 Cloud Hacking - Exploit a container stored on AWS' ECR!  
  
🎄 Learn the basics of docker container images  
🎄 Learn about AWS' Elastic Container Registry (ECR)  
🎄 Inspect a container, identify & exploit security issues

Premise

Grinch Enterprises has been gloating about their attack on an underground forum. We know they were specifically targeting organizations in a campaign they've themed "Advent of Cyber" (AOC) - what a frustrating coincidence. Tracing the user back over time - we also encountered a reference to using AWS Elastic Container Registry (ECR) to store container images they use as infrastructure in their attacks. Let's see if we can find out more about the attack tooling Grinch Enterprises is using.

Getting Started

You'll need to start your AttackBox to run commands using the Docker container tool. Containers are a virtualization mechanism similar to Virtual Machines (VMs), and container images are based on the [Open Container Initiative](https://opencontainers.org/) [Distribution Specification](https://github.com/opencontainers/distribution-spec/blob/main/spec.md). However, when someone talks about "Docker" or "containers", they often are talking about multiple container technologies that work together. Specifically, the term "Docker" is used to describe:

* Docker API - a local communication interface on a configured Linux machine, with standardized commands used to communicate with a Docker Daemon.
* Docker Daemon - a process that runs on your machine (the Docker daemon), to interact with container components such as images, data volumes, and other container artifacts.
* Docker Container Image Format - ultimately a .tar file. For Version 1, the docker image format was not strictly compliant with the OCI [Image Specification](https://github.com/opencontainers/image-spec). For our purposes, this won't change how we interact with container images in this exercise, but it does slightly change the format and content of a container image.

Now that we've gotten some of the basic terminologies out of the way, let's get started with today's learning objective.

Today's Learning Objective - AWS Elastic Container Registry - ECR Public Gallery

Today we'll be covering the basics of container images and AWS Elastic Container Registry ([ECR](https://aws.amazon.com/ecr/)) - an online registry for public and private container images. We'll learn about how to retrieve a container image from an online registry and inspect the elements of that container image to identify potential security issues.

Docker Images and Amazon Elastic Container Registry

In a cloud-native computing environment, containers are a first-choice solution for deploying infrastructure. Similar to virtual machines, containers serve as the compute fabric for many running applications and hosted processes in the cloud.

Once you've logged on to an AttackBox, you can run the following command to see the container images that are stored by default on your AttackBox:

docker images

which should return an output similar to the following:

Docker Images

root@ip-10-10-20-249:~**#** docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

remnux/ciphey latest ec11b47184f6 9 months ago 177MB

rustscan/rustscan 2.0.0 6890f34e17b0 12 months ago 41.6MB

bcsecurity/empire v3.5.2 cbd0b10f7f55 13 months ago 2.05GB

mpepping/cyberchef latest 36979d2c2b9e 17 months ago 639MB

root@ip-10-10-20-249:~#

Docker containers are stored in "repositories", which are a reference to file mappings the Docker daemon knows how to reach, which include the container .tar files. Each image in a repository will include an image tag, and images can be referenced using either their tag or Image ID.

For example:

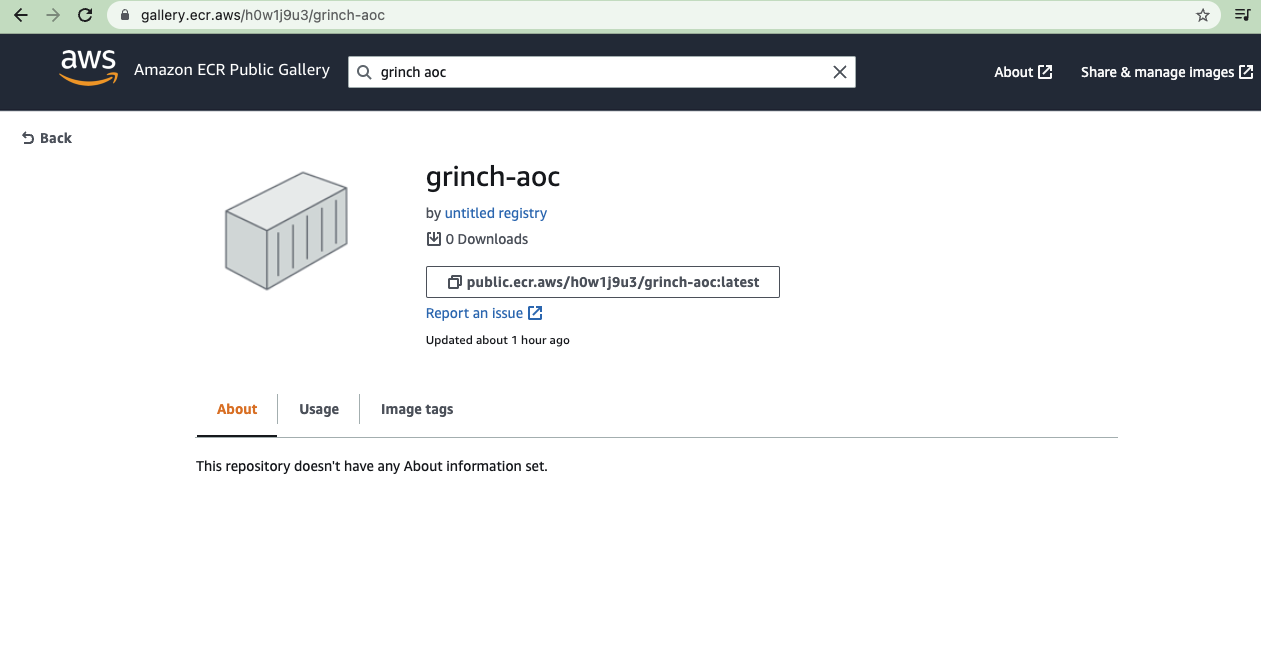
remnux/ciphy:latest

or

ec11b47184f6

Grinch Enterprise Attack Infrastructure

We've traced the Grinch Enterprises attack infrastructure back to a likely Elastic Container Registry that is publicly accessible:



[Link to Suspected AWS Public Container Gallery for Grinch Enterprises](https://gallery.ecr.aws/h0w1j9u3/grinch-aoc)

You can retrieve the potential Grinch Enterprises image by running the following command on your AttackBox:

docker pull public.ecr.aws/h0w1j9u3/grinch-aoc:latest

which returns will return an output similar to the following:

Docker Pull

root@ip-10-10-20-249:~**#** docker pull public.ecr.aws/h0w1j9u3/grinch-aoc:latest

latest: Pulling from h0w1j9u3/grinch-aoc

7b1a6ab2e44d: Pull complete

7181c3c4941b: Pull complete

148b30b9ae2d: Pull complete

6f5a7c388565: Pull complete

ef099323cb4a: Pull complete

de5bf7e2abf0: Pull complete

455d5424d859: Pull complete

b1ee65a7e02a: Pull complete

a47021107475: Pull complete

Digest: sha256:593c79eaaa1a905c533e389b0034022e074969da3936df648172c4efc8d421d8

Status: Downloaded newer image for public.ecr.aws/h0w1j9u3/grinch-aoc:latest

public.ecr.aws/h0w1j9u3/grinch-aoc:latest

root@ip-10-10-20-249:~#

You can run the container and interact with it by running the following command:

docker run -it public.ecr.aws/h0w1j9u3/grinch-aoc:latest

which will open a shell inside the container image, as indicated by the $. Once inside the container, we can do a little reconnaissance:

ls -la

which shows there are no regular files or subdirectories in the present working directory.

Docker Run

root@ip-10-10-20-249:~**#** docker run -it public.ecr.aws/h0w1j9u3/grinch-aoc:latest

**$** ls -la

total 20

drwxr-xr-x 2 newuser newuser 4096 Oct 21 20:31 .

drwxr-xr-x 1 root root 4096 Oct 21 20:31 ..

-rw-r--r-- 1 newuser newuser 220 Feb 25 2020 .bash\_logout

-rw-r--r-- 1 newuser newuser 3771 Feb 25 2020 .bashrc

-rw-r--r-- 1 newuser newuser 807 Feb 25 2020 .profile

$

Spoiler

A good place to check next is environment variables - in Linux and especially for containers, environment variables may be used to store secrets or other sensitive information used to configure the container at run-time.

So we try printenv to learn more about the environment configurations where we see:

printenv

**$** printenv

HOSTNAME=c633e29bb404

HOME=/home/newuser

TERM=xterm

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

api\_key=a90eac086fd049ab9a08374f65d1e977

PWD=/home/newuser

$

and we have stumbled on an api\_key which I'm guessing Grinch Enterprises didn't intend to leave behind.

Bonus Challenge

A container image is composed of a number of layers - perhaps there is sensitive information in an underlying container layer that Grinch Enterprises didn't clean up as part of their build process. One key reason that developers (and attackers) might package their application (or attack tools) in a container is that a container allows a developer to "freeze" an application and its dependencies into an image as part of the build process. The build process is part of the Software Development Lifecycle (SDLC), where applications and their dependencies are packaged together and tested prior to distribution and use.

Once an image is built, running the container image will always result in the same configuration state as specified at build-time. Container images are built from a source file known as aDockerfile. Dockerfiles are a list of new-line separated instructions that instruct the Docker daemon how to generate a container image. You can read an exhaustive explanation of how to write Dockerfiles in the [Dockerfile reference](https://docs.docker.com/engine/reference/builder/). You can see an example of a Dockerfile [here](https://docs.docker.com/samples/apt-cacher-ng/)>. In the case of Grinch Enterprises, we don't have the original Dockerfile - but with the container image, we have something just as good. Let's start by creating a new directory and saving the downloaded image as a .tar file.

1. Create a new directory: mkdir aoc

2. Change directory to the newly created directory:cd aoc

3. Save the container image as a .tar file: docker save -o aoc.tar public.ecr.aws/h0w1j9u3/grinch-aoc:latest

Docker Save

**$** exit

root@ip-10-10-20-249:~**#** mkdir aoc

root@ip-10-10-20-249:~**#** cd aoc/

root@ip-10-10-20-249:~/aoc**#** docker save -o aoc.tar public.ecr.aws/h0w1j9u3/grinch-aoc:latest

root@ip-10-10-20-249:~/aoc#

Once we have saved the image, we can further inspect the image by unpacking the compressed file

tar -xf aoc.tar

Note that I used the -v (verbose) option when I performed the command, and you can see the various files that are being unpacked:

tar -xvf

root@ip-10-10-20-249:~/aoc**#** tar -xvf aoc.tar

40ad0e404f6065a153d1b4d42e8b315be3504a08c21fadd6e5fde5982b45df18/

40ad0e404f6065a153d1b4d42e8b315be3504a08c21fadd6e5fde5982b45df18/VERSION

40ad0e404f6065a153d1b4d42e8b315be3504a08c21fadd6e5fde5982b45df18/json

40ad0e404f6065a153d1b4d42e8b315be3504a08c21fadd6e5fde5982b45df18/layer.tar

4416e55edf1a706527e19102949972f4a8d89bbe2a45f917565ee9f3b08b7682/

4416e55edf1a706527e19102949972f4a8d89bbe2a45f917565ee9f3b08b7682/VERSION

4416e55edf1a706527e19102949972f4a8d89bbe2a45f917565ee9f3b08b7682/json

4416e55edf1a706527e19102949972f4a8d89bbe2a45f917565ee9f3b08b7682/layer.tar

4cc7bdb0ea56d31f57a373d0e7ce0d633ae86dc327087fccf103c8d97f0cc9c4/

4cc7bdb0ea56d31f57a373d0e7ce0d633ae86dc327087fccf103c8d97f0cc9c4/VERSION

4cc7bdb0ea56d31f57a373d0e7ce0d633ae86dc327087fccf103c8d97f0cc9c4/json

4cc7bdb0ea56d31f57a373d0e7ce0d633ae86dc327087fccf103c8d97f0cc9c4/layer.tar

4f62ae56d8d3b96d5fbe86da8a3f7bf6e9195d360b922cd7b162e17619c50664/

4f62ae56d8d3b96d5fbe86da8a3f7bf6e9195d360b922cd7b162e17619c50664/VERSION

4f62ae56d8d3b96d5fbe86da8a3f7bf6e9195d360b922cd7b162e17619c50664/json

4f62ae56d8d3b96d5fbe86da8a3f7bf6e9195d360b922cd7b162e17619c50664/layer.tar

619ddb982b75f0eb6c9f48624e6a0d20be227e893599d8dea05dbdddc8b14e2b/

619ddb982b75f0eb6c9f48624e6a0d20be227e893599d8dea05dbdddc8b14e2b/VERSION

619ddb982b75f0eb6c9f48624e6a0d20be227e893599d8dea05dbdddc8b14e2b/json

619ddb982b75f0eb6c9f48624e6a0d20be227e893599d8dea05dbdddc8b14e2b/layer.tar

9dedacd92213db743681db2e8d5b3247fd79ce266495d061a381c4c0441ce15d/

9dedacd92213db743681db2e8d5b3247fd79ce266495d061a381c4c0441ce15d/VERSION

9dedacd92213db743681db2e8d5b3247fd79ce266495d061a381c4c0441ce15d/json

9dedacd92213db743681db2e8d5b3247fd79ce266495d061a381c4c0441ce15d/layer.tar

a3c1e603ab4385e0b411423e70314651bb371561c45a2bc90951fa05da9ad3c4/

a3c1e603ab4385e0b411423e70314651bb371561c45a2bc90951fa05da9ad3c4/VERSION

a3c1e603ab4385e0b411423e70314651bb371561c45a2bc90951fa05da9ad3c4/json

a3c1e603ab4385e0b411423e70314651bb371561c45a2bc90951fa05da9ad3c4/layer.tar

aa7f7d1cdeacc3a446e297814a6c13a42006dc8a99baad72c0c50383d69ac551/

aa7f7d1cdeacc3a446e297814a6c13a42006dc8a99baad72c0c50383d69ac551/VERSION

aa7f7d1cdeacc3a446e297814a6c13a42006dc8a99baad72c0c50383d69ac551/json

aa7f7d1cdeacc3a446e297814a6c13a42006dc8a99baad72c0c50383d69ac551/layer.tar

f886f00520700e2ddd74a14856fcc07a360c819b4cea8cee8be83d4de01e9787.json

fa28cd504eaba5e76b168c5149551371fbeb3bc0f51d18485fe401a411c2dd17/

fa28cd504eaba5e76b168c5149551371fbeb3bc0f51d18485fe401a411c2dd17/VERSION

fa28cd504eaba5e76b168c5149551371fbeb3bc0f51d18485fe401a411c2dd17/json

fa28cd504eaba5e76b168c5149551371fbeb3bc0f51d18485fe401a411c2dd17/layer.tar

manifest.json

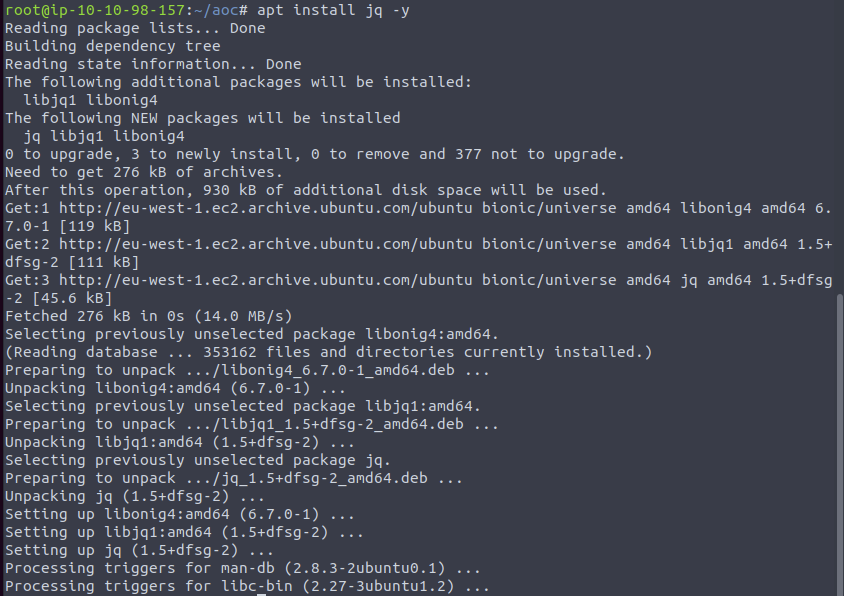
repositories

root@ip-10-10-20-249:~/aoc#

These files represent the various container image layers, with the exception of the manifest.json file. manifest.json represents the "manifest" of container image layers that compose the final container image we were just inside. Let's take a look at this image using a tool called "jq" to "pretty-print" the output for easier readability:

Note: On an attack box, jq is now pre-installed and you can skip this step

1. Install jq: apt install jq -y



2. Print the contents of manifest.json to the terminal using jq to pretty-print: cat manifest.json | jq

cat manifest.json

root@ip-10-10-20-249:~/aoc**#** cat manifest.json | jq

[

{

"Config": "f886f00520700e2ddd74a14856fcc07a360c819b4cea8cee8be83d4de01e9787.json",

"RepoTags": [

"public.ecr.aws/h0w1j9u3/grinch-aoc:latest"

],

"Layers": [

"a3c1e603ab4385e0b411423e70314651bb371561c45a2bc90951fa05da9ad3c4/layer.tar",

"619ddb982b75f0eb6c9f48624e6a0d20be227e893599d8dea05dbdddc8b14e2b/layer.tar",

"40ad0e404f6065a153d1b4d42e8b315be3504a08c21fadd6e5fde5982b45df18/layer.tar",

"aa7f7d1cdeacc3a446e297814a6c13a42006dc8a99baad72c0c50383d69ac551/layer.tar",

"4f62ae56d8d3b96d5fbe86da8a3f7bf6e9195d360b922cd7b162e17619c50664/layer.tar",

"9dedacd92213db743681db2e8d5b3247fd79ce266495d061a381c4c0441ce15d/layer.tar",

"fa28cd504eaba5e76b168c5149551371fbeb3bc0f51d18485fe401a411c2dd17/layer.tar",

"4416e55edf1a706527e19102949972f4a8d89bbe2a45f917565ee9f3b08b7682/layer.tar",

"4cc7bdb0ea56d31f57a373d0e7ce0d633ae86dc327087fccf103c8d97f0cc9c4/layer.tar"

]

}

]

root@ip-10-10-20-249:~/aoc#

Note the first piece of information in the file is "Config", which represents the underlying configurations and commands used to build the container image -

f886f00520700e2ddd74a14856fcc07a36c819b4cea8cee8be83d4de01e9787.json

This configuration file is also located in the root of the unpacked container image directory:

ls

root@ip-10-10-20-249:~/aoc**#** ls

40ad0e404f6065a153d1b4d42e8b315be3504a08c21fadd6e5fde5982b45df18

4416e55edf1a706527e19102949972f4a8d89bbe2a45f917565ee9f3b08b7682

4cc7bdb0ea56d31f57a373d0e7ce0d633ae86dc327087fccf103c8d97f0cc9c4

4f62ae56d8d3b96d5fbe86da8a3f7bf6e9195d360b922cd7b162e17619c50664

619ddb982b75f0eb6c9f48624e6a0d20be227e893599d8dea05dbdddc8b14e2b

9dedacd92213db743681db2e8d5b3247fd79ce266495d061a381c4c0441ce15d

a3c1e603ab4385e0b411423e70314651bb371561c45a2bc90951fa05da9ad3c4

aa7f7d1cdeacc3a446e297814a6c13a42006dc8a99baad72c0c50383d69ac551

aoc.tar

f886f00520700e2ddd74a14856fcc07a360c819b4cea8cee8be83d4de01e9787.json

fa28cd504eaba5e76b168c5149551371fbeb3bc0f51d18485fe401a411c2dd17

manifest.json

repositories

root@ip-10-10-20-249:~/aoc#

and can be inspected in the same manner as manifest.json:

cat f886f00520700e2ddd74a14856fcc07a36c819b4cea8cee8be83d4de01e9787.json | jq

cat .json Config part 1

root@ip-10-10-20-249:~/aoc**#** cat f886f00520700e2ddd74a14856fcc07a360c819b4cea8cee8be83d4de01e9787.json | jq

{

"architecture": "amd64",

"config": {

"Hostname": "",

"Domainname": "",

"User": "newuser",

"AttachStdin": false,

"AttachStdout": false,

"AttachStderr": false,

"Tty": false,

"OpenStdin": false,

"StdinOnce": false,

"Env": [

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",

"api\_key=a90eac086fd049ab9a08374f65d1e977"

],

"Cmd": null,

"Image": "sha256:035522c2043f6036e879810cfffe0db9665ebb09e1852339231fd805daad5325",

"Volumes": null,

"WorkingDir": "/home/newuser",

"Entrypoint": [

"sh"

],

"OnBuild": null,

"Labels": null

},

"container": "7b422a5dd0a2a59167ae476fcc18f7ae9a094c02de40b4b4effd42a5d032bae4",

"container\_config": {

"Hostname": "7b422a5dd0a2",

"Domainname": "",

"User": "newuser",

"AttachStdin": false,

The first section of the manifest configuration file walks through the final image configuration as intended to run on a container host system. However, this next section is of particular interest to an attacker - here's how the container image was built. You can see each section broken up by curly braces, and some of the sections have an extra line indicating "empty\_layer": true.

cat .json Config part 2

"created": "2021-10-21T20:31:17.236366166Z",

"docker\_version": "20.10.7",

"history": [

{

"created": "2021-10-16T00:37:47.226745473Z",

"created\_by": "/bin/sh -c #(nop) ADD file:5d68d27cc15a80653c93d3a0b262a28112d47a46326ff5fc2dfbf7fa3b9a0ce8 in / "

},

{

"created": "2021-10-16T00:37:47.578710012Z",

"created\_by": "/bin/sh -c #(nop) CMD [\"bash\"]",

"empty\_layer": true

},

{

"created": "2021-10-20T16:16:12.499990187Z",

"created\_by": "/bin/sh -c apt-get upgrade && apt-get update"

},

{

"created": "2021-10-20T16:16:46.080121757Z",

"created\_by": "/bin/sh -c apt install curl -y"

},

{

"created": "2021-10-21T20:22:41.837170259Z",

"created\_by": "/bin/sh -c apt install python3 -y"

},

{

"created": "2021-10-21T20:23:42.130217528Z",

"created\_by": "/bin/sh -c apt install pip -y"

},

{

"created": "2021-10-21T20:23:52.8316757Z",

"created\_by": "/bin/sh -c apt install git -y"

},

{

"created": "2021-10-21T20:31:13.639594181Z",

"created\_by": "/bin/sh -c git clone https://github.com/hashicorp/envconsul.git root/envconsul/"

},

{

"created": "2021-10-21T20:31:14.315738313Z",

"created\_by": "/bin/sh -c #(nop) WORKDIR /root/envconsul",

"empty\_layer": true

},

{

"created": "2021-10-21T20:31:14.645450256Z",

"created\_by": "/bin/sh -c #(nop) ADD file:cba528c0d7ba7c0c89ad4ce3e550dc4b3128c2804d4dc75daaf1421759f6d664 in . "

},

{

"created": "2021-10-21T20:31:15.914695012Z",

Each of these sections are describing a particular command run by the Docker daemon at the time that the image was built, and if the "empty\_layer": true configuration is not listed as part of the section definition, then the container layer is retained in the overall container image as one of the layers listed in the manifest.json file. Of particular interest - we notice the container containers a tool called envconsul that is pulled from Github. Reviewing the Github repository for [envconsul](https://www.github.com/hashicorp/envconsul) - the about description states envconsul is a tool that allows a user to "Launch a subprocess with environment variables using data from @hashicorp Consul and Vault." Noting that this source code was cloned into a root directory - perhaps there is something sensitive related to envconsul that a regular container user isn't intended to see at Grinch Enterprises. Let's dig through the image layers and see if we can find out what is so sensitive about envconsul.

We can more closely inspect the layers by switching to the sub-directories representing the layers in the unpacked container root directory. As we switch between these layers, we notice one layer of special interest:

cat config.hcl

root@ip-10-10-20-249:~/aoc**#** cd 4416e55edf1a706527e19102949972f4a8d89bbe2a45f917565ee9f3b08b7682/

root@ip-10-10-20-249:~/aoc/4416e55edf1a706527e19102949972f4a8d89bbe2a45f917565ee9f3b08b7682**#** tar -xvf layer.tar

root/

root/envconsul/

root/envconsul/config.hcl

root@ip-10-10-20-249:~/aoc/4416e55edf1a706527e19102949972f4a8d89bbe2a45f917565ee9f3b08b7682**#** cat root/envconsul/config.hcl

**#** This denotes the start of the configuration section for Consul. All values

**#** contained in this section pertain to Consul.

consul {

​

**#** This is the address of the Consul agent. By default, this is

**#** 127.0.0.1:8500, which is the default bind and port for a local Consul

**#** agent. It is not recommended that you communicate directly with a Consul

**#** server, and instead communicate with the local Consul agent. There are many

**#** reasons for this, most importantly the Consul agent is able to multiplex

**#** connections to the Consul server and reduce the number of open HTTP

**#** connections. Additionally, it provides a "well-known" IP address for which

**#** clients can connect.

address = "127.0.0.1:8500"

​

**#** This controls the retry behavior when an error is returned from Consul.

**#** Envconsul is highly fault tolerant, meaning it does not exit in the face

**#** of failure. Instead, it uses exponential back-off and retry functions

**#** to wait for the cluster to become available, as is customary in distributed

Spoiler

This layer contains a config.hcl file - as we look at this file in the container image layer - it is clear that sensitive configurations are maintained in the file. Let's use Linux command-line tool grepand see if we can return a "secret" or a "token"...and there it is on line 4 when grepping with the string 'token'. I wonder if the Grinch Enterprise developers knew that the container image cached all of the container layers? Either way, now we can turn the tables on Grinch Enterprises and access their Vault cluster with all its secrets!

grep token

root@ip-10-10-20-249:~/aoc/4416e55edf1a706527e19102949972f4a8d89bbe2a45f917565ee9f3b08b7682**#** cat root/envconsul/config.hcl | grep 'token'

**#** This is the token to use when communicating with the Vault server.

**#** assumption that you provide it with a Vault token; it does not have the

**#** incorporated logic to generate tokens via Vault's auth methods.

token = "TOKEN"

# This tells Envconsul to load the Vault token from the contents of a file.

# - by default Envconsul will not try to renew the Vault token, if you want it

# to renew you will need to specify renew\_token = true as below.

# - Envconsul will periodically stat the file and update the token if it has

# vault\_agent\_token\_file = "/path/to/vault/agent/token/file"

# This tells Envconsul that the provided token is actually a wrapped

# token that should be unwrapped using Vault's cubbyhole response wrapping

unwrap\_token = true

**#** This option tells Envconsul to automatically renew the Vault token given.

**#** If you are unfamiliar with Vault's architecture, Vault requires tokens be

**#** automatically renew the token at half the lease duration of the token. The

**#** you want to renew the Vault token using an out-of-band process.

**#** There is an exception to the default such that if vault\_agent\_token\_file is

**#** set, either from the command line or the above option, renew\_token defaults

**#** token itself.

renew\_token = true

root@ip-10-10-20-249:~/aoc/4416e55edf1a706527e19102949972f4a8d89bbe2a45f917565ee9f3b08b7682#

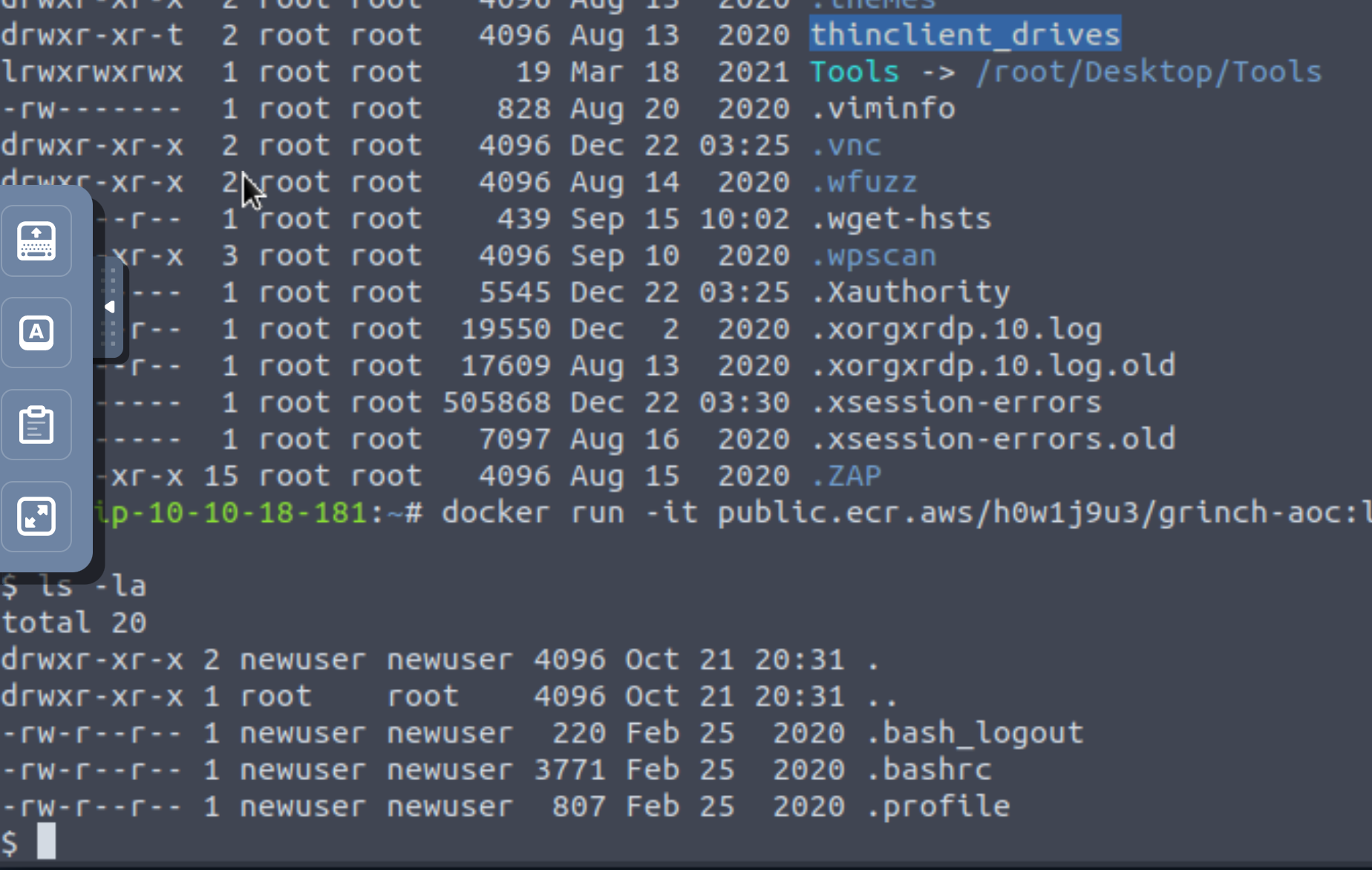
***Answer the questions below***

What command will list container images stored in your local container registry?

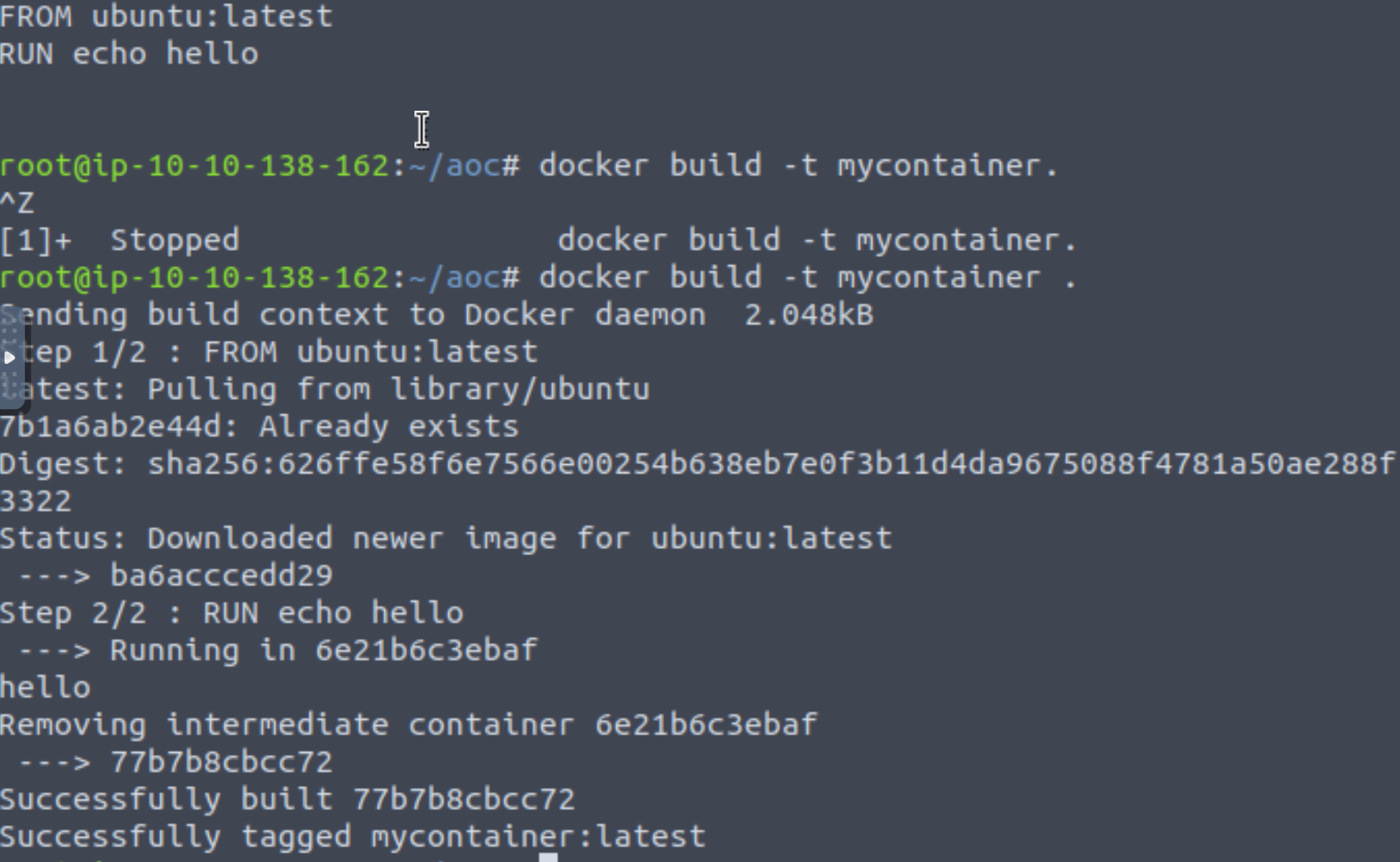


 Submit

What command will allow you to save a docker image as a tar archive?

What is the name of the file (including file extension) for the configuration, repository tags, and layer hash values stored in a container image?

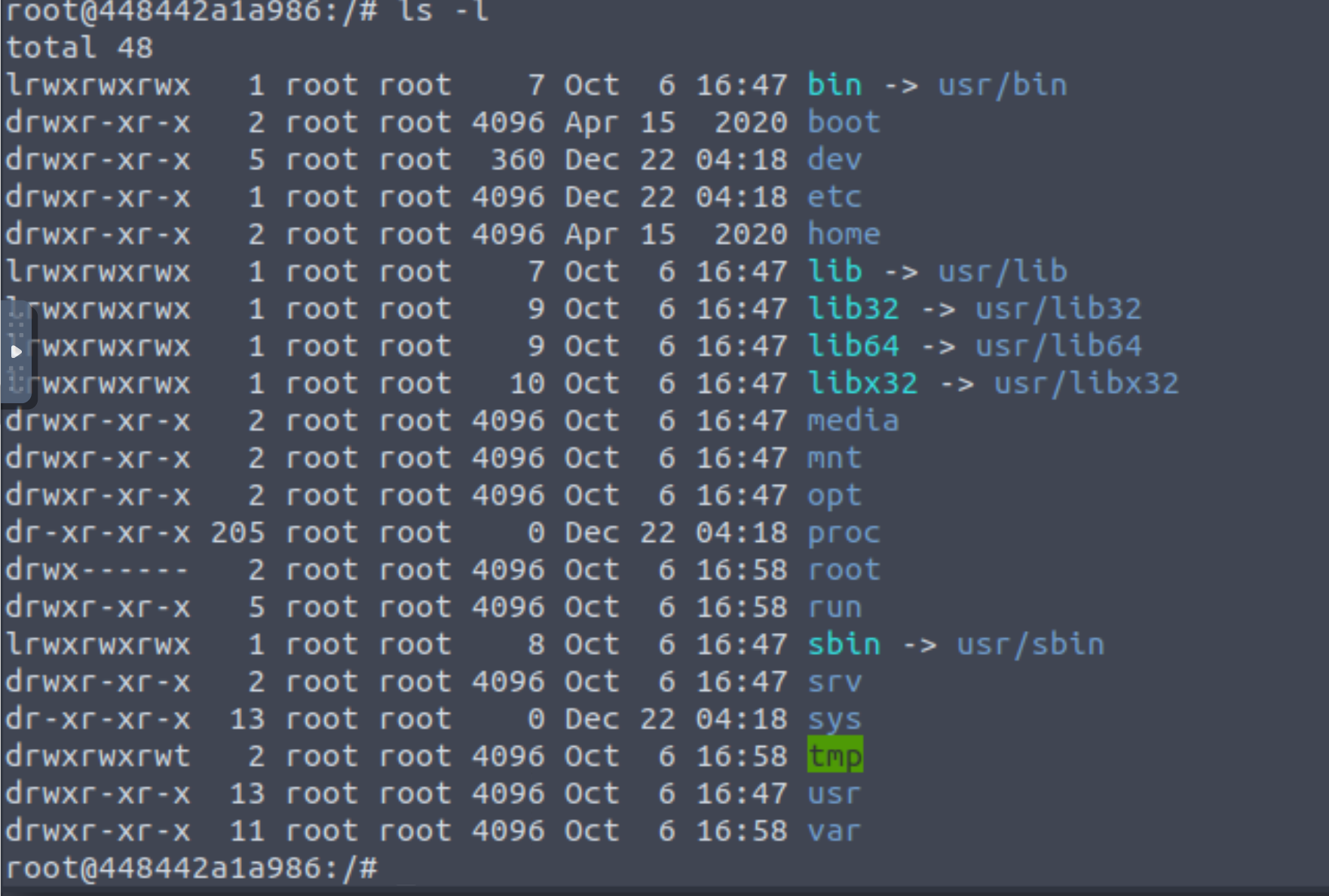
Build dockerfile:- thereby running



api\_key=a90eac086fd049ab9a08374f65d1e977

Docker run -it mycontainer

There are layer to docker container gone with build dockerfile



The layer of container can seen using tar xvf

What is the token value you found for the bonus challenge?

Jq the config file you see in manifest file showing api, image ,user , potential commands or list of previous commands for the container . particularly interesting in envconsol github showing readme showing use case and find tar layer file into config.hcl

