32C3 CTF: Docker writeup

Jan 2, 2016 • By saelo

docker was a pwnable worth 250 points during 32C3 CTF 2015. The goal was to escape from a (slightly non-standard) docker container configuration.

Here's the scenario:

We are given ssh access to a box (`ssh://eve@136.243.194.40`) as user "eve". On the box we see the following:

```
eve@docker:/$ ls -lh /home/*/*
-rwsr-xr-x 1 root root 8.4K Dec 27 13:40 /home/adam/dockerrun
-r-------- 1 adam root   53 Dec 27 14:32 /home/adam/flag
```

So the goal is pretty clear: read `/home/adam/flag` which is only possible as the "adam" user.

The dockerrun binary executes the following command line as root:

```
/usr/bin/docker run -it --privileged=false -u 1337:65534 --cap-
drop=ALL --net=host ubuntu /bin/bash
```

Which results in a `/bin/bash` being opened in a docker container running as the "adam" user (uid 1337). The docker container lives inside its own mount namespace. We cannot see the "real" `/home/adam` (`/` is a new mountpoint) and thus cannot access the flag.

Ok.

Exploitation

There is one particularly interesting command line argument to docker: `--net=host`. From the documentation:

> *–net=host — Tells Docker to skip placing the container inside of a separate network stack. In essence, this choice tells Docker to not containerize the container's networking! While container processes will still be confined to their own filesystem and process list and resource limits, a quick ip addr command will show you that, network-wise, they live "outside" in the main Docker host and have full access to its network interfaces. Note that this does not let the container reconfigure the host network stack — that would require –privileged=true — but it does let container processes open low-numbered ports like any other root process. It also allows the container to access local network services like D-bus. This can lead to processes in the container being able to do unexpected things like restart your computer. You should use this option with caution.*

The key information here is that `--net=host` allows the container access to networking services on the host!

At this point one needs knowlege of two slightly advanced linux features:

- Abstrack Unix Socket: While "normal" Unix sockets are bound to the filesystem (and thus inaccessible from inside the container), an "abstract" Unix socket lives in its own namespace and thus *is* accessible from within the container (due to `--net=host`).
- File descriptors over Unix sockets: Using this, it is possible to send file descriptors over a Unix socket from one process to another.

See where this is going?

Here's the plan:

1. Open an abstract Unix socket and bind it to some arbitrary name
2. Start the docker container
3. Inside the docker container, connect to the previously created Unix socket
4. From outside, open `/home/adam` (since we cannot directly open `/home/adam/flag`) and send it via the Unix socket to the process inside the container
5. Inside the container we can now `fchdir(received_fd)` then `system("cat flag")` (or use openat()) and read the flag

To do that I wrote some C code for the server (running outside the container) and the client (running inside the container). I uploaded the source code for the server (since gcc was available), then uploaded the base64 encoded client binary (since no gcc was available) inside the container:

```
eve@docker:~$ cd /tmp
eve@docker:/tmp$ cat << EOF > server.c
<source code here>
EOF
eve@docker:/tmp$ gcc -o server server.c
eve@docker:/tmp$ ./server &
eve@docker:/tmp$ /home/adam/dockerrun
I have no name!@docker1404:/$ cd /tmp
I have no name!@docker1404:/tmp$ cat << EOF | base64 -d > client
<base64 encoded client binary here>
EOF
I have no name!@docker1404:/tmp$ chmod +x client
I have no name!@docker1404:/tmp$ ./client
THANK YOU MARIO!

BUT OUR FLAG IS IN
ANOTHER CHROOT!
```

Hm ok, almost. So apparently we also need to break out of a chroot. Let's do the following: Instead of just reading the flag after doing fchdir() on the received file descriptor we will instead spawn a bash process and see how things look like. We will also send a file descriptor for `/` instead of `/home/adam` this time. Doing this results in this slightly amusing bash prompt:

```
I have no name!@docker1404:/tmp$ ./client
I have no name!@docker1404:(unreachable)/chroot$ cd ..
I have no name!@docker1404:(unreachable)/$ ls
bin  boot  chroot  dev  etc  flag  home  initrd.img
initrd.img.old  lib  lib64  lost+found  media  mnt  nsjail  opt
proc  root  run  sbin  srv  sys  tmp  usr  var  vmlinuz
vmlinuz.old
I have no name!@docker1404:(unreachable)/$ cat flag
32C3_its_even_e4s1er_as_ro0t
```

voilà :)

So why did this work? Inside the docker container our root directory is the mount point created by docker (something like `/dev/mapper/docker-8:1-930171-95805d296c80d42164fbe2fb43c6af74e3e894825187831c176cf1918ff3512a on /`). Once we have done fchdir() to the outside root directory we are already outside of our own root directory and can thus move around freely in the filesystem. This is similar to the classic chroot escape which chroot()s itself to a subdirectory and chdir()s to a location outside the new root (but inside the old).

Note that this is however no universal chroot escape: In our case we needed root privileges inside the chroot (the dockerrun binary) to create a new mount namespace. On the other hand, if we try to create a new user namespace, then perform the chroot escape inside of it (where we are root) the kernel will block us (kernel/user_namespace.c):

```
/*
 * Verify that we can not violate the policy of which files
 * may be accessed that is specified by the root directory,
 * by verifing that the root directory is at the root of the
 * mount namespace which allows all files to be accessed.
 */
if (current_chrooted())
    return -EPERM;
```

The code for the server and client can be found in our github repository.

show Disqus comments

KITCTF -|- visit us on github twitter