



BIRMINGHAM CITY
University

CMP5329 Logbook

DRAFT VERSION

Lewis Higgins - Student ID 22133848

January - March, 2024

Contents

Introduction	2
1 Lab 1 - OpenSSL	3
1.1 Version checking and ciphers	3
1.2 Symmetric encryption	4
1.2.1 DES symmetric encryption	4
1.2.2 AES256 symmetric encryption and decryption	5
1.3 Asymmetric encryption	6
1.3.1 Generating an RSA private key	6
1.3.2 Storing DES3 & passphrase encrypted RSA keys in a file	7
1.3.3 Getting a public key from the private key	8
1.3.4 Obtaining a message/file digest	9
1.3.5 Signing a digest	10
2 Lab 2 - Usage of GPG	12
3 Lab 5 - Discretionary Access Control	13
4 Lab 6 - Password Cracking	14
Conclusion	15

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.

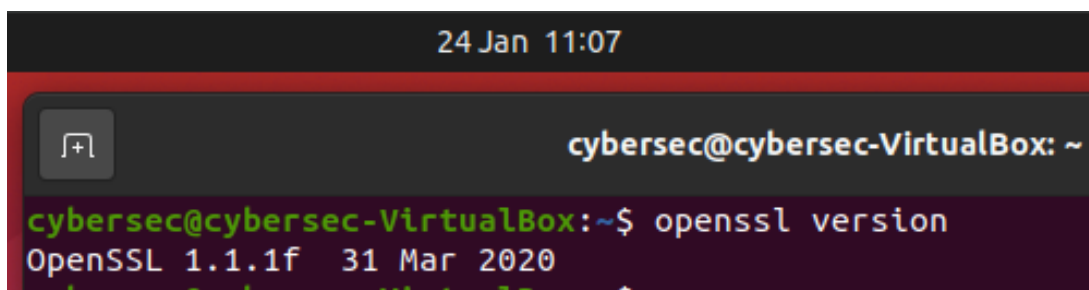
As per module specifications, screenshots taken in each lab include the date and time at which they were taken.

Lab 1 - 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usage of a Linux (Ubuntu distro) VM on a host machine, and the usage of OpenSSL to encrypt and decrypt data using the (outdated) DES algorithm and AES256 symmetric encryption algorithm.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

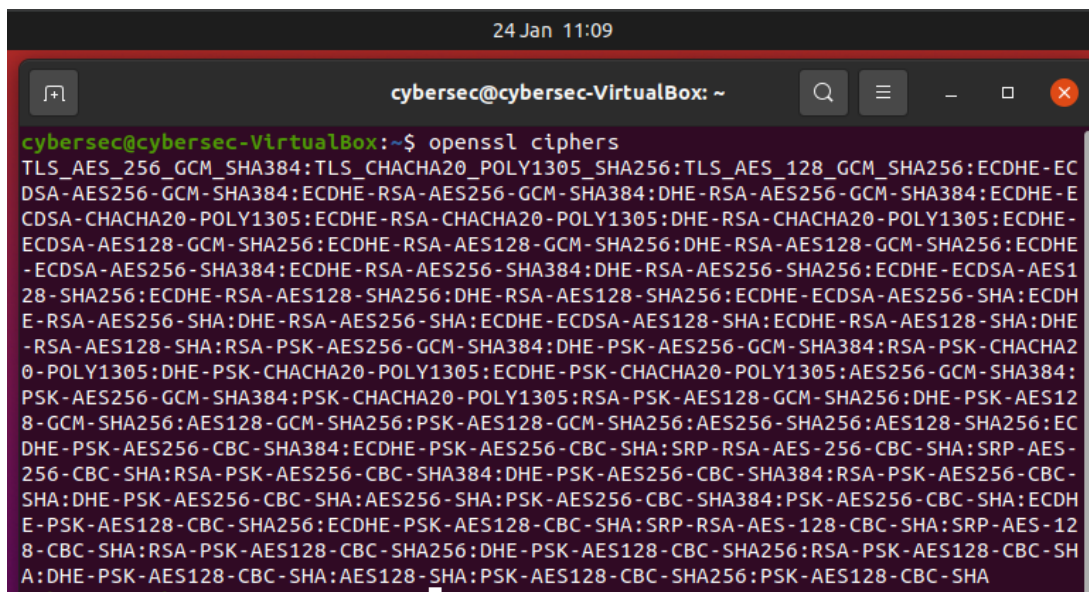
To check the installed version of OpenSSL, the command "openssl version" can be used. The provided virtual machine from [the CMP5329 Moodle page](#) uses OpenSSL version 1.1.1f, dated 31st March 2020.¹



```
24 Jan 11:07
cybersec@cybersec-VirtualBox: ~
cybersec@cybersec-VirtualBox:~$ openssl version
OpenSSL 1.1.1f 31 Mar 2020
```

Figure 1.1: Getting the OpenSSL version

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



```
24 Jan 11:09
cybersec@cybersec-VirtualBox: ~
cybersec@cybersec-VirtualBox:~$ openssl ciphers
TLS_AES_256_GCM_SHA384:TLS_CHACHA20_POLY1305_SHA256:TLS_AES_128_GCM_SHA256:ECDHE-EC
DSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:DHE-RSA-AES256-GCM-SHA384:ECDHE-E
CDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:DHE-RSA-CHACHA20-POLY1305:ECDHE-
ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES128-GCM-SHA256:ECDHE
-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA384:DHE-RSA-AES256-SHA384:ECDHE-ECDSA-AES1
28-SHA256:ECDHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES256-SHA:EC DH
E-RSA-AES256-SHA:DHE-RSA-AES256-SHA:EC DH E-ECDSA-AES128-SHA:EC DH E-RSA-AES128-SHA:DHE
-RSA-AES128-SHA:RSA-PSK-AES256-GCM-SHA384:DHE-PSK-AES256-GCM-SHA384:RSA-PSK-CHACHA2
0-POLY1305:DHE-PSK-CHACHA20-POLY1305:EC DH E-PSK-CHACHA20-POLY1305:AES256-GCM-SHA384:
PSK-AES256-GCM-SHA384:PSK-CHACHA20-POLY1305:RSA-PSK-AES128-GCM-SHA256:DHE-PSK-AES12
8-GCM-SHA256:AES128-GCM-SHA256:PSK-AES128-GCM-SHA256:AES256-SHA256:AES128-SHA256:EC
DHE-PSK-AES256-CBC-SHA384:EC DH E-PSK-AES256-CBC-SHA:SRP-RSA-AES-256-CBC-SHA:SRP-AES-
256-CBC-SHA:RSA-PSK-AES256-CBC-SHA384:DHE-PSK-AES256-CBC-SHA384:RSA-PSK-AES256-CBC-
SHA:DHE-PSK-AES256-CBC-SHA:AES256-SHA:PSK-AES256-CBC-SHA384:PSK-AES256-CBC-SHA:EC DH
E-PSK-AES128-CBC-SHA256:EC DH E-PSK-AES128-CBC-SHA:SRP-RSA-AES-128-CBC-SHA:SRP-AES-12
8-CBC-SHA:RSA-PSK-AES128-CBC-SHA256:DHE-PSK-AES128-CBC-SHA256:RSA-PSK-AES128-CBC-SH
A:DHE-PSK-AES128-CBC-SHA:AES128-SHA:PSK-AES128-CBC-SHA256:PSK-AES128-CBC-SHA
```

Figure 1.2: Getting the OpenSSL ciphers

¹This 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

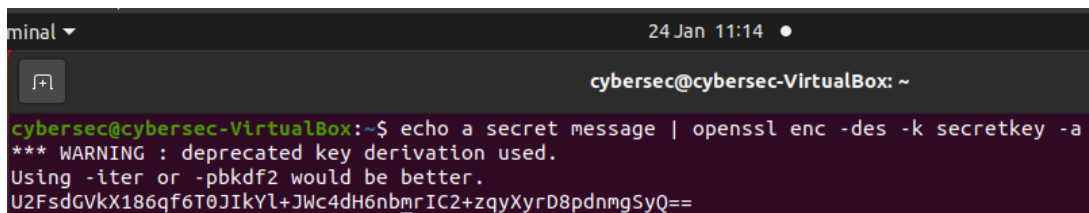
Symmetric cryptography refers to the process of transferring data that has been encrypted by a single key. Both the sender and receiver of this data use the same key to encrypt and decrypt the data. Symmetric encryption does not allow for non-repudiation, as it cannot necessarily be proven that data 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:

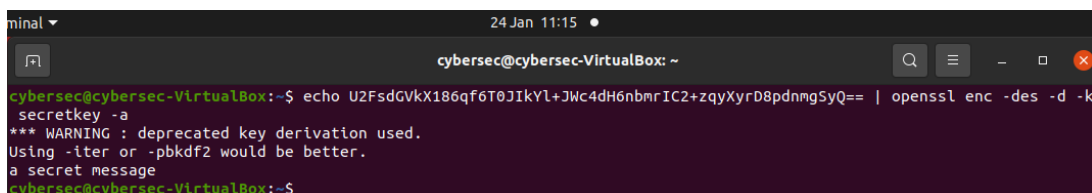
- enc - Encode
- -des - Using the DES algorithm
- -k secretkey - Using the key "secretkey"
- -a - Writes the ciphertext in base64, allowing it to be properly displayed. If this flag isn't used, the text will not be legible, showing question mark symbols instead.



```
minal 24 Jan 11:14
cybersec@cybersec-VirtualBox: ~
cybersec@cybersec-VirtualBox:~$ echo a secret message | openssl enc -des -k secretkey -a
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
U2FsdGVkX186qf6T0JIKyL+JWc4dH6nbmRIC2+zqyXyrD8pdnmgSyQ==
```

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.



```
minal 24 Jan 11:15
cybersec@cybersec-VirtualBox: ~
cybersec@cybersec-VirtualBox:~$ echo U2FsdGVkX186qf6T0JIKyL+JWc4dH6nbmRIC2+zqyXyrD8pdnmgSyQ== | openssl enc -des -d -k secretkey -a
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
a secret message
cybersec@cybersec-VirtualBox:~$
```

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:

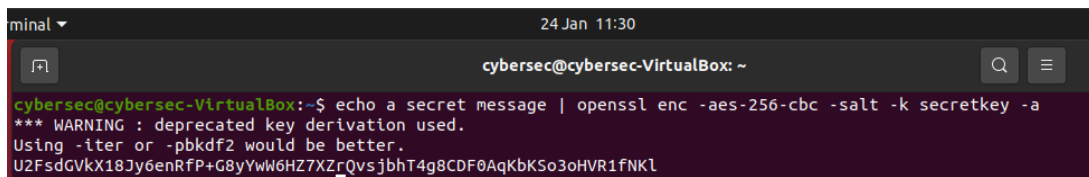
- The ciphertext is passed to OpenSSL rather than the plaintext, as we typically wouldn't know the original plaintext in this scenario, which is what we are aiming to achieve by decrypting it.
- -d - Indicates that the text should be decrypted rather than encrypted.

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.



```
minal 24 Jan 11:30
cybersec@cybersec-VirtualBox: ~
cybersec@cybersec-VirtualBox:~$ echo a secret message | openssl enc -aes-256-cbc -salt -k secretkey -a
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
U2FsdGVkX18Jy6enRfP+G8yYwW6HZ7XZrQvsjbhT4g8CDF0AqKbKSo3oHVR1fNKl
```

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.

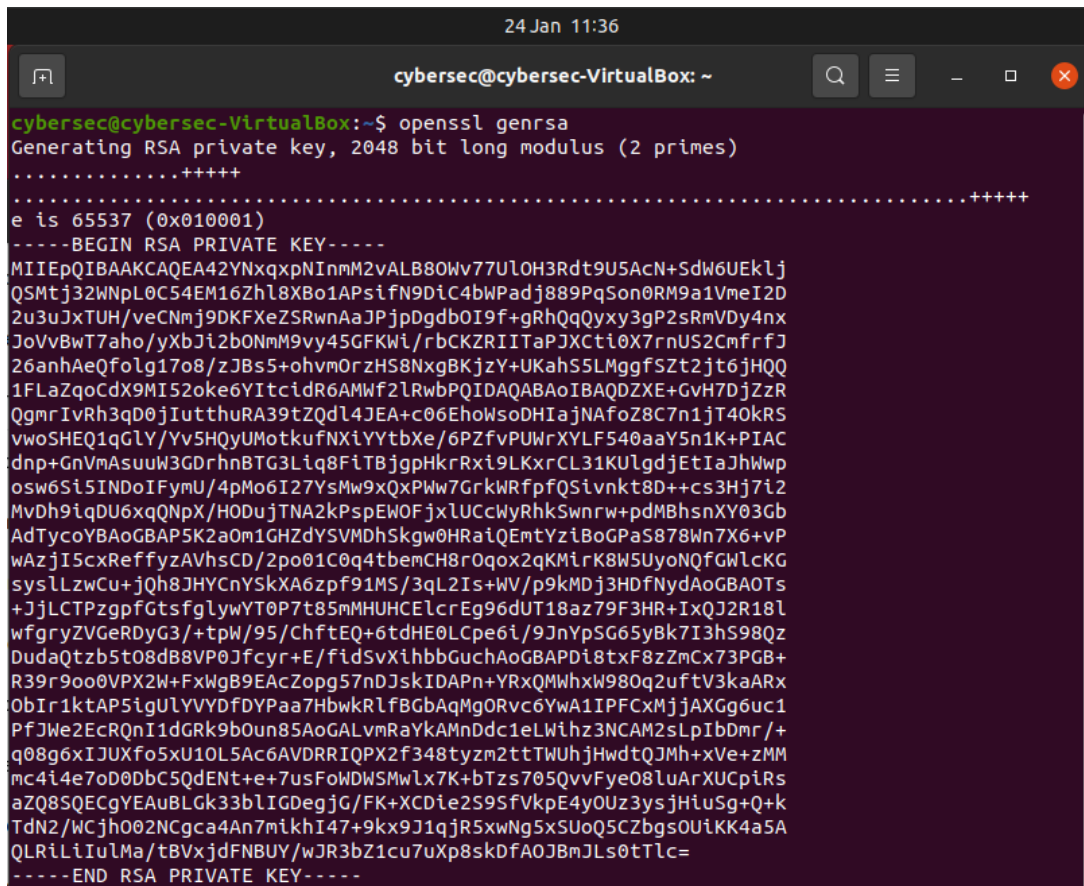
1.3 Asymmetric encryption

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.



```

24 Jan 11:36
cybersec@cybersec-VirtualBox: ~
cybersec@cybersec-VirtualBox:~$ openssl genrsa
Generating RSA private key, 2048 bit long modulus (2 primes)
.....+++++
.....+++++
e is 65537 (0x010001)
-----BEGIN RSA PRIVATE KEY-----
MIIEpQIBAAKCAQEAA42YNxqxpNInmM2vAlB80Wv77U1OH3Rdt9U5AcN+SdW6UEklj
QSMtj32WNpL0C54EM16Zhl8XBo1APsifN9DiC4bwPadj889PqSon0RM9a1VmeI2D
2u3uJxTUH/veCNmj9DKFXeZSRwnAaJPjpDgdbOI9f+gRhQqQxy3gP2sRmVDy4nx
JoVvBwT7aho/yXbJi2bONmM9vy45GFKWi/rbCKZRITaPJXCTi0X7rnUS2CmfrfJ
26anhAeQfo1g17o8/zJBs5+ohvmOrzHS8NxbKjzY+UKahS5LMggf5Zt2jt6jHQQ
1FLaZqoCdX9MI52oke6YItcIdR6AMWf2lRwbPQIDAQABAOIBAQDZXE+GvH7DjZzR
QgmrIvRh3Qd0jIutthuRA39tZQdl4JEA+c06EhoWsoDHIAjNafoZ8C7n1jt40kRS
vwoSHEQ1qGLY/Yv5HQyUMotkufNXiYytbXe/6PZfvPUWrXYLF540aaY5n1K+PIAC
dnp+GnVmAsuuW3GDrhnBTG3Liq8FiTBjgphKrRxI9LKxrCL31KULgdjEtIaJhWwp
osw6Si5INDoIFymU/4pMo6I27YsMw9xQxPw7GrkWRfpfQSiVnkt8D++cs3Hj7i2
MvDh9iqDU6xqQNPx/HODujTNA2kPspEWOFjxLUCCwYRhKSwrnrw+pdMBhsnXY03Gb
AdTycoYBAoGBAP5K2aOm1GHZdYSVMDhSkGw0HRaiQEmtYziBoGPas87Wn7X6+vP
wAzjIScxReffyzAVhsCD/2po01C0q4tbemCH8r0qox2qKMirK8W5UyoNQfGWLcKG
sysLLzwCu+jQh8JHYCnYSkXA6zpf91MS/3qL2Is+WV/p9kMDj3HDfNydaoGBAOTs
+JjLCTPzgpfgtsfgyWYT0P7t85mMHUHCeLcrEg96dUT18az79F3HR+IxQJ2R18L
wfgryZVGeRdyG3/+tpw/95/ChftEQ+6tdHE0LCpe6i/9JnYpSG65yBk7I3hS98QZ
DudaQtzb5t08dB8VP0Jfcyr+E/fldsvXiHbbGuchAoGBAPDi8txF8zZmCx73PGB+
R39r9oo0VPX2W+FxWgB9EAcZopg57nDJsKIDAPn+YRXQMWhxW980q2uftV3kaARx
ObIr1ktAP5igULVYDFDYpaa7HbwkRlFBGbaQMgORvc6YwA1IPFCxMjjAXGg6uc1
PfJWe2EcrQnI1dGRk9b0un85AoGALvmRaYkAMnDdc1eLWihz3NCAM2sLpIbDmr/+
q08g6xIJUXFo5xU10L5Ac6AVDRRIQPX2f348tyzm2ttTWuhjHwdtQJmH+xVe+zMM
mc4i4e7oD0DbC5QdENT+e+7usFoWDSMWlx7K+bTzs705QvvFye08luArXUCpiRs
azQ8S5EQCgYEAuBLGk33bLIGDegjG/FK+XCDie2S9SfVkpE4yOUz3ysjHtuSg+Q+k
TdN2/Wcjho02NCgca4An7mikhI47+9kx9J1qjR5xwNg5xSUoQ5CZbgsOUiKK4a5A
QLRiLiIuLma/tBVxjdfNBuY/wJR3bZ1cu7uXp8skDfAOJBMJLs0tTlc=
-----END RSA PRIVATE KEY-----

```

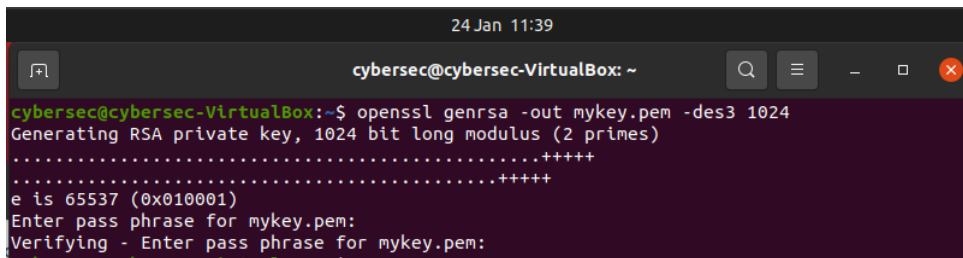
Figure 1.6: Generating a 2048-bit private key using genrsa.

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".



```

24 Jan 11:39
cybersec@cybersec-VirtualBox: ~
cybersec@cybersec-VirtualBox:~$ openssl genrsa -out mykey.pem -des3 1024
Generating RSA private key, 1024 bit long modulus (2 primes)
.....+++++
.....+++++
e is 65537 (0x010001)
Enter pass phrase for mykey.pem:
Verifying - Enter pass phrase for mykey.pem:

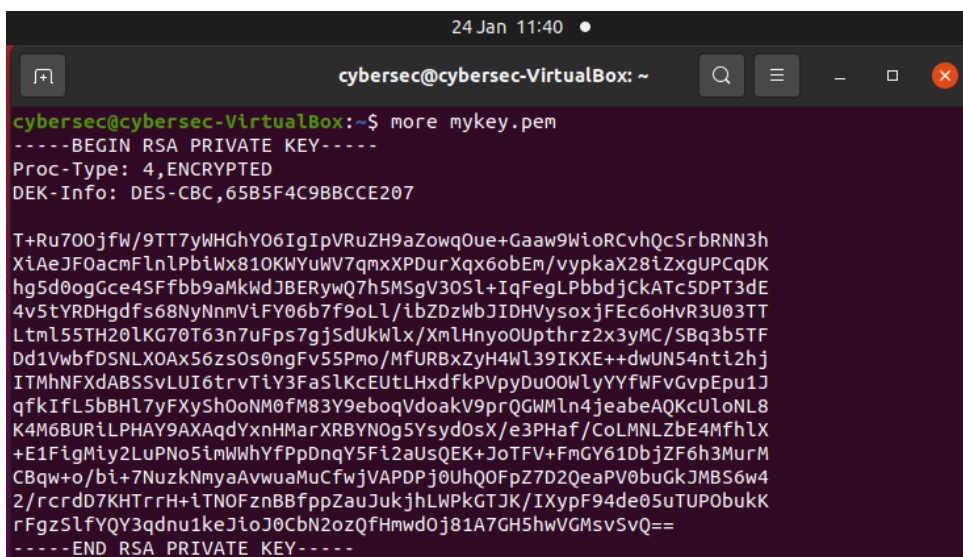
```

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".



```

24 Jan 11:40
cybersec@cybersec-VirtualBox: ~
cybersec@cybersec-VirtualBox:~$ more mykey.pem
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4, ENCRYPTED
DEK-Info: DES-CBC,65B5F4C9BBCCE207

T+Ru700jfw/9TT7yWHGhY06IgiPVRuZH9aZowqOue+Gaaw9WioRCvhQcSrbrRNN3h
XiAeJF0acmFlnlPbiWx810KWYUwV7qmxXPDurXqx6obEm/vypkaX28iZxgUPCqDK
hg5d0ogGce4SFfbb9aMkwdJBERywQ7h5MSgV30Sl+IqFegLPbbdjckATc5DPT3dE
4v5tYRDHgdFs68NyNnmViFY06b7f9oLl/lbZDzWbJIDHvysoxjFEc6oHvR3U03TT
Ltm15STH20lKG70T63n7uFps7gjSduKwLx/Xm1Hnyo0Upthr22x3yMC/SBq3b5TF
Dd1VwbFD5NLX0Ax56zs0s0ngFv55Pmo/MFURBxZyH4Wl39IKXE++dwUN54nti2hj
ITMhNFXdABSSvLUI6trvtiY3FaSLKcEutLHxdfkPvPyDu00WlyYYfWfVgVpEpu1J
qfkiFl5bBHl7yFXySh0oNM0fM83Y9eboqVdoakV9prQGWMLn4jeabeAQKcUloNL8
K4M6BURiLPHAY9AXAqdYxnHMarXRBYN0g5Ysyd0sX/e3PHaf/CoLMNLZbE4MfhLX
+E1FiGmIy2LuPno5imWWhYfPpDnqY5Fi2aUsQEK+JoTFV+FmGY61DbjZF6h3MurM
CBQw+o/bi+7NuzkNmyaAvwuaMuCfwjVAPDPj0UHQFpZ7D2QeaPV0buGkJMS6w4
2/rcrd07KHTrrH+iTNOFznBBfppZauJukjhLWPkGTJK/IXypF94de05uTUP0buk
rFgzSLfYQY3qdnui1keJioJ0CbN2ozQfHmwd0j81A7GH5hwVGMsvSvQ==
-----END RSA PRIVATE KEY-----

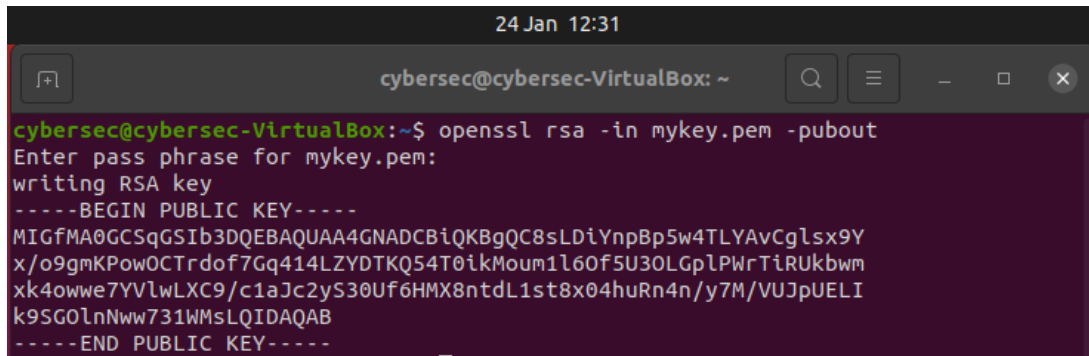
```

Figure 1.8: The key stored in mykey.pem.

³The 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

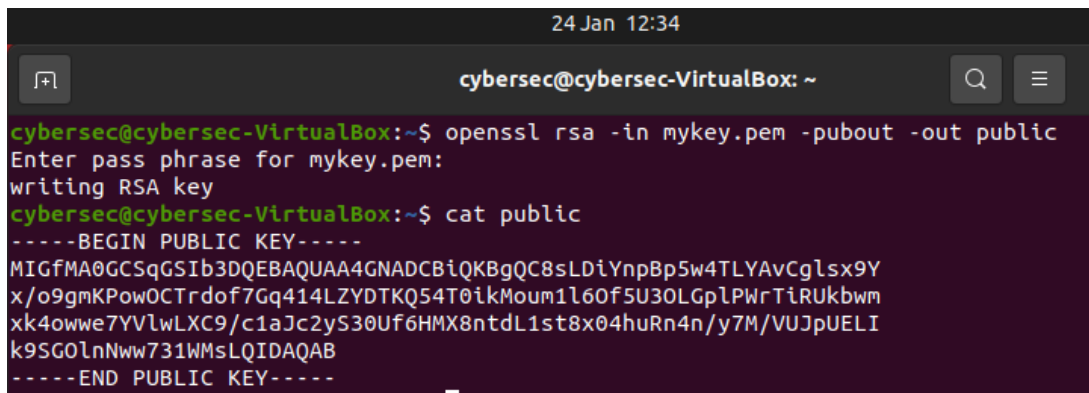
The private key stored into "mykey.pem" by the previous command can be accessed again to generate a public key, which would be used when data is encrypted. To do so, the passphrase we set earlier (secretkey) must be entered.



```
24 Jan 12:31
cybersec@cybersec-VirtualBox: ~
cybersec@cybersec-VirtualBox:~$ openssl rsa -in mykey.pem -pubout
Enter pass phrase for mykey.pem:
writing RSA key
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC8sLDiYnpBp5w4TLyAvCglsx9Y
x/o9gmKPowOCTrdof7Gq414LZYDTKQ54T0ikMoum1l6Of5U30LGplPwrTiRukbwm
xk4owwe7YVlwLXC9/c1aJc2yS30Uf6HMX8ntdL1st8x04huRn4n/y7M/VUJpUELI
k9SGOlNnw731WmsLQIDAQAB
-----END PUBLIC KEY-----
```

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.



```
24 Jan 12:34
cybersec@cybersec-VirtualBox: ~
cybersec@cybersec-VirtualBox:~$ openssl rsa -in mykey.pem -pubout -out public
Enter pass phrase for mykey.pem:
writing RSA key
cybersec@cybersec-VirtualBox:~$ cat public
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC8sLDiYnpBp5w4TLyAvCglsx9Y
x/o9gmKPowOCTrdof7Gq414LZYDTKQ54T0ikMoum1l6Of5U30LGplPwrTiRukbwm
xk4owwe7YVlwLXC9/c1aJc2yS30Uf6HMX8ntdL1st8x04huRn4n/y7M/VUJpUELI
k9SGOlNnw731WmsLQIDAQAB
-----END PUBLIC KEY-----
```

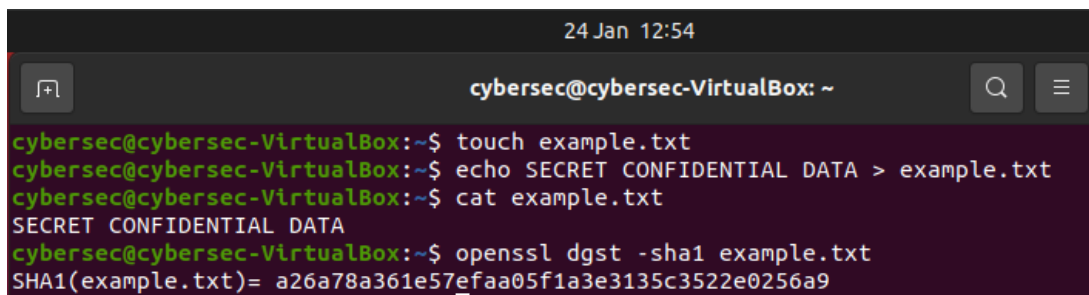
Figure 1.10: Storing the public key in a file.

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.

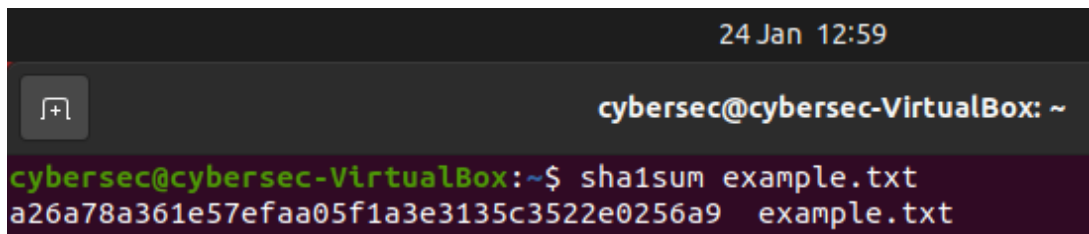
OpenSSL can generate digests using its "dgst" command.



```
24 Jan 12:54
cybersec@cybersec-VirtualBox: ~
cybersec@cybersec-VirtualBox:~$ touch example.txt
cybersec@cybersec-VirtualBox:~$ echo SECRET CONFIDENTIAL DATA > example.txt
cybersec@cybersec-VirtualBox:~$ cat example.txt
SECRET CONFIDENTIAL DATA
cybersec@cybersec-VirtualBox:~$ openssl dgst -sha1 example.txt
SHA1(example.txt)= a26a78a361e57efaa05f1a3e3135c3522e0256a9
```

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`.



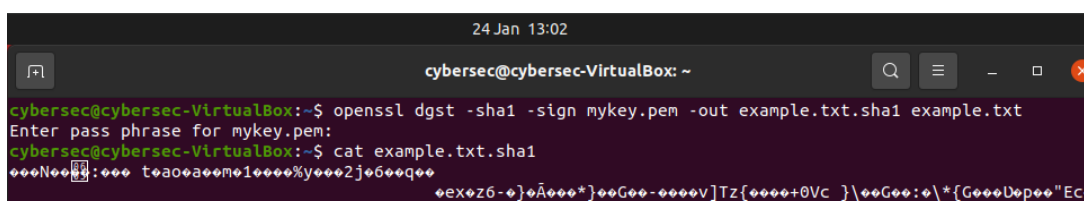
```
24 Jan 12:59
cybersec@cybersec-VirtualBox: ~
cybersec@cybersec-VirtualBox:~$ sha1sum example.txt
a26a78a361e57efaa05f1a3e3135c3522e0256a9 example.txt
```

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.

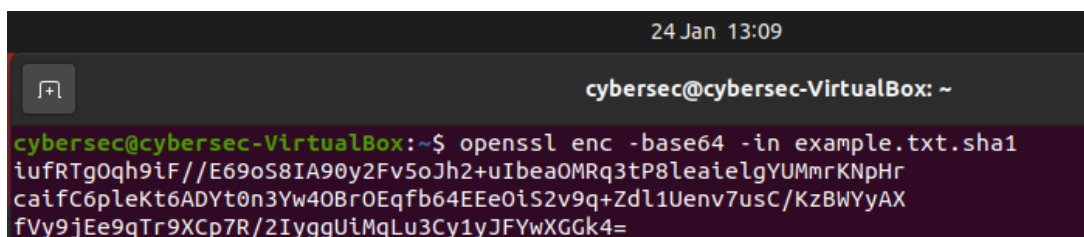


```

24 Jan 13:02
cybersec@cybersec-VirtualBox: ~
cybersec@cybersec-VirtualBox:~$ openssl dgst -sha1 -sign mykey.pem -out example.txt.sha1 example.txt
Enter pass phrase for mykey.pem:
cybersec@cybersec-VirtualBox:~$ cat example.txt.sha1
-----BEGIN-----
tAo0a0m01000%y0002j0600q00
0ex0z6-0}0Ã000*}00G00-0000v]Tz{0000+0Vc }00G00:0}*{G000D0p00"E0c0
-----END-----
  
```

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.



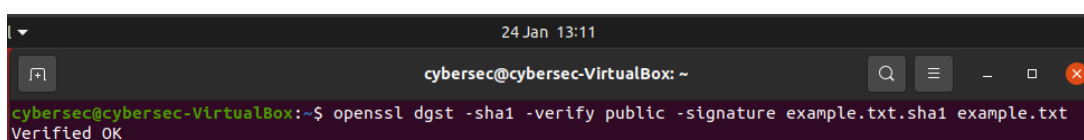
```

24 Jan 13:09
cybersec@cybersec-VirtualBox: ~
cybersec@cybersec-VirtualBox:~$ openssl enc -base64 -in example.txt.sha1
iufRTg0qh9iF//E69oS8IA90y2Fv5oJh2+uIbeaOMRq3tP8leaieIgyUMmrKNpHr
caifC6pleKt6ADYt0n3Yw40Br0Eqfb64EEe0iS2v9q+Zd1lUenv7usC/KzBWYyAX
fVy9jEe9qTr9XCp7R/2IyggUiMqLu3Cy1yJFYwXGGk4=
  
```

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.

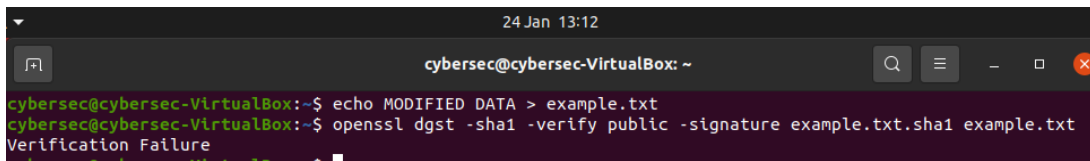


```

24 Jan 13:11
cybersec@cybersec-VirtualBox: ~
cybersec@cybersec-VirtualBox:~$ openssl dgst -sha1 -verify public -signature example.txt.sha1 example.txt
Verified OK
  
```

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.



```
24 Jan 13:12
cybersec@cybersec-VirtualBox: ~
cybersec@cybersec-VirtualBox:~$ echo MODIFIED DATA > example.txt
cybersec@cybersec-VirtualBox:~$ openssl dgst -sha1 -verify public -signature example.txt.sha1 example.txt
Verification Failure
```

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.

Lab 2 - Usage of GPG

Lab 5 - Discretionary Access Control

Lab 6 - Password Cracking

Conclusion

Text text text text

Bibliography

- 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](https://doi.org/10.1016/S0169-7552(96)00089-X).