INFA 723 - Cryptography Conway, Kiera

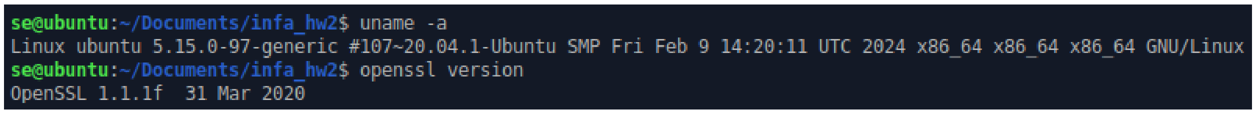
Points	96 / 100
Weight Achieved	9.6 / 10
see 4.7	

Homework 2

Track A

LAB 1 - Set up OpenSSL Testing Environment

If you already have an OpenSSL testing environment on your computer, you do not need to do Lab1. Go to your Linux terminal, and type the command `$uname -a` and copy the output to a file. Include the output as part of your homework assignment.



```
se@ubuntu:~/Documents/infa_hw2$ uname -a
Linux ubuntu 5.15.0-97-generic #107~20.04.1-Ubuntu SMP Fri Feb 9 14:20:11 UTC 2024 x86_64 x86_64 x86_64 GNU/Linux
se@ubuntu:~/Documents/infa_hw2$ openssl version
OpenSSL 1.1.1f  31 Mar 2020
```

LAB 2 - Use OpenSSL Command Line

1. (5pts) What is Heartbleed bug? How do you find out if your OpenSSL library is vulnerable to the Hearbleed bug? How to fix the vulnerability?

(Hint: find more information about Heartbleed bug at http://heartbleed.com/)

The Heartbleed bug (CVE-2014-0160) is a serious vulnerability in the OpenSSL library. SSL is a cryptographic protocol used to secure internet communications by establishing encrypted connections between a client and a server to protect data during transmission. One functionality of SSL was the ‘heartbeat extension’, which allowed a client to send a 'heartbeat request' to a server to check if it is still responsive and receive a ‘heartbeat response’ from the server. The vulnerability of this extension was caused by a flaw in the way the server handled the heartbeat requests. When a client sent a heartbeat request, it included a length field that indicated how much data it was expecting in the response. However, this length field was not properly validated by the server. Therefore, an attacker could send a maliciously crafted heartbeat request with a length field that claimed to be larger than the actual data sent by the client, and since the server did not properly validate this length field, it would respond by sending back data from its memory. This data could include sensitive information such as encryption keys, user credentials, and other confidential data.

Since this vulnerability was only present on OpenSSL versions 1.0.1 through 1.0.1f, the best way to determine if an OpenSSL library is vulnerable to the Heartbleed bug is by checking its version. On Linux machines, this can be accomplished with the following command:

```
$openssl version -a
```

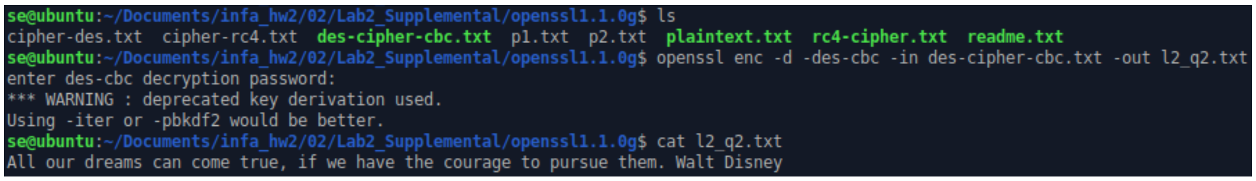
If the returned version falls within the vulnerable range, the optimal course of action is to upgrade to a patched version. If upgrading is not a possibility, developers can mitigate the risk by removing the handshake extension from the code by using the following compile option:

```
-DOPENSSL_NO_HEARTBEATS
```

However, it is advisable to prioritize upgrading to a patched version as soon as possible. [1]

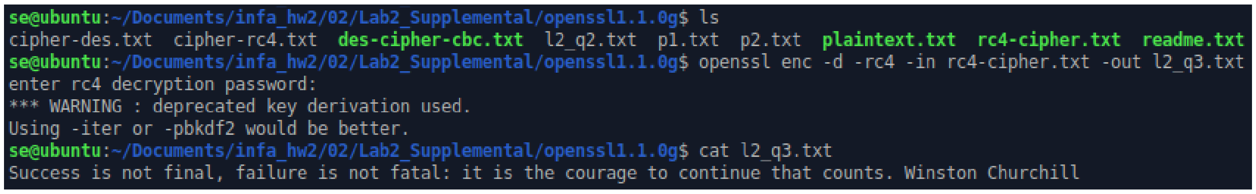
2. (5pts) In Lab2 folder, des-cipher-cbc.txt is encrypted using DES in CBC mode. The password used for encryption is password. Decrypt the file and enclose the plaintext in your solution document.

All our dreams can come true, if we have the courage to pursue them. Walt Disney


```
se@ubuntu:~/Documents/infa_hw2/02/Lab2_Supplemental/openssl1.1.0g$ ls
cipher-des.txt  cipher-rc4.txt  des-cipher-cbc.txt  p1.txt  p2.txt  plaintext.txt  rc4-cipher.txt  readme.txt
se@ubuntu:~/Documents/infa_hw2/02/Lab2_Supplemental/openssl1.1.0g$ openssl enc -d -des-cbc -in des-cipher-cbc.txt -out l2_q2.txt
enter des-cbc decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
se@ubuntu:~/Documents/infa_hw2/02/Lab2_Supplemental/openssl1.1.0g$ cat l2_q2.txt
All our dreams can come true, if we have the courage to pursue them. Walt Disney
```

3. (5pts) In Lab2 folder, rc4-cipher.txt is encrypted using RC4 algorithm. The password used for encryption is madison. Decrypt the file and enclose the plaintext in your solution document.

Success is not final, failure is not fatal: it is the courage to continue that counts. Winston Churchill


```
se@ubuntu:~/Documents/infa_hw2/02/Lab2_Supplemental/openssl1.1.0g$ ls
cipher-des.txt  cipher-rc4.txt  des-cipher-cbc.txt  l2_q2.txt  p1.txt  p2.txt  plaintext.txt  rc4-cipher.txt  readme.txt
se@ubuntu:~/Documents/infa_hw2/02/Lab2_Supplemental/openssl1.1.0g$ openssl enc -d -rc4 -in rc4-cipher.txt -out l2_q3.txt
enter rc4 decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
se@ubuntu:~/Documents/infa_hw2/02/Lab2_Supplemental/openssl1.1.0g$ cat l2_q3.txt
Success is not final, failure is not fatal: it is the courage to continue that counts. Winston Churchill
```

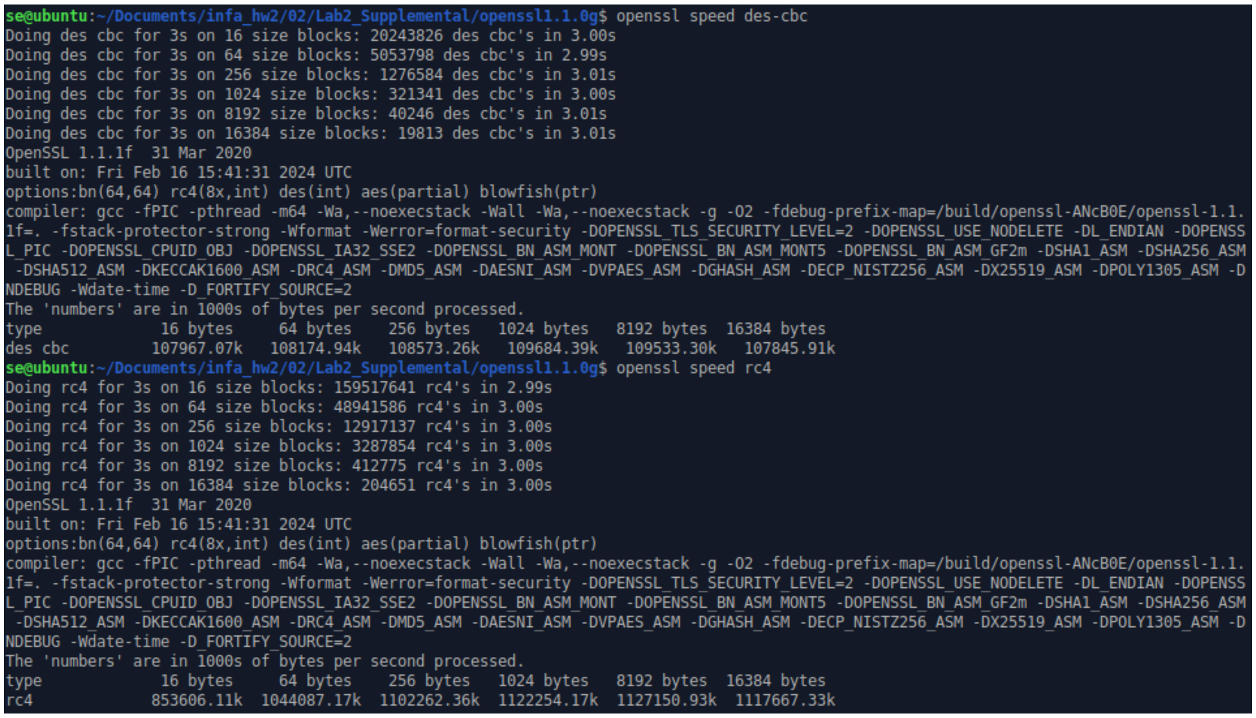
4. (10pts) In Lab2, you benchmarked DES CBC and RC4 ciphers. For 64 bits blocks, what's the encryption/decryption throughput of DES CBC and RC4 ciphers in your testing environment? Which cipher is faster? Why?

Based on benchmark results, RC4 demonstrates a higher throughput compared to DES CBC across all block sizes, including 64-bit blocks. In fact, even for the smallest block size of 16 bytes, RC4 processes approximately 8 times more data per second than DES CBC. This disparity in performance stems from the distinctive design approaches of the two cipher; RC4 functions as a stream cipher while DES CBC operates as a block cipher. This means that RC4 processes data bit-by-bit without the need for block alignment. For example, block ciphers such as DEC CBC require data to be divided into fixed-size blocks prior to encryption/decryption; this may require overhead such as padding and alignment adjustments. To do this, the entire data block must be available to the block cipher prior to encryption/decryption, versus stream ciphers which operate in real time.

Also, block ciphers encrypt in 'rounds', where during each encryption round, several operations are performed on an input block to transform it into an encrypted output block. Mnay of these operations are complex and include things like substitution, permutation, and mixing of data with an encryption key. In DES, each block undergoes 16 rounds of these operations, which can quickly become resource-intensive. In contrast, stream ciphers use efficient and simple XOR (exclusive OR) operations to encrypt and decrypt data. As a result, stream ciphers such as RC4 exhibit a higher throughput and faster processing speeds than block ciphers like DES CBC.

Table 1 - Throughput for 64 bytes

Algorithm	*Throughput (bytes per second)*
DES CBC	108174.94 k
RC4	1044087.17 k



```
se@ubuntu:~/Documents/infa_hw2/02/Lab2_Supplemental/openssl1.1.0g$ openssl speed des-cbc
Doing des cbc for 3s on 16 size blocks: 20243826 des cbc's in 3.00s
Doing des cbc for 3s on 64 size blocks: 5053798 des cbc's in 2.99s
Doing des cbc for 3s on 256 size blocks: 1276584 des cbc's in 3.01s
Doing des cbc for 3s on 1024 size blocks: 321341 des cbc's in 3.00s
Doing des cbc for 3s on 8192 size blocks: 40246 des cbc's in 3.01s
Doing des cbc for 3s on 16384 size blocks: 19813 des cbc's in 3.01s
OpenSSL 1.1.1f  31 Mar 2020
built on: Fri Feb 16 15:41:31 2024 UTC
options:bn(64,64) rc4(8x,int) des(int) aes(partial) blowfish(ptr)
compiler: gcc -fPIC -pthread -m64 -Wa,--noexecstack -Wall -Wa,--noexecstack -g -O2 -fdebug-prefix-map=/build/openssl-ANcB0E/openssl-1.1.
1f=. -fstack-protector-strong -Wformat -Werror=format-security -DOPENSSL_TLS_SECURITY_LEVEL=2 -DOPENSSL_USE_NODELETE -DL_ENDIAN -DOPENSS
L_PIC -DOPENSSL_CPUID_OBJ -DOPENSSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_MONT5 -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_ASM
 -DSHA512_ASM -DKECCAK1600_ASM -DRC4_ASM -DMD5_ASM -DAESNI_ASM -DVPAES_ASM -DGHASH_ASM -DECP_NISTZ256_ASM -DX25519_ASM -DPOLY1305_ASM -D
NDEBUG -Wdate-time -D_FORTIFY_SOURCE=2
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes  16384 bytes
des cbc         107967.07k   108174.94k   108573.26k   109684.39k   109533.30k   107845.91k
se@ubuntu:~/Documents/infa_hw2/02/Lab2_Supplemental/openssl1.1.0g$ openssl speed rc4
Doing rc4 for 3s on 16 size blocks: 159517641 rc4's in 2.99s
Doing rc4 for 3s on 64 size blocks: 48941586 rc4's in 3.00s
Doing rc4 for 3s on 256 size blocks: 12917137 rc4's in 3.00s
Doing rc4 for 3s on 1024 size blocks: 3287854 rc4's in 3.00s
Doing rc4 for 3s on 8192 size blocks: 412775 rc4's in 3.00s
Doing rc4 for 3s on 16384 size blocks: 204651 rc4's in 3.00s
OpenSSL 1.1.1f  31 Mar 2020
built on: Fri Feb 16 15:41:31 2024 UTC
options:bn(64,64) rc4(8x,int) des(int) aes(partial) blowfish(ptr)
compiler: gcc -fPIC -pthread -m64 -Wa,--noexecstack -Wall -Wa,--noexecstack -g -O2 -fdebug-prefix-map=/build/openssl-ANcB0E/openssl-1.1.
1f=. -fstack-protector-strong -Wformat -Werror=format-security -DOPENSSL_TLS_SECURITY_LEVEL=2 -DOPENSSL_USE_NODELETE -DL_ENDIAN -DOPENSS
L_PIC -DOPENSSL_CPUID_OBJ -DOPENSSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_MONT5 -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_ASM
 -DSHA512_ASM -DKECCAK1600_ASM -DRC4_ASM -DMD5_ASM -DAESNI_ASM -DVPAES_ASM -DGHASH_ASM -DECP_NISTZ256_ASM -DX25519_ASM -DPOLY1305_ASM -D
NDEBUG -Wdate-time -D_FORTIFY_SOURCE=2
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes  16384 bytes
rc4             853606.11k  1044087.17k  1102262.36k  1122254.17k  1127150.93k  1117667.33k
```

LAB 3 - SEND AND RECEIVE A FILE THROUGH A SOCKET CONNECTION

1. (5 pts) Check the source code, client.c and server.c. Describe the differences of the implementation on the client side and the server side.

For this socket communication, `client.c` handles the client-side connection process. It initiates a connection to the user-specified host and port, or defaults to port 3456 if none is provided. Then it establishes a connection to the designated port and IP using the `connect()` function, where it subsequently reads the user-specified plaintext file. By parsing and printing the file 8 characters at a time, it is able to send the contents to the server in chunks until reaching the end of the file (EOF). Upon completion, the client closes both the plaintext file and the socket. The execution of client.c concludes upon successful transmission of the plaintext file.

On the server side, `server.c` handles the configuration of the specified port, or defaults to port 3456 if none is provided. However, unlike the client, the server employs `bind()` and `listen()` functions after socket creation instead of `connect()`. The `bind()` function associates the socket with the specified port and signals the OS to accept incoming connections. Then by entering an infinite loop, the server continuously listens for new connections and calls `accept()` whenever a client attempts to connect. Upon success, it creates a unique identifier (`new_fd`) for the created socket and forks a child process to handle the communication separately - which leaves the parent free to continue accepting connections. The child process continues on to receive data from the client, displays it to the console, and then closes the associated socket. The execution of `server.c` concludes upon user intervention.

While both `client.c` and `server.c` use similar socket operations for communication, their approaches to initiating and managing connections differ due to their distinct objectives. In `client.c`, the `connect()` and `send()` functions are employed to establish and transmit data over an established connection. In contrast, `server.c` begins by using the `bind()` and `listen()` functions to prepare the socket for incoming connection requests from clients. Since the client is acting as the initiator of the connection, the server uses these functions to listen for incoming connections on a specific port. Therefore, while each program handles network communication, `client.c` focuses on connections establishment and data transmission, while `server.c` focuses on reception and acceptance of incoming connections.

2. (5pts) What is a port in an operating system? What are the differences between a TCP port and UDP port?

Operating system ports are conceptual entities that facilitate communication between network applications and services. In the context of the transport layer, such as TCP and UDP, ports are vital as they enable the OS to accurately and efficiently route traffic to its intended destination. The primary difference between TCP and UDP lies in their prioritization of either reliability or speed. For example, TCP prioritizes reliability by establishing a secure connection through a 3-Way Handshake Process before it transmits data. By doing this, it is able to ensure a complete, in order, and error-free data delivery. In comparison, UDP does not require a formal connection is established prior to delivery, so it bypasses this handshake process. As a result, the transmission of data is much faster, but there is a risk of data arriving out of order or being lost altogether.

Neither protocol is inherently better than the other, as they are each useful in specific scenarios. For example, UDP's speed advantage makes it suitable for scenarios where real-time data transmission takes precedence over complete delivery, such as in video transfers and online gaming. In these contexts, the immediate arrival of data outweighs concerns about its order or completeness; users would prefer that the video playback remains smooth and responsive, even if occasional frames are dropped. On the other hand, TCP's reliability makes it preferable for applications where data integrity is more important than speed, such as in file transfers and emails. In these contexts, users prioritize maintaining the accuracy of their important files or messages to prevent corruption, loss, or tampering, even if it means sacrificing some speed in the process. As a result, the ultimate choice between TCP and UDP depends on the specific requirements of the application or service, with each protocol offering distinct advantages depending on the desired balance between reliability and speed. [2]

3. (5pts) Which applications or protocols use ports 21, 22, 23, 80, 443, 3306, 3389?

Table 2 - Common Ports. *[3] [4]*

Port	*Protocol*	*Application*
21	TCP	File Transfer Protocol (FTP)
22	TCP	Secure Shell (SSH) SCP (Secure Copy Protocol)
23	TCP	Telnet
80	TCP	Hypertext Transfer Protocol (HTTP)
443	TCP	HTTP over SSL (HTTPS)
3306	TCP	MySQL Database
3389	TCP	Remote Desktop Protocol (RDP), Terminal Server

LAB 4 - USE OpenSSL TO ENCRYPT, DECRYPT, SEND, AND RECEIVE A FILE THROUGH A SOCKET

1. (5pts) Check the source code, client.c and server.c. Describe how the OpenSSL encryption and decryption functions are used in the source code.

In lab 4, the client and server versions extend the functionality of lab 3 by integrating encryption and decryption capabilities using the DES algorithm from the OpenSSL crypto library. While the lab 3 versions focused solely on establishing socket connections and exchanging plaintext data, the lab 4 versions incorporate encryption in the client and decryption in the server, thereby enhancing the security of communication between them.

DES (Data Encryption Standard) is a symmetric-key block cipher that uses a fixed block size of 64 bits and a single key for both encryption and decryption. By using a Feistel cipher structure, DES begins the process by dividing each block into two halves. One half undergoes permutations and substitutions based on the encryption key before it is recombined with the unchanged half using the XOR operation. This iterative process repeats for 16 rounds of encryption until the plaintext is fully converted into ciphertext. Then, during decryption, the same key and algorithm are applied to reverse the process. [5]

The lab 4 version of the client program utilizes this DES algorithm to expand the capabilities of the lab 3 client version. Similar to the lab 3 version, it establishes a connection to the a port and IP using the `connect()` function and then reads in a user-specified plaintext file. However, instead of transferring this file in the plaintext version, it integrates encryption through the DES algorithm from the OpenSSL crypto library before it to the server. The main modification occurs between lines 86-106 of the code where the DES key for encryption is initialized and configured.

```
// init des
DES_cblock key, input, output;
DES_key_schedule sched;

fprintf(stderr, "Setting up the DES library...\n");
DES_string_to_key("Mary had a little lamb, it's fleece as white as
snow. Everywhere that Mary went, the lamb would surely go...",
&key);

fprintf(stderr, "Schedualling the key...\n");

switch(DES_set_key_checked(&key, &sched)){
    case -1:
        fprintf(stderr, "Bad parity\n");
        _exit(42);
        break;

    case -2:
        fprintf(stderr, "Key is weak\n");
        _exit(42);
        break;
}
```

This section initializes `DES_cblock` variables to store the DES key, input data, and output data, generates a DES key from a provided string using `DES_string_to_key()`, and establishes the key schedule using `DES_set_key_checked()` to ensure its strength [6]. It then proceeds to open the file and use an iterative encryption process to encrypt blocks of 8 bytes at a time until it reaches end of file (EOF); this operation is executed on line 136 with the `DES_ecb_encrypt()` function using the predetermined key schedule.

```
if ((i == 8) || (c == EOF))
{
    // encryption
    DES_ecb_encrypt(&input, &output, &sched, DES_ENCRYPT);

    if (send(sockfd, output, 8, 0) == -1)
    {
        perror("Client: send() error lol!");
    }

    // print cipher text
    fprintf(stdout, "%c%c%c%c%c%c%c%c",
        output[0], output[1], output[2],
        output[3], output[4], output[5],
        output[6], output[7]);

    i = 0;
    memset(input, 0, sizeof(input));
}
```

The remainder of the program continues similar to the client program in lab 3, where the data is sent to the server in chunks until end-of-file (EOF) is reached. However, instead of transmitting plaintext data, lab 4 client sends ciphertext. The execution of `client.c` concludes upon the successful transmission of the plaintext file.

Similar to the client code, the updated server in lab 4 utilizes the DES algorithm to enhance the functionality of the previous version. As it now receives encrypted data from the client instead of plaintext, the server must employ the OpenSSL crypto library to decrypt the received ciphertext. The first part of this process occurs between lines 124-145 of the code, where the server initializes and configures the DES key for decryption.

```
// init des
DES_cblock key, input, output;
DES_key_schedule sched;
int c, i, j;

fprintf(stderr, "Setting up the DES library...\n");
DES_string_to_key("Mary had a little lamb, it's fleece as white as
snow. Everywhere that Mary went, the lamb would surely go...",
&key);

fprintf(stderr, "Schedualling the key...\n");
switch(DES_set_key_checked(&key, &sched))
{
    case -1:
    fprintf(stderr, "Bad parity\n");
    _exit(42);
    break;

case -2:
    fprintf(stderr, "Key is weak\n");
    _exit(42);
    break;
}
```

First, the server executed the same intialization process as the client by initializing DES_`cblock` variables to store the DES key, input data, and output data, generating a DES key from a provided string using `DES_string_to_key()`, and establishing the key schedule using `DES_set_key_checked()` to ensure its strength [6]. Then, after handling the connection process, it enters a decryption loop at 189-205:

```c
while (1)
{
    input[i] = buf[j];
    i++; j++;

    if (i == 8)
    {
      DES_ecb_encrypt(&input, &output, &sched, DES_DECRYPT);
      fprintf(stdout, "%c%c%c%c%c%c%c%c",
          output[0], output[1], output[2],
          output[3], output[4], output[5],
          output[6], output[7]);

      i = 0;
    }

    if (buf[j] == '\0')
        break;
}
```

As shown, the server uses an iterative decryption process to decrypt blocks of 8 bytes at a time until it reaches a null terminator '`\0`'; this operation is executed on line 196 with the `DES_ecb_encrypt()` function using the predetermined key schedule. Afterward, the server returns to its core function from lab 3 by displaying data from the client to the console and then closing the associated socket. The execution of `server.c` concludes upon user intervention.

In conclusion, the updated versions of both the client and server programs significantly improve the capabilities of their original versions by integrating encryption and decryption functionalities using the DES algorithm from the OpenSSL crypto library. Unlike the previous versions, which were primarily focused on establishing socket connections and exchanging plaintext data between the client and server, these new versions prioritize encryption (client) and decryption (server). This process enhances security and confidentiality in data transmission between the client and server.

2. (5pts) This lab demonstrates how to protect confidentiality via encryption for socket communications. As we discussed in the OSI security model, what security services are desired to protect data transmitted via socket communications? How to ensure/implement these security services as you identified?

- *Authentication*: The ability to confirm the identity of a sender is from its claimed source
    - *Peer Entity Authentication*: The ability to verify the identities of connected parties
        - Secure communication protocols like TLS/SSL can provide built-in support for mutual authentication and certificate-based authentication
    - *Data origin authentication*: The ability to verify the claimed source of received data
        - Signing data before transmissions with digital signatures can provide proof of the sender's identity

- *Access Control*: The ability to restrict access to resources by defining and enforcing policies to manage user permissions and privileges
    - Regular monitoring, updates, and patches can limit remote access and disable unnecessary services or ports

- *Data Confidentiality*: The ability of a system to ensure that transmitted data is protected from passive attacks and viewed only by authorized parties
    - Implementing encryption algorithms such as AES (Advanced Encryption Standard) or DES (Data Encryption Standard) can ensure that data is encrypted before transmission over sockets
    - Using secure communication protocols like TLS (Transport Layer Security) or SSL (Secure Sockets Layer) can establish encrypted connections between communicating entities

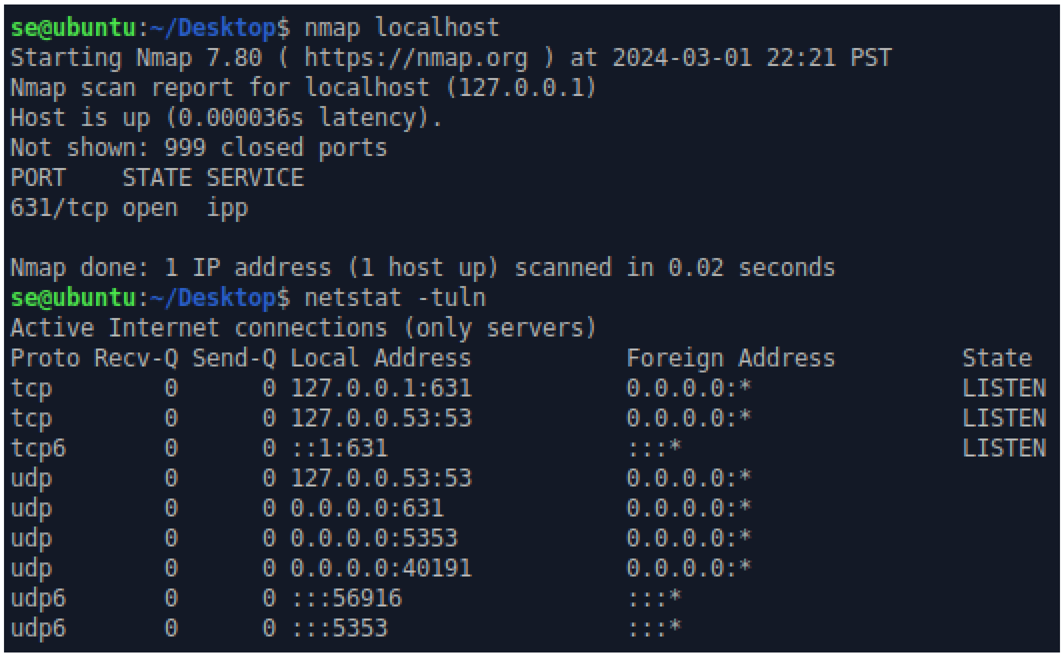
- *Data Integrity*: The ability of a system to ensure that data is modified only by authorized parties and are received as sent, with no "duplication, destruction, insertion, modification, reordering, or replays" [7]
    - Computing and comparing cryptographic hashes (e.g., SHA-256) can ensure data integrity before and after transmission.

- *Nonrepudiation*: The ability of a system to ensure that neither the sender nor the receiver can deny a message that has been transmitted
    - Using digital signatures to sign messages or transactions can provide evidence of the sender's identity

- *Availability Service*: The ability of a system to ensure that data can be accessed by any authorized parties on demand
    - Implementing thorough error handling and monitoring practices can help identify, Idiagnose, and address availability issues

3. (5pts) How to detect open ports in a computer?

The best way to detect open ports on a computer involves using a port scanner. There are many different types of these, including popular online or local tools such as Nmap/Zenmap, Netstat, and Nessus.



```
se@ubuntu:~/Desktop$ nmap localhost
Starting Nmap 7.80 ( https://nmap.org ) at 2024-03-01 22:21 PST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000036s latency).
Not shown: 999 closed ports
PORT    STATE SERVICE
631/tcp open  ipp

Nmap done: 1 IP address (1 host up) scanned in 0.02 seconds
se@ubuntu:~/Desktop$ netstat -tuln
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN
tcp6       0      0 ::1:631                 :::*                    LISTEN
udp        0      0 127.0.0.53:53           0.0.0.0:*
udp        0      0 0.0.0.0:631             0.0.0.0:*
udp        0      0 0.0.0.0:5353            0.0.0.0:*
udp        0      0 0.0.0.0:40191           0.0.0.0:*
udp6       0      0 :::56916                :::*
udp6       0      0 :::5353                 :::*
```

4. (10pts) How to exploit an open port in a computer? List at least two different approaches to exploit an open port.

Open ports pose significant risks as they may act as potential exploitatble entry points into a network, often as a result of outdated versions, misconfigurations, or unpatched flaws. One common approach to exploiting open ports involves identifying and targeting known vulnerable services or applications running on them. Attackers can leverage tools like Metasploit to automate and streamline this process in order to exploit security flaws such as buffer overflows and command injections. Through these exploits, attackers can gain unauthorized access to the target system or launch denial-of-service (DoS) or distributed denial-of-service (DDoS) attacks.

Another method to exploit open ports is through brute force attacks, particularly targeting services like SSH, which rely on username and password authentication. SSH's susceptibility to these attacks makes it vulnerable to automated attempts to guess login credentials. Metasploit also contains a brute force option to simplify this process and allow attackers to systematically try different password combinations until they gain access. If access is gained, attackers can execute malicious code, compromise sensitive data, or disrupt system operations. For these reasons, it is important that individuals implement the previously mentioned security services. [7] [8]

5. (10pts) An initial port scanning doesn't show any open ports in a network. Does it indicate the network is secure? What else you can do to exploit the network?

If an initial port scan does not show any open ports in a network, this does not necessarily mean that the network is secure. One possible reason could be because the ports are intentionally hidden or obscured from port scans through various techniques such as enabling port authentication rules, firewall rules, port knocking, or using non-standard port numbers. This is not fool proof, however, as sophisticated attackers may employ advanced scanning techniques or reconnaissance methods to bypass these rules and uncover hidden ports and vulnerabilities in the network [9]. Additionally, even if there are not any open ports on a network can exploit other vulnerabilities such as application layer weaknesses, social engineering, zero-day exploits, insider threats, and advanced persistent threats (APTs) to compromise the network's security [10].

6. (5pts) What is a meet-in-middle attack?

A meet-in-the-middle attack was developed to exploit certain vulnerabilities in cryptographic systems, particularly block ciphers, by exhaustively searching for the encryption key. In order to execute this attack, attackers require a known plaintext-ciphertext pair for reference, which are then encrypted and decrypted using all possible keys. The intermediate values generated during this process are then compared to identify potential matches and ultimately deduce the encryption key. To reduce the computational cost and complexity required in brute-force methods, these values can be stored and compared against a table to identify potential matches and deduce the encryption key efficiently [7].

7. (10pts) Is it possible to perform encryptions in parallel on multiple blocks of plaintext in CBC mode? How about decryption?

Since the input of Cipher Block Chaining (CBC) is the XOR result of the upcoming plaintext block and the previous ciphertext block, it would not be possible to perform encryptions in parallel. This is also true for decryption, because, although the decryption algorithm is applied to each cipher block individually, the resulting output is then XOR with the previous ciphertext block to generate the corresponding plaintext block [7].

8. (10pts) What is SSL splitting? Explain a scenario how to use SSL splitting to monitor SSL/TLS traffic?

SSL splitting is method for ensuring the integrity of data handled by proxies, without having to adjust client-side software. Unlike conventional proxy configurations that rely on a mutual trust between clients and servers, SSL splitting introduces a novel approach to maintaining data integrity. By leveraging the mutual authentication feature of the Secure Sockets Layer (SSL) protocol, SSL splitting allows the central server and the proxy server to use digital certificates to establish and verify their identities.

In SSL splitting, the central server, which holds the original data, initiates an SSL connection with the proxy server. During this SSL handshake process, both the central server and the proxy server authenticate each other using digital certificates. This mutual authentication ensures that both parties can trust each other's identities. Then, once the SSL connection is established, the central server sends SSL record authenticators to the proxy server along with the actual data payloads. These authenticators serve as cryptographic proofs of the data's authenticity and integrity. Upon receiving this data the proxy server merges them into a single data stream, which is then forwarded through SSL to the client. copies all data transmissions passing through the SSL server and forwards them to the proxy server for further analysis.

SSL splitting can be used to to monitor SSL/TLS traffic by intercepting the encrypted communication between clients and servers at the point of the proxy server. Since, the proxy must have access to the encryption keys in order to re-encrypt the merged data stream, this interception would allow monitoring and analysis of the encrypted traffic. And since, from the client's perspective, the data received via SSL splitting appears to be transmitted directly from the central server through a normal SSL connection, the client wouldn't necessarily be aware that their traffic was intercepted at the proxy. Essentially, the SSL record authenticators combined with the data payloads ensure that the client can trust the integrity and authenticity of the data, despite it passing through the proxy server [11].

References

[1] F. Valsorda, "The Heartbleed Bug," [Online]. Available: https://heartbleed.com/. [Accessed 2024].

[2] B. Gorman, "TCP vs UDP: What's the Difference and Which Protocol Is Better?," Avast, 23 February 2023. [Online]. Available: https://www.avast.com/c-tcp-vs-udp-difference#:~:text=Yes%2C%20TCP%20and%20UDP%20ports,comply%20with%20user%20datagram%20protocols..

[3] B. Lee, "Open ports and their vulnerabilities," *SpecOps,* 07 September 2021.

[4] J. Stretch, "Common Ports," Packet Life, [Online]. Available: https://packetlife.net/media/library/23/common_ports.pdf.

[5] C. Pfleeger, S. L. Pfleeger and L. Coles-Kemp, Security in Computing, 6th ed., Pearson Education, Inc., 2023.

[6] E. Young, "des_set_key_checked(3) - Linux man page".

[7] W. Stallings, Cryptography and Network Security: Principles and Practice, 6th ed., Upper Saddle, New Jersey: Prentice Hall Press, 2013.

[8] DRD_, "Gain SSH Access to Servers by Brute-Forcing Credentials," 2019.

[9] H. Al-Bahadili and A. H. Hadi, "Network Security Using Hybrid Port Knocking," *IJCSNS International Journal of Computer Science and Network Security,* vol. 10, August 2010.

[10] A. Rot and B. Olszewski, "Advanced Persistent Threats Attacks in Cyberspace. Threats, Vulnerabilities, Methods of Protection," in *Federated Conference on Computer Science and Information*, 2017.

[11] C. Lesniewski-Laas and M. F. Kaashoek, "SSL splitting: securely serving data from untrusted caches".