NÄTVERKSSNIFFER I PYTHON

Ett Examensarbete utförd av

Irik Alimbaev

vid YH-utbildning
«IT-SÄKERHETSTEKNIKER»
hos Edugrade
2023

Handledare:
Stefan Holmberg

Innehållsförteckning

Sammanfattning

I den här rapporten beskriver jag sårbarheten i arp-protokollet och hur hackaren kan förfalska ARP-meddelanden för att avlyssna trafiken. En sådan attack kallas Man-In-The-Middle och sker på datalänklagret av OSI-modellen. Därför lägger jag huvudfokus på att utveckla Python Network sniffer som kan upptäcka en anomali i de mottagna ARP-svaren.

Grundtanken är att skriptet måste inkluderas i uppstarten av operativsystemet, det vill säga att sniffer bör köras och fortsätta fungera tillsammans med andra bakgrundsprocesser.

Medan datorn är ansluten till nätverket, tar skriptet emot varje ARP-meddelande för att bearbeta och upptäcka ARP-spoofing.

Man kan också utföra nätverksanalys med hjälp av WireShark. Fördelen med min sniffer är att den analyserar nätverket automatiskt under hela datorns arbetsflöde.

Men för tillfället är skriptet inte helt färdigt, eftersom det fortfarande inte finns någon funktionalitet för att skicka larm om spoofing till email eller messenger.

Det här projektet hjälpte mig att fördjupa min förståelse om hur information överförs i nätverket och höja färdigheten i Python-programmering. Jag fick värdefull erfarenhet av att arbeta med nätverksdata på lägre nivå, det vill säga med råbytes. I sin tur, förmågan att hantera och tolka råbytes i Python, lägger grunden för vidareutvecklingen av automation till nätverkssäkerhet.

Introduktion

IT-säkerhet omfattar många aspekter och en av de viktiga är nätverkssäkerhet. För att ta emot och skicka data mellan enheter över nätverket, används i första hand Ethernet-protokollet.

På länknivå packas data och skickas / tas emot över nätverket i Ethernet-ramar. I sin kärna består ramen av en logisk sekvens av bitar vilket bär MAC och IP-adress med andra viktiga nyckelvariabler.

Det är inte svårt för en hackare att fejka dessa bitar med att bädda in MAC-adressen av sin enhet och routern IP-adressen för att sedan fånga upp anslutningen.

När en enhet ansluter till nätverket, sker direkt en automatisk kommunikation, bestående av ARP-förfrågningar och ARP-svar för att sedan upprätta en anslutning.

Men denna automatisering är en sårbarhet som en hackare kan utnyttja. Kort sagt kan hackaren från sin enhet skicka falska ARP-svar så att legitima enheterna därefter sparar hackarens MAC-adress till sin arp-cache. Då kommer alla ramar att skickas till hackarens MAC-adress.

En sådan attack som kallas "*Man In The Middle",* sker på datalänklagret. Därför är det viktigt för en IT-säkerhetstekniker att börja trafikanalys från den andra nivån av OSI-modellen.

För att analysera nätverket finns det redan färdiga verktyg, som till exempel WireShark. I detta projekt är uppgiften att göra nätverksanalys automatiserad. För att uppnå automatisering använde jag programmeringsspråket Python för att skapa en egen Python Network Sniffer (vidare förkortning PNS)

Bakgrund

Lågnivå processer är vanligtvis osynliga och därför är det kritisk att spåra trafiken om man vill ha kontroll över vad som händer i nätverket och reagera så snabbt som möjligt.

Trafik mellan en dator och en router på nätverket kan avlyssnas av en angripare, samt för en legitim användare kommer datorn att fungera som vanligt.

Självklart finns det välkända verktyg för nätverksanalys, exempelvis WireShark. Men hackare säger inte i förväg när de kommer att utföra MITM-attack.

Därför skulle det vara användbart att ha en automatiserad nätverkssniffer som kontrollerar kommunikation mellan enheter löpande.

Det finns också en färdig modul för python, som kallas Scapy. Men för mig personligen var det viktigt att skriva min egen grundkod, så att jag i framtiden kunde utveckla och anpassa min PNS efter olika behov.

Nästa viktiga punkt för mig är att utforska i detalj varje byte i Ethernet-ramen. För att spara sedan varje logisk sektor av bytes i den egna variabeln, så att säga plocka isär Ethernet-ramen med mina egna händer och få en känsla för hur allt verkligen fungerar.

Syfte

Detta projekt syftar till att skriva en grundläggande kod i Python för att upptäcka arp-spoofing och meddela om en attack genom att skicka till e-post eller messenger.

Vid uppstart av operativsystemet bör PNS startas automatiskt. Programmet ska börja med en oändlig loop som tar emot bytes från gränssnittet.

Nästan omedelbart skickas APR-förfrågningar för att få riktiga APR-svar från alla enheter i nätverket. Därefter kommer skriptet att extrahera enhetens verkliga IP- och MAC-adress från de mottagna ARP-ramarna för att sedan binda dem till varandra.

Den oändliga loop fortsätter att ta emot och bearbeta ARP-ramar. Om något ARP-svar anländer med en IP-adress som redan är bunden till en annan MAC-adress, bör programmet upptäcka en missmatchning och svara enligt programmerad åtgärd.

Avgränsning

För att PNS ska fungera krävs det att man skriver tre variabler manuellt i själva koden.

Just nu finns det inget automatiskt mottagning av dessa variabler i koden. Därför hittills måste man manuellt skriva in namnet av gränssnittet, IP- och MAC-adressen av aktuell datorn i själva koden.

Men detta är en tillfällig begränsning på grund av bristen på tid för att skriva en fullfjädrad programkod.

PNS fungerar bara på Linux. Eftersom på Windows fungerar inte socket. Så jag har ännu inte hittat en lösning för att ta emot och skicka råbytes från gränssnittet i Windows. Koden är skriven för att acceptera och bearbeta råbytes. Det enda som krävs är att hitta lösning i Windows för att få råbytes från gränssnittet. Och då kommer programmet att fungera som planerat.

PNS är inte programmerad ännu för att skicka ett meddelande när en missmatch hittas. Denna funktion kräver ingen speciell utveckling. Från början planerade jag att använda gmail, men nu blockerar Google loggning genom skriptet. Därför behöver jag bara hitta ett alternativ.

Metod

Jag använde Pycharm för att utveckla Python Network Sniffer. WireShark hjälpte mycket för att utforska varje byte i ARP-ramar samt kontrollera och säkerställa att min PNS fångar alla ramar som WireShark gör.

För att köra och testa skriptet använde jag VirtualBox. I virtuell miljö skapade tre stycken virtuella maskin. En pfSense-Router, en Kali Linux och en Linux-mint. Sedan i Linux mint körde PNS och med hjälp av Ettercap i Kali Linux utförde arp-spoofing attack mot Linux-mint och pfSense-Router.

Genomförande

Insamling av information

Jag startade mitt projekt med Internetsökningar efter Python Networksniffer. På YouTube och Google hittade jag två lämpliga alternativ.

Det första använder Python-bibliotek som heter socket. Med hjälp av socket tar man emot råbytes från gränssnittet och sedan utvecklar man en egen kod för varje sektion av råbytes för att printa ut informationen på ett begripligt mänskligt språk.

I det andra alternativet föreslogs att använda en färdig modul - Scapy. Det är faktiskt ett kraftfullt Python-modul som har många funktioner för att hantera nätverksdata på låg nivå. Men den här varianten kräver mer studier av dokumentationen, och dessutom har jag redan investerat mycket tid i början för att utveckla varianten med socket.

Under utveckling av det första alternativet lärde jag mig den grundläggande koden för att ta isär Ethernet-ramen byte efter byte.

Varje protokoll skiljer sig i sin struktur och tar tid för forskning. Jag fokuserade på arp-protokollet och började utforska strukturen i detalj.

Parallellt använde jag en WireShark för att beräkna byte sekvenser för varje sektion i Ethernet-ramen samt för att kontrollera output från min Python Network Sniffer.

Allmän princip för ARP-Sniffer

Grunden för skriptet ligger i att ta emot och skicka så kallas raw bytes från socket. Raw bytes är i hexadecimalt format och är gemensamma för alla operativsystem. De råa byte sedan matas in som argument till `arp_parser(raw_bytes)` funktionen.

Men programmet börjar med `arp_sniffer()` funktionen som kör loop med `arp_parser(raw_bytes)` för att hämta in varje gång Raw bytes från socket och bearbeta.

`send_arp_request()` - Scannar nätverket för att få riktiga MAC- och IP-adresser från alla enheter i nätverket.

Import

För att kunna ta emot och skicka råbytes måste man importera socket och deklarera sedan en variabel för att ansluta till gränssnittet.

```
import socket
```

`import time` - kommer bromsa funktionen, så att data skickas i rätt tid.
`import binascii` - för att konvertera decimalt till hexadecimalt format

Programmet behöver två processer samtidigt, en av dem kommer alltid att vara i lyssningsläge, det vill säga ta emot och bearbeta inkommande ARP-meddelanden, den andra processen kommer att skicka test-ARP-förfrågningar med något intervall.

`from threading import Thread` - För att kunna ha två processer parallellt.

Variabler

```
s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)
s.bind((gränssnitt, 0))
```

`s =` variabel kommer att användas bara för att skicka råbytes.
För att ta emot råbytes kommer det att användas r = variabel, som skapas i den funktionen med en separat processortid (egen Thread).

`interface = 'enp0s3' -` Gränssnitt Namn
`my_ip = '192.168.1.101'` Ip-adress för den här datorn
`my_mac = '08:00:27:be:43:b6'` MAC-adressen för den här datorn.

I det här skedet av att skriva koden kan skriptet ännu inte automatiskt få gränssnitt namnet tillsammans med MAC- och IP-adressen. Därför är det nödvändigt att skriva dessa variabler manuellt direkt i koden.

`ip_mac = {}` Ordbok som lagrar IP-adressen associerad till MAC-adress.

ARP-Parser

De första 14 byten i en Ethernet-ram har alltid samma struktur. Detta är 6 bytes av mottagaren MAC-adressen, 6 bytes av avsändaren MAC-adressen och 2 byte för protokolltyp. Om värde av dessa två byte i hexadecimal är **0806** så är det helt klart en ARP-förfrågan.

För att ta emot råbytes och tolka för vidare operationer skapade jag funktionen:

```
arp_parser(raw_bytes)
```

Men för att filtrera endast ARP trafiken det måste vara villkor precis i början som extraherar ramtypen av de 2 bytes, från 12 till 14:

```
if raw_data[12:14].hex() == '0806':
```

All återstående kod i `arp_parser(raw_bytes)` funktionen kommer endast att vara under detta villkor. Eftersom beräkningen för varje byte kommer att vara bara för ett Ethernet-ram med ett ARP-protokoll.

`arp_parser(raw_bytes)` returnerar också huvud variablarna som:

```
sender_mac_addr, sender_ip_addr, target_mac_addr, target_ip_addr, arp_message, opcode_v.
```

ARP-Sniffer

`arp_sniffer()` är funktionen för vilken en separat processortid kommer att tilldelas och en egen socket-variabel kommer att deklareras för att ta emot råbyte.

```
r =  socket.socket(socket.AF_PACKET,socket.SOCK_RAW)
```

Sedan While loop ska vänta tills råbytes kommer från gränssnittet till variabeln: r. Så snart kommer någon data till gränssnittet, sparas dessa råa byte i `raw_data` variabeln.

```
while True:
    raw_data = r.recvfrom(60)[0]
```

Efter det kommer de råa byten att mattas in i parsing funktionen för att extrahera MAC och IP adresser.

```
    sender_mac_addr,
    sender_ip_addr,
    target_mac_addr,
    target_ip_addr,
    arp_message,
    opcode_v = arp_parser(raw_data)
```

<u>Därefter kommer det viktigaste:</u>

- Först kontrolleras den ifall avsändarens IP-adress finns inte i ordboken och arp-typen är REPLY:

```
if opcode_v == "REPLY" and
f'{".".join(map(str, sender_ip_addr))}' not in
ip_mac.keys():
```

- Om det inte finns i ordboken, sparas avsändarens IP-adress tillsammans med avsändares MAC-address in i ordlistan:

```
ip_mac[f'{".".join(map(str, sender_ip_addr))}'] =
f'{sender_mac_addr.hex(":")}'
```

Eftersom ordboken lagrar två värden i ett par, kommer de att lagras som ett par vilket består av en IP-adress som en nyckel och ett MAC-adress som en värde. Samtidigt printas ett meddelande att sniffaren fått ARP-REPLY:

```
print(arp_message)
```

Och nu utförs kontrollen för ARP-spoofing:

```
elif f'{".".join(map(str, sender_ip_addr))}' in
ip_mac.keys() \
and ip_mac[f'{".".join(map(str, sender_ip_addr))}']
!= sender_mac_addr.hex(':'):
```

Varje sparat IP adress har sin associerade MAC-adress.
Så, om ip-adressen, vilken kom från det nya ARP-svaret redan finns i ordboken, då betyder det att enhetens mac-adress är redan bunden till denna ip-adress.

I händelse av att den här IP-adressen från den nya ARP-svaret har någon annan MAC-adress än den som var bundet tidigare, upptäcks en felmatchning och programmet printar ett meddelande:

```
print(f"---!!!--- arp spoofing detected ---!!!--->>>
({arp_message})".upper())
```

Network scanner

Funktionen `send_arp_request()` bygger upp en Ethernet-frame med en ARP-begäran för att ta emot ARP-svar från tillgängliga enheter i nätverket. Således utförs en nätverksskanning för att registrera IP-adresser och sedan binda dem till aktuell MAC-adresser.

Alla variabler är konstanta förutom den begärda IP-adressen.

```
ethernet_broadcast = b'\xff\xff\xff\xff\xff\xff'
ethernet_source =
binascii.unhexlify(my_mac.replace(':', ''))
e_h_p_h_p_o =
b'\x08\x06\x00\x01\x08\x00\x06\x04\x00\x01'
sender_mac_addr =
binascii.unhexlify(my_mac.replace(':', ''))
sender_ip_addr = bytes(map(int, my_ip.split('.')))
target_mac_addr = b'\x00\x00\x00\x00\x00\x00'
```

Den IP-adressen från listan mellan .1-.255 infogas i hexadecimalt i Ethernet-ramen och sedan sparas i variabeln. Den här variabeln innehåller nu ARP-begäran som består av råa bytes och redo att skickas via socket.

```
for i in range(1, 256):
send_target_ip = my_ip[:10] + str(i)
ip_bytes = bytes(map(int,
send_target_ip.split('.')))
target_ip_addr = ip_bytes
arp_frame = ethernet_broadcast + \
                    ethernet_source + \
                    e_h_p_h_p_o + \
                    sender_mac_addr + \
                    sender_ip_addr + \
                    target_mac_addr + \
                    target_ip_addr
    s.send(arp_frame
```

Resultat

Som ett resultat har jag ett grundläggande skript som körs från konsolen. Programmet visar de hittade enheterna i nätverket och visar IP-adressen tillsammans med MAC-adressen:

```
Host list:
192.168.1.1 08:00:27:e1:79:5d
192.168.1.100 08:00:27:22:46:4f
192.168.1.156 0a:00:27:00:00:00
```

Inkommande ARP-trafik fortsätter att visas som ett vanligt REPLY meddelande:

```
REPLY - 192.168.1.101 at 08:00:27:be:43:b6
```

Så snart den inkommande ARP från en annan enhet har samma IP-adress av den andra host, kommer programmet att visa ett larm meddelande:

```
---!!! ARP SPOOFING DETECTED !!!->>>
(REPLY - 192.168.1.1 AT 08:00:27:22:46:4F)
```

Output i konsol:

```
Host list:
192.168.1.1 08:00:27:e1:79:5d
192.168.1.100 08:00:27:22:46:4f
192.168.1.156 0a:00:27:00:00:00
=====================================================
---!!! ARP SPOOFING DETECTED !!!->>> (REPLY - 192.168.1.1 AT 08:00:27:22:46:4F)
---!!! ARP SPOOFING DETECTED !!!->>> (REPLY - 192.168.1.1 AT 08:00:27:22:46:4F)
REPLY - 192.168.1.101 at 08:00:27:be:43:b6
---!!! ARP SPOOFING DETECTED !!!->>> (REPLY - 192.168.1.1 AT 08:00:27:22:46:4F)
```

Diskussion och slutsatser

Programmet är inte komplett och ganska begränsat. Eftersom det bara fungerar i Linux och funktionaliteten för att skicka larm till mail eller telefon har inte slutförts. För varje maskin krävs att man manuellt skriver in i programkoden, nätverkskort data som MAC- och IP-adresser samt namnet på gränssnittet.

I början tappade jag mycket tid på att testa små saker. Om jag skulle påbörja liknande arbete nästa gång, så skulle jag först försöka utveckla den viktigaste funktionen.

Så när det var dags att skriva denna rapport var jag fortfarande fast i koden för att avsluta de mindre funktionerna.

Huvudmålet var dock att skriva logiken för att beräkna falska ARP-svar, och resten kväver inte så mycket att utveckla, men tar tid.

Referenser

Grundläggande kod för parsing av Ethernet-protokoll och tcp-paket:
https://www.youtube.com/watch?v=WGJC5vT5YJo&list=PL6gx4Cwl9DGDdduy0IPDDHYnUx66Vc4ed

Implementering för att skicka meddelandet till e-post:
https://www.youtube.com/watch?v=g_j6ILT-X0k

Enkel scapy networksniffer:
https://an4ndita.medium.com/write-your-own-arp-spoof-detector-in-python-coding-for-cyber-security-program-5-9c23ff5e6175

Bilagor

Full kod av Python Network Sniffer:

```
import socket
import time
import binascii
from threading import Thread


interface = 'enp0s3'
my_ip = '192.168.1.101'
my_mac = '08:00:27:be:43:b6'

ip_mac = {}

s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)
s.bind((interface, 0))


def arp_parser(raw_ether):
    if raw_ether[12:14].hex() == '0806':
        dst_mac = raw_ether[:6]
        src_mac = raw_ether[6:12]
        eth_type = raw_ether[12:14].hex()
        ether_data = raw_ether[14:]
        hardware_type = raw_ether[14:16]
        protocol_type = raw_ether[16:18]
        hardware_size = raw_ether[18:19]
        protocol_size = raw_ether[19:20]
        opcode = raw_ether[20:22]
        sender_mac_addr = raw_ether[22:28]
        sender_ip_addr = raw_ether[28:32]
        target_mac_addr = raw_ether[32:38]
        target_ip_addr = raw_ether[38:42]
        rest = raw_ether[42:]
        arp_message = ''
        opcode_v = ''

        if int(opcode.hex()) == 1:
            opcode_v = "REQUEST"
        elif int(opcode.hex()) == 2:
            opcode_v = "REPLY"
        src_ipv4 = '.'.join(map(str, sender_ip_addr))
        dst_ipv4 = '.'.join(map(str, target_ip_addr))
        if dst_mac != b'\xff\xff\xff\xff\xff\xff' and sender_ip_addr != b'\x00\x00\x00\x00'
and opcode_v == 'REPLY':
            arp_message = f'{opcode_v} - {src_ipv4} at {sender_mac_addr.hex(":")}'
        if sender_mac_addr != b'\x00\x00\x00\x00\x00\x00' and \
                sender_ip_addr != b'\x00\x00\x00\x00' and \
                target_mac_addr == b'\x00\x00\x00\x00\x00\x00' and \
                target_ip_addr != b'\x00\x00\x00\x00' and \
                sender_ip_addr != target_ip_addr and opcode_v == 'REQUEST':
            arp_message = f'{opcode_v} - Who is at {target_ip_addr} Tell to
{sender_mac_addr}'

        return sender_mac_addr, sender_ip_addr, target_mac_addr, target_ip_addr,
arp_message, opcode_v
```

```
def arp_sniffer():

    r = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(3))
    while True:
        raw_data = r.recvfrom(60)[0]
        if raw_data[12:14].hex() == '0806':
            sender_mac_addr, sender_ip_addr, target_mac_addr, target_ip_addr, arp_message,
opcode_v = arp_parser(raw_data)

            if opcode_v == "REPLY" and f'{".".join(map(str, sender_ip_addr))}' not in
ip_mac.keys():
                ip_mac[f'{".".join(map(str, sender_ip_addr))}'] =
f'{sender_mac_addr.hex(":")}'
                print(arp_message)

            elif f'{".".join(map(str, sender_ip_addr))}' in ip_mac.keys() \
                    and ip_mac[f'{".".join(map(str, sender_ip_addr))}'] !=
sender_mac_addr.hex(':'):
                print(f"---!!!--- arp spoofing detected ---!!!--->>>
({arp_message})".upper())

def send_arp_request_255():

    ethernet_broadcast = b'\xff\xff\xff\xff\xff\xff'
    ethernet_source = binascii.unhexlify(my_mac.replace(':', ''))
    e_h_p_h_p_o = b'\x08\x06\x00\x01\x08\x00\x06\x04\x00\x01'
    sender_mac_addr = binascii.unhexlify(my_mac.replace(':', ''))
    sender_ip_addr = bytes(map(int, my_ip.split('.')))
    target_mac_addr = b'\x00\x00\x00\x00\x00\x00'
    time.sleep(1)
    for i in range(1, 256):
        send_target_ip = my_ip[:10] + str(i)
        ip_bytes = bytes(map(int, send_target_ip.split('.')))
        target_ip_addr = ip_bytes
        arp_frame = ethernet_broadcast + \
                    ethernet_source + \
                    e_h_p_h_p_o + \
                    sender_mac_addr + \
                    sender_ip_addr + \
                    target_mac_addr + \
                    target_ip_addr

        s.send(arp_frame)
        time.sleep(0.01)

    print("============================================================")



def main():

    Thread(target=arp_sniffer).start()
    send_arp_request_255()
    print("Host list:")
    for key, value in ip_mac.items():
        print(key + " " + value)
    print("============================================================")


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