Introduction

Traditional network security involves protecting on-premise or internal networks from attacks and exfiltration. An internal network presents a simpler set of requirements because everything on the internal network is fairly easy to identify as trusted. Generally, traditional network security uses firewalls and intrusion detection/prevention solutions to identify untrusted, external network access.

However, as remote and hybrid workspaces have become more necessary and popularized, enterprises have encountered new challenges regarding how to secure their networks when dealing with remote employees and cloud-based vendors. Although the vendors of the cloud resources share some responsibility for security, management is often difficult when dealing with multiple vendors who all have different management expectations.

Cloud computing has brought a great deal of benefit to both large- and small-scale organizations. Primarily, organizations can now scale their infrastructure without incurring high capital costs. They can hire more employees without needing to increase their physical infrastructure. They can also expand their services and product development by leveraging cloud-based solutions, which can often be cheaper and less of a demand on the physical infrastructure.

For this reason, network security professionals should be aware of the two major methods of managing cloud network security: virtual networks and zero-trust models. A virtual network is used by professionals to essentially define a perimeter by creating a virtual network among the cloud resources and using firewalls to protect the perimeter.

Zero-trust fundamentally operates as if there are no perimeters, meaning there is no such thing as an “internal” trusted network regardless of the physical setup of the network. Under zero-trust, all networks are hostile, there are always threats, and it is essential to authorize and authenticate everything.

Your company is moving its web services to run on cloud-based machines. To minimize vendor lock-in (when a client must use a vendor because switching to another is too costly), your architecture uses a private Docker registry for holding production code images and runs services as Docker images on virtual instances from various cloud providers. This does not require any advanced provider features, and most providers have a method of running customer-managed virtual instances. However, this means all traffic is over the public internet. In this lab, you will be focusing on implementing a zero-trust model in which every cloud resource will implement strict access controls.

Lab Overview

This lab has **three** parts, which should be completed in the order specified.

1. In the first part of the lab, you will use OpenSSL to create x509 certificates for using mutual TLS among the components of the network.

2. In the second part of the lab, you will configure a Docker registry and client to use mutual TLS.

3. In the third part of the lab, you will configure a front end web server and an origin web server to use mutual TLS to secure their communication channel.

Finally, you will explore the virtual environment on your own. You will answer questions and complete challenges that allow you to use the skills you learned in the lab to conduct independent, unguided work - similar to what you will encounter in a real-world situation.

Learning Objectives

Upon completing this lab, you will be able to:

1. Explain how the Zero Trust Model can be used to secure cloud networks.

2. Create x509 certificates for mutual TLS.

3. Enable mutual TLS for protecting Docker registries.

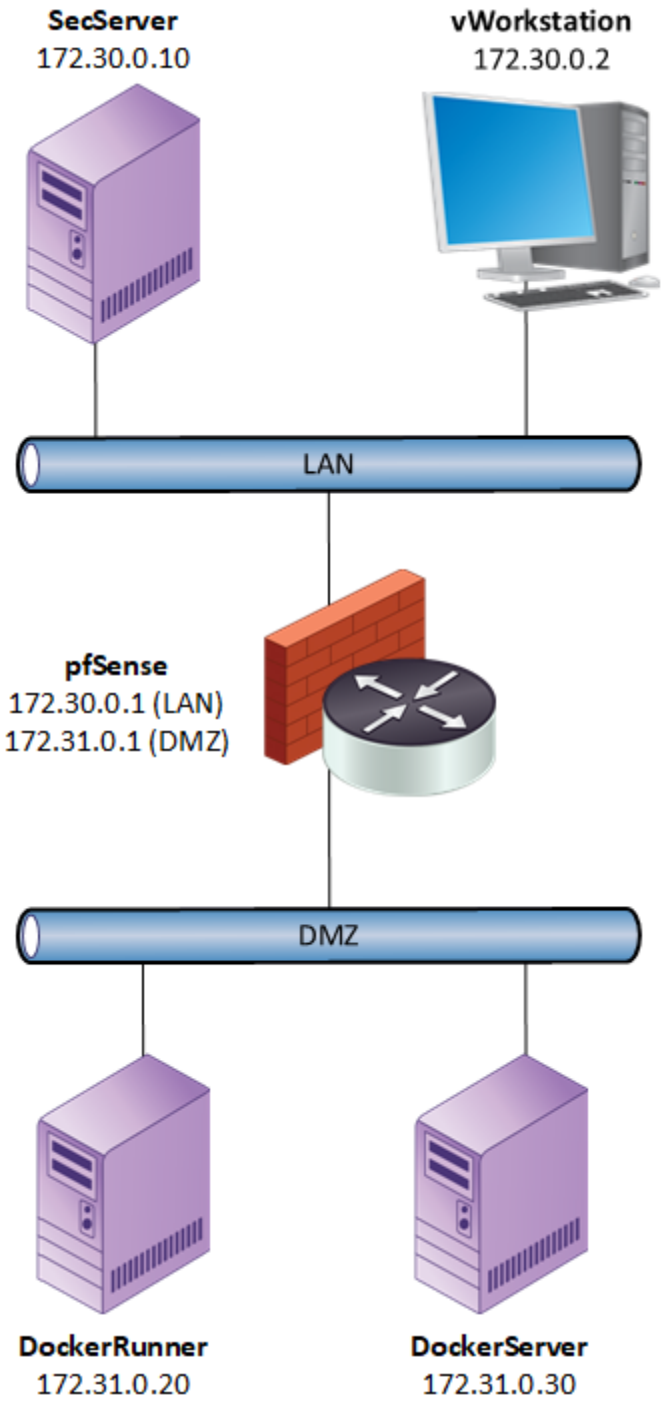
4. Enable mutual TLS for protecting origin servers.

Topology

This lab contains the following virtual machines. Please refer to the network topology diagram below.

- vWorkstation
- SecServer
- pfSense
- DockerServer

- DockerRunner



Tools and Software

The following software and/or utilities are required to complete this lab. Students are encouraged to explore the internet to learn more about the products and tools used in this lab.

- nginx
- Docker
- openssl

Deliverables

Upon completion of this lab, you are required to provide the following deliverables to your instructor:

Hands-On Demonstration

1. Lab Report file, including screen captures of the following:

- At least the first 12 lines of the output for the root certificate
- The X509v3 extensions of the registry certificate
- The unknown certificate authority error
- The bad certificate error
- A successful pull of registry:5000/alpine on DockerRunner
- The running frontend and backend containers
- The successful page load from App (http) without mutual TLS active
- The successful page load from origin (http)
- The successful page load from App (http) with mutual TLS active
- The unsuccessful page load from origin (https)

Challenge and Analysis

1. Lab Report file, including screen captures of the following:

- At least the first 12 lines of the developer certificate
- A successful pull of registry:5000/alpine on SecServer
- The output of the verification command for the developer certificate

Hands-On Demonstration

Note: In this section of the lab, you will follow a step-by-step walk-through of the objectives for this lab to produce the expected deliverable(s).

1. **Review** the **Tutorial**.

   Frequently performed tasks, such as making screen captures and downloading your Lab Report, are explained in the Cloud Lab Tutorial. The Cloud Lab Tutorial is available from the User menu in the upper-right corner of the Student Dashboard. You should review these tasks before starting the lab.

2. **Proceed** with **Part 1**.

Part 1: Create Certificates for Using Mutual TLS

Note: Using a zero-trust model requires all resource access to be authenticated and authorized. Traditional user authentication uses username and password pairs. However, in the zero-trust model, there are many more agents that need to access resources, and interactive authentication is not an option for some of them.

One way to secure access is by using x509 certificates. The base standard for these certificates can be found here, which contains some highly detailed information. For the purposes of mutual TLS, an x509 certificate establishes a link between one or more identifiers and a public key. The certificate is signed by a certificate authority (CA). If you trust the CA, then a certificate provides you with a verified public key for the identifiers in the certificate.

To use the certificates for TLS, you need a Public Key Infrastructure (PKI). A common use of TLS is for verifying the owners of a website to establish a secure connection using https. In this case, the PKI is formed by a set of CAs that are trusted by browser. Websites that wish to provide secure connections must obtain x509 certificates from one of the CAs that connects the domain name for the site to the key pair used by the web server.

For a zero-trust model, both sides of the communication should be authenticated. Thus, both sides must have an x509 certificate. All parties must also agree on the set of CAs that are trusted. A common solution is to create a private CA. The root certificate for that CA is then provided to each party to be used for verification.

You need to enable authentication for all privileged access. The first concerns are as follows:

- Setting up access to the docker registry

- Traffic between the front-end containers and the origin containers for web services

You can use mutual TLS to enable both client and server authentication for TLS connections. You can also create your own PKI based on a root certificate you control. You will be using the SecServer to create certificates on openssl. You will begin by creating a root certificate.

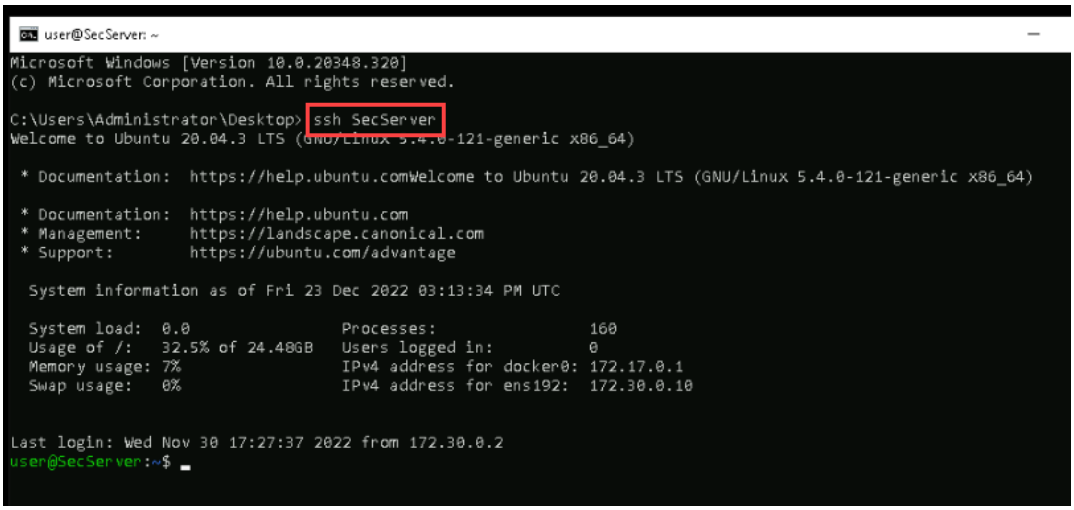
1. If necessary, **log in** to vWorkstation using the following credentials.
   Username: `Administrator`
   Password: `P@ssw0rd!`

2. On the vWorkstation desktop, **double-click** the **Command Prompt icon** to open a command prompt window.

Command Prompt

Note: This command prompt shortcut has been set up so that the working directory is the Desktop.

3. At the command prompt, **type** `ssh SecServer` and **press Enter** to start a new session.



ssh SecServer command

Note: You should see a shell prompt for user@SecServer. Windows has OpenSSH installed by default. The team has set up the machines to use SSH key pairs for authentication, so no passwords are needed for connecting. They also created a configuration file to ease the process.

4. At the command prompt, **type** `cd CA` and **press Enter** to change the working directory on the server.

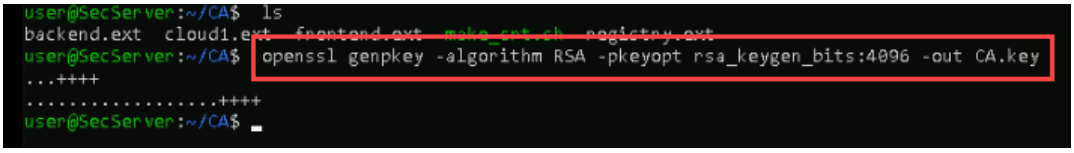
cd command

5. At the command prompt, **type** `ls` and **press Enter** to see the files that are already there.

ls command

Note: There are five files listed. The four ".ext" files contain options for openssl that will be used when signing certificates. The details will be discussed later. The script contains commands to create a certificate. Before using the script, you will create some certificates manually. The first step is to create the private key for the root certificate of your certificate authority.

6. In the command prompt, **type** **`openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -out CA.key`** and **press Enter** to create a key pair.



openssl genpkey command

Note: The openssl program has many uses. One of those is creating keys. The arguments used in the previous command are the following:

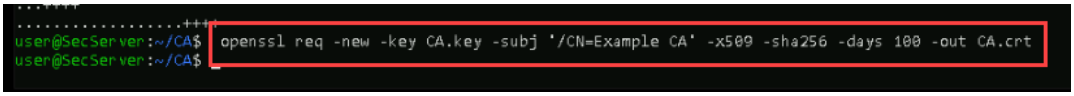
- **genpkey** is the openssl subcommand that indicates you want to create a private key.
- **-algorithm rsa** specifies the private key should be generated using the RSA algorithm (i.e., generate an RSA private key).
- **-pkeyopt rsa_keygen_bits:4096** indicates the number of bits to use for the key. Today (2022), 2048 bits is a reasonable size for RSA keys for production server certificates.

However, a certificate authority key will typically have a longer life, so using a larger key size for longer-term security is warranted.

- **-out CA.key** specifies the name of the file to be used for the private key. While only one file is generated, both the private key and the public key can be determined from the file. Since the file contains the private key, it must be protected. This file will be generated with 0600 permissions (read and write for the owner only) to help with this protection.

For simplicity, you have not included a password to protect this key. In a real production system, the root certificate key would need to be well protected. When the command runs, you will see a series of dots and plus signs as the key is generated. Next, you will create the self-signed certificate.

7. At the command prompt, **type** `openssl req -new -key CA.key -subj '/CN=Example CA' -x509 -sha256 -days 100 -out CA.crt` and **press Enter** to create the certificate**.**



openssl req command

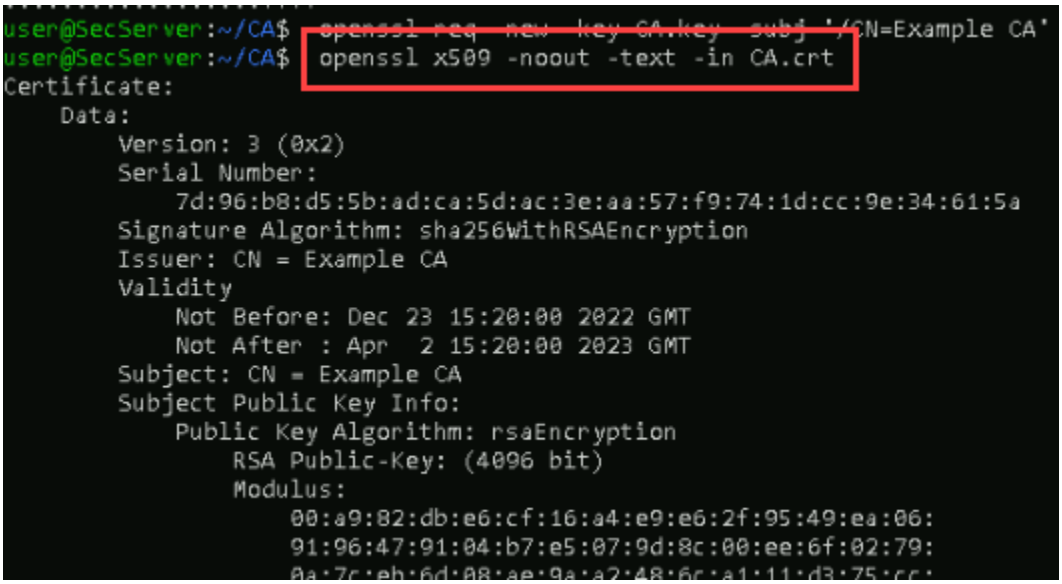
Note: This command combines two steps in the typical process of creating an x509 certificate. The first step is to create a certificate request. The second step is creating a certificate and signing it based on the request using the certificate authority key. For the root certificate, the same key is used to create the request and to sign it. For this reason, a root certificate may also be referred to as a self-signed certificate. The individual arguments are as follows:

- **req** is the openssl subcommand to generate a request.
- **-new** indicates that a new certificate request should be created.
- **-key CA.key** specifies the key to use for the request.

- **-subj '/CN=Example CA'** sets the distinguished name for the request.
- **-x509** indicates that the request should be signed using the same key used for the request. Effectively, this indicates the output should be a self-signed certificate instead of a certificate request.
- **-sha256** specifies the hash that should be used as a part of the signing process.
- **-days 100** gives the number of days the certificate should be considered valid from the current date.
- **-out CA.crt** gives the name of the file for the resulting certificate.

The resulting file is not human readable. For x509 certificates, there is a very specific encoding process that is used to create a sequence of bytes to represent the certificate. To inspect the contents of a certificate, you can use another subcommand of openssl.

8. At the command prompt, **type** `openssl x509 -noout -text -in CA.crt` and **press Enter** to see the certificate contents.



openssl x509 command

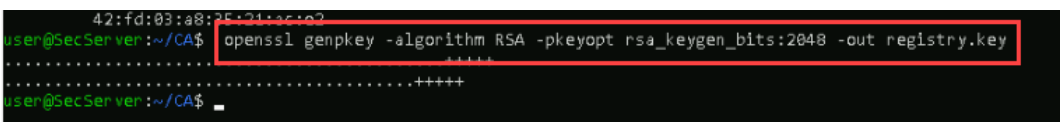
9. **Make a screen capture** showing at least the first 12 lines of the output for the root certificate.

Note: The output shows the details of the certificate in a human-readable form when possible. Some of the fields are essentially big integers, which are shown using a hex (hexadecimal, or base16) encoding. The certificate makes the claim that the distinguished name given by the Subject field is linked to the public key given by the Subject Public Key Info field for the time span indicated by the Validity field. This claim is signed as being valid by the issuer using the issuer's private key.
Farther down in the output of the command is a section called X509v3 extensions. This section can contain additional constraints or information the signing certificate authority is placing on the certificate. This root certificate contains three extensions, which openssl includes by default for new CAs:

- **X509v3 Subject Key Identifier** – a hash of the public key of the subject of the certificate
- **X509v3 Authority Key Identifier**– a hash of the public key of the issuer of the certificate
- **X509v3 Basic Constraints**– specify if this is a CA

Since this is a self-signed certificate, the Subject and the Issuer are the same, so the Subject and Authority Key Identifier hashes should be the same.
The basic constraints include CA:TRUE to indicate that the certificate can be used for a certificate authority. Now you can use CA.key and CA.crt to create and sign certificates for each agent in your network. First, you will create a certificate for the Docker registry.

10. At the command prompt, **type** **`openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 -out registry.key`** and **press Enter** to create the key pair.



openssl genpkey command

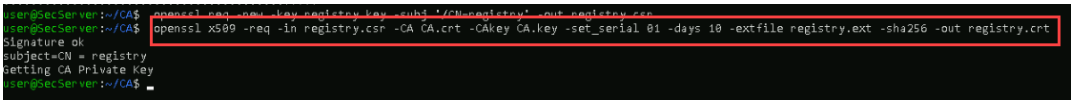
Note: Here you have used a command similar to the one for the CA key. A server certificate will typically have a shorter life span than a certificate authority, so using a 2048-bit key is reasonable.

11. At the command prompt, **type** `openssl req -new -key registry.key -subj '/CN=registry' -out registry.csr` and **press Enter** to create a certificate request**.**

openssl req command

Note: Now the process is split into two steps. First, the request is created (as you have just done above); then it is signed by the certificate authority. The arguments used for the req command are the same as those used to create the self-signed root certificate. However, the extra arguments about signing are not included. Those will be included in the command to sign the request.

12. At the command prompt, **type** `openssl x509 -req -in registry.csr -CA CA.crt -CAkey CA.key -set_serial 01 -days 10 -extfile registry.ext -sha256 -out registry.crt` and **press Enter**.



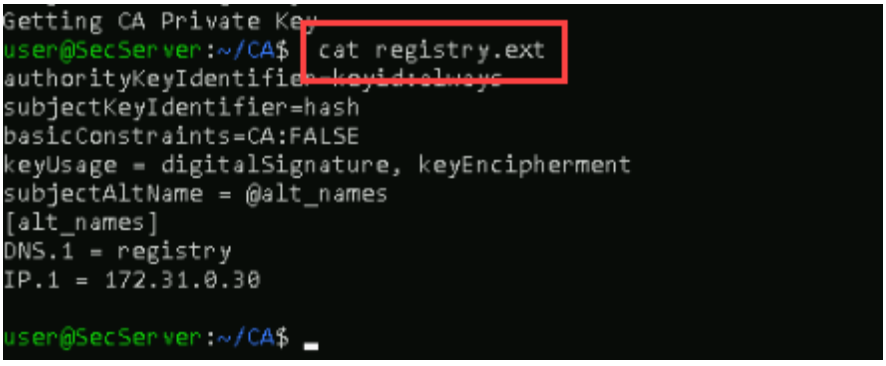
openssl x509 command

Note: The x509 subcommand also used for signing requests. The arguments are the following:

- **x509** is the subcommand to use.
- **-req** indicates that a certificate request will be provided as input.
- **-in registry.csr** specifies the input certificate request file.
- **-CA CA.crt** gives the certificate to be used as the issuer.
- **-CAkey CA.key** gives the key file corresponding to the issuer certificate.
- **-set_serial 01** indicates the serial number to use for the generated certificate.
- **-extfile registry.ext** gives a file containing the extensions to include in the certificate.

The other arguments have the same meaning as in the earlier command used to generate the self-signed or root certificate. Note that the life span here is shorter. The extensions file needs a little more explanation.

13. At the command prompt, **type** `cat registry.ext` and **press Enter** to view the extensions file.



cat registry.ext command

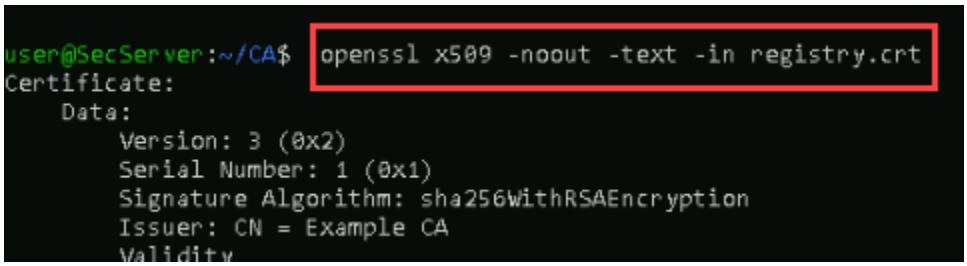
Note: This file has an INI file format, where sections are indicated by lines beginning with labels in square brackets. Other lines are key-value lines. In the default section (with no label), the extensions file gives a list of the extensions to be used. Here, your team has decided on the following, the first three which you may recall from the CA's root certificate:

- **subjectKeyIdentifier (SKI)**–a hash of the public key of the subject.
- **authorityKeyIdentifier (AKI)**–a hash of the public key used to sign the certificate.
- **basicConstraints** –specify if this is a CA,.
- **keyUsage**– intended uses for the certificate. The full details of these are outside the scope of this lab, but they are important in production systems.
- **subjectAltName**– gives a section that contains alternate identifiers for the subject of the certificate.

The SKI and AKI provide ways to identify the keys used in a certificate. You should notice that unlike in the root certificate, the hashes here are not the same, on account of the subject and issuer being different entities (i.e., this certificate is not self-signed).

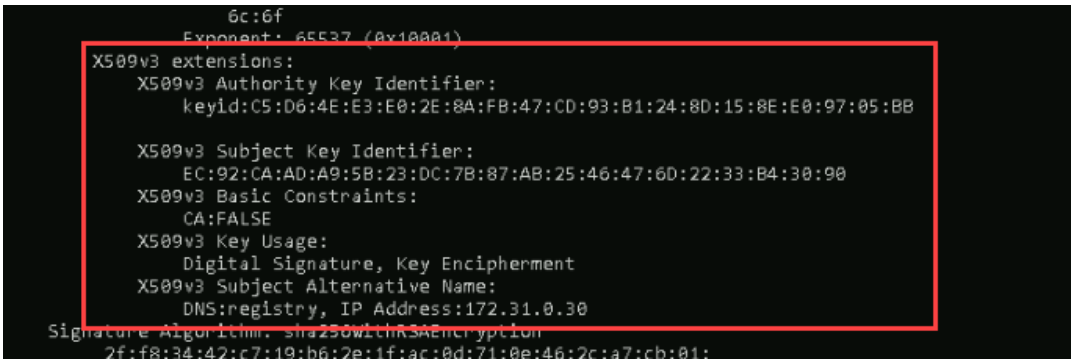
The subjectAltName extension allows multiple names to be associated with a particular agent, in addition to the distinguished name in the subject. Actually, the subject alternative name (SAN) is the recommended location for all distinguished names on a certificate, as most browsers no longer consider the subject's singular Common Name (CN) as a valid identifier without a matching SAN entry. In the corresponding section, you can add alternate identifiers of specific kinds. Here, a DNS name of registry and an IP address of 172.31.0.30 are also associated with the public key of the certificate. You can see these extensions in the generated certificate.

14. At the command prompt, **type** `openssl x509 -noout -text -in registry.crt` and **press Enter** to view the registry certificate**.**



openssl x509 command

15. **Make a screen capture** showing the X509v3 extensions of the registry certificate.



x509v3 extensions in the openssl x509 command results

Note: This certificate will allow the Docker registry to be authenticated. However, for the zero-trust model, you also want the Docker runner to authenticate to the registry, so you will need to create a certificate for that machine as well. It is possible to combine the steps to generate the new key and certificate request into one command.

16. At the command prompt, **type** `openssl req -newkey rsa:2048 -nodes -keyout cloud1.key -subj '/CN=cloud1' -out cloud1.csr` and **press Enter.**

openssl req command

Note: The following new arguments are used to generate the new key.

- **-newkey rsa:2048** says to generate a new certificate request, as well as a new private key with the specified algorithm and bit size..
- **-nodes** indicates that the key file should not be password protected/encrypted. Referring to the Data Encryption Standard (DES) algorithm, this option should be read as “no-DES”.
- **-keyout cloud1.key** indicates where to save the new key.

When the command executes, you should see output similar to that for the key generation command. The next step is to sign the request as was done for the registry certificate. Be sure to use "cert" for the final certificate file extension because it will be needed later.

17. At the command prompt, **type** **`openssl x509 -req -in cloud1.csr -CA CA.crt -CAkey CA.key -set_serial 02 -days 10 -extfile cloud1.ext -sha256 -out cloud1.cert`** and **press Enter** to sign the certificate**.**

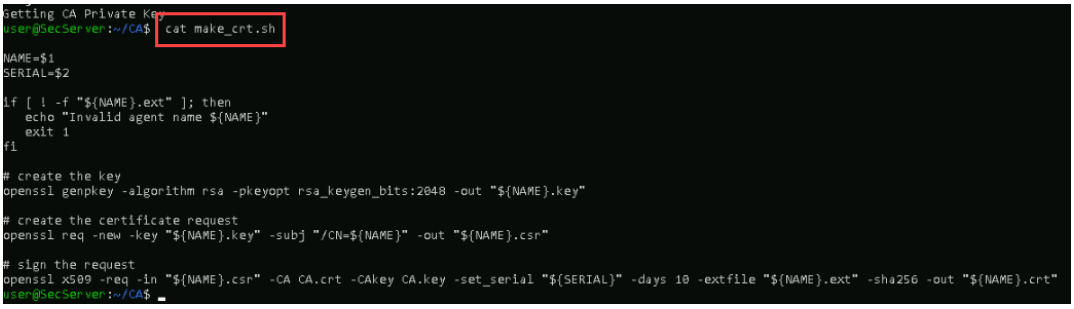
openssl x509 command

Note: Notice that the serial number is different. Each certificate signed by a particular CA must have a unique serial number. The recommended practice for production systems is to use random non-sequential numbers that are algorithmically generated. In this lab, you are using small integers for

simplicity.

You will also need keys for the web service frontend and the web service backend. Next you will create the certificates for each of these and sign them. A simple script has been created to simplify the process.

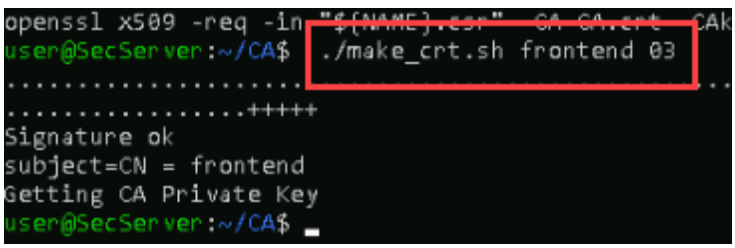
18. At the command prompt, **type** `cat make_crt.sh` and **press Enter** to see the script.



cat make_crt.sh command

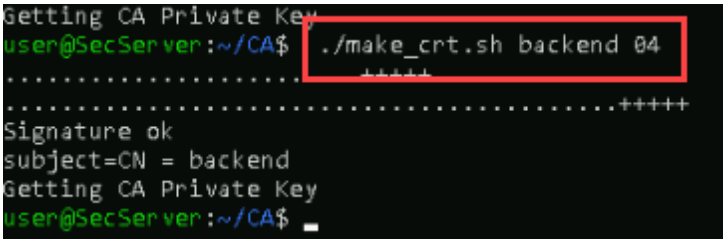
Note: The script takes two parameters, a name and a serial number. If there is no file "name.ext," then the script will exit. This makes the script create and sign only requests for which a configuration has been created. Then the script uses three commands to generate the key pair, create the certificate request, and then sign the request, all which should look familiar.

19. At the command prompt, **type** `./make_crt.sh frontend 03` and **press Enter** to create the frontend certificate.



make_crt.sh frontend command

20. At the command prompt, **type** `./make_crt.sh backend 04` and **press Enter** to create the backend certificate.



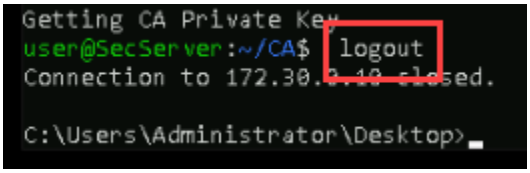
make_crt.sh backend command

Note: You have used a mostly manual process to create the certificates to be used for mutual TLS in this lab. In a real production system, you would need to fulfill the requirements of a real PKI. Some of those responsibilities include maintaining certificate revocation lists to allow issued certificates to be revoked and running a server to respond to queries about revoked certificates. Your team would also develop policies about how the distinguished name fields and the subjectAltNames should be used for identifying agents in your network.

Key management is also a critical issue for a PKI. Generally, the private key for a machine should be generated on that machine or using specialized hardware to protect keys. Then the certificate request can be sent to the CA without sending the private key over any networks. Usually, there will also be at least one intermediate CA used in the process. The root CA will sign a certificate for the intermediate CA, which can be done in a machine that is offline. Then the intermediate CA can be used on a network-connected machine for signing agent certificates. It is possible for the chain of certificates to have more intermediates as well.

For a system that has agents that can be added automatically depending on demand, the steps of managing the PKI need to be automated. There are many systems available for managing a PKI that provide varying levels of completeness. For auto-scaling systems, this needs to be automated so that you can automatically provision new certificates for new instances. Many organizations run into identity management issues that center largely around the lack of know-how, too many or too few administrators generating or having access to keys, and lack of proper tracking of certificates. Automated systems and standardizing your practices will go a long way in creating the right security protocols to avoid future breaches.

21. At the command prompt, **type** `logout` and **press Enter** to end the session with SecServer.



logout command

Part 2: Enable Zero Trust for Docker

Note: In this next part of the lab, you will work with Docker and create a zero-trust infrastructure. Docker provides a good example of a zero-trust model's utility, as containers make traditional firewall-based security more complex and more cumbersome to maintain. Docker containers may have different IPs than the host on which they are running, and Docker networks may even be migrated between platforms (e.g., from on-premise to a cloud environment). Especially in a multi-cloud environment, which may be running microservices and container workloads across many networks, it becomes more tenable to control access based on individual users and workloads (*who*) rather than location in the infrastructure (*where*).

You will begin by focusing on the private Docker registry running in this environment. As an alternative to using the online Docker Hub to store and retrieve Docker images, you may host your own private Docker registry using a public Docker image called 'registry'. A private registry provides greater control over images available to the runners in your environment. However, you are required to configure access to that registry, ensuring that only those with the appropriate authentication details may interact with it.

Your goal is to make it so that:

- the Docker registry requires clients to have a valid certificate.
- the Docker engine on clients requires the registry to have a valid certificate.
- all "valid" certificates are signed by your root CA key.

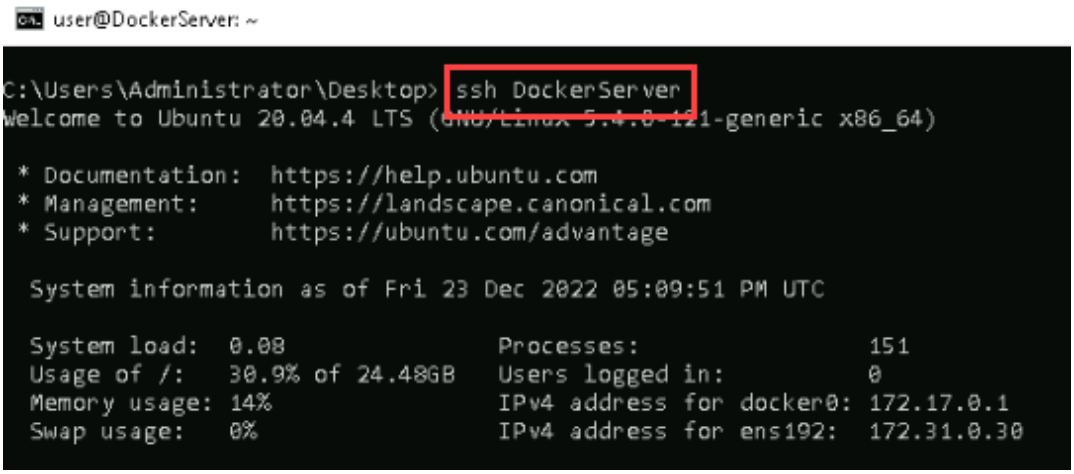
By requiring mutual TLS between the Docker agents, you can prevent unauthorized agents from pushing images to your registry and also prevent unauthorized agents from pulling images from the

registry. Your team has already prepared the DockerServer machine with a running registry. They have also pulled a few images that can be used during testing to ensure that the registry will function properly without accessing Docker Hub.

The default Docker registry configuration does not use TLS. Your first task is to update the configuration of the registry container so that it will use TLS. This can be accomplished by modifying the configuration file for the registry. The default file is part of the container image.

1. If necessary, on the vWorkstation desktop, **double-click** the **Command Prompt icon** to open a command prompt window.

2. At the command prompt, **type** `ssh DockerServer` and **press Enter** to connect to the docker registry machine**.**



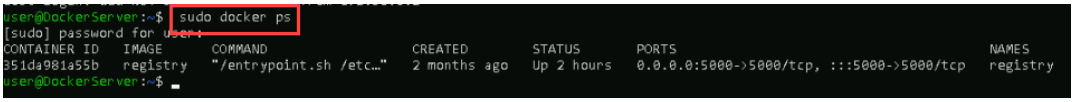
ssh DockerServer command

Note: You should now see a prompt starting with user@DockerServer. This is a shell on the machine that serves as the registry for Docker images.

3. In the DockerServer session, **type** `sudo docker ps` and **press Enter** to view the running

containers**.**

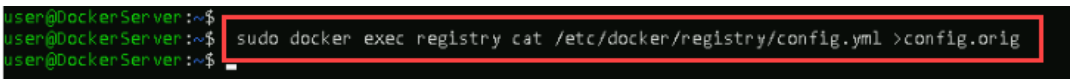
When requested, **type** `password` and **press Enter** to provide the sudo password.



sudo docker ps command

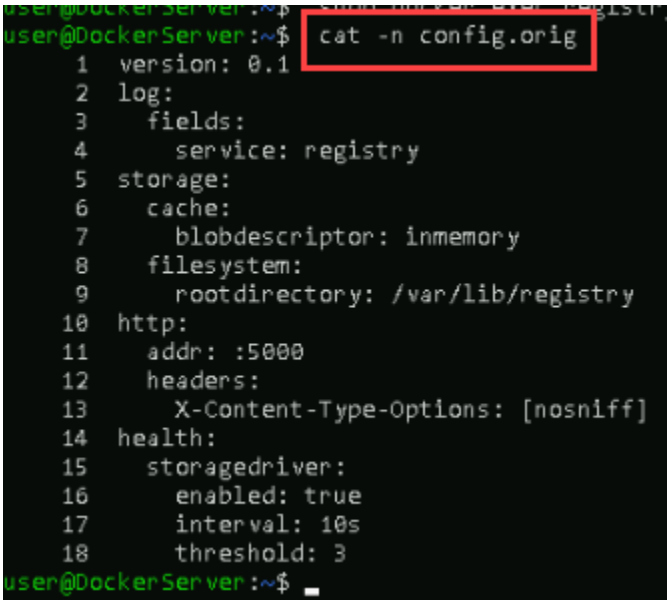
Note: You can use this same password if you are ever prompted for a sudo password again. The Docker ps command will give you a list of the running Docker containers on the machine. You should see one entry named *registry* using the registry image. The status should indicate that it is currently running.

4. In the DockerServer session, **type** `sudo docker exec registry cat /etc/docker/registry/config.yml >config.orig` and **press Enter** to copy the default configuration from the running Docker container to your DockerServer machine.



sudo docker exec command

5. In the DockerServer session, type `cat -n config.orig` and **press Enter** to view the configuration file with line numbers.



cat config.orig command

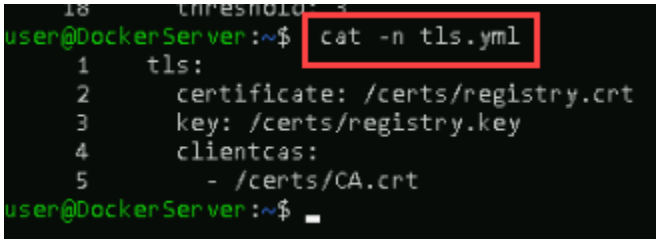
Note: The docker exec command allows you to run a command in the context of a container. Here you have used cat to extract the contents of the default configuration file located at /etc/docker/registry/config.yml in the container. The redirection of the output saved the results to a local file on the host. All the lines in the default configuration can be retained. A few notable lines are the following:

- line 9 rootdirectory: /var/lib/registry
  This indicates that the registry should save information about images in the specified folder of the container file system. The container was originally run with the DockerServer machine's /home/user/registry_data folder mapped to this path, so all the registry data can be preserved even if the current container is deleted and restarted.

- line 11 addr : :5000
  This indicates what address the registry should listen on inside the container. This port is used with the -p option in the Docker run command to expose the port to the outside network.

To enable TLS, you need to add a tls section to the http part to specify the files to use for the server key and certificate as well as a list of CA certificates to accept for verifying clients. This should be added to the http section so that it can be inserted after line 13.

6. In the DockerServer session, **type** `cat -n tls.yml` and **press Enter** to view the lines to be added.



cat tls.yml command

Note: Lines 2 and 3 give the files to use for the server certificate and key. Line 5 gives a certificate that can be used to verify client certificates. The paths used here must be paths that will be valid in the running container.

7. In the DockerServer session, **type** `sed -e "13 r tls.yml" config.orig >config.yml` and **press Enter** to insert the tls configuration.

sed >config.yml command

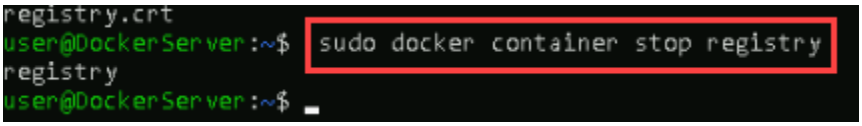
Note: This command uses sed (stream editor) to insert the contents of tls.yml after line 13 in the file config.orig. The output is directed to the file config.yml. Now you need to retrieve the required certificates and keys that you created on SecServer in Part 1 . You will accomplish this using the scp (secure copy) command which uses SSH to transfer files.

8. In the DockerServer session, **type** `scp SecServer:/home/user/CA/\{CA.crt,registry.key,registry.crt\} certs/.` and **press Enter** to copy the required files to the certs folder.

scp command

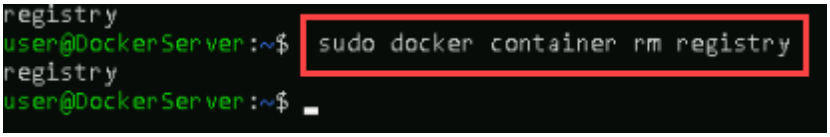
Note: You now have a new configuration and all the required files on the DockerServer machine. The container must be restarted with the new configuration to make it effective. This will take a few steps.

9. In the DockerServer session, **type** `sudo docker container stop registry` and **press Enter** to stop the currently running version.



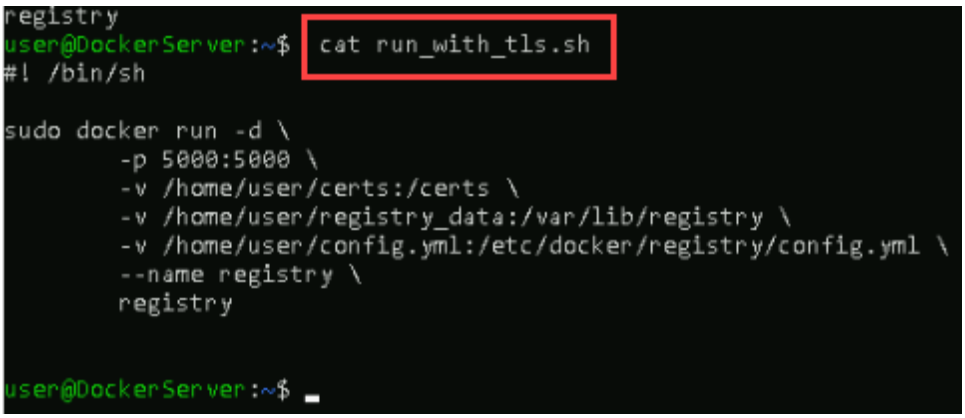
sudo docker container stop command

10. In the DockerServer session, **type** `sudo docker container rm registry` and **press Enter** to remove the container.



sudo docker container rm command

11. In the DockerServer session, **type** `cat run_with_tls.sh` and press Enter to view the command for running the container.



cat run_with_tls.sh command

Note: This Docker command has multiple arguments, which have the following meanings:

- run
  Create a new container and run it.

- -d
  Run the Docker container detached.

- -p 5000:5000
  Expose port 5000 of the Docker container on port 5000 of the host.

- -v /home/user/certs:/certs
  Mount the host directory /home/user/certs from the host at /certs in the container.

- -v /home/user/registry_data:/var/lib/registry
  Mount the host directory /home/user/registry_data at /var/lib/registry in the container. This host directory contains all the image data that was previously pushed to the registry server. Keeping

this volume preserves that data for the updated server.

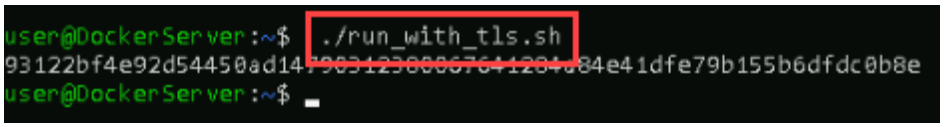
- -v /home/user/config.yml:/etc/docker/registry/config.yml
  Mount the new configuration file as the default configuration file in the container file system.

- --name registry
  Set the name of the created container.

- registry
  The name of the image to use to create the container.

Since the name of the registry container is going to be the same, the original container had to first be stop and removed.

12. In the DockerServer session, **type** `./run_with_tls.sh` and **press Enter** to run the container.



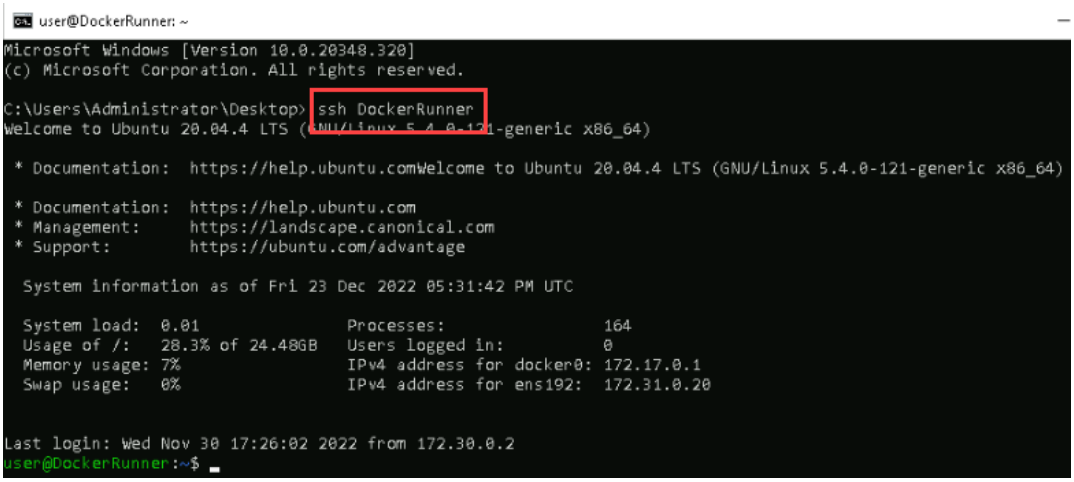
Execute run_with_tls.sh script

Note: You should see the container id for the new running container. Now you need to set up DockerRunner to use mutual TLS to connect to the registry.

13. On the vWorkstation desktop, **double-click** the **command prompt icon** to open a new command prompt.

14. At the new command prompt, **type** `ssh DockerRunner` and **press Enter** to connect to the

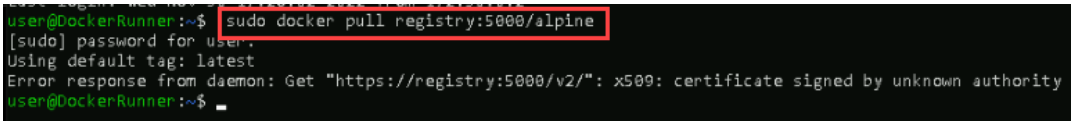
machine.



ssh DockerRunner command

15. In the DockerRunner session, **type** `sudo docker pull registry:5000/alpine` and **press Enter** to attempt to pull an image.

    When requested, **type** `password` and **press Enter** for the sudo password.



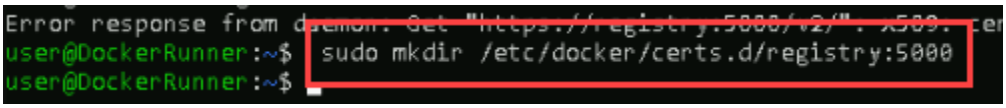
sudo docker pull command

Note: You should get an error message from this attempt. The Docker engine attempts to make an https connection to the registry at registry:5000. The registry then responds with the certificate that was generated for the server (registry.crt). However, Docker engine on DockerRunner does not yet know about the certificate authority used to sign the certificate. It issues an error that the certificate is signed by an unknown authority.

16. **Make a screen capture** showing the unknown certificate authority error.

Note: To add a CA, you need to copy the CA certificate that was generated in Part 1 from SecServer to the appropriate location on DockerRunner.

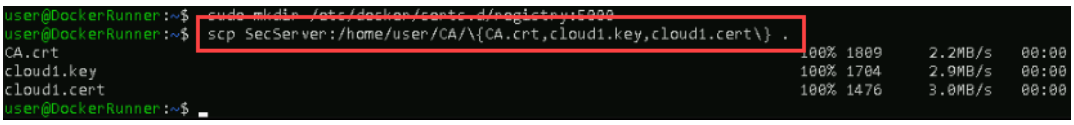
17. In the DockerRunner session, **type** `sudo mkdir /etc/docker/certs.d/registry:5000` and **press Enter** to create a directory to hold the certificate.



sudo mkdir command

Note: The Docker engine allows you to specify certificates on a per registry basis. Each registry can have a folder under /etc/docker/certs.d. Inside that folder, any certificates that end with ".crt" will be trusted as certificate authorities.

18. In the DockerRunner session, **type** `scp SecServer:/home/user/CA/\{CA.crt,cloud1.key,cloud1.cert\} .` and **press Enter** to copy the required files.

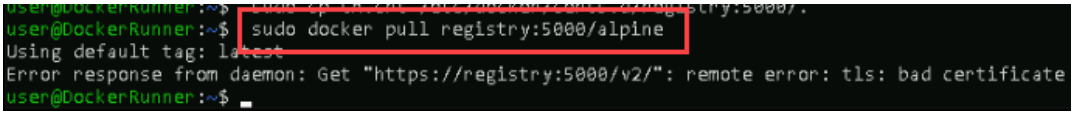


scp command

19. In the DockerRunner session, **type** **`sudo cp CA.crt /etc/docker/certs.d/registry:5000/.`** and **press Enter** to install the CA certificate.

sudo cp command

20. In the DockerRunner session, **type** **`sudo docker pull registry:5000/alpine`** and **press Enter** to attempt to pull an image.



sudo docker pull command

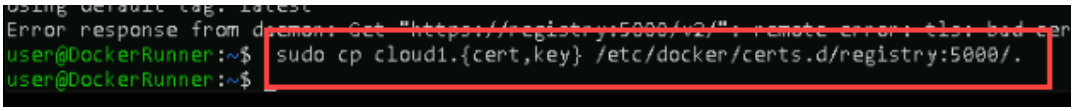
Note: This should still fail, but with a new error. This time it is because you have not yet specified your client certificate.

21. **Make a screen capture** showing the bad certificate error.

Note: Client certificates can also be specified on a per registry basis by installing files in the same folder. For client certificates, the extension should be .cert instead of .crt . To use a client certificate, you need both the key and the certificate.

22. In the DockerRunner session, **type** **`sudo cp cloud1.{cert,key}`**

`/etc/docker/certs.d/registry:5000/.` and **press Enter** to install the certificate and key.



sudo cp command

23. In the DockerRunner session, **type** `sudo docker pull registry:5000/alpine` and **press Enter** to attempt to pull an image.

Note: This attempt should finally succeed. You have established a secure connection between the DockerRunner machine and the registry on DockerServer.

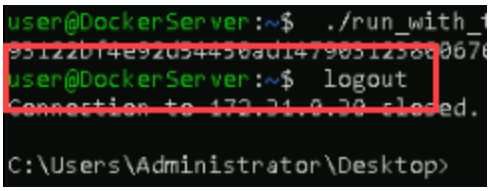
24. **Make a screen capture** showing a successful pull of registry:5000/alpine on DockerRunner.

Note: At this point, you have configured the registry on DockerServer and a client on DockerRunner to use mutual TLS for communication. Both machines rely on the certificate authority identified by the same certificate CA.crt that was generated in Part 1.

- The registry will reject clients without a valid certificate signed by CA.crt.
- The client will connect to registry:5000 only if it has a valid certificate issued by CA.crt.

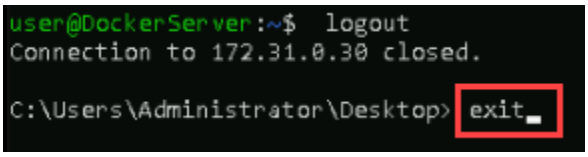
The session with DockerServer is not needed for the next part, so you can end that session.

25. Return to your DockerServer session, **type** `logout` and **press Enter** to end the session.



logout command

26. At the command prompt, **type** `exit` and **press Enter** to close the window.



exit command

Part 3: Using Zero Trust for a Web Service

Note: The example architecture you are using has a frontend server and a backend server. In this lab, there will be a single instance of each running in a container. One of the main reasons to divide a web service into two parts like this is for scaling. The backend server is responsible for generating pages based on user requests. Sometimes that process can be relatively slow. However, it is often possible to cache the results if they do not change frequently.

The frontend server acts as the primary interface for end users. There can be multiple frontend instances that cooperate to perform load balancing and caching of the results from the backend, or origin, server. There can also be multiple backend servers if the number of non-cached requests that must be handled is high enough.

With the traditional security model, the communication between the frontend and backend servers is over a controlled private network. The network is assumed to be trusted. That means that no authentication is needed between the two servers.

In a cloud environment, it is possible to achieve a similar configuration. Most cloud providers give means of creating a virtual private network within their own cloud. However, mixing different cloud providers into the same virtual network is more difficult. By using this method, you are also placing more trust in the cloud providers' network security.

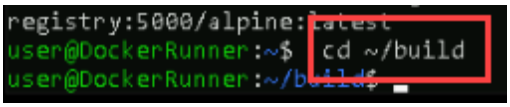
The zero-trust model uses an architecturally simpler configuration. No special networks need to be set up between the frontend and backend servers. Each server simply needs to have an address that is

accessible on the public internet. Because the access is open, all communication needs to be authenticated.

For your testing scenario, your team has created a simple web application to produce a single test page indicating the nature of the connection between the frontend and the backend servers. The frontend server is using nginx. The first step is to build the Docker image for the backend server.

1. If necessary, on the vWorkstation desktop, **double-click** the **command prompt icon**, **type** `ssh DockerRunner` and **press Enter** to start a session on DockerRunner.

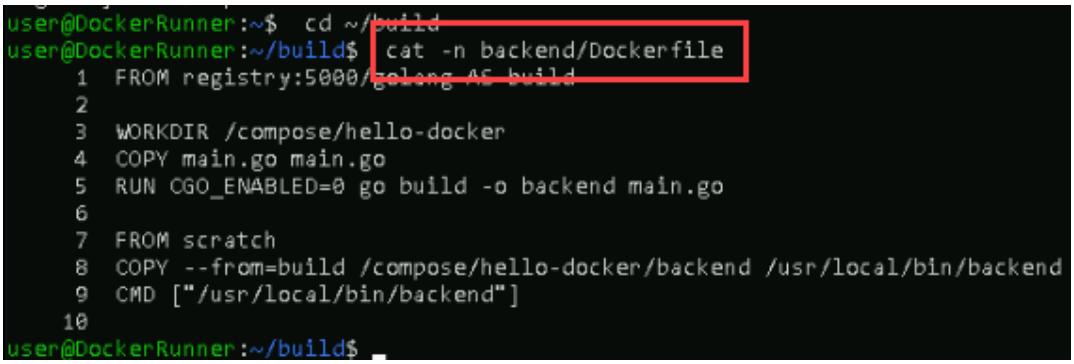
2. In the DockerRunner session, **type** `cd ~/build` and **press Enter** to change the working directory.



cd command

Note: You will use DockerRunner to simulate using a machine to build Docker images for your application and push them to the registry. The folder~/build/backend contains source code and a Dockerfile for a simple web application for testing.

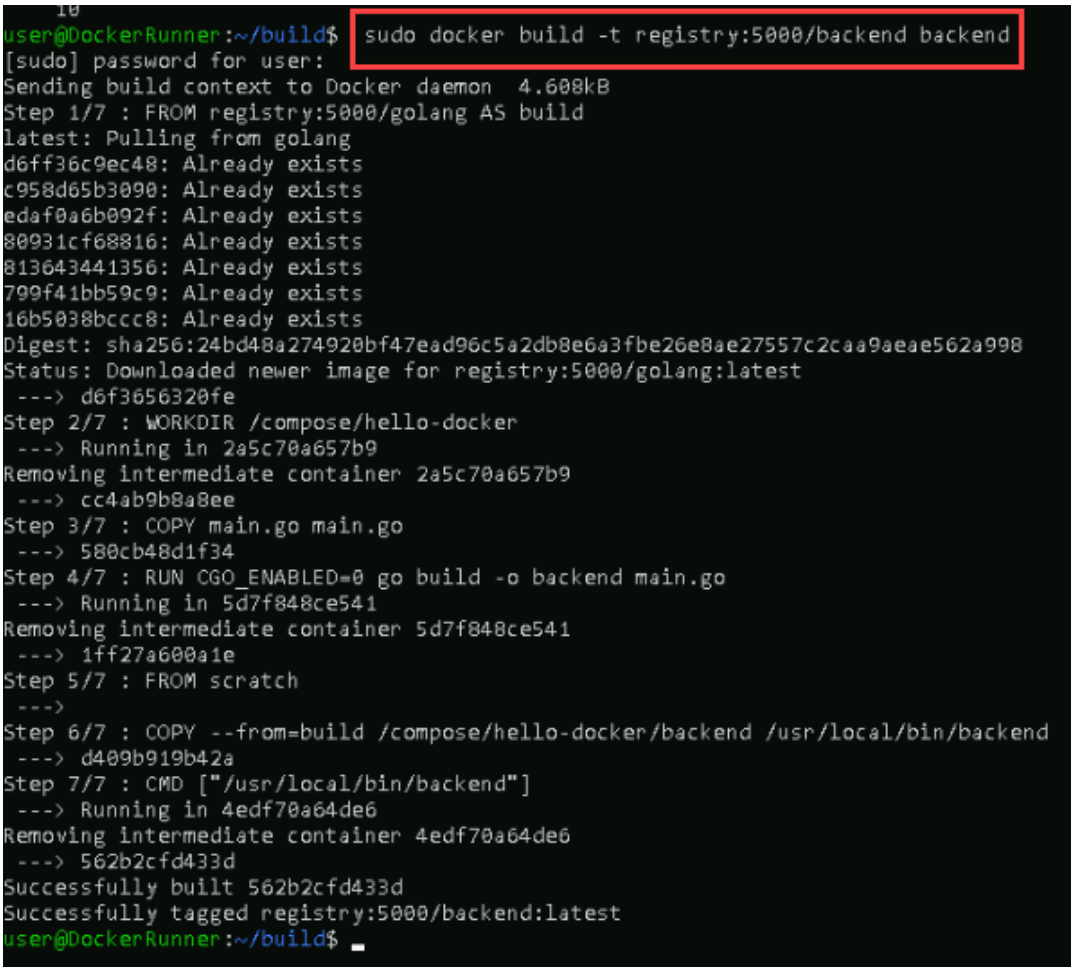
3. In the DockerRunner session, **type** `cat -n backend/Dockerfile` and **press Enter** to view the Dockerfile for the backend.



cat Dockerfile command

Note: Lines 1 through 5 give instructions for building a Go application using the registry:5000/golang Docker image. Lines 7 and 8 create a new image and copy the binary from the build image. Then line 9 specifies the command to run in the final container.

4. In the DockerRunner session, **type** **`sudo docker build -t registry:5000/backend backend`** and **press Enter** to build the backend server image.



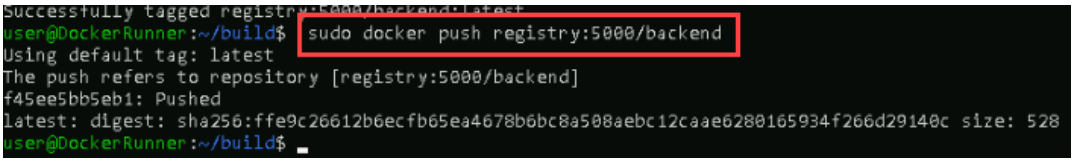
sudo docker build command

Note: The Docker build subcommand can be used to create images using a Dockerfile. The other arguments are:

- **-t registry:5000/backend** sets the tag for the resulting image
- **backend** is the folder containing the Dockerfile

The backend image should be pushed to the local registry to save it for use by other machines as needed.

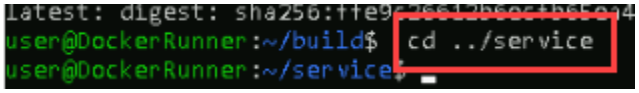
5. In the DockerRunner session, **type** `sudo docker push registry:5000/backend` and **press Enter** to push the backend image.



sudo docker push command

Note: The front end uses the publicly available nginx image with a custom configuration file mounted into the container. Since the configuration can be modified by the commands to create and start a container, no custom image needs to be created.

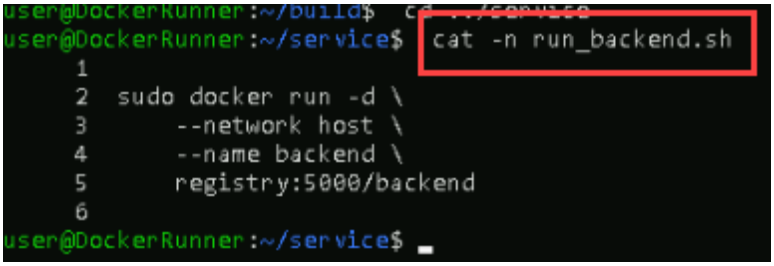
6. In the DockerRunner session, **type** `cd ../service` to change the working directory.



cd command

Note: The next step is to create and run the containers. In the final production system, the frontend and backend containers will not necessarily be running on the same host. The communication would potentially occur over public internet channels. To emulate this, the containers will be run attached to the host network.

7. In the DockerRunner session, **type** `cat -n run_backend.sh` to view the script to launch the backend.



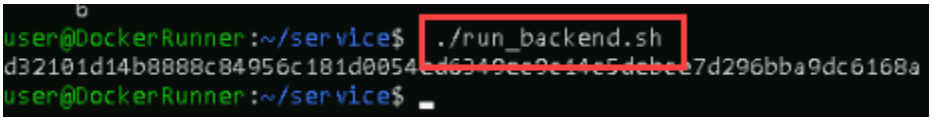
cat run_backend.sh command

Note: This script uses the following arguments for the Docker run subcommand:

- **-d** to run detached
- **--network host** to use host networking
- **--name backend** to set the container name
- **registry:5000/backend** is the image identifier

Use the script to run the backend service.

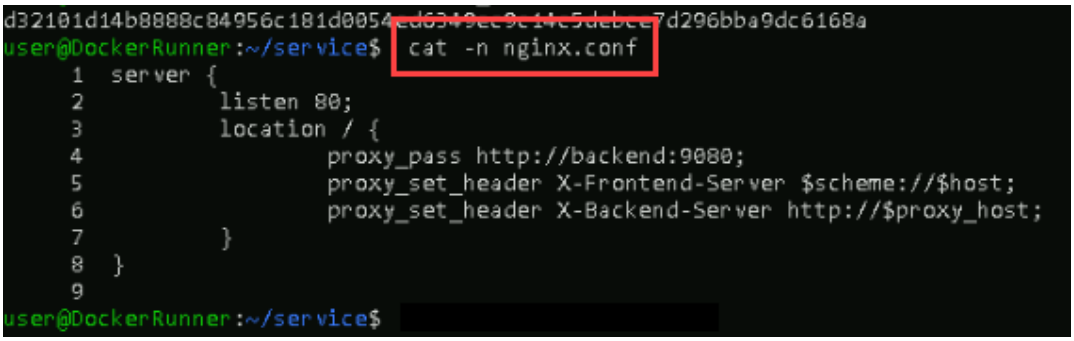
8. In the DockerRunner session, **type** `./run_backend.sh` and **press Enter** to run the backend.



Execute run_backend.sh script

Note: You should see the ID of the newly created container. The frontend service uses a custom configuration for the publicly available nginx Docker image.

9. In the DockerRunner session, **type** `cat -n nginx.conf` and **press Enter** to view the nginx configuration.



cat nginx.conf command

Note: This is a simple nginx configuration to create a reverse proxy. The nginx server listens on port 80 of the container. For any requests, it passes it to the backend server listening at http://backend:9080. Lines 5 and 6 set custom headers in the request from the frontend to the backend that are used by the test application.

10. In the DockerRunner session, **type** `cat -n run_frontend.sh` to view the script to launch the frontend.

```
user@DockerRunner:~/service$ cat -n run_frontend.sh
     1
     2  sudo docker run -d \
     3       --network host \
     4       -v /home/user/service/nginx.conf:/etc/nginx/conf.d/default.conf \
     5       --add-host backend:172.31.0.20 \
     6       --name frontend \
     7       registry:5000/nginx
     8
user@DockerRunner:~/service$
```

cat run_frontend.sh command

Note: The frontend command includes an additional argument **--add-host backend:172.31.0.20**. This argument makes the hostname backend resolve to 172.31.0.20 inside the container. This allows the nginx configuration to use the name backend.

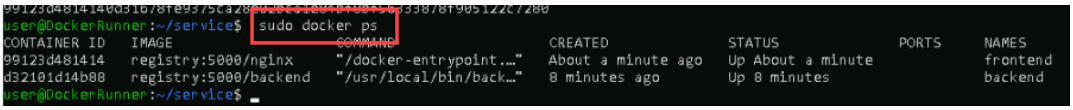
11. In the DockerRunner session, **type** `./run_frontend.sh` and **press Enter** to run the frontend.

```
user@DockerRunner:~/service$ ./run_frontend.sh
Unable to find image 'registry:5000/nginx:latest' locally
latest: Pulling from nginx
2408cc74d12b: Already exists
dd61fcc63eac: Already exists
f9686e628075: Already exists
ceb5504faee7: Already exists
ce5d272a5b4f: Already exists
136e07b65aca: Already exists
Digest: sha256:20a1077e25510e824d6f9ce7af07aa02d86536848ddab3e4ef7d1804608d8125
Status: Downloaded newer image for registry:5000/nginx:latest
99123d4814140d31678fe9375ca28e02bca1e84bf8bf56333878f905122c7280
user@DockerRunner:~/service$
```

Execute run_frontend.sh script

Note: Since the nginx image was not yet on DockerRunner, it first pulls it from DockerServer. Now you should confirm that the containers are actually running.

12. In the DockerRunner session, **type** `sudo docker ps` and **press Enter** to view the current Docker processes**.**



sudo docker ps command

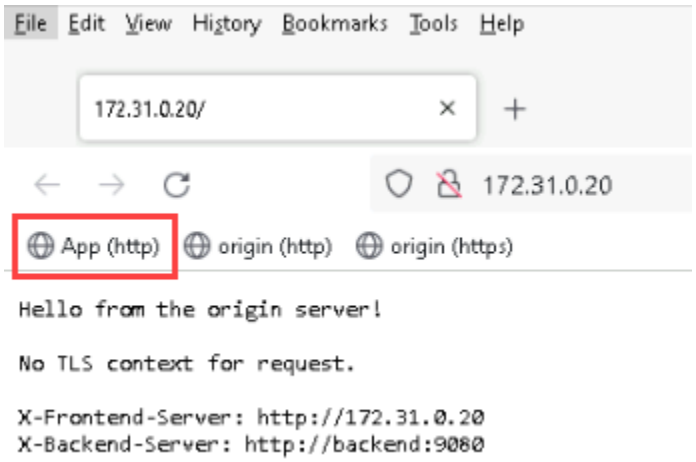
13. **Make a screen capture** showing the running frontend and backend containers.

Note: You should see two containers running. To test that the application is working as expected, you can try to access the page using a web browser.

14. On the vWorkstation desktop, **double-click** the **Firefox icon** to open a web browser.

Firefox icon

15. In the bookmark toolbar, **click** the **App (http) shortcut** to navigate to http://172.31.0.20.

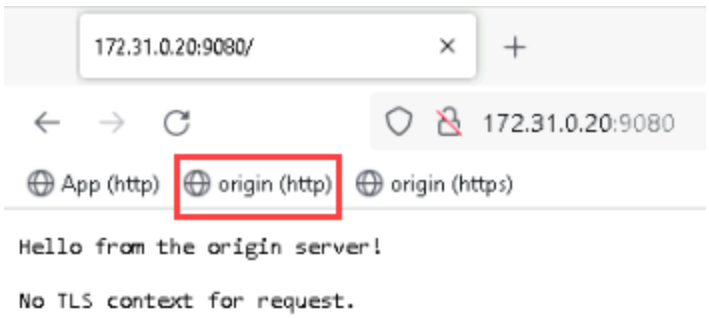


App (http) bookmark and successful page load without TLS

Note: This is the expected path for an end user accessing the application. You should see a page that says "Hello from the origin server!" The page also provides information from the backend server about how the request was received and handled. In this case, there is a line indicating that the frontend to backend request did not use TLS. There are also two lines indicating the custom headers that are added by the frontend nginx configuration. These headers report the scheme and host for both the frontend and backend. You can see that the frontend server is listening on http://172.31.0.20 using the standard port (80). It sends the request to the origin server at http://backend:9080.

16. **Make a screen capture** showing the successful page load from App (http).

17. In the bookmark toolbar, **click** the **origin (http) shortcut** to navigate to http://172.31.0.20:9080.



origin (http) bookmark and successful page load from exposed origin server

Note: You should again see a web page that says “Hello from the origin server!” There is still no TLS being seen by the backend server because the browser used http. Notice that the lines for the custom headers are missing. This is because the frontend server was not involved in the request.

18. **Make a screen capture** showing the successful page load from origin (http).

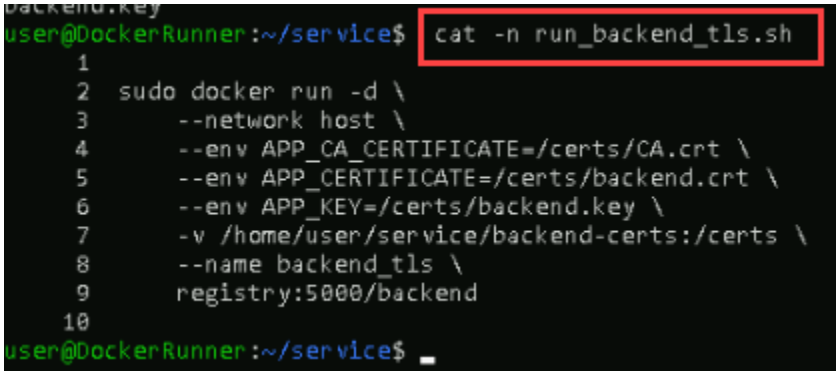
Note: For a production service, end users should not be able to make direct requests to the origin server. While it does not cause any problems with this simple page, in a real application, it could be dangerous. To prevent end user access to the origin server, even when it is visible over the network, you can use mutual TLS. The backend server is written to check for three environment variables to specify the root certificate, server certificate, and server key. You need to copy these from SecServer where they were generated in Part 1 to DockerRunner.

19. **Minimize** the **Firefox browser**.

20. In the DockerRunner session, **type** `scp SecServer:/home/user/CA/\{CA.crt,backend.crt,backend.key\} backend-certs/.` and **press Enter** to copy the required files for the backend.

scp command

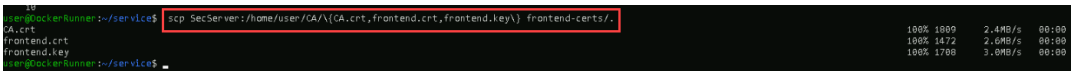
21. In the DockerRunner session, **type** `cat -n run_backend_tls.sh` and **press Enter** to see the command to run the TLS-enabled backend.



cat run_backend_tls.sh command

Note: The command is very similar to the original one, with the addition of several --env VALUE arguments to set environment variables, as well as a volume(-v) to mount the certificates folder into the container. The application is designed to use these variables to find the certificate and key files when running with TLS enabled. Note that the container is given a different name as well.

22. In the DockerRunner session, **type** `scp SecServer:/home/user/CA/\{CA.crt,frontend.crt,frontend.key\} frontend-certs/.` and **press Enter** to copy the required files for the front end.



scp command

23. In the DockerRunner session, **type** `cat -n run_frontend_tls.sh` and **press Enter** to see the command to run the TLS-enabled frontend.

```
user@DockerRunner:~/service$ cat -n run_frontend_tls.sh
     1
     2  sudo docker run -d \
     3       --network host \
     4       -v /home/user/service/nginx_tls.conf:/etc/nginx/conf.d/default.conf \
     5       -v /home/user/service/frontend-certs:/certs \
     6       --add-host backend:172.31.0.20 \
     7       --name frontend_tls \
     8       registry:5000/nginx
     9
user@DockerRunner:~/service$
```

cat run_frontend_tls.sh command

Note: Here, the frontend uses a different configuration file and includes a volume to mount the certificates folder into the container.

24. In the DockerRunner session, **type** `cat -n nginx_tls.conf` and **press Enter** to see the nginx configuration for using TLS.

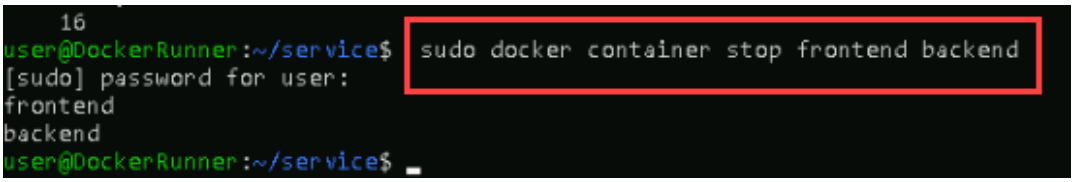
```
     9
user@DockerRunner:~/service$ cat -n nginx_tls.conf
     1  server {
     2          listen 80;
     3          location / {
     4                  proxy_pass https://backend:9443;
     5                  proxy_ssl_certificate         /certs/frontend.crt;
     6                  proxy_ssl_certificate_key     /certs/frontend.key;
     7                  proxy_ssl_protocols           TLSv1.2 TLSv1.3;
     8                  proxy_ssl_ciphers             HIGH:!aNULL:!MD5;
     9                  proxy_ssl_trusted_certificate /certs/CA.crt;
    10                  proxy_ssl_verify              on;
    11                  proxy_ssl_verify_depth        2;
    12                  proxy_set_header X-Frontend-Server $scheme://$host;
    13                  proxy_set_header X-Backend-Server https://$proxy_host;
    14          }
    15  }
    16
user@DockerRunner:~/service$
```

cat nginx_tls.conf command

Note: Note that the nginx server is still listening on port 80. The https connection is from the frontend to the backend, not from the client to the frontend. There are several new proxy_ssl_* settings that

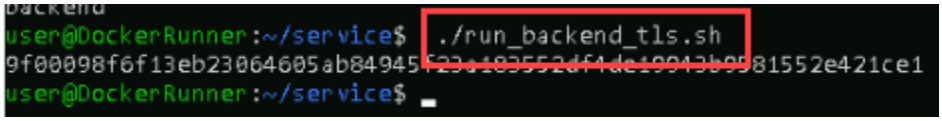
provide nginx with the information needed to set up the TLS connections. The configuration also includes lines 12 and 13 to set the custom headers. Since the same port is used by backend_tls, you need to stop the previous versions from running.

25. In the DockerRunner session, **type** `sudo docker container stop frontend backend` and **press Enter** to stop the containers.



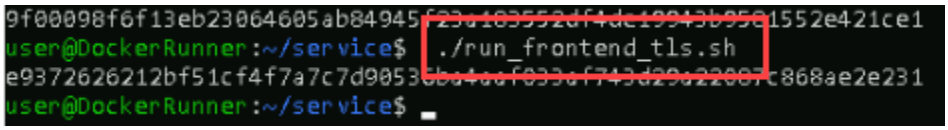
sudo docker container stop command

26. In the DockerRunner session, **type** `./run_backend_tls.sh` and **press Enter** to start the TLS-enabled backend.



Execute run_backend_tls.sh script

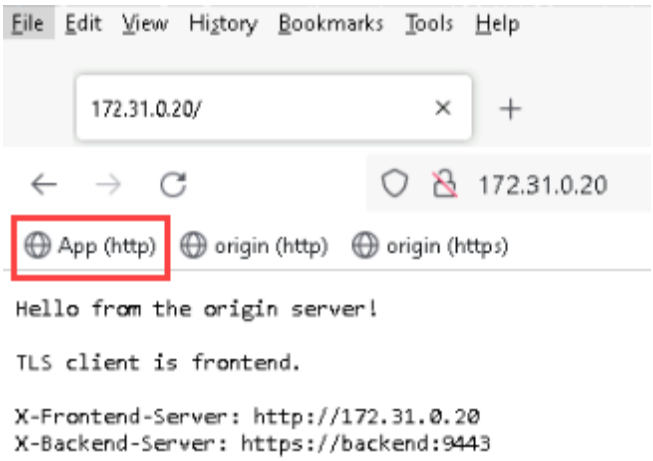
27. In the DockerRunner session, **type** `./run_frontend_tls.sh` and **press Enter** to start the TLS-enabled frontend.



Execute run_frontend_tls.sh script

28. **Restore** the **Firefox window**.

29. In the bookmark toolbar, **click** the **App (http) shortcut** to navigate to http://172.31.0.20.

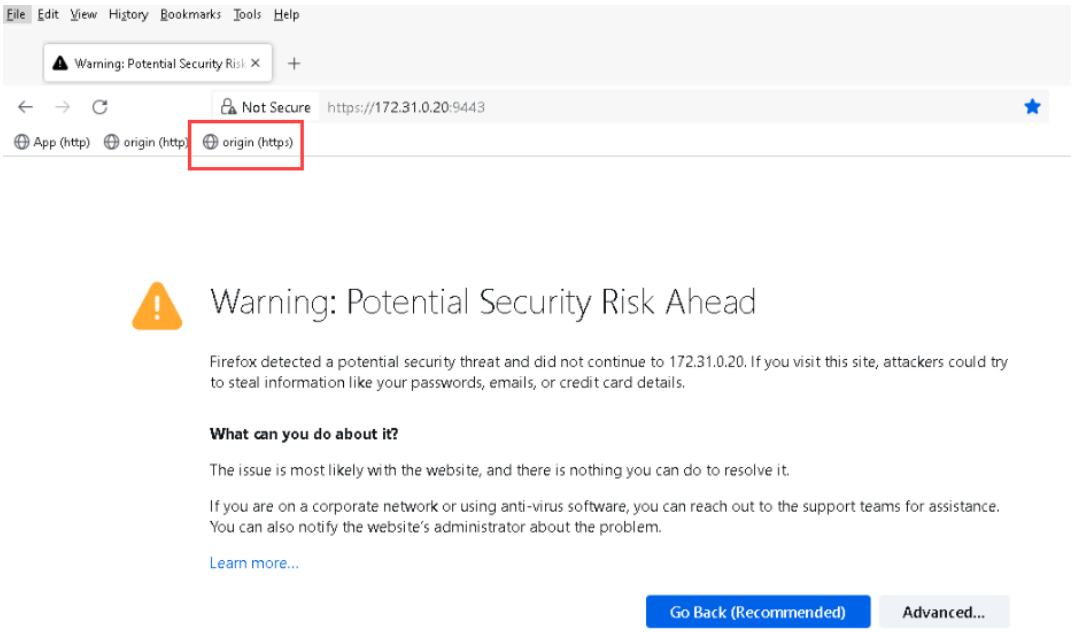


App (http) bookmark and successful page load with frontend TLS client

30. **Make a screen capture** showing the successful page load from App (http) with mutual TLS on.

Note: You should see a successful page load. Now there is a TLS client seen by the backend server. You can also see that the X-Backend-Server indicates that https was used. The final test is to try to talk to the origin server directly.

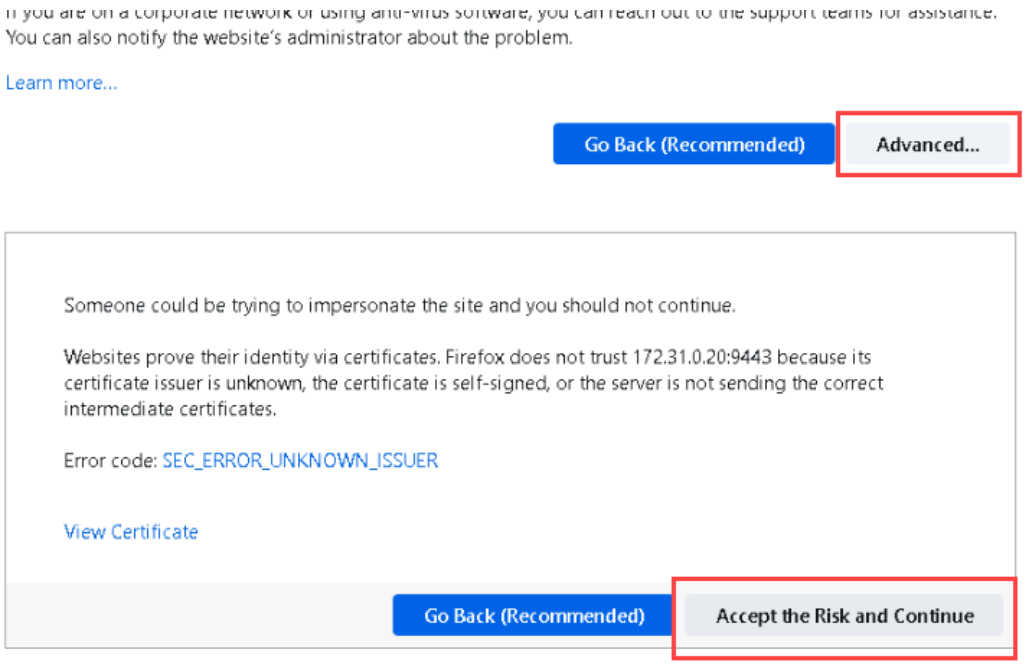
31. In the bookmark toolbar, **click** the **origin (https) shortcut** to navigate to https://172.31.0.20:9443.



origin (https) bookmark and self-signed certificate error

Note: Since the origin server uses a certificate signed by your custom CA, it is not recognized by the browser, so you will see an error page about a self-signed certificate.

32. On the error page, **click Advanced**, **scroll down**, and then **click Accept the Risk and Continue** to load the page.



Advanced Security options in Firefox

Note: Now you can see the result returned by the origin server. The server detected that no client certificate was given, so it refused to provide the regular page.

33. **Make a screen capture** showing the unsuccessful load from origin (https).

Note: If you have been assigned the Challenge & Analysis section, do not reset your lab.

Challenge and Analysis

Note: The following scenario is provided to allow independent, unguided work, similar to what you will encounter in a real situation.

Part 1: Enable Mutual TLS for the Development Machine

During the lab, you built the backend server image on DockerRunner, where the image was also going to be used. In the production environment, you will be using separate machines for building the images and running them. The developer machines will need to be able to talk to the registry with mutual TLS as well.

Use the certificate authority setup on SecServer to create a new client certificate with a distinguished name '/CN=dev'. Configure the Docker engine on SecServer so that it can communicate with the registry on DockerServer.

1. **Make a screen capture** showing at least the first 12 lines of the client certificate

2. **Make a screen capture** showing a successful pull of registry:5000/alpine on SecServer.

Part 2: Check Certificates Manually

As part of your planning for troubleshooting and recovery processes, you need to be able to manually verify a certificate's validity. Research the options for the openssl tool to determine how to verify that a certificate is valid.

1. **Make a screen capture** showing the output of the verification command for the developer certificate.