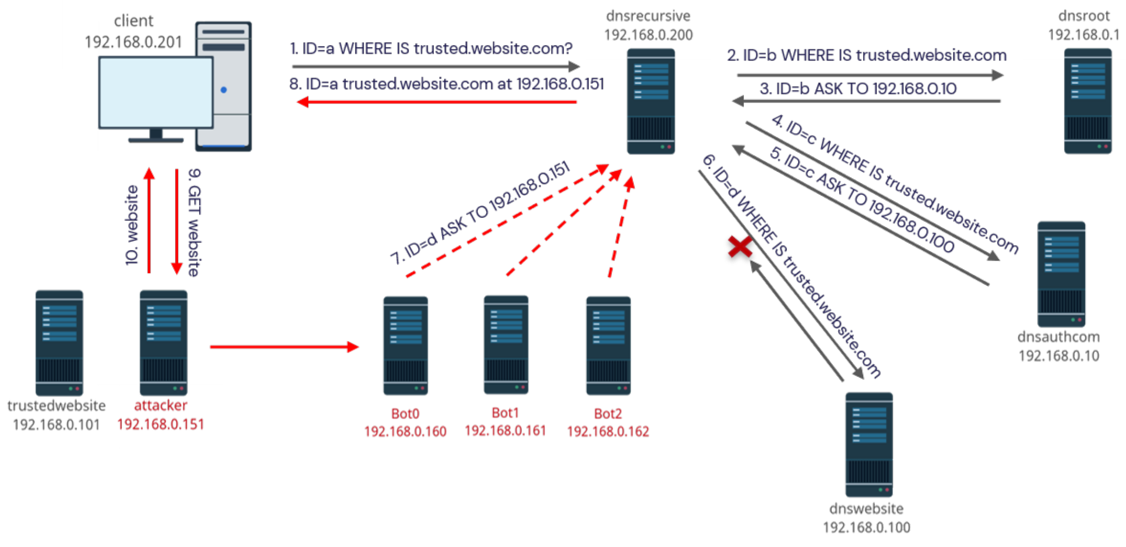

client
192.168.0.201
1. ID=a WHERE IS trusted.website.com?
8. ID=a trusted.website.com at 192.168.0.151
dnsrecursive
192.168.0.200
2. ID=b WHERE IS trusted.website.com
3. ID=b ASK TO 192.168.0.10
dnsroot
192.168.0.1
4. ID=c WHERE IS trusted.website.com
5. ID=c ASK TO 192.168.0.100
dnsauthcom
192.168.0.10
6. ID=d WHERE IS trusted.website.com
dnswebsite
192.168.0.100
7. ID=d ASK TO 192.168.0.151
10. website
9. GET website
trustedwebsite
192.168.0.101
attacker
192.168.0.151
Bot0
192.168.0.160
Bot1
192.168.0.161
Bot2
192.168.0.162

Figure 3.1: Flow followed by the messages in a cache poisoning attack

What is crucial in this attack is being able to forge a valid answer for the request made from the *dnsrecursive* server. What is especially challenging is being able to guess the Query ID of the request that, with the last implementation of BIND, is randomly chosen, causing, in the worst case scenario, the need of sending 65536 different packets (assuming the knowledge of the source port or the use of a fixed source port as in this laboratory).

3.2 The attack

To perform the attack the system relies on a python script running on the attacker machine. This script consists in a constant scan of the packages exchanged between the *dnsrecursive* and the *dnswebsite*. For each intercepted package it is checked if it is a DNS request and, in case of an affirmative result, the Query ID of the message is extrapolated and it is given to the *bots* that will start flooding the *dnsrecursive* with fake responses.
The *bots* are instead three separated machines that simple run a python script that listen on a specific port and wait for new request. Each request should also contain a parameter, the QID, that will be use for forging the fake responses. Each response consists in an A type record containing the malicious association between the requested domain and the IP of the attacker.
In the simulation there are 3 *bots*, this is done in order to give to the reader an idea on how the attack is performed in a real case scenario. Usually, the attacker does not know the actual QID of the request so it has to guess it. This process is especially tricky if the QID is randomly chosen bringing the attacker to send a huge number of fake responses. Having a short time to perform that, usually a distributed solution is used.

3.3 Reproducing the attack

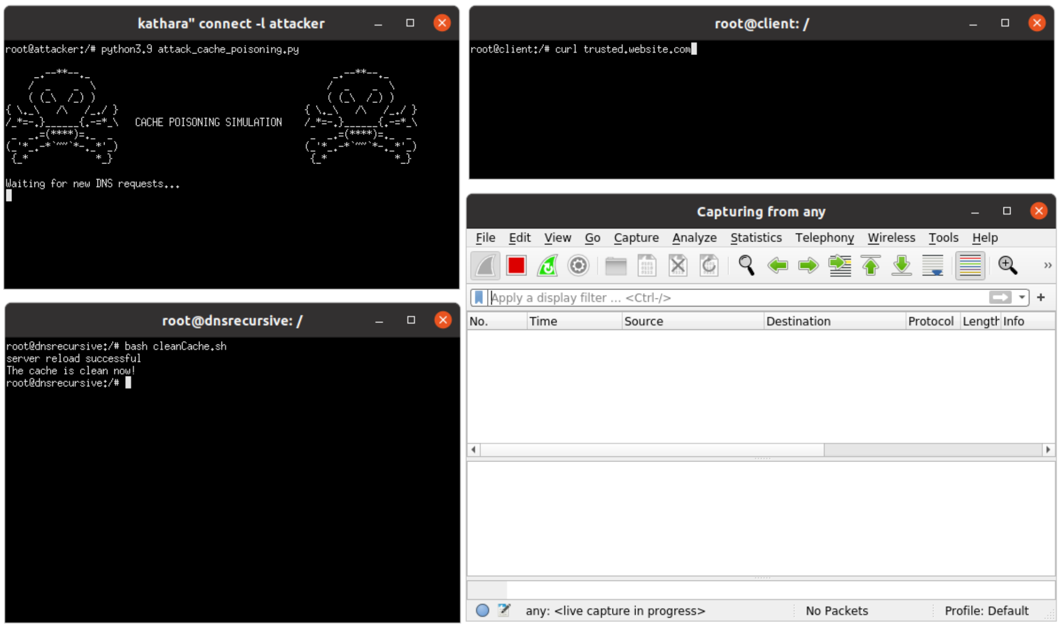
To reproduce the attack in the laboratory the needed components are the *attacker*, the *client*, the *dnsrecursive* and also *Wireshark* as in the previous chapter. Other than that, there is also the need to perform a little setup (the terminal in which the commands should be run is the one in the square brackets):

- Clean the cache of the *dnsrecursive* server. [*dnsrecursive*]: *bash cleanCache.sh*
- Restart the capture of *Wireshark* by pressing the green restart icon on the top left of the *Wireshark* window
- Start the actual attack: [*attacker*]: *python3.9 attack_cache_poisoning.py*
- Prepare the request of the client but wait before send it: [*client*] *curl trusted.website.com*

The situation should look like the one in figure 3.2.

If everything is ready just send the curl request from the *client* and wait until a response like the one in figure 3.3 is shown.

It may happen that sometimes the response of the request is a *server fail* or the trusted one (*192.168.0.100*). The reason behind it can be found in some latency inside the Kathará network

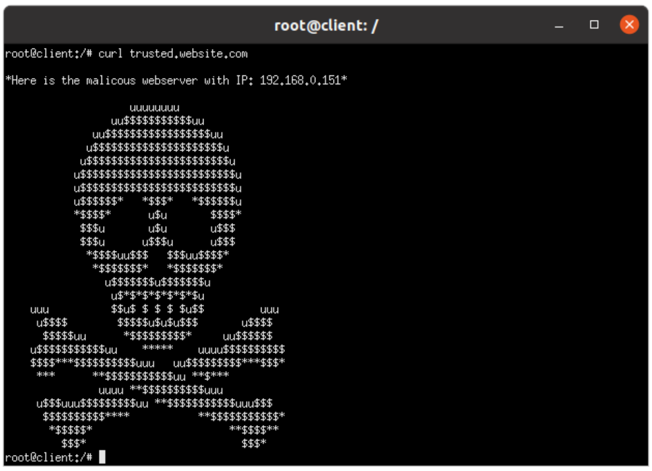

kathara" connect -l attacker
root@client: /
root@dnsrecursive: /
Capturing from any
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
No. Time Source Destination Protocol Length Info
any: <live capture in progress>
No Packets
Profile: Default

```
root@attacker:/# python3.9 attack_cache_poisoning.py

      _.--**--._                                                  _.--**--._
     /          \                                                /          \
    ( (_\   /_) )                                               ( (_\   /_) )
{ \._\   /\   /_./ }                                        { \._\   /\   /_./ }
/_*=-.}______{.-=*_\   CACHE POISONING SIMULATION    /_*=-.}______{.-=*_\
 _ _.=(****)=._ _                                          _ _.=(****)=._ _
(_'*_.-*`^^`*-._*'_)                                      (_'*_.-*`^^`*-._*'_)
 {_*              *_}                                        {_*              *_}

Waiting for new DNS requests...
```

```
root@client:/# curl trusted.website.com
```

```
root@dnsrecursive:/# bash cleanCache.sh
server reload successful
The cache is clean now!
root@dnsrecursive:/#
```

Figure 3.2: Summary of the setup



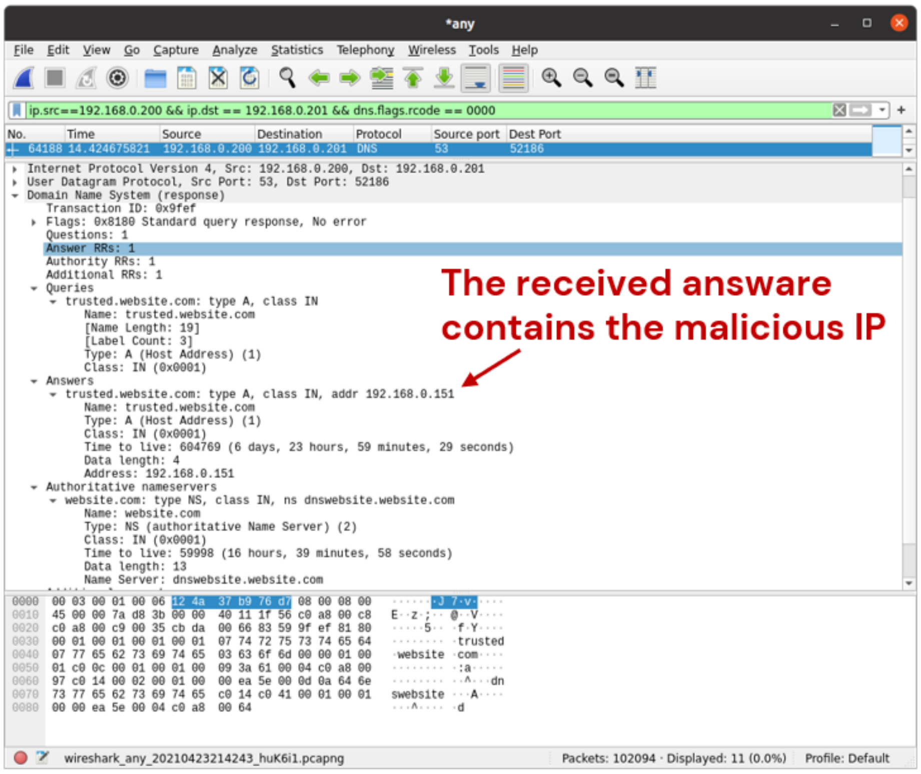
```
root@client:/# curl trusted.website.com

*Here is the malicous webserver with IP: 192.168.0.151*

                    uuuuuuuu
                 uu$$$$$$$$$$$uu
              uu$$$$$$$$$$$$$$$$$uu
             u$$$$$$$$$$$$$$$$$$$$$u
            u$$$$$$$$$$$$$$$$$$$$$$$u
           u$$$$$$$$$$$$$$$$$$$$$$$$$u
           u$$$$$$$$$$$$$$$$$$$$$$$$$u
           u$$$$$$*   *$$$*   *$$$$$$u
           *$$$$*      u$u       $$$$*
            $$$u       u$u       u$$$
            $$$u      u$$$u      u$$$
             *$$$$uu$$$   $$$uu$$$$*
              *$$$$$$$*   *$$$$$$$*
                u$$$$$$$u$$$$$$$u
                 u$*$*$*$*$*$*$u
    uuu          $$u$ $ $ $ $u$$         uuu
     u$$$$        $$$$$u$u$u$$$       u$$$$
      $$$$$uu      *$$$$$$$$$*     uu$$$$$$
    u$$$$$$$$$$$uu    *****    uuuu$$$$$$$$$$
    $$$$***$$$$$$$$$$uuu   uu$$$$$$$$$***$$$*
     ***      **$$$$$$$$$$$uu **$***
               uuuu **$$$$$$$$$$uuu
     u$$$uuu$$$$$$$$$uu **$$$$$$$$$$$uuu$$$
      $$$$$$$$$$****           **$$$$$$$$$$$*
       *$$$$$*                      **$$$$**
         $$$*                         $$$*
root@client:/#
```

Figure 3.3: Expected result of the attack

or if the *bots'* malicious responses cannot win the race again the *dnswebsite*. In this case simply perform again the first two steps of the setup.
If everything worked smoothly the message received from the *client* should be pretty different from the one saw in the previous chapter. This is a very hard situation to detect client side because, except from the case of a very evident difference between the expected answer and the received one like in this case, there is not any reason to doubt about the response.
Now it is time to move on and have a look to the intercepted packages, so open *Wireshark* and apply some filters to see just the packages in our interest. First have a look of the exchanged messages between the *client* and the *dnsrecursive*. We also want to see the messages without any error so the one with the all the four error flags all set to 0. To filter this messages simply type, in the filter section of *Wireshark*:

ip.src == 192.168.0.200 && ip.dst == 192.168.0.201 && dns.flags.rcode == 0000

In this way the only intercepted package will be the one containing the actual answer to the query of the client. Having a deeper look into the DNS field it is easy to see how the domain has been associated to the malicious IP, in this case the *192.168.0.151* as we can see in the figure 3.4.


*any
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
The received answare contains the malicious IP
Packets: 102094 · Displayed: 11 (0.0%)
Profile: Default

```
ip.src==192.168.0.200 && ip.dst == 192.168.0.201 && dns.flags.rcode == 0000

No.    Time            Source          Destination     Protocol  Source port  Dest Port
64188  14.424675821    192.168.0.200   192.168.0.201   DNS       53           52186

Internet Protocol Version 4, Src: 192.168.0.200, Dst: 192.168.0.201
User Datagram Protocol, Src Port: 53, Dst Port: 52186
Domain Name System (response)
   Transaction ID: 0x9fef
   Flags: 0x8180 Standard query response, No error
   Questions: 1
   Answer RRs: 1
   Authority RRs: 1
   Additional RRs: 1
   Queries
      trusted.website.com: type A, class IN
         Name: trusted.website.com
         [Name Length: 19]
         [Label Count: 3]
         Type: A (Host Address) (1)
         Class: IN (0x0001)
   Answers
      trusted.website.com: type A, class IN, addr 192.168.0.151
         Name: trusted.website.com
         Type: A (Host Address) (1)
         Class: IN (0x0001)
         Time to live: 604769 (6 days, 23 hours, 59 minutes, 29 seconds)
         Data length: 4
         Address: 192.168.0.151
   Authoritative nameservers
      website.com: type NS, class IN, ns dnswebsite.website.com
         Name: website.com
         Type: NS (authoritative Name Server) (2)
         Class: IN (0x0001)
         Time to live: 59998 (16 hours, 39 minutes, 58 seconds)
         Data length: 13
         Name Server: dnswebsite.website.com

0000  00 03 00 01 00 06 12 4a  37 b9 76 d7 08 00 08 00   ······J 7·v·····
0010  45 00 00 7a d8 3b 00 00  40 11 1f 56 c0 a8 00 c8   E··z·;·· @··V····
0020  c0 a8 00 c9 00 35 cb da  00 66 83 59 9f ef 81 80   ·····5·· ·f·Y····
0030  00 01 00 01 00 01 00 01  07 74 72 75 73 74 65 64   ········ ·trusted
0040  07 77 65 62 73 69 74 65  03 63 6f 6d 00 00 01 00   ·website ·com····
0050  01 c0 0c 00 01 00 01 00  09 3a 61 00 04 c0 a8 00   ········ ·:a·····
0060  97 c0 14 00 02 00 01 00  00 ea 5e 00 0d 0a 64 6e   ········ ··^···dn
0070  73 77 65 62 73 69 74 65  c0 14 c0 41 00 01 00 01   swebsite ···A····
0080  00 00 ea 5e 00 04 c0 a8  00 64                     ···^···· ·d

wireshark_any_20210423214243_huK6i1.pcapng
```

Figure 3.4: Expected result of the attack

It is also interesting to see the exchanged messages between the *dnsrecursive* and the *bots*. To filter this, it is useful to use the Query ID shown in the attacker prompt, in this case is: 7533. In case of multiple intercepted packets use the last QID shown. To identify the DNS packet associated to that specific QID type in the filter options of *Wireshark*:

dns.id == 7533 && dns.flags.rcode == 0000

As done before, it is possible to see both the request made from the *dnsrecursive* and the response coming from the *bots* (that seems to be legitimate but contains a malicious IP). It is also interesting to notice how the source IP of the packet matches the one of the *dnswebsite*. This is because the bot script spoofs also the IP of the *dnsrecursive*.

Now it is also possible to check the cache of the *dnsrecursive* server in the same way showed in the previous chapter. As it can be seen the situation appears to be normal but, looking at the association between *trusted.website.com* and the related IP, can be seen how it has been associated to *192.168.0.151*. Also the *TTL* (*Time To Live*) is much longer than a normal record. This is due to how the packet has been created by the *bots*, which specified a TTL of almost a week during which all the requests coming from the *client* will be automatically redirected to the malicious website giving to the attacker the possibilities to perform other type of attack such as phishing.

4. Kaminsky

4.1 Overview

The second type of attack recreated in this laboratory is the Kaminsky attack. The image 4.1 represents the flow of messages within our network and the more interesting point is labelled with the number 5. Indeed, the attacker, upon intercepting the query from the recursive DNS to the authoritative DNS, wakes up the bots and initiates the attack. The bots will generate a flood of possible responses trying to guessing the correct QID and win the race against the authoritative DNS. If the attack is successful, the recursive DNS will cache a wrong entry stating that the correct DNS is at a malicious IP. In the following query, the DNS recursive will think that it is going to contact the *dnswebsite* (192.168.0.100) while instead it is going to contact the *dnsattacker* (192.168.0.150). As it is easy to see, the situation is much worst if compared to the simple cache poisoning proposed in the previous chapter since, now, it is possible to poison an entire zone.

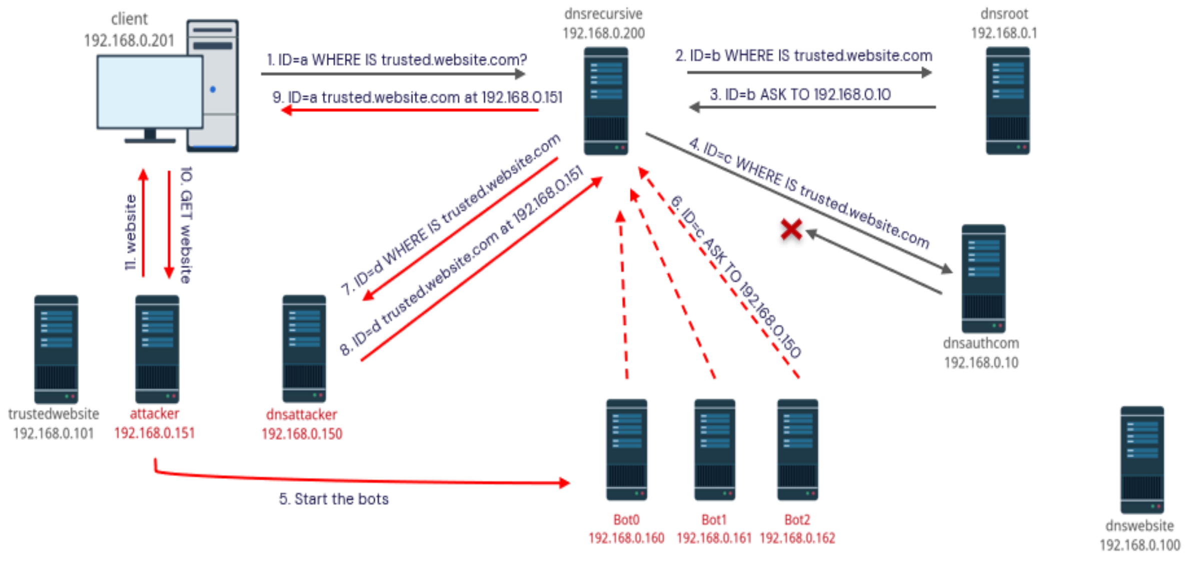
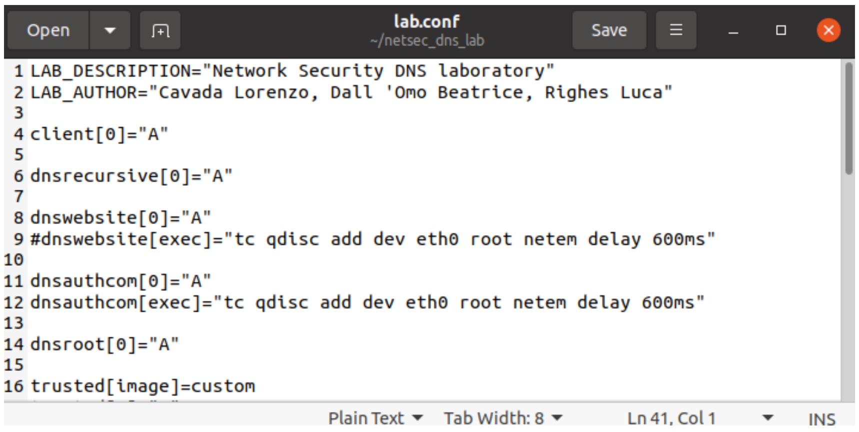

client
192.168.0.201
1. ID=a WHERE IS trusted.website.com?
9. ID=a trusted.website.com at 192.168.0.151
dnsrecursive
192.168.0.200
2. ID=b WHERE IS trusted.website.com
3. ID=b ASK TO 192.168.0.10
dnsroot
192.168.0.1
4. ID=c WHERE IS trusted.website.com
6. ID=c ASK TO 192.168.0.150
7. ID=d WHERE IS trusted.website.com
8. ID=d trusted.website.com at 192.168.0.151
10. GET website
11. website
dnsauthcom
192.168.0.10
trustedwebsite
192.168.0.101
attacker
192.168.0.151
dnsattacker
192.168.0.150
5. Start the bots
Bot0
192.168.0.160
Bot1
192.168.0.161
Bot2
192.168.0.162
dnswebsite
192.168.0.100

Figure 4.1: Kaminsky flow

4.2 The attack

The first step of the attack process is changing the configuration of the laboratory in order to remove the delay from the *dnswebsite* and add it to the *dnsattacker*. The operations that the reader should perform are listed below while the intended result after the changing is reported in figure 4.2.

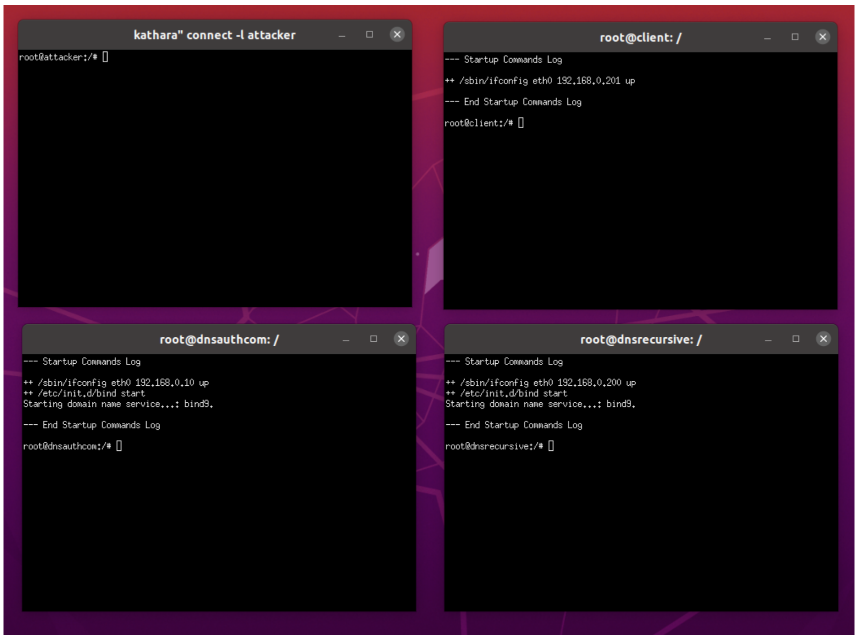
- Close the laboratory from the bash: *kathara lclean* (it may take some seconds)
- Open the *lab.conf* file: *gedit lab.conf*
- Comment the line 9 with an # before of the record
- Delete the # from line 12
- Save the file with *CTRL+s* and close it
- Restart the laboratory: *kathara lstart*



```
1 LAB_DESCRIPTION="Network Security DNS laboratory"
2 LAB_AUTHOR="Cavada Lorenzo, Dall 'Omo Beatrice, Righes Luca"
3
4 client[0]="A"
5
6 dnsrecursive[0]="A"
7
8 dnswebsite[0]="A"
9 #dnswebsite[exec]="tc qdisc add dev eth0 root netem delay 600ms"
10
11 dnsauthcom[0]="A"
12 dnsauthcom[exec]="tc qdisc add dev eth0 root netem delay 600ms"
13
14 dnsroot[0]="A"
15
16 trusted[image]=custom
```

Figure 4.2: lab.conf after changes

In case of encountering any errors during the restarting, the reader can run the *kathara wipe* command (to erase all the previous data) and then type again the *kathara lstart* command. Once that the laboratory is running again, the first step is reordering the terminals. The only ones necessary to perform the attacks are: *attacker*, *client*, *dnsrecursive* and *dnsauthcom* (as reported in the figure 4.3).
Since the configuration file of the laboratory has been changed, it is also important to verify that everything is still working correctly before performing the attack. So, from the *client* terminal the reader should launch the command *curl trusted.website.com* and receive back the same string as the one shown in figure 4.4 (if the terminal reports an error, just re-enter the same command).



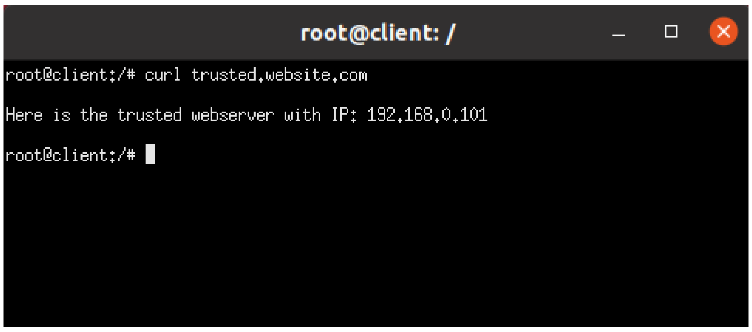
```
root@attacker:/# []
```

```
--- Startup Commands Log

++ /sbin/ifconfig eth0 192.168.0.201 up

--- End Startup Commands Log

root@client:/# []
```

```
--- Startup Commands Log

++ /sbin/ifconfig eth0 192.168.0.10 up
++ /etc/init.d/bind start
Starting domain name service...: bind9.

--- End Startup Commands Log

root@dnsauthcom:/# []
```

```
--- Startup Commands Log

++ /sbin/ifconfig eth0 192.168.0.200 up
++ /etc/init.d/bind start
Starting domain name service...: bind9.

--- End Startup Commands Log

root@dnsrecursive:/# []
```

Figure 4.3: Terminals



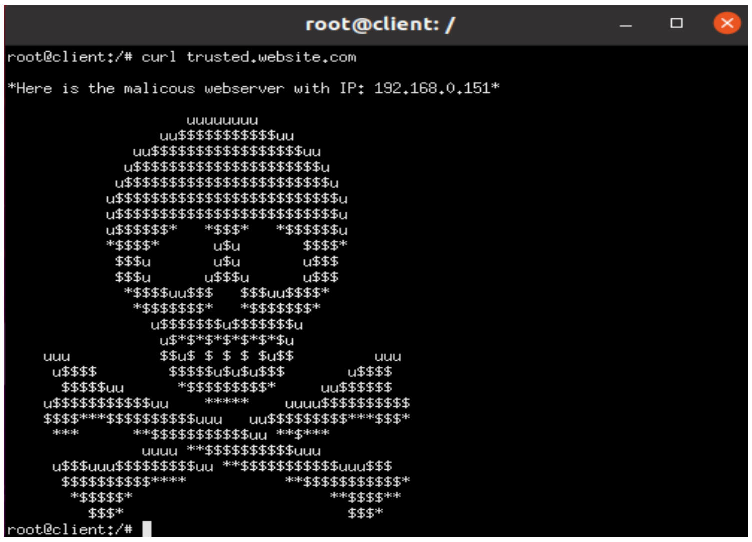
```
root@client:/# curl trusted.website.com

Here is the trusted webserver with IP: 192.168.0.101

root@client:/#
```

Figure 4.4: Normal response

Now, it is possible to replicate the attack following the commands listed below (the terminal in which the commands should be run is the one in the square brackets):

- Clean the cache of the recursive DNS - [*dnsrecursive*]: *bash cleanCache.sh*
- Run the attacker script - [*attacker*]: *python3.9 attack_kaminsky.py*
- Start *tcpdump* - [*dnsauthcom*]: *tcpdump -n -t port 53*
- Perform the query - [*client*]: *curl trusted.website.com*

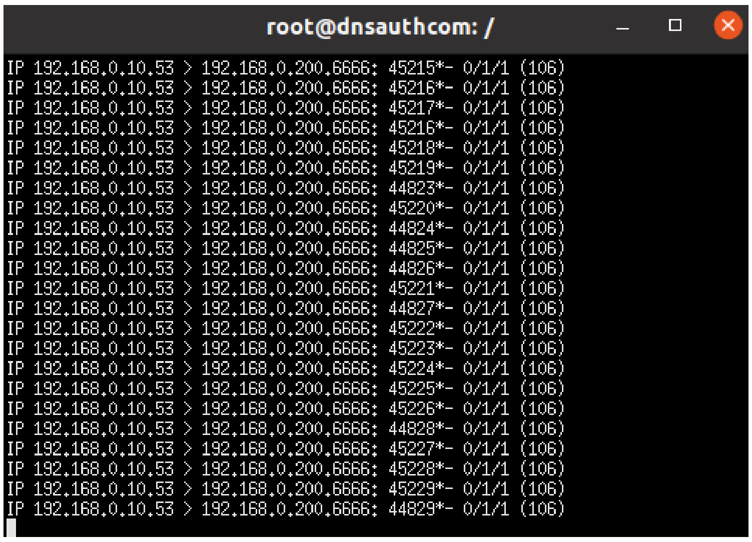
The expected result is the one reported in the figure 4.5 showing that the website retrieved is the malicious one. If the reader encounters any errors in this process can simply perform again the steps proposed above. This may happen due to delays in the Kathará network or if the bots do not win the race against the *dnsauthcom*.



```
root@client:/# curl trusted.website.com

*Here is the malicous webserver with IP: 192.168.0.151*

                    uuuuuuuu
                 uu$$$$$$$$$$$uu
              uu$$$$$$$$$$$$$$$$$uu
             u$$$$$$$$$$$$$$$$$$$$$u
            u$$$$$$$$$$$$$$$$$$$$$$$u
           u$$$$$$$$$$$$$$$$$$$$$$$$$u
           u$$$$$$$$$$$$$$$$$$$$$$$$$u
           u$$$$$$*   *$$$*   *$$$$$$u
           *$$$$*      u$u       $$$$*
            $$$u       u$u       u$$$
            $$$u      u$$$u      u$$$
             *$$$$uu$$$   $$$uu$$$$*
              *$$$$$$$*   *$$$$$$$*
                u$$$$$$$u$$$$$$$u
                 u$*$*$*$*$*$*$u
    uuu          $$u$ $ $ $ $u$$         uuu
     u$$$$        $$$$$u$u$u$$$       u$$$$
      $$$$$uu      *$$$$$$$$$*     uu$$$$$$
    u$$$$$$$$$$$uu    *****    uuuu$$$$$$$$$$
    $$$$***$$$$$$$$$$uuu   uu$$$$$$$$$***$$$*
     ***      **$$$$$$$$$$$uu **$***
               uuuu **$$$$$$$$$$uuu
     u$$$uuu$$$$$$$$$uu **$$$$$$$$$$$uuu$$$
      $$$$$$$$$$****           **$$$$$$$$$$$*
       *$$$$$*                      **$$$$**
         $$$*                         $$$*
root@client:/#
```

Figure 4.5: Malicious response

This time Wireshark will not be used but instead an overview of the flood of messages generated by the bots is proposed. The reader should already noticed the series of messages appeared in the *dnsauthcom* terminal (similar to what reported in figure 4.6). The interesting part here is understanding that what the bots do is basically forging a series of fake responses. Such responses spoof the IP of the authoritative DNS (192.168.0.10) and try to guess the possible QID. Of course, QIDs reported in the column are not incremental because there are different bots and different threads used in each bot at the same time.



```
IP 192.168.0.10.53 > 192.168.0.200.6666: 45215*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 45216*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 45217*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 45216*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 45218*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 45219*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 44823*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 45220*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 44824*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 44825*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 44826*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 45221*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 44827*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 45222*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 45223*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 45224*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 45225*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 45226*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 44828*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 45227*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 45228*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 45229*- 0/1/1 (106)
IP 192.168.0.10.53 > 192.168.0.200.6666: 44829*- 0/1/1 (106)
```

Figure 4.6: Fake responses

5. Mitigations

Nowadays these attacks are no more possible since specific patches have been applied to eliminate these vulnerabilities and to make harder for an attacker to guess the right Query Id.

5.1 Upgrade Bind9

A possible mitigation concerns the Bind Upgrade up to version 9. It is quite unnecessary to call this a mitigation since older versions are no more easily available, indeed there is no support for any of previous versions. It is important to know that the vulnerabilities discovered in the versions below 4 of Bind were the sequential Query ID of the request coming from the Recursive DNS and the predictability of the source port. The solutions introduced was to randomize both the Query ID and the source port in order to make hard for an attacker to guess the right Query ID. In this way the entropy is increased from 16 to 32 bits and now the attacker has $2^{32} = 4.294.967.296$ chances to guess the ID and any answer that does not match both source port and Query ID will be dropped. Therefore, now the attacker needs more time and more computational power in order to have a successful attack even if in reality not all the the 16 bits can be used for the source port because of the reserved values.

5.2 DNSSEC

Another possible mitigation is the use of DNSSEC, a suite of IETF specifications for securing the DNS information. It has been proposed as a way to bring cryptographic assurance to results provided by DNS entities. In fact, it is a secure implementation of the DNS protocol that implements DNS authentication on top of normal DNS exchange. It uses a public key cryptography to digitally sign all the responses that come from Root DNS, from the Authoritative DNS and from the dnswebsite. In our network it means that the Recursive DNS will only accept data coming from these specific entities and not from other sources. Moreover, for each DNS request and response, DNSSEC provides also integrity checks and by providing these two additional features it allows the cache of the recursive DNS not to be poisoned and only the legitimated DNS responses are accepted and processed by the Recursive DNS. Despite this the adoption of DNSSEC protocol has been slowed down due to several issues that include the need to:

- adapt the DNSSEC protocol to the size of the entire Internet, without breaking the backwards compatibility;
- develop the DNSSEC infrastructure for the variety of DNS servers and clients;
- reach an agreement on who should have control of the top-level domains keys.