# DNS cache poisoning 101
# Homework report #3
# Ethical Hacking

Andrea Lacava 1663286, Ilaria Clemente 1836039,
Matteo Attenni 1655314

May 27, 2019

**Abstract**

In this document we describe the Kaminsky DNS cache poisoning attack. Starting from a brief introduction on what is the DNS protocol we describe how you can implement a "cache poisoning" attack in a few simple steps. Then we provide an analysis of the implementation scenario of our work with all the audits carried out and the description of the attack code attached to this documentation that we used to demonstrate the danger of this vulnerability. Finally we show the victory flag captured during the attack.

## Contents

## 1 Introduction

DNS (Domain Name System) is a hierarchical name system used to identify network hosts. The main function of a DNS server is to execute the URL-translation which is the process of converting URLs (Uniform Resource Locator) into numeric IP addresses, such as 8.8.8.8, allowing normal Internet users to remember a simple name to get the address of a service. An example of DNS packet can be seen in figure 1.

In February 2008, security-related researcher Dan Kaminsky discovered a vulnerability that could be used to alter the normal function of the DNS. The attack aims to target DNS's URL-translation function so that an infected DNS server will respond with a wrong IP address for an URL. In particular, a corrupted DNS server will reply with the IP address of a malicious website when asked to resolve URLs within a non malicious server like google.com. This would redirect unsuspected users of DNS to the malicious website that may contain fake login forms or any conceivable threat.

The mechanism of DNS is that when it first receives a request from a user it first checks if it can give an authoritatively answer based on a local database that contains all the pairings of domain names with IP addresses. If the query matches a corresponding record in its local database the system will give an authoritative answer otherwise the server will look for information inside the cache, and if a positive answer is given the query is completed. If even after this process there is no match found the query process can continue employing recursion to get an answer.

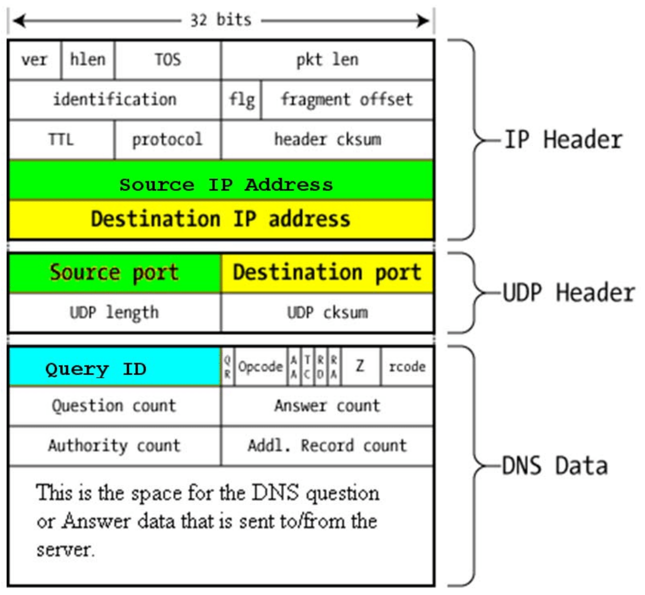


Figure 1: DNS generic packet

## 2 Scenario and IPs - Question 1

Before starting the attack, We set up the virtual machine by changing the *config.json* file with our parameters. The secret we've used is "*IAmRat2019!*" and We also define a route in vulnDNS host versus our host, a Linux based OS:

```
#!/bin/bash
sudo route add default gw 192.168.56.1
```

Then We begin to inspect the virtual machine network using this *nmap* command:

```
#!/bin/bash
ip a # just to know the subnet of the vm
sudo nmap -sn -PE 192.168.56.1/24
```

Though these commands we find that there are three active hosts in this subnet:

1. 192.168.56.1, our IP.
2. 192.168.56.100, with whom we had no involvement during the attack.
3. 192.168.56.101, that with the next commands it would turn out to be the vulnDNS IP.

Then We decide to have a look up by querying these unknown hosts and we obtain that only 192.168.56.101 is reachable using the *dig* command, which gives you the ability to run DNS queries:

```
#!/bin/bash
dig NS bankofallan.co.uk @192.168.56.x # prototype of the command
```

By the results obtained We found out that 192.168.56.100 gives no response and 192.168.56.101 instead reply to us by giving the IP address of *bankofallan.co.uk*: 10.0.0.1. Once acquired the target IP and once spoofed the IP of Bank of Allan, which will later be useful for creating false DNS responses, our inspection continues by analyzing the other hosts.

On the other host (.100) deep inspections hasn't revealed any information apart from the fact that it does not affect in any way during the phase of sniffing, communication and attack. Our assumption then is that the host is just a client but we have not further investigated because it is not directly related to our goal.

We also found out on one of our PCs the presence of a fourth active host, precisely with IP address 192.168.56.102 which, in the same way as 192.168.56.100, is extraneous and useless for the purpose of the attack.

The description of the attack is represented in figure 2, in which there are the essential steps of the interactions for the whole operation to be successful. It is possible to identify two key areas that represent the attacker and the target, i.e. the server that is going to be attacked through cache poisoning. Continuing to refer to the figure, the highlights of the attack are:

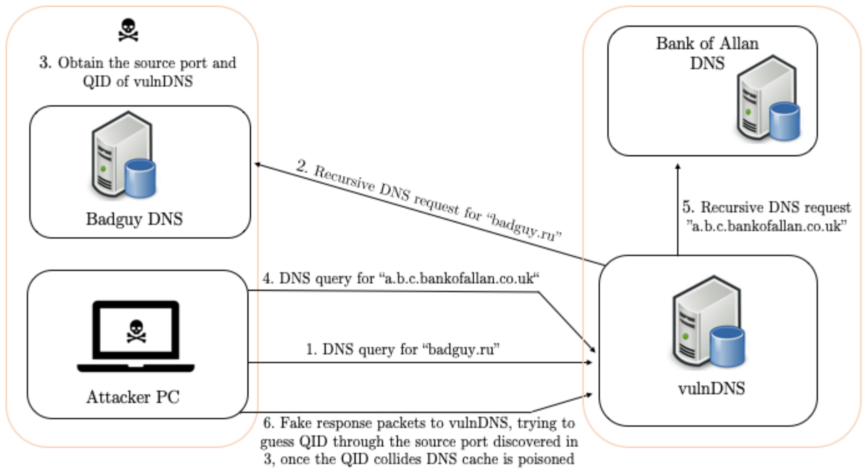

3. Obtain the source port and QID of vulnDNS
Badguy DNS
Attacker PC
Bank of Allan DNS
vulnDNS
2. Recursive DNS request for "badguy.ru"
5. Recursive DNS request "a.b.c.bankofallan.co.uk"
4. DNS query for "a.b.c.bankofallan.co.uk"
1. DNS query for "badguy.ru"
6. Fake response packets to vulnDNS, trying to guess QID through the source port discovered in 3, once the QID collides DNS cache is poisoned

Figure 2: Diagram representing the attack in pills.

1. DNS recursive request to vulnDNS about what IP matches with *badguy.ru*.
2. VulnDNS will forward the request to badguyDNS, which is an authority for it and is owned by us.
3. Analizing the query packet badguyDNS obtains vulnDNS' actual query id and the destination port it uses for DNS answers.
4. DNS recursive request to vulnDNS about what IP matches with *random.bankofallan.co.uk*[1].
5. VulnDNS will forward the request to Bank of Allan's DNS because of our imposition to make a recursive request.
6. We flood the network with fake response packets to vulnDNS source port discovered in the previous steps, pretending to be the authoritative server of *bankofallan.co.uk* with IP address 10.0.0.1 generating random and incremental query ids until it matches the one used by vulnDNS in sending packets.
7. (Not present in the figure) Once this query id is found, the cache of vulnDNS is poisoned for a ttl, time to live, very very long, so every request to vulnDNS about *bankofallan.co.uk* will be redirect on our server. At this point the attack can be considered completed successfully.

Another more synthetic representation of the attack can be seen in figure 3.

[1]The subdomain *random* is used just as a placeholder and it could be any URL encoded valid string.

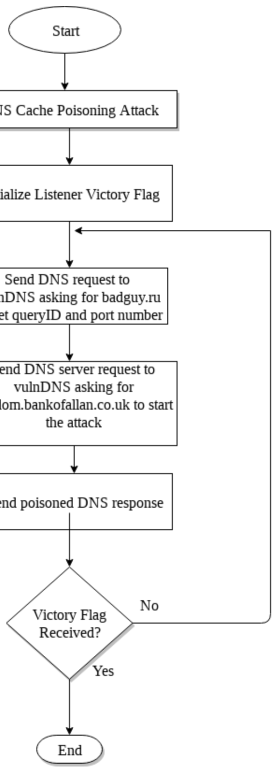

Start
DNS Cache Poisoning Attack
Initialize Listener Victory Flag
Send DNS request to vulnDNS asking for badguy.ru to get queryID and port number
send DNS server request to vulnDNS asking for random.bankofallan.co.uk to start the attack
Send poisoned DNS response
Victory Flag Received?
No
Yes
End

Figure 3: Funnel of the attack from the point of view of the attacker.

## 3 Code description and strategy - Question 2

*To craft a packet, you have to be a packet, and learn how to swim in the wires and in the waves.*[2]

*- Jean-Claude Van Damme*

The code We produce to complete our work can't be started on Windows because the main library We used is Linux system call heavy dependant[3].

The code is written in Python and it is divided into three main threads: the first one simulates the *badguy.ru* DNS server and collects recursive requests from vulnDNS in order to retrieve a starting query id and the destination port it uses for DNS answers, the second is the main attack vector, which we will discuss in detail later, and the last one only has the function of capturing the victory flag. Every thread is started by the main function and they're up while the global variable *ATTACK_GOING_ON* is set to *True*.

[2] One of the scapy console startup quotes.

[3] This is true according to scapy documentation[1], in any case the code has never been tested on Microsoft systems.

The DNS server thread launches his function called *dns_server_routine()*, that using the function *scapy.sniff()* captures the recursive DNS request sent by the attack thread. By analyzing the packet the threads finds out and set the *TARGET_PORT* and the *STARTING_QUERY_ID* and it passes on to the attack thread. Since sometimes the port We've choose to receive the request is not an ordinary DNS port, scapy can't always correctly interpret the UDP packet as DNS. In order to resolve this we've used an external parsing function found on stackoverflow [2]. The last operation the function does before restarting to listen for new packets is to answer to vulnDNS.

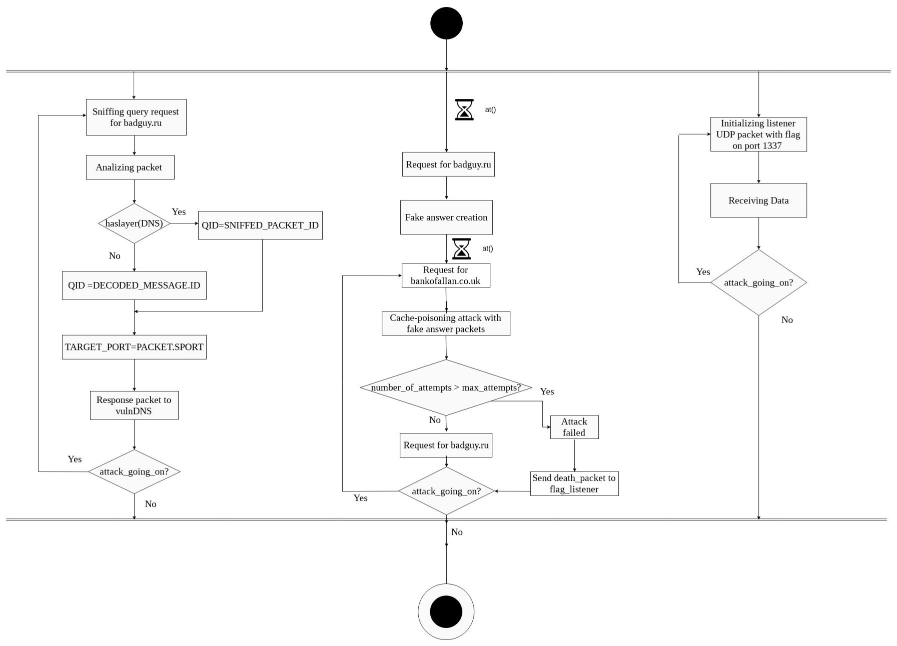
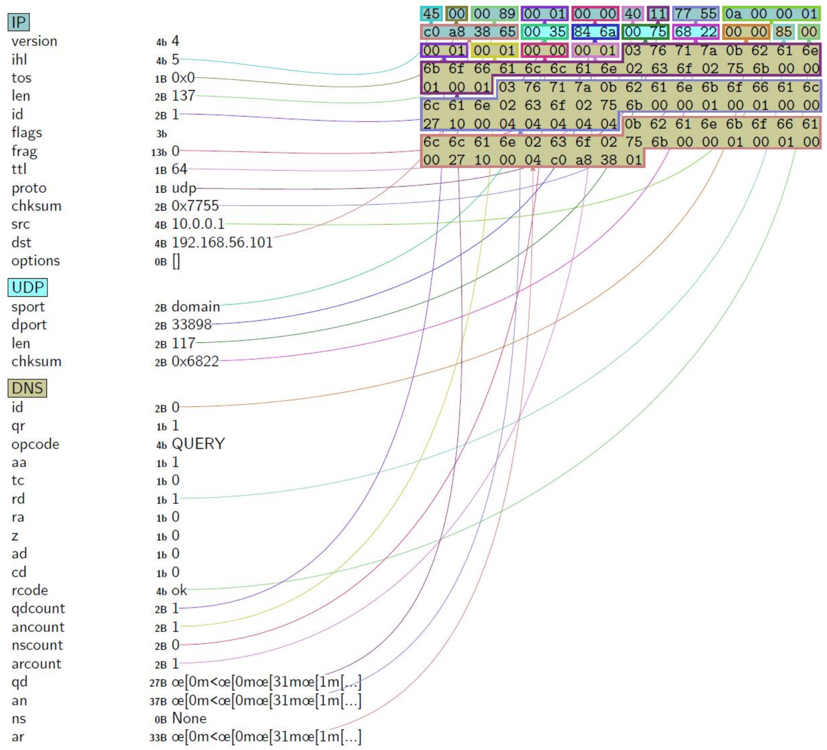


Figure 4: Activity diagram of the software.

The attack thread launches another function called *bite_the_rat()* which in the fist instance sends the DNS requests for *badguy.ru* in order to get starting query id and source port to vulnDNS in the previous thread, then it creates the fake answer packets of *bankofallan.co.uk*'s DNS. At this point the thread sends a recursive DNS request for bankofallan.co.uk to vulnDNS and then it flood with the fake answers the network. These packets will be sent one after the other to vulnDNS until our query id matches with the one of vulnDNS and in that case the attack is successful.

The last thread launches the function called *flag_victory_listener()* simply opens a socket and listens any UDP packet on port 1337. When a packet arrives the program assumes that the attack is over and sets *ATTACK_GOING_ON* to *False*.

To make the code more reliable and robust we've also added an attack attempts counter that at every iteration checks in *bite_the_rat()* thread if the maximum number of attempts has been reached. In this case the attack thread will send a *"death_flag_pkt"* packet to the listener to make the program exit gracefully. This has been done just for debug purposes and in a theoretical infinite run the program shall always poison the vulnDNS' cache. A more specific representation of the work of the threads is in figure 4.



```
IP
version     4b  4
ihl         4b  5
tos         1B  0x0
len         2B  137
id          2B  1
flags       3b
frag        13b 0
ttl         1B  64
proto       1B  udp
chksum      2B  0x7755
src         4B  10.0.0.1
dst         4B  192.168.56.101
options     0B  []
UDP
sport       2B  domain
dport       2B  33898
len         2B  117
chksum      2B  0x6822
DNS
id          2B  0
qr          1b  1
opcode      4b  QUERY
aa          1b  1
tc          1b  0
rd          1b  1
ra          1b  0
z           1b  0
ad          1b  0
cd          1b  0
rcode       4b  ok
qdcount     2B  1
ancount     2B  1
nscount     2B  0
arcount     2B  1
qd          27B œ[0m<œ[0mœ[31mœ[1m[...]
an          37B œ[0m<œ[0mœ[31mœ[1m[...]
ns          0B  None
ar          33B œ[0m<œ[0mœ[31mœ[1m[...]
```

```
45 00 00 89 00 01 00 00 40 11 77 55 0a 00 00 01
c0 a8 38 65 00 35 84 6a 00 75 68 22 00 00 85 00
00 01 00 01 00 00 00 01 03 76 71 7a 0b 62 61 6e
6b 6f 66 61 6c 6c 61 6e 02 63 6f 02 75 6b 00 00
01 00 01 03 76 71 7a 0b 62 61 6e 6b 6f 66 61 6c
6c 61 6e 02 63 6f 02 75 6b 00 00 01 00 01 00 00
27 10 00 04 04 04 04 04 0b 62 61 6e 6b 6f 66 61
6c 6c 61 6e 02 63 6f 02 75 6b 00 00 01 00 01 00
00 27 10 00 04 c0 a8 38 01
```

Figure 5: Dump of an example of attack packet.

In figure 5 is also possible to see the hexadecimal dump of one the attack typical packet, the fake answer generated in the *bite_the_rat()* thread. It is interesting to note the various fields set for the attack:

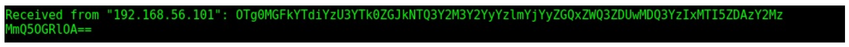
- Source IP is the one of *bankofallan.co.uk*.
- Destination IP is the one of vulnDNS.
- Destination port is the one sniffed by *dns_server_routine()*.
- qr = 1 means this packet is a response.
- rd = 1 means recursion desired.

- The query id is the only value that changes from the various packets. In this case is 0 because this is the dump of the first packet generated.
- The DNS records which are the payload of the attack are not explicitly represented, so to have more information about them please refer directly to the code.

A *.pcap* dump of the attack packet represented in figure 5 is also present in the directory and can be easily inspected with Wireshark, Tshark or every related tool.

The code also contains documented minor auxiliary functions that have been omitted from this documentation due to their simplicity.

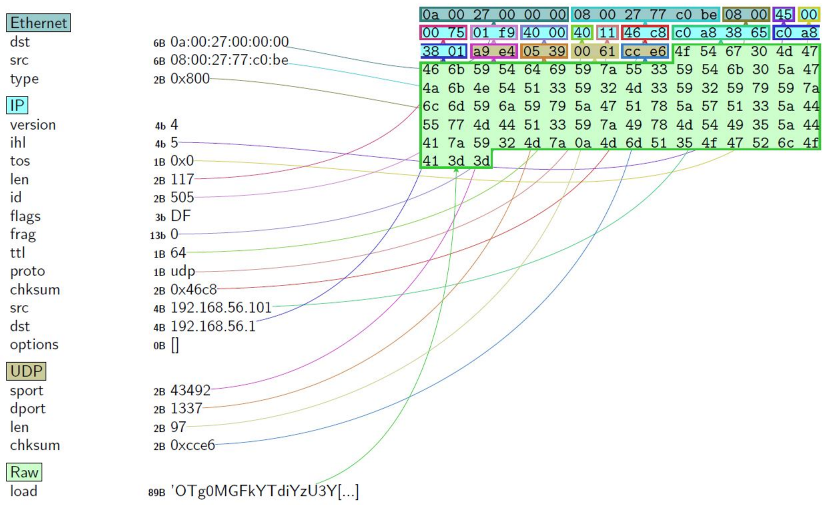
## 4 Victory Flag - Question 3



```
Received from "192.168.56.101": OTg0MGFkYTdiYzU3YTk0ZGJkNTQ3Y2M3Y2YyYzlmYjYyZGQxZWQ3ZDUwMDQ3YzIxMTI5ZDAzY2Mz
MmQ5OGRlOA==
```

Figure 6: Victory flag obtained during the attack.

The flag we found is *"OTg0MGFkYTdiYzU3YTk0ZGJkNTQ3Y2M3Y2YyYz lmYjYyZGQxZWQ3ZDUwMDQ3YzIxMTI5ZDAzY2Mz MmQ5OGRlOA=="* and it is also reported in figure 6. In the packet visualization in figure 7 is interesting to note the destination port, 1337 and that after UDP level there's only the flag which is partially reported here. A *.pcap* dump of the victory UDP packet represented in figure 6 is also present in the directory.



```
Ethernet
  dst        6B 0a:00:27:00:00:00
  src        6B 08:00:27:77:c0:be
  type       2B 0x800
IP
  version    4b 4
  ihl        4b 5
  tos        1B 0x0
  len        2B 117
  id         2B 505
  flags      3b DF
  frag      13b 0
  ttl        1B 64
  proto      1B udp
  chksum     2B 0x46c8
  src        4B 192.168.56.101
  dst        4B 192.168.56.1
  options    0B []
UDP
  sport      2B 43492
  dport      2B 1337
  len        2B 97
  chksum     2B 0xcce6
Raw
  load      89B 'OTg0MGFkYTdiYzU3Y[...]
```

```
0a 00 27 00 00 00 08 00 27 77 c0 be 08 00 45 00
00 75 01 f9 40 00 40 11 46 c8 c0 a8 38 65 c0 a8
38 01 a9 e4 05 39 00 61 cc e6 4f 54 67 30 4d 47
46 6b 59 54 64 69 59 7a 55 33 59 54 6b 30 5a 47
4a 6b 4e 54 51 33 59 32 4d 33 59 32 59 79 59 7a
6c 6d 59 6a 59 79 5a 47 51 78 5a 57 51 33 5a 44
55 77 4d 44 51 33 59 7a 49 78 4d 54 49 35 5a 44
41 7a 59 32 4d 7a 0a 4d 6d 51 35 4f 47 52 6c 4f
41 3d 3d
```

Figure 7: Dump of the victory packet.

## References

[1] Scapy package python. Page of scapy documentation. https://scapy.readthedocs.io/en/latest/index.html. Accessed: May 27, 2019.

[2] StackOverflow.com. Reading dns packets in python. https://stackoverflow.com/questions/16977588/reading-dns-packets-in-python. Accessed: May 27, 2019.

[3] Steve Friedl's Unixwiz.net Tech Tips. An illustrated guide to the kaminsky dns vulnerability. http://unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html. First published: 2008/08/07. Accessed: May 27, 2019.