

Introduction

In this lab, you'll learn how to generate RSA private and public key pairs using the OpenSSL utility.

OpenSSL is a commercial-grade utility toolkit for Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. It's also a general-purpose cryptography library. OpenSSL is licensed under an Apache-style license, which means that you're free to get it and use it for commercial and non-commercial purposes (subject to some simple license conditions).

You'll access OpenSSL through SSH onto the Linux VM instance associated with your lab.

What you'll do

- **OpenSSL:** You'll explore what generating key pairs looks like using OpenSSL by using SSH to access the Linux instance.
- **Encrypt and decrypt:** You'll use the key pair to encrypt and decrypt some small amount of data.
- **Verify:** You'll use the key pair to sign and verify data to ensure its accuracy.

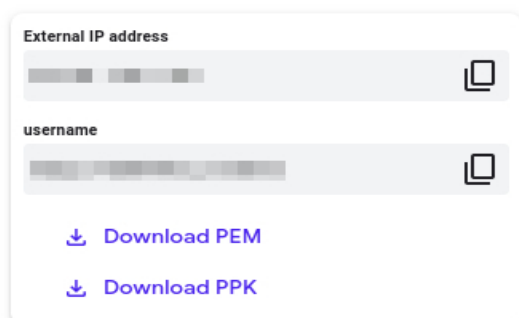
Start the lab

You'll need to start the lab before you can access the materials in the virtual machine OS. To do this, click the green "Start Lab" button at the top of the screen.

Note: For this lab you are going to access the **Linux VM** through your **local SSH Client**, and not use the **Google Console (Open GCP Console)** button is not available for this lab).

Start Lab

After you click the "Start Lab" button, you will see all the SSH connection details on the left-hand side of your screen. You should have a screen that looks like this:



External IP address

username

[Download PEM](#)

[Download PPK](#)

Accessing the virtual machine

Please find one of the three relevant options below based on your device's operating system.

Note: Working with Qwiklabs may be similar to the work you'd perform as an **IT Support Specialist**; you'll be interfacing with a cutting-edge technology that requires multiple steps to access, and perhaps healthy doses of patience and persistence(!). You'll also be using **SSH** to enter the labs -- a critical skill in IT Support that you'll be able to practice through the labs.

Option 1: Windows Users: Connecting to your VM

In this section, you will use the PuTTY Secure Shell (SSH) client and your VM's External IP address to connect.

Download your PPK key file

You can download the VM's private key file in the PuTTY-compatible **PPK** format from the Qwiklabs Start Lab page. Click on **Download PPK**.

 [Download PEM](#)

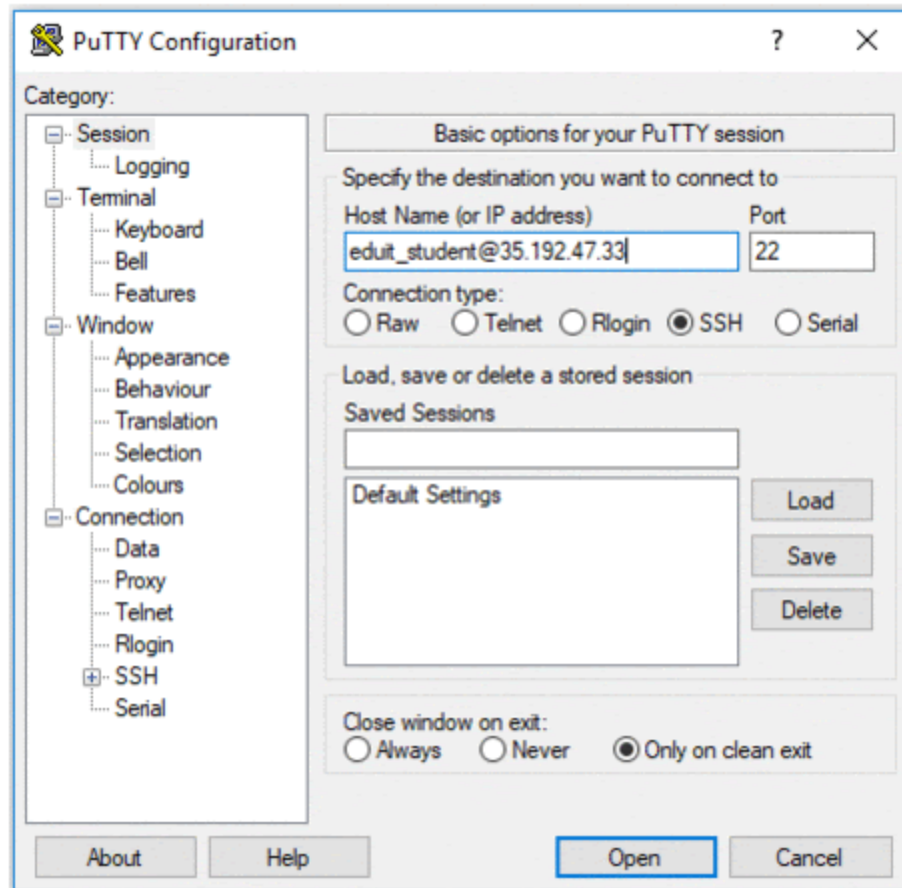
 [Download PPK](#)



Connect to your VM using SSH and PuTTY

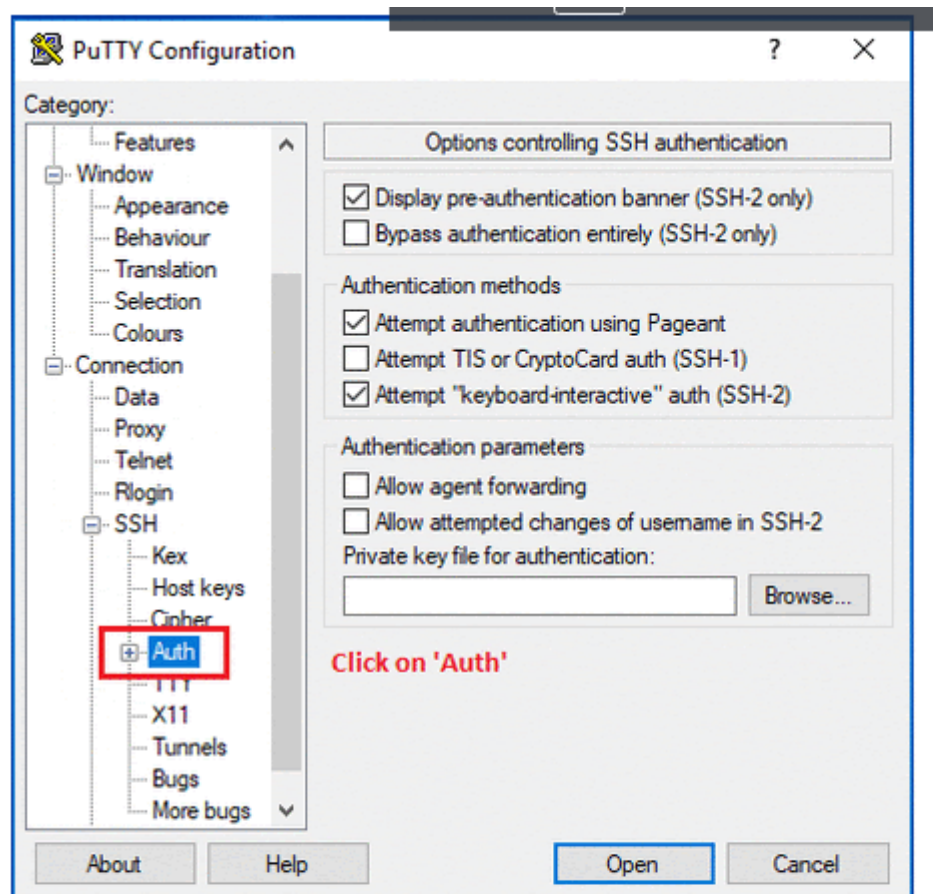
1. You can download Putty from [here](#)
2. In the **Host Name (or IP address)** box, enter `username@external_ip_address`.

Note: Replace **username** and **external_ip_address** with values provided in the lab.



3. In the **Category** list, expand **SSH**.
4. Click **Auth** (don't expand it).
5. In the **Private key file for authentication** box, browse to the PPK file that you downloaded and double-click it.
6. Click on the **Open** button.

Note: PPK file is to be imported into PuTTY tool using the Browse option available in it. It should not be opened directly but only to be used in PuTTY.



7. Click **Yes** when prompted to allow a first connection to this remote SSH server. Because you are using a key pair for authentication, you will not be prompted for a password.

Common issues

If PuTTY fails to connect to your Linux VM, verify that:

- You entered `<username>@<external ip address>` in PuTTY.
- You downloaded the fresh new PPK file for this lab from Qwiklabs.
- You are using the downloaded PPK file in PuTTY.

Option 2: OSX and Linux users: Connecting to your VM via SSH

Download your VM's private key file.

You can download the private key file in PEM format from the Qwiklabs Start Lab page. Click on **Download PEM**.

Download PEMDownload PPK

Connect to the VM using the local Terminal application

A **terminal** is a program which provides a **text-based interface for typing commands**. Here you will use your terminal as an SSH client to connect with lab provided Linux VM.

1. Open the Terminal application.

- To open the terminal in Linux use the shortcut key **Ctrl+Alt+t**.
- To open terminal in **Mac** (OSX) enter **cmd + space** and search for **terminal**.

2. Enter the following commands.

Note: Substitute the **path/filename for the PEM** file you downloaded, **username** and **External IP Address**.

You will most likely find the PEM file in **Downloads**. If you have not changed the download settings of your system, then the path of the PEM key will be **~/Downloads/qwikLABS-XXXXX.pem**

```
chmod 600 ~/Downloads/qwikLABS-XXXXX.pem
```

```
ssh -i ~/Downloads/qwikLABS-XXXXX.pem username@External Ip Address
```

```
gcpstagineduit1370_student@35.239.106.192:~$ ssh -i ~/Downloads/qwikLABS-L923-42090.pem gcpstagineduit1370_student@35.239.106.192
The authenticity of host '35.239.106.192 (35.239.106.192)' can't be established.
ECDSA key fingerprint is SHA256:vrz8B4aYUtruFh0A6wZn6Ozy1oqqPEfh931oIvxiTm8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '35.239.106.192' (ECDSA) to the list of known hosts.
Linux linux-instance 4.9.0-9-and64 #1 SMP Debian 4.9.168-1+deb9u2 (2019-05-13) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
gcpstagineduit1370_student@linux-instance:~$
```

Option 3: Chrome OS users: Connecting to your VM via SSH

Note: Make sure you are not in **Incognito/Private mode** while launching the application.

Download your VM's private key file.

You can download the private key file in PEM format from the Qwiklabs Start Lab page. Click on **Download PEM**.



Connect to your VM

1. Add Secure Shell from [here](#) to your Chrome browser.
2. Open the Secure Shell app and click on **[New Connection]**.



3. In the **username** section, enter the username given in the Connection Details Panel of the lab. And for the **hostname** section, enter the external IP of your VM instance that is mentioned in the Connection Details Panel of the lab.



4. In the **Identity** section, import the downloaded PEM key by clicking on the **Import...** button beside the field. Choose your PEM key and click on the **OPEN** button.

Note: If the key is still not available after importing it, refresh the application, and select it from the **Identity** drop-down menu.

5. Once your key is uploaded, click on the **[ENTER] Connect** button below.



6. For any prompts, type **yes** to continue.
7. You have now successfully connected to your Linux VM.

You're now ready to continue with the lab!

Generating keys

Before you can encrypt or decrypt anything, you need a private and a public key, so let's generate those first!

Generating a private key

Remember, a key pair consists of a public key that you can make publicly available, and a private key that you need to keep secret. Shhhh. :) When someone wants to send you data and make sure that no one else can view it, they can encrypt it with your public key. Data that's encrypted with your public key can only be decrypted with your private key, to ensure that only you can view the original data. This is why it's important to keep private keys a secret! If someone else had a copy of your private key, they'd be able to decrypt data that's meant for you. Not good!

First, let's generate a 2048-bit RSA private key, and take a look at it. To generate the key, enter this command into the terminal:

```
openssl genrsa -out private_key.pem 2048
```

You should see the following output (or something very similar) :

```
gcpstaging3645_student@linux-instance:~$ openssl genrsa -out private_key.pem 2048
Generating RSA private key, 2048 bit long modulus
.....+++
.....+++
e is 65537 (0x10001)
gcpstaging3645_student@linux-instance:~$
```

This command creates a 2048-bit RSA key, called "private_key.pem". The name of the key is specified after the "-out" flag, and typically ends in ".pem". The number of bits is specified with the last argument. To view your new private key, use "cat" to print it to the screen, just like any other file:

```
cat private_key.pem
```


The contents of the private key file should look like a large jumble of random characters. This is actually correct, so don't worry about being able to read it:

```
gcpstaging3645_student@linux-instance:~$ cat private_key.pem
-----BEGIN RSA PRIVATE KEY-----
MIIEpQIBAAKCAQEAuwuFTdlse/9pi9mW2RklrLHKFx8ejgyJeizxjyX94Rlmz7Q2
8hC5rpKBbRMtv7FooqVk+lAem50N/LaZtCa2tlx6DUImrCcvkw7jVvp0RQlkrd6X
L2mNMq9WeQTLgJ2wezCpcO3ZXmoAtHOPYfobztK7HH5j7iRBUJ9Gt1r87sYpRkW
b9ijB+Fs2gaa/js3w5pAiHpV5/YacPQmAcHcTmgIeupa+TAI3CEKwS3fsNsVFKT1
bcxynr6lqen4N0lzFSuyIOf0kkEJaYcaO57WRSkt7UnUbi6Dz0g+lgVxly/Ys2Et
t6jnsoATLa58bik0yqqTxj9FjibwcWBvjaJv8QIDAQABaoIBAQCkkFJMWAr2XZVD
EcGCaHmZIyMmjJ6ZY7Zg48Hzq/1eWrb17xyYhNaZHRT6HnGVwkTJNLNO98a2Ksvv
l8hLk/QVJe/MUnMr654TyHmEQP6i7szBnKFveXJdMZ7nAqz5D8QB3qJlVQ513tWX
7kQsc5ybrxQo0ZqiHwvIHxpcn5UBjZ3X33otS+GFmuMD4gmFMxAu8hNGyewxsc1k
dvtKTLMO1XGXtdkCWILSRk+V7RW3s5JN8Vd4izfxpuGRbSlX6Ph11VRM88H8fErI
vCvQOe9NMC0twacfGnMIA3raB8sSYk2joariSpbf4e8aGl/vlqKlq5iGBZuD3uNY
g0OxFN0xAoGBAOM6drIdNoZvGmzfKr6fzWUhl9kiyFs1SdHGTGH1iIpN2DfOK+2f
akLgYUTPYO3TH82Uwita+njDhVIAaciF+yVLDtMXfydZzPn00Xefy+E03SAlQD0
fH7T+l2TiRRB7QsrGPBn+WKDtBvfK4/iLrwB5gHSPQqBXExm6wZ5LXvbAoGBANK6
hpW5xR7YemI2QkpERfLGF/5mitekkDXik+BiYf1l13VnnAY0NHY2W163TGwnLZ9U
8X1ktDWViq9+dTduTr8HKt5UmKAR39Ay36gxliXXMYCs1x2HOJJw7jb5xaJrG4xP
ywH1vwBMAdWd/tJKNFJVhJdqdCHBERyFwYcJ0uUjAoGBAJLJUaP9+xsc6pUqFRLP
aH/MPQOn1IYXBNjdALrHQk0VvHdQWFBiivxotm0dCiCYIhNLonQOpb8djyJJ4gW
gRW2RfXof0pd8yB4waKA+OhZhbc0GStiz7i8PlCNWtAq0BrttCWWiOpsM/QevOX
yVjqYQsMJSKELnusgGbOIPPBAoGBAM07X1DpOXAaMSrSLsagXD/zvEufYW1FPHHg
n+w2tNcTiugRMboNeYEnohzK8uAuT9ED7D7FcCGpYiT5t9Shz+DdvnK+bvmvZvcS
me23RKrnQ+K6KCEUPXbAEHPd8jWXeQIOB4ktgpi7AKUzWMJpDEpBhqPBzFah6Ehj
vBuTA2GzAoGAdZ9UjKj5JC5SSEClMF08DCY+GRpyFTbv3oVVt1vOgGEvvHRIHm8w
zo+N/gacRuZSa5BZHRAgajolyYg/P014wHclH26ref7FpWFMvOHQasS9dt1uLTfV
9nNznJxiRDeJSQHnjC0zdYs90P0cADNrbsX36CIvoXGtulnbEw2FWXA=
-----END RSA PRIVATE KEY-----
gcpstaging3645_student@linux-instance:~$
```

Head's up: Your private key will look similar to this, but it won't be the same. This is super important, because if openssl was generating the same keys over and over, we'd be in serious trouble!

Generating a public key

Now, let's generate the public key from the private key, and inspect that one, too. Now that you have a private key, you need to generate a public key that goes along with it. You can give that to anyone who wants to send you encrypted data. When data is hashed using your public key, nobody will be able to decrypt it unless they have your private key. To create a public key based on a private key, enter the command below. You should see the following output:

```
openssl rsa -in private_key.pem -outform PEM -pubout -out public_key.pem
```

```
gcpstaging3645_student@linux-instance:~$ openssl rsa -in private_key.pem -outform PEM -pubout -out public_key.pem
writing RSA key
gcpstaging3645_student@linux-instance:~$
```

You can view the public key in the same way that you viewed the private key. It should look like a bunch of random characters, like the private key, but different and slightly shorter:

```
cat public_key.pem
```

```
gcpstaging3645_student@linux-instance:~$ cat public_key.pem
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAuwuFTdlse/9pi9mW2Rkl
rLHKFx8ejgyJeizxjyX94Rlmz7Q28hC5rpKBbRMtv7FooqVk+lAem50N/LaZtCa2
tlx6DUImrCcvkw7jVvp0RQlkrd6XL2mNMq9WeQTLgJ2wezCpcO3ZXmoAtHOPYfob
ztK7HH5j7iRBUJ9Gt1r87sYpRkwWb9ijB+Fs2gaa/js3w5pAiHpV5/YacPQmAcHc
TmgIeupa+TAI3CEKwS3fsNsVFkT1bcxynr6lqen4N0lzFSuyIOf0kkEJaYcaO57W
RSkt7UnUbi6Dz0g+lgVxly/Ys2Ett6jnsoATLa58bik0yqqTxj9FjibwcWBvjaJV
8QIDAQAB
-----END PUBLIC KEY-----
gcpstaging3645_student@linux-instance:~$
```

Head's up: Like your private key, your public key will look different than the one in this image.

Now that both of your keys have been created, and you can start using them to encrypt and decrypt data. Let's dive in!

Encrypting and decrypting

You'll simulate someone encrypting a file using your public key and sending it to you, which allows you (and only you!) to decrypt it using your private key. Similarly, you can encrypt files using other people's public keys, knowing that only they will be able to decrypt them.

You'll create a text file that contains some information you want to protect by encrypting it. Then, you'll encrypt and inspect it. To create the file, enter the command below. It will create a new text file called "secret.txt" which just contains the text, "This is a secret message, for authorized parties only". Feel free to change this message to anything you'd like.

```
echo 'This is a secret message, for authorized parties only' > secret.txt
```

Then, to encrypt the file using your public key, enter this command:

```
openssl rsautl -encrypt -pubin -inkey public_key.pem -in secret.txt -out
secret.enc
```

This creates the file "secret.enc", which is an encrypted version of "secret.txt". Notice that if you try to view the contents of the encrypted file, the output is garbled. This is totally normal for encrypted messages because they're not meant to have their contents displayed visually.

Here's an example of what displaying the encrypted file may look like:

[illegible]

The encrypted file will now be ready to send to whoever holds the matching private key. Since that's you, you can decrypt it and get the original contents back. Remember that we must use the private key to decrypt the message, since it was encrypted using the public key. Go ahead and decrypt the file, using this command:

```
openssl rsautl -decrypt -inkey private_key.pem -in secret.enc
```

This will print the contents of the decrypted file to the screen, which should match the contents of "secret.txt":

```
gcpstaging3645_student@linux-instance:~$ openssl rsautl -decrypt -inkey private_key.pem -in secret.enc
This is a secret message, for authorized parties only
gcpstaging3645_student@linux-instance:~$
```

Creating a hash digest

Now, you'll create a hash digest of the message, then create a digital signature of this digest. Once that's done, you'll verify the signature of the digest. This allows you to ensure that your message wasn't modified or forged. If the message was modified, the hash would be different from the signed one, and the verification would fail.

To create a hash digest of the message, enter this command:

```
openssl dgst -sha256 -sign private_key.pem -out secret.txt.sha256 secret.txt
```

This creates a file called "secret.txt.sha256" using your private key, which contains the hash digest of your secret text file.

With this file, anyone can use your public key and the hash digest to verify that the file hasn't been modified since you created and hashed it. To perform this verification, enter this command:

```
openssl dgst -sha256 -verify public_key.pem -signature secret.txt.sha256 secret.txt
```

This should show the following output, indicating that the verification was successful and the file hasn't been modified by a malicious third party:

```
gcpstaging3645_student@linux-instance:~$ openssl dgst -sha256 -verify public_key.pem -signature secret.txt.sha256 secret.txt
Verified OK
gcpstaging3645_student@linux-instance:~$
```

If any other output was shown, it would indicate that the contents of the file had been changed, and it's likely no longer safe.

MD5

Press **F11** to exit full screen

Let's kick things off by creating a text file containing some data. Feel free to substitute your own text data, if you want. This command creates a text file called "file.txt" with a single line of basic text in it:

```
echo 'This is some text in a file, just so we have some data' > file.txt
```

You'll now generate the MD5 sum for the file and store it. To generate the sum for your new file, enter this md5sum command:

```
md5sum file.txt > file.txt.md5
```

This creates the MD5 hash, and saves it to a new file. You can take a look at the hash by printing its contents to the screen, using this command:

```
cat file.txt.md5
```

This should print the hash to the terminal, which should look something like this:

```
gcpstaging3643_student@linux-instance:~$ cat file.txt.md5
c7a8ef893898f9a6b380eb4ec1e87113  file.txt
gcpstaging3643_student@linux-instance:~$
```

More importantly, you can also verify that the hash is correct, and that the original file hasn't been tampered with since the sum was made. To do this, enter this command and see the following output, which indicates that the hash is valid:

```
md5sum -c file.txt.md5
```



```
gcpstaging3643_student@linux-instance:~$ md5sum -c file.txt.md5
file.txt: OK
gcpstaging3643_student@linux-instance:~$
```

Verifying an invalid file

Press F11 to exit full screen

Next, we'll demonstrate the security of this process by showing how even a single-character change to the file results in a different hash. First, you'll create a copy of the text file, and insert a single space at the end of the file. Feel free to use any text-editor that you'd like. Head's up that we've included instructions for making this change in Nano. To make a copy of the file, enter this command:

```
cp file.txt badfile.txt
```

Then generate a new md5sum for the new file:

```
md5sum badfile.txt > badfile.txt.md5
```

Note that the resulting hash is **identical** to the hash for our original file.txt despite the filenames being different. This is because hashing only looks at the data, not the metadata of the file.

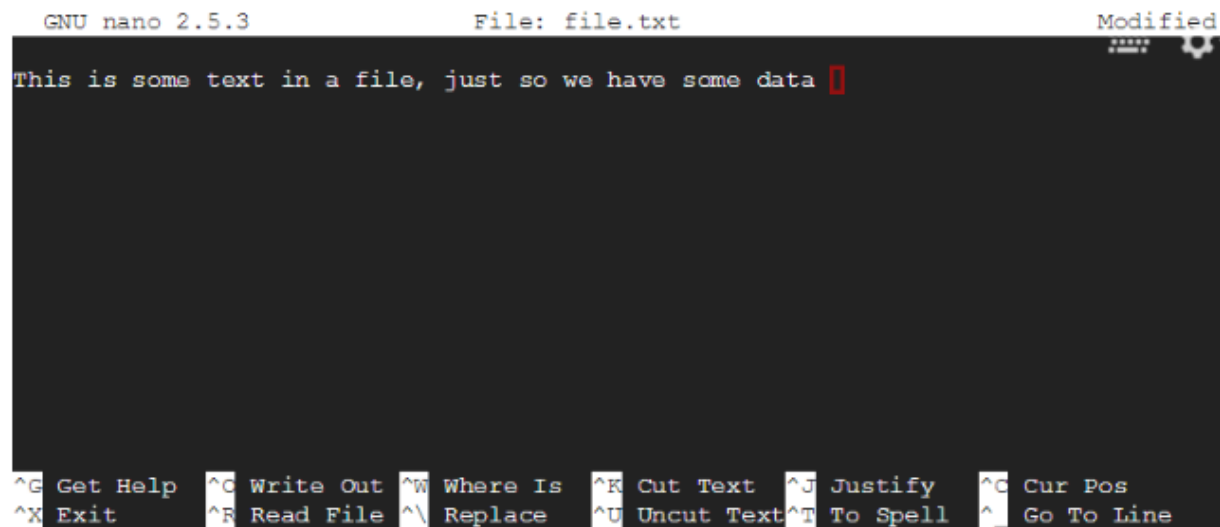
```
cat badfile.txt.md5
```

```
cat file.txt.md5
```

To open the text file in Nano, enter this command:

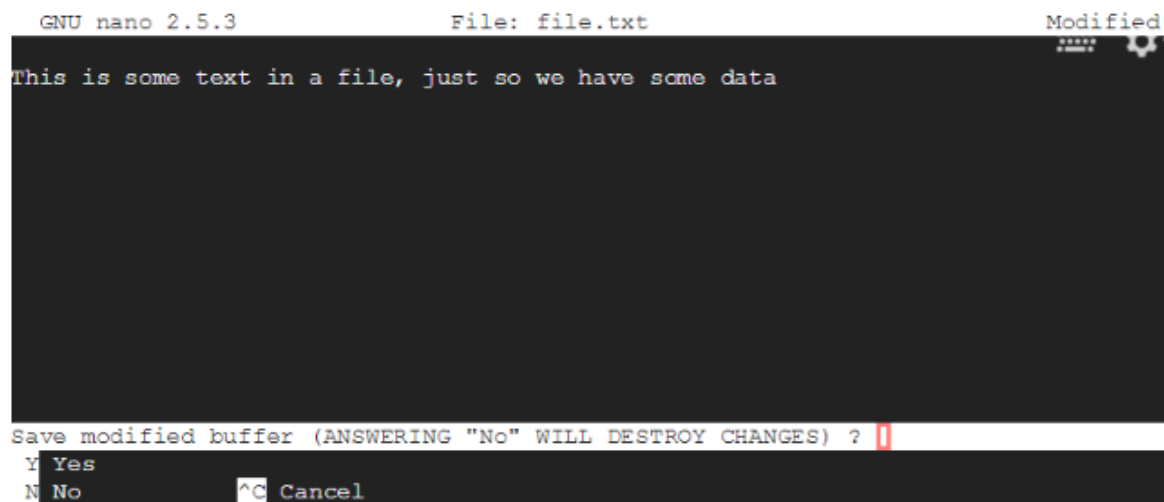
```
nano badfile.txt
```

This will open the file in the text editor. To add a space to the end of the file, use the arrow keys (not the mouse!) to move the cursor to the end of the line of text. Then, press the spacebar to add a space character to the end of the file. Your screen should look like this image:



```
GNU nano 2.5.3 File: file.txt Modified
This is some text in a file, just so we have some data |
^G Get Help ^C Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line
```

To save the file, press **ctrl+X**. You should see this message:



```
GNU nano 2.5.3 File: file.txt Modified
This is some text in a file, just so we have some data
Save modified buffer (ANSWERING "No" WILL DESTROY CHANGES) ? |
Y Yes
N No ^C Cancel
```

Confirm by typing **Y** for **yes**, then press **Enter** to confirm.

This will take you back to the normal terminal screen. Now that you've made a very minor change to the file, try verifying the hash again. It should fail verification this time, showing that any change at all will result in a different hash. Try to verify it by entering this command again:




```
md5sum -c badfile.txt.md5
```


You should see a message that shows that the verification wasn't successful:

```
gcpstagingedit259_student@linux-instance:~$ md5sum -c badfile.txt.md5
badfile.txt: FAILED
md5sum: WARNING: 1 computed checksum did NOT match
gcpstagingedit259_student@linux-instance:~$
```

Click *Check my progress* to verify the objective.



md5sum failure

Check my progress

To see how different the hash of the edited file is, generate a new hash and inspect it:

```
md5sum badfile.txt > new.badfile.txt.md5
```

content_c

```
cat new.badfile.txt.md5
```

Check out how it's different from our previously generated hash:

```
gcpstagingedit259_student@linux-instance:~$ md5sum badfile.txt > new
gcpstagingedit259_student@linux-instance:~$ cat new.badfile.txt.md5
3d121c97162462ce6e3448636ba13359  badfile.txt
gcpstagingedit259_student@linux-instance:~$
```

For reference, here are the contents of the original sum:

```
gcpstaging3643_student@linux-instance:~$ cat file.txt.md5
c7a8ef893898f9a6b380eb4ec1e87113  file.txt
gcpstaging3643_student@linux-instance:~$
```

SHA

Let's do the same steps, but for SHA1 and SHA256 hashes using the shasum tool. Functionally, the two work in very similar ways, and their purpose is the same. But SHA1 and SHA256 offer stronger security than MD5, and SHA256 is more secure than SHA1. This means that it's easier for a malicious third party to attack a system using MD5 than one using SHA1. And because SHA256 is the strongest of the three, it's currently widely used.

SHA1

To create the SHA1 sum and save it to a file, use this command:

```
shasum file.txt > file.txt.sha1
```

View it by printing it to the screen, like you've done before:

```
cat file.txt.sha1
```

```
gcpstaging3643_student@linux-instance:~$ cat file.txt.sha1
f06cc8cbdbbf9f89c9f1addab16aba57555b0cb  file.txt
gcpstaging3643_student@linux-instance:~$
```

Now, verify the hash using the command below. (Like before, this would fail if the original file had been changed.)

```
shasum -c file.txt.sha1
```

You should see the following output, indicating that the verification was a success:

```
gcpstaging3643_student@linux-instance:~$ shasum -c file.txt.sha1
file.txt: OK
gcpstaging3643_student@linux-instance:~$
```

SHA256

The same tool can be used to create a SHA256 sum. The "-a" flag specifies the algorithm to use, and defaults to SHA1 if nothing is specified. To generate the SHA256 sum of the file, use this command:

```
shasum -a 256 file.txt > file.txt.sha256
```

You can output the contents of this file, the same as before:

```
cat file.txt.sha256
```

SHA256's increased security comes from it creating a longer hash that's harder to guess. You can see that the contents of the file here are much longer than the SHA1 file:

```
gcpstaging3643_student@linux-instance:~$ cat file.txt.sha256
911166859be8d9fa946ed2badd2bfc7a504466384cee3e44b0a4f63db20030eb  file.txt
gcpstaging3643_student@linux-instance:~$
```

Finally, to verify the SHA256 sum, you can use the same command as before:

```
shasum -c file.txt.sha256
```

Conclusion

Congrats! You've successfully created and verified hashes using three of the most common hashing algorithms: MD5, SHA1, and SHA256. You've also demonstrated the security of these hashes by showing that even very small changes to the file result in a new hash and cause the verification to fail.