Het manipuleren van netwerkpakketten met Python/Scapy

Een introductie

Thomas Verweij, november 2017

Inhoud

Inleiding

Het beheersen en begrijpen van netwerktechnologie zijn belangrijke onderdelen voor ethical hackers en andere IT-professionals die met het netwerk te maken hebben. Vele tools zijn er dan ook geschreven om netwerken mee te scannen, te monitoren, te verbinden, op te bouwen en aan te vallen. Ze richten zich vaak op een specifiek doel, zoals het scannen van open poorten of het monitoren van *arp* verkeer, waardoor het aantal tools in de gereedschapskist, en daarmee de vereiste kennis, flink kan toenemen. Dit wordt een probleem als het IT'ers ervan weerhoudt om te blijven leren. Programmeur Philippe Biondi heeft een tool ontwikkeld die dit probleem tracht te omzeilen; een universele tool genaamd Scapy.

Deze tekst heeft als hoofddoel een introductie te geven van de implementatie en toepassingen van deze tool. In het eerste hoofdstuk wordt uiteengezet wat Scapy is en op welke manier Scapy zich onderscheidt van andere tools. Daarna, in hoofdstuk twee, wordt er aan de hand van voorbeelden gefocust op de basisfunctionaliteiten. Er wordt uitgelegd hoe Scapy geïnstalleerd kan worden en op welke manieren Scapy gestart kan worden. Vervolgens wordt getoond hoe netwerkpakketten kunnen worden gemanipuleerd, verzonden en opgevangen. In hoofdstuk drie wordt aan de hand van een wat uitgebreider voorbeeld -een door mij geschreven wachtwoord-sniffer- duidelijk hoe Scapy kan worden ingezet bij een concrete toepassing.

1. Wat is Scapy?

Scapy is een zeer krachtige tool die gemaakt is om netwerkpakketten te manipuleren. Dit houdt in dat deze met Scapy gemaakt, verzonden, ontvangen en ontleed kunnen worden. Scapy is geschreven in Python en is in te zetten als module in Python scripts. Ook is er de mogelijkheid om met een interpreter te werken, een extensie van de Python interpreter. De mogelijkheden zijn hierdoor zeer uitgebreid. De fantasie van de programmeur is de limiet.

Klassieke netwerktaken die met Scapy uitgevoerd kunnen worden zijn bijvoorbeeld tracerouting, scanning, probing, 802.11 frame-injectie en natuurlijk combinaties hiervan. Scapy heeft dus de potentie inzetbaar te zijn als vervanger voor tools als hping, arpspoof, wireshark, p0f, nmap, tcpdump, enzovoort. Deze tools zijn vaak gemaakt voor een specifiek doel. Echter, een hacker of netwerkbeheerder bevindt zich vaak in een situatie waar hij een doel voor ogen heeft waar nog geen tool voor gemaakt is. Met Scapy heeft de programmeur een tool om op een lager niveau netwerktaken te verrichten zonder dat er extreem veel code voor nodig is.
Het doel van Scapy is minder specifiek dan eerder genoemde en richt zich op pakketten in het algemeen. Scapy interpreteert dus geen data, wat bij tools voor specifieke doeleinden vaak wel het geval is. Dit is een voordeel omdat veel tools die interpreteren soms een foutieve interpretatie doen. Scapy volgt de denkwijze dat data-interpretatie een taak van de gebruiker is. Als je met Scapy bijvoorbeeld een ‘TCP Reset’ ontvangt, dan is het aan de gebruiker om te bepalen of dit betekent dat een netwerkpoort gesloten is. Het gebruik van Scapy zelf is niet ingewikkeld. De kracht van scapy komt pas tot zijn recht in combinatie met de netwerkkennis die de gebruiker bezit. Zonder die kennis is het alsof je een hamer geeft aan een kind en dan verwacht dat hij meubelmaker is.
Er zijn dus goede redenen om Scapy te gebruiken. Scapy maakt een redelijk aantal onafhankelijke tools overbodig en vereist dat de gebruiker zich verdiept in netwerkpakketten en protocollen.

2. Werking

2.1 Installatie

Scapy is oorspronkelijk geschreven in Python 2.x. Voor Python 3.x compatibiliteit is er een fork beschikbaar genaamd Scapy3k[1]. Scapy is beschikbaar voor Windows, Linux en macOS. Elk van deze platforms heeft ondersteuning voor pip, Python's package manager. Dit is dan ook de beste tool om Scapy mee te installeren.

2.1.1 Pip

Scapy:

```
# pip install scapy
```

Scapy3k:

```
# pip3 install scapy-python3
```

Afhankelijk van het platform gebruikt men pip of pip3 als package manager voor Python 3.x.

2.1.2 Debian-gebaseerd

Scapy:

```
# apt install python-scapy
```

Scapy3k:

```
# apt install python3-scapy
```

2.1.3 Arch

Scapy:

```
# pacman -S scapy
```

Scapy3k:

```
# pacman -S scapy3k
```

[1] https://github.com/phaethon/scapy

2.2 Scapy starten

Scapy is op twee manieren te gebruiken:

1. via een interpreter
2. als module in Python-scripts

Wanneer men Scapy in interpreter-modus wilt opstarten gebruikt men **sudo** om Scapy root-rechten te geven. Zonder root-rechten kan Scapy geen pakketten verzenden of ontvangen.

```
$ sudo scapy
WARNING: No route found for IPv6 destination :: (no default route?). This affects only IPv6
INFO: Please, report issues to https://github.com/phaethon/scapy
WARNING: IPython not available. Using standard Python shell instead.
Welcome to Scapy (3.0.0)
>>>
```

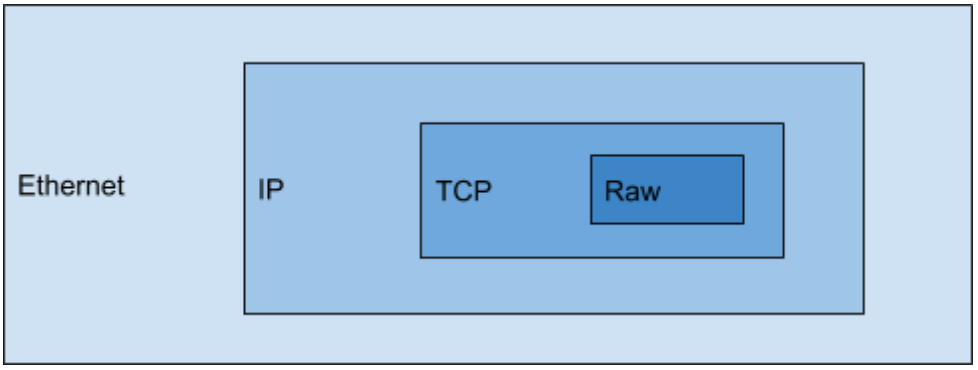
Na wat waarschuwingen (die men voor algemeen gebruik kan negeren) opent er een interpreter en kan men aan de slag.

Om Scapy in een script te gebruiken importeert men de module als volgt:

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

2.3 Pakketten

Scapy heeft de mogelijkheid om complete netwerkpakketten laag voor laag op te bouwen. Hierbij wordt gewerkt volgens het OSI-model. Er wordt gebruik gemaakt van lagen zodat de onderdelen van pakketten los van elkaar zijn te definiëren en op te bouwen. Elke laag is de *payload* van een laag van hoger niveau:



In het volgende voorbeeld wordt er een pakket-object opgebouwd en toegekend aan de variabele *pkt*, vervolgens wordt het resultaat samengevat weergegeven*:*

```
>>> pkt = IP()/TCP()/"GET / HTTP/1.0\r\n\r\n"
>>> pkt
<IP  frag=0 proto=tcp |<TCP  |<Raw  load='GET / HTTP/1.0\r\n\r\n' |>>>
```

Er is hier een pakket uit drie lagen opgebouwd:

1. IP()
2. TCP()
3. "GET / HTTP/1.0\r\n\r\n"

De lagen worden samen tot een pakket gevormd door middel van een "/".
Er is in dit pakket sprake van raw data (3) (dit kan simpelweg als string worden gedefinieerd) dat wordt ingekapseld in een tcp (2) segment wat vervolgens weer wordt ingekapseld in een IP pakket (1).

Wanneer men dit pakket nog wilt inkapselen in een layer-2 frame, dan doet men dit als volgt:

```
>>> pkt2 = Ether()/pkt
>>> pkt2
<Ether  type=0x800 |<IP  frag=0 proto=tcp |<TCP  |<Raw  load='GET / HTTP/1.0\r\n\r\n' |>>>>
```

Zoals hierboven weergegeven vult Scapy automatisch bepaalde waarden in. Namelijk het ethernet-type en IP-fragment. Zodra er een laag wordt toegevoegd worden de benodigde waarden in de lagen eronder automatisch aangepast. Zie bijvoorbeeld proto in de IP laag, dat eerder al de waarde 'tcp' heeft gekregen door het toevoegen van een TCP laag.

Tot nu toe zijn de lagen Ethernet(), IP(), TCP() en Raw() gebruikt. Voor een complete lijst met protocollen die worden ondersteund door Scapy gebruikt men **ls()**:

```
>>> ls()
AH        : AH
ARP       : ARP
ASN1_Packet : None
BOOTP     : BOOTP
CookedLinux : cooked linux
DHCP      : DHCP options
DHCP6     : DHCPv6 Generic Message)
DHCP6OptAuth : DHCP6 Option - Authentication
DHCP6OptBCMCSDomains : DHCP6 Option - BCMCS Domain Name List
DHCP6OptBCMCSServers : DHCP6 Option - BCMCS Addresses List
DHCP6OptClientFQDN : DHCP6 Option - Client FQDN
...
```

Elk van deze protocollen kan worden gestapeld als laag door middel van "/". De programmeur is compleet vrij in zijn keuze hoe deze lagen te stapelen.
Elke laag heeft een aantal attributen die een waarde toegekend kunnen krijgen. Om een overzicht van die attributen te krijgen gebruikt men **ls([laagnaam])**:

```
>>> ls(IP)
version    : BitField           = (4)
ihl        : BitField           = (None)
tos        : XByteField         = (0)
len        : ShortField         = (None)
id         : ShortField         = (1)
flags      : FlagsField         = (0)
frag       : BitField           = (0)
ttl        : ByteField          = (64)
proto      : ByteEnumField      = (0)
chksum     : XShortField        = (None)
src        : Emph               = (None)
dst        : Emph               = ('127.0.0.1')
options    : PacketListField    = ([])
```

Naast de naam van het attribuut staan hier ook het data-type en de standaardwaarde. Het datatype kan soms ook dat van een nieuwe laag zijn. Een voorbeeld hiervan is de laag *DNSQR(),* die als waarde toegekend kan worden aan het attribuut *qd* in de *DNS()* laag.

Neem nu het eerder gemaakte pakket *pkt*. Het instellen van het *dst* (destination) attribuut van de IP laag kan je als volgt doen:

```
>>> pkt[IP].dst='174.129.25.170'
>>> pkt
<IP  frag=0 proto=tcp dst=174.129.25.170 |<TCP  |<Raw  load='GET / HTTP/1.0\r\n\r\n' |>>>
```

Het pakket kan dus worden aanroepen als een dictionary[2] door IP tussen blokhaken achter het pakket te zetten. Hiermee krijgt men toegang tot de attributen van deze laag en kunnen de waarden worden toegekend en uitgelezen.

Het pakket-object kent verder nog een aantal functies. De twee belangrijkste zijn **summary()** en **show()**, waarbij *summary()* in één regel een overzicht van het pakket geeft en *show()* alle parameters van het pakket laat zien.

```
>>> pkt.summary()
'IP / TCP 192.168.1.175:ftp_data > 174.129.25.170:http S / Raw'
```

[2] Een datatype in Python

```
>>> pkt.show()
###[ IP ]###
  version= 4
  ihl= None
  tos= 0x0
  len= None
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= tcp
  chksum= None
  src= 192.168.1.175
  dst= 174.129.25.170
  \options\
###[ TCP ]###
     sport= ftp_data
     dport= http
     seq= 0
     ack= 0
     dataofs= None
     reserved= 0
     flags= S
     window= 8192
     chksum= None
     urgptr= 0
     options= {}
###[ Raw ]###
        load= 'GET / HTTP/1.0\r\n\r\n'
```

2.3.1 Sets van pakketten

Het is mogelijk om een complete set van pakketten te maken en toe te kennen aan een variabele. In het volgende voorbeeld wordt er een set IP pakketten gemaakt omdat het *dst* attribuut als waarde een compleet subnet krijgt:

```
>>> pkts = IP(dst="www.google.com/30")
>>> [p for p in pkts][3]
[<IP  dst=216.58.212.196 |>, <IP  dst=216.58.212.197 |>, <IP  dst=216.58.212.198 |>, <IP
dst=216.58.212.199 |>]
```

[3] Hier wordt een techniek genaamd *list comprehension* gebruikt.
http://www.secnetix.de/olli/Python/list_comprehensions.hawk

De domeinnaam www.google.com wordt door Scapy's ingebouwde *DNS-resolver* automatisch herleid naar een IP adres. Met de aanduiding '/30' achter de domeinnaam wordt het subnetmasker aangeduid.

Veelgebruikte functies op sets van pakketten zijn **nsummary()** en **filter()**. Bij de volgende voorbeelden wordt aangenomen dat variabele *a* een set pakketten van gevarieerde aard (TCP, UDP, ICMP) representeert.
Met *nsummary()* kan een genummerd overzicht van de pakketten in set *a* worden weergegeven:

```
>>> a.nsummary()
0000 Ether / IP / UDP 192.168.1.175:54581 > 172.217.17.78:https / Raw
0001 Ether / IP / UDP 172.217.17.78:https > 192.168.1.175:54581 / Raw
0002 Ether / fe80::1aa6:f7ff:fe3e:cb35 > ff02::1 (0) / IPv6ExtHdrHopByHop / ICMPv6MLQuery / Raw
0003 Ether / IP / TCP 192.168.1.175:34704 > 104.199.64.165:https PA / Raw
...
```

Met *filter()* kan een nieuwe subset worden gegenereerd op basis van een filter. Als argument gebruikt *filter()* een lambda functie die een waarheidswaarde teruggeeft. In het volgende voorbeeld wordt er op UDP pakketten gefilterd.

```
>>> a.filter(lambda x: UDP in x).nsummary()
0000 Ether / IP / UDP 192.168.1.175:54581 > 172.217.17.78:https / Raw
0001 Ether / IP / UDP 172.217.17.78:https > 192.168.1.175:54581 / Raw
0002 Ether / IP / UDP 192.168.1.2:42927 > 255.255.255.255:7437 / Raw
0003 Ether / IP / UDP 192.168.1.175:58773 > 108.177.119.189:https / Raw
...
```

Elk pakket in de set (door de lambda functie behandeld als *x*) wordt hier stuk voor stuk getest op de aanwezigheid van een UDP segment ('*UDP in x*' resulteert in *True* of *False*). Vervolgens wordt er een overzicht gegeven van de nieuwe set met *nsummary()*.

2.4 Verzenden en ontvangen

Scapy's ingebouwde functionaliteit voor het verzenden van pakketten zijn **send()** en **sendp()**. Waarbij *send()* gebruikt wordt voor layer-3 pakketten, veelal IP of IPv6, en *sendp()* voor layer-2 pakketten zoals Ethernet en 802.11 frames. Ter illustratie; hier wordt met beide functies een ICMP pakket naar 8.8.8.8 verzonden:

```
>>> send(IP(dst="8.8.8.8")/ICMP(), count=1, loop=0)
.
Sent 1 packets.
```

```
>>> sendp(Ether(dst="ff:ff:ff:ff:ff:ff")/IP(dst="8.8.8.8")/ICMP(),   iface="eth0", count=1, loop=0)
.
Sent 1 packets.
```

Omdat we met *sendp()* op layer-2 opereren, moet er nog een netwerkinterface worden gespecificeerd met *iface="eth0"*. Bij het verzenden van een pakket met *send()* is dit niet nodig. Met het argument *count* is het mogelijk om het aantal te verzenden pakketten te geven, in dit geval is dat er één. Met *loop=1* is het tevens mogelijk om continu pakketten te blijven versturen. In ons geval is hier geen sprake van (*loop=0*).

Voor veel doeleinden is het niet genoeg om alleen pakketten te kunnen versturen, vaak wil men nog een antwoord ontvangen. Hiervoor bestaan de functies **sr()**, **sr1()**, **srp()** en **srp1()**. Deze functies versturen allemaal één of meerdere pakket en wachten daarna op antwoord. Net als bij *send()* en *sendp()* duidt de 'p' aan het einde op het gebruik van layer-2. De '1' achter de naam betekent dat de functie na het ontvangen van eerste antwoord wordt gestopt. De functies *sr()* en *srp()* blijven wachten tot alle pakketten beantwoord worden of tot er een time-out optreedt. In het volgende voorbeeld wordt een ICMP pakket naar 8.8.8.8 verstuurd en wordt het antwoord toegekend aan variabele *b.*

```
>>> b = sr1(IP(dst="8.8.8.8")/ICMP())
Begin emission:
Finished to send 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
>>> b
<IP  version=4 ihl=5 tos=0x0 len=28 id=18734 flags= frag=0 ttl=61 proto=icmp chksum=0x1a88
src=8.8.8.8 dst=10.14.0.14 options=[] |<ICMP  type=echo-reply code=0 chksum=0x0 id=0x0
seq=0x0 |>>
```

Waar *sr1()* het eerste antwoord als pakket-object teruggeeft, geeft *sr()* twee sets van pakketten terug: een set beantwoorde pakketten en een set onbeantwoorde pakketten.

```
>>> r  = sr(IP(dst=["8.8.8.8/30"])/ICMP(), timeout=5)
Begin emission:
Finished to send 4 packets.
.*.........
Received 11 packets, got 1 answers, remaining 3 packets
>>> r
(<Results: TCP:0 UDP:0 ICMP:1 Other:0>, <Unanswered: TCP:0 UDP:0 ICMP:3 Other:0>)
```

Als extra argument wordt nog een *timeout* van vijf seconden opgegeven, zodat de functie vijf seconden na het laatst ontvangen antwoord vanzelf stopt. De set beantwoorde en onbeantwoorde pakketten kan vervolgens respectievelijk met *r[0]* en *r[1]* worden aanroepen.

```
>>> r[0].nsummary()
0000 IP / ICMP 10.14.0.14 > 8.8.8.8 echo-request 0 ==> IP / ICMP 8.8.8.8 > 10.14.0.14 echo-reply 0
>>> r[1].nsummary()
0000 IP / ICMP 10.14.0.14 > 8.8.8.10 echo-request 0
0001 IP / ICMP 10.14.0.14 > 8.8.8.11 echo-request 0
0002 IP / ICMP 10.14.0.14 > 8.8.8.9 echo-request 0
```

De set van beantwoorde pakketten bestaat vervolgens weer uit een lijst van paren van het verzonden pakket en het antwoord. Een voorbeeld van een paar uit set r:

```
>>> r[0][0]
(<IP  frag=0 proto=icmp dst=8.8.8.8 |<ICMP  |>>, <IP  version=4 ihl=5 tos=0x0 len=28 id=26968
flags= frag=0 ttl=61 proto=icmp chksum=0xfa5d src=8.8.8.8 dst=10.14.0.14 options=[] |<ICMP
type=echo-reply code=0 chksum=0x0 id=0x0 seq=0x0 |>>)
```

Het verzonden pakket van het paar:

```
>>> r[0][0][0]
<IP  frag=0 proto=icmp dst=8.8.8.8 |<ICMP  |>>
```

Ontvangen antwoord op het verzonden pakket:

```
>>> r[0][0][1]
<IP  version=4 ihl=5 tos=0x0 len=28 id=26968 flags= frag=0 ttl=61 proto=icmp chksum=0xfa5d
src=8.8.8.8 dst=10.14.0.14 options=[] |<ICMP  type=echo-reply code=0 chksum=0x0 id=0x0
seq=0x0 |>>
```

Het is ook mogelijk om pakketten te blijven sturen, zelfs als er al een antwoord is ontvangen. Hiervoor gebruikt men **srloop()** of **srploop()** voor *layer-2*. Met het optionele argument *count* specificeert men het aantal pakketten dat men wilt versturen. Zonder *count* blijft *srloop()* tot in het oneindige pakketten sturen.

```
>>> l = srloop(IP(dst="8.8.8.8")/ICMP(), count=3)
RECV 1: IP / ICMP 8.8.8.8 > 10.14.0.14 echo-reply 0
RECV 1: IP / ICMP 8.8.8.8 > 10.14.0.14 echo-reply 0
RECV 1: IP / ICMP 8.8.8.8 > 10.14.0.14 echo-reply 0
```

2.5 Scanning en sniffing

Met de *sr()* functie is het mogelijk om netwerk-scans uit te voeren. Dit kan worden gerealiseerd door een attribuut in een laag van het te verzenden pakket een *reeks* van waarden te geven. Om bijvoorbeeld een poortscan op host 192.168.1.1 te doen, geeft men voor het attribuut *dport* de reeks (1,1024) op:

```
>>> scan = sr(IP(dst='192.168.1.1')/TCP(flags='S', dport=(1, 1024)), timeout=5)
Begin emission:
..**..Finished to send 1024 packets.
.........................
Received 34 packets, got 2 answers, remaining 1022 packets
>>> scan[0].summary()
IP / TCP 192.168.1.175:ftp_data > 192.168.1.1:domain S ==> IP / TCP 192.168.1.1:domain >
192.168.1.175:ftp_data SA / Padding
IP / TCP 192.168.1.175:ftp_data > 192.168.1.1:http S ==> IP / TCP 192.168.1.1:http >
192.168.1.175:ftp_data SA / Padding
```

Er wordt hier een SYN scan uitgevoerd op de poortnummers 1 tot en met 1024. Er is voor poort 80 (HTTP) en 53 (domain) een tcp segment met de SYN/ACK flag ontvangen (SA). Dit wijst erop dat deze poorten open staan.

Een andere belangrijke functie in Scapy is **sniff()**. Deze functie stelt men in staat om al het verkeer dat door een netwerkinterfacekaart wordt ontvangen of verzonden, af te vangen. Aangezien er vaak veel onbelangrijk netwerkverkeer langs komt is het belangrijk dat er goede filters ingesteld kunnen worden. In *sniff()* is hiervoor het *filter* argument beschikbaar. De waarde van dit argument is een BPF [4]-stijl filter. Als er bijvoorbeeld tien *HTTP* pakketten opgevangen moeten worden, dan kan dit als volgt:

```
>>> cap = sniff(filter="tcp port 80", count=10)
>>> cap
<Sniffed: TCP:10 UDP:0 ICMP:0 Other:0>
```

Er wordt een set van pakketten teruggegeven. Deze set kan bijvoorbeeld weer worden gebruikt om verder onderzoek op te verrichten door middel van het toepassen van extra filters met *filter()*. Overige opties naast *filter* en *count* zijn onder andere *iface, store* en *prn.*

- *iface* netwerkinterfacekaart (standaard wordt over elke interface afgevangen)
- *store* optie om de opgevangen pakketten in het geheugen te bewaren
- *prn* lambda functie om op elk afgevangen pakket uit te voeren
- *lfilter* lambda functie om pakketten te filteren (als alternatief voor *filter*)

[4] Berkeley Packet Filter, o.a. gebruikt in Wireshark en tcpdump

Als men een *HTTP* pakket ('tcp port 80'), verzonden of ontvangen door interface *eth0*, wil laten zien met *show()*, dan kan dit als volgt:

```
>>> cap = sniff(filter="tcp port 80", iface="eth0", prn=lambda x: x.show(), count=1)
###[ Ethernet ]###
  dst= 00:1d:09:0a:e5:b8
  src= 10:c3:7b:ad:56:fc
  type= 0x800
###[ IP ]###
     version= 4
     ihl= 5
     tos= 0x0
     len= 453
     id= 5724
     flags= DF
     frag= 0
     ttl= 64
     proto= tcp
     chksum= 0xa3c7
     src= 192.168.1.175
     dst= 192.30.252.153
     \options\
###[ TCP ]###
        sport= 54776
        dport= http
        seq= 1805204044
        ack= 220627982
        dataofs= 8
        reserved= 0
        flags= PA
        window= 229
        chksum= 0x1597
        urgptr= 0
        options= [('NOP', None), ('NOP', None), ('Timestamp', (4467750, 770469975))]
###[ Raw ]###
           load= 'GET / HTTP/1.1\r\nHost: fluffhoofd.net\r\nConnection: keep-alive\r\nUser-Agent:
Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.62
Safari/537.36\r\nUpgrade-Insecure-Requests: 1\r\nAccept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8\r\nDN
T: 1\r\nAccept-Encoding: gzip, deflate\r\nAccept-Language:
nl-NL,nl;q=0.9,en-US;q=0.8,en;q=0.7\r\n\r\n'
```

Merk de lambda functie als argument van *prn* op. Op elk pakket *x* wordt dus de functie *show()* aangeroepen.

2.6 Pcap bestanden

Vaak is het belangrijk om, bijvoorbeeld na het uitvoeren van *sniff()*, pakketten op te kunnen slaan in een bestand met het .pcap formaat. Hiervoor gebruikt men de functie **wrpcap()**:

```
>>> p = sniff(count=10)
>>> wrpcap("packets.pcap",p)
```

Er worden hier tien pakketten opgevangen en opgeslagen in 'packets.pcap'. Dit bestand is ook geschikt om uit te lezen met een tool als Wireshark. Mocht men met Scapy een .pcap bestand willen openen, dan kan dit met **rdpcap()**:

```
>>> q = rdpcap("packets.pcap")
```

2.7 Overige functies

Om een compleet overzicht te krijgen van Scapy's ingebouwde functies gebruikt men het commando **lsc()**. Zie hiervoor ook bijlage 1. In het volgende overzicht staan enkele functies en methodes die de moeite waard zijn om op te noemen.

`traceroute([IP-adres])`	Geeft een set pakketten van alle hops naar [IP-adres].
`fuzz([laag])`	Geeft willekeurige waardes aan niet expliciet gewijzigde parameters in [laag].
`[Packet].sprintf([vorm])`	Geeft een geformatteerde *string* terug op basis van een [vorm] (bijvoorbeeld: "%TCP.flags%").
`[Packet].command()`	Geeft het commando waarmee een pakket geregenereerd kan worden.
`[PacketList].sessions()`	Geeft een dictionary terug met een samenvatting van elke TCP of UDP sessie in een set pakketten, gekoppeld aan een lijst van pakketten in die sessie.

Meer informatie over het gebruik en de argumenten van een specifieke functie is te vinden op de help-pagina van de functie, op te vragen via **help([functienaam])**.

Naast de functies in de uitvoer van *lsc()* is het ook belangrijk om te weten hoe men globale opties in kan stellen via *conf.* Hiermee kan men bijvoorbeeld de standaard netwerkinterface instellen:

```
>>> conf.iface="eth0"
```

Verder krijgt men met Scapy de mogelijkheid om een systeem-onafhankelijke routing tabel te hanteren, instelbaar via **conf.route**.
Voor verdere informatie over de routing optie, zie **help(conf.route)**. Een complete lijst met configuratie-opties is simpelweg op te vragen met **conf**.

```
>>> conf
...
iface      = 'tun0'
iface6     = 'lo'
interactive = True
interactive_shell = 'ipython'
ipv6_enabled = True
ipython_embedded = True
...
```

3. Voorbeeld

3.1 Wordlist sniffer

In het volgende voorbeeld schrijven we een Python script waarmee we *plaintext* data uit netwerkpakketten kunnen doorzoeken op bepaalde steekwoorden. Deze woorden halen we uit een *wordlist*-bestand; simpelweg een lijst met woorden. Met dit script kunnen we bijvoorbeeld *FTP* wachtwoorden in onbeveiligde verbindingen opvangen zonder dat we elk pakket handmatig moeten bekijken.

Eerst openen we een nieuw Python script en importeren we Scapy:

```
1   #!/usr/bin/env python3
2   from scapy.all import *
```

In de volgende stap declareren we de variabelen *wordlist*, de bestandsnaam van onze woordenlijst, en *words,* waar we de afzonderlijke woorden in opslaan.

```
3   wordlist = "wordlist"
4   words = []
```

We willen nu de woorden uit het bestand *'wordlist'* (die elk op een nieuwe regel moeten staan) toevoegen aan de lijst *words.* Dit doen we als volgt.

```
5   with open(wordlist) as file:
6       words = file.read().splitlines()
```

Dan komt nu het belangrijkste deel van ons script; het definiëren van de *callback-functie* die aangeroepen zal worden door *sniff()*. Deze functie (7) zal verantwoordelijk zijn voor de uitvoer van ons script.

```
7    def get_plaintext(packet):
8        if Raw in packet:
9            text = str(packet[Raw].load)
10           if any(word.lower() in text.lower() for word in words):
11               dst = packet[IP].dst
12               src = packet[IP].src
13               print("#### {} > {} ####".format(dst, src))
14               print(text)
15               print()
```

De functie, we noemen hem *get_plaintext*, krijgt een packet-object als argument. Eerst (8) wordt er gekeken of het pakket een *Raw* laag bevat.
De *Raw*-laag bevat de data die voor ons interessant is, zoals wachtwoorden in onbeveiligde verbindingen (*telnet, FTP, HTTP, IMAP, enz*). Van deze laag kennen we de waarde van *load* toe aan de variabele *text* (9). Vervolgens willen we kijken of elk woord uit onze woordenlijst voorkomt in *text.* Hiervoor gebruiken we list-comprehension *(10)*. Als er een match is wordt *text* samen met de IP-adressen van bestemming en bron naar standaard output uitgevoerd (13, 14).
Nu hoeven we alleen nog maar de sniff-functie toe te voegen. We geven onze functie *get_plaintext* mee als callback aan het argument *prn.* Verder willen we alleen de output van ons script zien, dus is het niet nodig om het resultaat van *sniff()* op te slaan in het geheugen ('store=0').

```
16  sniff(prn=get_plaintext, store=0)
```

3.2 Uitvoering

We zullen nu het script opslaan als *wordsniffer.py*[5]. Voordat we ons script uitvoeren maken we in de map waarin we het script uitvoeren een bestand genaamd 'wordlist' aan. Hierin zetten we op afzonderlijke regels woorden als 'password', 'pass', 'user', 'username' en 'authorization'. Op basis deze woorden zullen de pakketten dan worden gefilterd.
We kunnen nu het script starten. Vergeet *sudo* niet om Scapy root rechten te geven.

```
$ sudo ./wordsniffer.py
#### 141.138.123.456 > 192.168.1.175 ####
b'USER willy\r\n'

#### 192.168.1.175 > 141.138.123.456 ####
b'331 Password required for willy\r\n'

#### 141.138.123.456 > 192.168.1.175 ####
b'PASS wachtwoord123\r\n'
```

We zijn er in geslaagd een *FTP* sessie te onderscheppen. We hebben het wachtwoord 'wachtwoord123' van gebruiker 'willy' weten te achterhalen.
Natuurlijk is het zo dat men met dit script alleen verkeer van en naar de machine waarop het script draait kan onderscheppen. Als men het verkeer van andere machines in een netwerk wilt afluisteren dan zal er eerst een man-in-the-middle aanval moeten worden uitgevoerd of het script moet worden uitgevoerd op een centrale router of switch.

[5] Zie bijlage 2 voor het complete script

Conclusie

Scapy is een krachtige tool zonder een al te steile leercurve. Wat Scapy echter zo krachtig maakt is de combinatie van de Python interpreter met een object-model dat netwerkprotocollen op een laag niveau beschrijft. Hierdoor wordt er van de gebruiker, voor succesvol gebruik, een stevige kennisbasis van netwerkprotocollen geëist. Vanwege de gigantische uitgebreidheid van Scapy's mogelijkheden is deze tool onmisbaar in de gereedschapskist van professionele hackers en ontwikkelaars die veel met netwerken te maken hebben. We kunnen met Scapy niet alleen pakketten maken, we kunnen er onder andere ook pakketten mee analyseren, poortscans mee doen en wachtwoord sniffers mee maken. Het feit dat Scapy een vervanger kan zijn voor een grote reeks bestaande tools maakt het de moeite waard om je te verdiepen in de werking van deze tool.

Bronnen

Literatuur:
Seitz, J (2015). *Black Hat Python*. San Francisco: No Starch Press

Artikelen:
Biondi, Philippe. *Scapy.* http://www.secdev.org/projects/scapy/
Biondi, Philippe. *Documentation.* https://scapy.readthedocs.io/en/latest/
Dobelis, Eriks. *Scapy3k.* https://phaethon.github.io/scapy/api/usage.html
thePacketGeek (2013). *Building Network Tools with Scapy*.
https://thepacketgeek.com/scapy-p-01-scapy-introduction-and-overview/

Bijlagen

1. Output van lsc()

```
>>> lsc()
arpcachepoison      : Poison target's cache with (your MAC,victim's IP)
couple
arping              : Send ARP who-has requests to determine which hosts
are up
bind_layers         : Bind 2 layers on some specific fields' values
bridge_and_sniff    : Forward traffic between two interfaces and sniff
packets exchanged
corrupt_bits        : Flip a given percentage or number of bits from
bytes
corrupt_bytes       : Corrupt a given percentage or number of bytes from
bytes
defrag              : defrag(plist) -> ([not fragmented],
[defragmented],
defragment          : defragment(plist) -> plist defragmented as much as
possible
dyndns_add          : Send a DNS add message to a nameserver for "name"
to have a new "rdata"
dyndns_del          : Send a DNS delete message to a nameserver for
"name"
etherleak           : Exploit Etherleak flaw
fragment            : Fragment a big IP datagram
fuzz                : Transform a layer into a fuzzy layer by replacing
some default values by random objects
getmacbyip          : Return MAC address corresponding to a given IP
address
hexdiff             : Show differences between 2 binary strings
hexdump             : --
hexedit             : Run external hex editor on a packet or bytes. Set
editor in conf.prog.hexedit
is_private_addr     : Returns True if the IPv4 Address is an RFC 1918
private address.
is_promisc          : Try to guess if target is in Promisc mode. The
target is provided by its ip.
linehexdump         : --
ls                  : List  available layers, or infos on a given layer
mtr                 : A Multi-Traceroute (mtr) command:
promiscping         : Send ARP who-has requests to determine which hosts
are in promiscuous mode
```

```
rdpcap              : Read a pcap file and return a packet list
send                : Send packets at layer 3
sendp               : Send packets at layer 2
sendpfast           : Send packets at layer 2 using tcpreplay for
performance
sniff               : Sniff packets
split_layers        : Split 2 layers previously bound
sr                  : Send and receive packets at layer 3
sr1                 : Send packets at layer 3 and return only the first
answer
srbt                : send and receive using a bluetooth socket
srbt1               : send and receive 1 packet using a bluetooth socket
srflood             : Flood and receive packets at layer 3
srloop              : Send a packet at layer 3 in loop and print the
answer each time
srp                 : Send and receive packets at layer 2
srp1                : Send and receive packets at layer 2 and return
only the first answer
srpflood            : Flood and receive packets at layer 2
srploop             : Send a packet at layer 2 in loop and print the
answer each time
tdecode             : Run tshark to decode and display the packet. If no
args defined uses -V
traceroute          : Instant TCP traceroute
tshark              : Sniff packets and print them calling pkt.show(), a
bit like text wireshark
wireshark           : Run wireshark on a list of packets
wrpcap              : Write a list of packets to a pcap file
```

2. wordsniffer.py

```
#!/usr/bin/env python3
from scapy.all import *

wordlist = "wordlist"
words = []

with open(wordlist) as file:
    words = file.read().splitlines()

def get_plaintext(packet):
    if Raw in packet:
        text = str(packet[Raw].load)
        if any(word.lower() in text.lower() for word in words):
            dst = packet[IP].dst
            src = packet[IP].src
            print("#### {} > {} ####".format(dst, src))
            print(text)
            print()

sniff(prn=get_plaintext, store=0)
```