67607 – Exercise 2 – BIND9 DNS Cache Poisoning

Instructed by Amit Klein, written by Agam Ebel
(Based on a work by Ilay Bahat and David Keisar Schmidt)

2024-2025

Contents

1 BIND9 DNS Cache Poisoning

In this section, we will introduce the BIND9 DNS cache poisoning attack described by Amit Klein in 2007 and outline the steps involved in setting up the environment necessary for mounting the attack.

Your goal is to poison BIND9's cache so that it will resolve (from cache) the name `www.example.cybercourse.com` into the IP address `6.6.6.6`.

1.1 Our Environment

Our BIND9 DNS server is vulnerable to cache poisoning and deployed within a docker container. The container is built by installing the necessary packages and configuring the BIND9 DNS Server.

We also have an authoritative name server deployed within a docker container. It will be used as a "root" name server. We limit our work only to domains that are subdomains of `cybercourse.com`. This domain serves as our effective root, and no record outside this domain should ever be attacked in this exercise. Moreover, all of the containers are on an isolated network, so trying to request any record outside this domain (`cybercourse.com`) will fail.
Please note that we also added a delay to the root server on purpose, to make sure we can inject our spoofed packets before the genuine answer arrives from the root.

The attacker will control two components: an authoritative name server and a client. The client will try to poison BIND9's cache by sending spoofed DNS responses. Each component will run in a separate container.

We also deployed another client within a docker container. We will use it to check if our attack was successful by sending a request to `www.example.cybercourse.com` and see if we got our poisoned record as an answer (for example, we can do it by issuing a `ping` request to the domain and observe to which IP the request was sent). All the containers are connected to a virtual LAN named `cp-net`, this is an isolated network where we operate.

1.2 Introduction

DNS cache poisoning aims to make a DNS resolver return an incorrect response so that users who resolve names to addresses through this DNS server and then connect to the IP addresses returned from it are directed to the wrong IP address. We do that by inserting false information into the DNS resolver cache.

An example of a DNS cache poisoning attack is "BIND9 DNS Cache Poisoning" by Amit Klein in 2007. In the paper, he shows that one can predict up to 10 values for the next transaction ID (from now on we will refer to it as "TXID") when the TXID is even, and because the UDP source port is static in BIND 9.4.1, it suffices to predict the TXID and by that spoof DNS answers and poison the cache, which is what we will demonstrate in this section.

1.3 General Information

- To log in to the environment, use username "cpEnv" and password 5260
- When compiling your attack files, use `gcc -Wall -Wextra -Werror` *`program-name`*`.c -lldns -lpcap -o` *`desired-compiled-file-name`*
  Compile your source code from within the right container (the attacker's authoritative name server code from the attacker's authoritative name server container and the attacker's client code from the attacker's client container), and make sure there are no compilation warnings! We will take points off for any compilation warnings.
- In order to find the next 10 TXIDs, we need to know the values of the masks the registers use (which we will refer to as "taps"). The first LFSR register uses "tap1" and the second uses "tap2".
  TAP1 value: `0x80000057`
  TAP2 value: `0x80000062`
- We can use LDNS library to simplify creating and processing DNS packets.
- The attack target is a BIND9 DNS caching server.
- It is assumed that the domain (name) we try to poison (i.e. we try to inject a record for) is not yet cached (or that its cache entry has expired).
- The containers are connected to a virtual LAN named `cp-net`. This is an isolated network and the attack needs to be developed only over this network!
  Note that the containers are connected only to this network, so when checking and developing your attack, do it from inside the containers only!
- All of the components are deployed within docker containers and have the following IP and MAC addresses:

Container name	IP address	MAC address
vulnerable BIND9 DNS resolver	`192.168.1.203`	`02:42:ac:11:00:03`
"effective root" authoritative name server	`192.168.1.204`	`02:42:ac:11:00:04`
attacker's authoritative name server	`192.168.1.201`	`02:42:ac:11:00:01`
attacker's "client"	`192.168.1.202`	`02:42:ac:11:00:02`
"legitimate client"	`192.168.1.205`	`02:42:ac:11:00:05`

- You can work on your source code file from within the VM (you should place the attacker's authoritative name server files under the directory /home/cpenv/Desktop/attackerAuth on the VM), and you'll be able to access it in the attacker's authoritative name server container under /tmp/attackerAuthNS (which is mounted to the /home/cpenv/Desktop/attackerAuth). For the attacker's client files, you should place the files under the directory /home/cpenv/Desktop/Mclient on the VM), and you'll be able to access it in the attacker's client container under /tmp/mclient (which is mounted to the /home/cpenv/Desktop/Mclient).

  Therefore, it is enough to change the source code itself inside the "attackerAuth" directory on the VM (or in case of the attacker's client source code, it is enough to change the source code itself inside the "Mclient" directory on the VM) and the changes will also applied in the right container. You must compile the source code from inside the right container (with the following command: gcc -Wall -Wextra -Werror attacker-file.c -lpcap -lldns -o compiled-file-name) and ensure that there are no compilation warnings, and run the executable from the containers. This is how we will compile and run your code ourselves (in our container environment which is practically identical to yours, so this guarantees you're working with the same compilation and library environment we use), and we will take off points for compilation warnings (and lots of points if it fails to compile or do the job).

1.3.1 BIND9 Cache Poisoning Attack Reminders

We saw that in order to poison BIND9's cache, we first need to know if the transaction ID (TXID) is even. If so, we have 10 options for the next TXID. We can find them in the following way:

1. If both rightmost bits in the LFSR registers are "0":

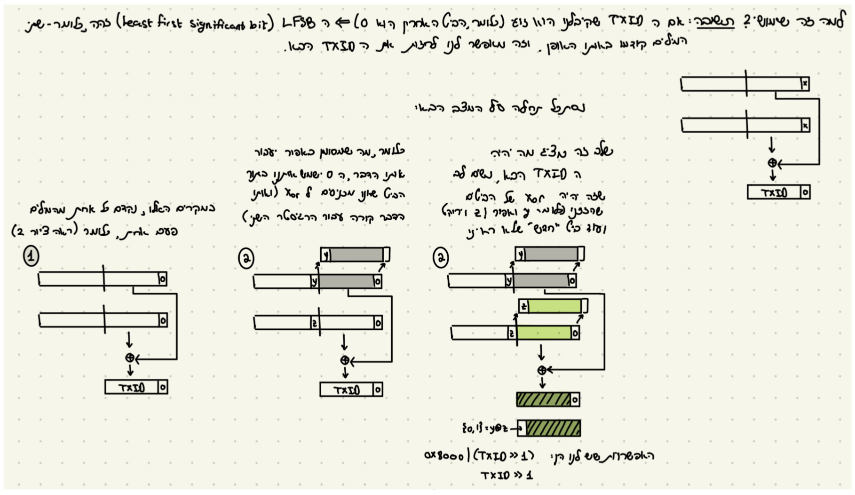


Figure 1: Finding the next TXID when both rightmost bits in the LFSR registers are "0"

2. If both rightmost bits in the LFSR registers are “1”:

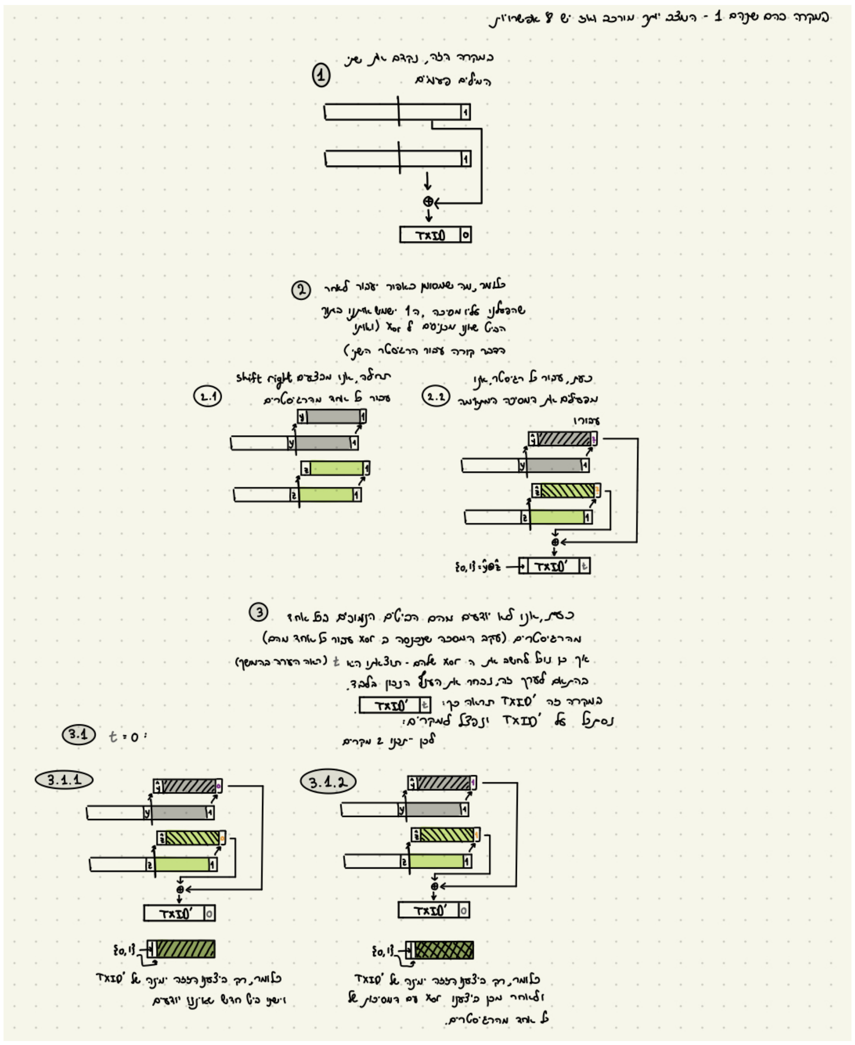
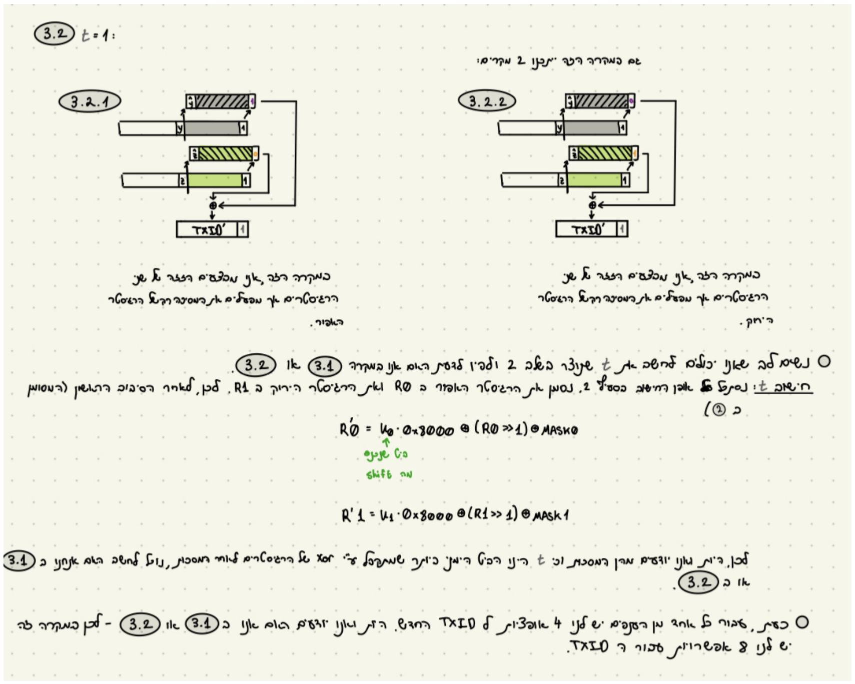


Figure 2: Finding the next TXID when both rightmost bits in the LFSR registers are “1”



גם במקרה הזה "יתכנו" 2 מקרים:

במקרה הזה, אנו מבצעים הזזה של שני הרגיסטרים אך מפעילים את המסיכה רק על הרגיסטר האפור.

במקרה הזה, אנו מבצעים הזזה של שני הרגיסטרים אך מפעילים את המסיכה רק על הרגיסטר הירוק.

○ נשים לב שאנו יכולים לחשב את t שנוצר בשלב 2 ולפיו לדעת האם אנו במקרה 3.1 או 3.2.
חישוב t: נעקוב אחר החישוב בסעיף 2. נסמן את הרגיסטר האפור ב R0 ואת הרגיסטר הירוק ב R1. לכן, לאחר הסיבוב הראשון (המסומן ב ②)

$$R'0 = \underset{\text{ביט שנכנס מה shift}}{u_0} \cdot 0x8000 \oplus (R0 \gg 1) \oplus MASK0$$

$$R'1 = u_1 \cdot 0x8000 \oplus (R1 \gg 1) \oplus MASK1$$

לכן, היות ואנו יודעים מהן המסכות, וכי t הינו הביט הימני ביותר שמתקבל ע"י xor של הרגיסטרים לאחר המסכות, ניתן לחשב האם אנחנו ב 3.1 או ב 3.2.

○ כמו כן, עבור כל אחד מן הענפים יש לנו 4 אופציות ל TXID החדש. היות ואנו יודעים האם אנו ב 3.1 או 3.2 - לכן במקרה זה יש לנו 8 אפשרויות עבור ה TXID.

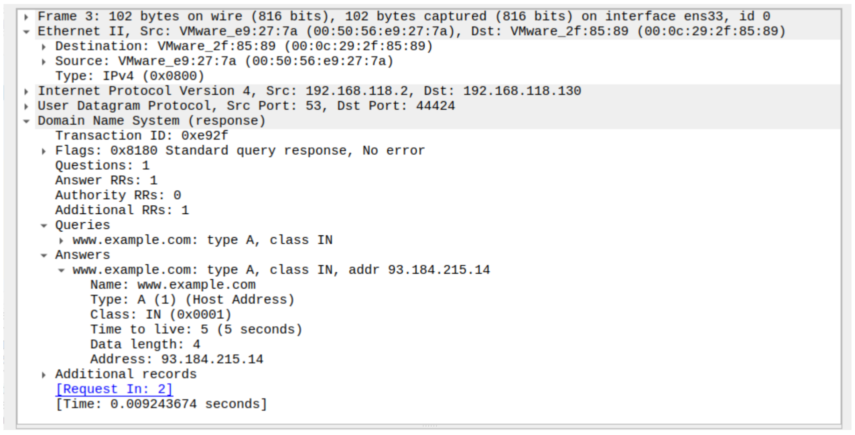
Figure 3: Finding the next TXID when both rightmost bits in the LFSR registers are “1” (Contd.)

It is also important to note that the 16-bit transaction ID is the 16 least significant bits of the XOR value of the two variables. It is serialized with the most significant byte first, then least significant byte (big-endian style).
As stated in the paper: “Note that the probability of the values is not uniform: since the probability for two 0 bits is $\frac{1}{2}$, it follows that each of the two values associated with this branch has probability $\frac{1}{4}$, while the probability of the two 1 bits is $\frac{1}{2}$, which means that each value of the eight values associated with this branch has probability $\frac{1}{16}$”.

1.3.2 Notes about Wireshark

As stated on the Wireshark website: “Wireshark is a network packet analyzer. A network packet analyzer presents captured packet data in as much detail as possible.”
Wireshark is a free, open-source network analyzer. One can download it for Windows, Linux, and macOS from here (note that we already installed Wireshark on the exercise environment). We will use it to analyze the traffic in our virtual LAN (mostly focusing on DNS packets), debug the attack attempts, and build the attack. Herein we provide a short introduction to using Wireshark.

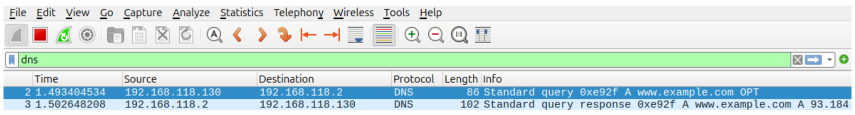
When launching Wireshark, we start by choosing an interface to listen on. After we choose an interface, we can see which packets are sent on this network and the content of each packet. For example, here is a screenshot of packet details for a captured response to a DNS query for `www.example.com` we sent earlier (we can see the details of a packet by double-clicking on the packet line – the packet details will appear in a pop-up window) as we can see here:



```
▸ Frame 3: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface ens33, id 0
▾ Ethernet II, Src: VMware_e9:27:7a (00:50:56:e9:27:7a), Dst: VMware_2f:85:89 (00:0c:29:2f:85:89)
  ▸ Destination: VMware_2f:85:89 (00:0c:29:2f:85:89)
  ▸ Source: VMware_e9:27:7a (00:50:56:e9:27:7a)
    Type: IPv4 (0x0800)
▸ Internet Protocol Version 4, Src: 192.168.118.2, Dst: 192.168.118.130
▸ User Datagram Protocol, Src Port: 53, Dst Port: 44424
▾ Domain Name System (response)
    Transaction ID: 0xe92f
  ▸ Flags: 0x8180 Standard query response, No error
    Questions: 1
    Answer RRs: 1
    Authority RRs: 0
    Additional RRs: 1
  ▾ Queries
    ▸ www.example.com: type A, class IN
  ▾ Answers
    ▾ www.example.com: type A, class IN, addr 93.184.215.14
        Name: www.example.com
        Type: A (1) (Host Address)
        Class: IN (0x0001)
        Time to live: 5 (5 seconds)
        Data length: 4
        Address: 93.184.215.14
  ▸ Additional records
    [Request In: 2]
    [Time: 0.009243674 seconds]
```

Figure 4: Packet content in Wireshark

We can also apply a “display filter” in order to show only certain types of packets (such as DNS, UDP, TCP...) and packets that were sent to/from specific ports. For example, for the DNS request we sent in the previous example, we can add a filter only for DNS packets, as we can see here:


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
dns

   Time          Source            Destination       Protocol  Length Info
 2 1.493404534   192.168.118.130   192.168.118.2     DNS           86 Standard query 0xe92f A www.example.com OPT
 3 1.502648208   192.168.118.2     192.168.118.130   DNS          102 Standard query response 0xe92f A www.example.com A 93.184.
```

Figure 5: Packet capturing with filter

For further reading about Wireshark display filters, please refer to `https://www.wireshark.org/docs/man-pages/wireshark-filter.html`

1.3.3 Notes about DNS records

Recall that we have many types of DNS resource records (from now on, we will refer to them as “RR”). Two useful types that we will use in this attack are `CNAME` records and `A` records.

The **wire**[1] structure of a DNS RR is the following: `<name> <ttl> <class> <type> <rdlength> <radata>`. For example, an `A` record for `www.example.com` looks like this: `www.example.com 500 IN A 10.20.30.40` (notice that this is not the real IPv4 address of `www.example.com`, but a made-up one only for demonstrating).

`A` records are a fundamental component of the Domain Name System (DNS). Those records provide the IPv4 address of a given domain (for IPv6 addresses, we use `AAAA` records), and the most common usage of those records is IP address lookup (matching a domain name

[1] i.e. as it appears on the network

to an IPv4 address), there are also less common usages for this kind of record which we will not mention here.

A "canonical name" (CNAME) record points from an alias domain to a "canonical" domain and they must point to a domain (and not to an IP address). `CNAME` record can point to an alias domain, where the alias domain itself points to another alias name until we get to a record that contains a canonical name for which we have an `A` record. This practice is called "CNAME chaining".

To read more about DNS records, please see `https://www.ionos.com/digitalguide/hosting/technical-matters/dns-records/`

1.4 Starting the environment

Our attack environment consists of a VM and 5 containers. First, download the environment at the following link.
Note – the environment disk requires at least 35GB free on your computer, make sure to have enough free space to upload the disk.
Start the virtual machine using the instructions for uploading a virtual disk to a VM which we provided in the "Environment Setup" file.
Next, run `./run_root_container.sh` which is located in `/home/cpenv/Desktop/` to start our "root name server" container and start the "named" daemon of the name server (to enable the resolution process), we can do that in the following way:

1. `cd Desktop`
2. `./run_root_container.sh`
3. `named -4` (to start the "named" daemon. Notice that we run it from inside the container)
4. `lsof -i -P -n | grep LISTEN` (to make sure our server is listening. Notice that we run it from inside the container)

To start the BIND9.4.1 recursive resolver:

1. `docker start bind9res`
2. `docker exec -it bind9res /bin/bash`
3. `named -4` (to start the "named" daemon. Notice that we run it from inside the container)
4. `lsof -i -P -n | grep LISTEN` (to make sure our server is listening. Notice that we run it from inside the container)

To start the Attacker's authoritative name server:

1. `docker start attacker-auth`
2. `docker exec -it attacker-auth /bin/bash`

To start the Attacker's client:

1. `docker start attacker-client`
2. `docker exec -it attacker-client /bin/bash`

To start the "legitimate client" (which we will use only for checking our attack):

1. `docker start client`
2. `docker exec -it client /bin/bash`
3. From inside the container, we need to change the `/etc/resolv.conf` file to enable resolution from the client, we can do it by changing the nameserver to our recursive resolver (we can edit the file from within the container using `vim`). Change the `nameserver` directive per the comment.

```
# Generated by Docker Engine.
# This file can be edited; Docker Engine will not make further changes once
# it has been modified.

nameserver 127.0.0.11 # here - change to 192.168.1.203
search localdomain
options edns0 trust-ad ndots:0

# Based on host file: '/etc/resolv.conf' (internal resolver)
# ExtServers: [host(127.0.0.53]
# Overrides: []
# Option ndots from: internal
```

1.5 EX2 – attack reconstruction guidelines

This attack contains many levels and could be hard to follow. For your convenience, we added the attack flow you should follow to implement the attack.

1. Send a request for `www.attacker.cybercourse.com` from the attacker's client to the recursive resolver.
2. According to the parity of the TXID, the attacker's authoritative name server operates as follows:

(a) If the TXID is even, the attacker's authoritative name server sends the (even) TXID and the source port to the attacker's client, and also sends to the BIND9 DNS resolver a response that contains a CNAME record for the domain we try to poison the resolver's cache with (`www.example.cybercourse.com`).

(b) If the TXID is odd, it will respond with a CNAME record for another resource in the attacker's authoritative name server (i.e. `ww1.attacker.cybercourse.com`), and this is repeated until an even TXID is found.

3. Finish the analysis of the next TXID (notice that we provided some of the analysis in subsubsection 1.3.1). This is a fundamental step for the attack to succeed, as we need it to predict the next TXID.

4. In case the TXID is even and the attacker's client got the TXID (and the resolver's source port) from the attacker's authoritative name server, it will calculate the next TXID (10 possible values), and create 10 spoofed answers to poison BIND9's cache, providing an A record with value `6.6.6.6`.

5. During the time the query is "in the air", the attacker's client tries to spoof the answer by sending the 10 spoofed DNS answers to the BIND9 recursive resolver.

6. To check if the attack was successful, the client then asks for `www.example.cybercourse.com` and sees if the returned IP by the resolver is `6.6.6.6`. We can use `dig` for sending DNS requests to the recursive resolver for the name `www.example.cybercourse.com`. Use `dig @`*`IP-addr-of-DNS-server the-desired-site`*.

1.6 Tips and Tricks

- All of the components are deployed within docker containers and have the following IP and MAC addresses:

Container name	IP address	MAC address
vulnerable BIND9 DNS resolver	192.168.1.203	02:42:ac:11:00:03
'effective root" authoritative name server	192.168.1.204	02:42:ac:11:00:04
attacker's authoritative name server	192.168.1.201	02:42:ac:11:00:01
attacker's "client"	192.168.1.202	02:42:ac:11:00:02
"legitimate client"	192.168.1.205	02:42:ac:11:00:05

- To clear the "legitimate client" stub resolver cache, restart the container (we can do that by using `docker restart client` (where "client" is the container name)) and configure `resolv.conf` again, as explained in item 1.4.

- Do not forget to flush the recursive resolver cache between one attack attempt to another. We can do that using `rndc flush` inside the recursive resolver container.

- Use LDNS library for creating your DNS requests and responses packets (notice that it is already installed in the attacker's authoritative name server and client containers).
- There are many ways to send packets to the different nodes in the network, some of them are the use of `raw sockets`, `cooked sockets`, or `libpcap` (for injecting packets inside the network. Notice, that you will need to build the packet from layer 2 if you choose `libpcap`).
- Use Wireshark to monitor and inspect the packets you send over the virtual LAN. It can help you debug and build your attack attempt.
- To listen to the traffic on our virtual LAN with Wireshark from the VM, we first find the ID of our docker network using `docker network inspect cp-net`, check what is the ID value, and choose the `br-`*`cp-net network ID`* interface to listen on.
- From within the containers, the interface we will send our traffic on is `eth0`.
- When developing your attack attempt, we recommend building it incrementally. Start by sending a "real" packet (not part of the attack flow) and ensure it works as intended. Next, implement the attacker's name server and the attacker's client logic. Finally, integrate all the components to create the complete attack flow.
- Test your code! Try to run your attack on the provided server by sending the spoofed packets and checking the output on the server container (using `dig`) or in the client itself when sending another request to the resolver for the domain `www.example.cybercourse.com`. Ensure this is consistent by running several attack cycles (between each attack cycle, do not forget to clean the BIND9.4.1 resolver cache by using `rndc flush`. Also, clear the legitimate client stub resolver cache!)
- To check your attack before submitting, make sure to check that you get your spoofed IP when sending a request to `www.example.cybercourse.com` from the "legitimate client" container (i.e. we can do it by sending a request to the domain using `ping` and make sure we get the spoofed address (`6.6.6.6`) instead of the legitimate address). Make sure to clear the legitimate client stub resolver cache first!
- We can see the genuine IP address that `www.example.cybercourse.com` is mapped to by sending a DNS request using `dig` to the 'root" name server.
- When compiling each file of your solution attempt, use `gcc -Wall -Wextra -Werror` *`program-name`*`.c -lldns -lpcap -o` *`desired-compiled-file-name`*.
  Compile your source code from within the right container (the attacker's authoritative name server code from the attacker's authoritative name server container and the attacker's client code from the attacker's client container), and make sure there are no compilation warnings! We will take points off for any compilation warnings.

1.6.1 Tips for Debugging

1. For debugging purposes, we can use "netstat" for finding open UDP ports in our resolver.

2. For debugging purposes, we can use `dig` for sending DNS requests to some resolver/name server in the network. Use `dig @`*`IP-addr-of-DNS-server the-desired-site`*`.`

1.7 What to Submit

1.7.1 Important Remarks

Please follow these instructions carefully to ensure full credit for your submission. Failure to meet any of these requirements will result in a deduction of points.

- Make sure to send only 10 spoofed packets in your attack attempt!

- When compiling your attack files, use `gcc -Wall -Wextra -Werror` *`program-name`*`.c -lldns -lpcap -o` *`desired-compiled-file-name`*
  Compile your source code from within the right container (the attacker's authoritative name server code from the attacker's authoritative name server container and the attacker's client code from the attacker's client container), and make sure there are no compilation warnings! We will take points off for any compilation warnings.

- When running the compiled code, run it without any command line arguments (for each of your files).

- Make sure your code does not print anything to the screen before submitting your attack attempt.

- The exit code from the attacker's client and attacker's authoritative name server code should be 0.

- Please clean up your code before submission – remove redundant code, add useful comments, use reasonable names for variables and functions, etc. – but please double-check that everything still works afterwards!

1.7.2 Submission

The submission is done via Moodle in pairs. Please submit a **single zip file named "ex2.zip"** which contains the following files:

1. Your attacker's client code (in `C`). The file should be named "ex2_client.c".
   When compiling the file, use `gcc -Wall -Wextra -Werror` *`program-name`*`.c -lldns -lpcap -o` *`desired-compiled-file-name`*
   Compile your source code from within the attacker's client container, and make

sure there are no compilation warnings! We will take points off for any compilation warnings.

2. Your attacker's authoritative name server code (in `C`). The file should be named "ex2_server.c".
   When compiling the file, use `gcc -Wall -Wextra -Werror` *`program-name`*`.c -lldns -lpcap -o` *`desired-compiled-file-name`*
   Compile your source code from within the attacker's authoritative name server container, and make sure there are no compilation warnings! We will take points off for any compilation warnings.

3. A file containing a short explanation of your attack attempt, how you built the attacker's authoritative name server and the attacker's client.
   The file should be named "explanation.txt".

4. A TXT file contains a single line with the submitters' IDs (separated by a comma. i.e. 123456789,987654321). If you submit the exercise alone (after getting approval from the course staff), enter your ID only (without a comma, i.e. 123456789).
   The file should be named "readme.txt".

5. Please pay attention to file names (typos...) and do not submit any folders or extra files – just a "flat" zip file with those four files only.

1.8 Resources

- Amit Klein, "BIND9 DNS Cache Poisoning", 2007
- `https://www.cloudflare.com/learning/dns/dns-cache-poisoning/`
- `https://www.cloudflare.com/learning/dns/dns-records/dns-a-record/`
- `https://www.cloudflare.com/learning/dns/dns-records/dns-cname-record/`
- `https://www.ionos.com/digitalguide/hosting/technical-matters/dns-records/`
- `https://www.nlnetlabs.nl/documentation/ldns/design.html`
- `https://www.nlnetlabs.nl/projects/ldns/about/`
- `https://www.wireshark.org/docs/wsug_html_chunked/ChapterIntroduction.html`