DNS Cache Poisoning Attack

Ethical Hacking Lab

Federica Consoli MAT. 1538420

April 24, 2018

Abstract

The goal of the assignment was to simulate a DNS Cache Poisoning attack, following the structure of Dan Kaminsky's attack. In this report I will discuss the steps taken to prepare for the attack, how it was actually implemented and what results were obtained.

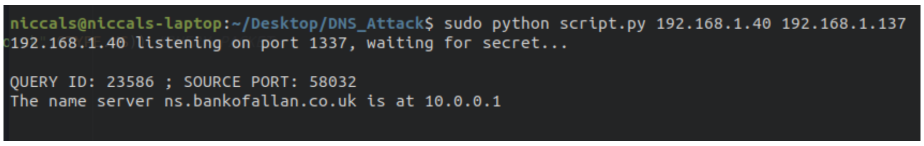
1 Getting started

The idea behind this attack is to make the DNS accept a specially forged packet. By design, a DNS accepts a response packet only if it matches one of its pending queries. For this to happen, the following conditions need to be satisfied:

- the response arrives on the same UDP port it was sent from
- the response's query ID matches the one of the pending query
- the response's question section matches the one of the pending query

The first step is to obtain the query id and the source port from the DNS server.

I crafted a query packet for the DNS server, asking it to look up the address for `badguy.ru`. Since this is a lab environment, `badguy.ru` resolves to my machine's address, while in a real setup it could resolve, for example, to the address of a webserver owned by the attacker (which would allow him to see the incoming traffic on port 53). I then sent the query to the DNS on port 53 by using a socket. The DNS then proceeds to contact my machine (which is authoritative for `badguy.ru`), which allowed me to retrieve the information I needed by using a second socket listening on port 53.



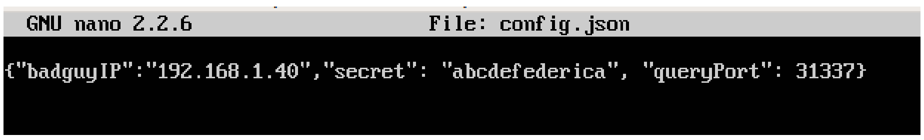
```
niccals@niccals-laptop:~/Desktop/DNS_Attack$ sudo python script.py 192.168.1.40 192.168.1.137
192.168.1.40 listening on port 1337, waiting for secret...

QUERY ID: 23586 ; SOURCE PORT: 58032
The name server ns.bankofallan.co.uk is at 10.0.0.1
```

Figure 1: First steps

Using a similar approach, I queried the DNS server for the name server responsible for the victim's domain; by analyzing the response I received I was able to extract both the hostname and the IP address of the name server I was looking for.

This step is necessary because the source ip for the forged DNS responses needs to be spoofed, or else the response would not be accepted by the DNS, since it is not coming from an authoritative server for the targeted domain. Once I obtained the IP address, I added it as a static address to my machine using the command `ip addr add 10.0.0.1 dev lo`, which later allowed me to perform the spoofing.

I then proceeded to configure the file `config.json` on my virtual machine as shown below. Finally, I connected the VM to my laptop using bridged networking (sometimes it is necessary to manually set a default gateway for the VM, or else it will not be able to send out any packet).



```
  GNU nano 2.2.6                    File: config.json

{"badguyIP":"192.168.1.40","secret": "abcdefederica", "queryPort": 31337}
```

Figure 2: the VM configuration file

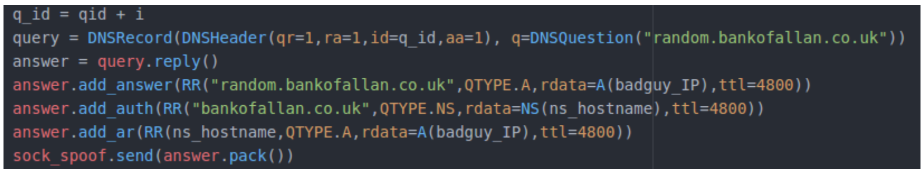
2 Implementation

The attack was implemented in Python, using the `dnslib` library to easily create and manipulate DNS packets. The program is launched via `sudo python bad_guy_IP target_IP`, where `bad_guy_IP` is the attacker's IP addres (in this case, my own machine's), while `target_IP` represents the IP of the target (in this case, the VM).

After checking if the arguments are correct, the program spawns two concurring threads, with two different target function: `poison` and `listen`.

The `poison` function is the one that actually implements the attack. As mentioned before, this function uses two functions (`get_qid_and_port` and `get_ns`) to retrieve the information needed to carry out the attack, namely the query ID, the source port of the DNS, the IP and the hostname of the target's name server.

After this, the program creates a socket for UDP traffic and binds it to the IP address obtained from `get_ns` (which was previously added to one of my network interfaces). It then queries the DNS for the address of the domain `random.bankofallan.co.uk` and starts flooding the server with response packets.



```
q_id = qid + i
query = DNSRecord(DNSHeader(qr=1,ra=1,id=q_id,aa=1), q=DNSQuestion("random.bankofallan.co.uk"))
answer = query.reply()
answer.add_answer(RR("random.bankofallan.co.uk",QTYPE.A,rdata=A(badguy_IP),ttl=4800))
answer.add_auth(RR("bankofallan.co.uk",QTYPE.NS,rdata=NS(ns_hostname),ttl=4800))
answer.add_ar(RR(ns_hostname,QTYPE.A,rdata=A(badguy_IP),ttl=4800))
sock_spoof.send(answer.pack())
```

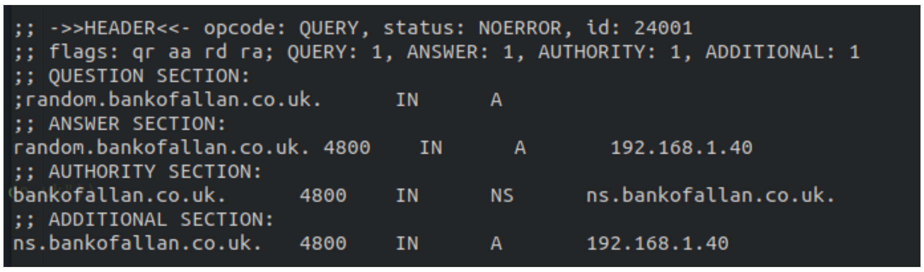
Figure 3: Response structure

From the code snippet above, we can notice the following:

1. the *query ID* is incremented at each iteration, to try and match the query ID generated from the DNS server (which is randomly incremented)

2. the *answer* section contains the resolution for the requested domain, with an IP address of choice (in a real attack, this would be the address of the attacker's web server).

3. the *authority* section contains the hostname for the authoritative name server for `bankofallan.co.uk`, which was previously retrieved with the function `get_ns`. However, in the *additional record* section, the name server is linked to the IP of my machine: this means that, in a real setup, an attacker would become the owner of the whole zone after a successful poisoning.



```
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 24001
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 1
;; QUESTION SECTION:
;random.bankofallan.co.uk.      IN      A
;; ANSWER SECTION:
random.bankofallan.co.uk. 4800    IN      A       192.168.1.40
;; AUTHORITY SECTION:
bankofallan.co.uk.       4800    IN      NS      ns.bankofallan.co.uk.
;; ADDITIONAL SECTION:
ns.bankofallan.co.uk.    4800    IN      A       192.168.1.40
```

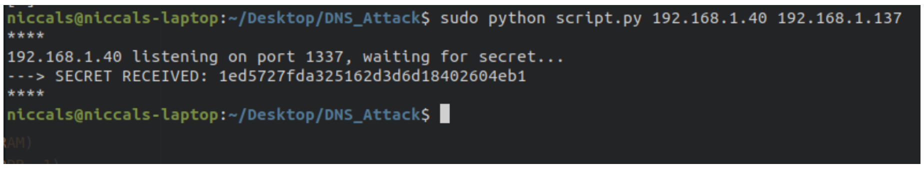
Figure 4: The forged response

The program generates 50 responses for each query: after that, it will invoke the `get_qid_and_port` function to retreive the new current query ID, and it will try again (until the secret is received).

The `listen` function, on the other hand, creates a socket listening on port 1337. This is where the *secret* is received, which indicates that the cache was successfully poisoned. When the secret is received, it is simply printed by the function.

3 The secret

As mentioned before, when I successfully carried out the attack, I received a *secret* on port 1337. The string I received represents the hash of the string I inserted in the field ”secret” in the `config.json` file (as previously shown in Figure 2).



```
niccals@niccals-laptop:~/Desktop/DNS_Attack$ sudo python script.py 192.168.1.40 192.168.1.137
****
192.168.1.40 listening on port 1337, waiting for secret...
---> SECRET RECEIVED: 1ed5727fda325162d3d6d18402604eb1
****
niccals@niccals-laptop:~/Desktop/DNS_Attack$
```

Figure 5: The secret hash