CSE354/CSE554: Networks and Systems Security II

Exercise 2 Write-up

Madhav Kansil (2022270)

1. Playing with eCryptfs

For this I set up Debian as my choice of distribution. For installing this on debian run `sudo apt install ecryptfs-utils`.

Commands used are pretty straightforward

```
# Making secret directory
mkdir secret

# mounting through ecryptfs
sudo mount -t ecryptfs secret/ secret/

# adding a file
cd secret/
echo "Contents" > very_secret.txt

# Reading file with eCryptfs mounted
cat very_secret.txt

# unmounting secret
cd ..
sudo umount secret/

# reading after umount
cat secret/very_secret.txt

# Final cat gives us garbage
```

The `-t` option tells the kernel to attach the filesystem found on the device in this case `eCryptfs`.

```
madhav@osiris:~$ sudo mount -t ecryptfs secret/ secret/
[sudo] password for madhav:
Select key type to use for newly created files:
 1) tspi
 2) passphrase
Selection: 2
Passphrase:
Select cipher:
 1) aes: blocksize = 16; min keysize = 16; max keysize = 32
 2) blowfish: blocksize = 8; min keysize = 16; max keysize = 56
 3) des3_ede: blocksize = 8; min keysize = 24; max keysize = 24
 4) twofish: blocksize = 16; min keysize = 16; max keysize = 32
 5) cast6: blocksize = 16; min keysize = 16; max keysize = 32
 6) cast5: blocksize = 8; min keysize = 5; max keysize = 16
Selection [aes]:
Select key bytes:
 1) 16
 2) 32
 3) 24
Selection [16]:
Enable plaintext passthrough (y/n) [n]:
Enable filename encryption (y/n) [n]:
Attempting to mount with the following options:
  ecryptfs_unlink_sigs
  ecryptfs_key_bytes=16
  ecryptfs_cipher=aes
  ecryptfs_sig=f4b5e0835cd34347
Mounted eCryptfs
madhav@osiris:~$ cd secret/
madhav@osiris:~/secret$ echo "Contents" > very_secret.txt
madhav@osiris:~/secret$ cat very_secret.txt
Contents
madhav@osiris:~/secret$ cd ..
madhav@osiris:~$ sudo umount secret/
madhav@osiris:~$
```

Figure 1. Commands for `eCryptfs`

```
madhav@osiris:~$ cat secret/very_secret.txt
```

Figure 2. Garbage Read

2. AES enabled echo server

```
# For commands as text see 3.2

# On Server Side
```

Figure 3. Server One-liner

```
# On Client Side
```

Figure 4. Client One-liner

2.1. Explanation

`PS:` I have encoded the encryption in base64 (using openssl base64) to send them to and fro. This is because I used `openssl cms` which gives a MIME format. That format has line breaks and new line character. Therefore to send the encryption in one long string base64 is used.

Also it is assumed that both server and client know the `nc.key` i.e. the key used to encrypt the data and `secretkeyid`.

2.1.1 Common Commands

1. `openssl cms -encrypt -secretkey $(cat nc.key) -secretkeyid "14092024" -wrap aes256-wrap -aes-256-gcm`
   This command is openssl cms command to encrypt with AES-GCM.

   - Using `openssl cms -encrypt` we set the mode of openssl to cms encrypt.
   - `-secretkey` is the secret key used in encryption process. Key is a 32 byte hex and is assumed to be in the `pwd` while running this command. Key is passed to the `-secretkey` option by using `$(cat nc.key)`.
   - `-secretkeyid` is the key identifier needed for the symmetric key. It should be hex. I have chosen my 20th birthday as the key id.
   - Then finally by using `-wrap aes256-wrap` the key is wrapped for AES256 cipher.
   - Then we pass our cipher which is `-aes-256-gcm`.
   - Due to the length of the command, this command would be referred to as `enc_cmd` in the report.

2. `openssl cms -decrypt -secretkey $(cat nc.key) -secretkeyid "14092024"`
   This command is openssl cms command to decrypt with AES-GCM.

   - Using `openssl cms -decrypt` we set the mode of openssl to cms decrypt.
   - Rest of the flag are same as `enc_cmd`.
   - For the rest of the report this command will be refered to as `dec_cmd`.

3. `openssl enc -base64 -A` :- Used to encode to base64. The `-A` option makes the output without line breaks and newline characters. This will be referred to as `base_enc`.

4. `openssl enc -base64 -d -A` :- For base64 decoding. Will be referred as `base_dec`.

2.1.2 Server Command

Flow of command

```
echo -n '' > ~/session; tail -f ~/session
| while read line; do echo "hello $(echo
$line | base_dec | dec_cmd)" | enc_cmd |
base_enc;done | nc -lvkp 9999 > ~/session
```

1. `echo -n '' > ~/session` : This line creates a new file named session with nothing in it. File is created in user's home directory. This file is important as using this file we are able to achieve a listening loop for the server. (See Appendix (3.1))

2. `tail -f ~/session` :- This `tail` command with `-f` option "follows" the file that is if anything is added to the file it gets displayed at stdout.

3. `while read line; do echo "hello $(echo $line | ...) ..."` This first decodes and decrypts the name sent by client and then prepends hello to that name and encrypts and encodes the hello string and sent it back to client. While loop reads the stdout into $line.

4. `nc -lvkp 9999 > ~/session` starts a netcat session on port 9999 and any thing that is sent by client gets written to session file.

2.1.3 Client Commands

Flow of command

```
while true; do read line; (echo "$line" |
enc_cmd | base_enc;echo) | nc -w 1 localhost
9999 | base_dec | dec_cmd;done
```

1. `while true; do read line;` This reads name/proper noun from stdin.
2. `(echo "$line" | enc_cmd | base_enc;echo)` in a subshell we encrypt and encode and add a new line at the end of base64 encoded string. New line is appended for formatting and parsing purposes.
3. `nc -w 1 localhost 9999` opens a netcat connection to localhost at port 9999. The option `-w 1` timesout the connection after 1 second.
4. Everytime a new proper noun is sent. A new connection is established.

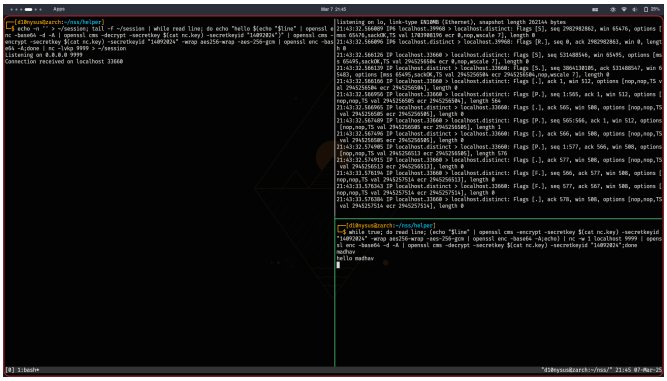

Figure 5. Tcpdump and output

3. BONUS

There are a few assumptions.

- Both the server and client know the secret phrase for creating HMAC.
- There was an ambiguity in the problem statement, whether to use AES encrypted channel used in Problem 2 or not. I have used that encryption to send files to send files from server (sender) to client (receiver). That is the HMAC that needs to be compared is sent through encrypted channel and file is first encrypted then sent.
- HMAC computed is for encrypted file. Not for the unencrypted file. This is because if our channel is to get compromised. Our HMAC would not match and file would not even get decrypted.

3.1. New Commands

1. Commands like `enc_cmd` and `dec_cmd` remain the same. Some modification like `enc_cmd(filename)` means that file by the name of filename is encrypted.
2. `openssl dgst -sha256 -hmac "madhav" <filname>` gives the hmac sha256 for the file given with a passcode (my name in this case).
3. `tr -d '\r'` this is translate command in bash. This is used for cutting the carriage return (\r). Carriage return is added by openssl in their encrypt and decrypt implementation. To make HMACs consistent we need to trim this.
4. `tar -cf - <filename1> <filename2>` this command with options `-cf` creates a new tar archive which gets sent to stdout due to presence of `-`.
5. `tar -xvf - > /dev/null 2>&1` This command extracts the tar archive received. Due to presence of `-` names of the files get written to stdout. to supress this its output is sent to /dev/null and `2>&1` redirects stderr to stdout. These tar commands have been taken from this article.
6. `nc -lvp 9999`. In this command `-k` flag is missing, this is because there is no requirement for to and for communication as we are emulating scp.

3.2. Explanation

3.2.1 Server Side

Flow of Command

```
  enc_cmd(file.txt); hmac_creation(encrypted.file)
| enc_cmd > encrypted.hmac; tar -cf -
encrypted.file encrypted.hmac | nc -lvp 9999;
rm encrypted.*
```

1. First we take the file which need to be copied and encrypts it with AES-GCM. Then we create HMAC for that encrypted file and encrypt it to the file named encrypted.hmac.
2. All of the files are then packaged into an tar archive and a new netcat server is spawned so that any client that connects to it receives the encrypted files.

3.2.2 Client Side

Flow of Command

```
nc -w 1 localhost 9999 | tar unpack;
if [["$(dec_cmd(encrypted.hmac) | trim)"
== "$(hmac_create(encrypted.file))"]];
then echo "HMAC Matched ..  Decrypting";
dec_cmd(encrypted.file);else echo "No
Match";fi
```

1. A netcat client is connected to the server. Files received the are unpacked using tar command.
2. Now in our `pwd` we have both encrypted.file and encrypted.hmac. Now we compare hmac received from the server by comparing whether `dec_cmd(encrypted.hmac)` equals to the hmac of the encrypted.file received.
3. If HMACs are same then an acknowledgment is printed and the file is decrypted and saved in `pwd`.
4. If not an appropriate message is printed.

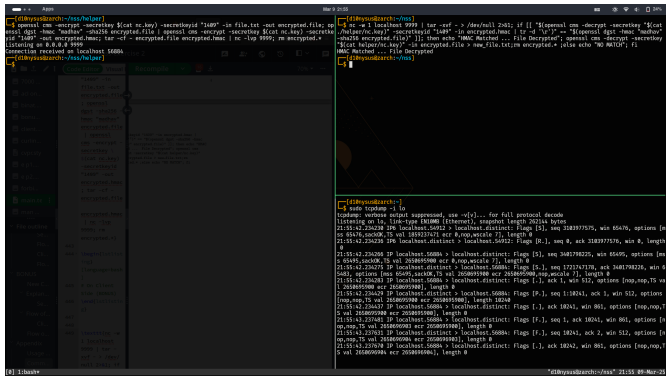

Figure 6. Tcpdump for SCP emulation

4. Appendix

4.1. Usage of Session File

By this stack overflow article to redirect stdin and stdout to netcat one can use file descriptors and pipes. This method is also laid out in man page of netcat (screenshot attached). In this one can build a server which does not exit/quit after servicing one query. The example showed a persistent connection to client. This use case fits with the problem statement given as the server should not stop/quit after receiving a single proper noun.

This is why I have used a session file. `tail` command follows it using `-f` option and any changes

Figure 7. Netcat man page

happen are written to stdout, which is then piped and read by the server for decryption. Session file contains the encoded text recieved from client. Due to ”>” redirection at each new input file is overwritten. This also helps in memory management as file would not increase in size with increase in number of queries.

Session file only helps in keeping the server alive after a request has been processed. All the encryption decryption is done through pipes and bash commands and no variables are used to store intermediate results.

4.2. Commands as text

```
# On Server Side
```

```
echo -n '' > ~/session; tail -f ~/session
| while read line; do echo "hello $(echo
"$line" | openssl enc -base64 -d -A |
openssl cms -decrypt -secretkey $(cat
nc.key) -secretkeyid "14092024")" | openssl
cms -encrypt -secretkey $(cat nc.key)
-secretkeyid "14092024" -wrap aes256-wrap
-aes-256-gcm | openssl enc -base64 -A;done |
nc -lvkp 9999 > ~/session
```

```
# On Client Side
```

```
while true; do read line; (echo "$line" |
openssl cms -encrypt -secretkey $(cat nc.key)
-secretkeyid "14092024" -wrap aes256-wrap
-aes-256-gcm | openssl enc -base64 -A;echo) |
nc -w 1 localhost 9999 | openssl enc -base64
-d -A | openssl cms -decrypt -secretkey $(cat
nc.key) -secretkeyid "14092024";done
```

```
# On Server Side (BONUS)
```

```
openssl cms -encrypt -secretkey $(cat
nc.key) -secretkeyid "1409" -in file.txt
-out encrypted.file; openssl dgst -sha256
-hmac "madhav" encrypted.file | openssl
cms -encrypt -secretkey $(cat nc.key)
-secretkeyid "1409" -out encrypted.hmac;
tar -cf - encrypted.file encrypted.hmac | nc
-lvp 9999; rm encrypted.*
```

```
# On Client Side (BONUS)
```

```
nc -w 1 localhost 9999 | tar -xvf -
> /dev/null 2>&1; if [[ "$(openssl cms
-decrypt -secretkey "$(cat ./helper/nc.key)"
```

```
-secretkeyid "1409" -in encrypted.hmac | tr
-d '\r')" == "$(openssl dgst -sha256 -hmac
"madhav" encrypted.file)" ]]; then echo "HMAC
Matched ...  File Decrypted"; openssl cms
-decrypt -secretkey "$(cat helper/nc.key)"
-in encrypted.file > new_file.txt;rm
encrypted.* ;else echo "NO MATCH"; fi
```