CMP5329 Logbook
DRAFT VERSION

Lewis Higgins - Student ID 22133848

January - March, 2024

Contents

Introduction

This logbook documents the work completed and knowledge gained across the CMP5329 labs, showcasing the use of a wide variety of security techniques and access control methods on a Linux OS. This logbook specifically covers the following labs:

- Lab 1, covering OpenSSL.
- Lab 2, covering simple usage of GPG.
- Lab 5, covering the use of Linux Discretionary Access Control commands.
- Lab 6, covering password cracking.

As per module specifications, screenshots taken in each lab include the date and time at which they were taken.

Example note

Additional notes, such as minor issues encountered or omitted screenshots due to work having been done in earlier labs, are documented using these orange notes.

Example important note

Critical issues that required special workarounds are documented using these red notes.

OpenSSL

This lab was an introduction to the usage of a Linux (Ubuntu distro) VM on a host machine, and the usage of OpenSSL to encrypt and decrypt data using the (outdated) DES algorithm and AES256 symmetric encryption algorithm. Additionally, this lab looked at RSA private keys used in asymmetric encryption, and how to generate and gather public and private keys, alongside message digests.

1.1 Version checking and ciphers

To check the installed version of OpenSSL, the command "openssl version" can be used. The provided virtual machine from the CMP5329 Moodle page uses OpenSSL version 1.1.1f, dated 31st March 2020.[1]

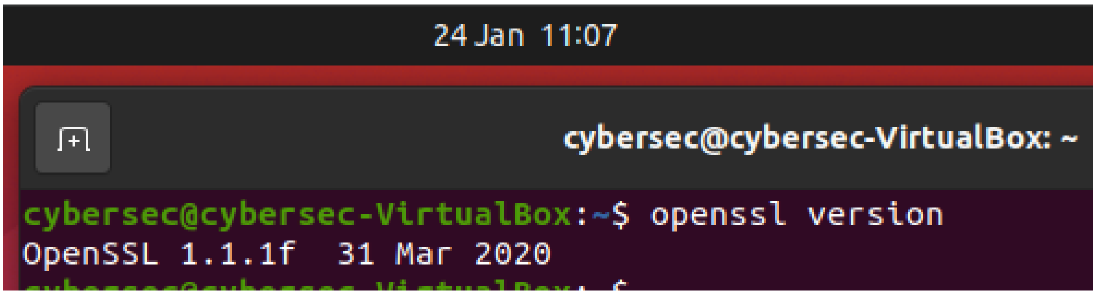

Figure 1.1: Getting the OpenSSL version

The list of OpenSSL ciphers can be viewed by executing the "openssl ciphers" command.

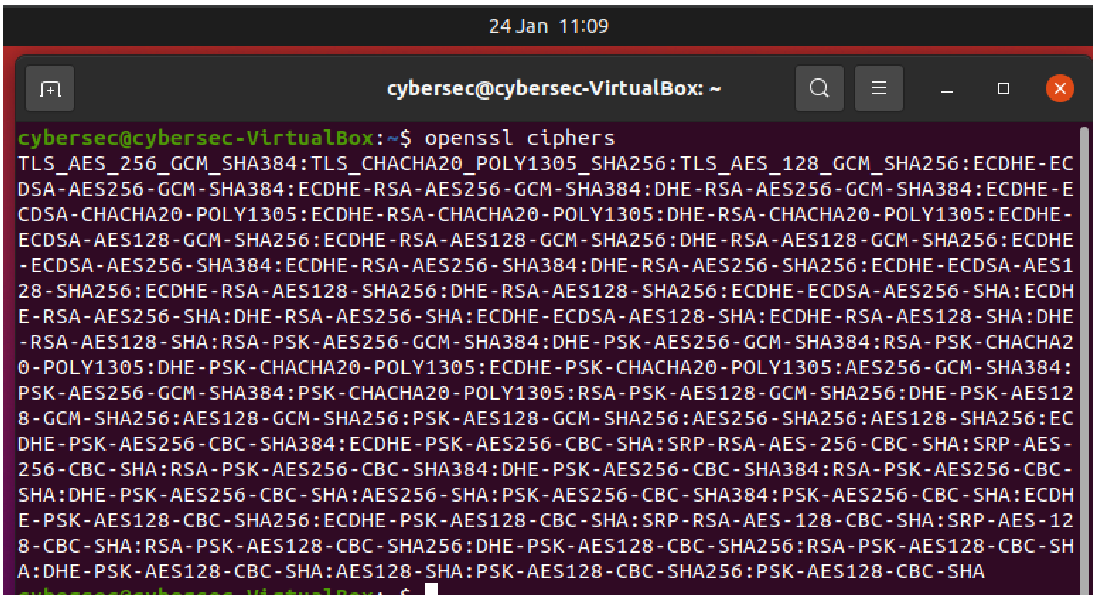

Figure 1.2: Getting the OpenSSL ciphers

[1]This is a heavily outdated version of OpenSSL, however I have continued to use it due to it being part of the module-provided resources.

1.2 Symmetric encryption

Symmetric cryptography refers to the process of transferring data that has been encrypted by a single key. Both the sender and receiver of this data use the same key to encrypt and decrypt the data. Symmetric encryption does not allow for non-repudiation, as it cannot necessarily be proven that data came from one sender and not someone else with the key, as there is no attached signature to prove the sender's identity.

1.2.1 DES symmetric encryption

OpenSSL can be used to encrypt plaintext into unreadable text known as ciphertext. Many algorithms exist to generate ciphertext, but for this example, the Data Encryption Standard (DES) symmetric encryption algorithm will be used. This is performed using the command
"openssl enc -des -k secretkey -a". [2] This command can be broken down into:

- enc - Encode
- -des - Using the DES algorithm
- -k secretkey - Using the key "secretkey"
- -a - Writes the ciphertext in base64/ASCII, allowing it to be properly displayed. If this flag isn't used, the text will not be legible, showing question mark symbols instead.

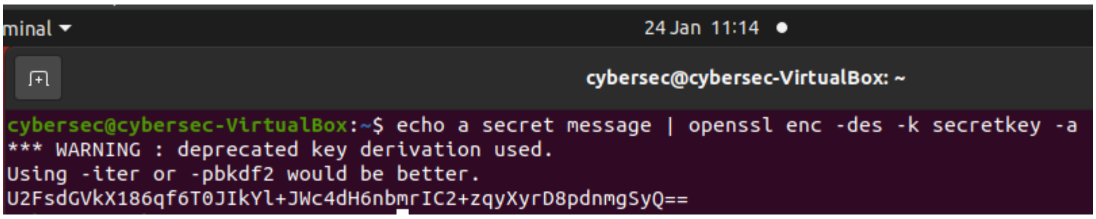

Figure 1.3: Converting "a secret message" to ciphertext using DES with key "secretkey".

This ciphertext can then be decoded if you know the key it was encoded with.

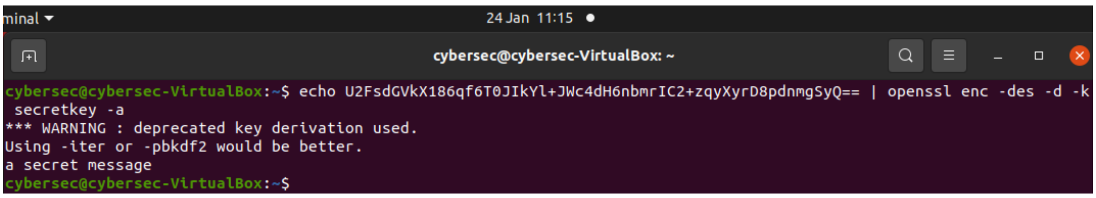

Figure 1.4: Decoding the ciphertext back to its original form using the key "secretkey".

This command is the same as in Figure 1.3, with two alterations:

- The ciphertext is passed to OpenSSL rather than the plaintext, as we typically wouldn't know the original plaintext in this scenario, which is what we are aiming to achieve by decrypting it.
- -d - Indicates that the text should be decrypted rather than encrypted.

[2] "echo a secret message | " passes the phrase "a secret message" as the text to be encrypted.

1.2.2 AES256 symmetric encryption and decryption

The DES algorithm is considered weak to today's standards due to how simple it is to brute-force using today's processing power. Because of this, newer algorithms were developed, with one of these being AES. AES256 refers to the key size, which is 256 bits for this variant, though you can use key sizes of 128 or 192 bits as well, which would be weaker.

I researched how to use this algorithm in OpenSSL, eventually finding this help page (Heinlein, 2016) which provided details on encrypting text using the AES-256-cbc cipher.

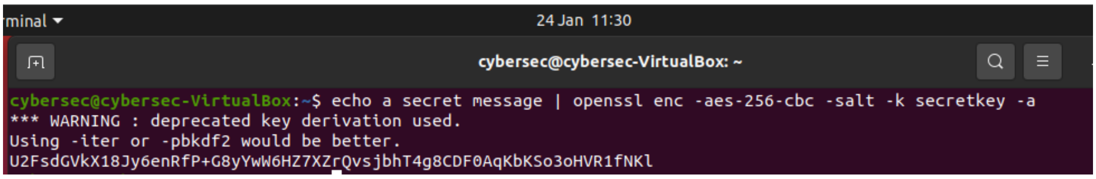

Figure 1.5: Encoding the plaintext with AES-256-cbc using the key "secretkey".

In this command, the -des flag is instead replaced by -aes-256-cbc, indicating the cipher to use, and the optional -salt flag was added, which salts the text to provide different ciphertext. Salting is the process of adding random data to the end of the text prior to encoding it, which will change the resulting ciphertext, making it harder to decrypt and increasing the strength of the encryption.

1.3 Asymmetric encryption

Asymmetric cryptography is the practice of using two keys when transmitting data: a public key used to encrypt data, and a private key used to decrypt it. For example: John is sending Alex an encrypted message. The message is encrypted using Alex's public key and sent to Alex. When Alex receives the message, it is decrypted using Alex's private key. Alex can prove that John sent this message because of the digital signature attached, which is generated from John's private key, and then verified using his public key. John does not ever know what Alex's private key is, nor does Alex know John's private key.

Data transferred this way has a digital signature attached, which allows for non-repudiation, as it cannot be denied that the data originated from the device with said signature. The signature is validated using the sender's private key.

1.3.1 Generating an RSA private key

OpenSSL can be used to generate these keys by using the "openssl genrsa" command.

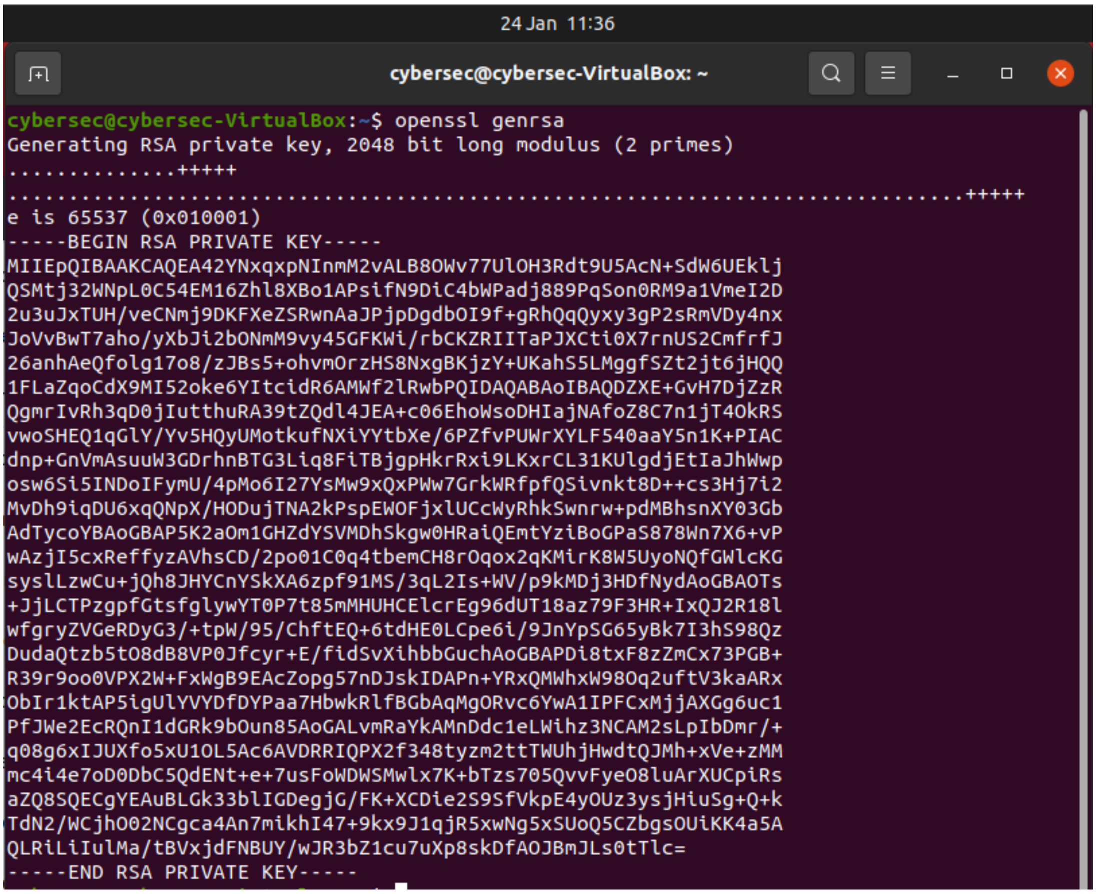

Figure 1.6: Generating a 2048-bit private key using genrsa.

This works to generate a private key, but this command merely outputs it to the console, and does not actually store it anywhere on the system.

1.3.2 Storing DES3 & passphrase encrypted RSA keys in a file

The key generated can then be encrypted using an encryption algorithm, which would be DES3 for this example, and a passphrase. Upon doing this, the key can be stored into a .pem file, also known as a Privacy Enhanced Mail file, which is a file format 'to provide the creation and validation of digital signatures, and in addition the encryption and decryption of signed data, based on asymmetric and symmetric cryptography.' (Kolletzki, 1996, p. 1894)

In this example, a 1024-bit key is created using DES3 and the passphrase [3] "secretkey".

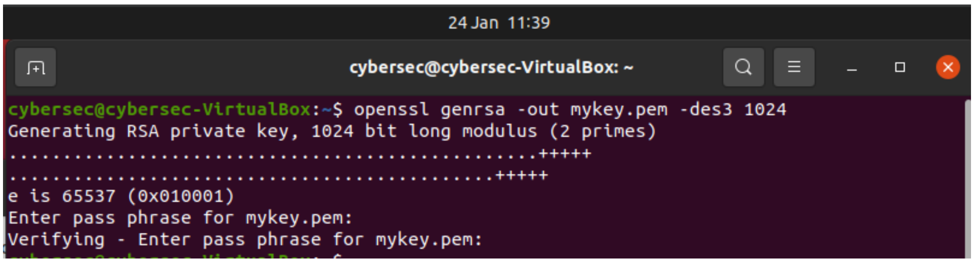

Figure 1.7: Generating and storing a 1024-bit private key using genrsa, DES3 and the passphrase "secretkey".

To break down this command:

- genrsa - Generate an RSA private key.
- -out mykey.pem - Save the generated key to the file "mykey.pem" in the current directory.
- -des3 - Using the DES3 encryption method.
- 1024 - The key will be 1024 bits.

We can then view this key by entering the command "more mykey.pem".

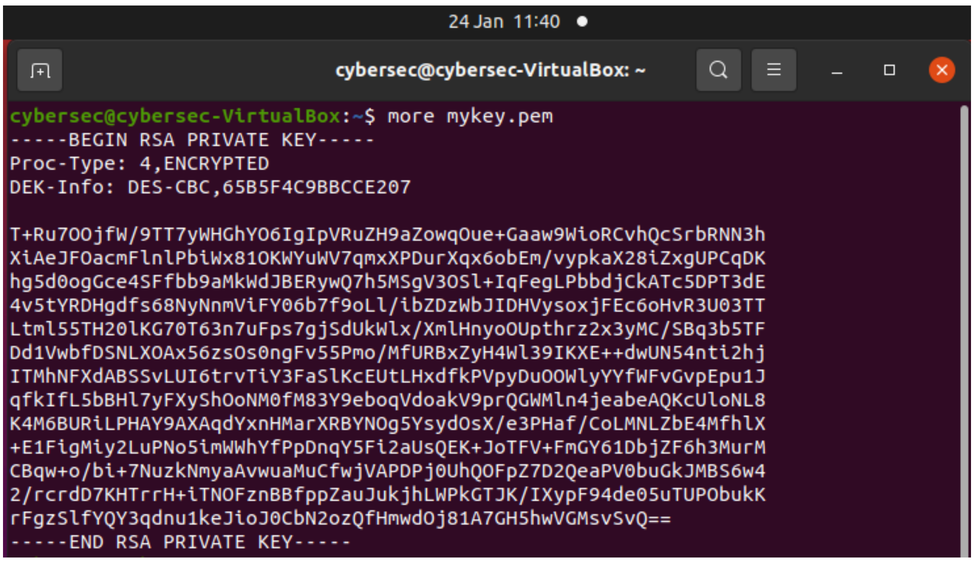

Figure 1.8: The key stored in mykey.pem.

[3]The passphrase is not visible when typed, and therefore does not show in the image.

1.3.3 Getting a public key from the private key

The private key stored into "mykey.pem" by the previous command can be accessed again to generate a public key, which would be used when data is encrypted. To do so, the passphrase we set earlier (secretkey) must be entered.

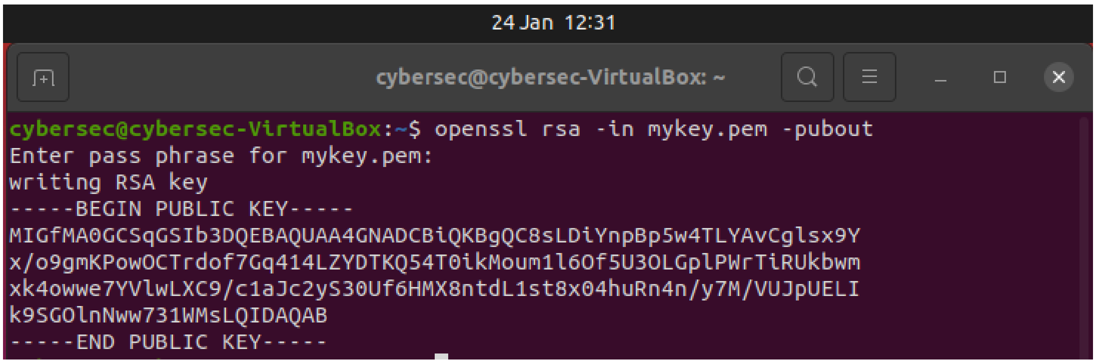

Figure 1.9: The public key generated from mykey.pem.

Breaking down this command, "-in mykey.pem" uses mykey.pem as the input for the command, and "-pubout" outputs the public key generated from the private key. However, this command only outputs said key to the console, so an altered version is necessary to save the key to a file.

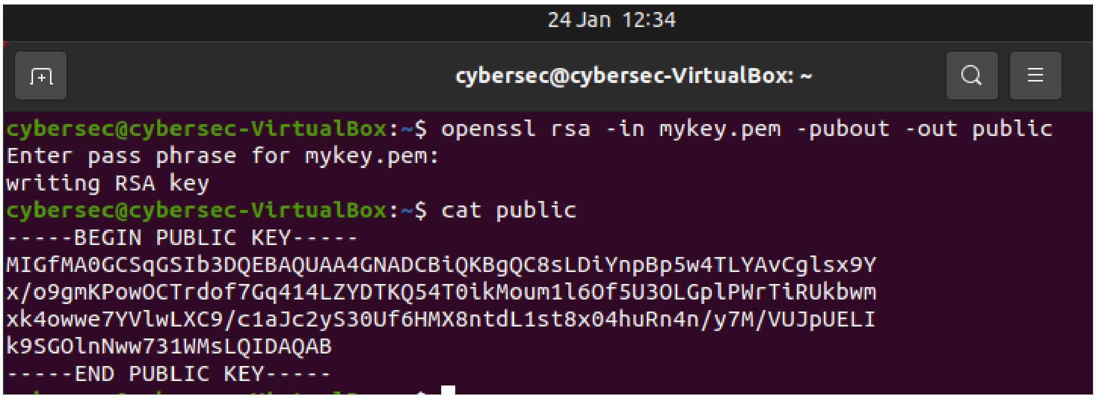

Figure 1.10: Storing the public key in a file.

The additional argument "-out public" is set in this command, meaning the public key will be written to a file called "public". This file can then be read using the cat command, revealing the public key seen in Figure 1.9.

1.3.4 Obtaining a message/file digest

When data is transmitted, it could become corrupted or potentially even intercepted and modified before it reaches its intended recipient. To mitigate the risks from this, files can have "digests", which are the result of hashing their contents. If the file is modified whatsoever, even by a single byte, the digest would be different, meaning the file has been corrupted or tampered with. These are also the basis of digital signatures; by encrypting a digest with your private key, you cannot deny that you were the source of the file.

OpenSSL can generate digests using its "dgst" command.

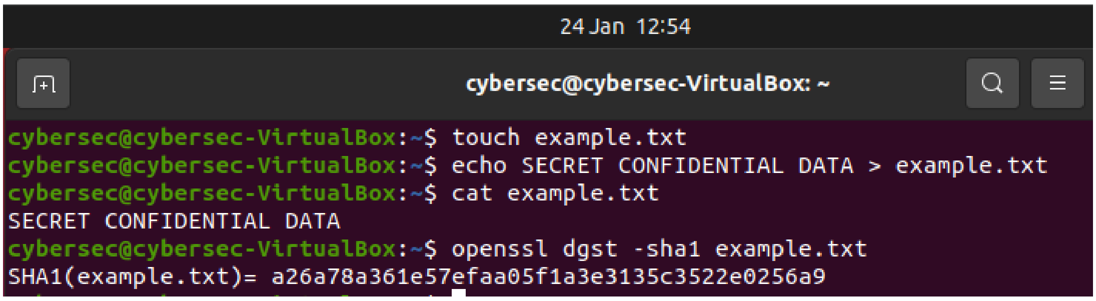

Figure 1.11: Creating a file, then getting the SHA1 digest of it.

This can also be verified by using a non-OpenSSL command, sha1sum, which returns the same digest as the OpenSSL dgst.

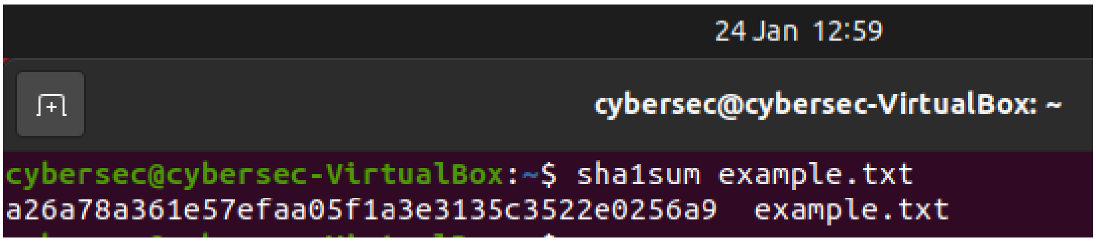

Figure 1.12: Verifying the digest.

1.3.5 Signing a digest

Non-repudiation is an important topic in cybersecurity - verifying the original sender of a file is an imperative task in the event of any issues that may arise such as malware being appended to the file, in either the prosecution or defense of the alleged sender. Signing a message digest using your private key definitively proves the device that data was sent from, meaning that it cannot be denied that the file was sent, nor who it was sent by.

The previously used "example.txt" can again be used here to generate a digest encrypted using the "mykey.pem" private key established earlier, which signs the digest.

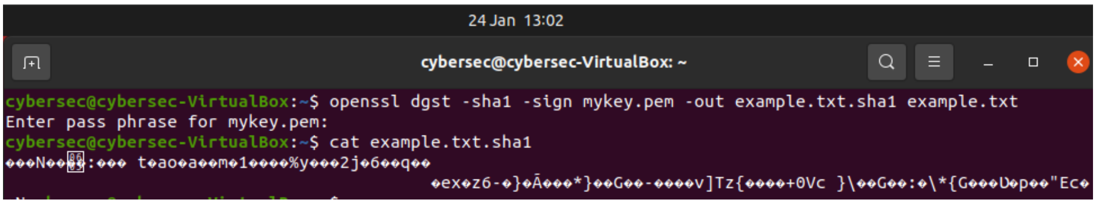

Figure 1.13: Writing a signed digest to a file.

Note that when we try to read this file, it is completely illegible, as it is not in a compatible format. To counteract this, it can be converted to Base64 using OpenSSL's "enc" command, passing the generated file as the argument.

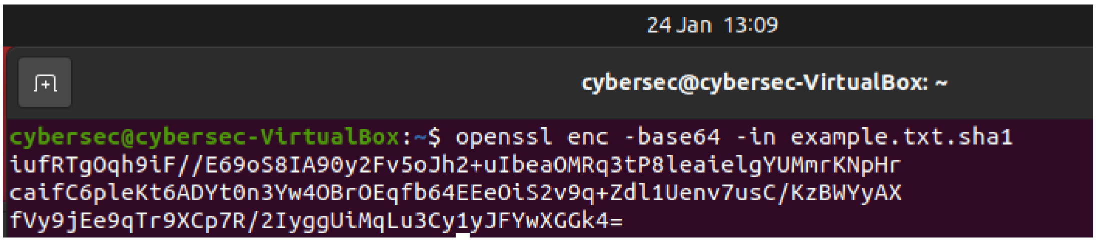

Figure 1.14: Encoding the signed digest to Base64.

This doesn't have any use other than allowing us to see the key in a Base64 format - the key functions even if it can't be conventionally read.

Now that we have the signed digest, it can be verified using the public key, which confirms the authenticity of the data in example.txt.

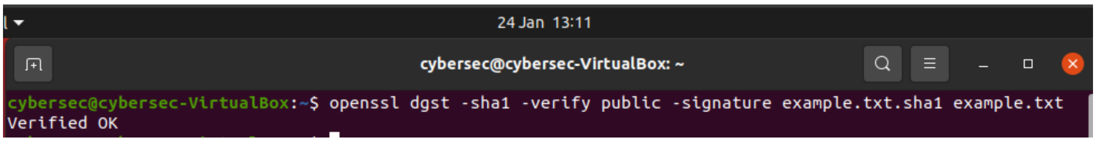

Figure 1.15: Verifying the signature of example.txt.

This returns "Verified OK" as intended, as the file has not been modified. If the file does get modified through either corruption or a threat actor, the digest would not be the same, which can be verified by modifying the file ourselves and then verifying the digest of the file once again.

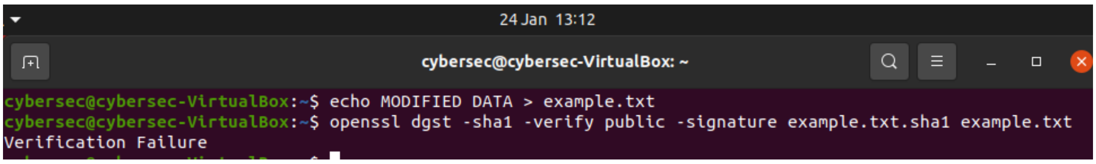

Figure 1.16: Failing to verify the signature of example.txt, as it has been modified.

This returns "Verification Failure", as the digest would now be different due to example.txt now containing different data than it did when the SHA1 digest was created.

Usage of GPG

Important note

In this lab, an incompatibility meant that instead of using the base GPG program, the alternative **GnuPG1** was used. Therefore, commands use the phrase "gpg1" instead of "gpg".

This lab expanded on the concepts of asymmetric encryption through the use of GPG/GnuPG (GNU Privacy Guard) to produce, sign and verify public and private keys.

2.1 Creating test users

For this lab, two test users were created and used to execute the necessary commands.

2.1.1 Elevating the terminal

To add users to the system, administrative privileges are required. To gain the necessary privileges, the command "sudo -s" or "sudo bash" can be entered (both commands are functionally identical) which will change the terminal to be at root level.

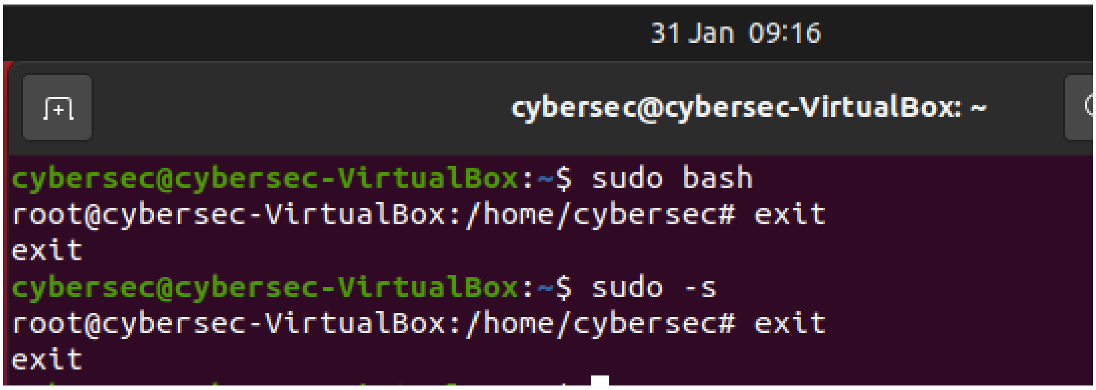

Figure 2.1: Elevating the terminal.

2.1.2 Creating Bob and Alice

With the elevated privileges gained from being a superuser, it is now possible to add users to the system using "adduser" followed by the given username. A password for the user will then be necessary, followed by optional information such as phone numbers, which are left blank for this lab.

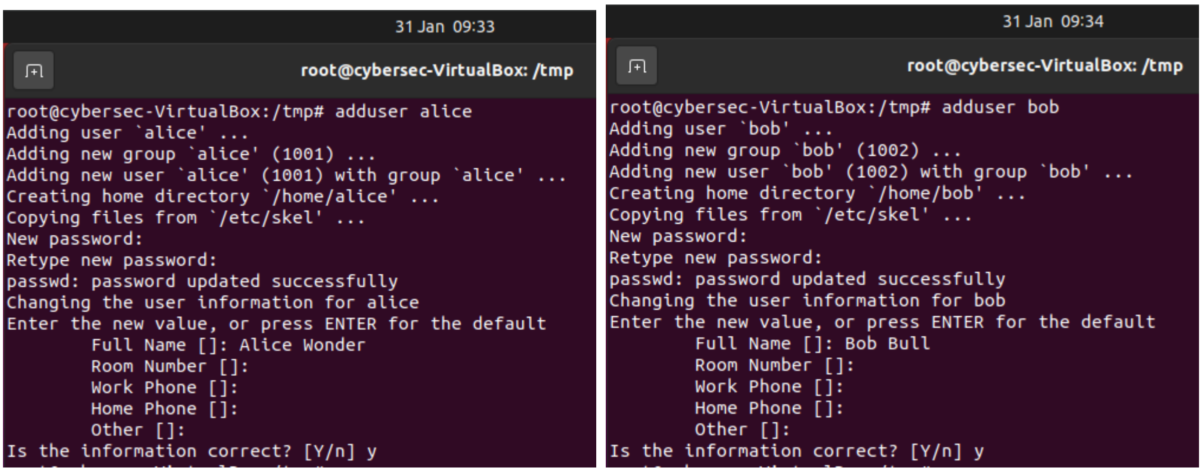

Figure 2.2: Creating users 'bob' and 'alice'.

For ease of access, multiple terminal tabs can be open at a time, so I elected to use one for the superuser root, and one each for Bob and Alice.

Figure 2.3: Multiple terminal tabs.

I also added these new users to the "sudo" group, allowing them to also use the sudo command to execute commands with elevated permissions.

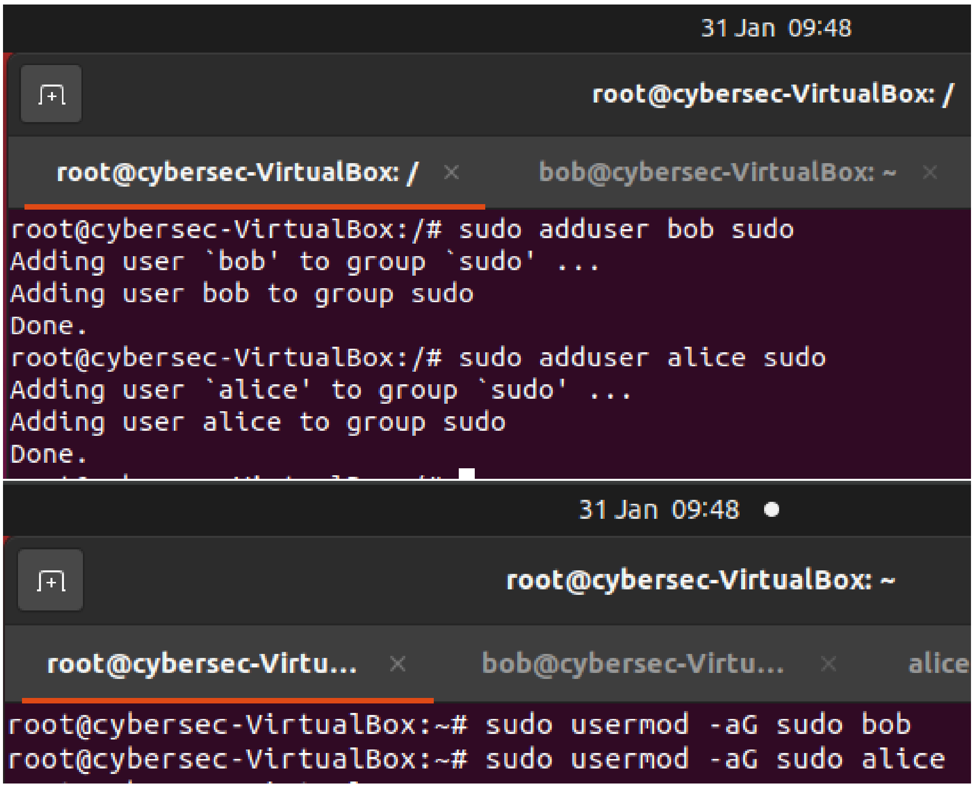

Figure 2.4: Adding bob and alice to sudo.

It is possible to switch the active terminal user using the command "su" followed by the account to switch to, and then the password of the given account.

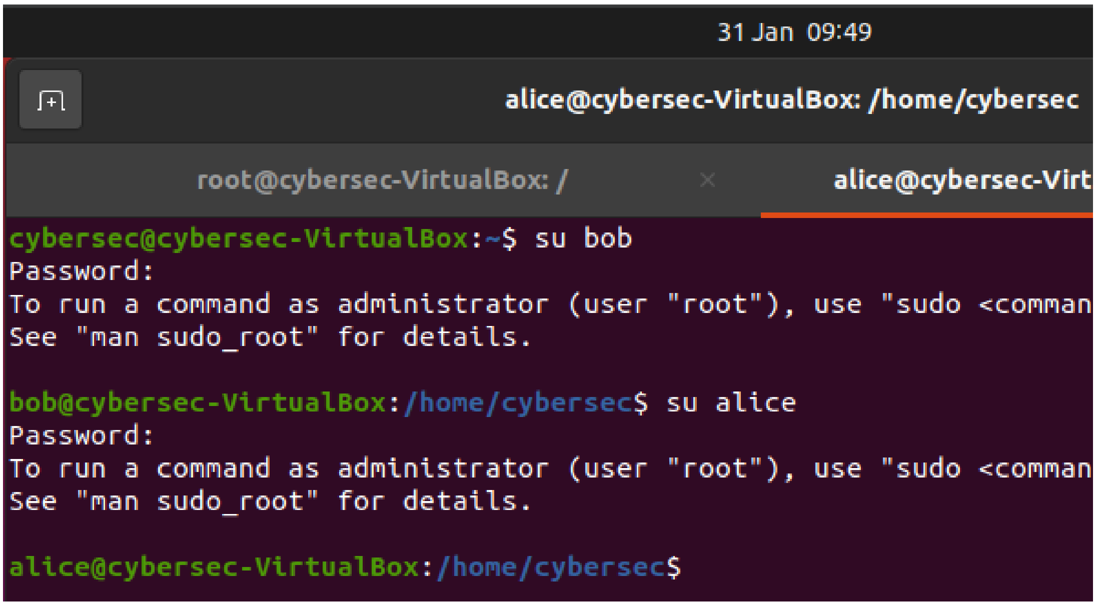

Figure 2.5: Switching the active terminal user. Note the prompt about running commands as an administrator, which signifies that they were successfully added to the sudo group.

2.2 Exchanging encrypted files over an insecure channel

For this section, assume that all commands have been executed on **both** the Bob and Alice user accounts unless stated otherwise.

On standard Linux distributions, the /tmp directory is a public directory accessible to all users. For this reason, it is therefore insecure, as every user on the system can read the files placed there.[1] To transfer files across insecure channels such as /tmp/, they should first be encrypted so that they can only be read and/or used by their intended recipient. Therefore, GNU Privacy Guard (GPG hereafter) can be used to generate and store public and private asymmetric keys.

2.2.1 Generating public/private key-pairs

To generate a private key, the command "gpg1 –gen-key" can be used.

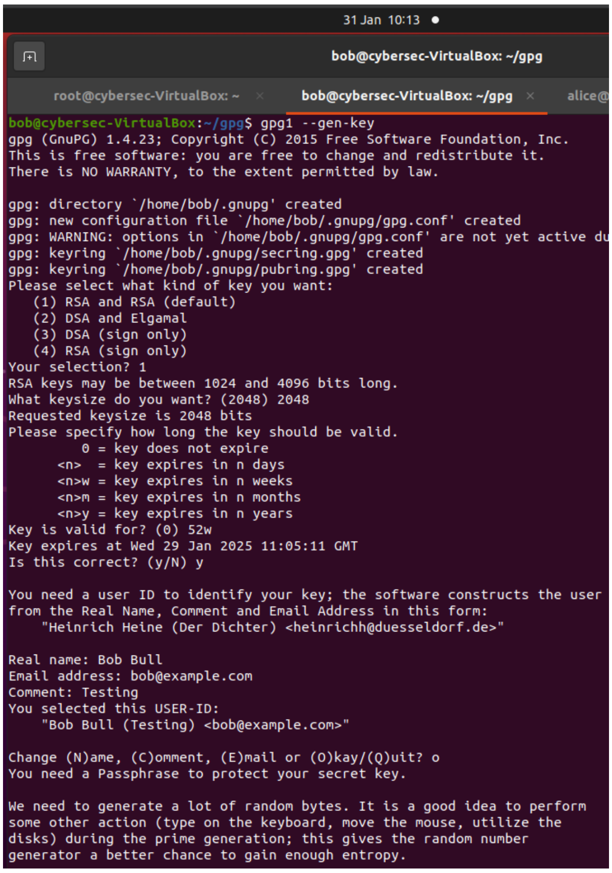

Figure 2.6: Generating a private key.

This will open a submenu where the user can select the kind of key they wish to generate, as well as the size and expiry date of the key. Once this is established, they must create a user ID if they didn't already have one, consisting of their full name, email address and an optional comment. While the key generates, the user is prompted to perform random inputs such as moving the mouse and typing on the keyboard to enhance the randomness of the generated key. A key was also generated for Alice.

[1]However, they cannot update/change them without sudo permissions.

2.2.2 Exporting public keys

It is possible to export the public keys from the generated key-pairs using GPG's export command.

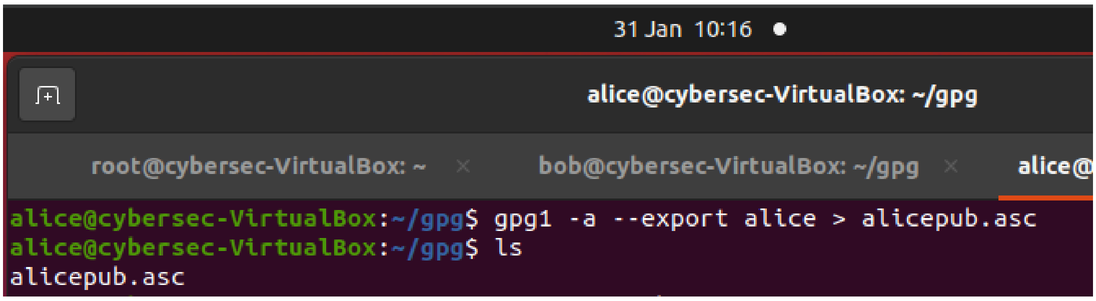

Figure 2.7: Exporting Alice's public key.

This exports the public key in ASCII format (due to the use of the -a flag) to the file "alicepub.asc". This can be seen by using "ls" to show the files in the directory.[2] Because this is Alice's **public** key, we are comfortable sharing this to the public /tmp/ directory where all users can see it.

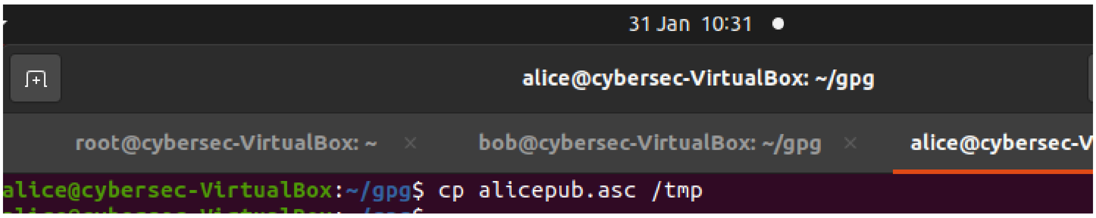

Figure 2.8: Copying Alice's public key to /tmp.

2.2.3 Importing and signing public keys

Now that Alice's public key is in /tmp, Bob can copy this to his own directory and import it using GPG.

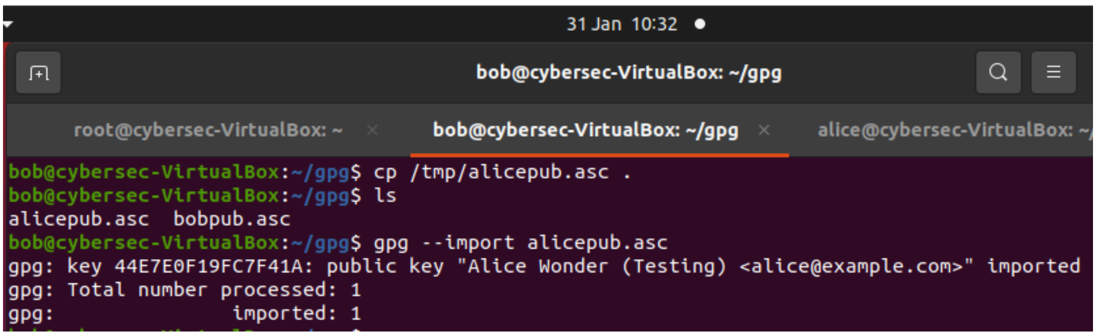

Figure 2.9: Importing Alice's public key as Bob.

[2]The file can be read using "cat alicepub.asc", but it is a 2048-bit key, so it would completely fill the terminal window.

Bob can then **sign** this key, verifying that he trusts that this key does belong to Alice. This is done by editing Alice's key as Bob, signing it, and then saving this.

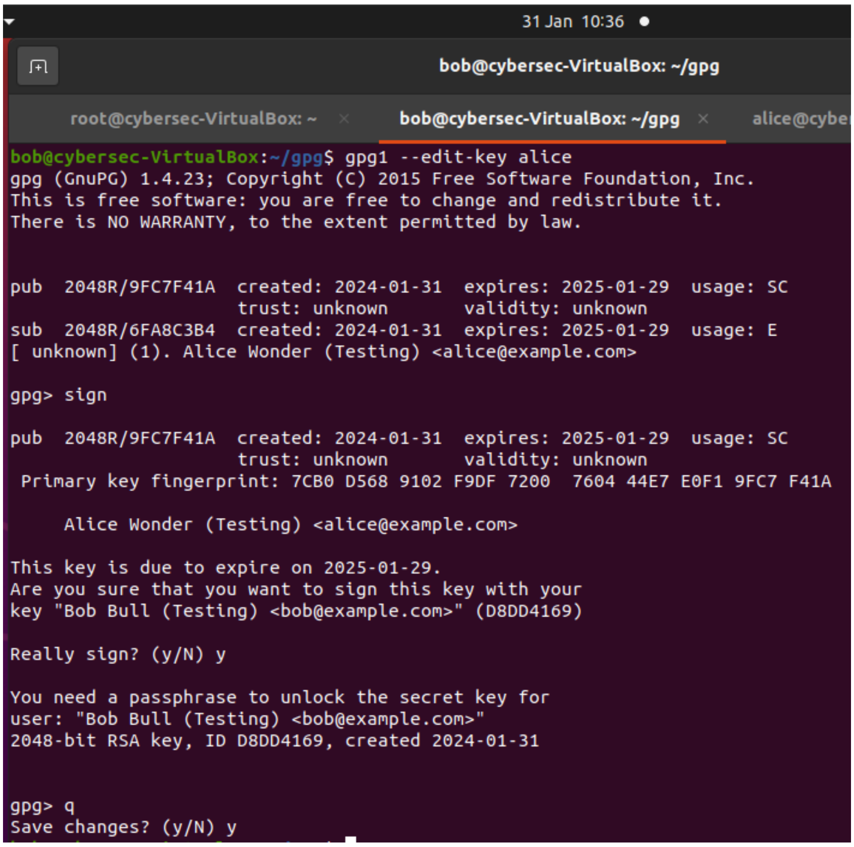

Figure 2.10: Bob signing Alice's public key.

2.2.4 Encrypting and decrypting data

Now that Alice and Bob have their key-pairs generated, they can transfer asymmetrically encrypted data to each other. This was tested by making a file, encrypting it using Alice's public key, and copying it to the /tmp directory.

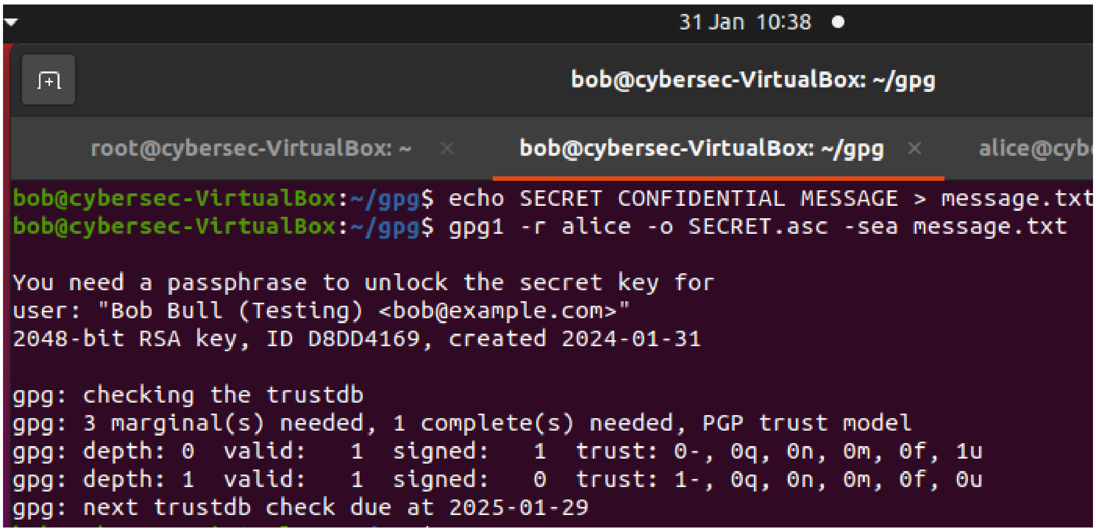

Figure 2.11: Making a file and encrypting it using Alice's public key.

This command can be broken down to its components:

- -r alice - Sets Alice as the recipient of the file by using her public key.
- -o SECRET.asc - Outputs the encrypted data to SECRET.asc.
- -sea message.txt - Sign and encrypt the contents of message.txt in ASCII format.

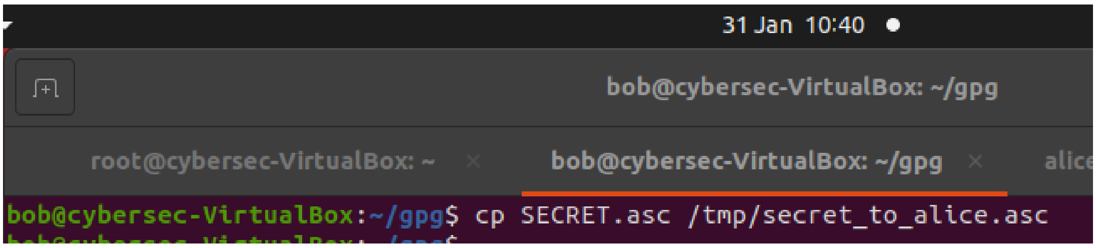

Figure 2.12: Copying the encrypted file to /tmp with the name "secret_to_alice.asc".

Alice can then decrypt "secret_to_alice.asc" and output the results to "message.txt", where they can then be read in human-legible form.

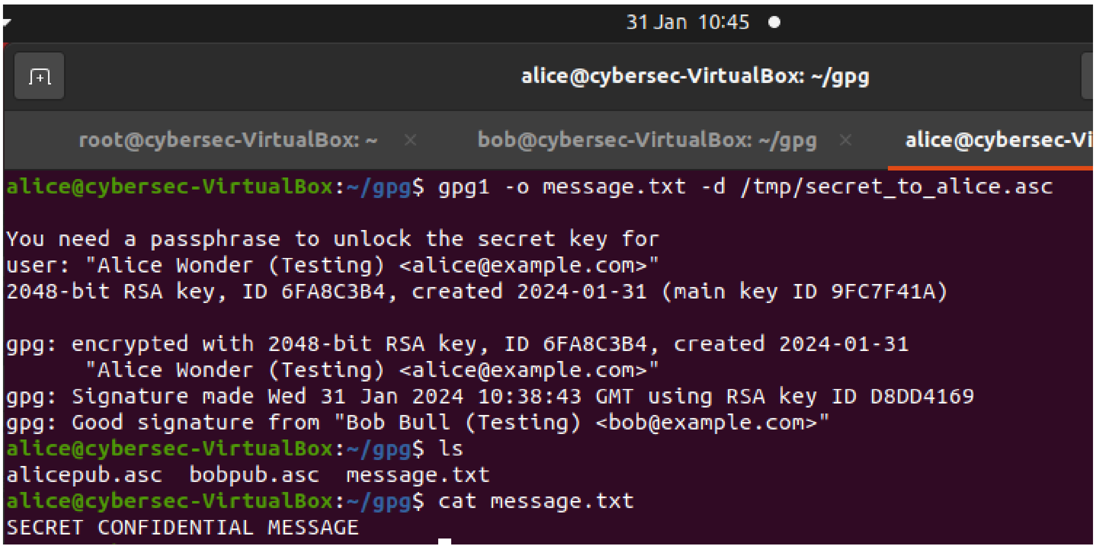

Figure 2.13: Decrypting the encrypted message and reading it.

Discretionary Access Control

This lab explored the use of Discretionary Access Control methods on a Linux system, which allows the owner of an object to assign the level of access that other entities will have to said object.

5.1 Creating test users and groups

Note

Creating users was already showcased and explained in further detail in Lab 2, specifically in figures 2.1, 2.2 and 2.4 of section 2.1.

For this lab, three test users "Pete", "Ali" and "Mary" were added to the system. Pete and Ali were assigned to the "Boys" group, whereas Mary was assigned to the "Girls" group.

5.1.1 Creating groups

With sudo privileges, additional groups can be added to the system.

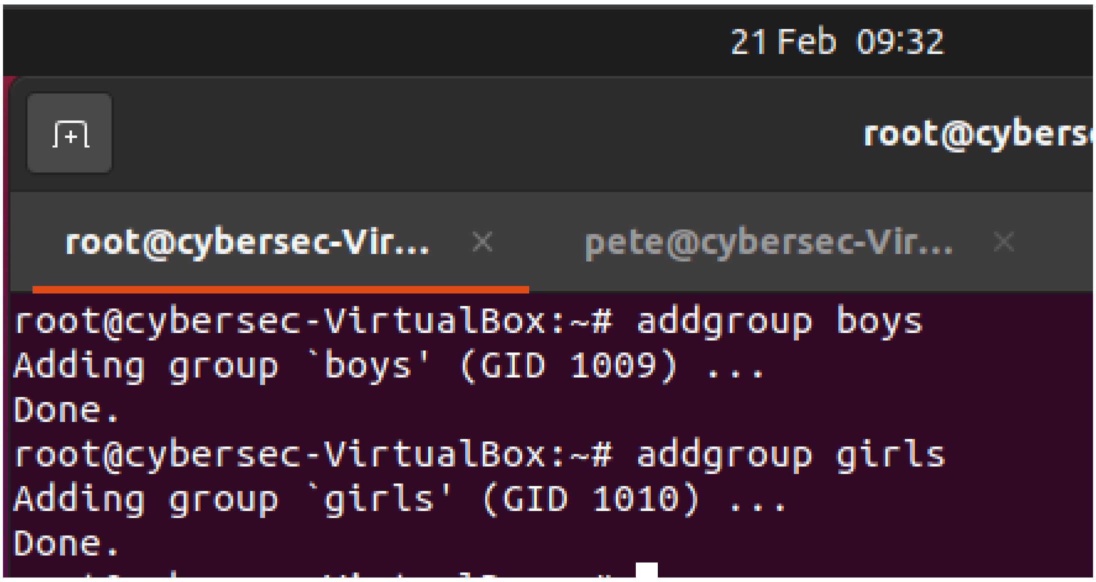

Figure 5.1: Making the "boys" and "girls" groups.

5.1.2 Adding users to groups

The new users were added to the groups mentioned above as well as the sudo group, enabling their user accounts to run commands with sudo privileges.

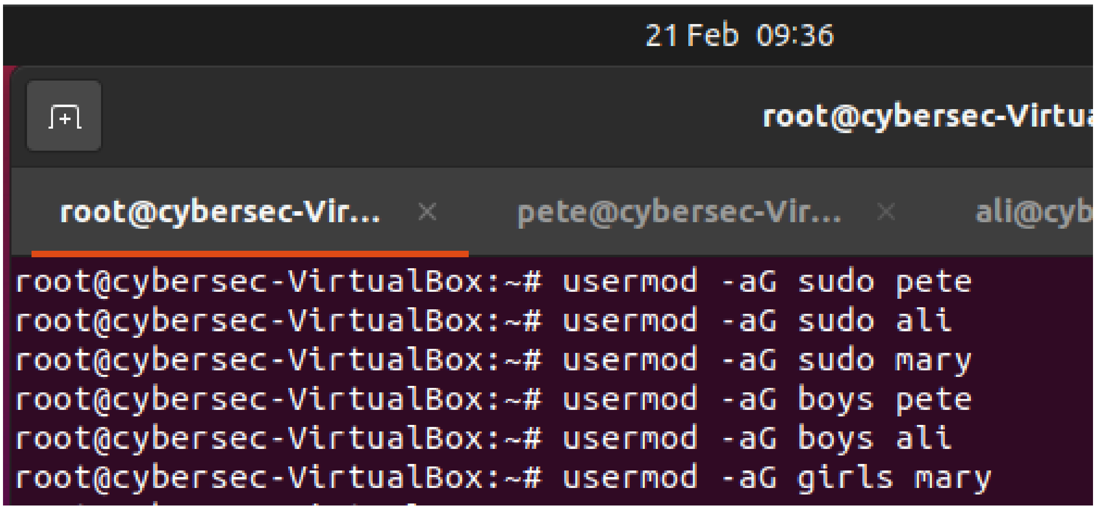

Figure 5.2: Adding the users to sudo and their respective groups.

We can verify which groups a given user is in by using the "groups [username]" command.

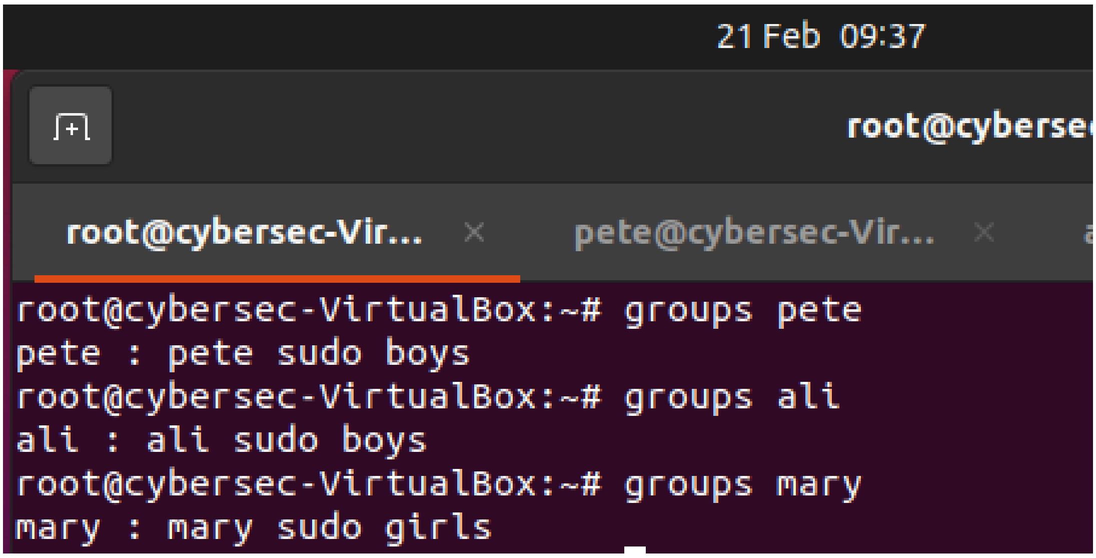

Figure 5.3: Verifying that the users were added to the groups.

This can also be checked by viewing all groups on the system via "getent groups".

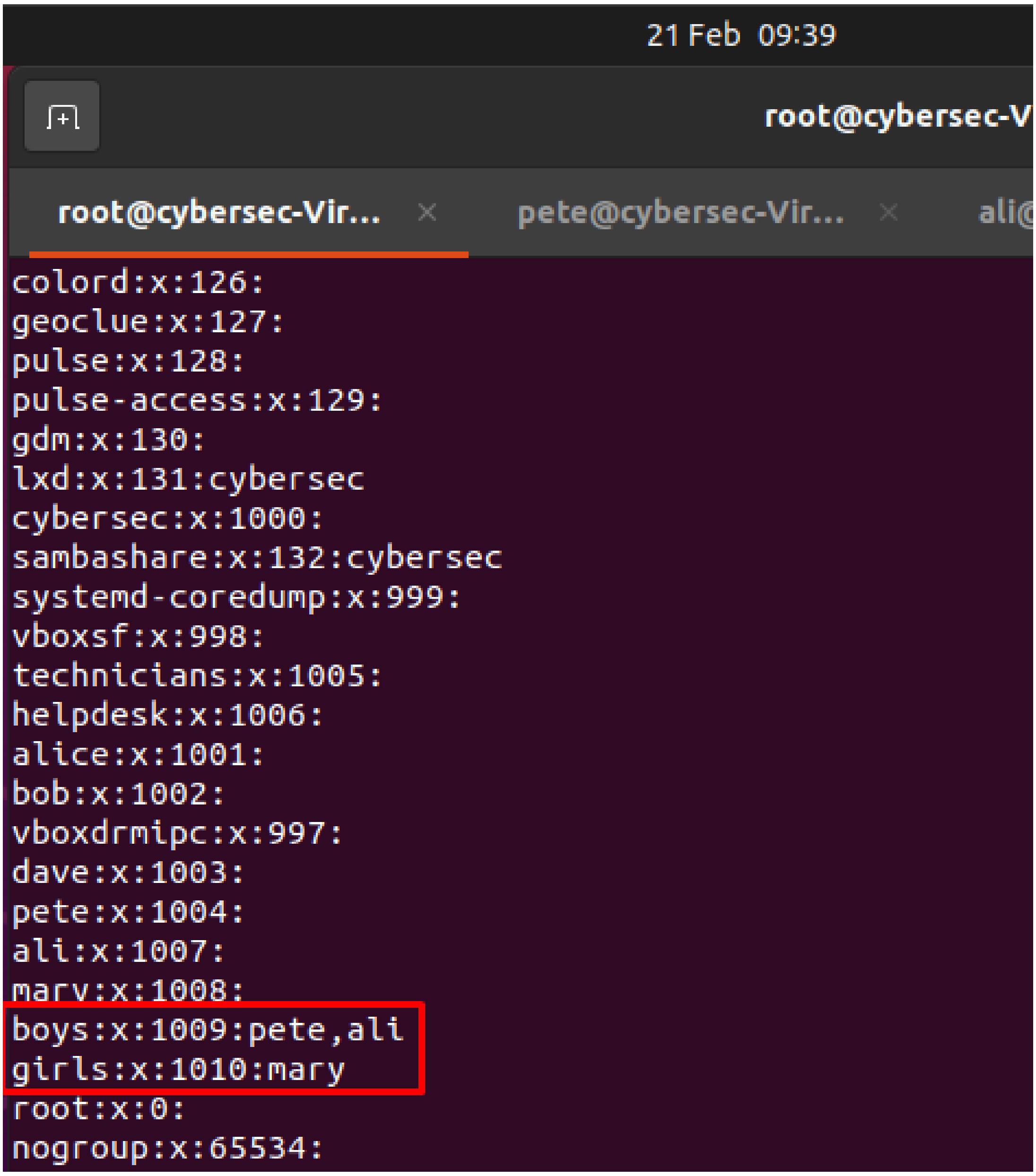

Figure 5.4: Seeing all groups (Boys and Girls are highlighted).

5.2 Using chmod and chgrp to assign permissions

Chmod is a command that changes the permissions for a given file or directory. It can change permissions for reading, writing and executing files for the owner of the file, a group of users, other users[1] or all users. I used this help page (*Unix File Permissions* n.d.) to assist in my learning of these commands as well as access control on UNIX systems.

5.2.1 Mnemonic and octal chmod

Chmod's arguments can be supplied as letters (mnemonic) or numbers (octal). Functionally, there is no difference between the two, as the permissions will be changed regardless. A mnemonic chmod command would be "chmod a rwx message.txt", which gives all users read, write and execute permissions for the file message.txt. An octal chmod command would be "chmod 777 message.txt", which gives the owner, group members and others read, write and execute permissions. The number 777 correlates to this because octal chmod operates by numeric sums. Reading permissions are the number 4, writing is the number 2 and executing is the number 1. Adding these together results in the number 7.

5.2.2 Restricting directory access

For the purposes of testing, a directory called D1 was added to user Mary's home. This directory was associated with the girls group via chgrp, and modified with an octal chmod command so that other users cannot access the directory whatsoever, but Mary herself and users of the girls group can read and execute from it.

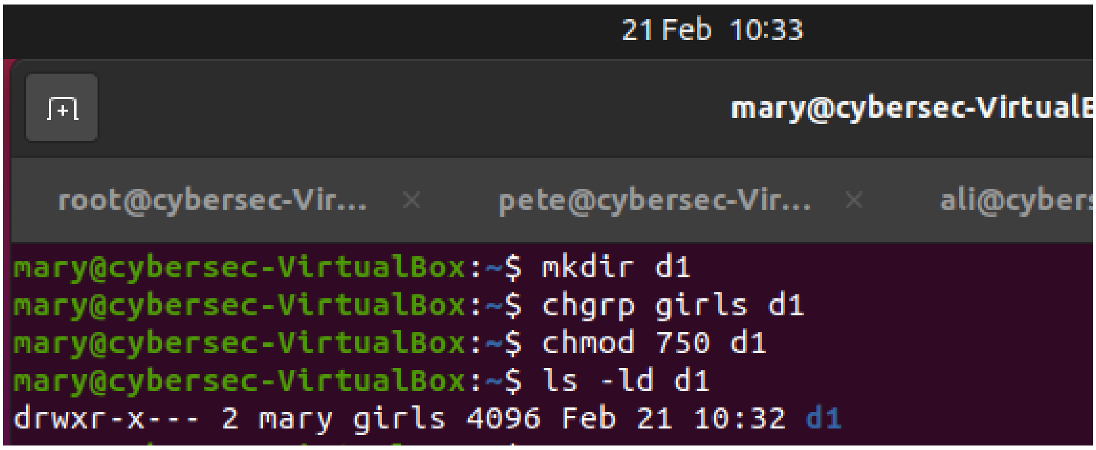

Figure 5.5: Creating D1 and modifying its permissions.

By using the command "ls -ld", the permissions of the directory are outputted. The returned message reveals that:

- The directory owner (Mary) has **R**ead, **W**rite and e**X**ecute permissions.
- Group members can **R**ead and e**X**ecute.
- Others can only e**X**ecute.

This can be tested using Ali and Pete's accounts, which are members of the boys group and not of the girls group, meaning they are considered as "other users".

[1] Defined as users that aren't the owner or in an associated group with permissions.

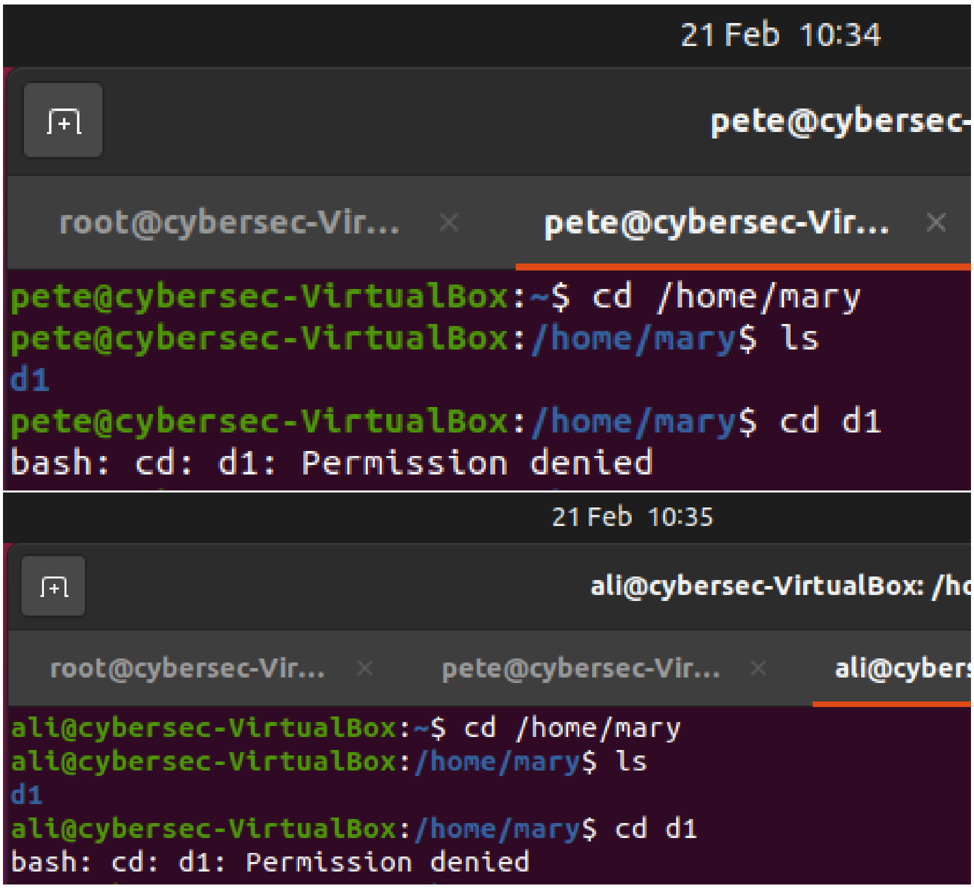

Figure 5.6: Attempting to access D1 as Pete and Ali.

Note

An issue arose here that I didn't have another user in the girls group to test access with. To fix this, I went back and added the existing Alice account from Lab 2 to the girls group with *sudo usermod -aG girls alice*.

Using another account in the girls group, we can test if other girls can access the directory, which succeeds.

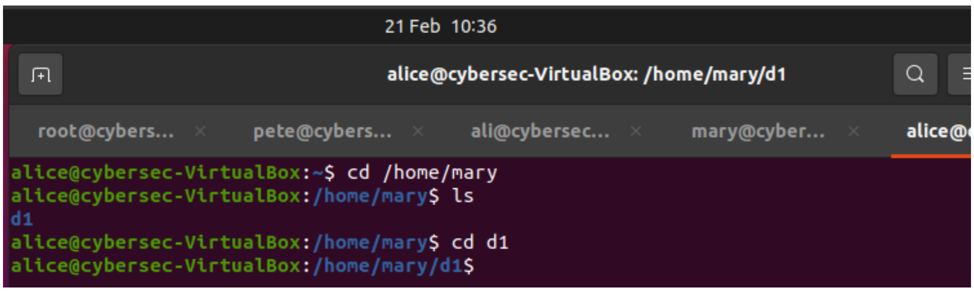

Figure 5.7: Successfully accessing D1 as Alice.

Because of the permissions set, Alice is permitted to read the d1 directory and execute files within it, but she cannot write any files of her own, which is expected behaviour.

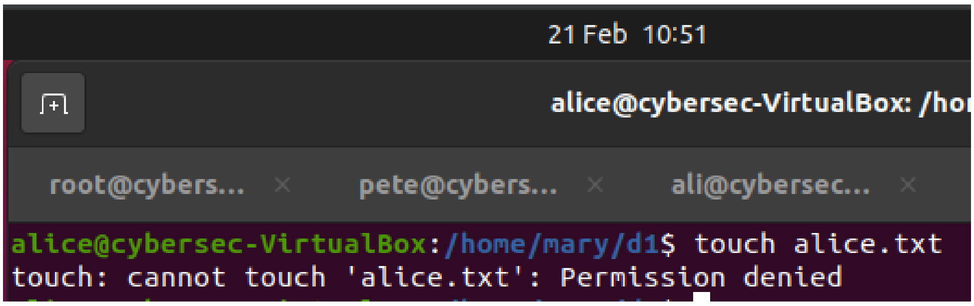

Figure 5.8: Failing to write to D1 as Alice.

To test this further, the permissions can then be modified[2] again to allow girls to write files, which will then allow Alice to make the file.

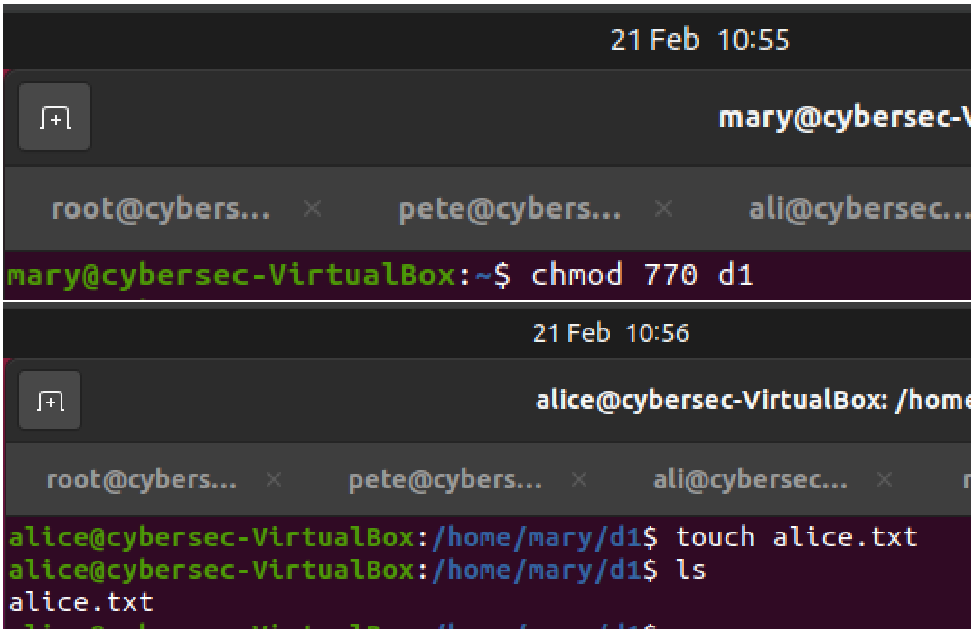

Figure 5.9: Modifying the permissions of D1, allowing Alice to write the file.

[2]770 is "rwxrwx—", which correlates to the file owner and members of the group having read, write and execute permissions, but other users have none.

5.2.3 Chgrp and chown

> **Note**
>
> I used different directory names (*BoysDirectory* instead of *Photos*), but I have still demonstrated all exercises from this lab.

Chown is a command similar to chgrp, which assigns a file or directory's ownership to a specific user. For this example, we will create a directory in the shared /home folder and assign group ownership to Boys via chgrp, and using chmod to allow all Boys all permissions, and all other users read & execute permissions.

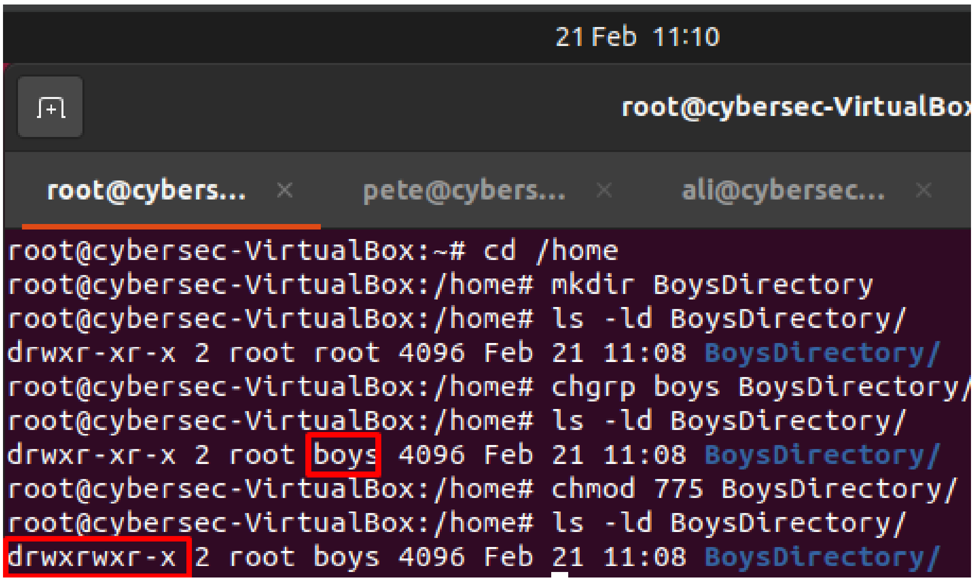

Figure 5.10: Making BoysDirectory and giving group ownership to Boys.

This new directory can be accessed by all users, but only written to by boys. We can now test chown by making two subdirectories within BoysDirectory, where one will be owned by Pete, and one by Ali. Both users also modify the permissions of their own directories to only be accessible by them.[3]

[3] 700 is "rwx——", which means only the owner may read, write and execute.

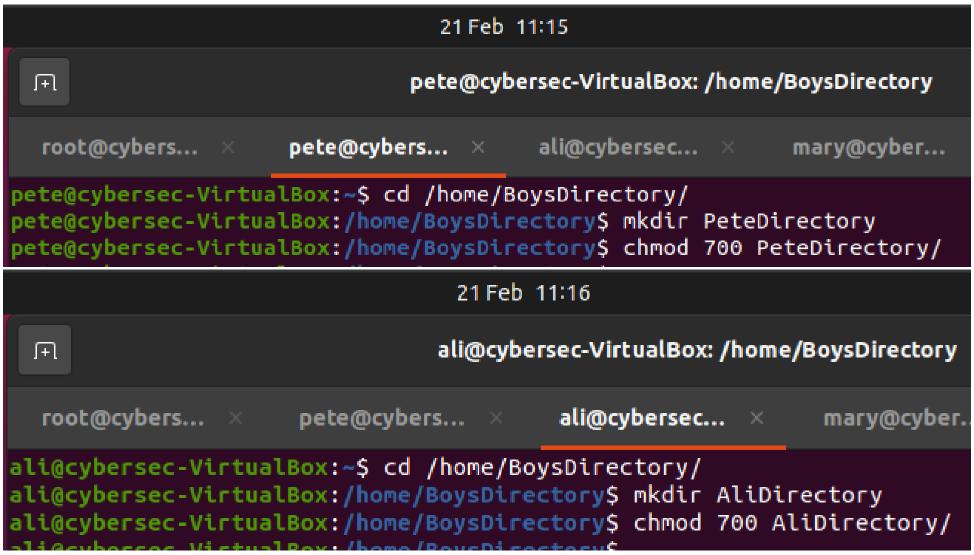

Figure 5.11: Creating and modifying each user's directories and access permissions.

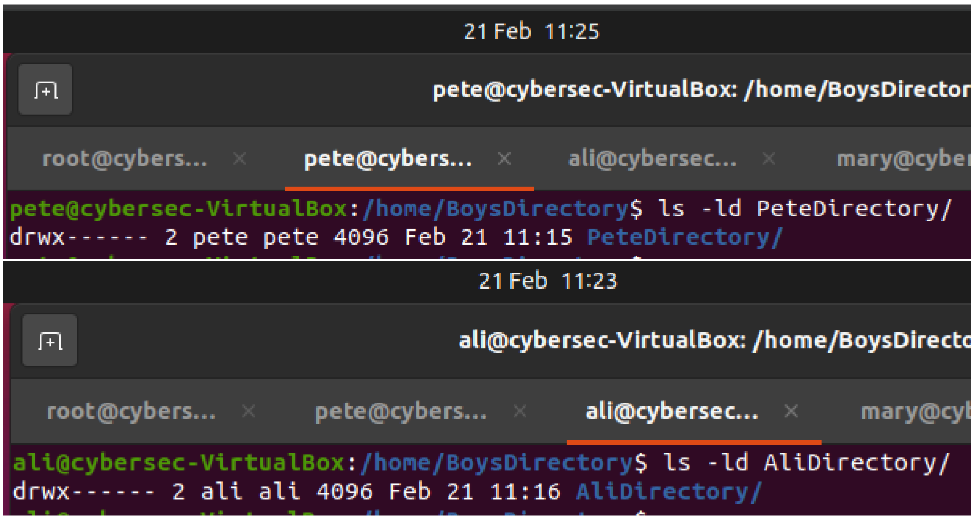

Figure 5.12: Viewing the permissions of Pete and Ali's directories.

We can then prove that only the owners of the directories may access them by first accessing their own directory, but then attempting to access the other user's directory:

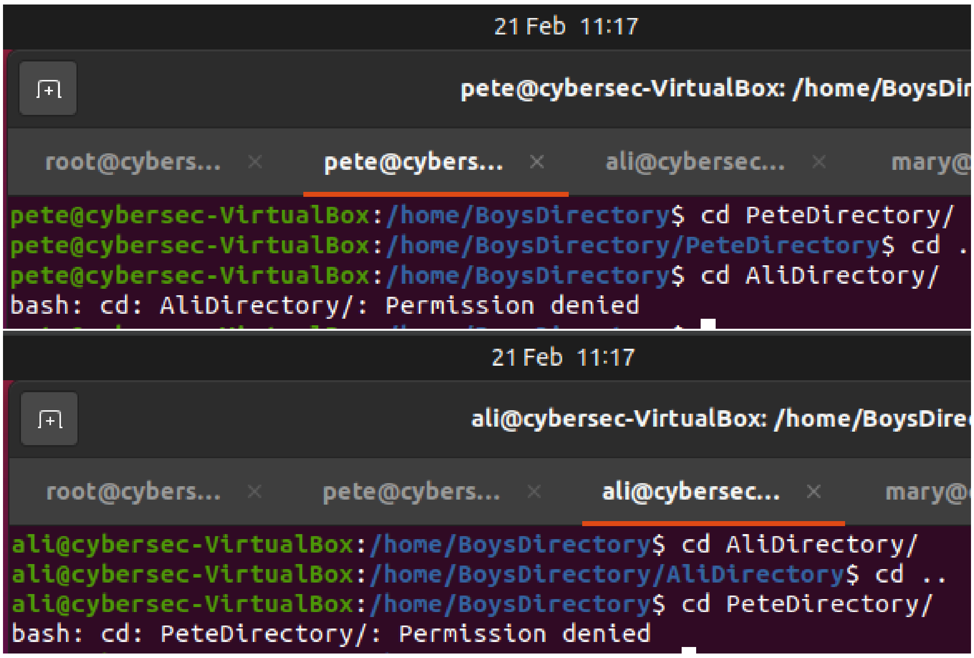

Figure 5.13: Successfully accessing their own directory, but failing to access the other because the user isn't the owner.

An important distinction to be made here is that these subdirectories are owned by Pete and Ali respectively. They do **not** inherit the boys group ownership by default, meaning that commands to change group permissions will be ineffective for other boys, as everyone who is not the owner is considered "other" unless chgrp is used, as seen in Figure 5.13.

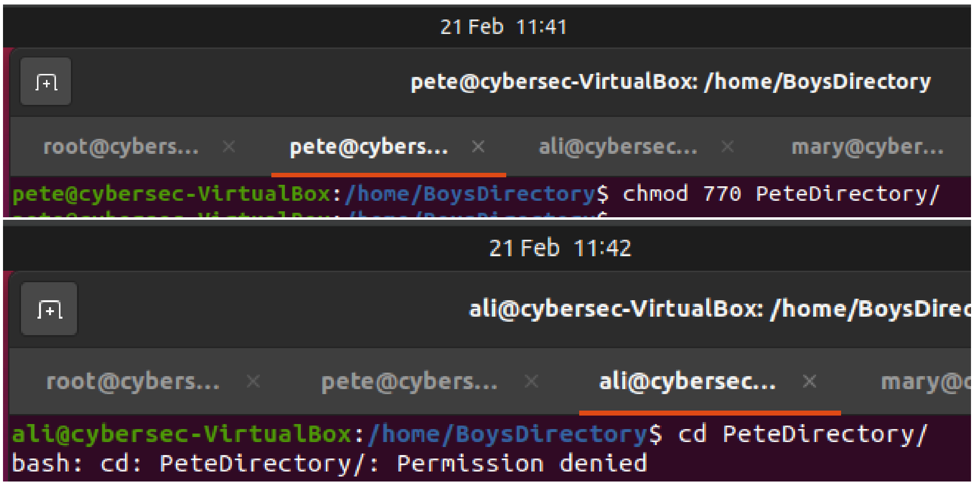

Figure 5.14: Giving group RWX access to PeteDirectory

Ali still can't access the directory because he is not part of the "Pete" group, but he (and any other users) would be able to use permissions given to "others".

Password Cracking

In this lab, a simple brute-force attack program written in C was used to crack a hashed account password.

6.1 Linux password storage

Linux systems store user account details across two files, */etc/passwd* and */etc/shadow*. I learned information about this from this site (Gite, 2024), which states that the public unencrypted ASCII file */etc/passwd* contains a line for each user on the system, with publicly accessible information such as username, user ID and group ID, whereas the encrypted */etc/shadow* file contains the encrypted passwords of users on the system.

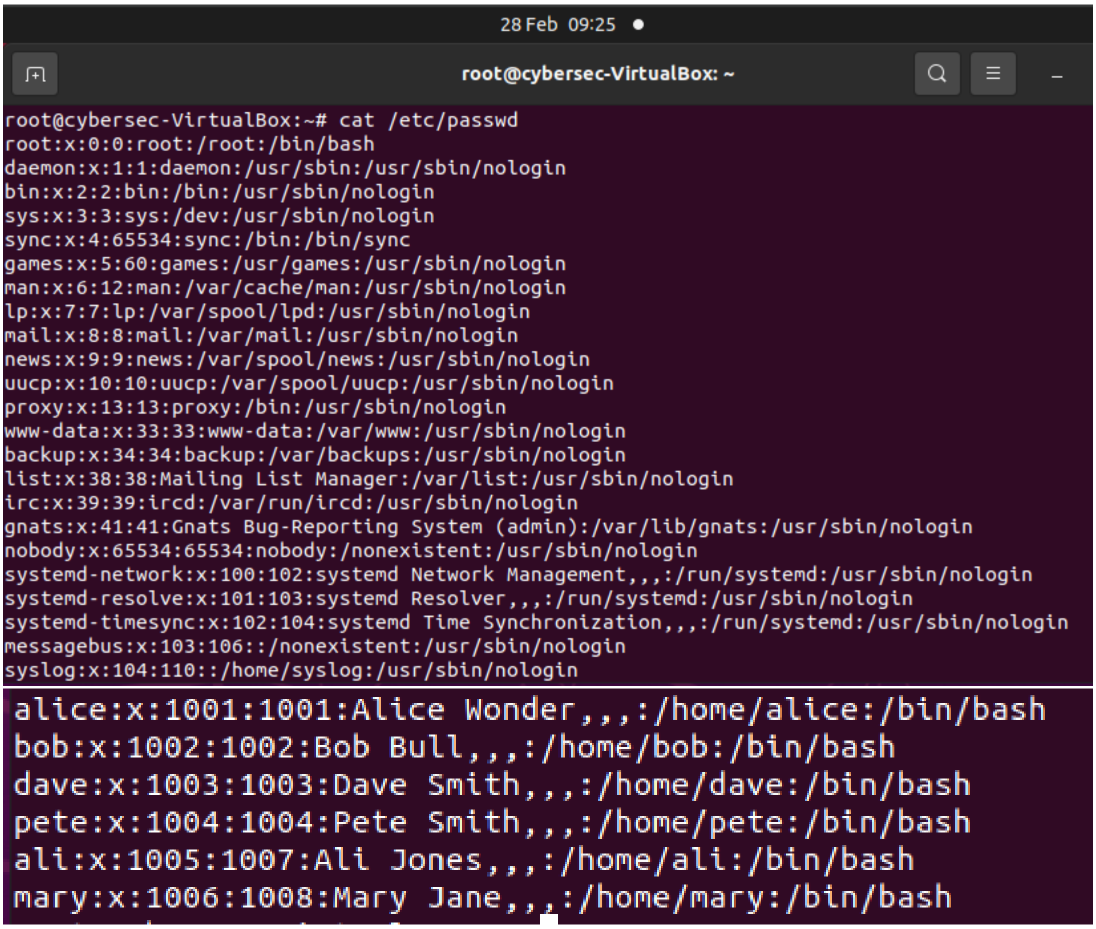

Figure 6.1: Some of the contents of /etc/passwd, with the created users from earlier labs.

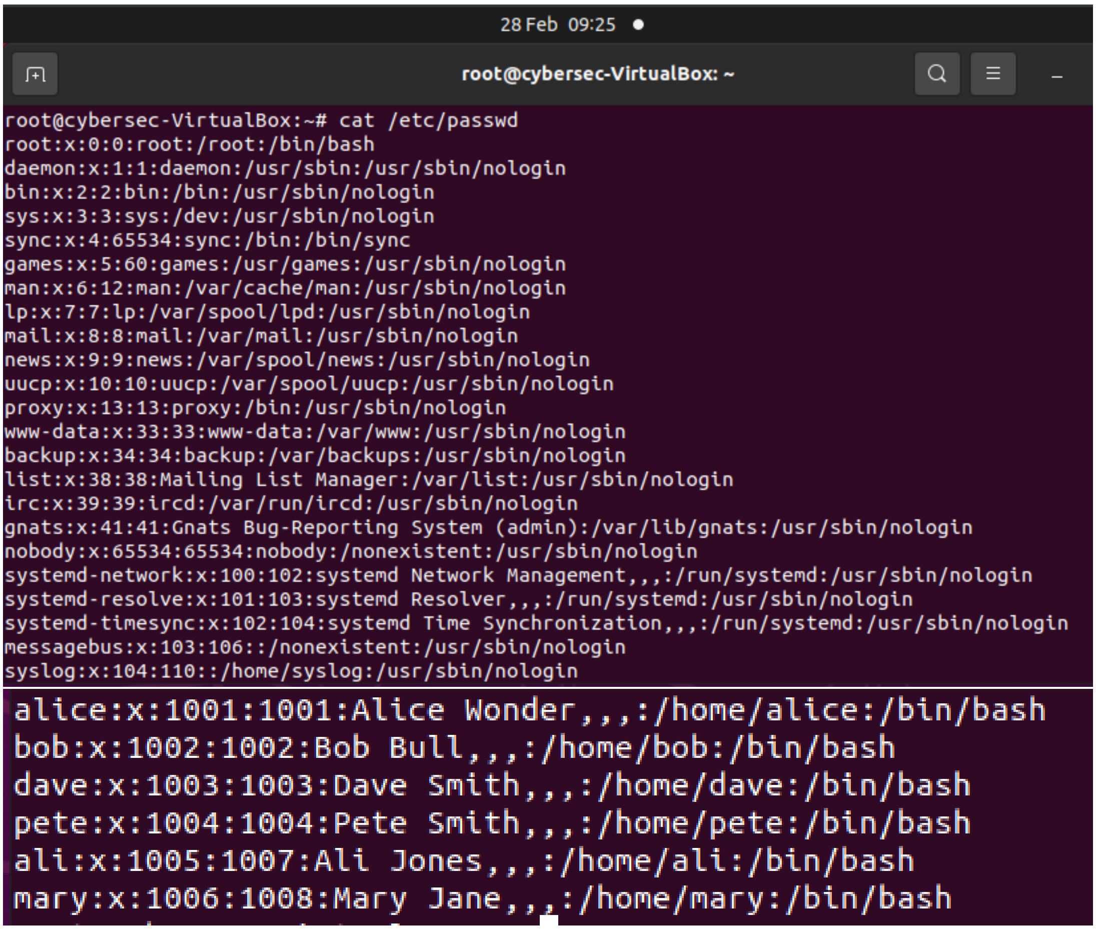

Figure 6.2: Some of the contents of /etc/shadow, with the created users from earlier labs.

6.2 crack.c

The provided code in crack.c is a small program that cracks passwords against the builtin Ubuntu dictionary, located in */usr/share/dict/words*.

6.2.1 Importing and compilation

First, this code must be ported into the Ubuntu VM using "nano crack.c", and pasting the code from Moodle into the file.

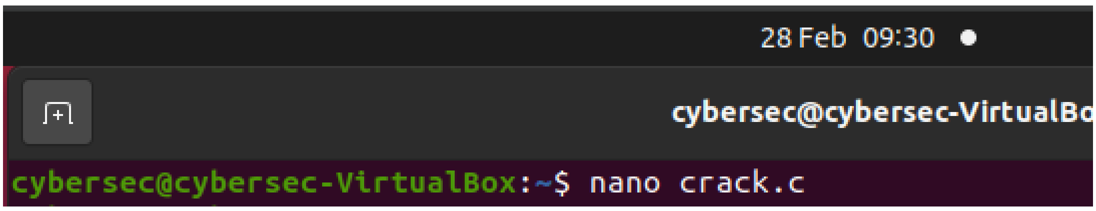

Figure 6.3: Porting crack.c into the VM.

Because C code can't be natively run, it must first be compiled. This creates an executable called 'crack'.

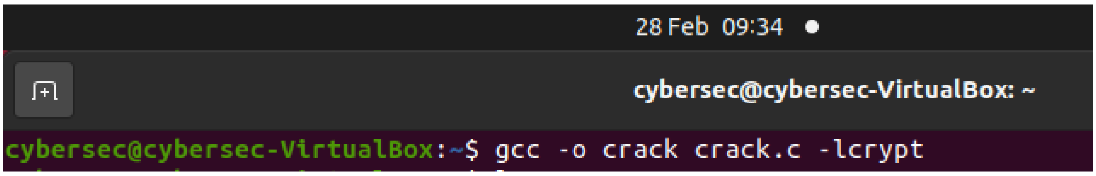

Figure 6.4: Compiling crack.c.

The '-lcrypt' argument supplies the Linux crypt library when compiling, allowing the crypt() function to be used more securely.

6.2.2 Creating a test user

To use the program, it will be necessary to create a new user, whose password can be found in the dictionary. For this, the user 'fred' will be created, and his password will be 'peach'. We can see if 'peach' appears in the dictionary, as well as the overall dictionary word count.

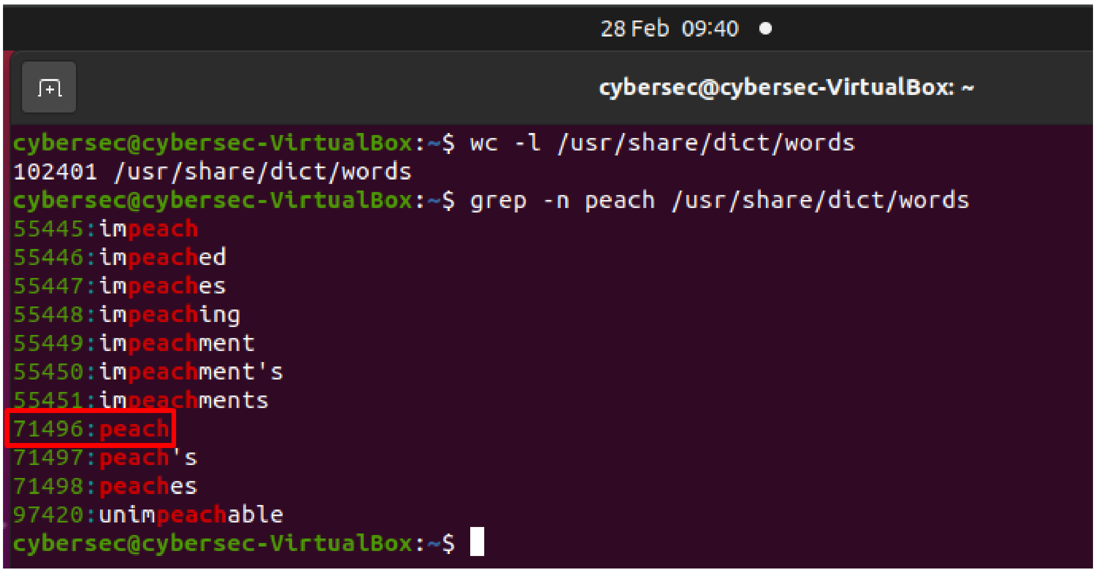

Figure 6.5: Checking the dictionary for the word 'peach', which is the 71496th entry.

Figure 6.6: Creating the 'fred' user, with the password 'peach'.

It will then be necessary to get the hashed version of Fred's password, which can be done using "cat /etc/shadow | grep fred | awk -F: '{ print $2 }'. This command will read the shadow file, selecting the row starting with 'fred'. Then, it will extract his hash by selecting the second column.

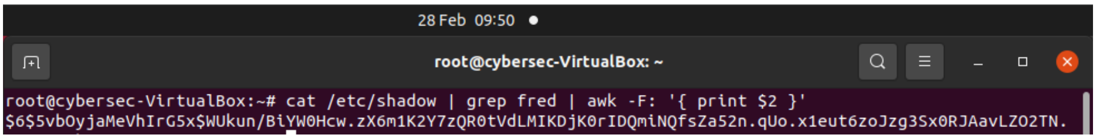

Figure 6.7: Viewing fred's hashed password.

6.2.3 Cracking the password

The program takes two arguments, with the first being the salt used on the password and the second being the entire hashed password. It requires the salt as an argument because it will apply the salt to each password it checks. As seen in Figure 6.7, Fred's entire hashed password is

```
$6$5vbOyjaMeVhIrG5x$WUkun/BiYW0Hcw.zX6m1K2Y7zQR0tVdLMIKDjK0rIDQmiNQfsZa
52n.qUo.x1eut6zoJzg3Sx0RJAavLZO2TN.
```

We can figure out the salt used on this password by looking at how it starts. The first 20 characters of this password are the salt, noticeable by how they are between two dollar signs. This can be supplied as the first argument for the compiled crack program, and the second argument would be the entire password, including the salt as well. Ultimately, this forms the following command:

```
./crack '$6$5vbOyjaMeVhIrG5x$'
'$6$5vbOyjaMeVhIrG5x$WUkun/BiYW0Hcw.zX6m1K2Y7zQR0tVdLMIKDjK0rIDQmiNQfsZa
52n.qUo.x1eut6zoJzg3Sx0RJAavLZO2TN.'
```

It is imperative to use **quotation** marks rather than speech marks, as Linux will otherwise incorrectly interpret the arguments given due to there being dollar signs in the hash, meaning that the crack will be unsuccessful.

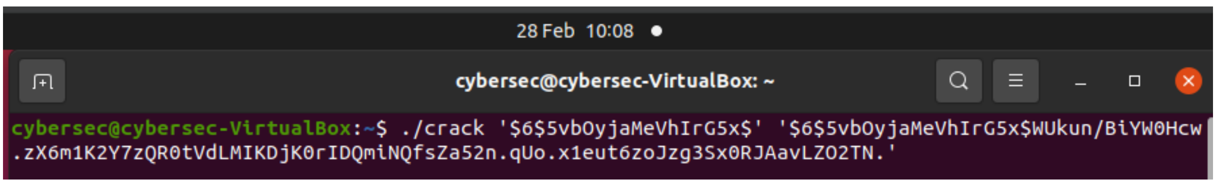

Figure 6.8: Entering the command.

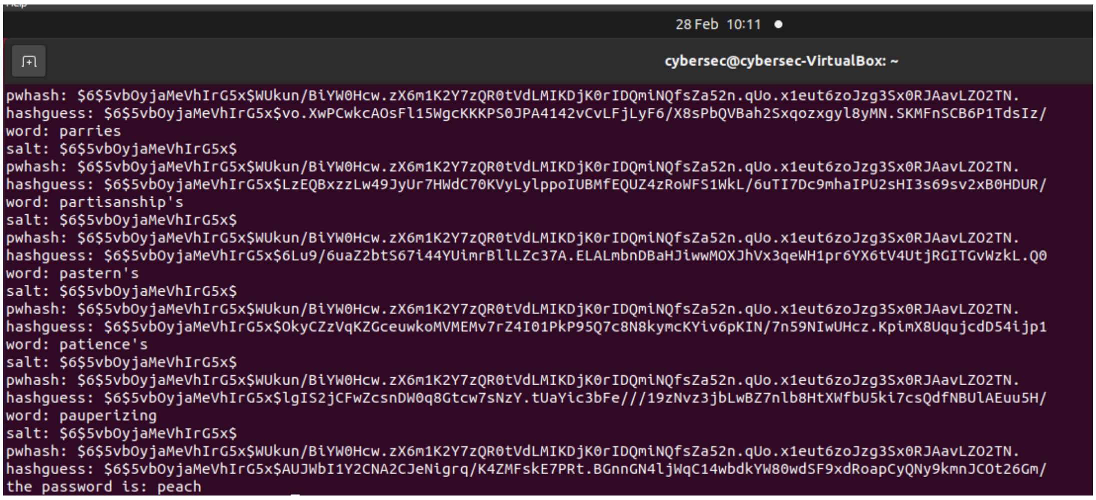

Figure 6.9: The hashed password is revealed as 'peach'.

Note the differing timestamps, showing how this took around 3 minutes to execute, even on a 10-core Intel i5-12600k. This is because the word 'peach' occurs so late in the dictionary as seen in Figure 6.5.

Conclusion

Remember to change the title page to not be yellow!

Check Lab2 Sec 2.1.1, is sudo -s actually the same? Also in Figure 2.4, both of those are the same.
Looking at your header, specifically in Lab 5 as well, it may be worth removing the "Lab 1 -" "Lab 2 -" parts of the chapter names as you already did it using chapter numbering.

You've not done what Lab 5 asks you to. Yes, you've demonstrated the knowledge, but it's not in the specific way they wanted. Redo it with the /home/photos folder. Could just mv it for rename but timestamps may snitch?

Bibliography

Gite, V. (19th Feb. 2024). *Understanding /etc/passwd File Format*. URL: https://www.cyberciti.biz/faq/understanding-etcpasswd-file-format/ (visited on 02/03/2024).

Heinlein, P. (13th Sept. 2016). *OpenSSL Command-Line HOWTO*. URL: https://www.madboa.com/geek/openssl/#how-do-i-simply-encrypt-a-file (visited on 24/01/2024).

Kolletzki, S. (1996). 'Secure internet banking with privacy enhanced mail — A protocol for reliable exchange of secured order forms'. In: *Computer Networks and ISDN Systems* 28 (14), pp. 1891–1899. DOI: https://doi.org/10.1016/S0169-7552(96)00089-X.

Unix File Permissions (n.d.). URL: https://docs.nersc.gov/filesystems/unix-file-permissions/ (visited on 21/02/2024).