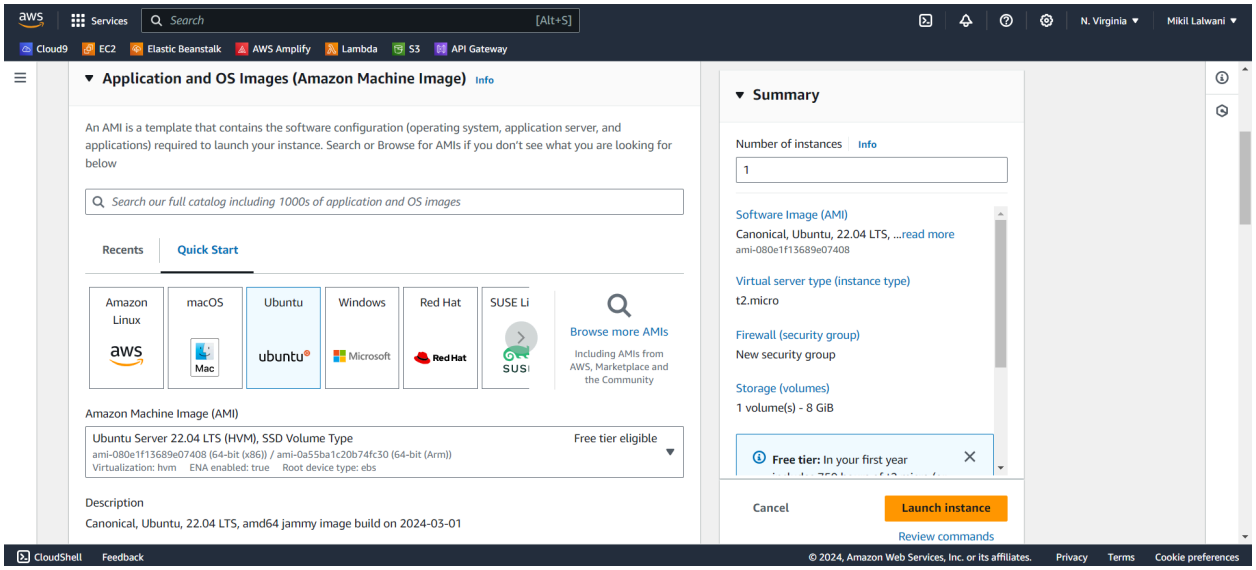
Aim -
To study and implement Security as a Service on AWS/Azure.

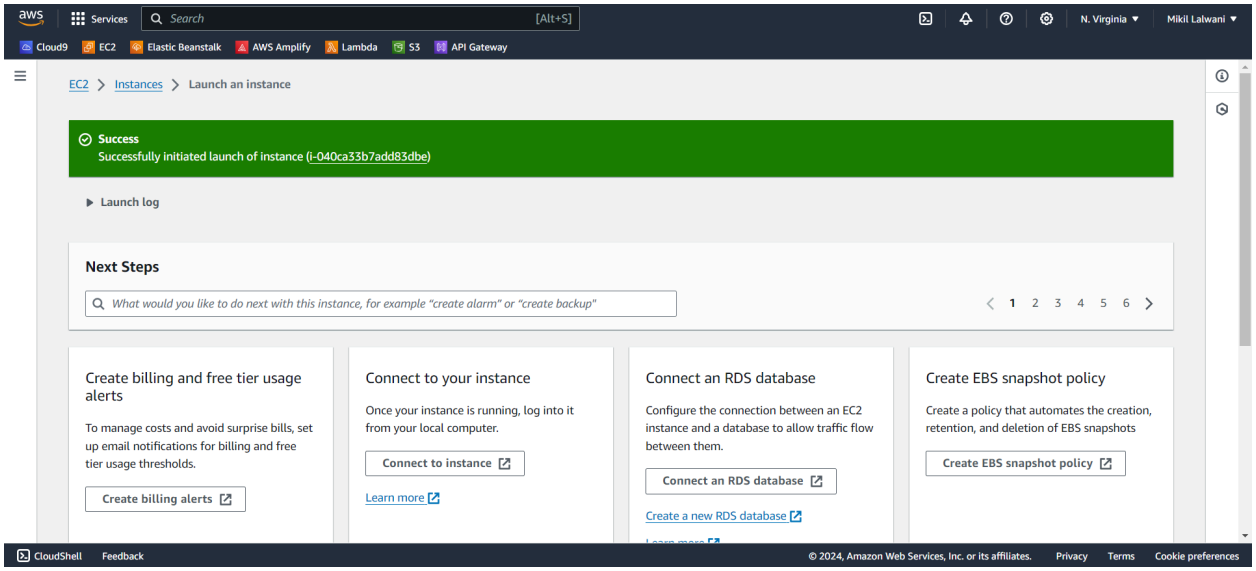
Theory -

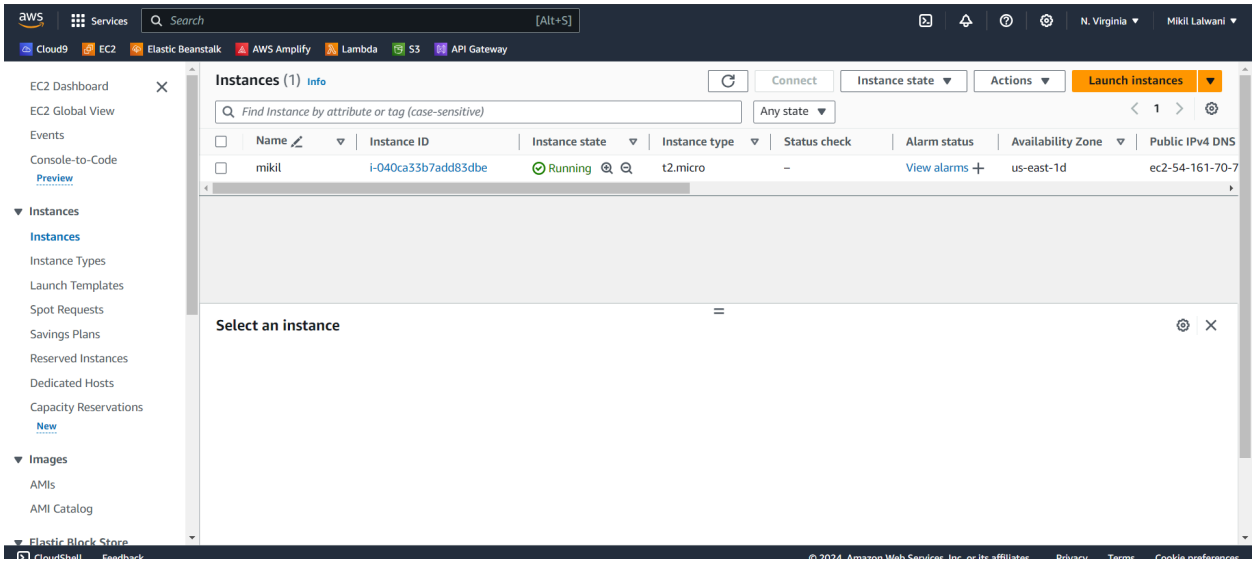
A business model called SECaaS, or Security as a Service, offers security to IT companies on a subscription basis. A superior security platform is provided by the outsourced approach, which lowers the total cost of ownership than the business could supply on its own. With the use of cloud computing, security for the company is maintained by an outside party. For the necessary computational and storage resources to run their websites and apps, many enterprises rely on security services. SECaaS is impressed by the “Security as a Service (SaaS)” model as applied to implement security kind services and doesn’t need on-premises hardware, avoiding substantial capital outlays. These security services typically embody authentication, antivirus, anti-malware/spyware, intrusion detection, penetration testing, and security event management, among others.
The former method of doing things involved paying direct pricing for hardware as well as ongoing fees for licenses to allow for the usage of that security code, which made it much more expensive. Instead, security as a service makes it simple and rational to use similar technologies.
Security can be availed by the following alliances:

- Encryption: makes the data unreadable until it has been authentically decoded, or encrypted.
- Network security: Network access management protocols are used to secure and keep an eye on network services.
- Email security: Protects against email frauds, spam, phishing, malware etc.
- Identification: Users can access with a valid login ID and legal permission, else forbids if it is not authenticated.
- Data loss prevention: Tools are built to monitor and secure data to protect from data loss.

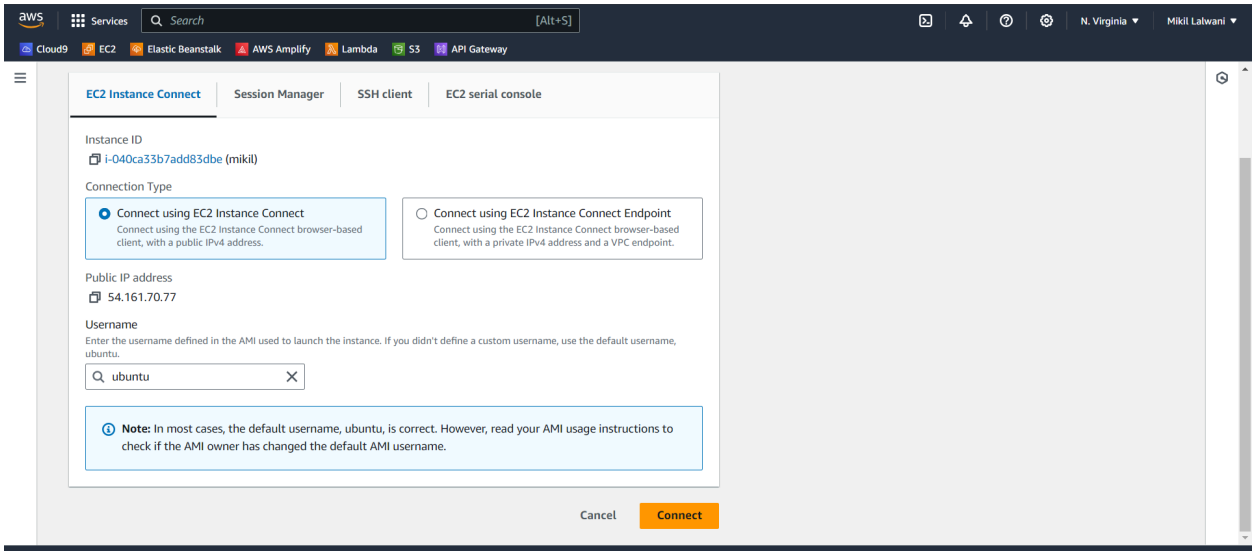
Procedure-
Login to your AWS Management Console.
Create a new AWS EC2 instance with Ubuntu OS.

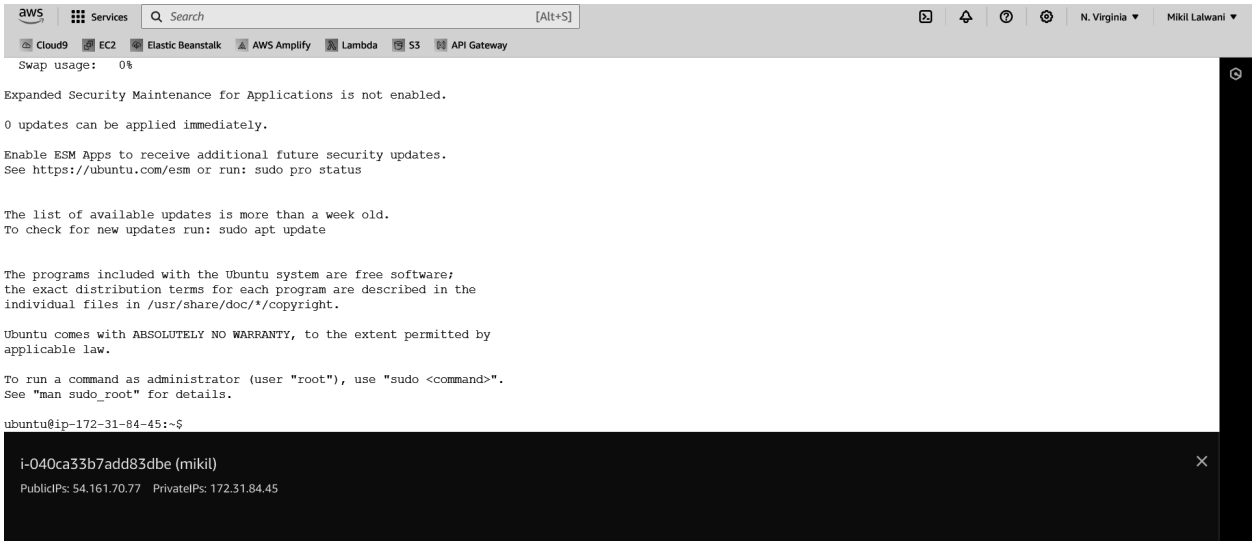

Services
Search
[Alt+S]
N. Virginia
Mikil Lalwani
Cloud9
EC2
Elastic Beanstalk
AWS Amplify
Lambda
S3
API Gateway
Application and OS Images (Amazon Machine Image) Info
An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. Search or Browse for AMIs if you don't see what you are looking for below
Search our full catalog including 1000s of application and OS images
Recents
Quick Start
Amazon Linux
macOS
Ubuntu
Windows
Red Hat
SUSE Li
Browse more AMIs
Including AMIs from AWS, Marketplace and the Community
Amazon Machine Image (AMI)
Ubuntu Server 22.04 LTS (HVM), SSD Volume Type
Free tier eligible
ami-080e1f13689e07408 (64-bit (x86)) / ami-0a55ba1c20b74fc30 (64-bit (Arm))
Virtualization: hvm ENA enabled: true Root device type: ebs
Description
Canonical, Ubuntu, 22.04 LTS, amd64 jammy image build on 2024-03-01
Summary
Number of instances Info
1
Software Image (AMI)
Canonical, Ubuntu, 22.04 LTS, ...read more
ami-080e1f13689e07408
Virtual server type (instance type)
t2.micro
Firewall (security group)
New security group
Storage (volumes)
1 volume(s) - 8 GiB
Free tier: In your first year
Cancel
Launch instance
Review commands
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


Services
Search
[Alt+S]
N. Virginia
Mikil Lalwani
Cloud9
EC2
Elastic Beanstalk
AWS Amplify
Lambda
S3
API Gateway
EC2 > Instances > Launch an instance
Success
Successfully initiated launch of instance (i-040ca33b7add83dbe)
Launch log
Next Steps
What would you like to do next with this instance, for example "create alarm" or "create backup"
1 2 3 4 5 6
Create billing and free tier usage alerts
To manage costs and avoid surprise bills, set up email notifications for billing and free tier usage thresholds.
Create billing alerts
Connect to your instance
Once your instance is running, log into it from your local computer.
Connect to instance
Learn more
Connect an RDS database
Configure the connection between an EC2 instance and a database to allow traffic flow between them.
Connect an RDS database
Create a new RDS database
Create EBS snapshot policy
Create a policy that automates the creation, retention, and deletion of EBS snapshots
Create EBS snapshot policy
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


Services
Search
[Alt+S]
N. Virginia
Mikil Lalwani
Cloud9
EC2
Elastic Beanstalk
AWS Amplify
Lambda
S3
API Gateway
EC2 Dashboard
EC2 Global View
Events
Console-to-Code Preview
Instances
Instances
Instance Types
Launch Templates
Spot Requests
Savings Plans
Reserved Instances
Dedicated Hosts
Capacity Reservations New
Images
AMIs
AMI Catalog
Elastic Block Store
Instances (1) Info
Connect
Instance state
Actions
Launch instances
Find Instance by attribute or tag (case-sensitive)
Any state
Name
Instance ID
Instance state
Instance type
Status check
Alarm status
Availability Zone
Public IPv4 DNS
mikil
i-040ca33b7add83dbe
Running
t2.micro
–
View alarms
us-east-1d
ec2-54-161-70-7
Select an instance

Connect with the EC2 instance using the AWS built in console.





General Hardening

In this first step, you will implement some initial hardening configurations to improve the overall security of your SSH server.
The exact hardening configuration that is most suitable for your own server depends heavily on your own threat model and risk threshold. However, the configuration you'll use in this step is a general secure configuration that will suit the majority of servers.
You will edit the main OpenSSH configuration file in /etc/ssh/sshd_config to set the majority of the hardening options in this tutorial. Before continuing it is a good idea to create a backup of your existing configuration file so that you can restore it in the unlikely event that something goes wrong.
Create a backup of the file using the following cp command:
sudo cp /etc/ssh/sshd_config /etc/ssh/sshd_config.bak

This will save a backup copy of the file to /etc/ssh/sshd_config.bak.

```
ubuntu@ip-172-31-84-45:~$ sudo cp /etc/ssh/sshd_config /etc/ssh/sshd_config.bak
ubuntu@ip-172-31-84-45:~$
```

Next, review the current default OpenSSH configuration options that correspond to the settings in /etc/ssh/sshd_config. To do this, run the following command:
sudo sshd -T
This will run OpenSSH server in extended test mode, which will validate the full configuration file and print out the effective configuration values.

You can now open the configuration file using nano or your favorite text editor to begin implementing the initial hardening measures:
sudo nano /etc/ssh/sshd_config
The first hardening option is to disable logging in via SSH as the root user. Set the PermitRootLogin option to no by uncommenting or editing the line in sshd_config:
PermitRootLogin no
This option will prevent a potential attacker from logging into your server directly as root. It also encourages good operational security practices on your part, such as operating as a non-privileged user and using sudo to escalate privileges only when absolutely needed.

```
#LoginGraceTime 2m
PermitRootLogin no
#StrictModes yes
#MaxAuthTries 6
#MaxSessions 10
```

Next, you can limit the maximum number of authentication attempts for a particular login session by configuring the MaxAuthTries option in sshd_config
MaxAuthTries 3
A standard value of 3 is acceptable for most setups, but you may wish to set this higher or lower depending on your own risk threshold.

```
# Authentication:

#LoginGraceTime 2m
PermitRootLogin no
#StrictModes yes
MaxAuthTries 3
File to insert [from ./]:
^G Help                                          M-F New Buffer
^C Cancel                                        M-N No Conversion
```

If required, you can also set a reduced login grace period, which is the amount of time a user has to complete authentication after initially connecting to your SSH server in sshd_config

LoginGraceTime 20m

The configuration file specifies this value in seconds.

```
#LogLevel INFO

# Authentication:

LoginGraceTime 20m
PermitRootLogin no
#StrictModes yes
MaxAuthTries 3
#MaxSessions 10

^G Help          ^O Write Out
^X Exit          ^R Read File
```

Setting this to a lower value helps to prevent certain denial-of-service attacks where multiple authentication sessions are kept open for a prolonged period of time.

If you have configured SSH keys for authentication, rather than using passwords, disable SSH password authentication to prevent leaked user passwords from allowing an attacker to log in sshd_config

PasswordAuthentication no

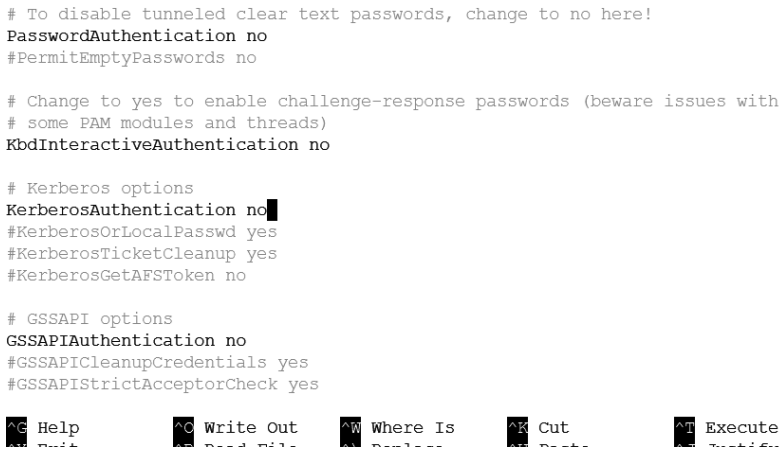
```
# Don't read the user's ~/.rhosts and ~/.shosts files
#IgnoreRhosts yes

# To disable tunneled clear text passwords, change to no here!
PasswordAuthentication no
#PermitEmptyPasswords no

# Change to yes to enable challenge-response passwords (beware issues with

^G Help          ^O Write Out     ^W Where Is      ^K Cut           ^T Execute       ^C
^X Exit          ^R Read File     ^\ Replace       ^U Paste         ^J Justify       ^/
```

As a further hardening measure related to passwords, you may also wish to disable authentication with empty passwords. This will prevent logins if a user's password is set to a blank or empty value in sshd_config

PermitEmptyPasswords no

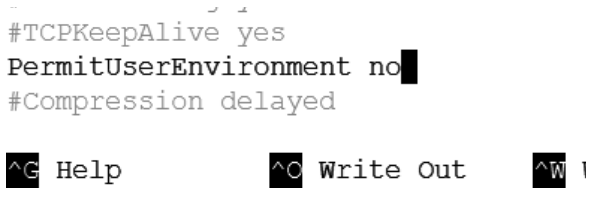
In the majority of use cases, SSH will be configured with public key authentication as the only in-use authentication method. However, OpenSSH server also supports many other authentication methods, some of which are enabled by default. If these are not

required, you can disable them to further reduce the attack surface of your SSH server in sshd_config
ChallengeResponseAuthentication no
KerberosAuthentication no
GSSAPIAuthentication no



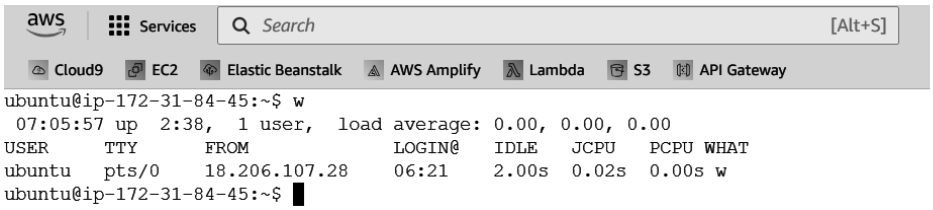
X11 forwarding allows for the display of remote graphical applications over an SSH connection, but this is rarely used in practice. Disable it if you are not running a graphical environment on your server in sshd_config
X11Forwarding no

OpenSSH server allows connecting clients to pass custom environment variables. For for example, a client can attempt to set its own $PATH or to configure terminal settings. However, like X11 forwarding, these are not commonly used, so you can disable the option in most cases in sshd_config
PermitUserEnvironment no



Implementing an IP Address Allowlist
You can use IP address allow lists to limit the users who are authorized to log in to your server on a per-IP address basis. In this step, you will configure an IP allowlist for your

OpenSSH server.
In many cases, you will only be logging on to your server from a small number of known, trusted IP addresses. For example, your home internet connection, a corporate VPN appliance, or a static jump box or bastion host in a data center.
By implementing an IP address allowlist, you can ensure that people will only be able to log in from one of the pre-approved IP addresses, greatly reducing the risk of a breach in the event that your private keys and/or passwords are leaked.
You can identify the IP address that you're currently connecting to your server with by using the w command:
W



This will output something similar to the following:
Locate your user account in the list and take a note of the connecting IP address. Here we use the example IP of 13.233.177.5
In order to begin implementing your IP address allowlist, open the OpenSSH server configuration file in nano or your preferred text editor:
sudo nano /etc/ssh/sshd_config
You can implement IP address allowlists using the AllowUsers configuration directive, which restricts user authentications based on username and/or IP address.
Your own system setup and requirements will determine which specific configuration is the most appropriate. The following examples will help you to identify the most suitable one:
Restrict all users to a specific IP address:
AllowUsers *@18.206.107.28

```
# Example of overriding settings on a per-user basis
#Match User anoncvs
#       X11Forwarding no
#       AllowTcpForwarding no
#       PermitTTY no
#       ForceCommand cvs server
AllowUsers *@18.206.107.28
```

Restrict all users to a specific IP address range using Classless Inter-Domain Routing (CIDR)
notation:
AllowUsers *@18.206.107.28/24

```
#       X11Forwarding no
#       AllowTcpForwarding no
#       PermitTTY no
#       ForceCommand cvs server
AllowUsers *@18.206.107.28/24
```

Restrict all users to a specific IP address range (using wildcards):
AllowUsers *@18.206.107.*

```
#       X11Forwarding no
#       AllowTcpForwarding no
#       PermitTTY no
#       ForceCommand cvs server
AllowUsers *@18.206.107.28.*
```

```
^G Help         ^O Write Out    ^W Where I
```

Restrict all users to multiple specific IP addresses and ranges:
AllowUsers *@18.206.107.28 *@18.206.107.29 *@192.0.2.0/24 *@172.16.*.1

```
#       X11Forwarding no
#       AllowTcpForwarding no
#       PermitTTY no
#       ForceCommand cvs server
AllowUsers *@18.206.107.28 *@18.206.107.29 *@192.0.2.0/24 *@172.16.*.1
```

```
^G Help         ^O Write Out    ^W Where Is     ^K Cut          ^T Execute
```

Disallow all users except for named users from specific IP addresses:
AllowUsers sammy@18.206.107.28 alex@18.206.107.29

```
#Match User anoncvs
#       X11Forwarding no
#       AllowTcpForwarding no
#       PermitTTY no
#       ForceCommand cvs server
AllowUsers sammy@18.206.107.28 alex@18.206.107.29
```

Restrict a specific user to a specific IP address, while continuing to allow all other users to log in without restrictions:
Match User ashley
AllowUsers sammy@18.206.107.28

```
#       X11Forwarding no
#       AllowTcpForwarding no
#       PermitTTY no
#       ForceCommand cvs server
AllowUsers sammy@18.206.107.28
```

```
^G Help          ^O Write Out     ^W Where Is
```

Save and close the file, and then proceed to test your configuration syntax:
sudo sshd -t

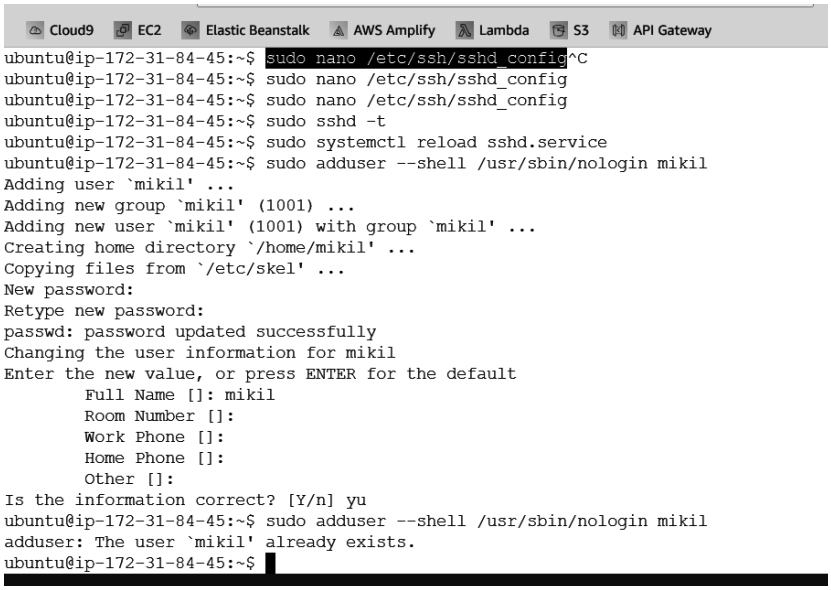
```
ubuntu@ip-172-31-84-45:~$ sudo nano /etc/ssh/sshd_config
ubuntu@ip-172-31-84-45:~$ sudo sshd -t
ubuntu@ip-172-31-84-45:~$
```

If no errors are reported, you can reload OpenSSH server to apply your configuration:
sudo systemctl reload sshd.service

```
ubuntu@ip-172-31-84-45:~$ sudo nano /etc/ssh/sshd_config
ubuntu@ip-172-31-84-45:~$ sudo sshd -t
ubuntu@ip-172-31-84-45:~$ sudo systemctl reload sshd.service
ubuntu@ip-172-31-84-45:~$
```

Restricting the Shell of a User

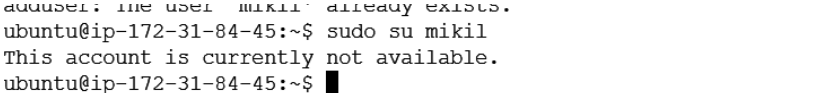
In this step, you'll explore the various options for restricting the shell of an SSH user.
In addition to providing remote shell access, SSH is also great for transferring files and other data, for example, via SFTP. However, you may not always want to grant full shell access to users when they only need to be able to carry out file transfers.
There are multiple configurations within OpenSSH server that you can use to restrict the shell environment of particular users. For instance, in this tutorial, we will use these to create SFTP-only users.
Firstly, you can use the /usr/sbin/nologin shell to disable interactive logins for certain user accounts, while still allowing non-interactive sessions to function, like file transfers, tunneling, and so on.
To create a new user with the nologin shell, use the following command:
sudo adduser --shell /usr/sbin/nologin mikil



If you then attempt to interactively log in as one of these users, the request will be rejected:

sudo su mikil

This will output something similar to the following message:



Despite the rejection message on interactive logins, other actions such as file transfers will still be allowed.

Advanced Hardening

In this final step, you will implement various additional hardening measures to make access to your SSH server as secure as possible.

OpenSSH servers can impose restrictions on a per-key basis. Specifically, restrictions can be applied to any public keys that are present in the .ssh/authorized_keys file. This ability is particularly useful to control access for machine-to-machine sessions, as well as providing the ability for non-sudo users to control the restrictions for their own user account.

While you can apply most of these restrictions at the system or user level using the /etc/ssh/sshd_configuration file, it can be advantageous to implement them at the key-level as well, to provide defense-in-depth and an additional failsafe in the event of accidental system-wide configuration errors.

Begin by opening your .ssh/authorized_keys file in nano or your preferred editor:

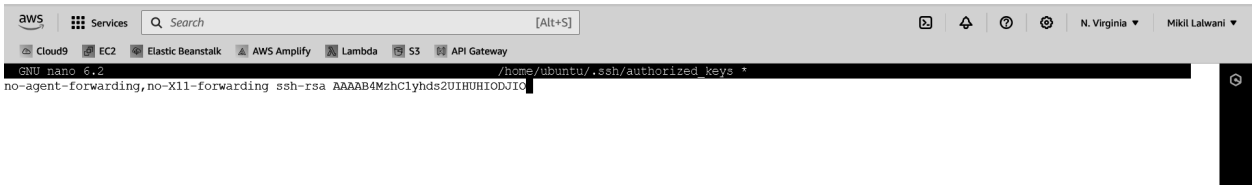
nano ~/.ssh/authorized_keys

Once you've opened your authorized_keys file, you will see that each line contains an SSH public key, which will most likely begin with something like ssh-rsa AAAB....

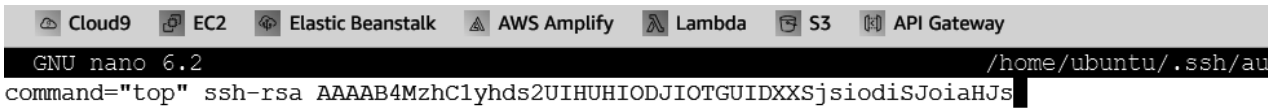
Additional configuration options can be added to the beginning of the line, and these will only apply to successful authentications against that specific public key. The following restriction options are available:

- no-agent-forwarding: Disable SSH agent forwarding.
- no-port-forwarding: Disable SSH port forwarding.
- no-pty: Disable the ability to allocate a tty (i.e. start a shell).
- no-user-rc: Prevent execution of the ~/.ssh/rc file.
- no-X11-forwarding: Disable X11 display forwarding.

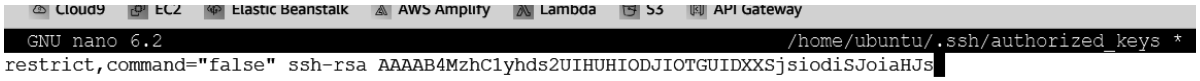
You can apply these to disable specific SSH features for specific keys. For example, to disable agent forwarding and X11 forwarding for a key, you would use the following configuration:



You may also wish to consider using the command option, which is very similar to the ForceCommand option described in Step 3. This doesn't provide a direct benefit if you're already using ForceCommand, but it is good defense-in-depth to have it in place, just in the unlikely event that your main OpenSSH server configuration file is overwritten, edited, and so on. For example, to force users authenticating using a specific key to execute a specific command upon login, you can add the following configuration:



Finally, to best use the per-key restrictions for the SFTP-only user that you created in Step 3, you can use the following configuration:



The restrict option will disable all interactive access, and the command="false" option acts as a second line of defense in the event that the ForceCommand option or nologin shell were to fail.

Save and close the file to apply the configuration. This will take effect immediately for all new logins, so you don't need to reload OpenSSH manually.

In conclusion, you reviewed your OpenSSH server configuration and implemented various hardening measures to help secure your server. The options that you configured

have reduced the overall attack surface of your server by disabling unused features and locking down the access of specific users.

Conclusion -

Thus, we successfully studied and implemented Security as a Service on AWS/Azure.