Akaljed's Notes

Various tech stuff

Scapy As Webserver

with 3 comments

This is about a scriptable webserver based on Scapy that represents a hybrid of a Wireshark equivalent (with Deep Packet Inspection) and a server. The standard webserver (Apache) doesn't give full control over the communication flow and doesn't allow the user to intervene or to capture the communication stream. Scapy can do both.

Introduction

On Linux, the most easiest way to serve clients from the Internet with a html or binary file is using *netcat* (*nc*). Netcat is able to do almost every kind of simple network operation and is very robust.

In order to serve the client with data, an exemplary *index.html* is needed. Also, a *favico.ico* would be nice, especially as *Mozilla Firefox browser* always asks for this file, regardless of whether there is a favico.ico given in the html or not.

Assuming that our index.html and favico.gif are in */home/user1/* and given that the index.html contains *text only*, we create a file '*server.sh*' with following content:

```
sudo nc -vnlp 80 -q 0 < /home/user1/index.html
sudo nc -vnlp 80 -q 0 < /home/user1/favico.ico
```

Do not forget to make the file executable! *192.168.1.1* is assumed to be the IP address of the Linux server in this example. Start the terminal and type *"sudo ./server.sh"* on the Linux server. If you are connected within a LAN, start Firefox on another connected system (client) and type "http://192.168.1.1” in the address field of Firefox.

By pressing strg-c, the server can be stopped within a second. Instead of http port 80, another port can be specified. High ports don't need a sudo prefix. The client (using a standard browser) then needs to additionally specify this port instead of using default port 80. (Example: 192.168.1.1:2222) Fortunately, netcat automatically handles the http protocol and creates the correct http header for a corresponding file.

Very easy isn't it? Of course it is also possible to specify an image instead of sending a html. The image will appear in the client's browser.

Note that the connecting user will receive the *same file* every time, regardless of what he typed in the address field of his browser after the server IP (or DNS). By enhancing the shell script and implementing new functions, more intelligence can be added (where needed). However, we won't discuss any shell scripting here.

Scapy

A well-known multi-purpose tool for networking materia is *Scapy*. Scapy is a python-based interface, which can do almost everything, regardless of the complexity. It is defined by its extreme flexibility. Scapy works in kernel mode (ring-0), so you need to be root.

Abilities of Scapy:

- Deeply understands the OSI Layer
- Can read any traffic passing the network card (just as Wireshark)
- Can create, modify and play with raw binary network packets.
- Can create new layers at any layer (for example a packet with three tcp layers, which of course doesn't make sense at all. Be careful with this and don't crash the target system.)
- The value of each single field of any Layer can be specified (or modified in an existing packet).
- Uses python (either interactive or as python script) and
- ..therefore is scriptable.
- Can be programmed to behave as a webserver.

As listed, Scapy is able to print data from every packet and can be programmed to *dynamically react on the arrival of specific data*. The user can precisely specify the values he's interested into or simply print all data of a packet.
Here's a mini example of how to capture three TCP packets with destination port 80, while only interested in the IP address of the remote sender and our own TCP port.

```
----8<-------------------------------------

#!/usr/bin/python
from scapy.all import *
# Akaljed Dec 2010, http://www.akaljed.wordpress.com

# Capture & print 3 TCP packets with dport 80
# The 'sourceIPs' variable is a buffer with 5 network packets.
# A network packet contains all layers, not just TCP.
sourceIPs = sniff(filter="tcp and port 80",count=3,prn=lambda x:x.sprintf("{IP:%IP.src%: %TCP.dport%}"))

print "  "
if sourceIPs[0].dport==80: print "specify action A"
if sourceIPs[1].dport==80: print "specify action B"
if sourceIPs[2].dport==80: print "specify action C"
# You get the idea. Instead of printing text, specify other actions.

----8<-------------------------------------
```

Here another example for capturing icmp packets. It may be useful to show DDos-attacks against the system.

```
----8<-------------------------------------

#!/usr/bin/python
from scapy.all import *
# Akaljed Dec 2010, http://www.akaljed.wordpress.com

print "   "
print "Listening for next 5 ICMP messages... (Press strg-c to abort)"
print "   "

# Capture 5 ICMP packets and print the sender's IP address and the ICMP data, and the ICMP type (echo request for example)
# The ICMP data can be used to identify the senders OS.
# Or sometimes, the sender might be trying to send a secret message to you. ;)

sniff(filter="icmp and dst 192.168.1.1",count=5,prn=lambda x:x.sprintf("{IP: From %IP.src%: %ICMP.load% (%ICMP.type%)}"))

----8<-------------------------------------
```

Before executing Scapy scripts, it is necessary to *disable the Linux Kernels own responses*. If the Linux Kernel is allowed to answer arriving network packets, it will answer with RST-ACK, because the Linux Kernel believes that no port is open. Scapy does operate *besides* the Kernel (Paracommunication), *therefore the Kernel has no knowledge of what Scapy is doing*.

For TCP, disable the kernel's response with:

```
sudo iptables -A OUTPUT -p tcp --tcp-flags RST RST --sport 80 -j DROP
```

This will precisely kill only the *RST*-flagged TCP packets the Linux Kernel tries to send and leave other Kernel responses undamaged (ping will still work). As probably known, *iptables* instructions are not remembered after reboot. If not needed anymore, the instructions can be flushed immediately with the *-F* command. There is probably nothing that Scapy can't do. Let's make Scapy to behave as a server. The minimum conversation of a client connecting to a server is presented by:

- Syn (client)
- Syn-Ack (server)
- Ack (client)
- Read http request and send a index.html to client.
- Rst-Ack (Server)

This presents a full TCP conversation. More details about the *three-way handshake* (1. Syn, 2. Syn-Ack, 3. Ack) and the *TCP protocol* are found at the official source: the RFC 793, but also at Wikipedia. The following explanations need intermediate or advanced network knowledge.

The three-way handshake (RFC 793)

The communication between the connecting client and the webserver starts with the three-way handshake. Several variables are needed in the following, which will hold the different field values. For testing purposes, the communication can be done within a private LAN (strongly recommended). The server has the IP address of 192.168.1.1 and the client has the IP address of 192.168.1.2 (same as above):

Network setup for testing the Scapy Webserver

We begin the Python/Scapy script as usual:

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

The script waits for an incoming connection that concerns our IP address, i.e. *192.168.1.1*. It is possible to specify multiple IP addresses on Linux, that's why it might be intelligent to check for the correct IP address as seen below. We also filter for TCP traffic. We are not concerned about other protocols.

```
# Wait for the SYN of the client
a=sniff(count=1,filter="tcp and host 192.168.1.1 and port 80")

# Initializing some variables for later use.
ValueOfPort=a[0].sport
SeqNr=a[0].seq
AckNr=a[0].seq+1 # We are Syn-Acking, so this must be +1
```

Note that on Linux, Sequence numbers are relative, not absolute.

As we received the connecting SYN request of the client, we need to answer with a correct SYN-ACK to finish our part of the TCP diplomacy:

```
# Generating the IP layer:
ip=IP(src="192.168.1.1", dst="192.168.1.2")

# Generating TCP layer: src port 80, dest port of client,
# flags SA means "Syn-Ack", the AckNr ist +1, and the MSS shall be a default 1460.
TCP_SYNACK=TCP(sport=80, dport=ValueOfPort, flags="SA", seq=SeqNr, ack=AckNr, options=[('MSS', 1460)])

#send SYNACK to remote host AND receive ACK
ANSWER=sr1(ip/TCP_SYNACK)

# Capture next TCP packets with dport 80. (contains http GET request)
GEThttp = sniff(filter="tcp and port 80",count=1,prn=lambda x:x.sprintf("{IP:%IP.src%: %TCP.dport%}"))

# Updating the sequence number as well as the Ack number
AckNr=AckNr+len(GEThttp[0].load)
SeqNr=a[0].seq+1
```

By using the *sr1* function (alias *"send-and-receive-one-time"*), we already received the responding ACK of the client (data of client stored in ANSWER). Thus, client and server are connected with each other now. The 'real' communication can now be initiated from *any* side, both sides are equal. In this sense, the term "server" just means 'the system to which the client connects' whereas the term "client" means 'the system which initiates the connection'.

Any further communication must be inserted here. For a true interactive server, the article would become much too long. The correct steps for sending a html file are:

1. Wait and filter for the http GET request.
2. Interpret the http GET request (which html file does the client want?)
3. Generate a custom packet with http OK header + actual http data.
4. Send it and wait for the client's reply (via sr1 function).

To keep the code short, we simply assume that the client indeed sent a GET http request for receiving the index.html. Therefore, we only need to generate the http-header with the correct content length and append the 'index.html' data.

As this is an example how to use Scapy, we use this opportunity to print some data of the client. Note that any fields can be printed: the ethernet mac-address, the ttl, the FLAGS, the Sequence numbers, *just everything*. It is also possible to change the value of any field at any time. In our example, the most interesting data is probably the kind of browser our client is using and some other request-related data, but feel free to experiment with other fields.

```
# Print the GET request of the client (contains browser data and similar data).
# (Sanity check: size of data should be greater than 1.)
if len(GEThttp[0].load)>1: print GEThttp[0].load

# Generate custom http file content.
html1="HTTP/1.1 200 OK\x0d\x0aDate: Wed, 29 Sep 2010 20:19:05 GMT\x0d\x0aServer: Testserver\x0d\x0aConnection: Keep-Alive\x0d\x

# Generate TCP layer
data1=TCP(sport=80, dport=ValueOfPort, flags="PA", seq=SeqNr, ack=AckNr, options=[('MSS', 1460)])

# Construct whole network packet, send it and fetch the returning ack.
ackdata1=sr1(ip/data1/html1)

# Store new sequence number.
SeqNr=ackdata1.ack
```

At this point, we sent the html file to the client.

The next thing we will do is closing the connection. We use a *RST-ACK*. Not exactly the most diplomatic way, but it makes sure the client doesn't misunderstand us. When using FIN-ACK, many systems take their time to finally respond with their own FIN-ACK. As we are *impatient*, we want to have the connection closed *immediately*.

```
# Generate RST-ACK packet
Bye=TCP(sport=80, dport=ValueOfPort, flags="RA", seq=SeqNr, ack=AckNr, options=[('MSS', 1460)])

send(ip/Bye)

# the end.
```

Other than the *sr*-function above, '*send*' just sends the data and doesn't capture the answer. It is the right choice when the answer is irrelevant (which is the case here).

Last words and the whole Scapy Webserver script

Having full control generally means that *everything* must be done manually, the Linux Kernel can't help. Also, Scapy is *very slow* in comparison to compiled c++ code. However, in the standard situation, netcat should be enough: it is simple and robust. Only Entropy knows why somebody would ever want true control over his communication flow. 🙂

Finally, here is the whole Python/Scapy code ("*server.py*"):
(Be sure to observe the communication via Wireshark. There may be some unexpected communication that disrupt this example code).

```
----8<------------------------------------

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

# Interacts with a client by going through the three-way handshake.
# Shuts down the connection immediately after the connection has been established.
# Akaljed Dec 2010, http://www.akaljed.wordpress.com

# Wait for client to connect.
a=sniff(count=1,filter="tcp and host 192.168.1.1 and port 80")

# some variables for later use.
ValueOfPort=a[0].sport
SeqNr=a[0].seq
AckNr=a[0].seq+1

# Generating the IP layer:
ip=IP(src="192.168.1.1", dst="192.168.1.2")
# Generating TCP layer:
TCP_SYNACK=TCP(sport=80, dport=ValueOfPort, flags="SA", seq=SeqNr, ack=AckNr, options=[('MSS', 1460)])

#send SYNACK to remote host AND receive ACK.
ANSWER=sr1(ip/TCP_SYNACK)

# Capture next TCP packets with dport 80. (contains http GET request)
GEThttp = sniff(filter="tcp and port 80",count=1,prn=lambda x:x.sprintf("{IP:%IP.src%: %TCP.dport%}"))
AckNr=AckNr+len(GEThttp[0].load)
SeqNr=a[0].seq+1

# Print the GET request
# (Sanity check: size of data should be greater than 1.)
if len(GEThttp[0].load)>1: print GEThttp[0].load

# Generate custom http file content.
html1="HTTP/1.1 200 OK\x0d\x0aDate: Wed, 29 Sep 2010 20:19:05 GMT\x0d\x0aServer: Testserver\x0d\x0aConnection: Keep-Alive\x0d\x

# Generate TCP data
data1=TCP(sport=80, dport=ValueOfPort, flags="PA", seq=SeqNr, ack=AckNr, options=[('MSS', 1460)])

# Construct whole network packet, send it and fetch the returning ack.
ackdata1=sr1(ip/data1/html1)
# Store new sequence number.
SeqNr=ackdata1.ack

# Generate RST-ACK packet
Bye=TCP(sport=80, dport=ValueOfPort, flags="FA", seq=SeqNr, ack=AckNr, options=[('MSS', 1460)])

send(ip/Bye)

# The End

----8<------------------------------------
```

Have fun & happy experiments. 😉
—

Written by akaljed

December 12, 2010 at 6:05 pm

Posted in Deep Packet Inspection, Linux, Networking, Scapy, Tips & tricks, Uncategorized

Tagged with Deep Packet Inspection, Networking, Scapy

3 Responses

Subscribe to comments with RSS.

Newsletter Issue #576…

I found your entry interesting thus I've added a Trackback to it on my weblog :)…

The Tech Night Owl Newsletter — Cutting-Edge Tech Commentary

December 13, 2010 at 11:06 am

Reply

some missing here
after client send ack dan psh-ack with GET
the server should return ack then follow by response from server…

d

March 4, 2013 at 10:49 am

Reply

Scapy is much too slow to be used in real context. Slow enough that the client might sometimes retransmit packets which will break the fixed deterministic communication. So, only interesting for learning, experimenting and testing crazy ideas.

akaljed

April 1, 2013 at 8:07 pm