Kaminsky Attack Project

1 Project Overview

The objective of this project is for you to gain first-hand experience with an off-path 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 project focuses on a particular DNS attack technique, called *DNS Cache Poisoning attack*. In this off-path attack project, packet sniffing is not possible, so the attack becomes challenging. This lab covers the following topics:

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

Lab environment. This project runs on SEED Ubuntu 20.04 VM.

- Download and install latest Virtual Box using this link
- Download NS Project.ova using this link
- Double click on NS Project.ova will install machine on your Virtual Box which is a SEED Ubuntu-20.04 VM (username: seed password: dees) "sudo" privileges is given to seed user as in case if you want to run any program as root.

2. Project Environment Setup (Task 1)

The DNS cache poisoning attack mainly targets a local DNS server. We set up our own DNS server to conduct the attack experiments. The project environment needs four separate machines: one for the victim, one for the DNS server, and two for the attacker. The project environment setup is illustrated in Figure 1.

We put all these machines on the same LAN only for the sake of simplicity.

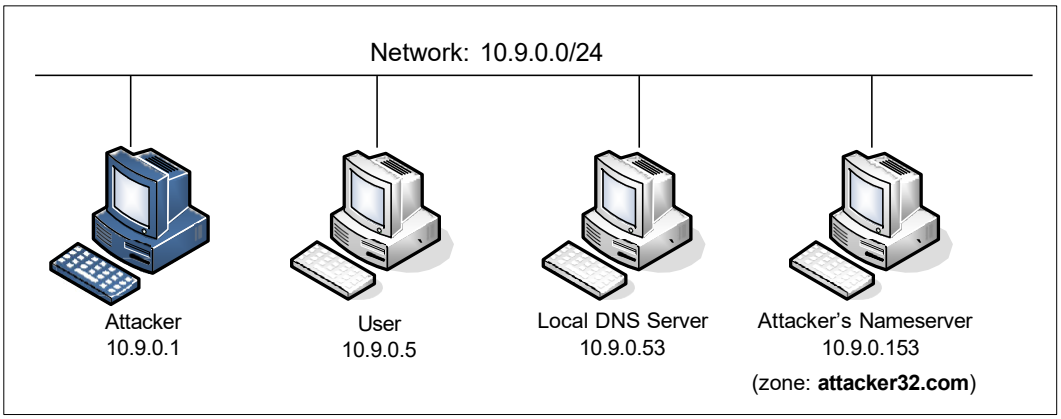

Figure 1: Environment setup for the experiment

1.1 Container Setup and Commands

Please download the ProjectSetup.zip file from the course's Teams channel, unzip it, enter the ProjectSetup folder, and use the docker-compose.yml file to set up the project environment. Detailed explanation of the content in this file and all the involved Dockerfile can be found from the user manual in this link. 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 frequently, we have created aliases for them in the .bashrc file (in the provided SEEDUbuntu 20.04 VM).

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

// 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.

1.2 About the Attacker Container

In this Project, 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 inside 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 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 it can be used inside the containers.

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

- *Host mode.* In this Project, 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 is 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.

1.3 Summary of the DNS Configuration

All the containers are already configured for this project. We provide a summary here. A 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 project, 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 following 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";
};
```

1.4 Testing the DNS Setup

From the User container, we will run a series of commands to ensure that our project setup is correct. In your project 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 contact TA for help). Please describe your observation in your project 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 responses 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, i.e., 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.

2 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 website 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.

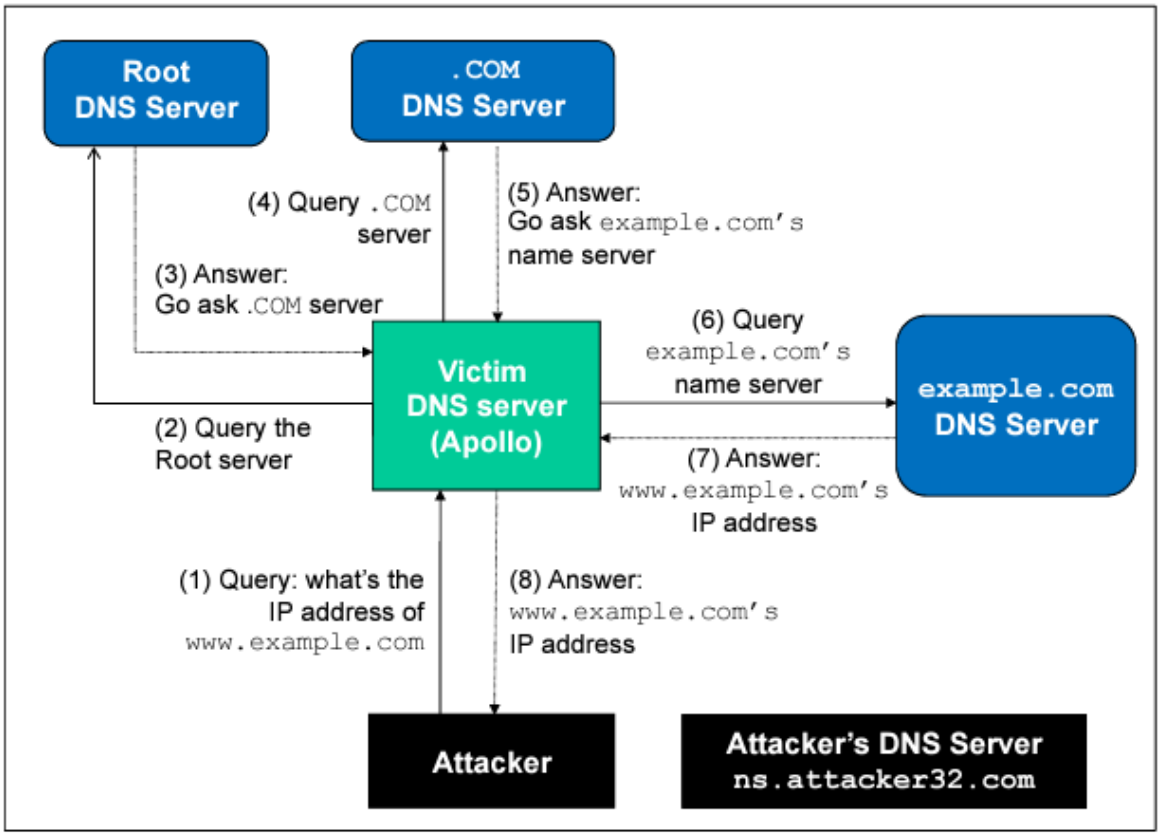

Figure 2: The complete DNS query process

2.1 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 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 project, 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, they will be accepted by Apollo. The attack will be successful.

When the attacker and the DNS server are not on the same LAN, which is the case for this project, the cache poisoning attack becomes difficult. The difficulty is mainly because the transaction ID in the DNS response packet must match that in the query packet. Because the transaction ID in the query is usually randomly generated, without sniffing 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 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.

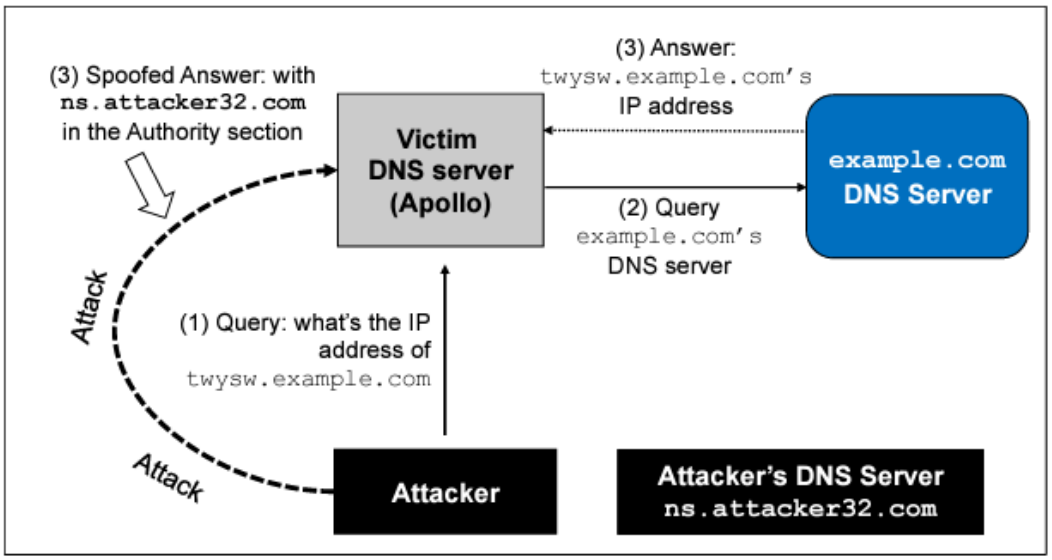

Figure 3: The Kaminsky Attack

However, the above hypothetical attack has overlooked the cache effect. In reality, if the attacker is not lucky enough to make a correct guess before the real response packet arrives, the 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 they have 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 [1]. 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] In this task, we will carry out 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 nameserver 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 that 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 that in the query, Apollo will accept and cache the spoofed answer. 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, you 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.

2.2 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.

You need to write a program to send out DNS queries to the target DNS server (i.e., the local DNS server in our setup). Your 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 you can use C or Python (using Scapy) to write this code. A Python code snippet is provided in the following (the +++'s are placeholders; you need to replace them with actual values):

```
from scapy.all import *

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
send(request)
```

2.3 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. You 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).

You 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; you need to replace them with the correct values that are needed in the Kaminsky attack. You need to explain why you pick those values.

```
from scapy.all import *
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
send(reply)
```

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

2.4 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 that 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 use a hybrid approach using both Scapy and C.

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, 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 use `attacker32.com` as the attacker's domain; if you use a different domain name, you should search for a different word).

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

2.5 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 observations (screenshots) in the project 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

3 Guidelines

To implement the Kaminsky attack, we can use Scapy to spoof packets. Unfortunately, the speed of Python is too slow; the number of packets generated per second is too low to make the attack successful. We will instead use a hybrid method consisting both Scapy and C.

In the hybrid approach, we 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 a raw socket to send out the packets. The following Scapy program creates a simple DNS reply packet, and saves it into a file.

```
#!/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))
```

Listing 1: generate dns reply.py

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 changes to these fields. We change the name in our reply to bbbbb.example.com, and then send out a spoofed DNS reply, 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];
```

4 Submission

You need to submit a detailed project 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.