The Kaminsky Attack Lab

1 Lab Overview

The objective of this lab is for students to gain the first-hand experience on the remote DNS cache poisoning attack, also called the Kaminsky DNS attack. DNS (Domain Name System) is the Internet's phone book; it translates hostnames to IP addresses and vice versa. This translation is through DNS resolution, which happens behind the scene. DNS attacks manipulate this resolution process in various ways, with an intent to misdirect users to alternative destinations, which are often malicious. This lab focuses on a particular DNS attack technique, called *DNS Cache Poisoning attack*. In another SEED Lab, we have designed activities to conduct the same attack in a local network environment, i.e., the attacker and the victim DNS server are on the same network, where packet sniffing is possible. In this remote attack lab, packet sniffing is not possible, so the attack becomes much more challenging than the local attack. This lab covers the following topics:

- DNS and how it works
- DNS server setup
- DNS cache poisoning attack
- Spoofing DNS responses
- Packet spoofing

Readings and videos. Detailed coverage of the DNS protocol and attacks can be found in the following:

- Chapter 18 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.
- Section 7 of the SEED Lecture, *Internet Security: A Hands-on Approach*, by Wenliang Du. See details at `https://www.handsonsecurity.net/video.html`.

Lab environment. This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website, and run the SEED VM on your own computer. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud.

2 Lab Environment Setup (Task 1)

The main target for DNS cache poisoning attacks is local DNS server. Obviously, it is illegal to attack a real server, so we need to set up our own DNS server to conduct the attack experiments. The lab environment needs four separate machines: one for the victim, one for the DNS server, and two for the attacker. The lab environment setup is illustrated in Figure 1.

We put all these machines on the same LAN only for the sake of simplicity. Students are not allowed to exploit this fact in their attacks; they should treat the attacker machine as a remote machine, i.e., the attacker cannot sniff packets on the LAN. This is different from the local DNS attack.

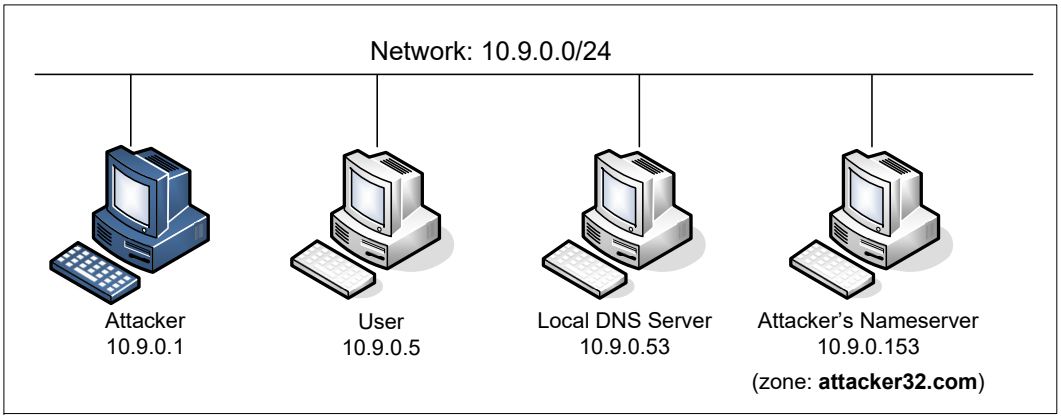

Figure 1: Environment setup for the experiment

2.1 Container Setup and Commands

Please download the `Labsetup.zip` file to your VM from the lab's website, unzip it, enter the `Labsetup` folder, and use the `docker-compose.yml` file to set up the lab environment. Detailed explanation of the content in this file and all the involved `Dockerfile` can be found from the user manual, which is linked to the website of this lab. If this is the first time you set up a SEED lab environment using containers, it is very important that you read the user manual.

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the `.bashrc` file (in our provided SEEDUbuntu 20.04 VM).

```
$ docker-compose build  # Build the container images
$ docker-compose up     # Start the containers
$ docker-compose down   # Shut down the containers

// Aliases for the Compose commands above
$ dcbuild       # Alias for: docker-compose build
$ dcup          # Alias for: docker-compose up
$ dcdown        # Alias for: docker-compose down
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the "`docker ps`" command to find out the ID of the container, and then use "`docker exec`" to start a shell on that container. We have created aliases for them in the `.bashrc` file.

```
$ dockps        // Alias for: docker ps --format "{{.ID}}  {{.Names}}"
$ docksh <id>   // Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#
```

```
// Note: If a docker command requires a container ID, you do not need to
//       type the entire ID string. Typing the first few characters will
//       be sufficient, as long as they are unique among all the containers.
```

If you encounter problems when setting up the lab environment, please read the "Common Problems" section of the manual for potential solutions.

2.2 About the Attacker Container

In this lab, we can either use the VM or the attacker container as the attacker machine. If you look at the Docker Compose file, you will see that the attacker container is configured differently from the other containers.

- *Shared folder.* When we use the attacker container to launch attacks, we need to put the attacking code inside the attacker container. Code editing is more convenient inside the VM than in containers, because we can use our favorite editors. In order for the VM and container to share files, we have created a shared folder between the VM and the container using the Docker `volumes`. If you look at the Docker Compose file, you will find out that we have added the following entry to some of the containers. It indicates mounting the `./volumes` folder on the host machine (i.e., the VM) to the `/volumes` folder inside the container. We will write our code in the `./volumes` folder (on the VM), so they can be used inside the containers.

  ```
  volumes:
          - ./volumes:/volumes
  ```

- *Host mode.* In this lab, the attacker needs to be able to sniff packets, but running sniffer programs inside a container has problems, because a container is effectively attached to a virtual switch, so it can only see its own traffic, and it is never going to see the packets among other containers. To solve this problem, we use the `host` mode for the attacker container. This allows the attacker container to see all the traffics. The following entry used on the attacker container:

  ```
  network_mode: host
  ```

  When a container is in the `host` mode, it sees all the host's network interfaces, and it even has the same IP addresses as the host. Basically, it is put in the same network namespace as the host VM. However, the container is still a separate machine, because its other namespaces are still different from the host.

2.3 Summary of the DNS Configuration

All the containers are already configured for this lab. We provide a summary here, so students are aware of these configurations. Detailed explanation of the configuration can be found from the manual.

Local DNS Server. We run the BIND 9 DNS server program on the local DNS server. BIND 9 gets its configuration from a file called `/etc/bind/named.conf`. This file is the primary configuration file, and it usually contains several `"include"` entries, i.e., the actual configurations are stored in those included files. One of the included files is called `/etc/bind/named.conf.options`. This is where the actual configuration is set.

- *Simplification.* DNS servers now randomize the source port number in their DNS queries; this makes the attacks much more difficult. Unfortunately, many DNS servers still use predictable source port number. For the sake of simplicity in this lab, we fix the source port number to `33333` in the configuration file.
- *Turning off DNSSEC.* DNSSEC is introduced to protect against spoofing attacks on DNS servers. To show how attacks work without this protection mechanism, we have turned off the protection in the configuration file.
- *DNS cache.* During the attack, we need to inspect the DNS cache on the local DNS server. The following two commands are related to DNS cache. The first command dumps the content of the cache to the file `/var/cache/bind/dump.db`, and the second command clears the cache.

```
# rndc dumpdb -cache    // Dump the cache to the specified file
# rndc flush            // Flush the DNS cache
```

- *Forwarding the `attacker32.com` zone.* A forward zone is added to the local DNS server, so if anybody queries the `attacker32.com` domain, the query will be forwarded to this domain's nameserver, which is hosted in the attacker container. The zone entry is put inside the `named.conf` file.

```
zone "attacker32.com" {
    type forward;
    forwarders {
        10.9.0.153;
    };
};
```

User machine. The user container `10.9.0.5` is already configured to use `10.9.0.53` as its local DNS server. This is achieved by changing the resolver configuration file (`/etc/resolv.conf`) of the user machine, so the server `10.9.0.53` is added as the first `nameserver` entry in the file, i.e., this server will be used as the primary DNS server.

Attacker's Nameserver. On the attacker's nameserver, we host two zones. One is the attacker's legitimate zone `attacker32.com`, and the other is the fake `example.com` zone. The zones are configured in `/etc/bind/named.conf`:

```
zone "attacker32.com" {
        type master;
        file "/etc/bind/attacker32.com.zone";
};

zone "example.com" {
        type master;
        file "/etc/bind/example.com.zone";
};
```

2.4 Testing the DNS Setup

From the User container, we will run a series of commands to ensure that our lab setup is correct. In your lab report, please document your testing results.

Get the IP address of `ns.attacker32.com`. When we run the following `dig` command, the local DNS server will forward the request to the Attacker nameserver due to the `forward` zone entry added to the local DNS server's configuration file. Therefore, the answer should come from the zone file (`attacker32.com.zone`) that we set up on the Attacker nameserver. If this is not what you get, your setup has issues. Please describe your observation in your lab report.

```
$ dig ns.attacker32.com
```

Get the IP address of `www.example.com`. Two nameservers are now hosting the `example.com` domain, one is the domain's official nameserver, and the other is the Attacker container. We will query these two nameservers and see what response we will get. Please run the following two commands (from the User machine), and describe your observation.

```
// Send the query to our local DNS server, which will send the query
// to example.com's official nameserver.
$ dig www.example.com

// Send the query directly to ns.attacker32.com
$ dig @ns.attacker32.com www.example.com
```

Obviously, nobody is going to ask `ns.attacker32.com` for the IP address of `www.example.com`; they will always ask the `example.com` domain's official nameserver for answers. The objective of the DNS cache poisoning attack is to get the victims to ask `ns.attacker32.com` for the IP address of `www.example.com`. Namely, if our attack is successful, if we just run the first `dig` command, the one without the `@` option, we should get the fake result from the attacker, instead of getting the authentic one from the domain's legitimate nameserver.

3 The Attack Tasks

The main objective of DNS attacks is to redirect the user to another machine B when the user tries to get to machine A using A's host name. For example, assuming `www.example.com` is an online banking site. When the user tries to access this site using the correct URL `www.example.com`, if the adversaries can redirect the user to a malicious web site that looks very much like `www.example.com`, the user might be fooled and give away his/her credentials to the attacker.

In this task, we use the domain name `www.example.com` as our attacking target. It should be noted that the `example.com` domain name is reserved for use in documentation, not for any real company. The authentic IP address of `www.example.com` is `93.184.216.34`, and its nameserver is managed by the Internet Corporation for Assigned Names and Numbers (ICANN). When the user runs the `dig` command on this name or types the name in the browser, the user's machine sends a DNS query to its local DNS server, which will eventually ask for the IP address from `example.com`'s nameserver.

The goal of the attack is to launch the DNS cache poisoning attack on the local DNS server, such that when the user runs the `dig` command to find out `www.example.com`'s IP address, the local DNS server will end up going to the attacker's nameserver `ns.attacker32.com` to get the IP address, so the IP

address returned can be any number that is decided by the attacker. As results, the user will be led to the attacker's web site, instead of to the authentic `www.example.com`.

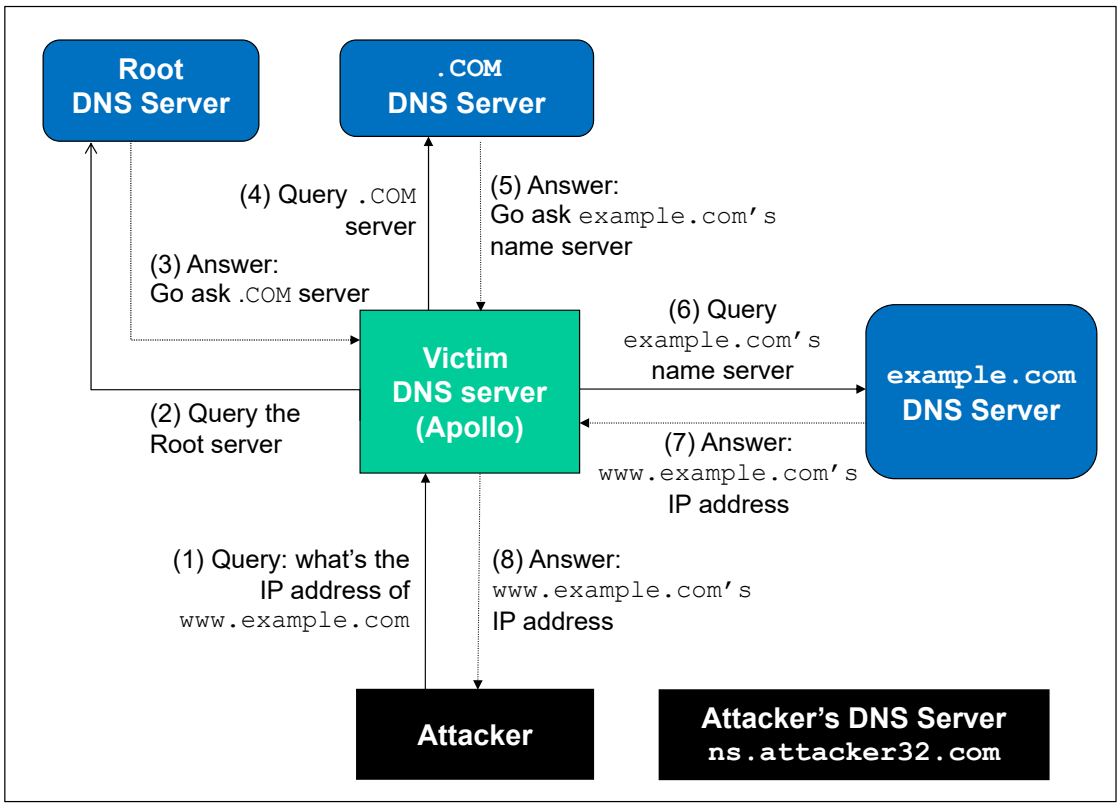

Figure 2: The complete DNS query process

3.1 How Kaminsky attack works

In this task, the attacker sends a DNS query request to the victim DNS server (`Apollo`), triggering a DNS query from `Apollo`. The query may go through one of the root DNS servers, the `.COM` DNS server, and the final result will come back from `example.com`'s DNS server. This is illustrated in Figure 2. In case that `example.com`'s nameserver information is already cached by `Apollo`, the query will not go through the root or the `.COM` server; this is illustrated in Figure 3. In this lab, the situation depicted in Figure 3 is more common, so we will use this figure as the basis to describe the attack mechanism.

While `Apollo` waits for the DNS reply from `example.com`'s name server, the attacker can send forged replies to `Apollo`, pretending that the replies are from `example.com`'s nameserver. If the forged replies arrive first, it will be accepted by `Apollo`. The attack will be successful.

If you have done our local DNS attack lab, you should realize that those attacks assume that the attacker and the DNS server are on the same LAN, i.e., the attacker can observe the DNS query message. When the attacker and the DNS server are not on the same LAN, the cache poisoning attack becomes more difficult. The difficulty is mainly caused by the fact that the transaction ID in the DNS response packet must match with that in the query packet. Because the transaction ID in the query is usually randomly generated, without seeing the query packet, it is not easy for the attacker to know the correct ID.

Obviously, the attacker can guess the transaction ID. Since the size of the ID is only 16 bits, if the attacker can forge K responses within the attack window (i.e. before the legitimate response arrives), the

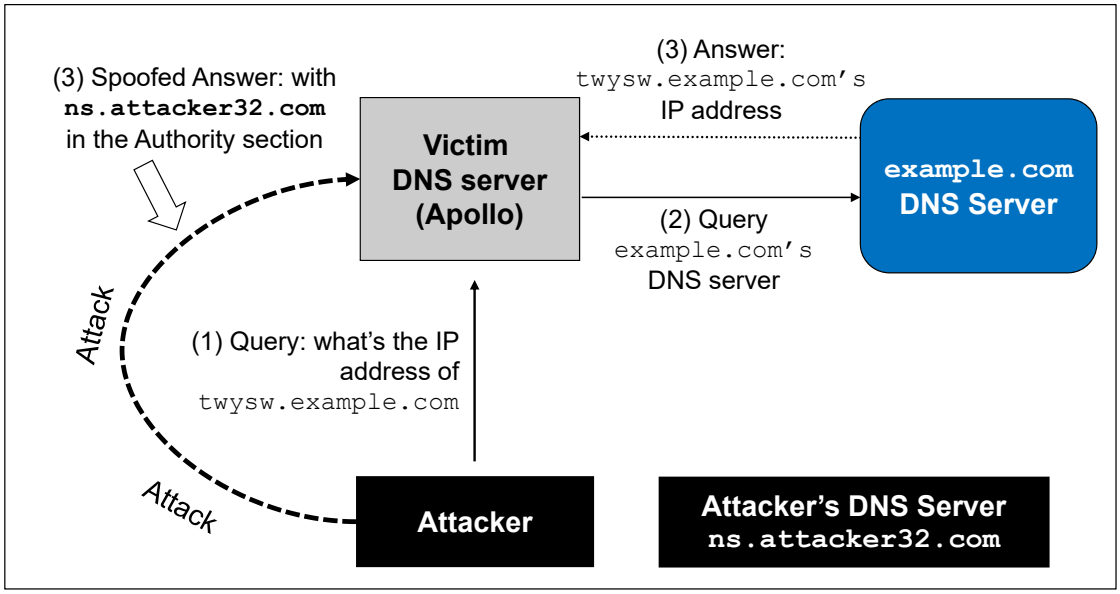

Figure 3: The Kaminsky Attack

probability of success is K over 2^{16}. Sending out hundreds of forged responses is not impractical, so it will not take too many tries before the attacker can succeed.

However, the above hypothetical attack has overlooked the cache effect. In reality, if the attacker is not fortunate enough to make a correct guess before the real response packet arrives, correct information will be cached by the DNS server for a while. This caching effect makes it impossible for the attacker to forge another response regarding the same name, because the DNS server will not send out another DNS query for this name before the cache times out. To forge another response on the same name, the attacker has to wait for another DNS query on this name, which means he/she has to wait for the cache to time out. The waiting period can be hours or days.

The Kaminsky Attack. Dan Kaminsky came up with an elegant technique to defeat the caching effect [2]. With the Kaminsky attack, attackers will be able to continuously attack a DNS server on a domain name, without the need for waiting, so attacks can succeed within a very short period of time. Details of the attacks are described in [1, 2]. In this task, we will try this attack method. The following steps with reference to Figure 3 outlines the attack.

1. The attacker queries the DNS Server `Apollo` for a non-existing name in `example.com`, such as `twysw.example.com`, where `twysw` is a random name.

2. Since the mapping is unavailable in `Apollo`'s DNS cache, `Apollo` sends a DNS query to the name-server of the `example.com` domain.

3. While `Apollo` waits for the reply, the attacker floods `Apollo` with a stream of spoofed DNS response, each trying a different transaction ID, hoping one is correct. In the response, not only does the attacker provide an IP resolution for `twysw.example.com`, the attacker also provides an "Authoritative Nameservers" record, indicating `ns.attacker32.com` as the nameserver for the `example.com` domain. If the spoofed response beats the actual responses and the transaction ID

matches with that in the query, Apollo will accept and cache the spoofed answer, and and thus Apollo's DNS cache is poisoned.

4. Even if the spoofed DNS response fails (e.g. the transaction ID does not match or it comes too late), it does not matter, because the next time, the attacker will query a different name, so Apollo has to send out another query, giving the attack another chance to do the spoofing attack. This effectively defeats the caching effect.

5. If the attack succeeds, in Apollo's DNS cache, the nameserver for example.com will be replaced by the attacker's nameserver ns.attacker32.com. To demonstrate the success of this attack, students need to show that such a record is in Apollo's DNS cache.

Task overview. Implementing the Kaminsky attack is quite challenging, so we break it down into several sub-tasks. In Task 2, we construct the DNS request for a random hostname in the example.com domain. In Task 3, we construct a spoofed DNS reply from example.com's nameserver. In Task 4, we put everything together to launch the Kaminsky attack. Finally in Task 5, we verify the impact of the attack.

3.2 Task 2: Construct DNS request

This task focuses on sending out DNS requests. In order to complete the attack, attackers need to trigger the target DNS server to send out DNS queries, so they have a chance to spoof DNS replies. Since attackers need to try many times before they can succeed, it is better to automate the process using a program.

Students need to write a program to send out DNS queries to the target DNS server (i.e., the local DNS server in our setup). Students' job is to write this program and demonstrate (using Wireshark) that their queries can trigger the target DNS server to send out corresponding DNS queries. The performance requirement for this task is not high, so students can use C or Python (using Scapy) to write this code. A Python code snippet is provided in the following (the +++'s are placeholders; students need to replace them with actual values):

```
Qdsec  = DNSQR(qname='www.example.com')
dns    = DNS(id=0xAAAA, qr=0, qdcount=1, ancount=0, nscount=0,
             arcount=0, qd=Qdsec)

ip  = IP(dst='+++', src='+++')
udp = UDP(dport=+++, sport=+++, chksum=0)
request = ip/udp/dns
```

3.3 Task 3: Spoof DNS Replies.

In this task, we need to spoof DNS replies in the Kaminsky attack. Since our target is example.com, we need to spoof the replies from this domain's nameserver. Students first need to find out the IP addresses of example.com's legitimate nameservers (it should be noted that there are multiple nameservers for this domain).

Students can use Scapy to implement this task. The following code snippet constructs a DNS response packet that includes a question section, an answer section, and an NS section. In the sample code, we use +++ as placeholders; students need to replace them with the correct values that are needed in the Kaminsky attack. Students need to explain why they pick those values.

```
name   = '+++'
```

```
domain = '+++'
ns     = '+++'

Qdsec  = DNSQR(qname=name)
Anssec = DNSRR(rrname=name,   type='A',  rdata='1.2.3.4', ttl=259200)
NSsec  = DNSRR(rrname=domain, type='NS', rdata=ns, ttl=259200)
dns    = DNS(id=0xAAAA, aa=1, rd=1, qr=1,
             qdcount=1, ancount=1, nscount=1, arcount=0,
             qd=Qdsec, an=Anssec, ns=NSsec)

ip    = IP(dst='+++', src='+++')
udp   = UDP(dport=+++, sport=+++, chksum=0)
reply = ip/udp/dns
```

Since this reply by itself will not be able to lead to a successful attack, to demonstrate this task, students need to use Wireshark to capture the spoofed DNS replies, and show that the spoofed packets are valid.

3.4 Task 4: Launch the Kaminsky Attack

Now we can put everything together to conduct the Kaminsky attack. In the attack, we need to send out many spoofed DNS replies, hoping one of them hits the correct transaction number and arrives sooner than the legitimate replies. Therefore, speed is essential: the more packets we can send out, the higher the success rate is. If we use Scapy to send the spoofed DNS replies like what we did in the previous task, the success rate is too low. Students can use C, but constructing DNS packets in C is non-trivial. We introduce a hybrid approach using both Scapy and C (see the SEED book for details).

With the hybrid approach, we first use Scapy to generate a DNS packet template, which is stored in a file. We then load this template into a C program, and make small changes to some of the fields, and then send out the packet. We have included a skeleton C code in `Labsetup/Files/attack.c`. Students can make changes in the marked areas. Detailed explanation of the code is given in the guideline section.

Check the DNS cache. To check whether the attack is successful or not, we need to check the `dump.db` file to see whether our spoofed DNS response has been successfully accepted by the DNS server. The following commands dump the DNS cache, and search whether the cache contains the word `attacker` (in our attack, we used `attacker32.com` as the attacker's domain; if students use a different domain name, they should search for a different word).

```
# rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
```

3.5 Task 5: Result Verification

If the attack is successful, in the local DNS server's DNS cache, the `NS` record for `example.com` will become `ns.attacker32.com`. When this server receives a DNS query for any hostname inside the `example.com` domain, it will send a query to `ns.attacker32.com`, instead of sending to the domain's legitimate nameserver.

To verify whether your attack is successful or not, go to the User machine, run the following two `dig` commands. In the responses, the IP addresses for `www.example.com` should be the same for both commands, and it should be whatever you have included in the zone file on the Attacker nameserver.

```
// Ask the local DNS server to do the query
$ dig www.example.com
```

```
// Directly query the attacker32 nameserver
$ dig @ns.attacker32.com www.example.com
```

Please include your observation (screenshots) in the lab report, and explain why you think your attack is successful. In particular, when you run the first `dig` commands, use Wireshark to capture the network traffic, and point out what packets are triggered by this `dig` command. Use the packet trace to prove that your attack is successful. Note that DNS results may be cached on the local DNS server after the first `dig` command is run. This could influence the results if you run the first `dig` command before using Wireshark. You can clear the cache using "`sudo rndc flush`" on the local DNS server, but that will require you to redo the attack.

4 Guidelines

To implement the Kaminsky attack, we can use Scapy to do the packet spoofing. Unfortunately, the speed of Python is too slow; the number of packets generated per second is too low to make the attack successful. It is better to use a C program. This could be quite challenging to many students, because constructing DNS packets using C is not very easy. I have developed a hybrid method, and have experimented with it in my own class. Using this approach, students' time spent on coding can be significantly reduced, so they can spend more time focusing on the actual attack.

The idea is to leverage the strength of both Scapy and C: Scapy is much more convenient in creating DNS packets than C, but C is much faster. Therefore we simply use Scapy to create the spoofed DNS packet, and save it to a file. We then load the packet into a C program. Even though we need to send a lot of different DNS packets during the Kaminsky attack, these packets are mostly the same, except for a few fields. Therefore, we can use the packet generated from Scapy as the basis, find the offsets where changes need to be made (e.g., the transaction ID field), and directly make changes. This will be much easier than creating the entire DNS packets in C. After the changes are made, we can use the raw socket to send out the packets. Details of such a hybrid method are provided in the Packet Sniffing and Spoofing chapter of the SEED book [1]. The following Scapy program creates a simple DNS reply packet, and saves it into a file.

Listing 1: `generate_dns_reply.py`

```
#!/usr/bin/env python3
from scapy.all import *

# Construct the DNS header and payload
name   = 'twysw.example.com'
Qdsec  = DNSQR(qname=name)
Anssec = DNSRR(rrname=name, type='A', rdata='1.1.2.2', ttl=259200)
dns    = DNS(id=0xAAAA, aa=1, rd=0, qr=1,
             qdcount=1, ancount=1, nscount=0, arcount=0,
             qd=Qdsec, an=Anssec)

# Construct the IP, UDP headers, and the entire packet
ip  = IP(dst='10.0.2.7', src='1.2.3.4', chksum=0)
udp = UDP(dport=33333, sport=53, chksum=0)
pkt = ip/udp/dns

# Save the packet to a file
with open('ip.bin', 'wb') as f:
  f.write(bytes(pkt))
```

In a C program, we load the packet from the file `ip.bin`, and use it as our packet template, based on which we create many similar packets, and flood the target local DNS servers with these spoofed replies. For each reply, we change three places: the transaction ID and the name `twysw` occurred in two places (the question section and the answer section). The transaction ID is at a fixed place (offset `28` from the beginning of our IP packet), but the offset for the name `twysw` depends on the length of the domain name. We can use a binary editor program, such as `bless`, to view the binary file `ip.bin` and find the two offsets of `twysw`. In our packet, they are at offsets `41` and `64`.

The following code snippet shows how we make change to these fields. We change the name in our reply to `bbbbb.example.com`, and then send out a spoofed DNS replies, with transaction ID being `1000`. In the code, the variable `ip` points to the beginning of the IP packet.

```
// Modify the name in the question field (offset=41)
memcpy(ip+41, "bbbbb" , 5);

// Modify the name in the answer field (offset=64)
memcpy(ip+64, "bbbbb" , 5);

// Modify the transaction ID field (offset=28)
unsigned short id = 1000;
unsigned short id_net_order = htons(id);
memcpy(ip+28, &id_net_order, 2);
```

Generate random names. In the Kaminsky attack, we need to generate random hostnames. There are many ways to do so. The following code snippet shows how to generate a random name consisting of 5 characters.

```
char a[26]="abcdefghijklmnopqrstuvwxyz";

// Generate a random name of length 5
char name[6];
name[5] = 0;
for (int k=0; k<5; k++)
   name[k] = a[rand() % 26];
```

Compile the program. To compile the program, we can use the following command:

```
$ gcc -o attack attack.c

// For Apple Silicon machines: use static binding
$ gcc -static -o attack attack.c
```

5 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.

References

[1] Wenliang Du. *Computer & Internet Security: A Hands-on Approach, 2nd Edition*. Self publishing, May 2019. ISBN: 978-1733003933. URL: `https://www.handsonsecurity.net`.

[2] D. Schneider. Fresh phish, how a recently discovered flaw in the internet's domain name system makes it easy for scammers to lure you to fake web sites. *IEEE Spectrum*, 2008. `http://spectrum.ieee.org/computing/software/fresh-phish`.