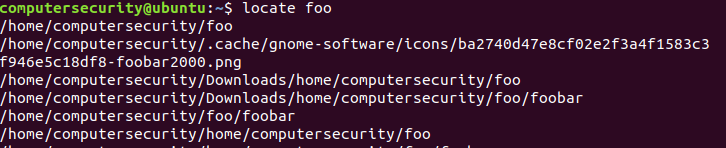
**Computer Security I Lab 6**

Welcome to the 6th lab within the Computer Security I course in this lab we are going to go over some more Linux commands which are important to understand. I understand that the last lab was a lot and this lab will also be a lot also so there is no time to waste, Let’s get down to the nitty gritty with the **locate** command.

**The Locate Command**

In Linux we have a couple of options in order to find a file or a directory we can use the “**find”** command or the **“locate”** command but since the find command is a little bit trickier and more complex we are going to use the locate command for this portion of the tutorial. In addition the locate command is faster and looks through a database within the virtual machine opposed to searching the entire file system(which is what the find command does).

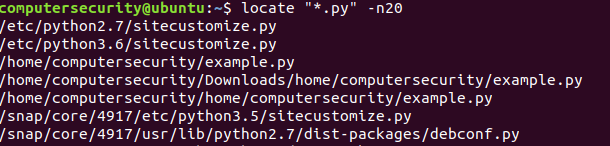
For the first example we are going to use the locate command and pass it through a file in my case it is going to be “foo” this will display all the times where foo is shown in Linux. Check out the screenshot below:  


**Figure 6.1:** Search for the word foo throughout the Ubuntu virtual machine

As you can see this is a straight forward command there isn’t much to it besides some flags and different file parameters.

In the above command we ran the locate command passed it parameter foo so it outputted in the terminal everything with foo.

Another helpful command is that we are able to limit the search queries to a specific number by using the –n flag for example if we want 20 python files returned (instead of 500+ files) we can just use the –n flag. In the screenshot below we show this flag works:



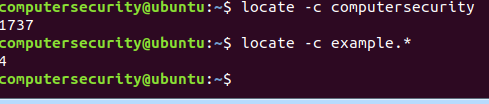
**Figure 6.2:** Search for the first 20 files within the Ubuntu machine that have the extension “.py”

In the command above we call the locate command which as we know locates a certain file or in our case with this example it locates any file with a .py extension this is because as we know that the “\*” (asterisks) is a wild card symbol with in Linux. But there is one more thing that is different in this command then others and that is that I have a pair of quotes around the “\*.py” instead of \*.py now these two things are very different and I don’t want you to forget this next part the shell acts erratically when there is no quotes and it doesn’t behave correctly it might miss a file or not get all the files that you are looking for. Meanwhile when we put quotes around the file it behaves in the correct manner.

Hence the command above is locating all the python files with a “.py” extension but only outputting the first 20 (instead of the 500+)

Typically we keep the quotes around the files and we can use our piping technique along with the grep command and filter through to the data for which file we want.

The next flag we will be utilizing is the “-c” flag this flag allows us to display the number of matching entries for a given file/directory. Check out the image below in order to see how the flag works:

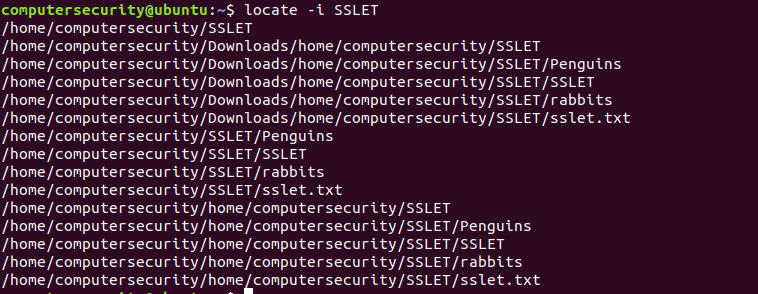


**Figure 6.3:** This figureillustrates the –c flag and counts how many files meet the criteria specified.

In the first line we use the locate –c command in order to count the number of matching entries of “computersecurity”. Once the command is executed we are able to see that there 1737 matches of the given directory.

In the second we started to look for a file similarly using the locate –c command we looked for all files with any extension (any extension because we used the wild card symbol) but started with the word “example”. In order to recap we use the locate –c command to count the number of occurrences that there is some file with the word example in it. For instance there could be example.py, example.html and etc… (In our case there is just example.py)

In the next example we are going to go through the “-I” flag. This flag will show us how to ignore case sensitive characters. For example let’s say we have a file named sslet.txt and SSLET.txt and we need both but can’t find either. We can run the following command which will ignore all case sensitivity and match the characters. In the screen shot below we will illustrate how this works:



**Figure 6.4:** The figure above uses the –l flag and ignores the case sensitivity of the SSLET directory.

The next important thing we are going to learn is how to refresh the mlocate database which is the database that is used to locate the different files within Ubuntu. In order to update the database we need “sudo” privileges for this to work. The command is simply sudo updated (see below how we did this):



**Figure 6.5:** Updating the database of files within our system.

As you can see we just called the sudo updatedb command and this command allows us to update out utility to work efficiently.

This is a high level view of how the locate command works in order to see more you will have to do more research on this command:

**The Watch command**

The watch command another command which allows for the display output in the terminal to be refreshed. This allows a command to be watched and produces functionally a similar output to the top command (when it is ordered). The command as others will always run until it is otherwise told to do so by (Ctrl +C)

The watch command as we will see can be used just like the top command to see what happening with in the Operating System in Real- Time. For instance let’s run the most basic command associated with the watch command it is “watch date”. This command is shown below it will allow for us to see the output of the date and time for our given location. Check out the two images below in order to see how we can make this work:



**Figure 6.6:** The figure above shows the most basic form of the watch command.

The screenshot below is the output of the watch date command on the top left where it says “Every 2.0s: date” this indicates that the output terminal instance will refresh itself every 2 seconds. Then on the top right it gives the current date and time of our terminal instance. Then on the second line it is the actual output of the date command that’s going to be refreshed every 2 seconds.



**Figure 6.7:** The output of the watch date command

Another important flag which can be used with the watch command is the “-d” flag this flag allows us to highlight the part of the output which is changing in between our given time frame. In order to see how this command works check it out below:



**Figure 6.8:** Using the –d flag “watch –d date” command we are able to see in the output instance what factor is changing.

As we can see from the “watch –d date” screenshot command above: the seconds is highlighted because the seconds is always changing. If we look closely we will see that after 30 the 3 remains constant and isn’t highlighted but the 0th digit is this is because the three isn’t changing only when Ubuntu detects a change is when that given change is highlighted. This makes more sense when we look at the output.

Another flag that we can introduce here is the “-n” flag this flag allows for us to create a custom time interval. So instead of 2 seconds maybe I want 10 seconds in order to do this we just pass the “-n” flag through alongside the value of seconds we want to be customized so in my case 10 seconds. In order to do this check out the screenshot’s below:



**Figure 6.9:** Updating the watch command every 10 seconds



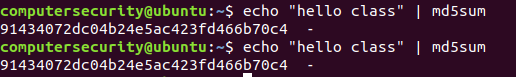
**Figure 6.10:** The output of the following command above “watch –n 10 date”

In the following screenshots we run the watch command and it updates every 10 seconds. The first screenshot is the actual command which we write into the terminal. Meanwhile the second command is the output.

**Basics of Cryptography**

In this section we are going to give a high level view of three cryptographic functions they are md5sum, sha512 and AES (you will go more into this in a later class) hence why this is a high level view.

In order to understand these though we need to understand some math first such as what is a hash function? A **hash function** is just a function that takes in a input value (this value could be a number, files, tar files, iso images, strings, anything like 89 or “hello class”, ubuntu.iso, example.py it could be anything) and creates an output value based on the input value. But for every x input value you will always receive the same output y value. For instance check out the example screenshot below if I write a output stream called “hello class” twice I get the same output(hence for every hash function x we get the same output y):



**Figure 6.11:** An example of how we can use md5sum

As we can see from the screen shot above the two md5sum values are the same.

The hash function md5sum as we can see takes in any input data and creates a 32 character hexadecimal output (in case you forgot hexadecimal is 0-9 and A-F most hash functions are outputted in hexadecimal format). But what really is md5sum. The md5sum is a hash function calculation which produces a 128 bit value to verify a file, value or text.

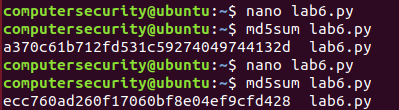
If we go through what is happening in the statement “echo “hello class” |md5sum” then we will see we are outputting the words “hello class” and taking that output(AKA piping) and making it in a cryptographic md5sum. (Note we talked about the echo command a while ago it is just a print statement for bash)

A hash function especially the ones we will see and the one’s you will deal within the real world are irreversible which means we can’t get back to our input if we only know the output. Hence it is like a one way street**. The only possible way for someone to get in is if they try every possible combination and this is actually called a brute-force attack.**

Hash functions are typically used for proving that something is the same as something else without revealing too much information… This statement might sound strange so let me explain it to you everything including files, image files and folders all have a specific md5sum so when it changes the md5sum changes hence we are able to identify according to that value when a directory or file is altered. For instance check out the example below:

For this example I’m going to create a python file named lab6.py and its contents will contain print(‘hello world’).After we display the md5sum for this file we will change it a little and we will see the value change.(check out the screenshot below):

1. In the first line we made a python file named lab6.py and put in it “print(‘Hello world’)”.
2. Then next we run the md5sum command on it and should get the following output “a370c61b712fd531c59274049744132d”.
3. After we do this we can then alter the contents of lab6.py file to “print(‘ Ahoy world’)”.
4. Once we change we will get an md5sum as listed “ecc760ad260f17060bf8e04ef9cfd428”.
5. The screenshot below illustrates the process:

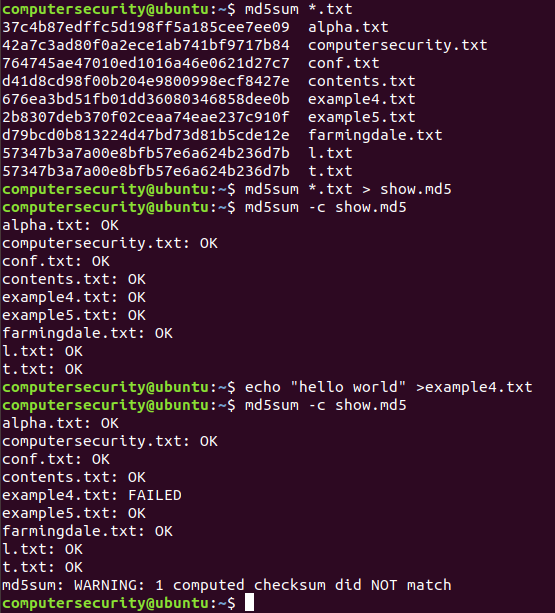


**Figure 6.12:** In the figure above wecreated a python file ran the md5sum on it. Then we altered the file and then ran the md5sum again to show the output is different when we alter it.

If you are still confused about the example above let me explain it to you like this:

*Joseph* and *Alinda* are solving a challenge Physics problem. Joseph gets the answer “56 m/s” and wants to prove to Alinda that he got the answer so Joseph will hash his answer through an md5sum. Once Joseph gives the hash to Alinda. Alinda can’t find out what the answer is from the hash. But when Alinda finds the answer herself she can hash her answer and if she gets the same result, then we know that Joseph did have the answer. Hence why I said the statement above on hashes they are typically used in the context of verifying information without revealing too much to the part who is verifying it.

Another thing we can do with md5sums is that we can create an md5 file and check our files to see if they all match. For instance in our example we will create a file called show.md5 and in it we will put all the necessary contents. After we do this we will check to see if the contents are the same. Check out the screen shot below to see how this works:



**Figure 6.13:** The figure above illustrates all the md5sum values for our text files. Then we dump the values into a file and try verify that our files are still the same and haven’t changed.

In the **first** command we run an md5sum on all text files. This will display all of the md5sum values for all our text files. We understand that the command **md5sum will display md5sum values** then we have the **\*.txt which will display all text files remember that \* is the wildcard symbol.**

In the **second** command “md5sum \*.txt >show.md5” we are outputting all the md5sum values into a file called show.md5

In the **third** command we introduce a new flag called the –c flag this flag will check the files currently and tell you if they have been altered we will either get an “OK” or a “FAILED” so when we run the command: “md5sum –c show.md5” this will cycle through the show.md5 file and compare it the current md5sums of our current files and if they match then it will say OK and if they don’t match it will say FAILED. As we can see from the figure above they all match so we received an output of “OK”

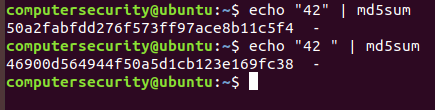
In the **fourth** command I put the contents hello world in a text file example4.txt (this file is already made from a prior lab).

In the **fifth** command I run the third command again and we see that the file example4.txt is different so this makes it fail the test and it throws us a warning to alert us.

In order to summarize the screenshot above we put all the .txt extension files in an md5 sum hash file called show.md5 then we compared to the original state of the files. After we did the comparison I altered one of the text files and re ran the command in order to show you what happens when it fails.

This is the core fundamentals of the md5sum command the next part there will be questions.

**Question6.7:**  Look at the screenshot below why are the md5sum’s different?



**Figure 6.14:** An example of an md5sum problem

**What on earth is SHA?**

Another cryptography algorithm which is commonly used is the “Secure Hash Algorithm”(SHA for short) this is a hashing algorithm which is used to determine the integrity of data, files, operating system images etc…(similar to md5 except the algorithm is different) This algorithm is a big deal when dealing with many things on the internet especially authentication mechanisms like trying to log on to a service, ensuring your data is not compromised, connecting to a “smart” device(Internet of Things) or anything which uses a certificate authority and many more. This algorithm generates a unique hash value (based on the algorithm) this is generated from any file and this newly generated value is able to tell us whether or not a file has been altered or not. This was a high level view of the SHA algorithm.

This will make more sense when we discuss SHA-256. The SHA-256 algorithm is a hash function which has a length of 256 bits or 32 bytes. Each message for a SHA algorithm is processed by something called a **block**. A block is a sequence of bytes or bits (it is nothing crazy).In our case the block size of the SHA-256 algorithm is 512 bit blocks or 64 bytes (we will get to this in a later course how we achieve this value). In addition to this it is important to note that each block that will be processed requires 64 **rounds**. A round is simply a collection of different parameters which goes through the algorithm and alters the given data in order to encrypt it. So In our case we need to go through 64 rounds on each block this will allow our data to be encrypted.

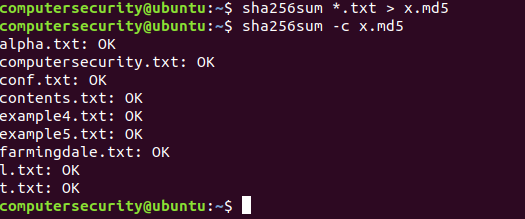
Without getting into the algorithm we can discuss why we need the SHA-256 algorithm. The need for SHA-256 is that it is smaller to implement, requires less bandwidth to store and transmit, and requires less memory to compute. The last reason we use SHA-256 is because unless you make your software very intricate SHA-512 (which is what Ubuntu uses) will not work there are too many systems which are not built to understand SHA-512. Hence due to the following reasons SHA-256 is the reasonable choice.

Let’s go through a couple of examples of how we would achieve the respective outputs. For the first part of this example we are going to deal with the sha256sum command. In the command below we will illustrate in its basic form how this works:



**Figure 6.15:** Executing a sample sha256sum command on a python file.

In the command above similar to how we used the md5sum command we can use the sha256sum command. We can see in the command above we wrote “sha256sum” this allows us to get the sha256 value of a given file (everything is the same as md5sum except the algorithm and the bit size and other stuff which we don’t need to worry about now.) Once we have the command sha256sum we pass it a parameter which is our file or directory or operating system image or anything. This enables us to verify the contents of the file have been unchanged. We are going to do the same thing as we did above for the md5sum command for the sha256 command. Check out the screen shot below:



**Figure 6.16:** In this figure we are putting all the sha256 values in a .md5 file extension and it will verify our values. (I know what you’re thinking why are we using .md5 when it’s a sha file. Truth is they are different algorithms but they do the same thing.)

In the screenshot above we took all the text files and turned them into sha256 sum values. Once that was done we dumped the values and contents in the x.md5 file (which we just made). Then after we do this we run the same command as we did with the md5sum and we are able to evaluate the file integrity of the different files.

**The openssl command**

Before we finish talking about the basics of cryptography. I want to touch upon one thing as you have seen we didn’t touch upon too much on the math behind cryptography but I promise you it exists and it requires a knowledge of Linear Algebra more specifically matrices. Since Linear Algebra isn’t a pre-req for this course there is no reason why we need to understand the math behind it in so much detail.

The next thing I want to cover is **openssl** and what that is. At a high level view openssl is a commercial grade open source software library which is built in the C programming language and is used for securing applications and communication channels in which computer networks communicate with. It is widely used in most every webserver and website that is out there. The library implements the main cryptographic operations such as symmetric encryption, public key, digital signature, x.509 certificates, hash functions and etc… OpenSSL also implements the SSL which is the Secure Sockets Layer protocol (this is standard security protocol for establishing encrypted links between a web server and a browser)

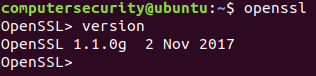
The first thing that we are going to do with this command is check the version which we are using and the list of commands which can be used.

In order to run our openssl commands we are going to first type openssl into the terminal as so:



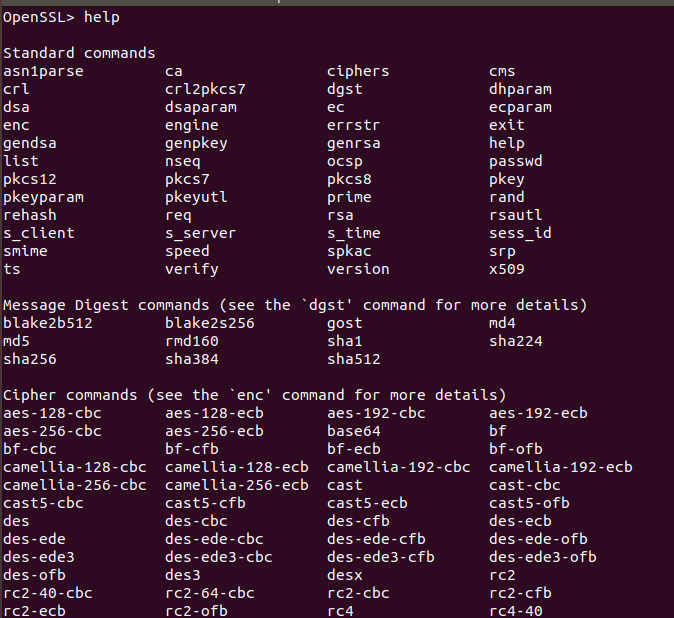
**Figure 6.17**:The figure above illustrates how to call the openssl command prompt.

Once we see that arrow we know that we are in the right command and the openssl command was successful in executing. After we do this we can type in “version” into the terminal this will display our current version of openssl:



**Figure 6.18:** The figure above gets the version of openssl which we are using.

After we do this we can list out all the commands which can be used with the encryption algorithms in order to do this we can just type in “help”. The help command is shown below:



**Figure 6.19:** The output of the help command through the openssl command prompt.

There are so many commands in this output that if I went through all of them we would be here for a long time it is good practice to familiarize yourself with them. But for this lab I’m going to list out a couple of the commands and what they do:

|  |  |
| --- | --- |
| ca | This commands enables us to create a certificate of authority |
| dgst | Computes hash functions |
| enc | Encrypt and decrypt using a secret key algorithm |
| X509 | Data managing for x509 certificate |
| verify | Checks for X509 certificates |
| rsa | Rsa data management |
| passwd | Generation of “hashed passwords” |
| aes-128-cbc | Encryption cipher command to encrypt in aes format it uses the CBC method |
| genrsa | Commands enables us to generate a pair of public/private keys for the RSA algorithm |
| rand | Generates a pseudo random bit string |

In reality these are all important commands I just touched upon a few of them which I have used. In the next part of this lab we will explore some more operations with the openssl command then following the command there will be questions.

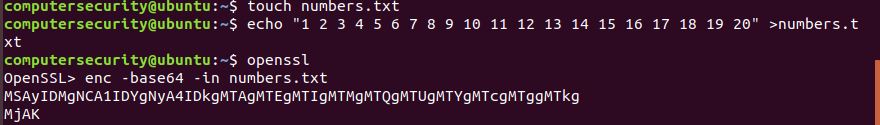
In order to do the next part we are going to exit out of the openssl part for a second (using the ctrl +D keys simultaneously).

In this example we are going to **create a file named numbers.txt**. After we will put the numbers 1-20 in it then we will encode the file and view its contents. When we need to encode (**NOT ENCRYPT… ENCODE THERES A DIFFERENCE**) numbers openssl encourages us to use base64 encoding.

(As I just addressed encoding and encryption are different. Encoding transforms data into another format use a certain scheme. Encodings main use is for maintain data usability. Meanwhile encryption transforms data into a format where only specific individuals can view the actual contents within the file. Encryption is used for maintaining data confidentiality and only can be viewed by people who have the same keys)

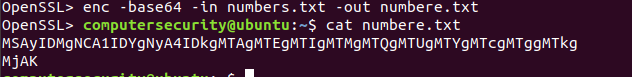
In the image below we are able to see the following:

1. On the **first** line we create a **text file** named **numbers.txt**
2. On the **second** line we out the numbers 1-20 into the file numbers.txt
3. On the **third** line we enter the command openssl which opens our openssl command prompt.
4. On the **fourth** line in our terminal within the openssl command prompt we enter in the command “enc –base64 –in numbers.txt” what this command is saying is to encode using the base64 methodology the file numbers.txt. (Read the statement as if you we were reading English and it makes sense)
5. On the **fifth** line we get the encoded value for the numbers.txt file



**Figure 6.20:** In the figure above, we are encrypting a file with the algorithm base64.

I have also additionally shown this screenshot below this command enables us to have our output outputted into a file (which in our case is numbere.txt). The only difference between the two commands is the –out flag which outputs the contents to a file specified. Then after we do this we use the cat command ad view the contents outside the openssl prompt.



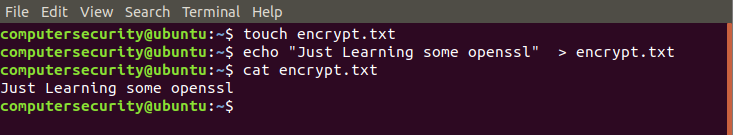
**Figure 6.21:** In the figure above, we are encrypting the output into a text file.

This is a very good and useful encoding algorithm but not practical for encryption due to the fact there is no secret key as we will see shortly. That is an imperative part to cryptography.

As you will see a lot of these encryption statements can be understood as if you were reading English. Let’s do an example with text now.

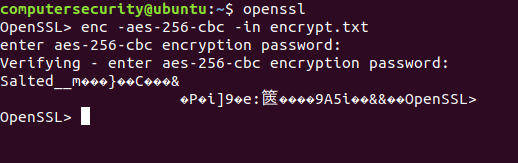
At a high level view In the next example we are going to **create a file named encrypt.txt as its contents we will put “Just Learning some openssl”** after we create the contents we will **then encrypt the text file with an AES 256 bit Cipher**. After we do this we will then decrypt the file and view our contents again.

Our **first** step in this process is to **create the file** encrypt.txt feel free to create it however you like. After we create the file **encrypt.txt** we then put the contents: **“Just learning some openssl”** (no quotes) inside the file. After we do that I verify that the contents are within the file using the cat command. Check out the screenshot below in order to see what I did:



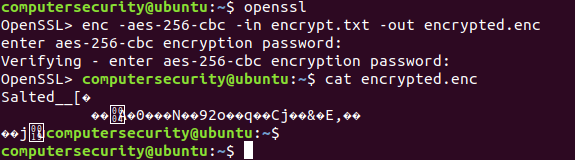
**Figure 6.22:** Creating a text file called encrypt.txt. This file will just have text in it.

Our **second** step is to encrypt the data through the openssl command. **First** we call the openssl command prompt, **second** we enter in the command “enc –aes-256-cbc –in encrypt.txt” which will encrypt the contents of our file. The **third** step is the AES-256-CBC will prompt you for an encryption password (secret key). **IN REAL LIFE YOU** **REALLY NEED TO MAKE IT SOMETHING MEANINGFUL AND SECURE.** For my example though I just made the password “hello” (no quotes). After we do this we will get an encrypted output. My Screenshot below illustrates how we can do this:



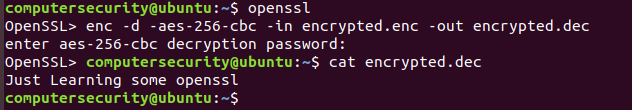
**Figure 6.23:** Encrypted text of encrypt.txt

Our **Alternate second step** is to output the encrypted contents into a file. In order to do this we just need to alter the initial openssl command to indicate that we are outputting the contents into a file. The command is as follows “enc –aes-256-cbc –in encrypt.txt –out encrypted.enc”(note we don’t need to use .enc it could also be .txt). After we do this it will then prompt us for a password again. Again mine is going to be “hello” but feel free to make your password whatever you want. After we do that we exit out of the openssl command (using ctrl+D) then run the cat command on our newly encrypted file:



**Figure 6.24:** Encrypting the file encrypt.txt then outputting the encrypted contents into a file called “encrypted.enc”

Our **third** **and final** step is going to be decrypting the file. In order to do this we are going to run the openssl prompt again. Once in there we are going to run the following command: “enc –d –aes-256-cbc –in encrypted.enc –out encrypted.dec” running the following command will allow our file to be decrypted. This is made possible solely by two things the “-d” flag which decrypts the data and the password which you encrypted the data with. In my case it was “hello” (no quotes) so once we enter “hello” into the password field our file is decrypted. We are able to verify this with exiting out of the openssl command prompt and typing in cat encrypted.dec



**Figure 6.25:** Decrypting our encrypted file and outputting the contents into another file.

**Public and Private Keys**

The last thing I want to talk about before the questions on openssl is the topic of public and private keys and what they are? And how we can generate them.

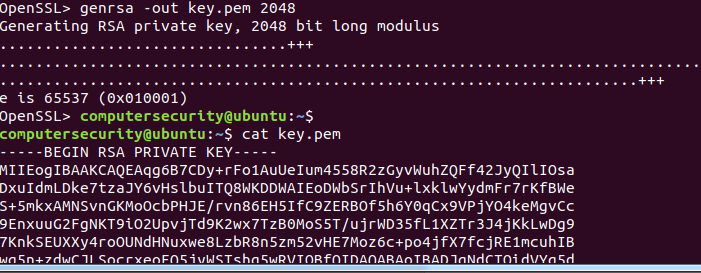
**Public and Private Keys** are comprised uniquely of long random hexadecimal characters (0-9 and A-F). **The public key** as the name suggests is made available to everyone via a public repository or file. This means as long as I have access to that file or repository anyone can view the encrypted file by decrypting it with the secret password. On the **contrary a private key** must remain confidential to its respective owner. Check out the examples below for when we would use a private key and public key:

For example if Joe wants to send sensitive data to Alinda and wants to be sure that only Alinda can read the encrypted data then we need to encrypt the data with Alinda’s key. Since only Alinda has access to her key as a result she is the only person who can decrypt the message.

Another example of this is when you SSH into a device or computer you have an encrypted connection between you and the device no one else can go on your connection because you have the public key.

Enough about this theoretical information about keys. Let’s generate some keys. For our first example we are going to generate a key of a certain length typically RSA enforces users to use 2048 bit keys or 3072 bit keys for extra security. In our example we are going to use a key length of 2048 bits which is 256 bytes (Remember to get from bits to bytes you divide by 8 and to vice versa you multiply by 8).

In our example below we can use the **genrsa** command in order to make a key file which has our 256 byte key length after we do this we can then output the key contents into a file let’s name it key.pem (.pem is a security file extension for keys which stands for privacy enhanced email). After we store the RSA key in there we can then exit out of the openssl prompt in our terminal and use the cat command on the RSA key to view it.



**Figure 6.26:** Generating a 2048-bit long RSA key.

The RSA Key which you copied above is you’re key if we were to connect to an Amazon Web Services EC2 instance or any other cloud instance through a PuTTY terminal or anything like that we would need a generated RSA Key provided by said company.

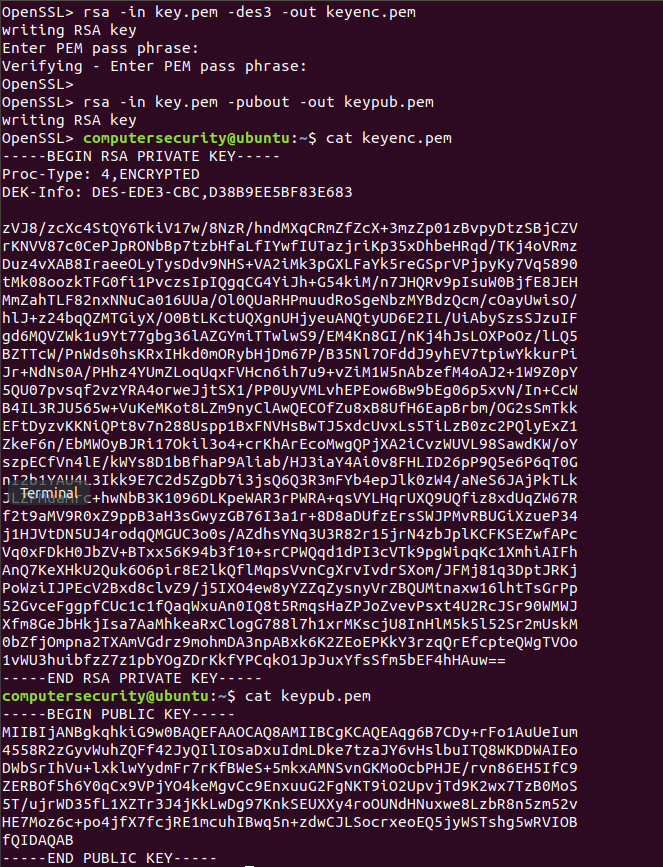
The next thing I’m going to show you is how we can see a more detailed breakdown of the key file. This is useful when looking to understand how the algorithm was designed. The screenshot below illustrates a small snippet to the output. The code below is saying using the rsa algorithm within the key.pem file display the text but not the actual long key just the information within it.



**Figure 6.27:** The inner workings of our RSA algorithm are shown in a snippet above.

The last thing which is important (besides seeing how your key is made as shown above) that we need to discuss really quick is when we have our private key how do we get a public key? And how do we encrypt our private key making it even harder for people to figure out what our private key is even so because now they would need our password to decrypt it. In the screen shot below we illustrate this principle.

In the example below we take our key.pem file which has our private RSA key and we turn it into an encrypted private key using the triple DES procedure. After we do that we then take our key.pem file which is still our private RSA key not encrypted then we run it through the –pubout command which will output our public key which we can share to other people if you choose to do so.



**Figure 6.28:**  In the figure above, we encrypted our private key. Then after we did that we made a public key.

This is going to be the end of this cryptography section for now I know we have covered a lot so if you have any questions just ask.