The Security 101 Virtual Machine

For this session, and for the following two labs, you may find it useful to work from a virtual machine which has been set up with the required tools (and not much else). If you have a Mac or Linux (or other Unix) OS, you might be able to use your machine directly so long as the required software packages are installed (but handling this would be up to you). For today, we require `openssh` and `gnupg`.

You can find the preconfigured VM from the course website on Blackboard. The link is under Materials, at the bottom of the page (below the course GPG key).

You will need to install and configure the VirtualBox hypervisor for your operating system in order to launch the VM. You can find this here: https://www.virtualbox.org/ or via your OS's package manager.

If you run into trouble getting VirtualBox or the VM to run, ask a TA for help!

A tour of cryptography on the internet

Start by opening https://wwwa.fen.bris.ac.uk/ , the home page of SAFE. You should hopefully get a green padlock symbol before the web address. We're going to investigate what cryptography is being used.

- If you're using *Google Chrome* or a similar browser (chromium, iron, …): click the padlock, then click "Details". A window opens. Press F5 to reload the page, then select the entry for the page (wwwa.fen.bris.ac.uk) under "Main Origin" in the new window. You will see among other things entries for *Protocol, Key Exchange* and *Cipher Suite*.
- If you're using *Mozilla Firefox*: click on the padlock or green button, then "More information" and "security" in the window that pops up. The "Technical details" section gives a string with the protocol, key exchange and cipher suite information (in that order).
- If you're using *Internet Explorer / Edge*: right-click the page and select "properties". The security information appears under "Connection".

Tasks:

1. What cryptographic protocols (protocol, key exchange, cipher suite etc.) are being used when you connect to https://wwwa.fen.bris.ac.uk ?
2. What about https://google.com ? (It might redirect you to google.co.uk, in which case look at that.)
3. Try the same on two more sites that you regularly visit. You might have to stick "https://" in front of the site name if it doesn't do this automatically.

Once you have seen a few cipher suites, look up the following terms. What do they stand for and which ones are public-key schemes (used for key exchange), which ones are block ciphers, which ones are modes of operation and which ones are hash functions?

1. RSA
2. GCM
3. AES (variants: AES-128, AES-256)
4. SHA (variants: SHA-1, SHA-2, SHA-256, SHA-384, SHA-512)
5. DHE and ECDHE
6. CBC
7. IDEA
8. DSS
9. Camellia
10. ECDSA

Which of these does your browser support? You can find this out by going to https://www.howsmyssl.com/ or https://www.ssllabs.com/ssltest/viewMyClient.html

(If you get a warning that you're insecure/vulnerable, you should probably update your browser and/or operating system at some point.)

An introduction to SSH

SSH is a protocol for remotely operating services in a cryptographically secure manner. Most often, you use it to log in to the command-line interface of a remote server in order to perform some operations on the server. We're going to have you log in to a server hosted within the department, called Snowy. First you'll do this using password authentication, and then you'll set up a public-private key pair to enable you to log in without using your university credentials.

Using a password

Type
`$ ssh <username>@snowy.cs.bristol.ac.uk`
where `<username>` is your university username. When you hit return, you will be prompted for your password. Type this in (note: you won't see the password echoed back) and press return again to log in.

You can look around your home directory on Snowy and see what's already there. Make sure to check for hidden directories with `ls -a` – what do you find, and what is it for?

For what we're going to do next, we're going to need a directory called '`.ssh`' in your home directory. You can create this with `mkdir -p .ssh` (make sure to include the . before the ssh).
Once you're done looking around on Snowy, log out with `exit`

Generating a key

Now you're going to generate a public-private key pair to use with SSH. Later we'll do this again with GPG, which is a little redundant, but the process is mostly the same. At the terminal, type: `ssh-keygen`

You'll see a prompt like the below:

```
Generating public/private rsa key pair.
Enter file in which to save the key (/home/student/.ssh/id_rsa)
```

You can just press enter here, and SSH will use the default file name and location to store the private key. Next you will be asked for a passphrase to protect your private key. You'll need to use this passphrase to unlock the key whenever you need to use it, so make sure it's memorable – or stored securely in a password manager.

You may have to wait a while for the key generator to gather enough entropy – if this happens, just do something else on your computer in the interim.

Eventually, you should see a message informing you of where your private key was saved (the path specified above) and where the matching public key was saved (typically the same

place plus '.pub'). You may also see a fingerprint and a visual randomart image, both of which are meant to help you identify your key.

You now have a public and a private SSH key pair. Your private key should always be kept private. But to use your public key to authenticate yourself, you'll need to share it. Specifically, you need to tell Snowy what your public key is, so it can encrypt a challenge which only you (using your private key) should be able to decrypt and answer.

To do this you need to first log in to Snowy using password-based login, and then add your public key to a file of authorised public keys that Snowy's ssh service can use when someone attempts to log in to your account. Using some features of SSH, you can do all this at once by typing the following into your local machine, followed by your university password at the prompt:

```
cat ~/.ssh/id_rsa.pub | ssh <username>@snowy.cs.bris.ac.uk 'cat
>> .ssh/authorized_keys'
```

Discuss what is happening here!

Everything going to plan, if you now try to log in, as before:

```
ssh <username>@snowy.cs.bristol.ac.uk
```

You'll get a new prompt, this time asking for the passphrase you set to protect access to your private key. SSH needs you to supply this so it can use your private key to decrypt and respond to a challenge from Snowy encrypted with the public key stored in the `.ssh/authorized_keys` file. If you type in your passphrase, you should authenticate and log in using your key pair.

If you want more details on using SSH with Snowy, including how to get a graphical session, see the CSS guidance at:

https://cssbristol.co.uk/tutorials/ssh-into-snowy/

Your SSH key can be used to authenticate you with a number of services in a similar manner. You could share your public key with services like Github, to avoid having to type out passwords when pushing commits (see: https://help.github.com/en/github/authenticating-to-github/connecting-to-github-with-ssh)

GPG on the command line

1. Check that GPG is installed

Type `gpg --version` at the command line to see if it is already installed, which is the case for all linux distributions I know of. You should get something like this:

```
$gpg --version
gpg (GnuPG) 1.4.16
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Home: ~/.gnupg
Supported algorithms:
Pubkey: RSA, RSA-E, RSA-S, ELG-E, DSA
Cypher: IDEA, 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH,
        CAMELLIA128, CAMELLIA192, CAMELLIA256
Hash: MD5, SHA1, RIPEMD160, SHA256, SHA384, SHA512, SHA224
Compression: Uncompressed, ZIP, ZLIB, BZIP2
```

2. Create a keypair

Type `gpg --gen-key` to create a keypair. It may first ask you which type: either "RSA and RSA" or "DSA and Elgamal" are okay. For a key size, choose at least 2048, which gets you around 128 bits of security depending on who you ask. You don't need to set an expiry date.

For "real name" and other things later on you can put whatever you want. Then you'll have to choose a passphrase and enter it twice. Sometimes, gpg will complain about entropy - in this case just leave it running for a while.

When your key is ready, try `gpg --list-keys` and you should see something like this output:

```
$gpg --list-keys
/home/cosc/csxdb/linux/.gnupg/pubring.gpg
------------------------------------
pub   3072D/3F4D9401 2016-08-11
uid                  Security 101 <matthew.john.edwards@bristol.ac.uk>
sub   3072g/94121854 2016-08-11
```

Of course there's also `gpg --list-secret-keys`. All your keys are stored in keyring files, normally in a hidden directory `.gnupg` under your home directory. The public keys are in `pubring.gpg` and the secret ones in `secring.gpg`.

To let someone send you encrypted messages, you need to give them a copy of your public key. Of course, you never give anyone your **secret** key - there's a clue in the name!

3. Exchange keys

Type `gpg -a --export` to export your keys. It will print them to the console, you can do `gpg -a --export > keys` to put them in a file named `keys`. The export command only exports public keys. If you have lots of keys, you can do `gpg -a --export [names...]` to export only some of them - you can use either names, e-mail addresses or key IDs (the 8-digit number in the second half of 2048D/34C4360E from the list keys command).

Find another student and exchange keys with them (you can email each other the keys--they're public keys after all).

To import a key, either type `gpg --import` and paste the key in the terminal then press enter and Control-D (on windows: Control-Z) to complete, or if it's in a file do `gpg --import [filename].`

Download and import the public key from the course website. You should see something like this:

```
gpg: key C20568FD3FD7A820: public key "Security 101
                        <matthew.john.edwards@bristol.ac.uk>" imported
gpg: Total number processed: 1
gpg:               imported: 1
```

4. Sign the keys you trust

Before you can encrypt with a key, you must tell GPG that you trust it. This is the (in)famous "web of trust" that Phil Zimmermann came up with.

First do `gpg --list-keys` to show the keys, then do `gpg --edit-key [name]` where name can again be a key ID or a name or email address included in the key. You only need to type a partial match so `gpg --edit-key security` should get the security 101 key.

You are now in the key editor and will see something like this:

```
pub  3072D/3F4D9401  created: 2016-08-11  expires: never       usage: SC
                     trust: unknown       validity: unknown
sub  3072g/94121854  created: 2016-08-11  expires: never       usage: E
[ unknown] (1). Security 101 <matthew.john.edwards@bristol.ac.uk>

Command>
```

Note the fields *trust: unknown* and *validity: unknown*. To mark a key as valid, you need to sign it with your own secret key.

At this point, you check that this key really comes from who it claims to. Type "fpr" and press enter to show the key fingerprint. Our Security 101 key has the following fingerprint:

9E8C 9C6F CC53 935F 1BF1 4E01 C205 68FD 3FD7 A820

Compare the key fingerprint that is shown on your console to the one in this document. For a key that you got from the student next to you, have them type `gpg --fingerprint` into their own console, check this against yours and make sure that they match. Then ask them to confirm that it really is their own key!

Once you're happy that the key really is the correct one, type "sign" and then confirm with y to sign the key - you'll be asked for your secret key password. Then type "quit" to close the key editor.

5. Trust

Signing a key marks it as "valid" so you can use it to encrypt things. The web of trust reduces the number of times you need to personally check a key: if you have a signed copy of A's public key, and a copy of B's public key signed by A, then you can tell GPG that you trust A enough that you're happy to accept B's key as well.

In the key editor, you can type "trust" and select an option from the menu. At this point, like everyone else who encounters this for the first time, you might wonder what the difference is between "I trust fully" and "I trust ultimately", or whether someone is taking liberties with the English language here.

The general idea - you can of course tweak the parameters - is that a key is allowed to be used for encryption if it satisfies any of the following:

- You have personally signed the key.
- Someone you trust "fully" has signed the key.
- At least three people that you trust "marginally" have signed the key.

There is also a default restriction on the length of paths for trusting keys to at most five: if you (A) sign B's key, B signs C's key and so on and everyone fully trusts the nest in the chain then GPG will accept keys up to E; for F it will complain that the chain is too long.

If you have several keys of your own, you can set them to "ultimate" trust (PGP used to call this "owner" trust). This works like "full" trust but ultimately trusted keys always count as depth 0 for the path-length rule, so they can have chains of five valid keys starting from themselves. Your own keypair that you created above will be automatically set to ultimately trusted.

6. Encrypt a file

Create a file with some text in it. You can now encrypt it with the command

```
gpg -r NAME --encrypt FILE
```

substituting the key name (try "security") and file name. This will create a file `FILE.gpg` with the ciphertext. Ciphertext files are binary i.e. may contain arbitrary bytes; you can also add the `-a` option before `-r` to produce base64-encoded ciphertexts if you want to send an

encrypted e-mail. In this case the file will be called FILE.asc by default (you can use --output OUTFILE to select a different name).

You can use multiple `-r` options to produce a ciphertext that can be decrypted by more than one key. Note that encrypting alone does not require any passphrase as you are only using the public key of the recipient.

7. Decrypt a file

Encrypt a file with another student's key and send it to them by e-mail. To decrypt a file you got this way, run `gpg --decrypt FILE.` You will be asked for your password, then the file will be saved to the directory in which you called the decrypt command.

8. Encrypt and sign

To sign a file as well as encrypting it, add `--sign` after `--encrypt` and before the file name. You will now be asked for your passphrase to create the signature. You can also sign files without encrypting them by just using `--sign` in which case the `FILE.gpg` file will just contain the signature and you need the original file too to verify it. Like with all gpg operations, `-a` produces base64 encoded output.

When you decrypt a file that is also signed, GPG will display a message such as

```
gpg: Signature made Tue 16 Aug 2016 09:53:23 BST using DSA key ID 3F4D9401
gpg: Good signature from "Security 101<matthew.john.edwards@bristol.ac.uk>"
```

If you want to verify a signature on a non-encrypted file, use `gpg --verify SIG` where you replace SIG with the name of the file containing the signature.