BIRMINGHAM CITY
University

CMP5329 Logbook

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 already 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 OpenSSL to encrypt and decrypt data using the DES and AES256 symmetric encryption algorithms, as well as 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, "openssl version" can be executed. The provided virtual machine from the CMP5329 Moodle page uses OpenSSL version 1.1.1f, dated 31st March 2020.

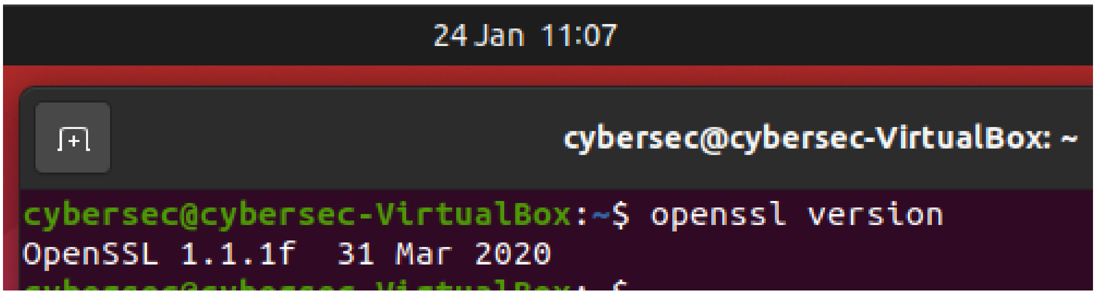

Figure 1.1: Getting the OpenSSL version

The list of OpenSSL ciphers can be viewed via "openssl ciphers".

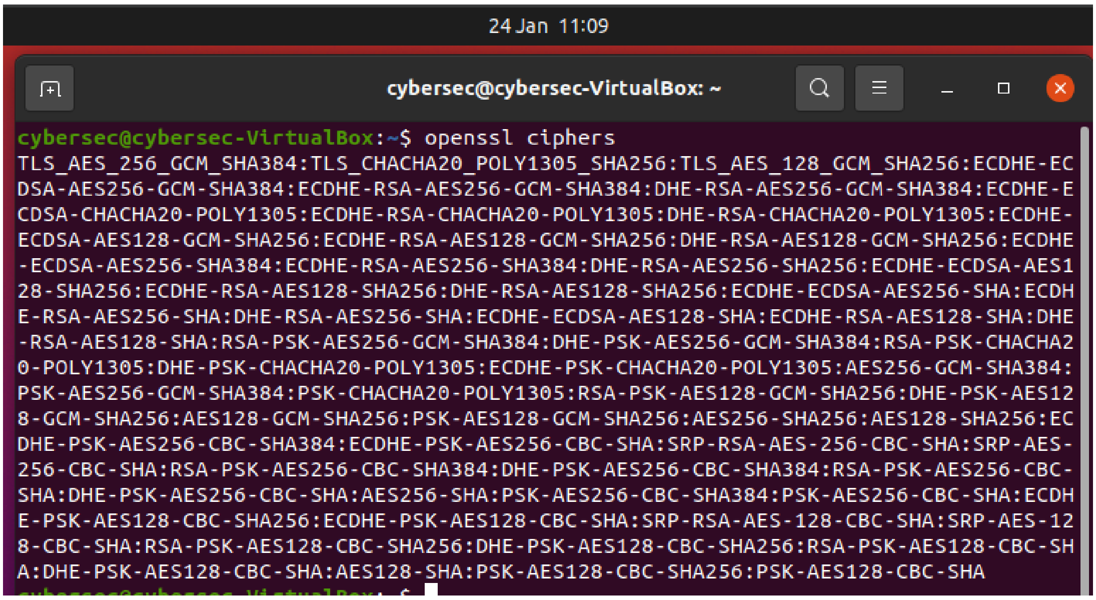

Figure 1.2: Getting the OpenSSL ciphers

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.

1.2.1 DES symmetric encryption

OpenSSL can be used to encrypt plaintext into ciphertext. Many algorithms exist to generate ciphertext, but the DES symmetric encryption algorithm will be used here.

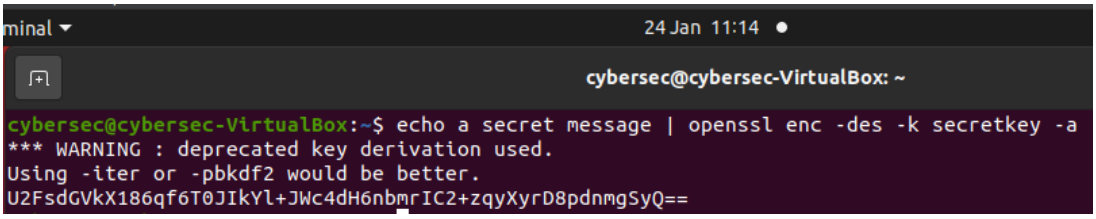

Figure 1.3: Converting "a secret message" to ciphertext using DES with key "secretkey".

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

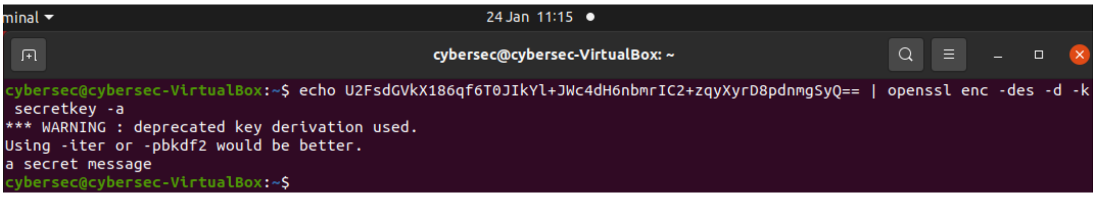

Figure 1.4: Decoding the ciphertext back to its original form using the key "secretkey".

1.2.2 AES256 symmetric encryption and decryption

The DES algorithm is considered weak due to how simple it is to brute-force using today's processing power. Newer algorithms were therefore developed, with one of these being AES.

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

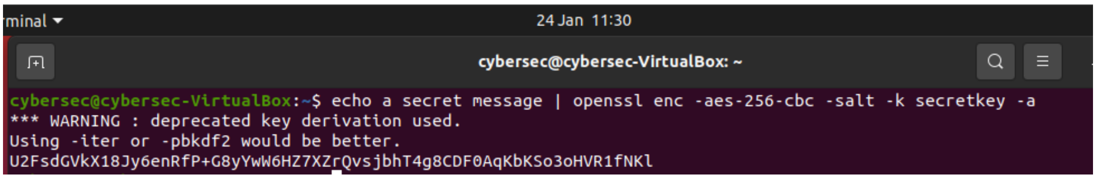

Figure 1.5: Encoding the plaintext with AES-256-cbc using the key "secretkey".

In this command the AES-256-cbc cipher is used, 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 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. This is unlike symmetric encryption which uses one key for both users, but can be much more secure. 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 user with the private key associated with the signature.

1.3.1 Generating an RSA private key

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

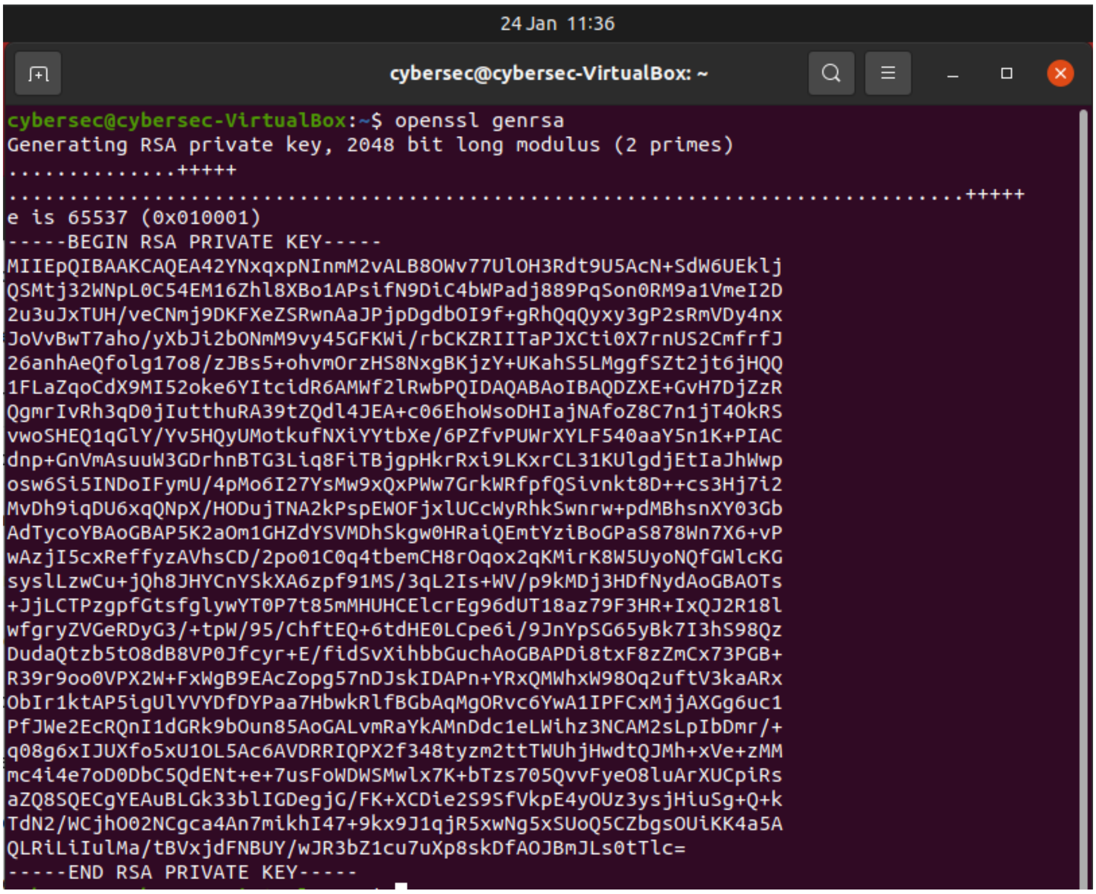

Figure 1.6: Generating a 2048-bit private key using genrsa.

1.3.2 Storing DES3 & passphrase encrypted RSA keys in a file

The key generated can then be encrypted using an encryption algorithm and a passphrase. It can then be stored into a Privacy Enhanced Mail (.pem) 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 "secretkey".

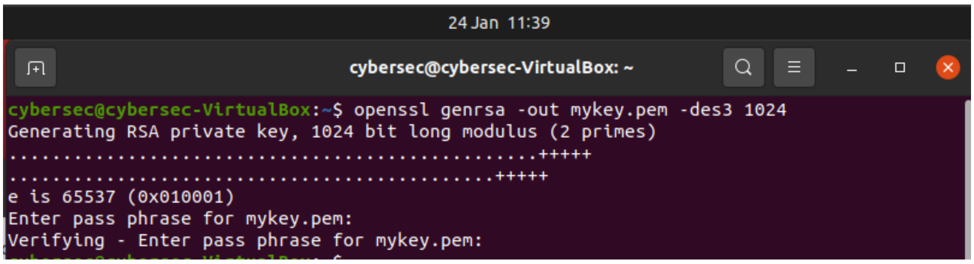

Figure 1.7: Generating and storing a 1024-bit private key using genrsa, DES3 and the passphrase "secretkey".

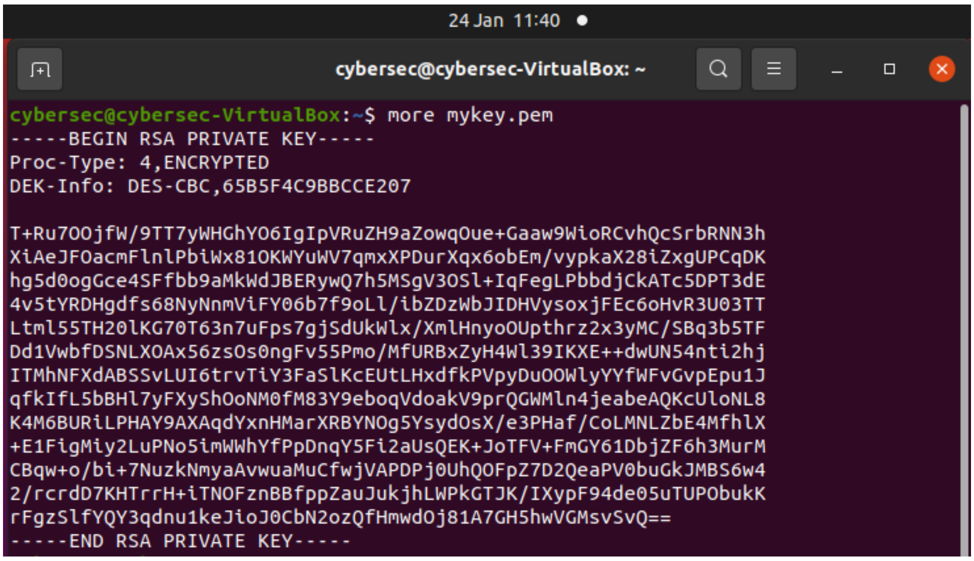

Figure 1.8: The key stored in mykey.pem.

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.

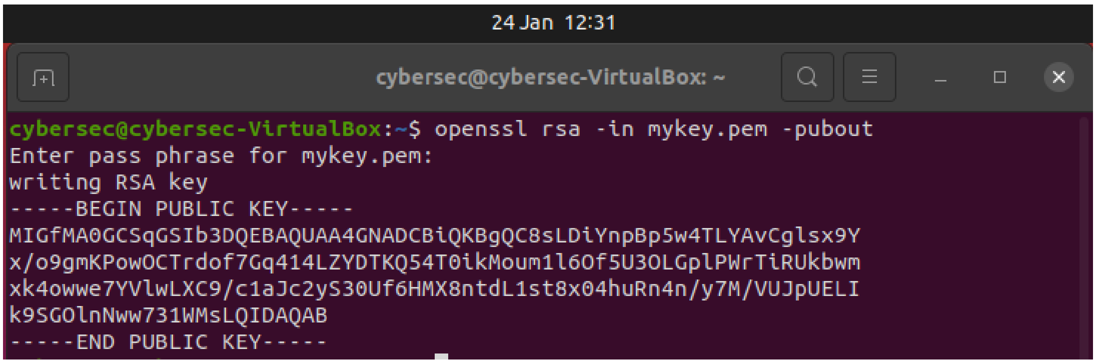

Figure 1.9: The public key generated from mykey.pem.

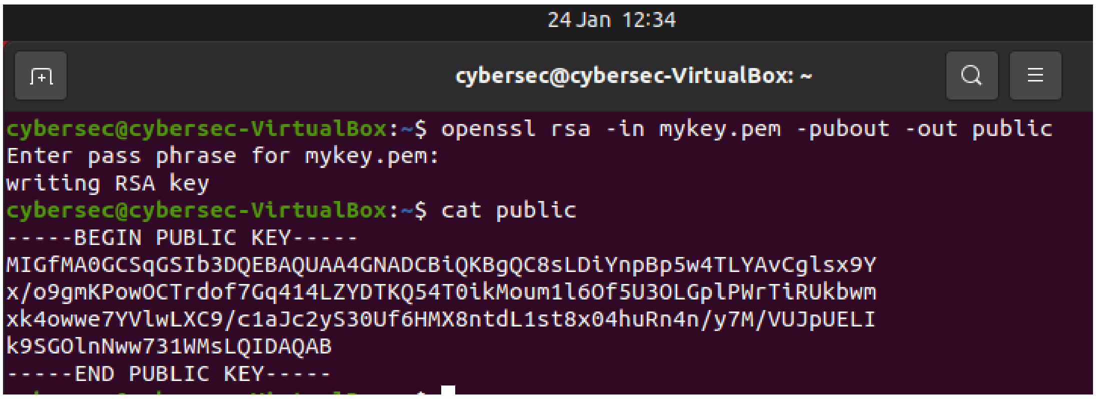

Figure 1.10: Storing the public key in a file.

"-out public" writes the key to a file called "public". This file can then be read using cat.

1.3.4 Obtaining a message/file digest

To mitigate the risks of data interception or corruption, files can have "digests", which are the result of hashing their contents. If the file is modified whatsoever, the digest would be different.

OpenSSL can generate digests using its "dgst" command.

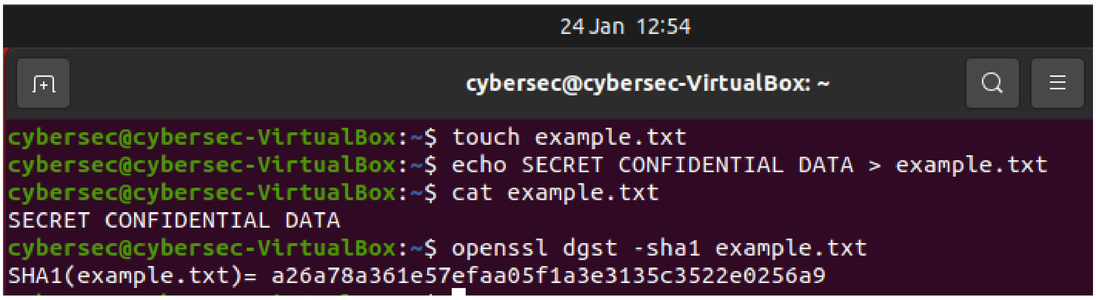

Figure 1.11: Creating a file, then getting the SHA1 digest of it.

This can also be verified by using sha1sum, which returns the same digest.

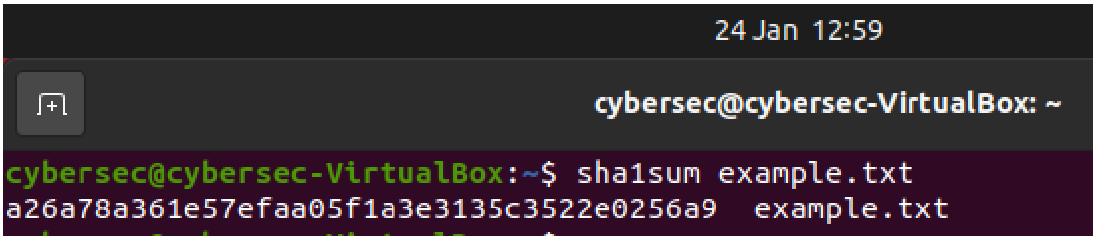

Figure 1.12: Verifying the digest.

1.3.5 Signing a digest

Signing a message digest using your private key definitively proves you sent it, 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.

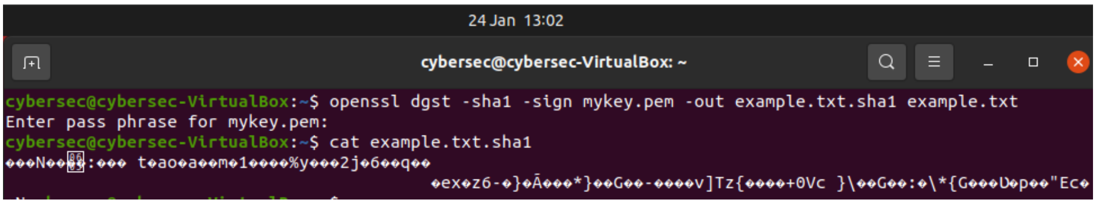

Figure 1.13: Writing a signed digest to a file.

Note that when we try to read this file, it is completely illegible, as it is not in Base64/ASCII format. It can be converted to Base64 using OpenSSL's "enc" command.

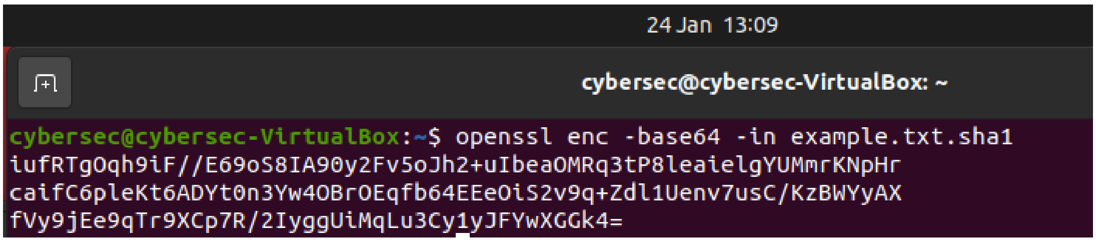

Figure 1.14: Encoding the signed digest to Base64.

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.

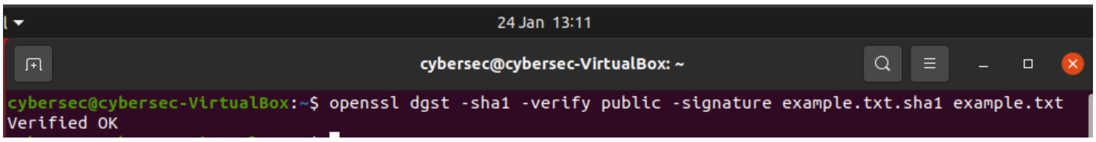

Figure 1.15: Verifying the signature of example.txt.

This returns "Verified OK" as intended. If the file does get modified through either corruption or a threat agent's interference, the digest would not be the same, seen below:

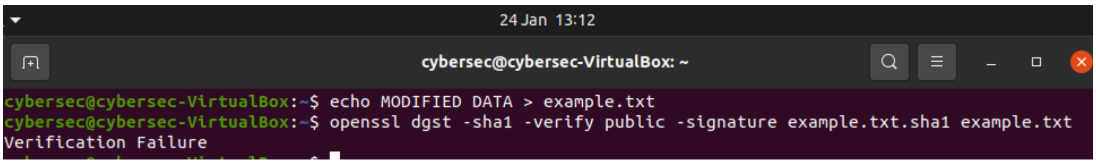

Figure 1.16: Failing to verify the signature of example.txt, as it has been modified.

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 which will change the terminal to be at root level.

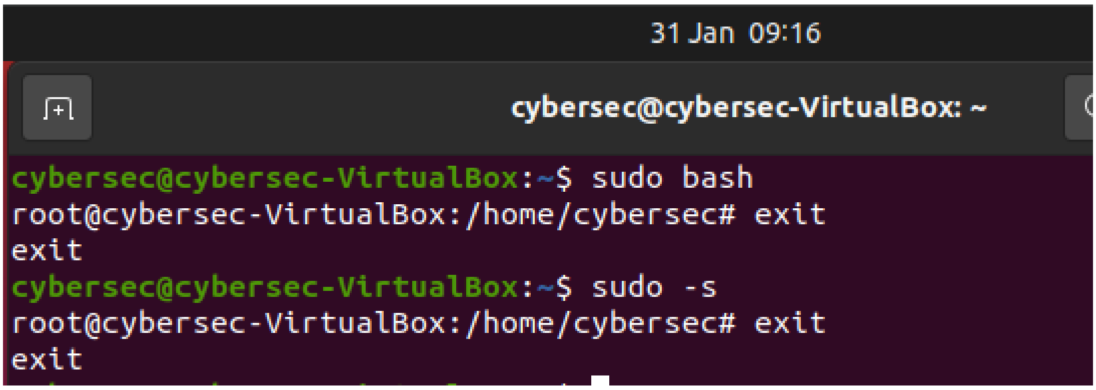

Figure 2.1: Elevating the terminal.

2.1.2 Creating Bob and Alice

With sudo permissions, users can be added 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 here.

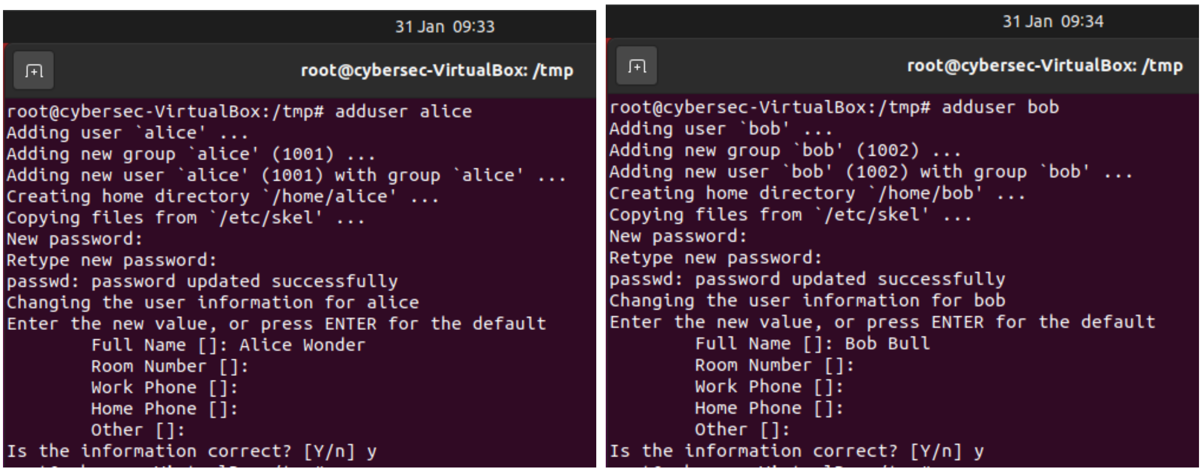

Figure 2.2: Creating users 'bob' and 'alice'.

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.

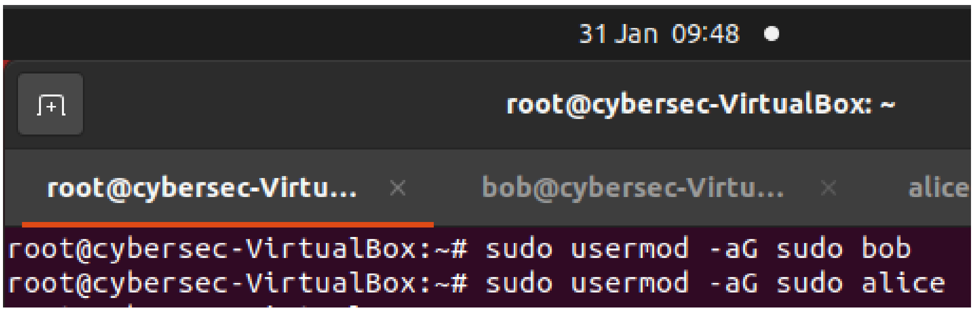

Figure 2.4: Adding bob and alice to sudo.

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.

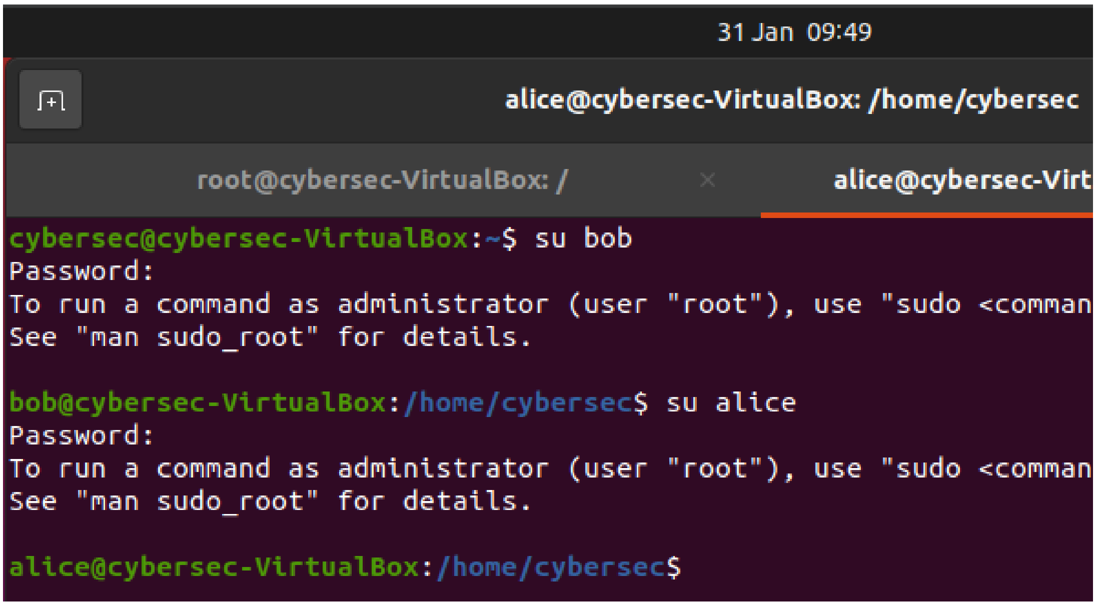

Figure 2.5: Switching the active terminal user.

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 where all users can read files placed there. 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, GPG can be used to generate and store public and private asymmetric keys.

2.2.1 Generating public/private key-pairs

"gpg1 –gen-key" generates a private key.

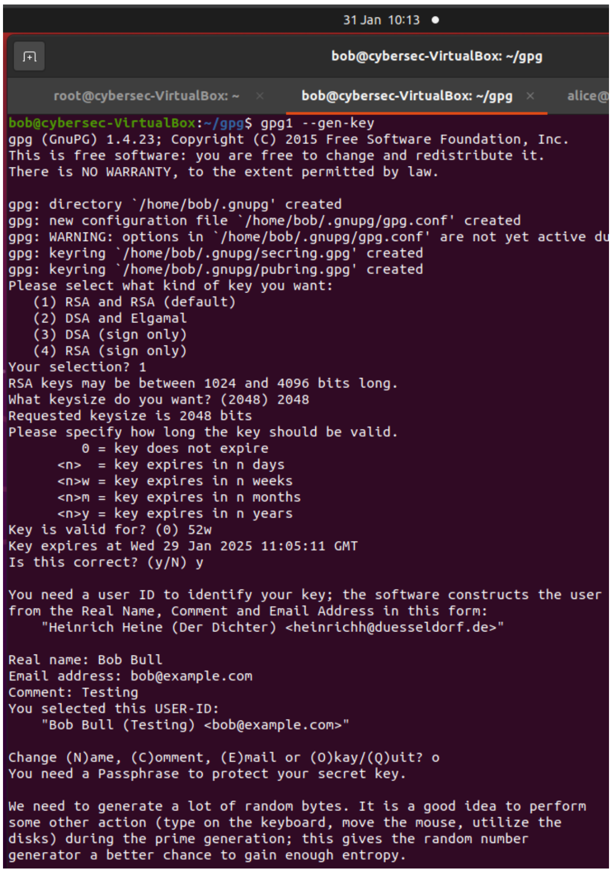

Figure 2.6: Generating a private key.

This will open a submenu where the user can select the kind of key they wish to generate, and its size and expiry date After this, they must create a user ID if one doesn't exist, with their full name, email address and an optional comment. While the key generates, the user is prompted to perform random inputs to enhance its entropy. A key was also generated for Alice.

2.2.2 Exporting public keys

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

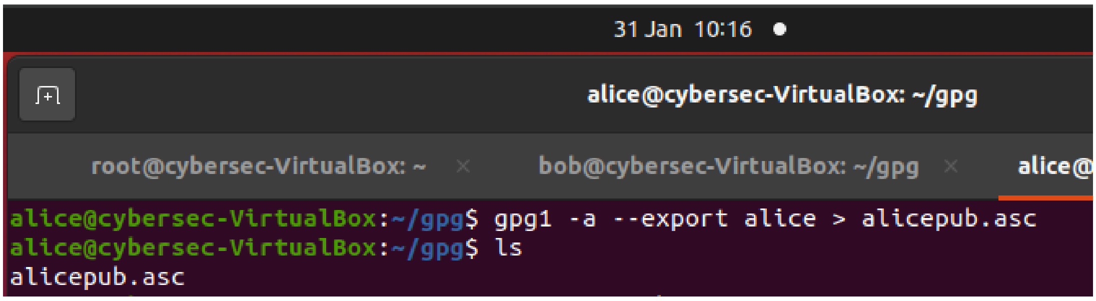

Figure 2.7: Exporting Alice's public key.

This exports the public key in ASCII format (due to the use of the -a flag) to the file "alicepub.asc". [1] Because this is Alice's **public** key, we are comfortable sharing this to the public /tmp/ directory where all users can see it.

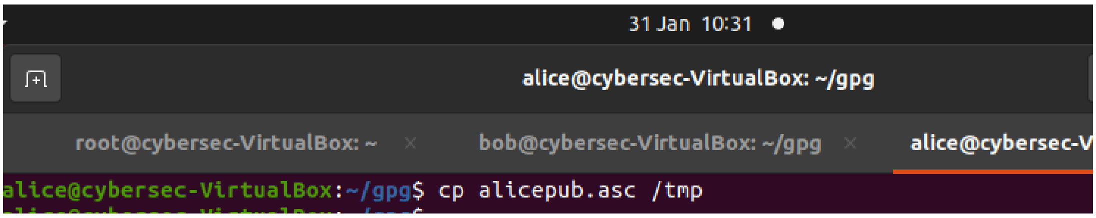

Figure 2.8: Copying Alice's public key to /tmp.

2.2.3 Importing and signing public keys

Bob can copy and import Alice's public key from /tmp.

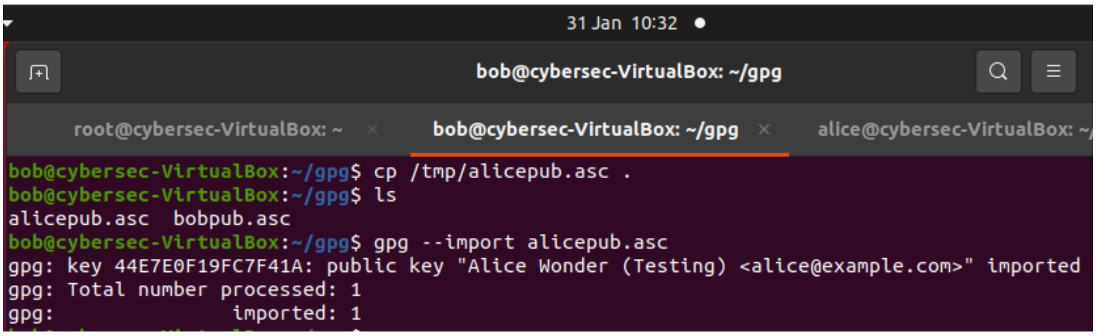

Figure 2.9: Importing Alice's public key as Bob.

[1] 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 and signing it.

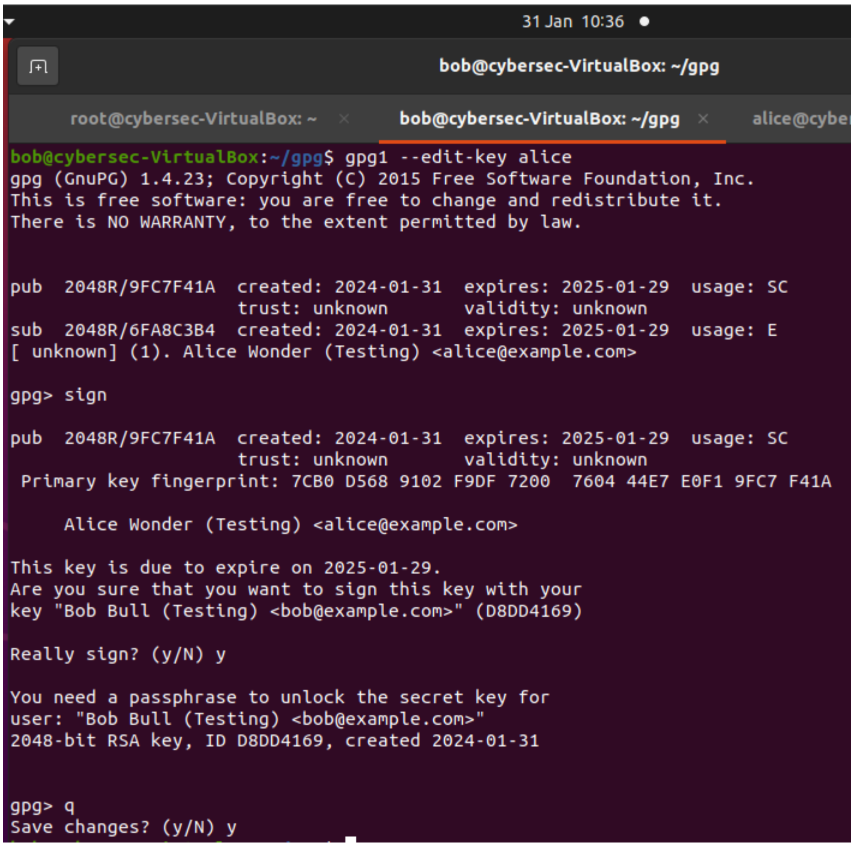

Figure 2.10: Bob signing Alice's public key.

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.

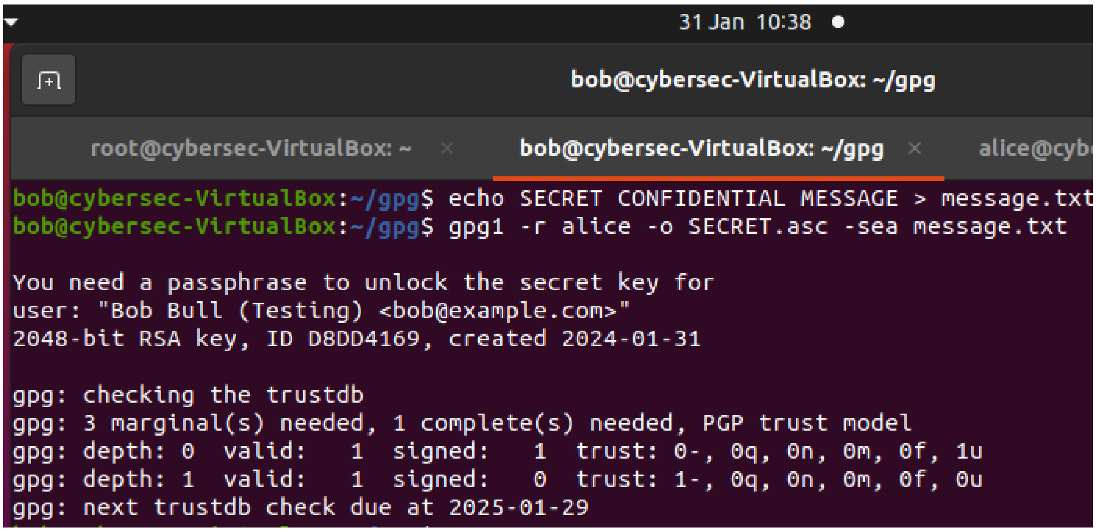

Figure 2.11: Making a file and encrypting it using Alice's public key.

This command can be broken down to its components:

- -r alice - Uses Alice's public key for encryption.
- -o SECRET.asc - Outputs the encrypted data to SECRET.asc.
- -sea message.txt - **S**ign and **e**ncrypt the contents of message.txt in **A**SCII format.

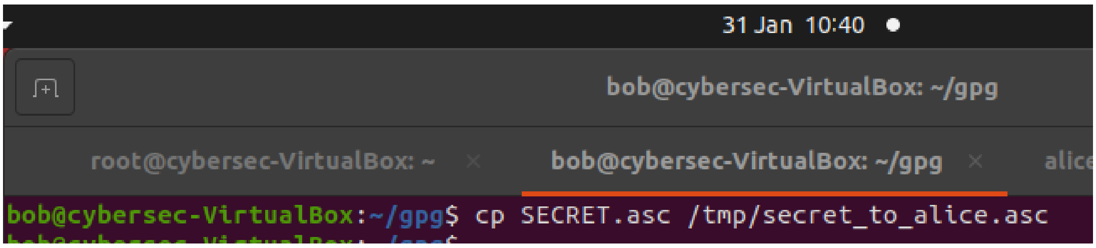

Figure 2.12: Copying the encrypted file to /tmp with the name "secret_to_alice.asc".

Alice can decrypt the file to "message.txt", where it can be read in human-legible form.

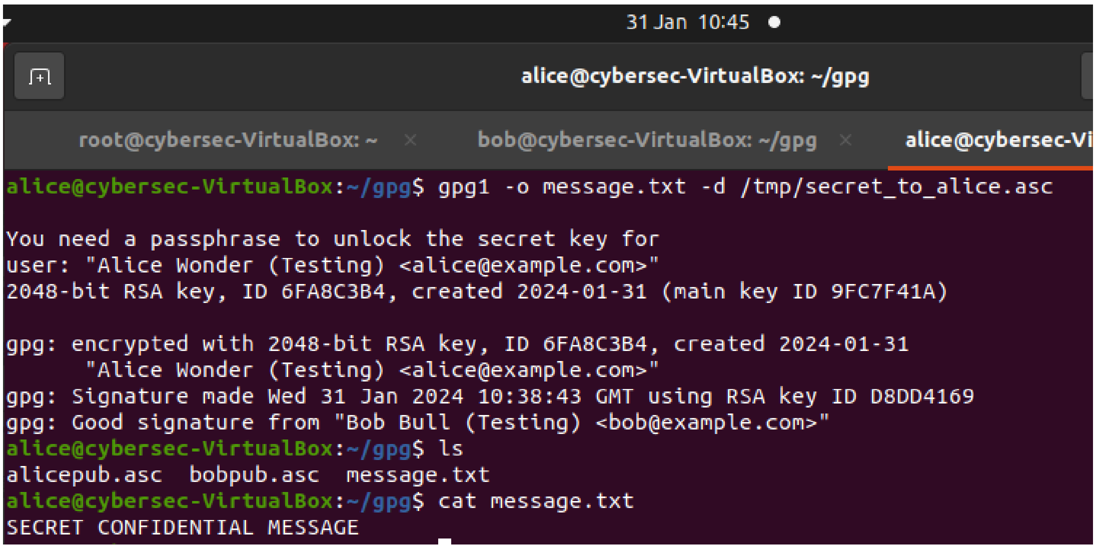

Figure 2.13: Decrypting the encrypted message and reading it.

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.

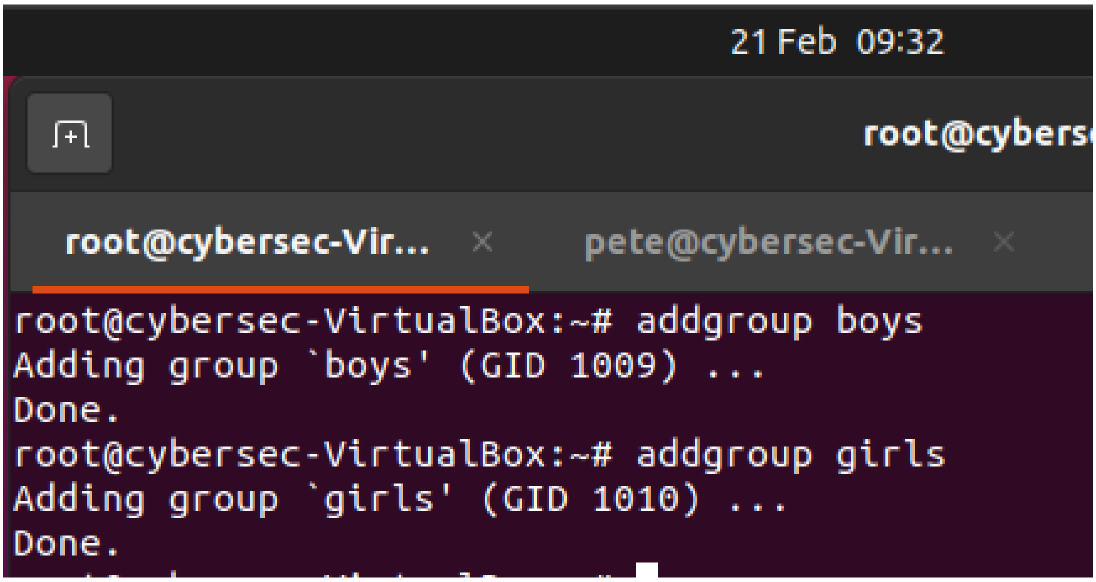

Figure 5.1: Making the "boys" and "girls" groups.

5.1.2 Adding users to groups

The new users were added to the groups mentioned above as well as the sudo group.

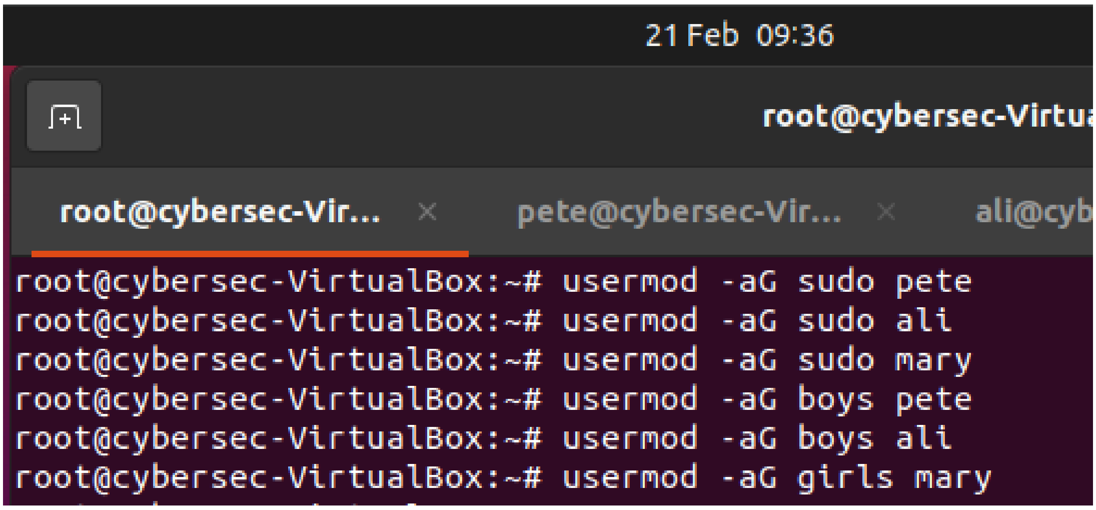

Figure 5.2: Adding the users to sudo and their respective groups.

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

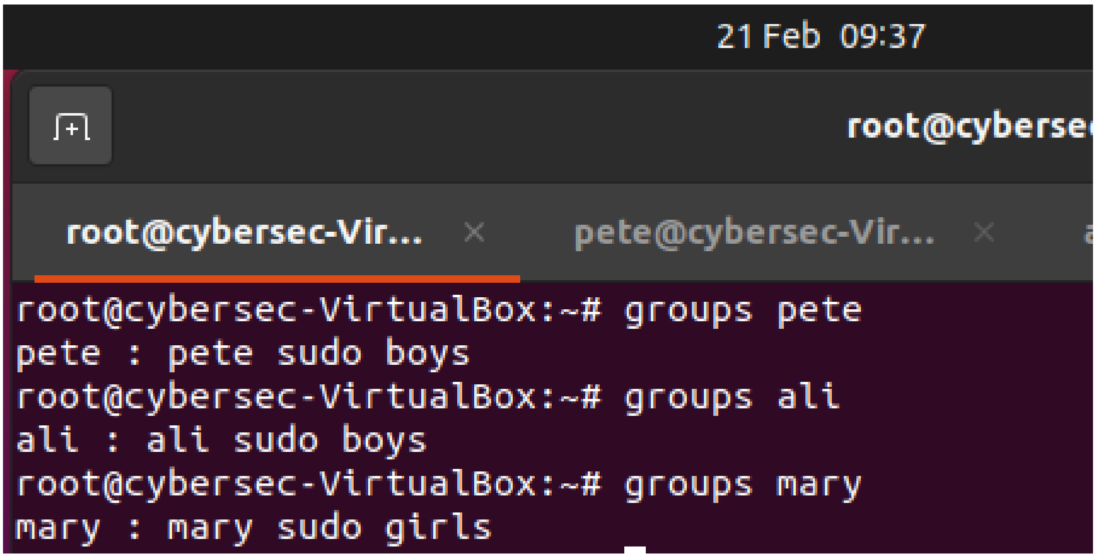

Figure 5.3: Verifying that the users were added to the groups.

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

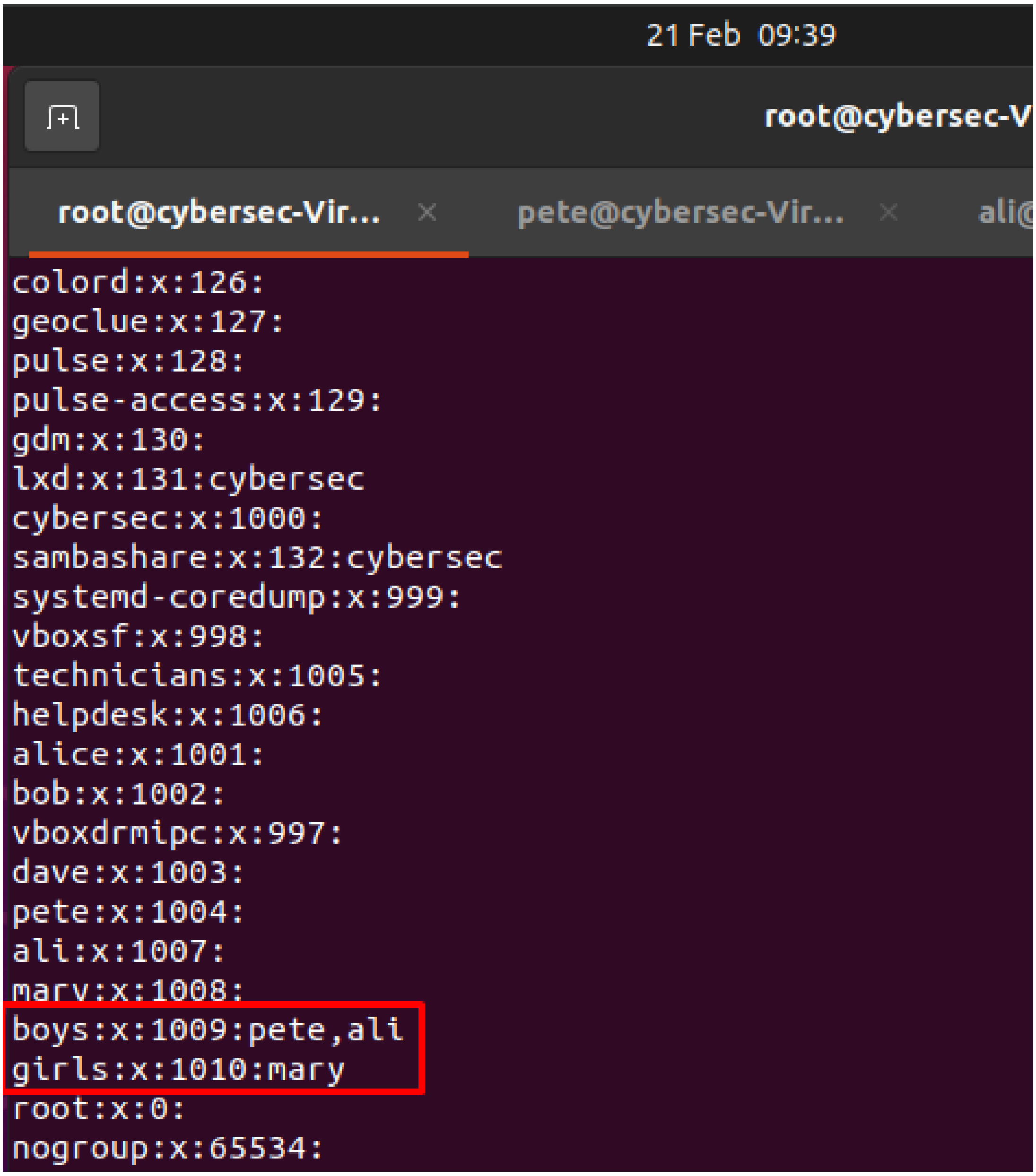

Figure 5.4: Seeing all groups (Boys and Girls are highlighted).

5.2 Using chmod and chgrp to assign permissions

Chmod 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 and other users[1]. I used this help page (NERSC, n.d.) to assist in my learning of these commands as well as access control on UNIX systems.

5.2.1 Restricting directory access

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

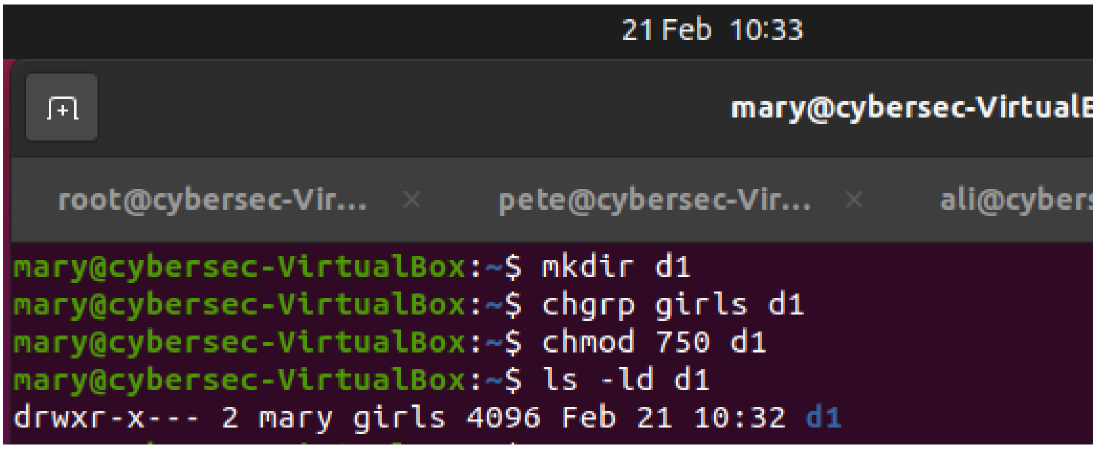

Figure 5.5: Creating D1 and modifying its permissions.

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 not members of the girls group, meaning they are "others".

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

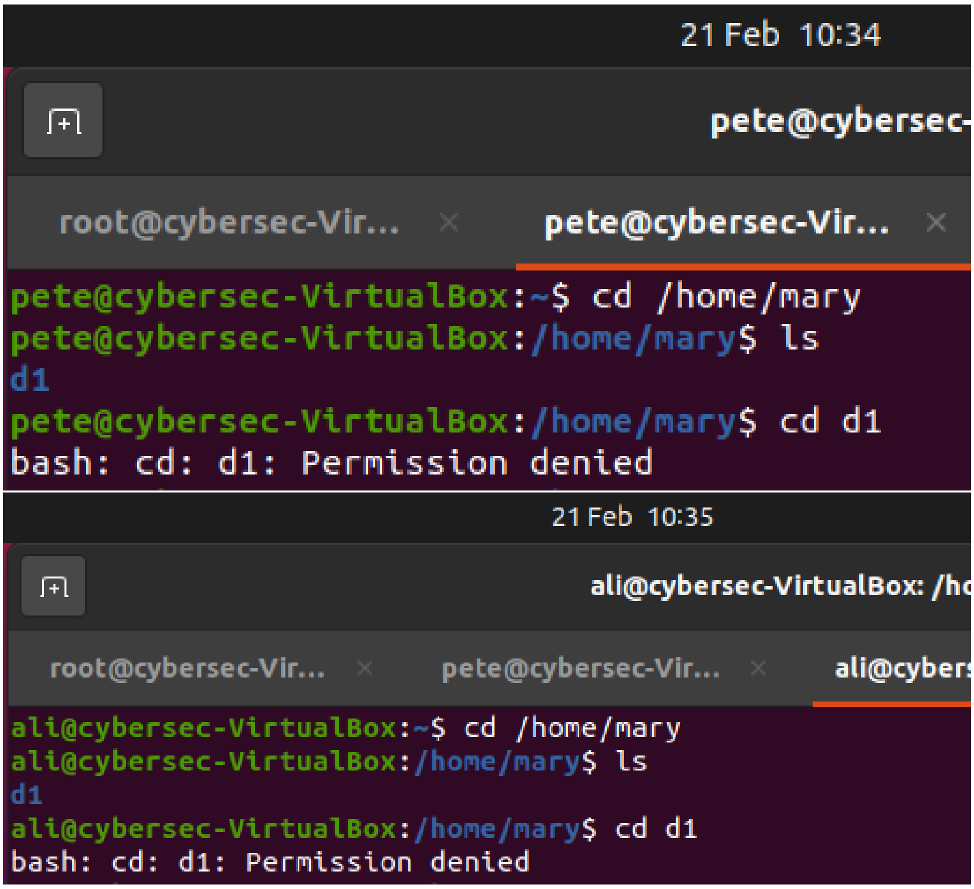

Figure 5.6: Attempting to access D1 as Pete and Ali.

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.

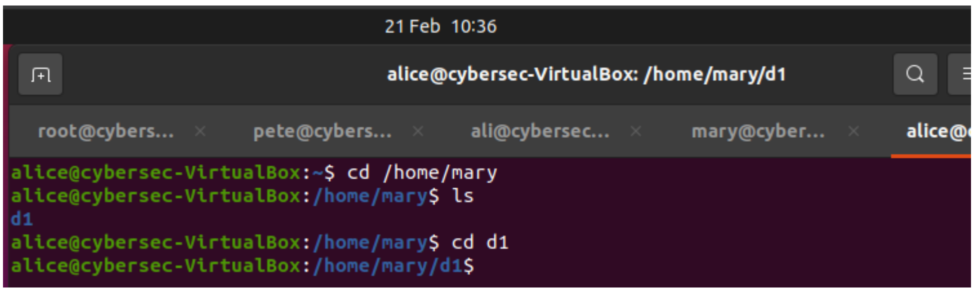

Figure 5.7: Successfully accessing D1 as Alice.

Alice is permitted to read the d1 directory and execute files within it, but she cannot write to it, as intended.

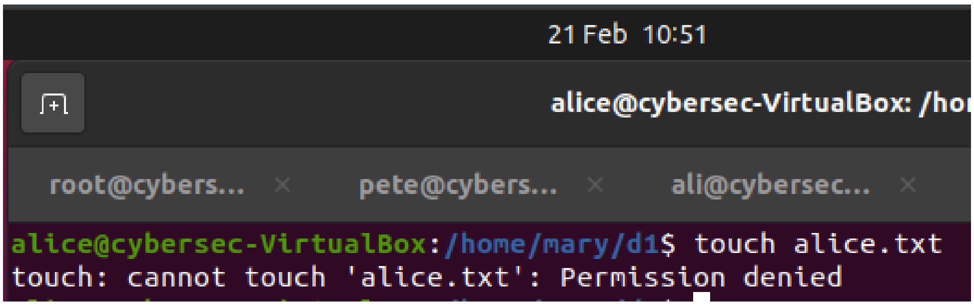

Figure 5.8: Failing to write to D1 as Alice.

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.

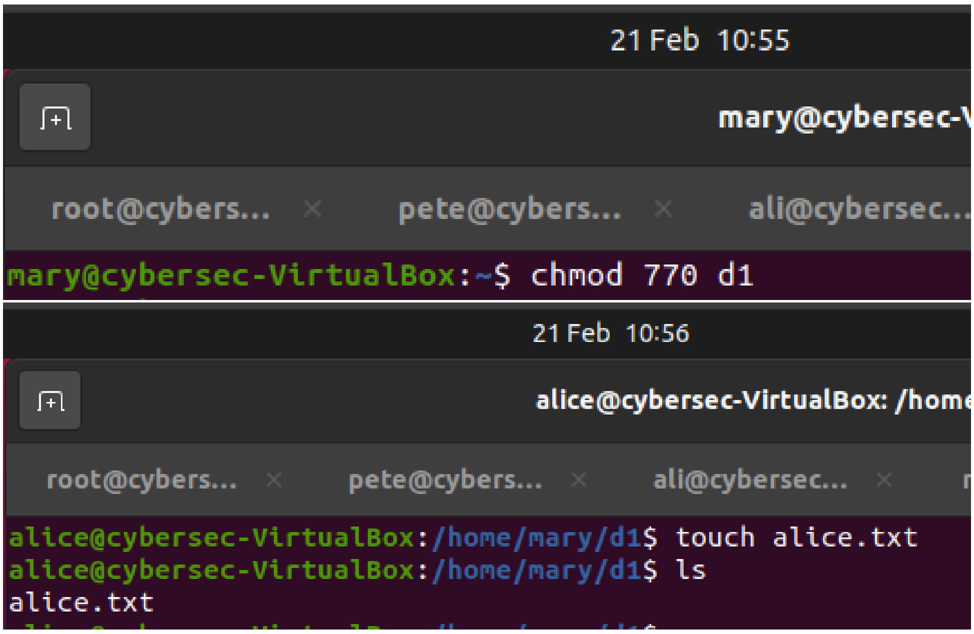

Figure 5.9: Modifying the permissions of D1, allowing Alice to write the file.

[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.

5.2.2 Chgrp and chown

Note

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

Chown 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.

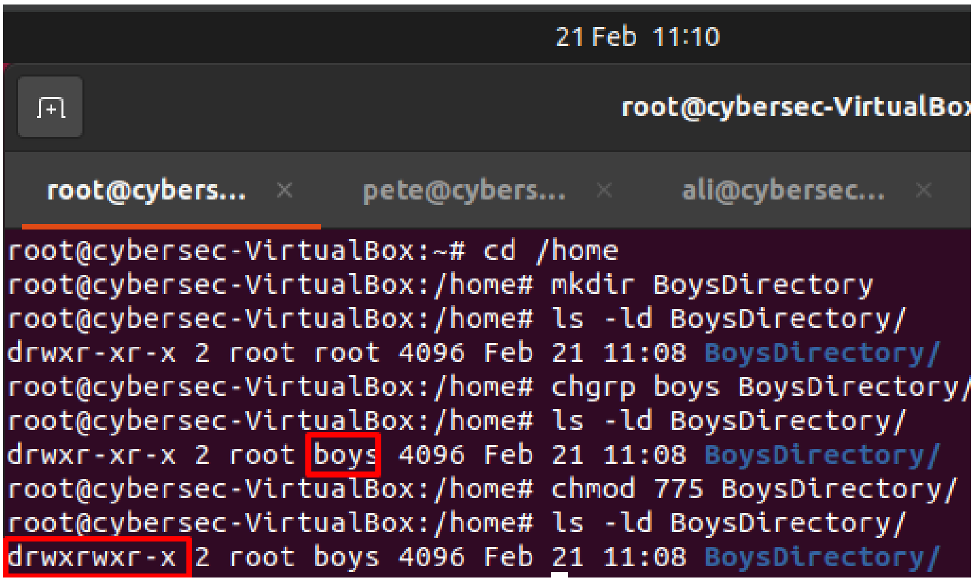

Figure 5.10: Making BoysDirectory and giving group ownership to Boys.

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.

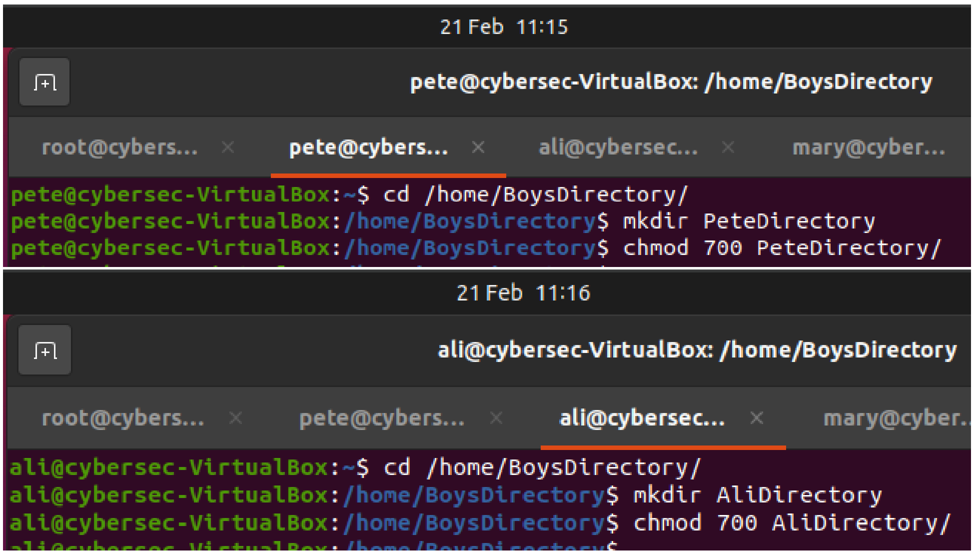

Figure 5.11: Creating and modifying each user's directories and access permissions.

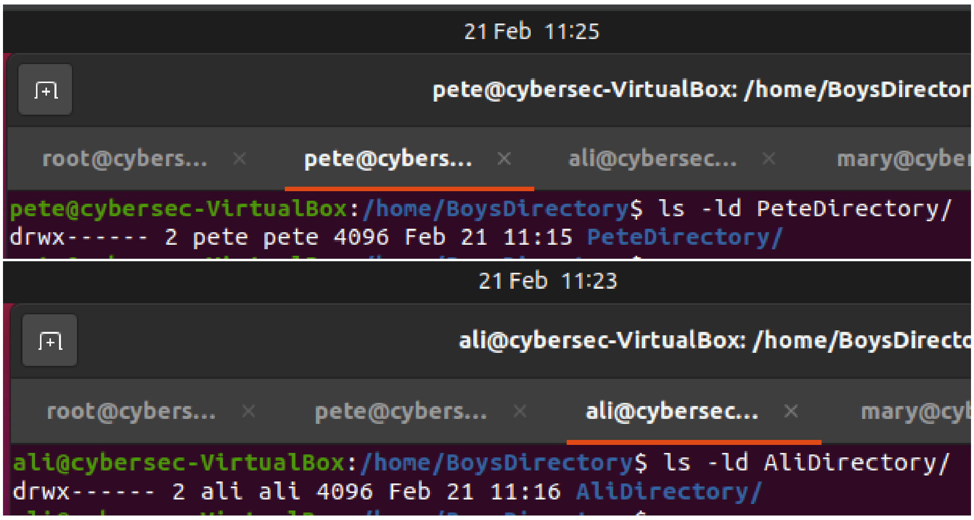

Figure 5.12: Viewing the permissions of Pete and Ali's directories.

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:

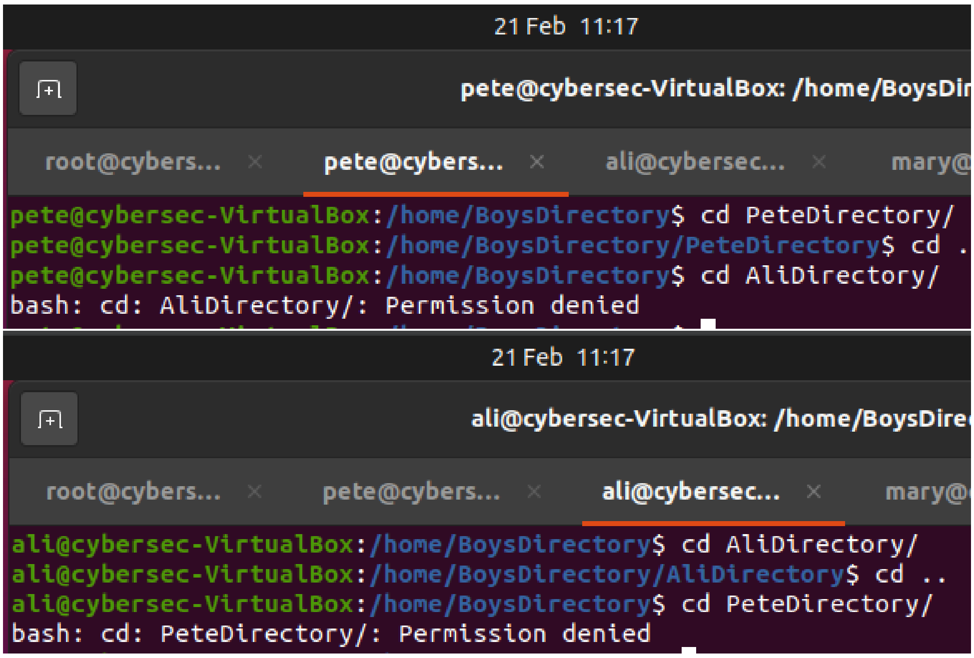

Figure 5.13: Successfully accessing their own directory, but failing to access the other because the user isn't the owner.

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.

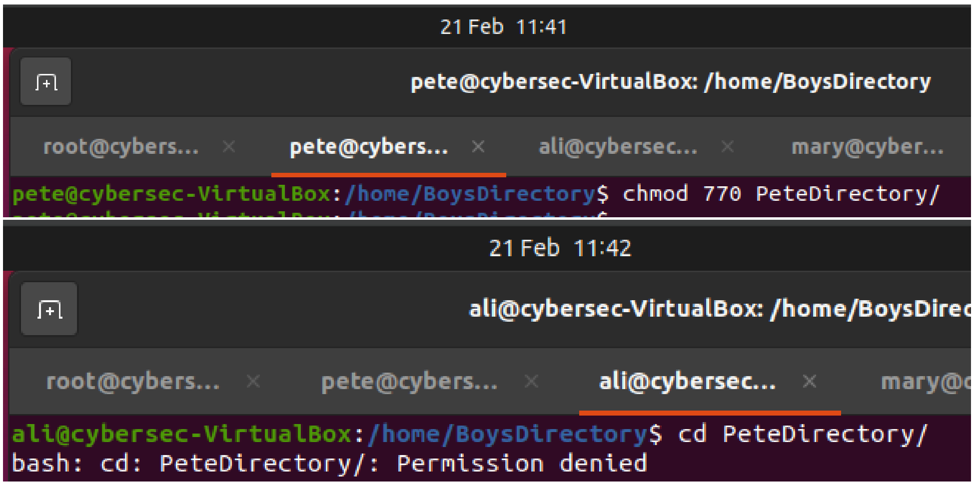

Figure 5.14: Giving group RWX access to PeteDirectory

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.

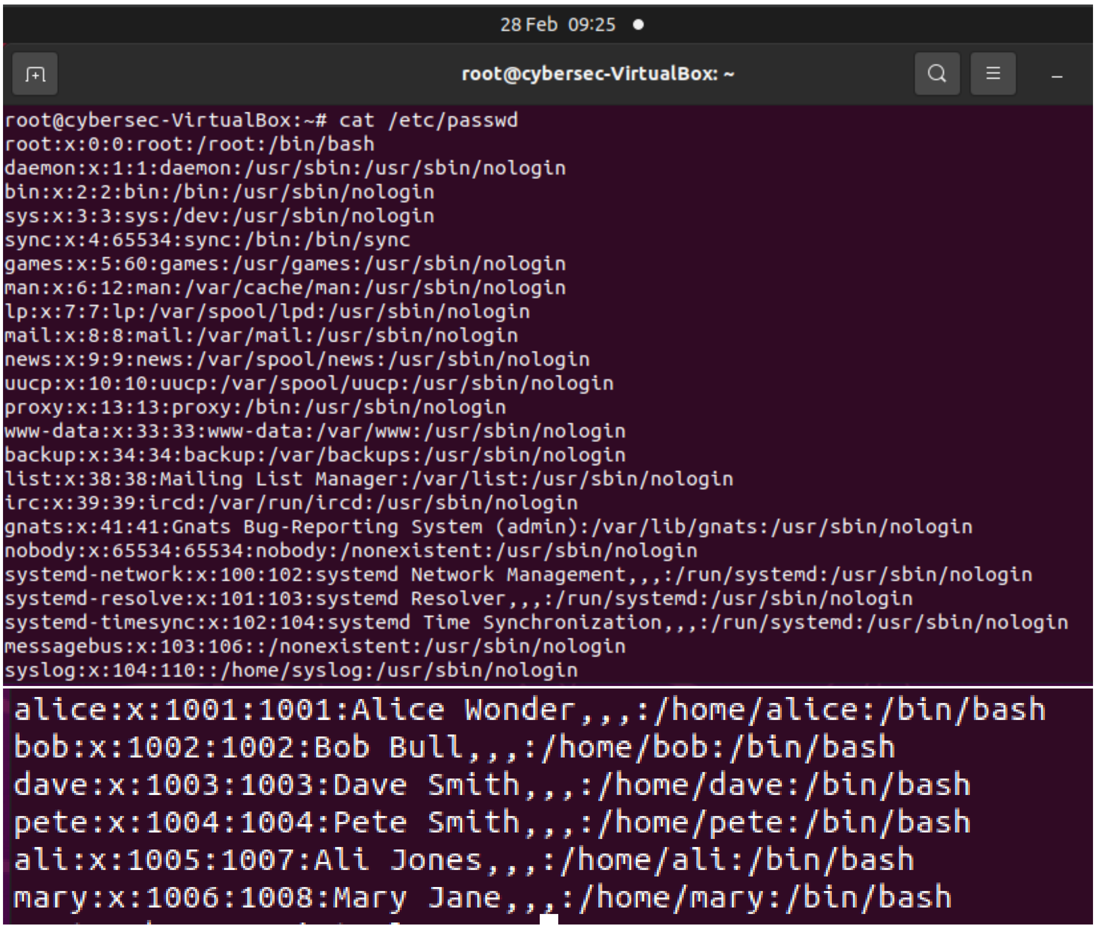

Figure 6.1: Some of the contents of /etc/passwd, with the created users from earlier labs.

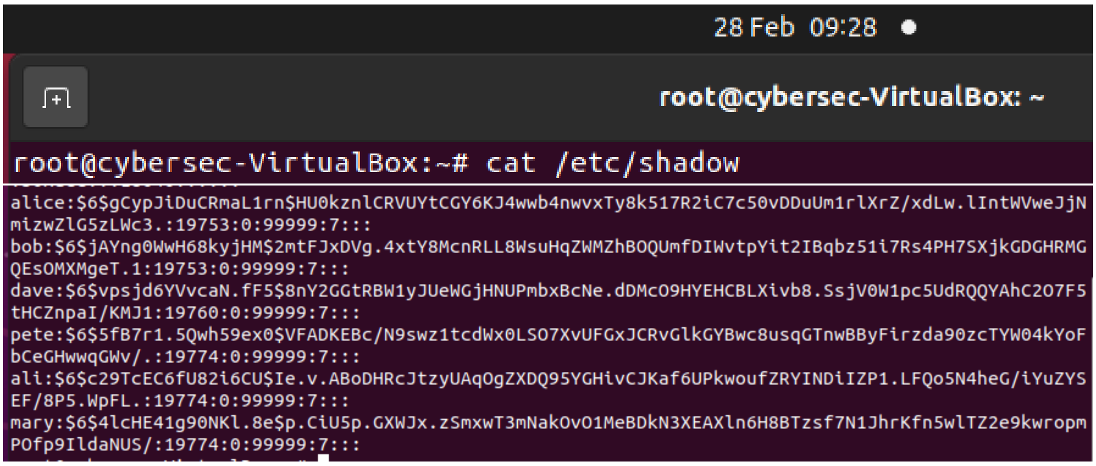

Figure 6.2: Some of the contents of /etc/shadow, with the created users from earlier labs.

6.2 crack.c

The provided code in crack.c is a small program that performs a dictionary attack, cracking hashed passwords by hashing each word in the dictionary, adding the salt and comparing the product to the hashed password. The Ubuntu dictionary is 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.

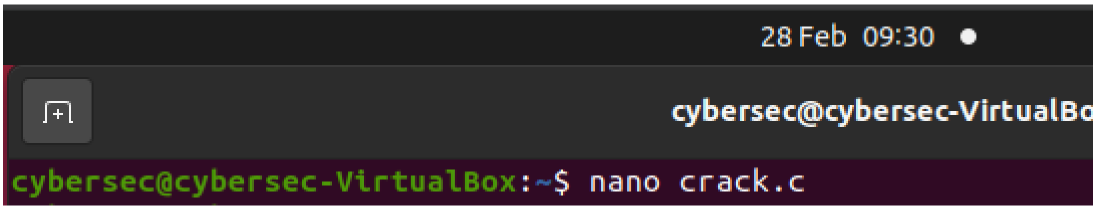

Figure 6.3: Porting crack.c into the VM.

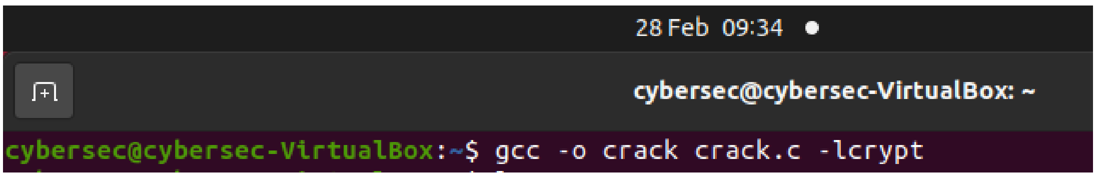

Figure 6.4: Compiling crack.c.

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.

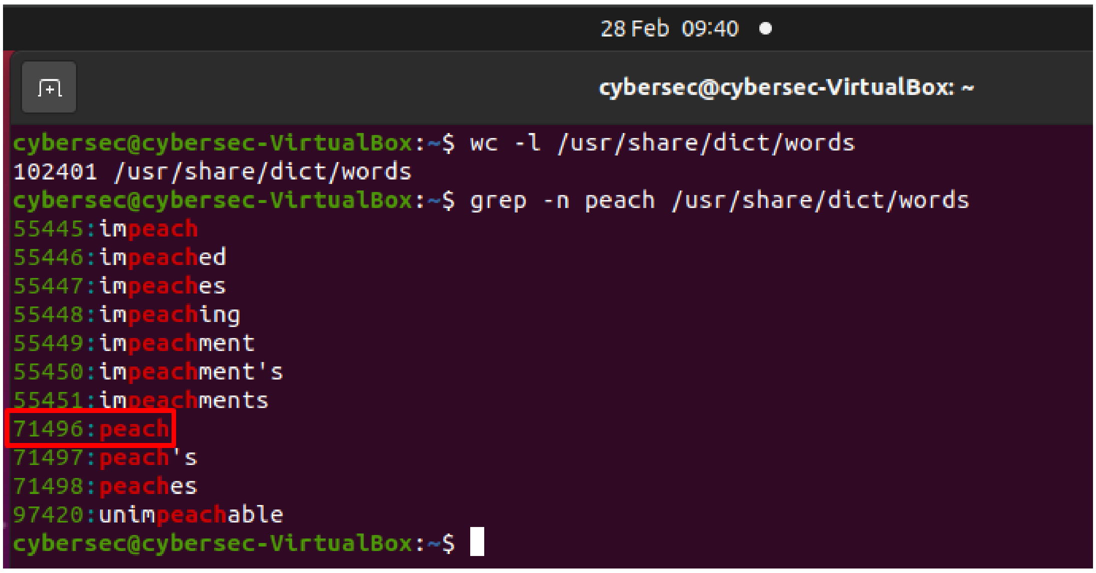

Figure 6.5: Checking the dictionary for the word 'peach', which is the 71496th entry.

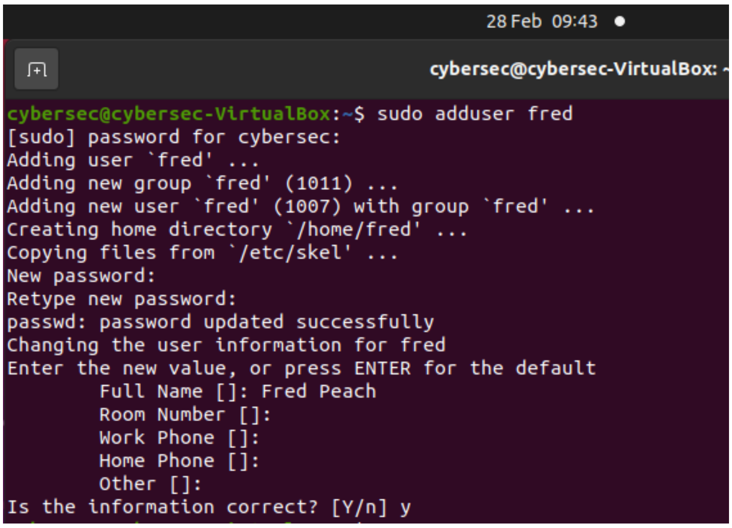

Figure 6.6: Creating the 'fred' user, with the password 'peach'.

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.

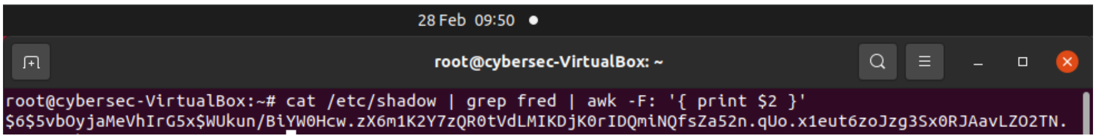

Figure 6.7: Viewing fred's hashed password.

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.

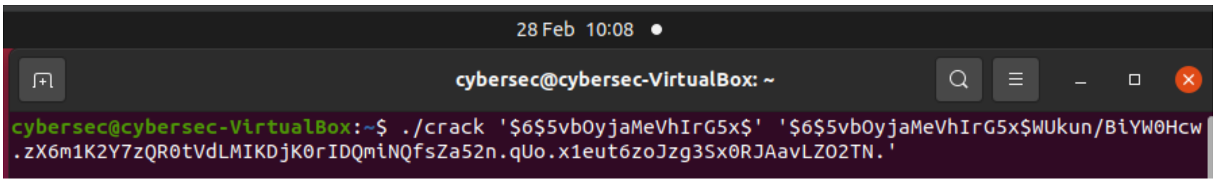

Figure 6.8: Entering the command.

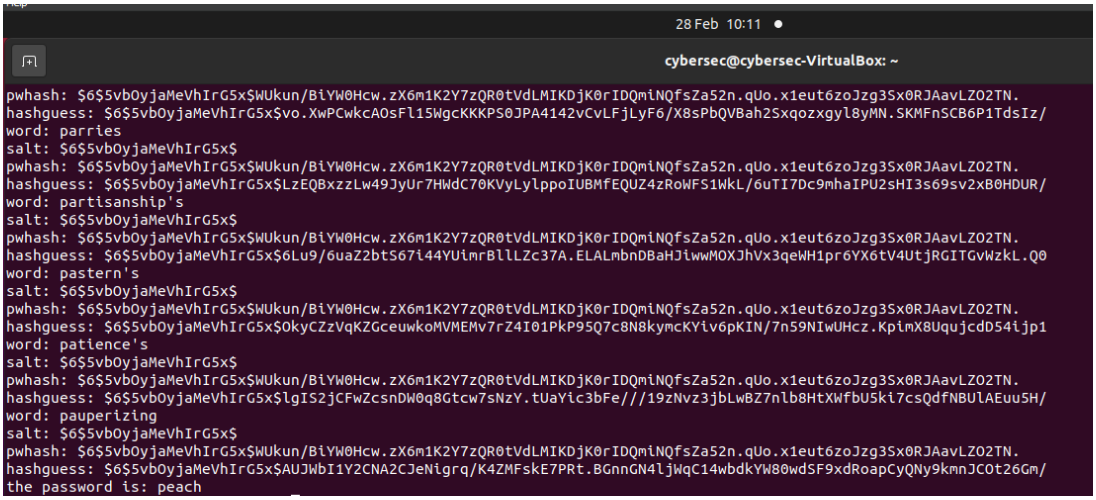

Figure 6.9: The hashed password is revealed as 'peach'.

Note the differing timestamps, showing how this took around 3 minutes to execute. This is because the word 'peach' occurs so late in the dictionary as seen in Figure 6.5.

Reflective report

With the world moving forward into an increasingly digital age, the security of data is paramount for corporations, businesses and general end-users alike. Sensitive data such as bank details and important corporate documents being stored on digital servers inspires countless threat agents to gain unauthorised access to all kinds of devices with each passing year. In 2016, the Identity Theft Resource Center and Cyber Scout reported 1,093 data breach incidents, up 40% from the 780 reported in 2015 (Xu et al., 2018).

It is for this reason that many strategies are employed to secure digital systems, such as cryptography. Cryptography is defined as the technique of obfuscating or coding data (Kaspersky, n.d.), and often refers to encryption in a cybersecurity context, wherein data transmitted and stored on systems is encrypted as an additional layer of security that ensures data cannot be read by anyone other than its intended recipient, even if it is intercepted during transmission or stolen. Asymmetric encryption is especially important, as it allows for non-repudiation, which provides assurance to the sender that its message was delivered, as well as proof of the sender's identity to the recipient. As a result, neither the sender nor the receiver can deny the message was sent and received (Awati, 2021).

While cryptography is essential in the security and integrity of data, it does not come without some disadvantages of its own, especially on portable and/or older, low-performance devices. Constant encryption and decryption of data using strong encryption algorithms such as AES256 can be performance-intensive, causing these devices to lag. Additionally, an unavoidable consequence of the security provided by strong encryption is that the data cannot be recovered without the key. Though this is intentional, there are likely to be scenarios wherein a user has lost their key and therefore all of their encrypted data, as cracking stronger encryption methods is not feasibly possible with current computational power in finite time (Popat and Mehta, 2019).

An additional cybersecurity measure utilised across a wide variety of systems is access control. One variant of this known as Discretionary Access Control (DAC) was showcased and explained in detail in Lab 5 of this logbook. Access control is defined as "an essential element of security that determines who is allowed to access certain data, apps, and resources." (Microsoft, n.d.). By limiting user access to only what is strictly necessary, risk can be significantly mitigated due to users being unable to modify or delete data they are not entitled to, intentionally or not.

Access control is also not without some issues of its own. Access control is predicated on authentication, meaning that if a threat agent were to gain access to an account with superior access than their own via methods such as phishing or password cracking via tools like Hashcat or John the Ripper, they could then access privileged data via impersonation. Additionally, access control is reliant upon authorisation, meaning that if access levels are not correctly set by administrators, it is possible that users could gain access to data without having to gain malicious access at all.

These two techniques are typically used in conjunction with each other on the vast majority of IT systems, which is known as Cryptographic Access Control. This is a vastly superior option to using just one of these techniques, as each technique amplifies the other. Cryptography mitigates access control's impersonation drawback because account passwords would be hashed and salted, making them much harder to brute force. It can also assist in role-based access control, assigning keys dependent upon a user's role and associated privilege level. With the application of cryptographic access control, digital systems can be much more secure and robust in the face of constantly evolving threats.

Bibliography

In order of appearance

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.

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

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).

Xu, M., K. M. Schweitzer, R. M. Bateman and S. Xu (2018). 'Modeling and Predicting Cyber Hacking Breaches'. In: *IEEE Transactions on Information Forensics and Security* 13 (11), pp. 2856–2871. DOI: 10.1109/TIFS.2018.2834227.

Kaspersky (n.d.). *What is Cryptography?* URL: https://www.kaspersky.com/resource-center/definitions/what-is-cryptography (visited on 19/03/2024).

Awati, R. (1st Aug. 2021). *What is nonrepudiation?* URL: https://www.techtarget.com/searchsecurity/definition/nonrepudiation (visited on 19/03/2024).

Popat, J. and U. Mehta (2019). 'Statistical security analysis of AES with X-tolerant response compactor against all types of test infrastructure attacks with/without novel unified countermeasure'. In: *IET circuits, devices & systems* 13 (8), p. 1117. DOI: 10.1049/iet-cds.2019.0083.

Microsoft (n.d.). *What is access control?* URL: https://www.microsoft.com/en-gb/security/business/security-101/what-is-access-control (visited on 19/03/2024).

]