Computer Security
Coursework Exercise 2

February 23, 2022

In this coursework we will touch upon a number of security-related topics, namely public key encrypting and decrypting with GPG, symmetric key encrypting and decrypting with OpenSSL, spoofing the sender of an email and performing a MitM attack. The deadline is 11 March 2022, 16:00.

1 Asymmetric Encryption with GPG

In this section you will learn to use GPG for day-to-day usage, most importantly including signing and verifying signatures. You will also have to prove your knowledge by solving a challenge. GPG (sometimes written as GnuPG) is the GNU Privacy Guard. GPG is an open source implementation of the OpenPGP standard for asymmetric encryption.

1.1 Introduction to GPG

The purpose of this part is to familiarise yourself with the GPG tool. You will see how to receive the public keys of other people and use them for encryption and signature verification. You will also learn how to generate private keys and use them for signing and decryption. The instructions that follow are specific for DICE machines, but should work on any Linux machine with slight variations. They should also work on MacOS machines with minimal adaptation. There exist some ports of GPG for Windows, but they are not supported in this coursework. If you prefer you can solve the exercise using only the terminal, so access to DICE using `ssh` should be sufficient.

1.1.1 Verifying Signatures

Verifying the integrity of software you download is important to ensure that your software hasn't been tampered with. This section will show you how to verify signatures if they are available. Your task is to download the Alpine Linux mini root filesystem armv7[1] and the corresponding signature[2] and verify it. Save the contents of the second link to a file. You should place both files in the same directory. The names of the files are important: the signature must have the same name as the file it signs, with the added extension '`.asc`'.

Before you can verify the signature however, you need to import the public key which was used to make it. These are also available from the Alpine Linux download page[3]. The fingerprint of the signing public key is `0482 D840 22F5 2DF1 C4E7 CD43 293A CD09 07D9 495A`. More on what a fingerprint is later. To receive the public key, execute:

```
gpg --recv-key '0482 D840 22F5 2DF1 C4E7 CD43 293A CD09 07D9 495A'
```

This could take a while. You should see a report of the key which was imported. Next, to verify the file itself, go to the directory where you downloaded the files and run:

[1] `http://dl-cdn.alpinelinux.org/alpine/v3.11/releases/armv7/alpine-minirootfs-3.11.3-armv7.tar.gz`

[2] `http://dl-cdn.alpinelinux.org/alpine/v3.11/releases/armv7/alpine-minirootfs-3.11.3-armv7.tar.gz.asc`

[3] `https://alpinelinux.org/downloads/`

```
gpg --verify alpine-minirootfs-3.11.3-armv7.tar.gz.asc
```

You should see a line stating '`Good signature from <person>`'. This indicates that the signature is valid, and that you have the signer's public key. You will also see a rather scary-looking warning, which indicates that you haven't assigned the public key a trust level. Proper GPG usage recommends to verify your correspondents' keys by checking their fingerprint and subsequently signing their key and setting your trust level towards them, however we will not focus on it here.

Keep in mind that the aforementioned steps do not rule out completely the possibility of a Man in the Middle attack. An attacker could hijack the legitimate site, replace the original public keys with his own, put a backdoor in the provided source code and sign it with his key. GPG itself can only rule out such attacks if you have out-of-band reasons to trust the validity of the provided fingerprint. Such an out-of-band reason is the acknowledgement that the website itself is valid through TLS security.

1.1.2 Generating a Keypair

In order to sign or receive encrypted messages, you will need your own key pair. To generate one, run:

```
gpg --gen-key
```

Complete the command line dialogue, and wait for the key to be generated. The default option for key type (RSA both for encrypting and signing) is sufficient. Note that, after the key generation phase, the underlying algorithms are handled by gpg under the hood, so you should never run into problems because of the key type of others. A key length of 4096 and an expiration date after one year are recommended, and the comment field should be left empty. Note that it is **highly** recommended to secure the key with a strong passphrase. Your private key is your digital identity, do not treat it lightly.

1.1.3 Key IDs

Many commands in GPG need to identify the key to use. The public keys available can be listed with the command '`gpg -k`', and the private keys with '`gpg -K`'. Each key is associated with a long (160 bits) hexadecimal ID, which can be used to refer to it, known as the fingerprint of the key. Add the option '`--fingerprint`' to the previous commands to display it. More conveniently, keys can also be referred to by their email address.

1.1.4 Key Management

Once you've generated a key, there are a few maintenance operations you may need to do from time to time.

1. Upload your public key to the keyserver at '`https://keys.openpgp.org`'. For doing this you should run '`gpg --send-keys <Key ID>`'.

2. Make sure you can receive a coursemate's public key. After they have uploaded theirs, run '`gpg --recv-keys <Key ID>`'. You may have to wait a few minutes for their key to propagate before receiving it. Note that in this situation, the key ID *must* be the full fingerprint; an email address does not suffice.

3. Generate a revocation certificate for your key, using the command '`gpg --gen-revoke <Key ID>`'. A revocation certificate can be used to invalidate your key pair. This is not something you want to do right now, however it is helpful to know what to do. The revocation certificate can be imported with '`gpg --import`', similarly to keys. The (now revoked) public key can then be pushed to a keyserver. This may be useful if you want to stop using the particular email address or your private key has leaked.

4. You can export your keys with the command '`gpg --export > gpg.keys`'. This will create a binary file 'gpg.keys', containing all public keys in your database. It is also possible to export private keys, using the command '`gpg --export-secret-keys > gpg_private.keys`'. When exported in this way, the keys are still encrypted with your passphrase.

1.1.5 Signing Messages

GPG signatures operate on files. The most basic way to sign a file is to execute '`gpg -b <file>`'. This will create a new file, called '`<file>.sig`', which contains the signature of the file with your private key. Adding the `-a` option will force the signature to be generated in an ASCII format, making it more convenient for embedding.

It is also possible to package the data together with the signature, by running '`gpg -s <file>`'. This is typically used in conjunction with encryption.

1.1.6 Encrypting and Decrypting Messages

To encrypt a message, double check that you have a coursemate's public key. Create a plain text file containing your message, and then encrypt it with '`gpg -e <file>`'. Send the newly created file to your coursemate. The same command can also be run with the `-s` option, to also sign the message, and the `-a` option to create an ascii-formatted message.

Hopefully you will have received an encrypted message from one of your coursemates. If not, ask someone to send you one. To decrypt the message, simply run '`gpg -d <file>`'.

2 Symmetric encryption with OpenSSL

OpenSSL is another toolkit for secure communication. In this section you will learn to use OpenSSL for symmetric key encryption. You will also have a challenge to solve.

The command '`openssl enc -cipher`', presents the ciphers that OpenSSL supports. The simple way of encrypting a file is executing the following command:

```
openssl enc <-algorithm type> -e -in <inputfile> -out <outputfile>
```
[4]

After executing the aforementioned command a password is needed for encryption. The same password should be used for decryption, and for decrypting -e is replaced with -d and encrypted file is inserted in `<inputfile>`.

For saving the encrypted file in base-64 format the argument -a or -base64 is added to the aforementioned command. The secret key of the encryption algorithm drives from the password. For creating a better secret key, it is highly recommended using the -pbkdf2 argument. It is worth mentionening that any kind of argument such as -a, -pbkdf2 that is used for encryption should be used for decryption as well. Otherwise the decryption will fail. Also the arguments should be written in order. For understanding the order of the arguments, visit `https://www.openssl.org/docs/man1.1.1/man1/openssl-enc.html`.

In OpenSSL, it is also possible to generate a random secret key for specific type of symmetric algorithm. For instance for generating a secret key for AES-256-CBC, the command '`openssl enc -aes-256-cbc -k -pbkdf2 -P -md sha256 -pbkdf2`' is executed.

3 Encrypted email exercise

In this exercise you will have to prove your ability to encrypt and decrypt messages correctly. This is the only marked exercise in the GPG and OpenSSL section.

1. Generate a keypair if you don't already have one and upload the public key to the keyserver as explained above[5].
2. Send an email from your student address with subject "fingerprint <your fingerprint>" and empty body to `s2006521@ed.ac.uk`. There should be *no whitespace* in the fingerprint. The only whitespace in the subject should be a single space between the word "fingerprint" and the actual fingerprint. For example, if your fingerprint is `DEAD BEEF`, the subject should be "fingerprint DEADBEEF". It should be notted that only those students who send email from their student email account will take mark for this part.

[4] the command '`openssl <algorithm type> -e -in <inputfile> -out <outputfile>` also works.

[5] The key does not necessarily have to be tied with your student email account, but you will have to have access to your student email account in order to complete the exercise.

3. You will receive through email a text, encrypted with the public key corresponding to the fingerprint you uploaded and you have to decrypt the text. The text contains 2 challenges one related to GPG, the other related to OpenSSl. You should solve both them. Note that due to technical difficulties on our end, the challenge may be arrive with a delay of some hours (especially at night).

4. For the GPG challenge, you should encrypt the rseponce of the challenge with the public key with fingerprint `D175 E7D4 A3C5 4B3F 004A F12E 8047 FE5B 9C5B 89F8` and for the OpenSSL challenge, you should encrypt the response of the challenge with AES algorithm with the secret information that you find in encrypted text.

5. Create 2 separated files containing *only* the answers and encrypt them. You should name the first encrypted file as `public-key-response.gpg` and second encrypted file as `symmetric-key-response.aes`.

6. Submit the encrypted answer using the "GPG-OpenSSL" link in BlackBoard Learn. The link should lead you to the CodeGrade submission platform, where you can submit your file.

4 Spoofing email sender

For this exercise, you will send us an email with a spoofed email sender field:

- The subject line of your email should be your student id
- The sender of your email should be `darth.vader@starwars.com`
- You will send your email to `s2006521@ed.ac.uk`.

One way of doing this is by using the `mailx` utility program. You are free to try this amongst yourselves before you actually send your email to us. Spoofing is probably easier from a DICE machine than from your computer. (You won't have to submit anything through Learn for this.)

5 (wo)Man in the Middle Attack

You are asked to mount a (wo)Man-in-the-Middle (MitM) attack against the toy implementation of an encrypted chat between terminals provided in `/afs/inf.ed.ac.uk/group/teaching/compsec/cw2/mitm/`.

5.1 High-level overview

When Alice and Bob hear about encryption, they immediately set out to implement an encrypted chat client so that they are sure no one eavesdrops their intimate discussions. They decide to use AES[6] to encrypt their messages, since everyone says it's the best. They also hear of the Diffie-Hellman key exchange[7] (DHKE) and figure it would be cool to use a new secret key for AES every time they connect.

5.1.1 AES

Just like every symmetric encryption scheme, AES consists of two algorithms:

- The encryption algorithm takes a key K_1 and a message M_1 as input and returns a ciphertext C_1 as output: $C_1 = Enc(K_1, M_1)$
- The decryption algorithm takes a key K_2 and a ciphertext C_2 as input and returns a message M_2 as output: $M_2 = Dec(K_2, M_2)$

[6] `https://en.wikipedia.org/wiki/Advanced_Encryption_Standard`
[7] `https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange`

If a message M is encrypted with key K and the resulting ciphertext C is decrypted with the same key K, the result of the decryption will be the original message M: $\forall K \forall M, M = Dec(K, Enc(K, M))$

A simple library for encrypting and decrypting using `pyaes` is provided in `symmetric.py`.

5.1.2 Diffie-Hellman Key Exchange

This is a protocol between two parties (say Alice and Bob) that want to obtain a common key that is unknown to anybody else. Their communication takes place over an insecure channel that anyone can eavesdrop.

A physical-world equivalent is the following: A group of people sit around a table and two of them want to speak in private. They can have a brief exchange (of very long numbers) *which everybody hears*. After that they will possess a common secret *that no one else knows*. They can use this secret as the key for encrypting, sending and decrypting private messages in plain sight.

We assume that both parties have agreed beforehand on a finite cyclic group G and a generator g of G. For production software, these parameters are standardised by cryptographers and hardcoded in the implementation by the developers.

These are the steps of the protocol:

- Alice chooses a random number x and calculates $a = g^x$.
- Alice sends a to Bob.
- Bob chooses a random number y and calculates $b = g^y$.
- Bob sends b to Alice.
  - Now all eavesdroppers know a and b, but not x and y.
- Alice derives the common secret b^x.
- Bob derives the common secret a^y.

Given that $(g^x)^y = (g^y)^x$, both Alice and Bob have derived the same common secret. Assuming that an eavesdropper cannot find x from a or y from b, we conclude that no one else can derive the common secret.

A simple library for doing the necessary steps of DHKE is provided in `diffie_hellman.py`. You can see how to use it in the `do_Diffie_Hellman()` function in `util.py`.

5.1.3 Putting it all together

The entire process of chatting is then as follows:

1. Alice and Bob establish a communication socket
2. They do DHKE over this socket
3. Bob encrypts his message under the derived key (with AES)
4. Bob sends the resulting ciphertext through the socket
5. Alice decrypts the ciphertext using the derived key
6. Alice reads the message

Steps 3–6 can be repeated as many times as desired, possibly with changed roles. (In our implementation, the process is repeated only twice, so Bob sends first, then Alice, then both parties terminate.)

5.1.4 MitM attack

The described approach sounds very reasonable. Unfortunately Alice and Bob overlooked a fatal flaw: When communicating over the internet (or even locally), one cannot know with certainty that they are speaking to the intended party, at least not without using some form of cryptographic *authentication*[8].

Going back to our round-table example, consider the case where every member of the group wears a different mask, uses a voice jammer and sits at random seats. Alice would be unable to recognize Bob. In an even worse scenario, if Bob happens to be missing from the table, someone with a good disguise could impersonate him and fool Alice into performing DHKE with him. This is why Alice and Bob should have agreed to only speak to each other after authenticating themselves.

Given that no authentication takes place, Eve the attacker is now able to do the following: After Alice opens his end of the socket and before Bob connects, Eve connects and performs a DHKE with Alice. Eve then opens a new socket and waits for Bob to connect. When Bob and Eve connect, they perform another DHKE. Now Eve can decrypt messages from one party, read them and reencrypt them for the other party. If she so wishes she can even send arbitrary messages, completely unrelated to the original ones. In short, she has complete control of the channel while Alice and Bob think they communicate with each other privately.

5.2 Implementation details

5.2.1 How to use the provided code

Open two terminals and navigate to the directory with the scripts. First run `python3 alice.py` in one and then `python3 bob.py` in the other (the order is important). You should see secure channel establishment, a couple of messages being exchanged and finally the channel closing.

5.3 Code overview

Open both aforementioned scripts with your favourite editor. Each of the two scripts calls `setup()` with its name and the name of the pre-agreed buffer file over which communication happens. This name is set in `const.py`. Then Alice waits for a message, while Bob sends it. Alice then prints the message and the roles are reversed. Finally both parties close their sockets.

Familiarise yourself with the scripts and understand which lines correspond to each of the steps above. You can optionally dive in the code of the various supporting sources as well.

5.4 Exercise

You will have to implement and submit `eve.py` via CodeGrade – use the "MitM" link to the submission platform in BlackBoard Learn. The attack should execute correctly when first `alice.py` is started in one terminal, then `eve.py` in a second and last `bob.py` in a third. `eve.py` should be followed by exactly one of the following three flags: `--relay`, `--break-heart` or `--custom`.

- If the flag is `--relay`, Eve should just relay the two messages from Alice to Bob and from Bob to Alice. In this case, the outputs of both `alice.py` and `bob.py` in the terminals should be identical to the case when the MitM attack isn't executed. Eve should be prepared to relay arbitrary messages, not just the hardcoded ones!
- With the `--break-heart` flag, Eve should change the messages so that Alice receives the message "`I hate you!`" and Bob receives "`You broke my heart...`". Remember, Eve still has to encrypt both messages correctly.
- As for the `--custom` flag, after receiving Bob's messsage, Eve must prompt the user to input a message to the terminal and then must send this message to Alice instead. The same should happen for Alice's message; Eve must prompt the user for a second message and this time send it to Bob.

[8] `https://en.wikipedia.org/wiki/Message_authentication_code`

Hint: Your solution will have to use the buffer file somehow. The function `os.rename()` will prove helpful.
Note: It may happen that a script dies without closing its socket gracefully. In that case, you should manually remove the remaining buffer file (by default called `buffer`) before restarting the scripts.

```
➜  public python alice.py
Hi Alice! Welcome to SuperSecureChat!
Establishing secure channel with your better half...
Established secure channel!
Waiting for message...
Bob said: "Chatting with you is boring..."
Message sent!
Closing socket...
Socket closed! Bye-bye!
```

```
➜  public python eve.py --custom
Hi Bob! Welcome to SuperSecureChat!
Establishing secure channel with your better half...
Established secure channel!
Eve thinks: "Nice, getting there..."
Hi Alice! Welcome to SuperSecureChat!
Establishing secure channel with your better half...
Established secure channel!
Eve thinks: "Hehe, it's working!"
Bob slurred: "I love you!"
Input what you would like Bob to say to Alice
Chatting with you is boring...
Alice slurred: "What?!"
Input what you would like Alice to say to Bob
I don't love you back though
Closing socket...
Socket closed! Bye-bye!
Closing socket...
Socket closed! Bye-bye!
```

```
➜  public python bob.py
Hi Bob! Welcome to SuperSecureChat!
Establishing secure channel with your better half...
Established secure channel!
Message sent! Waiting for reply...
Alice said: "I don't love you back though"
Closing socket...
Socket closed! Bye-bye!
```

Figure 1: The output of your `eve.py` does not have to match the example, but that of `alice.py` and `bob.py` have to.