ECE568 – Computer Security

Lab #4: DNS Cache Poisoning

Setup

For this lab, we will be using the patched DNS server program called BIND (Berkeley Internet Name Domain). Start by downloading and extracting the lab4.tar.gz file from the course website:

```
tar zxvf lab4.tar.gz
```

We next describe how to setup and configure BIND on your ECF machine.

Step 1: Configuration Files

There are two configuration files you need to set for this lab. They are `rndc.conf` and `named.conf`, located in the ./`etc` directory that you extracted. RNDC is short for Remote Name Daemon Control, which is useful for dumping BIND's server cache to check whether our attack has been successful or not. You need to specify few parameters in these files.

BIND listens on specific port for communication with `rndc`, which allows command-line administration. We will be using `rndc` to flush and dump BIND's cache. Since ECF workstations can have multiple users, you need to randomly pick two different port numbers (so that your processes doesn't collide with other students):

- `query-source port` parameter specifies the port number BIND will use to send its outgoing queries to external DNS servers.
- `listen-on port` parameter specifies the port number BIND will use to listen for DNS queries.

Note: you must pick port numbers **greater** than 1024 (as ports lower than that are reserved for processes running as the root user). One you have chosen a port number for your own copy of `rndc`, modify the bold-faced parameters in `rndc.conf` and `named.conf`:

`rndc.conf`

```
options {
        default-key "rndc-key";
        default-server 127.0.0.1;
        default-port <RNDC port number>;
};
```

`named.conf`

```
options {
        ...
        query-source port <query port>;
        ...
        listen-on-port <NAMED port number> { any; };
```

```
        ...
};

controls {
        inet 127.0.0.1 port <RNDC port number>
                allow { 127.0.0.1; } keys { "rndc-key"; };
};
```

Step 2: Start the DNS server

You can run BIND by running the script provided in the lab code:

```
./run_bind.sh
```

Step 3: Install Scapy

Scapy is a powerful tool for packet manipulation. To install Python packages locally, we will be using pip. Do the following to install scapy locally:

```
pip2 install scapy==2.3.3 --user
```

Take care to ensure that the version of scapy being installed is as shown above ie. 2.3.3. You may encounter warnings about `SNIMissingWarning` and `InsecurePlatformWarning`. You can safely ignore them.

Check if scapy is installed correctly:

```
$ python2
Python 2.7.18 (default, Oct 11 2021, 11:39:27)
[GCC 8.5.0 20210514 (Red Hat 8.5.0-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import scapy
>>> exit()
```

No output and the absence of an import error (ignore any warnings) indicates that we can proceed further.

When using scapy, you may get a lot of following errors:

```
        Exception RuntimeError: 'maximum recursion depth exceeded while calling a
Python object' in <type 'exceptions.AttributeError'> ignored
```

This error should not affect your lab and you can safely ignore them.
Further information on how to use scapy could be found at: https://scapy.readthedocs.io/en/latest/

Resources

In this lab, you will be performing a DNS Cache Poisoning attack as described in Lecture 19. More details on the attack can be found here:

- An Illustrated Guide to the Kaminsky DNS Vulnerability
- Analysis of the Cache Poisoning attack

Part 1: Getting familiar with dig

`dig` is a useful tool for sending DNS queries. Unless otherwise specified, `dig` will by default, try each of servers listed in `/etc/resolv.conf`.
Direct `dig` to the default DNS server on the ECF machines (not the BIND server) and figure out the following:

1. What is the `IPv4` address of `utoronto.ca`?
2. What are the names of name servers of `utoronto.ca` and their `IPv4` addresses?
3. What are the names of mail servers of `utoronto.ca` and their `IPv4` addresses?
4. Now direct `dig` to your local BIND server and repeat the above queries. Check to see if the output matches your results from 1-3.

Refer to the manual page (run "man dig") to learn how to call `dig` with the appropriate options in order to point it to the BIND server. For parts 1-3, create a file called `part1.txt` and write to it in the following format

1. <ipv4 address of utoronto.ca>
2. <name of name server #1>:<ipv4 address of name server #1>
2. <name of name server #2>:<ipv4 address of name server #2>
2 . ….
3. <name of mail server #1>:<ipv4 address of mail server #1>
3. ….

Note - there may be more than 1 name/mail server for `utoronto.ca`

Part 2: Write a DNS proxy that sits between dig and local DNS server

If we can inspect DNS queries and their replies, it is possible to forge them to attack users. Imagine redirecting a DNS request of `google.com` to a malicious server.

Due to security concerns in ECF, packet sniffing is disabled. However, we can simulate it by forcing DNS queries to go through proxies. Your goal is to write a proxy server that accepts DNS queries from `dig` and forwards them to the BIND server we setup earlier. It should also receive a DNS reply from the BIND server and forward it back to `dig` (unmodified).

Refer to the manual page to learn how to call `dig` with the appropriate options in order to point it to the proxy instead of the BIND server. Check to see if you receive the same output as when you point `dig` directly to the BIND server.

We have provided some starter code that can be used to build the proxy for Parts 2 and 3 in `dnsproxy_starter.py`

Your proxy should be run as:
`python2 dnsproxy_starter.py --port` **`<NAMED port number>`** `--dns_port` **`<query port>`**

Part 3: Spoof DNS reply using the DNS proxy

Your goal for this exercise is to use the proxy created in Part 2 to intercept and forge DNS replies. Look up `example.com` by sending:

```
dig  <options> example.com
```

Spoof the DNS reply such that `example.com`'s `IPv4` address is `1.2.3.4` instead and change its name servers to `ns.dnslabattacker.net`. We suggest using scapy to manipulate DNS packets. Information about DNS packet format can be found in the "DNS Primer" notes, included in the Lab materials (with full credit to Duke University, for a good, concise write-up of the protocol!).

Your proxy should be run as:

```
python2 dnsproxy_starter.py --port <NAMED port number> --dns_port <query port>
--spoof_response
```

Part 4: DNS cache poisoning attack

We will be implementing a DNS cache poisoning attack, also known as the Kaminsky attack.

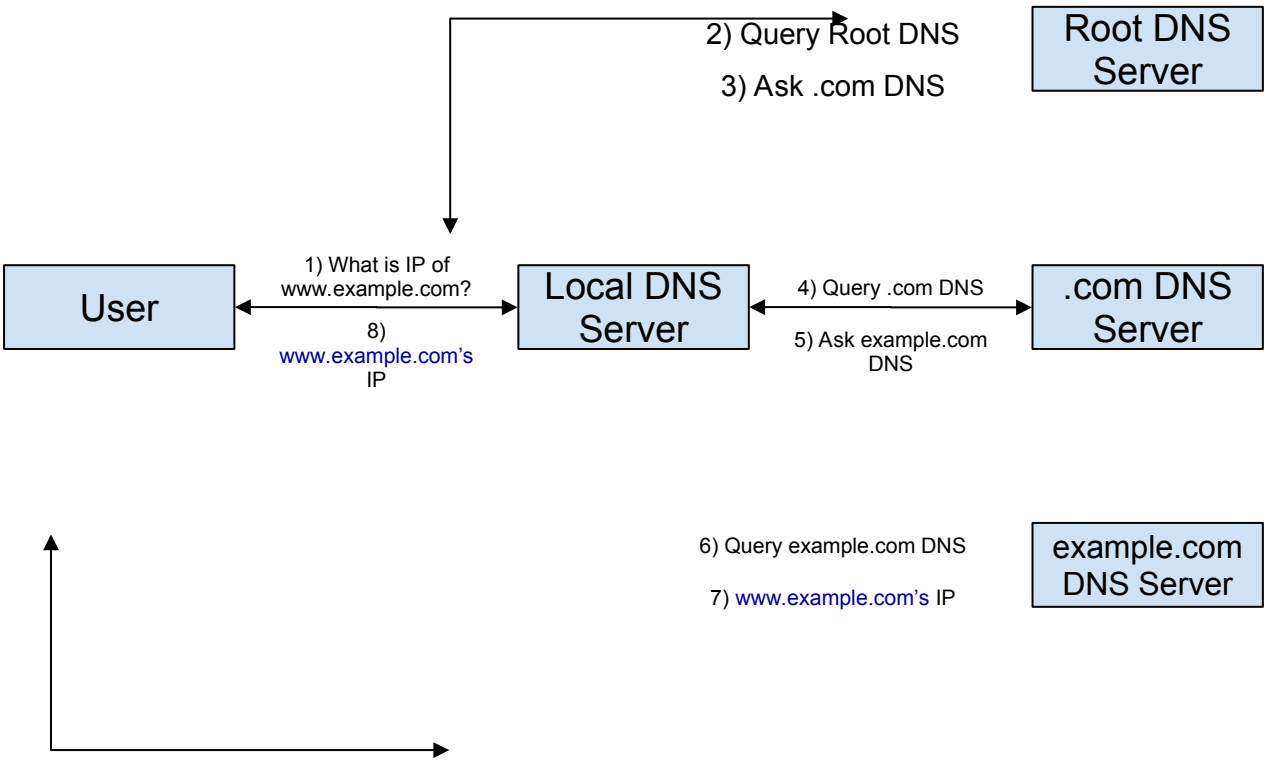

Figure 1: DNS query process

Figure 1 illustrates a complete DNS query process when a user submits a DNS query for www.example.com's IP address. Aside from returning www.example.com's IP address, the local DNS server (i.e. BIND) will also cache example.com's name server address. Therefore, when the

user sends a DNS query for another address in example.com domain (e.g. mail.example.com), the local BIND server will directly ask for its IP from the cached example.com's name server, illustrated in Figure 2.

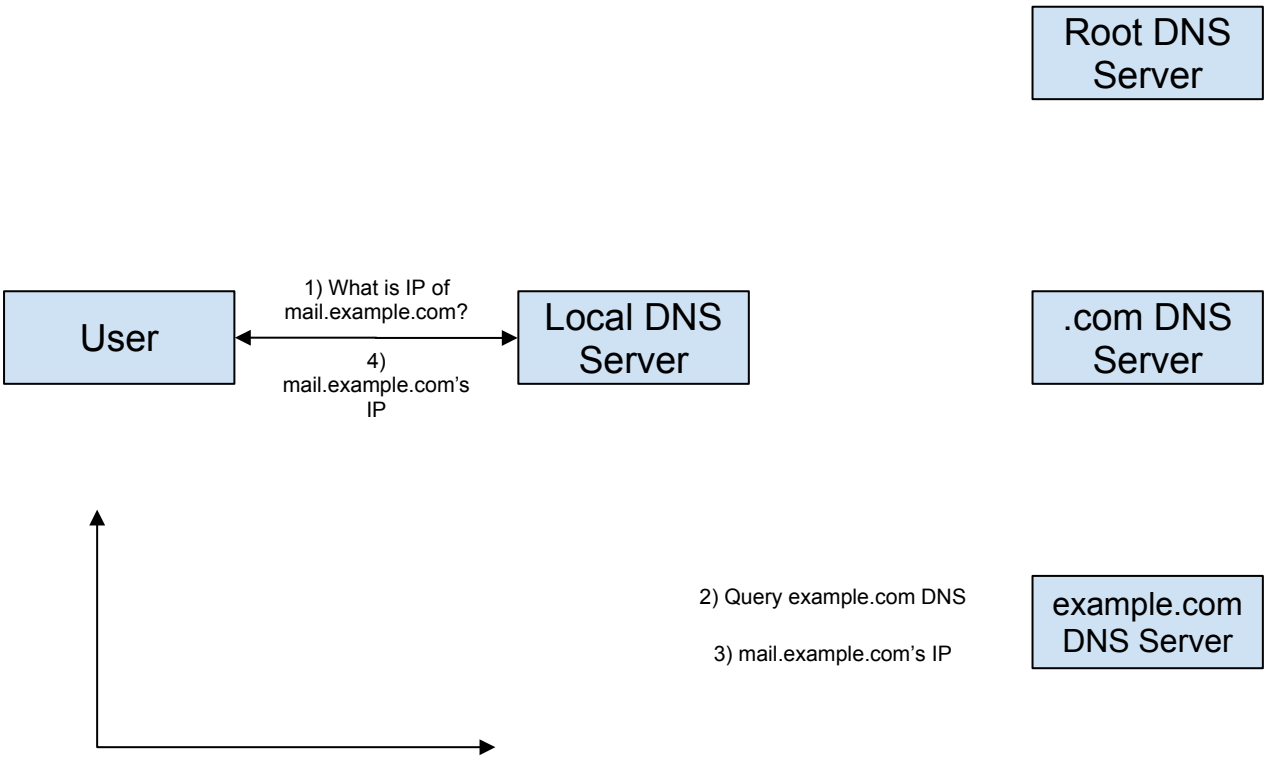

Figure 2: DNS query process when example.com DNS Server is cached

Imagine what would happen if an attacker forges DNS reply from example.com name server before the actual reply (i.e. message #3) reaches the local BIND server. The attacker can return arbitrary IP address of mail.example.com and furthermore overwrite example.com name server address that is cached at the local BIND server to a new fake address! (e.g. `ns.dnslabattacker.net`)

However, this is more difficult than it sounds because DNS replies include a transaction ID which must match with the ID from the corresponding DNS query. Transaction IDs for DNS queries are randomly generated and not visible to attackers. Furthermore, sending DNS queries to resolve the same domain name again will not be effective because the first result will be cached.For example, if you send DNS query for www.example.com twice, the 2nd DNS query will not trigger DNS query to example.com name server. This makes it impossible for the attacker to forge another response for the same domain name until the cache entry expires and thus, makes the attack impractical.

Attack Procedure:

Here are outline for the attack procedure, to successfully poison your BIND server:

1. Query the BIND server for a non-existing name in `example.com`, such as `twysw.example.com`, where `twysw` is a random name.
    a. Since the mapping is not available in the BIND server's cache, it will send out a DNS query to the name server of the `example.com` domain.

2. While the BIND server waits for the reply from example.com's name server, flood it with a stream of spoofed DNS replies, each with a different transaction ID, hoping one is correct.
    a. Even if the spoofed DNS reply fails, it does not matter because the next time, the attacker will query a different name.
3. Repeat steps 1-2 until the attack succeeds!

Goal:

Spoof DNS replies to overwrite example.com's name server's address to `ns.dnslabattacker.net`. You need to add fake `NS` (i.e. name server) record in your spoofed DNS reply to overwrite example.com's name server address. Use starter code part4_starter.py.

You can run it with:

```
./part4_starter.py --ip <bind ip> --port <NAMED port number> --query_port <query port>
```

You can figure out your workstation's ip with following command:

```
dig @resolver1.opendns.com +short myip.opendns.com
```

Hints:

1. BIND is patched such that you do not need to fake IP address of the spoofed packet to match with the IP address of the remote name server. This is required in actual attack scenario; however, spoofing IP address requires sudo privilege which is disabled in ECF lab.
2. scapy's sending functions such as: `sr1()`, `send()` will not work in ECF environment. These functions require sudo privileges which is disabled in ECF lab. Instead, you can use scapy to: a) build packets, b) modify packets. For sending and receiving packets, use Python's socket library. Example for using Python's socket library + scapy for building DNS packet is in `part4_starter.py`.
3. You can send a command through RNDC to enable logging queries received by your BIND server:

    ```
    ./bin/rndc –c etc/rndc.conf querylog
    ```

Attack Verification:

There are 2 ways to verify that your attack was successful:

Verification 1:

Using `dig`, send a DNS query to your BIND, asking for the name server of example.com. It should show that the name server of example.com is `ns.dbslabattacker.net`:

```
% ~/remote_dns_cache_poisoning]$ dig @127.0.0.1 NS example.com –p 1053

; <<>> DiG 9.8.2rc1-RedHat-9.8.2-0.68.rc1.el6_10.1 <<>> @127.0.0.1 NS example.com –p 1053
; (1 server found)
;; global options: +cmd
;; Got answer:
```

```
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 18490
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;example.com.                   IN    NS

;; ANSWER SECTION:
example.com.            3578   IN    NS    ns.dnslabattacker.net.

;; Query time: 7 msec
;; SERVER: 127.0.0.1#1053(127.0.0.1)
;; WHEN: Wed Nov 14 16:22:04 2018
;; MSG SIZE  rcvd: 64
```

Verification 2:

Inspect the BIND server's cache to determine if it is poisoned:

```
./bin/rndc -c etc/rndc.conf dumpdb -cache
less /tmp/dump.db // should be your cache dump location
```

A successful attack output should contain following:

```
example.com.            608390  NS      ns.dnslabattacker.net.
; additional
                        691171  DS      31406 8 1 (
                                        189968811E6EBA862DD6C209F75623D8D9ED
                                        9142 )
                        691171  DS      31406 8 2 (
                                        F78CF3344F72137235098ECBBD08947C2C90
                                        01C7F6A085A17F518B5D8F6B916D )
                        691171  DS      31589 8 1 (
                                        3490A6806D47F17A34C29E2CE80E8A999FFB
                                        E4BE )
                        691171  DS      31589 8 2 (
                                        CDE0D742D6998AA554A92D890F8184C698CF
                                        AC8A26FA59875A990C03E576343C )
                        691171  DS      43547 8 1 (
                                        B6225AB2CC613E0DCA7962BDC2342EA4F1B5
                                        6083 )
                        691171  DS      43547 8 2 (
                                        615A64233543F66F44D68933625B17497C89
                                        A70E858ED76A2145997EDF96A918 )
; additional
                        691171  RRSIG   DS 8 2 86400 (
                                        20181115052318 20181108041318 37490 com.
                                        jDj5TuwsRf2RyghnDvNsAKk116A5HqKyUBJh
                                        qCYowGVTGp5mcAaHfO1QlfZrYVRiwuS9aHU2
                                        zsHgA9A68UGG3RP3lJHNloJ/WMfaOnsZEM5C
                                        QVtiAupk5uMWidzWa8i7kIFcO9V9H96yt1zW
                                        N6ME8xBKX4CfCsLH4zJLgPLW7sg= )
```

Submission Instruction

You may work on this lab individually or in groups of two. Your code must be entirely your own original work. The assignment is due 11:59:59pm on Sunday, April 9th. Please submit a README file that contains your name(s), student number(s), and email(s) at the top, along with explanations for each part. You should submit `part1.txt`, `dnsproxy_starter.py` (for part2 and 3) and `part4_starter.py`.

```
submitece568s 4 README part1.txt dnsproxy_starter.py part4_starter.py
```

README format:

```
#first1 last1, studentnum1, e-mail1
#first2 last2, studentnum2, e-mail2

Part 1 Explanation:
...
```

Note: Please put your explanations in the README, not in the part#.txt files. If your files are not formatted as instructed, you may lose marks.

We have provided a submission checking script to help ensure that the automarker will process your files correctly:

```
/share/copy/ece568f/bin/check568_lab4 <directory of files>
```

Your solutions will be tested on the ECF machines. Please ensure that you test your solutions on these machines prior to submission.