Computer Security
Coursework Exercise 2

October 19, 2017

This coursework is a formative assessment. That means that we will be marking the submitted coursework, but that the mark will in no way contribute to your final mark for the course. The coursework is intended to help you learn more about computer security as well as help you understand how well you understand course material. We will be providing feedback by Thursday 2nd November. If you do want to receive feedback you need to submit your solutions before that date.

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.

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 Tor browser stable source code for Linux 64-bit[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 signes, 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 Tor website[3]. The fingerprint of the signing public key is `EF6E 286D DA85 EA2A 4BA7 DE68 4E2C 6E87 9329 8290`. More on what a fingerprint is later. To receive the public key, execute:

```
gpg --recv-key 'EF6E 286D DA85 EA2A 4BA7 DE68 4E2C 6E87 9329 8290'
```

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]https://www.torproject.org/dist/torbrowser/7.0.6/tor-browser-linux64-7.0.6_en-US.tar.xz
[2]https://dist.torproject.org/torbrowser/7.0.6/tor-browser-linux64-7.0.6_en-US.tar.xz.asc
[3]https://www.torproject.org/docs/signing-keys.html.en

```
gpg --verify tor-browser-linux64-7.0.6_en-US.tar.xz.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 Cryptographic Checksums

Many programs are not distributed with signature files. It is more common to offer a cryptographic checksum. While this is not a replacement for signatures, and provides far less security, it is helpful to know how to verify them.

There are multiple hash functions which are used for cryptographic checksums, the most common of which is **SHA256**. To compute the checksum of a file, the command `sha256sum <file>` can be used. Calculate the SHA256 checksum of the Tor source code, and verify that it matches the following:

```
d5e0b7803902d08868bae59de3f939d390c513cc944c9aa28be8dc730ac8e387
```

Please note that older hash functions such as SHA1 and especially MD5 are broken and not fit for security purposes anymore. SHA2 or even better SHA3 are recommended for cryptographic use in all projects.

1.1.3 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
```
[4]

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.

1.1.4 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.

1.1.5 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 '`hkp://keys.gnupg.net`'. You will have to set the '`keyserver`' option in '`~/.gnupg/gpg.conf`', and then run '`gpg --send-keys <Key ID>`'.

[4]On a DICE machine, you will first have to issue the command `gpg-agent --daemon --no-use-standard-socket` and then execute the command that is returned in order for the key generation to function properly.

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.

1.1.6 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.

1.1.7 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>`'.

1.2 Encrypted email challenge

In this challenge you will have to prove your ability to encrypt and decrypt messages correctly. The submission steps of this exercise should be completed while logged in on `vulcan`. Thus, before you start this challenge, ensure that you are logged in on `vulcan` or run "`ssh <uun>@vulcan.inf.ed.ac.uk`" and provide your passphrase if you are logged in on any other DICE machine.

1. Generate a private key if you don't already have one and upload it to the keyserver as explained above[5].

2. Submit a file named exactly '`fingerprint`' containing *only* your fingerprint to the Computer Security, Coursework 2 directory (`cs cw2`) using the `submit` DICE tool.

3. You will receive through email the challenge, encrypted with the public key corresponding to the fingerprint you uploaded. Decrypt it and solve the challenge.

4. Receive the key with fingerprint `D969 38C4 1C22 9F32 A8D5 5AD6 7C28 53A6 899B D0EF`[6].

5. Create a file containing *only* the answer and encrypt it using the public key that you just received. Use the GPG option '`-o solution`' when encrypting to create the encrypted file with the name '`solution`'.

[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 challenge.

[6]The corresponding email address is fake, therefore sending anything there is pointless.

6. Submit the encrypted answer as a file named exactly '`solution`'. If the answer is correct, you will receive a confirmation email.

2 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 `mickey.mouse@disney.com`
- You will send your email to `cw2@vaniea.com`.

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.

3 Password Cracking

This question will involve (partially) cracking a list of hashed passwords. The password lists are based on the 2009 RockYou password leaks. The files `rockyou-samples.md5.txt`, `rockyou-samples.sha1-salt.txt`, and `rockyou-samples.bcrypt.txt` store 100,000 password hashes each. These files can be found on DICE in the `/group/teaching/cs` directory. Each function is progressively more resistant to password cracking than the previous one.

3.1 Brute-forcing MD5

The file `rockyou-samples.md5.txt` contains MD5 hashes of passwords, encoded in hexadecimal. Write a program in a programming language of your choice, which brute forces all five character passwords, using only numbers and lowercase ASCII letters (0-9 and a-z). The program should create an output file `md5-cracked.txt` which contains the passwords corresponding to the hashes in rockyou-samples.md5.txt and the number of occurrences for each of them. As the output format, keep one entry per line, with each entry being of the form `n,password` (E.g. `10,apples`). Finally, create a sorted output by running:

```
sort -t, -k1n,1 -o md5-cracked-sorted.txt md5-cracked.txt
```

Submit the file `md5-cracked-sorted.txt`.

Be aware that the naive implementation of brute forcing will not be sufficient here. You will need to check each possible password against all hashes very quickly, it is therefore strongly advised to use hashmaps (or, even better, multisets).

3.2 Cracking Common Passwords with Salted SHA-1

The file `rockyou-samples.sha1-salt.txt` contains SHA-1 hashes of passwords with a salt. The format of each line is `$SHA1p$`*salt*`$`*hash*, where:

$$hash = H_{\texttt{SHA-1}}(salt || password)$$

The approach to brute-forcing from Part 1 can't be used in this case, as there is no fast way to check a given password against the entire list. Instead, write a program which tries the 25 most common passwords against the entire list, and reports how often each occurred (in the same format at in part 1). The 25 most common passwords are:

123456	12345	123456789	password	iloveyou
princess	1234567	rockyou	12345678	abc123
nicole	daniel	babygirl	monkey	lovely
jessica	654321	michael	ashley	qwerty
111111	iloveu	000000	michelle	tigger

Sort your results in the same way as in Part 1, and save them as `salt-cracked-sorted.txt`. Submit this file.

3.3 Cracking bcrypt?

The file `rockyou-samples.bcrypt.txt` contains bcrypt hashes of passwords. Bcrypt is a more modern hash function designed for use with passwords. Its primary feature is that a bcrypt hash is (comparatively) slower to compute, making brute force attacks far less effective than with the extremely efficient SHA-1 and MD5 hashes. The bcrypt hashes are stored in a standard format for bcrypt, and should be recognized by a bcrypt library of your choice. The hashes further automatically include a salt.

Write a program that finds the first five occurrences of the password `123456` by line number (counting from 1). Write each line number, in order, on a single line of the file `bcrypt-lines.txt`. Submit this file.