Technologia i rozwiązania

# Skanowanie sieci z Kali Linux

## Receptury

Bezpieczeństwo sieci w Twoich rękach!

Justin Hutchens

Helion

[PACKT] open source* PUBLISHING community experience distilled

Tytuł oryginału: Kali Linux Network Scanning Cookbook

Tłumaczenie: Grzegorz Kowalczyk

ISBN: 978-83-283-0987-6



First published in the English language under the title: 'Kali Linux Network Scanning Cookbook – (9781783982141)'.



Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
*ftp://ftp.helion.pl/przyklady/skakar.zip*

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
*http://helion.pl/user/opinie/skakar*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

# Spis treści

ROZDZIAŁ 2

# Wykrywanie hostów w sieci

Wykrywanie hostów w sieci to proces mający na celu wyszukiwanie i rozpoznawanie komputerów, serwerów, zapór sieciowych, routerów i innych urządzeń podłączonych do sieci. W kontekście testów penetracyjnych wykrywanie hostów jest zazwyczaj przeprowadzane po to, by zidentyfikować potencjalne cele ataku. Zadaniem podczas takiego skanowania nie jest gromadzenie szczegółowych informacji o hostach w sieci, ale wykrywanie obecności takich hostów i ich lokalizacji w sieci. W rezultacie przeprowadzonego wykrywania powinieneś otrzymać listę adresów IP hostów, którą możesz wykorzystać do dalszych analiz. W tym rozdziale omówimy wiele sposobów wykrywania hostów w sieci przy użyciu różnych protokołów sieciowych działających na 2., 3. i 4. warstwie modelu OSI. Znajdziesz tutaj następujące receptury:

- Skanowanie sieci na warstwie 2. przy użyciu programu Scapy.
- Skanowanie sieci na warstwie 2. przy użyciu programu ARPing.
- Skanowanie sieci na warstwie 2. przy użyciu programu Nmap.
- Skanowanie sieci na warstwie 2. przy użyciu programu NetDiscover.
- Skanowanie sieci na warstwie 2. przy użyciu programu Metasploit.
- Skanowanie sieci na warstwie 3. przy użyciu polecenia ping (ICMP).
- Skanowanie sieci na warstwie 3. przy użyciu programu Scapy.
- Skanowanie sieci na warstwie 3. przy użyciu programu Nmap.
- Skanowanie sieci na warstwie 3. przy użyciu programu fping.
- Skanowanie sieci na warstwie 3. przy użyciu programu hping3.
- Skanowanie sieci na warstwie 4. przy użyciu programu Scapy.
- Skanowanie sieci na warstwie 4. przy użyciu programu Nmap.
- Skanowanie sieci na warstwie 4. przy użyciu programu hping3.

Zanim jednak rozpoczniemy szczegółowe omawianie poszczególnych technik, musimy przedstawić kilka podstawowych zagadnień i pojęć. **Model OSI** (ang. *Open Systems Interconnection*) to standard zdefiniowany przez organizację **ISO** (ang. *International Organization for Standardization*), opisujący strukturę komunikacji sieciowej. Model OSI podzielony jest na siedem warstw ściśle ze sobą współpracujących i definiujących sposób, w jaki dane mogą być przekazywane między różnymi systemami. Wyższe warstwy modelu OSI są zwykle widoczne, a użytkownik jest ich świadomy, podczas gdy warstwy niższe działają w sposób zupełnie przezroczysty dla przeciętnego użytkownika, który nawet nie zdaje sobie sprawy z ich obecności. Poszczególne warstwy modelu OSI zostały przedstawione poniżej.

| Warstwa modelu OSI | Opis warstwy | Przykładowe protokoły |
|---|---|---|
| Warstwa 7.: warstwa aplikacji | Najwyższa warstwa protokołu OSI, wykorzystywana przez aplikacje do przesyłania danych w sieci. | HTTP, FTP, Telnet |
| Warstwa 6.: warstwa prezentacji | Warstwa 6. definiuje sposób formatowania bądź organizacji przesyłanych danych. | ASCII, JPEG, PDF, PNG, DOCX |
| Warstwa 5.: warstwa sesji | Zadaniem tej warstwy jest zarządzanie przebiegiem komunikacji (sesji) podczas połączenia między dwoma hostami sieciowymi, jej synchronizacja i zakończenie. | NetBIOS, PPTP, RPC, SOCKS |
| Warstwa 4.: warstwa transportowa | Warstwa transportowa zapewnia integralność transmisji i inne usługi odpowiadające za połączenie między hostami. | TCP, UDP |
| Warstwa 3.: warstwa sieciowa | Warstwa sieciowa jest odpowiedzialna za określanie trasy, jaką przesyłane są pakiety (trasowanie połączenia). | IPv4, IPv6, ICMP, IPSec |
| Warstwa 2.: warstwa łącza danych | Zadaniem warstwy łącza danych jest upakowanie strumienia danych w ramki i przesyłanie ich do warstwy fizycznej. | ARP |
| Warstwa 1.: warstwa fizyczna | Warstwa fizyczna jest odpowiedzialna za przesyłanie strumienia danych za pośrednictwem nośnika łączącego komunikujące się ze sobą urządzenia. | |

Zadaniem niższych warstw modelu OSI jest zapewnienie, że dane przesyłane w sieci dotrą do swojego miejsca przeznaczenia. Bardzo często protokoły sieciowe wykorzystywane na tych warstwach wymagają, aby systemy końcowe przesyłały odpowiedzi na otrzymane żądania, dzięki czemu tych protokołów mogą używać potencjalni napastnicy do identyfikacji hostów działających w sieci. Zanim przejdziemy do szczegółowych receptur, przedstawimy jeszcze kilka protokołów działających na poszczególnych warstwach oraz pokażemy, w jaki sposób mogą być wykorzystywane do wykrywania hostów w sieci.

Skanowanie sieci na warstwie 2. za pomocą protokołu ARP ma następujące zalety i wady:

- Zalety:
  - Duża szybkość.

- Duża niezawodność.
- Wady:
  - Nie pozwala na wykrywanie systemów zdalnych (protokół nieroutowalny).

Skanowanie sieci na warstwie 2. można przeprowadzać z wykorzystaniem protokołu **ARP** (ang. *Address Resolution Protocol*). ARP to protokół warstwy łącza danych, który zajmuje się przede wszystkim translacją logicznych adresów IP warstwy 3. na fizyczne adresy MAC warstwy 2. Kiedy dany system musi dowiedzieć się, jaki adres fizyczny MAC odpowiada docelowemu adresowi IP, rozsyła broadcast z żądaniem ARP do lokalnego segmentu sieci. Pakiet z żądaniem ARP po prostu zadaje wszystkim hostom w sieci pytanie: „Kto ma ten adres IP?". Po otrzymaniu takiego zapytania system, który ma poszukiwany adres IP, komunikuje się bezpośrednio z systemem zadającym pytanie, przesyłając odpowiedź ARP zawierającą adres MAC (druga warstwa modelu OSI). Po otrzymaniu odpowiedzi system wysyłający żądanie aktualizuje zawartość swojego bufora ARP, w którym tymczasowo zapisywany jest rekord wiążący poszukiwany adres IP z adresem MAC. Protokół ARP może być używany do wykrywania aktywnych hostów, ponieważ otrzymanie od zdalnego hosta odpowiedzi na przesłane żądanie nie wymaga przeprowadzenia żadnego uwierzytelniania ani autoryzacji.

W rezultacie dla potencjalnego napastnika mającego możliwość podłączenia się do sieci lokalnej wyszukanie działających w niej hostów jest zadaniem wręcz trywialnym. Można je wykonać poprzez wysyłanie serii żądań ARP do wszystkich adresów IP z danej podsieci i następnie zapisywanie adresów IP, z których przesłane zostały odpowiedzi. Wyszukiwanie hostów w sieci z wykorzystaniem protokołu ARP ma jednak swoje zalety i wady. Niewątpliwie jest to bardzo skuteczne rozwiązanie, głównie ze względu na fakt, że jest najszybszą i dającą najbardziej wiarygodne wyniki metodą skanowania. Z drugiej strony należy pamiętać, że niestety ARP jest protokołem nieroutowalnym i z tego względu może być używany jedynie do wykrywania hostów w sieci lokalnej.

Skanowanie sieci na warstwie 3. z wykorzystaniem protokołu ICMP ma następujące zalety i wady:

- Zalety:
  - Pozwala na wykrywanie hostów zdalnych (protokół routowalny).
  - Względnie duża szybkość działania.
- Wady:
  - Wolniejsze niż skanowanie ARP.
  - Protokół ICMP jest często blokowany przez zapory sieciowe.

Skanowanie na warstwie 3. jest prawdopodobnie najbardziej znaną i najpowszechniej stosowaną przez administratorów i personel techniczny metodą wykrywania aktywnych hostów w sieci. Słynne polecenie `ping`, które możesz znaleźć zarówno w systemach Windows, jak i Linux, działa właśnie na warstwie 3. i wykorzystuje protokół **ICMP** (ang. *Internet Control Message Protocol*). Choć protokół ICMP ma całkiem sporo interesujących funkcji, jedną z najbardziej użytecznych są komunikaty żądania echa (ang. *Echo Request*) i odpowiedzi na echo (ang. *Echo Reply*). Żądanie *ICMP Echo Request* jest technicznym odpowiednikiem sytuacji, w której jeden

system pyta drugi: „Hej, jesteś tam?”. Jak można się spodziewać, odpowiedź *ICMP Echo Reply* to komunikat, za pomocą którego zapytany system odpowiada: „Tak, jestem”. Aby sprawdzić, czy pod danym adresem IP kryje się jakiś host, nasz system może na ten adres przesłać żądanie *ICMP Echo Request*. Jeżeli host, który ma taki adres IP, jest włączony i odpowiednio skonfigurowany, po otrzymaniu takiego żądania prześle odpowiedź *ICMP Echo Reply*. Protokół ICMP może być z powodzeniem wykorzystywany do wykrywania hostów w całych sieciach, poprzez sukcesywne pingowanie w pętli kolejnych adresów IP z danego zakresu.

W wyniku takiej operacji powinieneś otrzymać listę adresów IP, z których nadesłane zostały odpowiedzi na ping. Skanowanie sieci na warstwie 3. jest bardzo efektywnym rozwiązaniem, ponieważ do wykrywania hostów wykorzystuje protokół routowalny, ale mimo to ma również swoje wady. Po pierwsze, skanowanie z użyciem protokołu ICMP nie jest tak szybkie jak w przypadku protokołu ARP. Po drugie, taki rodzaj skanowania nie daje tak pewnych rezultatów jak skanowanie ARP, ponieważ niektóre systemy są celowo skonfigurowane tak, aby nie odpowiadać na żądania *ICMP Echo Request*, a co gorsza, protokół ICMP jest bardzo często blokowany przez zapory sieciowe. Nie zmienia to jednak w niczym faktu, że jest to szybki, wygodny i często stosowany sposób wykrywania obecności hostów w zdalnych sieciach.

Skanowanie na warstwie 4. to bardzo efektywne rozwiązanie, ponieważ publicznie dostępne systemy znajdują się najczęściej w publicznej przestrzeni adresów IP i udostępniają różne usługi sieciowe za pośrednictwem protokołów **TCP** (ang. *Transmission Control Protocol*) lub **UDP** (ang. *User Datagram Protocol*). W źle zabezpieczonych środowiskach sieciowych odpowiedź ze zdalnego serwera można otrzymać po przesłaniu na jego adres IP niemal dowolnego żądania UDP lub TCP. Z drugiej strony, jeżeli w skanowanym środowisku zaimplementowana jest zapora sieciowa z analizą stanu pakietów (ang. *statefull filtering/statefull firewall*), otrzymać odpowiedź ze zdalnego hosta będzie można tylko w przypadku, kiedy żądanie SYN zostanie przesłane na konkretny port sieciowy, na którym działa określona usługa sieciowa. Co ciekawe, w większości sytuacji użycie odpowiednio dobranego żądania pozwala na wykrywanie hostów nawet w bardzo dobrze skonfigurowanych i zabezpieczonych środowiskach sieciowych. Biorąc jednak pod uwagę fakt, że istnieje 65 536 dostępnych portów UDP i TCP, pełne skanowanie usług na wszystkich portach może być bardzo czasochłonne. Z tego powodu najlepszym podejściem do skanowania sieci na warstwie 4. z użyciem protokołów TCP i UDP będzie wypracowanie rozsądnego kompromisu pomiędzy dokładnością i funkcjonalnością skanu.

Skanowanie sieci na warstwie 4. z wykorzystaniem protokołu TCP ma następujące zalety i wady:

- Zalety:
  - Pozwala na wykrywanie systemów zdalnych (protokół routowalny).
  - Daje bardziej wiarygodne wyniki niż ICMP (filtrowanie jest mniej powszechne lub implementowane selektywnie).
- Wady:
  - Obecność zapór sieciowych z analizą stanu pakietów może powodować, że otrzymane wyniki będą niejednoznaczne.
  - Dokładne skanowanie może zajmować wiele czasu.

Skanowanie sieci na warstwie 4. z wykorzystaniem protokołu TCP polega na wysyłaniu do potencjalnych adresów przeznaczenia pakietów TCP z ustawionymi różnymi flagami, które mogą powodować różne reakcje zdalnego hosta, pozwalające na jego identyfikację. Nieoczekiwane otrzymanie pakietu z ustawioną flagą `FIN` (ang. *Finish*) czy `ACK` (ang. *Acknowledge*) bardzo często może powodować odesłanie przez zdalnego hosta odpowiedzi `RST` (ang. *Reset*). Wysłanie pakietów `SYN` (ang. *Synchronize*) do zdalnego hosta bardzo często powoduje otrzymanie odpowiedzi `SYN+ACK` lub `RST`, w zależności od statusu danej usługi sieciowej. Zamiarem atakującego nie jest wymuszenie otrzymania konkretnej odpowiedzi, ale po prostu otrzymanie dowolnej odpowiedzi — dowolna odpowiedź ze zdalnego hosta jest dla nas wystarczającym potwierdzeniem, że taki host istnieje.

Skanowanie sieci na warstwie 4. z wykorzystaniem protokołu UDP ma następujące zalety i wady:

- Zalety:
  - Pozwala na wykrywanie systemów zdalnych (protokół routowalny).
  - Pozwala na wykrywanie systemów zdalnych, na których wszystkie usługi TCP są filtrowane przez zaporę sieciową.
- Wady:
  - Niespójny sposób użycia i filtrowanie odpowiedzi *ICMP Port Unreachable* (port niedostępny) powoduje, że zastosowanie tej metody do masowego skanowania sieci daje niemiarodajne rezultaty.
  - Specyficzne dla poszczególnych usług sieciowych metody sondowania powodują ograniczenie dokładności skanowania i zwiększają czas jego realizacji.

Skanowanie z wykorzystaniem protokołu UDP polega na wysyłaniu sondujących pakietów UDP do różnych portów sieciowych w celu wymuszenia odpowiedzi zdalnego systemu. Skanowanie UDP może być czasami bardzo efektywne w wykrywaniu hostów, na których wszystkie usługi TCP są filtrowane przez zaporę sieciową. Z drugiej jednak strony skanowanie UDP często jest niemiarodajne, ponieważ niektóre usługi odpowiadają na pakiety UDP za pomocą komunikatów *ICMP Port Unreachable*, podczas gdy inne mogą odpowiadać tylko na odpowiednio dobrane żądania dostosowane wyłącznie do określonych usług. Dodatkowo bardzo często pakiety ICMP są filtrowane przez reguły ruchu wychodzącego na zaporach sieciowych, co powoduje, że masowe skanowanie sieci za pomocą tej metody staje się problematyczne. Z tego powodu efektywne zastosowanie skanowania UDP wymaga zazwyczaj dobrania technik specyficznych dla poszczególnych usług sieciowych.

# Skanowanie sieci na warstwie 2. przy użyciu programu Scapy

Scapy to potężne, interaktywne narzędzie, za pomocą którego możesz przechwytywać, analizować i modyfikować ruch sieciowy. Co więcej, możesz również używać tego narzędzia do tworzenia i wstrzykiwania do sieci własnych pakietów sieciowych, zgodnych ze standardami

różnych protokołów komunikacyjnych. Scapy to także biblioteka funkcji języka Python, dzięki której możesz tworzyć bardzo wydajne skrypty przetwarzające i modyfikujące ruch sieciowy. W tym podrozdziale pokażemy, w jaki sposób możesz posłużyć się programem Scapy do skanowania sieci z wykorzystaniem protokołu ARP oraz jak przy użyciu biblioteki Scapy pisać w języku Python skrypty skanujące sieć na warstwie 2. modelu OSI.

## Przygotuj się

Aby można było za pomocą pakietu Scapy przeprowadzić skanowanie na warstwie 2., w sieci lokalnej musi działać przynajmniej jeden system, który będzie odpowiadał na żądania ARP. W przedstawionym przykładzie wykorzystujemy kombinację systemów Linux i Windows. Więcej szczegółowych informacji na temat instalacji i konfiguracji tych systemów w naszym środowisku testowym znajdziesz w rozdziale 1., w recepturach „Instalacja systemu Metasploitable2" oraz „Instalacja systemu Windows". Oprócz tego w tej recepturze będziemy używać edytorów tekstu, takich jak VIM czy Nano, do napisania skryptu skanującego w języku Python i zapisania go w systemie plików. Więcej szczegółowych informacji na temat pisania skryptów znajdziesz w rozdziale 1., w recepturze „Praca z edytorami tekstu VIM i Nano".

## Jak to zrobić?

Aby zrozumieć, jak działa skanowanie z wykorzystaniem protokołu ARP, użyjemy programu Scapy do utworzenia własnych pakietów sieciowych ARP, za pomocą których będziemy mogli wykrywać oraz identyfikować hosty w sieci LAN. By uruchomić program Scapy w systemie Linux, przejdź do okna terminala i wykonaj polecenie **`scapy`**. Następnie możesz użyć funkcji `display()` do wyświetlenia domyślnej konfiguracji obiektów ARP tworzonych w programie Scapy, tak jak to zostało przedstawione poniżej.

```
root@KaliLinux:~# scapy
Welcome to Scapy (2.2.0)
>>> ARP().display()
###[ ARP ]###
   hwtype= 0x1
   ptype= 0x800
   hwlen= 6
   plen= 4
   op= who-has
   hwsrc= 00:0c:29:fd:01:05
   psrc= 172.16.36.232
   hwdst= 00:00:00:00:00:00
   pdst= 0.0.0.0
```

Zauważ, że zarówno źródłowy adres IP, jak i adres MAC zostały automatycznie skonfigurowane na wartości odpowiadające hostowi, na którym został uruchomiony program Scapy. Wartości te nigdy nie są zmieniane, z wyjątkiem sytuacji, w której chcesz ukryć swój rzeczywisty adres źródłowy. Domyślna wartość kodu operacji ARP (ang. *ARP opcode*) jest automatycznie ustawiana

na wartość who-has, która określa, że generowany pakiet będzie żądał przesłania powiązania adresu IP z adresem MAC. W takiej sytuacji jedynym parametrem, który musisz podać, jest adres IP celu. Aby to zrobić, utworzymy nowy obiekt, przypisując funkcję ARP do zmiennej. Nazwa zmiennej obiektowej nie ma znaczenia (w naszym przykładzie będzie to arp_request). Kod został przedstawiony poniżej.

```
>>> arp_request = ARP()
>>> arp_request.pdst = "172.16.36.135"
>>> arp_request.display()
###[ ARP ]###
   hwtype= 0x1
   ptype= 0x800
   hwlen= 6
   plen= 4
   op= who-has
   hwsrc= 00:0c:29:65:fc:d2
   psrc= 172.16.36.132
   hwdst= 00:00:00:00:00:00
   pdst= 172.16.36.135
```

Zwróć uwagę, że za pomocą funkcji display() można sprawdzić, czy wartości parametrów konfiguracyjnych obiektu ARP zostały zmienione zgodnie z oczekiwaniami. W przykładzie użyjemy adresu IP hosta docelowego, działającego w naszym środowisku testowym. Aby wysłać gotowe żądanie ARP i wyświetlić otrzymaną odpowiedź, możemy teraz użyć funkcji sr1().

```
>>> sr1(arp_request)
Begin emission:
.....................................*Finished to send 1 packets.
Received 39 packets, got 1 answers, remaining 0 packets
<ARP hwtype=0x1 ptype=0x800 hwlen=6 plen=4 op=is-at
hwsrc=00:0c:29:3d:84:32 psrc=172.16.36.135 hwdst=00:0c:29:65:fc:d2
pdst=172.16.36.132 |<Padding
load='\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' |>>
```

Innym sposobem wykonania takiej samej operacji może być bezpośrednie wywołanie tej funkcji i przekazanie jej w wierszu polecenia wszystkich niezbędnych argumentów, tak jak to zostało pokazane na listingu poniżej. Dzięki takiemu rozwiązaniu możemy uniknąć konieczności tworzenia dodatkowych zmiennych, a także zrealizować całą operację za pomocą jednego wiersza kodu.

```
>>> sr1(ARP(pdst="172.16.36.135"))
Begin emission:
.......................*Finished to send 1 packets.
Received 26 packets, got 1 answers, remaining 0 packets
<ARP hwtype=0x1 ptype=0x800 hwlen=6 plen=4 op=is-at
hwsrc=00:0c:29:3d:84:32 psrc=172.16.36.135 hwdst=00:0c:29:65:fc:d2
pdst=172.16.36.132 |<Padding
load='\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' |>>
```

Jak widać, we wszystkich przedstawionych przypadkach po wysłaniu żądania otrzymywana jest odpowiedź wskazująca, że do adresu IP `172.16.36.135` przypisany jest adres MAC `00:0C:29:3D:84:32`. Jeżeli teraz ponownie wykonasz taką operację, ale jako argument wywołania podasz adres IP nieaktywnego hosta, nie otrzymasz żadnej odpowiedzi, a uruchomiona funkcja będzie przez cały czas analizować nadchodzący ruch na lokalnym interfejsie sieciowym.

Aby zatrzymać działanie funkcji, powinieneś nacisnąć kombinację klawiszy *Ctrl+C*. Zamiast tego możesz również podczas wywoływania funkcji zdefiniować maksymalny czas, po którym jej działanie zostanie zakończone (ang. *timeout*). Wykorzystanie parametru `timeout` nabiera szczególnego znaczenia, jeżeli używasz Scapy z poziomu skryptów języka Python. By ustawić maksymalny czas działania, musisz w wierszu polecenia użyć dodatkowego argumentu wywołania funkcji wysyłającej żądanie ARP, reprezentującego czas, wyrażony w sekundach, przez jaki funkcja będzie oczekiwała na nadejście odpowiedzi.

```
>>> arp_request.pdst = "172.16.36.134"
>>> sr1(arp_request, timeout=1)
Begin emission:
.....................................................................
............Finished to send 1 packets. ..............................
.....................................................................
.....................................
Received 3285 packets, got 0 answers, remaining 1 packets
>>>
```

Jeżeli teraz, po ustawieniu parametru `timeout`, wyślemy żądanie ARP do nieistniejącego hosta, po upływie oznaczonego czasu otrzymamy informację, że żadna odpowiedź nie została otrzymana. Co ciekawe, odpowiedzi na żądania wysłane przez funkcję mogą również być przypisane do zmiennej, dzięki czemu możemy dalej przetwarzać otrzymaną odpowiedź poprzez odwołanie się do takiej zmiennej, tak jak to zostało przedstawione poniżej.

```
>>> response = sr1(arp_request, timeout=1)
Begin emission:
................................*Finished to send 1 packets.
Received 37 packets, got 1 answers, remaining 0 packets
>>> response.display()
###[ ARP ]###
   hwtype= 0x1
   ptype= 0x800
   hwlen= 6
   plen= 4
   op= is-at
   hwsrc= 00:0c:29:3d:84:32
   psrc= 172.16.36.135
   hwdst= 00:0c:29:65:fc:d2
   pdst= 172.16.36.132
###[ Padding ]###
      load=
'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

Scapy może być również wykorzystywany jako biblioteka funkcji dla skryptów w języku Python, dzięki którym możesz zautomatyzować nużące, często powtarzane zadania. Dzięki połączeniu elastyczności Pythona z funkcjonalnością bibliotek Scapy możesz bez trudu napisać skrypt przechodzący w pętli przez kolejne adresy IP lokalnych podsieci i wysyłający do każdego z hostów odpowiednie żądanie ARP. Przykład w pełni funkcjonalnego skryptu, przeprowadzającego skanowanie sieci na warstwie 2., został przedstawiony poniżej.

```
#!/usr/bin/python

import logging
import subprocess
logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
from scapy.all import *

if len(sys.argv) != 2:
   print "Usage - ./arp_disc.py [interface]"
   print "Example - ./arp_disc.py eth0"
   print "Example will perform an ARP scan of the local subnet to which eth0 is
assigned"
   sys.exit()

interface = str(sys.argv[1])

ip = subprocess.check_output("ifconfig " + interface + " | grep 'inet addr' | cut -d
':' -f 2 | cut -d ' ' -f 1", shell=True).strip()
prefix = ip.split('.')[0] + '.' + ip.split('.')[1] + '.' + ip.split('.')[2] + '.'

for addr in range(0,254):
   answer=sr1(ARP(pdst=prefix+str(addr)),timeout=1,verbose=0)
      if answer == None:
         pass
      else:
         print prefix+str(addr)
```

Pierwszy wiersz skryptu wskazuje lokalizację interpretera języka Python, dzięki czemu skrypt może zostać wykonany bez konieczności podawania tej informacji w wierszu wywołania. Następnie skrypt importuje wszystkie funkcje Scapy oraz definiuje poziomy logowania, co pozwala na wyeliminowanie niepotrzebnych elementów w wynikach działania programu. Dalej importowana jest biblioteka `subprocess`, która pozwala na łatwe wyodrębnianie informacji z wywołań systemowych. Drugi blok kodu zawiera instrukcję warunkową, sprawdzającą, czy w wierszu poleceń została podana odpowiednia liczba argumentów wywołania skryptu. Jeżeli nie, na ekranie wyświetlana jest krótka informacja, składająca się z opisu składni, przykładu wywołania oraz określenia przeznaczenia skryptu.

Dalej znajduje się pojedynczy wiersz kodu, w którym argument wywołania skryptu zostaje przypisany do zmiennej `interface`. Kolejny blok kodu wykorzystuje funkcję `check_output()` z biblioteki `subprocess` do wywołania polecenia `ifconfig` i za pomocą poleceń `grep` i `cut` wyodrębnia z wyników jego działania adres IP interfejsu sieciowego podanego jako argument

wywołania skryptu. Wynik tej operacji zostaje przypisany do zmiennej `ip`. Następnie za pomocą funkcji `split` ze zmiennej `ip` wyodrębniany jest podciąg znaków reprezentujący adres `/24` tej podsieci. Na przykład jeżeli w zmiennej `ip` przechowywany jest adres `192.168.11.4`, to do zmiennej `prefix` zostanie przypisana wartość `192.168.11.`. W ostatnim bloku kodu umieszczona została pętla `for`, która realizuje właściwe skanowanie. Pętla przechodzi kolejno przez wartości od `0` do `254` i w każdej iteracji wartość licznika pętli jest dołączana do prefiksu sieci. W naszym przykładzie przedstawionym wcześniej żądanie ARP zostanie rozesłane do wszystkich hostów o adresach IP od `192.168.11.0` do `192.168.11.254`. Jeżeli dany host odeśle odpowiedź, na ekranie wyświetlony zostanie jego adres IP, wskazujący, że host o takim adresie jest aktywny. Po zapisaniu skryptu w lokalnym katalogu na dysku możesz spróbować uruchomić go z poziomu okna terminala, wpisując w wierszu wywołania kropkę, prawy ukośnik i nazwę skryptu, tak jak to zostało przedstawione w przykładzie poniżej.

```
root@KaliLinux:~# ./arp_disc.py
Usage - ./arp_disc.py [interface]
Example - ./arp_disc.py eth0
Example will perform an ARP scan of the local subnet to which eth0 is assigned
```

Jeżeli skrypt zostanie uruchomiony bez żadnych argumentów wywołania, na ekranie wyświetlony zostanie wspomniany wcześniej opis sposobu użycia, z którego wynika, że poprawne uruchomienie skryptu wymaga podania jednego argumentu, reprezentującego nazwę interfejsu sieciowego, który zostanie użyty do przeprowadzenia skanu. W przykładzie przedstawionym poniżej skrypt został wywołany dla interfejsu `eth0`.

```
root@KaliLinux:~# ./arp_disc.py eth0
172.16.36.1
172.16.36.2
172.16.36.132
172.16.36.135
172.16.36.254
```

Po uruchomieniu skrypt sprawdza, do jakiej podsieci lokalnej podłączony jest interfejs `eth0`, wykonuje skanowanie ARP i wyświetla na ekranie listę adresów IP hostów, które przesłały odpowiedź na żądanie ARP. Aby sprawdzić, jak ten skan działa na poziomie sieci, możesz wcześniej uruchomić program Wireshark, włączyć przechwytywanie ruchu sieciowego i po uruchomieniu skryptu obserwować, jak żądania ARP są wysyłane do kolejnych hostów i jak aktywne hosty odsyłają odpowiedzi, tak jak to pokazano na rysunku poniżej.

```
Broadcast           ARP          42 Who has 172.16.36.1?  Tell 172.16.36.67
Vmware_fd:01:05     ARP          60 172.16.36.1 is at 00:50:56:c0:00:08
Broadcast           ARP          42 Who has 172.16.36.2?  Tell 172.16.36.67
Vmware_fd:01:05     ARP          60 172.16.36.2 is at 00:50:56:ff:2a:8e
```

W razie potrzeby możesz łatwo przekierować wyniki działania skryptu do pliku tekstowego na dysku. Tego pliku będziesz mógł następnie użyć do przeprowadzania kolejnych skanów. Przekierowanie strumienia danych z wyjścia skryptu możesz uzyskać, dołączając na końcu polecenia znak większości (>) i nazwę pliku, w którym mają zostać zapisane dane, tak jak to zostało przedstawione w przykładzie poniżej.

```
root@KaliLinux:~# ./arp_disc.py eth0 > output.txt
root@KaliLinux:~# ls output.txt
output.txt
root@KaliLinux:~# cat output.txt
172.16.36.1
172.16.36.2
172.16.36.132
172.16.36.135
172.16.36.254
```

Po zakończeniu działania skryptu i zapisaniu wyników działania w pliku możesz użyć polecenia `ls` do sprawdzenia, czy plik wynikowy został utworzony, a następnie za pomocą polecenia `cat` wyświetlić zawartość tego pliku. W razie potrzeby możesz łatwo zmodyfikować skrypt tak, aby wysyłał żądania ARP tylko do hostów, których adresy IP znajdują się w pliku tekstowym, podanym jako argument wywołania skryptu. Aby to zrobić, musimy najpierw utworzyć plik tekstowy, w którym będzie zapisana lista adresów IP hostów do przeskanowania. Tę listę można przygotować za pomocą edytorów tekstu takich jak VIM czy Nano. Aby przetestować poprawność działania zmodyfikowanej wersji skryptu, powinieneś w tym pliku umieścić adresy zarówno aktywnych hostów (które odkryliśmy wcześniej), jak i kilka nieaktywnych adresów IP z tej podsieci. Plik z listą adresów możesz przygotować za pomocą jednego z poleceń przedstawionych poniżej.

```
root@KaliLinux:~# vim iplist.txt
root@KaliLinux:~# nano iplist.txt
```

Po utworzeniu pliku z listą adresów IP możesz za pomocą polecenia `cat` sprawdzić jego zawartość. Zakładając, że plik został poprawnie przygotowany i zapisany na dysku, na ekranie powinna się pojawić lista zawierająca adresy IP, które wpisałeś w edytorze tekstu.

```
root@KaliLinux:~# cat iplist.txt
172.16.36.1
172.16.36.2
172.16.36.232
172.16.36.135
172.16.36.180
172.16.36.203
172.16.36.205
172.16.36.254
```

Aby utworzyć skrypt, który będzie pobierał listę adresów IP do skanowania z pliku tekstowego, możesz albo zmodyfikować skrypt z poprzedniego ćwiczenia, albo po prostu utworzyć nowy skrypt. Ponieważ nasz skrypt będzie odczytywał dane z pliku na dysku, musimy do niego wstawić kilka nowych poleceń, realizujących w języku Python operacje plikowe. Przykład kodu takiego skryptu został przedstawiony poniżej.

```
#!/usr/bin/python

import logging
logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
from scapy.all import *
```

```
if len(sys.argv) != 2:
   print "Usage - ./arp_disc.py [filename]"
   print "Example - ./arp_disc.py iplist.txt"
   print "Example will perform an ARP scan of the IP addresses listed in iplist.txt"
   sys.exit()

filename = str(sys.argv[1])
file = open(filename,'r')

for addr in file:
   answer = sr1(ARP(pdst=addr.strip()),timeout=1,verbose=0)
   if answer == None:
      pass
   else:
      print addr.strip()
```

Jedyną istotną różnicą między tym skryptem a jego poprzednią wersją jest to, że zamiast zmiennej `interface` tworzona jest teraz zmienna obiektowa `file`. Plik, którego nazwa została przekazana do skryptu, zostaje otwarty za pomocą funkcji `open()`. Zauważ, że drugim argumentem wywołania tej funkcji jest litera r, wskazująca, że plik powinien zostać otwarty w trybie tylko do odczytu. Po otwarciu pliku pętla `for` pobiera z niego kolejne adresy IP, wysyła do nich żądanie ARP i wyświetla na ekranie adresy hostów, z których nadeszły odpowiedzi. Nowy skrypt może być uruchamiany w taki sam sposób jak jego poprzednia wersja:

```
root@KaliLinux:~# ./arp_disc.py
Usage - ./arp_disc.py [filename]
Example - ./arp_disc.py iplist.txt
Example will perform an ARP scan of the IP addresses listed in iplist.txt
```

Jeżeli skrypt zostanie uruchomiony bez żadnych argumentów wywołania, na ekranie wyświetlony zostanie wspomniany wcześniej opis sposobu użycia. Wynika z niego, że poprawne uruchomienie skryptu wymaga podania jednego argumentu, reprezentującego nazwę pliku tekstowego z listą adresów IP hostów, które powinny być przeskanowane. W przykładzie przedstawionym poniżej lista adresów IP znajduje się w pliku o nazwie *iplist.txt*, zlokalizowanym w bieżącym katalogu roboczym.

```
root@KaliLinux:~# ./arp_disc.py iplist.txt
172.16.36.2
172.16.36.1
172.16.36.132
172.16.36.135
172.16.36.254
```

Po uruchomieniu skrypt zacznie rozsyłać żądania ARP do hostów, których adresy IP znajdują się w pliku, i wyświetli na ekranie adresy hostów, z których otrzymane zostaną odpowiedzi. Każdy wyświetlony adres IP reprezentuje aktywny system, podłączony do sieci LAN. Podobnie jak pokazywaliśmy w poprzednim przykładzie, w razie potrzeby możesz bez trudu przekierować wyniki działania skryptu do pliku na dysku. Aby to zrobić, powinieneś na końcu wiersza wywołania skryptu dodać znak większości (>) i nazwę pliku, w którym mają zostać zapisane dane.

```
root@KaliLinux:~# ./arp_disc.py iplist.txt > output.txt
root@KaliLinux:~# ls output.txt
output.txt
root@KaliLinux:~# cat output.txt
172.16.36.2
172.16.36.1
172.16.36.132
172.16.36.135
172.16.36.254
```

Po zakończeniu działania skryptu i zapisaniu wyników w pliku możesz użyć polecenia `ls` do sprawdzenia, czy plik wynikowy został utworzony, a następnie za pomocą polecenia `cat` wyświetlić zawartość tego pliku.

## Jak to działa?

Skanowanie ARP w bibliotece Scapy jest realizowane za pomocą funkcji `sr1()` (ang. *send/receive one*; wyślij/odbierz jeden pakiet). Funkcja po wywołaniu wstrzykuje do sieci pakiet zdefiniowany przez argument wywołania i następnie oczekuje na nadejście odpowiedzi. W naszym przypadku rozsyłane jest żądanie ARP typu broadcast. Zastosowanie biblioteki Scapy znakomicie ułatwia implementację takiego rozwiązania w skryptach i pozwala na jego wykorzystanie do skanowania wielu systemów naraz.

# Skanowanie sieci na warstwie 2. przy użyciu programu ARPing

**ARPing** to narzędzie działające z poziomu wiersza poleceń konsoli. Jego funkcjonalność jest nieco zbliżona do funkcjonalności powszechnie używanego polecenia `ping`. Za pomocą tego narzędzia możesz sprawdzić, czy host o podanym adresie IP z sieci lokalnej jest aktywny. W tej recepturze pokażemy, w jaki sposób możesz używać programu ARPing do wykrywania aktywnych hostów działających w sieci lokalnej.

## Przygotuj się

Aby można było za pomocą pakietu ARPing przeprowadzić skanowanie na warstwie 2., w sieci lokalnej musi działać przynajmniej jeden system, który będzie odpowiadał na żądania ARP. W przedstawionych przykładach wykorzystujemy kombinację systemów Linux i Windows. Więcej szczegółowych informacji na temat instalacji i konfiguracji tych systemów w naszym środowisku testowym znajdziesz w rozdziale 1., w recepturach „Instalacja systemu Metasploitable2" oraz „Instalacja systemu Windows". Oprócz tego w tej recepturze będziemy używać edytorów

tekstu, takich jak VIM czy Nano, do napisania skryptu skanującego w języku Python i zapisania go w systemie plików. Więcej szczegółowych informacji na temat pisania skryptów znajdziesz w rozdziale 1., w recepturze „Praca z edytorami tekstu VIM i Nano".

## Jak to zrobić?

ARPing to narzędzie, które pozwala na wysyłanie żądań ARP i sprawdzanie, czy host docelowy jest aktywny i czy odpowiada na przesłane żądania. Do poprawnego działania program ARPing wymaga podania argumentu wywołania reprezentującego adres IP skanowanego hosta, tak jak to zostało zaprezentowane poniżej.

```
root@KaliLinux:~# arping 172.16.36.135 -c 1
ARPING 172.16.36.135
60 bytes from 00:0c:29:3d:84:32 (172.16.36.135): index=0 time=249.000 usec

--- 172.16.36.135 statistics ---
1 packets transmitted, 1 packets received, 0% unanswered (0 extra)
```

W przedstawionym przykładzie do adresu rozgłoszeniowego przesyłane jest pojedyncze żądanie ARP, zawierające żądanie przesłania fizycznego adresu hosta o adresie IP `172.16.36.135`. Jak łatwo zauważyć w wynikach działania, odpowiedź na przesłane żądanie nadesłał host o adresie MAC `00:0C:29:3D:84:32`. Program ARPing może być jeszcze bardziej efektywnie wykorzystywany do skanowania sieci na warstwie 2., jeżeli zostanie użyty w skrypcie powłoki *bash* skanującym wiele hostów jednocześnie. Aby skorzystać z tego narzędzia w skrypcie do skanowania wielu hostów, musimy najpierw znaleźć w wynikach działania unikatowy ciąg znaków, który pozwoli nam na odróżnienie sytuacji, gdy otrzymamy odpowiedź z aktywnego hosta, od tej, kiedy nie otrzymujemy żadnej odpowiedzi na wysłane żądanie. Aby znaleźć taki ciąg znaków, powinieneś wykonać polecenie `arping`, podając jako argument wywołania adres IP dowolnego nieaktywnego hosta, tak jak to zostało przedstawione poniżej.

```
root@KaliLinux:~# arping 172.16.36.136 -c 1
ARPING 172.16.36.136

--- 172.16.36.136 statistics ---
1 packets transmitted, 0 packets received, 100% unanswered (0 extra)
```

Analizując otrzymane odpowiedzi, łatwo zauważyć, że ciąg znaków `bytes from` pojawia się w wynikach działania polecenia `arping` tylko w sytuacji, kiedy badany adres IP należy do aktywnego hosta, a co więcej, w tym samym wierszu znajduje się również adres IP takiego hosta. Teraz, korzystając z polecenia `grep`, możemy łatwo wyodrębnić z wyników działania programu ARPing wiersze zawierające adresy IP aktywnych hostów, tak jak to zostało przedstawione w przykładzie poniżej.

```
root@KaliLinux:~# arping -c 1 172.16.36.135 | grep "bytes from"
60 bytes from 00:0c:29:3d:84:32 (172.16.36.135): index=0 time=291.000 usec
root@KaliLinux:~# arping -c 1 172.16.36.136 | grep "bytes from"
root@KaliLinux:~#
```

Zastosowanie polecenia grep do wyszukiwania w wynikach działania polecenia arping ciągu znaków bytes from pozwala na wyświetlenie wierszy zawierających adresy IP aktywnych hostów. Po wykonaniu takiej samej operacji dla nieaktywnego adresu IP nie zostają zwrócone żadne wyniki, co zostało zilustrowane za pomocą drugiego polecenia w przykładzie powyżej. Teraz, korzystając z polecenia cut z odpowiednio zdefiniowanym separatorem (opcja -d) i numerem pola (opcja -f), możemy łatwo wyodrębnić z tego ciągu znaków sam adres IP. Polecenie cut jest wykorzystywane w powłoce *bash* do dzielenia wierszy tekstu na elementy składowe w oparciu o podany separator i wyświetlania elementów o podanych numerach pól. Poprzez odpowiednie potokowanie poleceń powłoki możemy w łatwy sposób wyodrębnić z wyników działania polecenia arping adres MAC aktywnego hosta. Przykład takiego rozwiązania został pokazany poniżej.

```
root@KaliLinux:~# arping -c 1 172.16.36.135 | grep "bytes from"
60 bytes from 00:0c:29:3d:84:32 (172.16.36.135): index=0 time=10.000 usec
root@KaliLinux:~# arping -c 1 172.16.36.135 | grep "bytes from" | cut -d " " -f 4
00:0c:29:3d:84:32
```

W podobny sposób możemy bez problemu wyodrębnić adres IP aktywnego hosta; aby to zrobić, musimy tylko nieco zmodyfikować definicje separatorów i numery pól w wywołaniach polecenia cut:

```
root@KaliLinux:~# arping -c 1 172.16.36.135 | grep "bytes from"
60 bytes from 00:0c:29:3d:84:32 (172.16.36.135): index=0 time=328.000 usec
root@KaliLinux:~# arping -c 1 172.16.36.135 | grep "bytes from" | cut -d " " -f 5
(172.16.36.135):
root@KaliLinux:~# arping -c 1 172.16.36.135 | grep "bytes from" | cut -d " " -f 5 |
cut -d "(" -f 2
172.16.36.135):
root@KaliLinux:~# arping -c 1 172.16.36.135 | grep "bytes from" | cut -d " " -f 5 |
cut -d "(" -f 2 | cut -d ")" -f 1
172.16.36.135
```

Po znalezieniu sposobu na wyodrębnienie adresu IP z wyników działania polecenia arping możemy z łatwością zaimplementować takie rozwiązanie w pętli w skrypcie powłoki *bash*, który będzie skanował całą podsieć lokalną i wyświetlał adresy IP aktywnych hostów. Przykład takiego skryptu został przedstawiony na listingu poniżej.

```
#!/bin/bash

if [ "$#" -ne 1 ]; then
echo "Usage - ./arping.sh [interface]"
echo "Example - ./arping.sh eth0"
echo "Example will perform an ARP scan of the local subnet to which
eth0 is assigned"
exit
fi

interface=$1
prefix=$(ifconfig $interface | grep 'inet addr' | cut -d ':' -f 2 | cut -d ' ' -f 1 |
cut -d '.' -f 1-3)
```

```
for addr in $(seq 1 254); do
arping -c 1 $prefix.$addr | grep "bytes from" | cut -d " " -f 5 | cut -d "(" -f 2 |
cut -d ")" -f 1 &
done
```

W pierwszym wierszu naszego przykładowego skryptu zdefiniowana zostaje lokalizacja powłoki *bash*. Dalej znajdziesz blok kodu, którego zadaniem jest ustalenie, czy w wierszu wywołania skryptu został podany odpowiedni argument wywołania. Jest to realizowane poprzez proste sprawdzenie, czy liczba argumentów wywołania jest różna od 1. Jeżeli oczekiwany argument wywołania nie został podany, na ekranie wyświetlany jest opis sposobu użycia i skrypt kończy działanie. W opisie działania możemy znaleźć informację, że argumentem wywołania skryptu powinna być nazwa lokalnego interfejsu sieciowego. W kolejnym bloku kodu podany argument wywołania jest przypisywany do zmiennej o nazwie `interface`. Wartość tej zmiennej jest następnie wykorzystywana jako argument wywołania polecenia `ifconfig`, z którego wyników działania jest wyodrębniany prefiks adresu podsieci lokalnej. Na przykład jeżeli adres IP interfejsu sieciowego to `192.168.11.4`, do zmiennej `prefix` przypisany zostanie adres `192.168.11`. Następnie do generowania kolejnych wartości ostatniego oktetu adresu IP lokalnej podsieci `/24` wykorzystywana jest pętla `for`. Dla każdego z wygenerowanych adresów IP wywoływane jest polecenie `arping`. Wyniki działania tego polecenia są potokowane do polecenia `grep`, które „przepuszcza" dalej tylko wiersze zawierające ciąg znaków `bytes from` (a jak wiemy z naszych wcześniejszych rozważań, takie wiersze zawierają adresy IP aktywnych hostów w sieci). Wyniki działania polecenia `grep` są następnie potokowane przez serię poleceń `cut`, które wyodrębniają tylko adres IP. Zwróć uwagę, że blok poleceń pętli `for` zamiast średnikiem został zakończony znakiem `&`, dzięki czemu zadania realizowane w pętli będą wykonywane równolegle, a nie sekwencyjnie, co w drastyczny sposób wpłynie na zredukowanie czasu niezbędnego do przeskanowania podanego zakresu adresów IP. Poniżej przedstawiamy sposób wywołania naszego skryptu.

```
root@KaliLinux:~# ./arping.sh
Usage - ./arping.sh [interface]
Example - ./arping.sh eth0
Example will perform an ARP scan of the local subnet to which eth0 is assigned
root@KaliLinux:~# ./arping.sh eth0
172.16.36.1
172.16.36.2
172.16.36.132
172.16.36.135
172.16.36.254
```

W razie potrzeby możesz łatwo przekierować wyniki działania skryptu do pliku tekstowego na dysku. Aby to zrobić, powinieneś na końcu wiersza wywołania skryptu dodać znak większości (`>`) i nazwę pliku, w którym mają zostać zapisane dane. Przykład takiego rozwiązania został przedstawiony poniżej.

```
root@KaliLinux:~# ./arping.sh eth0 > output.txt
root@KaliLinux:~# ls output.txt
output.txt
root@KaliLinux:~# cat output.txt
```

```
172.16.36.1
172.16.36.2
172.16.36.132
172.16.36.135
172.16.36.254
```

Po zakończeniu działania skryptu i zapisaniu wyników działania w pliku możesz użyć polecenia `ls` do sprawdzenia, czy plik wynikowy został utworzony, a następnie za pomocą polecenia `cat` wyświetlić zawartość tego pliku. W razie potrzeby możesz łatwo zmodyfikować ten skrypt tak, aby wysyłał żądania ARP tylko do hostów, których adresy IP znajdują się w pliku tekstowym podanym jako argument wywołania. By to zrobić, musisz utworzyć plik tekstowy, w którym będzie zapisana lista adresów IP hostów do przeskanowania. Możesz również użyć tego samego pliku z listą adresów IP, z którego korzystaliśmy wcześniej w przykładzie ze skryptem Scapy, omawianym w poprzedniej recepturze.

```
#!/bin/bash

if [ "$#" -ne 1 ]; then
echo "Usage - ./arping.sh [input file]"
echo "Example - ./arping.sh iplist.txt"
echo "Example will perform an ARP scan of all IP addresses defined in iplist.txt"
exit
fi

file=$1

for addr in $(cat $file); do
arping -c 1 $addr | grep "bytes from" | cut -d " " -f 5 | cut -d "(" -f 2
| cut -d ")" -f 1 &
done
```

Jedyną większą różnicą pomiędzy tym skryptem a jego poprzednią wersją jest to, że zamiast nazwy interfejsu sieciowego podczas wywołania skryptu musisz podać nazwę pliku tekstowego zawierającego listę adresów IP do sprawdzenia. Argument wywołania skryptu jest przekazywany do zmiennej `file`. Następnie skrypt za pomocą pętli `for` przechodzi kolejno przez wszystkie adresy IP zapisane w pliku i dla każdego z nich wywołuje polecenie `arping`. Aby uruchomić skrypt, powinieneś w wierszu wywołania wpisać kropkę, prawy ukośnik i nazwę skryptu, tak jak to zostało przedstawione w przykładzie poniżej.

```
root@KaliLinux:~# ./arping.sh
Usage - ./arping.sh [input file]
Example - ./arping.sh iplist.txt
Example will perform an ARP scan of all IP addresses defined in iplist.txt
root@KaliLinux:~# ./arping.sh iplist.txt
172.16.36.1
172.16.36.2
172.16.36.132
172.16.36.135
172.16.36.254
```

Jeżeli skrypt zostanie uruchomiony bez żadnych argumentów, na ekranie zostanie wyświetlony krótki opis sposobu użycia, wskazujący, że argumentem wywołania powinien być plik tekstowy zawierający listę adresów IP. Po ponownym wywołaniu skryptu z właściwym argumentem skrypt rozpoczyna działanie i wyświetla na ekranie listę adresów IP aktywnych hostów w sieci lokalnej, wybranych z listy adresów IP zawartych w pliku. W razie potrzeby możesz bez trudu przekierować wyniki działania skryptu do pliku na dysku. Aby to zrobić, powinieneś na końcu wiersza wywołania skryptu dodać znak większości (>) i nazwę pliku, w którym mają zostać zapisane dane, tak jak to zostało przedstawione w przykładzie poniżej.

```
root@KaliLinux:~# ./arping.sh iplist.txt > output.txt
root@KaliLinux:~# ls output.txt
output.txt
root@KaliLinux:~# cat output.txt
172.16.36.1
172.16.36.2
172.16.36.132
172.16.36.135
172.16.36.254
```

Po zakończeniu działania skryptu i zapisaniu wyników działania w pliku możesz użyć polecenia `ls` do sprawdzenia, czy plik wynikowy został utworzony, a następnie za pomocą polecenia `cat` wyświetlić zawartość tego pliku.

## Jak to działa?

Program ARPing został napisany z myślą o sprawdzaniu, czy dany host w sieci LAN jest aktywny. Szybko okazało się jednak, że ze względu na prostotę i wygodę użycia można go w bardzo efektywny sposób wykorzystywać w skryptach powłoki *bash* do sekwencyjnego skanowania wielu hostów. Można tego dokonać poprzez przechodzenie w pętli przez kolejne adresy IP z badanego zakresu i przekazywanie ich jako argumenty wywołania programu ARPing.

# Skanowanie sieci na warstwie 2. przy użyciu programu Nmap

**Nmap** (ang. *Network Mapper*) to jedno z najbardziej funkcjonalnych i efektywnych narzędzi dostępnych w systemie Kali Linux. Program może być wykorzystywany do skanowania dużych sieci przy użyciu wielu różnych technik skanowania, które można w bardzo elastyczny sposób konfigurować i dostosowywać do własnych potrzeb. W dalszej części naszej książki będziemy bardzo często korzystać z tego skanera, a teraz pokażemy, w jaki sposób możesz używać go do skanowania sieci na warstwie 2.

## Przygotuj się

Aby można było za pomocą pakietu Nmap przeprowadzić skanowanie na warstwie 2., w sieci lokalnej musi działać przynajmniej jeden system, który będzie odpowiadał na żądania ARP. W omawianych przykładach wykorzystujemy kombinację systemów Linux i Windows. Więcej szczegółowych informacji na temat instalacji i konfiguracji tych systemów w naszym środowisku testowym znajdziesz w rozdziale 1., w recepturach „Instalacja systemu Metasploitable2" oraz „Instalacja systemu Windows".

## Jak to zrobić?

Zastosowanie skanera Nmap to kolejny sposób na przeprowadzenie skanowania sieci na warstwie 2. przy użyciu pojedynczego polecenia wykonywanego z poziomu wiersza poleceń konsoli systemu. Opcja `-sn` w dokumentacji programu Nmap jest opisana jako *Ping Scan*. Choć taki opis nieodparcie nasuwa skojarzenia z wykrywaniem hostów na warstwie 3., to jednak jest to skan adaptacyjny. Jeżeli argumentem wywołania Nmapa będą adresy IP z lokalnej podsieci, to skanowanie na warstwie 2. możesz przeprowadzić w następujący sposób:

```
root@KaliLinux:~# nmap 172.16.36.135 -sn
Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-16 15:40 EST
Nmap scan report for 172.16.36.135
Host is up (0.00038s latency).
MAC Address: 00:0C:29:3D:84:32 (VMware)
Nmap done: 1 IP address (1 host up) scanned in 0.17 seconds
```

Wykonanie takiego polecenia spowoduje wysłanie żądania ARP na adres rozgłoszeniowy sieci LAN i określenie na podstawie otrzymanej odpowiedzi, czy host o podanym adresie IP jest aktywny. W przypadku przedstawionym powyżej host jest aktywny i odpowiada na przesłane żądanie. Jeżeli argumentem wywołania programu będzie adres IP hosta, który nie jest aktywny, Nmap wyświetli na ekranie odpowiedni komunikat, tak jak to zostało zaprezentowane poniżej.

```
root@KaliLinux:~# nmap 172.16.36.136 -sn
Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-16 15:51 EST
Note: Host seems down. If it is really up, but blocking our ping probes, try -Pn
Nmap done: 1 IP address (0 hosts up) scanned in 0.41 seconds
```

Jeżeli chcesz skanować na warstwie 2. większą liczbę hostów, powinieneś w wierszu wywołania podać zakres adresów IP, używając notacji z myślnikiem. Przykładowo aby przeskanować całą podsieć /24, powinieneś w miejsce ostatniego oktetu adresu wpisać zakres `0-255`.

```
root@KaliLinux:~# nmap 172.16.36.0-255 -sn

Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-11 05:35 EST
Nmap scan report for 172.16.36.1
Host is up (0.00027s latency).
MAC Address: 00:50:56:C0:00:08 (VMware)
Nmap scan report for 172.16.36.2
```

```
Host is up (0.00032s latency).
MAC Address: 00:50:56:FF:2A:8E (VMware)
Nmap scan report for 172.16.36.132
Host is up.
Nmap scan report for 172.16.36.135
Host is up (0.00051s latency).
MAC Address: 00:0C:29:3D:84:32 (VMware)
Nmap scan report for 172.16.36.200
Host is up (0.00026s latency).
MAC Address: 00:0C:29:23:71:62 (VMware)
Nmap scan report for 172.16.36.254
Host is up (0.00015s latency).
MAC Address: 00:50:56:EA:54:3A (VMware)
Nmap done: 256 IP addresses (6 hosts up) scanned in 3.22 seconds
```

Wykonanie takiego polecenia spowoduje wysyłanie żądań ARP dla wszystkich hostów z podanego zakresu i wyświetlenie na podstawie otrzymanych odpowiedzi listy hostów, które są aktywne. W razie potrzeby zamiast z zakresu adresów IP możesz skorzystać z listy adresów zapisanej w pliku tekstowym. Aby to zrobić, powinieneś w wierszu wywołania skanera użyć opcji `-iL`, tak jak to zostało przedstawione poniżej.

```
root@KaliLinux:~# nmap -iL iplist.txt -sn

Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-16 16:07 EST
Nmap scan report for 172.16.36.2
Host is up (0.00026s latency).
MAC Address: 00:50:56:FF:2A:8E (VMware)
Nmap scan report for 172.16.36.1
Host is up (0.00021s latency).
MAC Address: 00:50:56:C0:00:08 (VMware)
Nmap scan report for 172.16.36.132
Host is up (0.00031s latency).
MAC Address: 00:0C:29:65:FC:D2 (VMware)
Nmap scan report for 172.16.36.135
Host is up (0.00014s latency).
MAC Address: 00:0C:29:3D:84:32 (VMware)
Nmap scan report for 172.16.36.180
Host is up.
Nmap scan report for 172.16.36.254
Host is up (0.00024s latency).
MAC Address: 00:50:56:EF:B9:9C (VMware)
Nmap done: 8 IP addresses (6 hosts up) scanned in 0.41 seconds
```

Kiedy zostanie użyta opcja `-sn`, Nmap najpierw próbuje wykryć hosta za pomocą żądania ARP na warstwie 2. i przechodzi do wysyłania na warstwie 3. żądań ICMP tylko wtedy, kiedy sprawdzany host nie jest zlokalizowany w sieci LAN. Zwróć uwagę, że w wynikach skanowania przeprowadzonego w podsieci lokalnej (podsieć prywatna `172.16.36.0/24`) wyświetlane są adresy

MAC aktywnych hostów. Dzieje się tak dlatego, że adresy MAC aktywnych hostów znajdują się w odpowiedziach na żądania ARP. Jeżeli jednak spróbujesz przeprowadzić taki sam skan dla wielu hostów znajdujących się w innej sieci lokalnej, to otrzymane wyniki nie będą zawierały adresów MAC, co zostało pokazane poniżej.

```
root@KaliLinux:~# nmap -sn 74.125.21.0-255

Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-11 05:42 EST
Nmap scan report for 74.125.21.0
Host is up (0.0024s latency).
Nmap scan report for 74.125.21.1
Host is up (0.00017s latency).
Nmap scan report for 74.125.21.2
Host is up (0.00028s latency).
Nmap scan report for 74.125.21.3
Host is up (0.00017s latency).
```

Jeżeli spróbujesz w ten sposób przeprowadzić skanowanie zdalnej podsieci (na przykład podsieci publicznej `74.125.21.0/24`), przekonasz się, że Nmap użyje warstwy 3. i adresy MAC nie zostaną wyświetlone. Takie zachowanie pokazuje, że kiedy to możliwe, Nmap wykorzystuje szybkie wykrywanie hostów na warstwie 2., ale w razie potrzeby automatycznie przechodzi do skanowania na warstwie 3. i używa routowalnego protokołu ICMP. Możesz to zaobserwować, posługując się programem Wireshark do monitorowania ruchu sieciowego generowanego przez Nmapa podczas wykonywania takiego skanu. Na rysunku zamieszczonym poniżej możesz zobaczyć, jak Nmap wykorzystuje żądania ARP do wykrywania hostów w lokalnym segmencie sieci.

| No. | Destination | Protocol | Info |
|---|---|---|---|
| 498 | Broadcast | ARP | Who has 172.16.36.102? Tell 172.16.36.232 |
| 499 | Broadcast | ARP | Who has 172.16.36.125? Tell 172.16.36.232 |
| 500 | Broadcast | ARP | Who has 172.16.36.163? Tell 172.16.36.232 |
| 501 | Broadcast | ARP | Who has 172.16.36.164? Tell 172.16.36.232 |
| 502 | Broadcast | ARP | Who has 172.16.36.196? Tell 172.16.36.232 |
| 503 | Broadcast | ARP | Who has 172.16.36.31? Tell 172.16.36.232 |

## Jak to działa?

Nmap to bardzo rozbudowany i funkcjonalny pakiet, dzięki któremu możesz szybko i wygodnie przeprowadzić wykrywanie hostów w sieci. Ogólna zasada jest prosta. Nmap rozsyła żądania ARP dla poszczególnych hostów na adres rozgłoszeniowy sieci i na podstawie otrzymanych odpowiedzi określa, czy dany host jest aktywny.

# Skanowanie sieci na warstwie 2. przy użyciu programu NetDiscover

**NetDiscover** to narzędzie, którego można używać do wykrywania hostów w sieci za pomocą aktywnej lub pasywnej analizy z wykorzystaniem żądań ARP. Program powstał z myślą o sieciach bezprzewodowych, ale z powodzeniem można go używać w sieciach kablowych. W tej recepturze pokażemy, w jaki sposób możesz korzystać z tego programu zarówno do aktywnego, jak i pasywnego skanowania sieci.

## Przygotuj się

Aby można było za pomocą pakietu NetDiscover przeprowadzić skanowanie na warstwie 2., w sieci lokalnej musi działać przynajmniej jeden system, który będzie odpowiadał na żądania ARP. W przedstawionych przykładach wykorzystujemy kombinację systemów Linux i Windows. Więcej szczegółowych informacji na temat instalacji i konfiguracji tych systemów w naszym środowisku testowym znajdziesz w rozdziale 1., w recepturach „Instalacja systemu Metasploitable2" oraz „Instalacja systemu Windows".

## Jak to zrobić?

NetDiscover to narzędzie, które zostało specjalnie zaprojektowane do wykrywania hostów za pomocą skanowania na warstwie 2. Program może być wykorzystywany do skanowania całych zakresów adresów IP, podawanych w notacji CIDR jako argument wywołania (opcja `-r`). Wynikiem działania programu jest tabela zawierająca w poszczególnych kolumnach kolejno: adres IP aktywnego hosta, odpowiadający mu adres MAC, liczbę otrzymanych odpowiedzi, rozmiar odpowiedzi oraz nazwę producenta karty sieciowej (dekodowaną w oparciu o adres MAC).

```
root@KaliLinux:~# netdiscover -r 172.16.36.0/24
Currently scanning: Finished!    |    Screen View: Unique Hosts

5 Captured ARP Req/Rep packets, from 5 hosts. Total size: 300

_____________________________________________________________________________
IP            At MAC Address      Count  Len    MAC Vendor
-----------------------------------------------------------------------------
172.16.36.1   00:50:56:c0:00:08   01     060    VMWare, Inc.
172.16.36.2   00:50:56:ff:2a:8e   01     060    VMWare, Inc.
172.16.36.132 00:0c:29:65:fc:d2   01     060    VMware, Inc.
172.16.36.135 00:0c:29:3d:84:32   01     060    VMware, Inc.
172.16.36.254 00:50:56:ef:b9:9c   01     060    VMWare, Inc.
```

NetDiscover może być również używany do skanowania adresów IP znajdujących się na liście zapisanej w pliku tekstowym na dysku. W takiej sytuacji zamiast podawać w wierszu wywołania zakresy adresów IP w formacie CIDR, możesz użyć opcji `-l` i podać nazwę pliku zawierającego listę adresów IP, tak jak to zostało przedstawione na listingu poniżej.

```
root@KaliLinux:~# netdiscover -l iplist.txt
Currently scanning: 172.16.36.0/24   |   Screen View: Unique Hosts

39 Captured ARP Req/Rep packets, from 5 hosts. Total size: 2340

_____________________________________________________________________________
IP            At MAC Address      Count  Len    MAC Vendor
-----------------------------------------------------------------------------
172.16.36.1   00:50:56:c0:00:08   08     480    VMWare, Inc.
172.16.36.2   00:50:56:ff:2a:8e   08     480    VMWare, Inc.
172.16.36.132 00:0c:29:65:fc:d2   08     480    VMware, Inc.
172.16.36.135 00:0c:29:3d:84:32   08     480    VMware, Inc.
172.16.36.254 00:50:56:ef:b9:9c   07     420    VMWare, Inc.
```

Kolejną, unikatową funkcją, która znacząco wyróżnia ten program spośród innych, jest możliwość przeprowadzania pasywnego wykrywania hostów w sieci. Rozsyłanie żądań ARP do wszystkich adresów IP w całej podsieci może czasami niepotrzebnie wzbudzić uwagę i spowodować wygenerowanie alarmów przez urządzenia zabezpieczające, takie jak **systemy IDS** czy **IPS** (ang. *Intrusion Detection System*, *Intrusion Prevention System* — systemy wykrywania i zapobiegania włamaniom). Znacznie mniej rzucającym się w oczy rozwiązaniem będzie w takiej sytuacji pasywne nasłuchiwanie ruchu sieciowego, analizowanie przesyłanych w sieci żądań i odpowiedzi ARP, a następnie budowanie na ich podstawie listy aktywnych hostów. Aby skorzystać z techniki pasywnego skanowania, powinieneś w wierszu wywołania programu umieścić opcję `-p`, tak jak to zostało przedstawione na listingu poniżej.

```
root@KaliLinux:~# netdiscover -p

Currently scanning: (passive)   |   Screen View: Unique Hosts

4 Captured ARP Req/Rep packets, from 2 hosts. Total size: 240

_____________________________________________________________________________
IP            At MAC Address      Count  Len    MAC Vendor
-----------------------------------------------------------------------------
172.16.36.132 00:0c:29:65:fc:d2   02     120    VMware, Inc.
172.16.36.135 00:0c:29:3d:84:32   02     120    VMware, Inc.
```

Przy użyciu takiej techniki skanowania interesujące nas informacje mogą być gromadzone znacznie wolniej, ponieważ opiera się ona na analizie pakietów przesyłanych w sieci w ramach normalnych operacji sieciowych, ale z drugiej strony praktycznie gwarantuje, że nie wzbudzi niechcianego zainteresowania. Skanowanie pasywne jest znacznie bardziej efektywne w sieciach bezprzewodowych, ponieważ karta sieciowa działająca w trybie monitorowania (ang. *promiscuous mode*) będzie otrzymywała odpowiedzi na żądania ARP przeznaczone dla innych urządzeń sieciowych. Aby NetDiscover pracował efektywnie w sieciach przełączanych, musisz mieć dostęp do urządzeń sieciowych z portami SPAN (ang. *Switch Port Analyzer Network*), urządzeń TAP (ang. *Testing Access Point*) lub po prostu mieć możliwość przeładowania tablic CAM (ang. *Content Addressable Memory*) w przełącznikach sieciowych, co spowoduje, że ruch sieciowy będzie rozsyłany na wszystkie porty przełącznika.

## Jak to działa?

Ogólna zasada wykrywania hostów w sieci za pomocą programu NetDiscover nie różni się niczym od innych, opisywanych wcześniej sposobów skanowania sieci na warstwie 2. z wykorzystaniem protokołu ARP. Główną różnicą pomiędzy programem NetDiscover a innymi narzędziami jest to, że NetDiscover dysponuje pasywnym mechanizmem skanowania, a w wynikach działania programu oprócz adresów IP pojawiają się również adresy MAC. Co prawda należy zaznaczyć, że w większości przypadków w kablowych sieciach przełączanych tryb pasywny jest praktycznie bezużyteczny, ponieważ w takich sieciach przechwycenie odpowiedzi na żądania ARP wymaga pewnych interakcji z wykrywanymi klientami (niezależnych od programu NetDiscover). Warto jednak pamiętać, że taki mechanizm jest do dyspozycji i że będzie znakomicie sprawdzał się w sieciach rozgłoszeniowych (opartych na hubach sieciowych) oraz w sieciach bezprzewodowych. NetDiscover rozpoznaje markę kart sieciowych poprzez analizę pierwszych trzech oktetów (pierwszych 24 bitów) adresu MAC, które identyfikują producenta i często są dobrą wskazówką, pozwalającą na rozpoznanie danego urządzenia sieciowego.

# Skanowanie sieci na warstwie 2. przy użyciu programu Metasploit

Metasploit to rozbudowane narzędzie, którego podstawowym przeznaczeniem jest wspomaganie przeprowadzania testów penetracyjnych (pakiet Metasploit będziemy bardziej szczegółowo omawiać w kolejnych rozdziałach książki). Warto jednak zauważyć, że oprócz tego pakiet Metasploit ma również wiele pomocniczych modułów, które pozwalają na skanowanie sieci i zbieranie informacji o działających w niej hostach. Jeden z takich modułów umożliwia skanowanie lokalnej podsieci z wykorzystaniem protokołu ARP. Jest to bardzo wygodne rozwiązanie szczególnie dla pentesterów, ponieważ zintegrowanie takiego skanera z Metasploitem pozwala na zredukowanie liczby narzędzi używanych podczas przeprowadzania testów penetracyjnych. W tej recepturze pokażemy, w jaki sposób możesz używać Metasploita do wykrywania hostów w sieci z wykorzystaniem protokołu ARP.

## Przygotuj się

Aby można było za pomocą pakietu Metasploit przeprowadzić skanowanie na warstwie 2., w sieci lokalnej musi działać przynajmniej jeden system, który będzie odpowiadał na żądania ARP. W przedstawionym przykładzie wykorzystujemy kombinację systemów Linux i Windows. Więcej szczegółowych informacji na temat instalacji i konfiguracji tych systemów w naszym środowisku testowym znajdziesz w rozdziale 1., w recepturach „Instalacja systemu Metasploitable2” oraz „Instalacja systemu Windows”.

## Jak to zrobić?

Choć pakiet Metasploit kojarzy się głównie z przeprowadzaniem testów penetracyjnych, to jednak można w nim znaleźć bardzo wiele różnych modułów pomocniczych, pozwalających na skanowanie sieci i zbieranie informacji o działających w niej hostach. W szczególności jeden z takich modułów może być wykorzystywany do wykrywania hostów w sieci za pomocą skanowania na warstwie 2. Aby uruchomić pakiet Metasploit, powinieneś z poziomu wiersza poleceń konsoli systemu wykonać polecenie `msfconsole`. Następnie za pomocą polecenia `use` musisz wybrać moduł, którego będziesz używać do skanowania sieci.

```
root@KaliLinux:~# msfconsole
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMM                  MMMMMMMMMM
MMMN$                           vMMMM
MMMNl  MMMMM             MMMMM  JMMMM
MMMNl  MMMMMMMN       NMMMMMMM  JMMMM
MMMNl  MMMMMMMMMNmmmNMMMMMMMM  JMMMM
MMMNI  MMMMMMMMMMMMMMMMMMMMMM  jMMMM
MMMNI  MMMMMMMMMMMMMMMMMMMMMM  jMMMM
MMMNI  MMMMM   MMMMMMM   MMMMM  jMMMM
MMMNI  MMMMM   MMMMMMM   MMMMM  jMMMM
MMMNI  MMMNM   MMMMMMM   MMMMM  jMMMM
MMMNI  WMMMM   MMMMMMM   MMMM#  JMMMM
MMMMR  ?MMNM             MMMMM .dMMMM
MMMMNm `?MMM             MMMM` dMMMMM
MMMMMMN  ?MM               MM? NMMMMMN
MMMMMMMMNe                   JMMMMMNMMM
MMMMMMMMMNm,                 eMMMMMNMMNMM
MMMMNNMNMMMMMNx         MMMMMMNMMNMMNM
MMMMMMMMNMMNMMMMm+..+MMNMMNMNMMNMMNMM
        http://metasploit.pro

Frustrated with proxy pivoting? Upgrade to layer-2 VPN pivoting with Metasploit
Pro—type 'go_pro' to launch it now.
       =[ metasploit v4.6.0-dev [core:4.6 api:1.0]
+ -- --=[ 1053 exploits - 590 auxiliary - 174 post
+ -- --=[ 275 payloads - 28 encoders - 8 nops

msf > use auxiliary/scanner/discovery/arp_sweep
msf auxiliary(arp_sweep) >
```

Po wybraniu żądanego modułu możesz użyć polecenia `show options` do wyświetlenia listy dostępnych opcji, tak jak to zostało przedstawione na listingu poniżej.

```
msf auxiliary(arp_sweep) > show options

Module options (auxiliary/scanner/discovery/arp_sweep):

   Name        Current Setting   Required   Description
   ----        ---------------   --------   -----------
```

```
   INTERFACE                    no        The name of the interface
   RHOSTS                       yes       The target address range or CIDR identifier
   SHOST                        no        Source IP Address
   SMAC                         no        Source MAC Address
   THREADS     1                yes       The number of concurrent threads
   TIMEOUT     5                yes       The number of seconds to wait for new
data
```

Na liście znajdziesz opcje pozwalające na wprowadzenie danych o celu, systemie źródłowym oraz innych ustawieniach skanowania. Większość niezbędnych informacji możesz uzyskać, wyświetlając konfigurację interfejsu sieciowego systemu skanującego. Na szczęście konsola pakietu Metasploit Framework pozwala na przekazywanie polecenia bezpośrednio do powłoki systemu operacyjnego. W przykładzie przedstawionym poniżej bez opuszczania konsoli pakietu Metasploit Framework wykonywane jest polecenie `ifconfig`, które po przekazaniu do powłoki systemu powoduje wyświetlenie konfiguracji interfejsów sieciowych.

```
msf auxiliary(arp_sweep) > ifconfig eth1
[*] exec: ifconfig eth1

eth1      Link encap:Ethernet HWaddr 00:0c:29:09:c3:79
          inet addr:172.16.36.180 Bcast:172.16.36.255 Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe09:c379/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:1576971 errors:1 dropped:0 overruns:0 frame:0
          TX packets:1157669 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:226795966 (216.2 MiB) TX bytes:109929055 (104.8 MiB)
          Interrupt:19 Base address:0x2080
```

Do przeprowadzenia skanowania użyjemy interfejsu `eth1`. Ze względu na fakt, że skanowanie na warstwie 2. pozwala na wykrywanie hostów wyłącznie w podsieci lokalnej, musimy najpierw sprawdzić adres IP i maskę podsieci systemu skanującego. W naszym przypadku adres IP i maska wskazują, że powinniśmy przeprowadzić skanowanie podsieci `172.16.36.0/24`. Oprócz tego w wynikach działania polecenia `ifconfig` możemy znaleźć adres MAC naszego systemu. Aby ustawić opcje konfiguracyjne wybranego modułu w pakiecie Metasploit, powinieneś skorzystać z polecenia `set`, po którym następuje nazwa opcji i przypisywana jej wartość, tak jak to zostało przedstawione na listingu poniżej.

```
msf auxiliary(arp_sweep) > set interface eth1
interface => eth1
msf auxiliary(arp_sweep) > set RHOSTS 172.16.36.0/24
RHOSTS => 172.16.36.0/24
msf auxiliary(arp_sweep) > set SHOST 172.16.36.180
SHOST => 172.16.36.180
msf auxiliary(arp_sweep) > set SMAC 00:0c:29:09:c3:79
SMAC => 00:0c:29:09:c3:79
msf auxiliary(arp_sweep) > set THREADS 20
THREADS => 20
msf auxiliary(arp_sweep) > set TIMEOUT 1
TIMEOUT => 1
```

Po zakończeniu ustawiania opcji konfiguracyjnych modułu możesz za pomocą polecenia `show options` ponownie wyświetlić dostępny zestaw opcji. Tym razem w wynikach działania tego polecenia powinieneś zobaczyć nowe wartości ustawionych przed chwilą opcji.

```
msf auxiliary(arp_sweep) > show options

Module options (auxiliary/scanner/discovery/arp_sweep):

   Name       Current Setting   Required   Description
   ----       ---------------   --------   -----------
   INTERFACE  eth1              no         The name of the interface
   RHOSTS     172.16.36.0/24    yes        The target address range or CIDR identifier
   SHOST      172.16.36.180     no         Source IP Address
   SMAC       00:0c:29:09:c3:79 no         Source MAC Address
   THREADS    20                yes        The number of concurrent threads
   TIMEOUT    1                 yes        The number of seconds to wait for new data
```

Po sprawdzeniu, czy wartości poszczególnych opcji są poprawne, możesz za pomocą polecenia `run` uruchomić skanowanie. W naszym przykładzie wybrany moduł powoduje wyświetlenie na ekranie krótkich informacji o hostach wykrytych za pomocą skanowania ARP. Oprócz adresu IP wykrytego hosta wyświetlana jest nazwa producenta jego karty sieciowej (ang. *NIC Vendor*; *Network Interface Card Vendor*), zakodowana w trzech pierwszych bajtach adresów MAC wykrytych hostów, co zostało pokazane na listingu poniżej.

```
msf auxiliary(arp_sweep) > run

[*] 172.16.36.1 appears to be up (VMware, Inc.).
[*] 172.16.36.2 appears to be up (VMware, Inc.).
[*] 172.16.36.132 appears to be up (VMware, Inc.).
[*] 172.16.36.135 appears to be up (VMware, Inc.).
[*] 172.16.36.254 appears to be up (VMware, Inc.).
[*] Scanned 256 of 256 hosts (100% complete)
[*] Auxiliary module execution completed
```

## Jak to działa?

I znów, podobnie jak poprzednio, ogólne zasady przeprowadzania w programie Metasploit skanowania z wykorzystaniem protokołu ARP pozostają niezmienione. Program rozsyła wiele żądań ARP i na podstawie odebranych odpowiedzi dokonuje identyfikacji aktywnych hostów. W wynikach działania przedstawionego w tej recepturze modułu pakietu Metasploit możesz znaleźć listę adresów IP wykrytych hostów oraz nazwy producentów ich kart sieciowych (umieszczone w nawiasach).

# Skanowanie sieci na warstwie 3. przy użyciu polecenia ping (ICMP)

Skanowanie na warstwie 3. jest chyba najbardziej rozpowszechnioną wśród administratorów i personelu technicznego IT metodą wykrywania hostów działających w sieci. To właśnie na warstwie 3. działa słynne polecenie `ping`, wykorzystujące do wykrywania aktywnych hostów protokół ICMP. W tej recepturze pokażemy, w jaki sposób możesz używać tego polecenia do wykrywania hostów w sieci.

## Przygotuj się

Zastosowanie polecenia `ping` do wykrywania hostów na warstwie 3. nie wymaga żadnego środowiska testowego, ponieważ bardzo wiele systemów działających w sieci Internet domyślnie odpowiada na żądania *ICMP Echo*. Z drugiej jednak strony, jeżeli nie znasz dokładnie przepisów prawa i regulacji, którym podlegasz w tym zakresie, rekomendowanym rozwiązaniem jest przeprowadzanie skanowania wyłącznie we własnym, dedykowanym środowisku testowym. Aby można było za pomocą polecenia `ping` przeprowadzić skanowanie na warstwie 3., w testowanej sieci musi działać przynajmniej jeden system, który będzie odpowiadał na żądania *ICMP Echo*. W przedstawionym przykładzie wykorzystujemy kombinację systemów Linux i Windows. Więcej szczegółowych informacji na temat instalacji i konfiguracji tych systemów w naszym środowisku testowym znajdziesz w rozdziale 1., w recepturach „Instalacja systemu Metasploitable2" oraz „Instalacja systemu Windows". Oprócz tego w tej recepturze będziemy używać edytorów tekstu, takich jak VIM czy Nano, do napisania skryptów skanujących i zapisania ich w systemie plików. Więcej szczegółowych informacji na temat pisania skryptów znajdziesz w rozdziale 1., w recepturze „Praca z edytorami tekstu VIM i Nano".

## Jak to zrobić?

Zapewne zdecydowana większość użytkowników komputerów (a zwłaszcza Ci, którzy pracują w branży IT) miała okazję korzystać z polecenia `ping`. Aby przy użyciu tego polecenia sprawdzić, czy dany host jest aktywny, powinieneś jako argument wywołania polecenia podać adres IP testowanego hosta, tak jak to zostało przedstawione na listingu poniżej:

```
root@KaliLinux:~# ping 172.16.36.135
PING 172.16.36.135 (172.16.36.135) 56(84) bytes of data.
64 bytes from 172.16.36.135: icmp_req=1 ttl=64 time=1.35 ms
64 bytes from 172.16.36.135: icmp_req=2 ttl=64 time=0.707 ms
64 bytes from 172.16.36.135: icmp_req=3 ttl=64 time=0.369 ms
^C
--- 172.16.36.135 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.369/0.809/1.353/0.409 ms
```

Po wykonaniu tego polecenia do podanego adresu IP zostanie wysłane żądanie *ICMP Echo*. Aby jednak zdalny host odpowiedział na takie żądanie, spełnionych musi być kilka warunków, które zostały przedstawione poniżej:

1. Podany adres IP musi być przypisany do badanego hosta.
2. Zdalny host musi być włączony i podłączony do sieci.
3. Pomiędzy systemem skanującym a badanym hostem musi istnieć połączenie sieciowe.
4. Badany host musi być skonfigurowany tak, aby odpowiadać na nadchodzące żądania ICMP.
5. Na całej trasie połączenia pomiędzy systemem skanującym a badanym hostem nie może być żadnego urządzenia sieciowego, które zostało skonfigurowane do blokowania ruchu ICMP.

Jak widać, pomyślne wykrywanie hostów za pomocą żądań ICMP jest uzależnione od wielu zmiennych czynników. Z tego względu taki rodzaj skanowania nie zawsze jest miarodajny, ale za to w przeciwieństwie do ARP, protokół ICMP jest routowalny i dzięki temu może być używany do wykrywania hostów zlokalizowanych poza siecią LAN. Zauważ, że w wynikach działania polecenia `ping`, przedstawionych powyżej, pojawia się ciąg znaków `^C`, który oznacza, że do zatrzymania pracy tego polecenia została użyta kombinacja klawiszy *Ctrl+C*. W przeciwieństwie do swojego odpowiednika zaimplementowanego w systemie Windows, linuksowa wersja polecenia `ping` domyślnie wysyła żądania ICMP do badanego hosta w niekończącej się pętli (czyli w praktyce aż do momentu, w którym użytkownik przerwie działanie tego polecenia). Jeżeli jednak chciałbyś ograniczyć liczbę wysyłanych żądań ICMP, powinieneś w wierszu wywołania polecenia `ping` dodać opcję `-c`. Dzięki zastosowaniu tej opcji po wysłaniu określonej liczby żądań ICMP polecenie `ping` automatycznie zakończy działanie. Przykład takiego polecenia został przedstawiony poniżej.

```
root@KaliLinux:~# ping 172.16.36.135 -c 2
PING 172.16.36.135 (172.16.36.135) 56(84) bytes of data.
64 bytes from 172.16.36.135: icmp_req=1 ttl=64 time=0.611 ms
64 bytes from 172.16.36.135: icmp_req=2 ttl=64 time=0.395 ms

--- 172.16.36.135 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 0.395/0.503/0.611/0.108 ms
```

Jeżeli chcesz przeprowadzić skanowanie na warstwie 3. większej liczby adresów IP, polecenia `ping` możesz użyć w skrypcie powłoki, podobnie jak to robiliśmy w przypadku polecenia `arping`. Aby napisać taki skrypt, musimy sprawdzić, jakie są wyniki działania polecenia `ping` dla aktywnych i nieaktywnych hostów. W tym celu najpierw wyślemy „ping" do hosta, o którym wiemy, że na pewno jest aktywny i podłączony do sieci, a następnie wyślemy kolejnego „pinga", tym razem do nieaktywnego adresu IP. Cały proces został przedstawiony poniżej.

```
root@KaliLinux:~# ping 74.125.137.147 -c 1
PING 74.125.137.147 (74.125.137.147) 56(84) bytes of data.
64 bytes from 74.125.137.147: icmp_seq=1 ttl=128 time=31.3 ms
```

```
--- 74.125.137.147 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 31.363/31.363/31.363/0.000 ms
root@KaliLinux:~# ping 83.166.169.231 -c 1
PING 83.166.169.231 (83.166.169.231) 56(84) bytes of data.

--- 83.166.169.231 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

Podobnie jak to miało miejsce w przypadku polecenia `arping`, unikatowy ciąg znaków `bytes from` pojawia się w wynikach działania polecenia `ping` tylko w sytuacji, kiedy badany adres IP należy do aktywnego hosta, a co więcej, w tym samym wierszu znajduje się również adres IP takiego hosta. Teraz korzystając z kombinacji poleceń `grep` i `cut`, możemy łatwo wyodrębnić z wyników działania polecenia `ping` adresy IP aktywnych hostów, tak jak to zostało przedstawione w przykładzie poniżej.

```
root@KaliLinux:~# ping 74.125.137.147 -c 1 | grep "bytes from"
64 bytes from 74.125.137.147: icmp_seq=1 ttl=128 time=37.2 ms
root@KaliLinux:~# ping 74.125.137.147 -c 1 | grep "bytes from" | cut -d " " -f 4
74.125.137.147:
root@KaliLinux:~# ping 74.125.137.147 -c 1 | grep "bytes from" | cut -d " " -f 4
| cut -d ":" -f 1
74.125.137.147
```

Umieszczając taką sekwencję zadań w pętli przetwarzającej cały zakres adresów IP, możemy szybko zidentyfikować aktywne hosty działające w danej podsieci. Wynikiem działania skryptu będzie lista adresów IP aktywnych hostów. Przykład skryptu powłoki *bash* realizującego takie zadanie został przedstawiony poniżej.

```
#!/bin/bash

if [ "$#" -ne 1 ]; then
echo "Usage - ./ping_sweep.sh [/24 network address]"
echo "Example - ./ping_sweep.sh 172.16.36.0"
echo " Example will perform an ICMP ping sweep of the 172.16.36.0/24
network"
exit
fi

prefix=$(echo $1 | cut -d '.' -f 1-3)

for addr in $(seq 1 254); do
ping -c 1 $prefix.$addr | grep "bytes from" | cut -d " " -f 4 | cut -d ":" -f 1 &
done
```

W pierwszym wierszu przedstawionego skryptu zdefiniowana zostaje lokalizacja interpretera powłoki *bash*. Dalej następuje blok kodu, który ustala, czy w wierszu wywołania skryptu został podany odpowiedni argument wywołania. Jest to realizowane poprzez proste sprawdzenie, czy liczba argumentów wywołania jest różna od 1. Jeżeli oczekiwany argument wywołania nie

został podany, na ekranie wyświetlany jest opis sposobu użycia i skrypt kończy działanie. W opisie działania możemy znaleźć informację, że argumentem wywołania skryptu powinien być adres podsieci /24. W kolejnym wierszu kodu z podanego argumentu wyodrębniany jest prefiks adresu podsieci. Na przykład jeżeli argumentem wywołania skryptu był adres 192.168.11.0, do zmiennej prefix zostanie przypisana wartość 192.168.11. Następnie do generowania kolejnych wartości ostatniego oktetu adresu IP lokalnej podsieci /24 wykorzystywana jest pętla for. Dla każdego z wygenerowanych adresów IP wywoływane jest polecenie arping. Wyniki działania tego polecenia są potokowane do polecenia grep, które „przepuszcza" dalej tylko wiersze zawierające ciąg znaków bytes from (a jak wiemy z naszych wcześniejszych rozważań, takie wiersze zawierają adresy IP aktywnych hostów w sieci). Wyniki działania polecenia grep są następnie potokowane przez serię poleceń cut, które wyodrębniają tylko adres IP. Zwróć uwagę, że blok poleceń pętli for zamiast średnikiem został zakończony znakiem &, dzięki czemu zadania realizowane w pętli będą wykonywane równolegle, a nie sekwencyjnie, co w drastyczny sposób wpłynie na zredukowanie czasu niezbędnego do przeskanowania podanego zakresu adresów IP. Gotowy skrypt możesz uruchomić z poziomu okna terminala, wpisując w wierszu wywołania kropkę, prawy ukośnik i nazwę skryptu, tak jak to zostało przedstawione w przykładzie poniżej.

```
root@KaliLinux:~# ./ping_sweep.sh
Usage - ./ping_sweep.sh [/24 network address]
Example - ./ping_sweep.sh 172.16.36.0
Example will perform an ICMP ping sweep of the 172.16.36.0/24 network
root@KaliLinux:~# ./ping_sweep.sh 172.16.36.0
172.16.36.2
172.16.36.1
172.16.36.232
172.16.36.249
```

Jeżeli skrypt zostanie uruchomiony bez żadnych argumentów wywołania, na ekranie wyświetlony zostanie wspomniany wcześniej opis sposobu użycia. Jeżeli argumentem wywołania będzie adres podsieci, skrypt rozpocznie działanie i na ekranie wyświetlona zostanie lista aktywnych hostów działających w podanej podsieci. Jak już pokazywaliśmy w poprzednich recepturach, wyniki działania skryptu możesz w prosty sposób przekierować do pliku tekstowego na dysku. Aby to zrobić, powinieneś na końcu wiersza wywołania skryptu dodać znak większości (>) i nazwę pliku, w którym mają zostać zapisane dane.

```
root@KaliLinux:~# ./ping_sweep.sh 172.16.36.0 > output.txt
root@KaliLinux:~# ls output.txt
output.txt
root@KaliLinux:~# cat output.txt
172.16.36.2
172.16.36.1
172.16.36.232
172.16.36.249
```

Po zakończeniu działania skryptu i zapisaniu wyników działania w pliku możesz użyć polecenia ls do sprawdzenia, czy plik wynikowy został utworzony, a następnie za pomocą polecenia cat wyświetlić zawartość tego pliku.

## Jak to działa?

W branży IT polecenie `ping` jest powszechnie znanym narzędziem pozwalającym na wykrywanie aktywnych hostów w sieci. Polecenie `ping` zostało jednak napisane z myślą o testowaniu pojedynczych hostów i nie ma wbudowanych mechanizmów pozwalających na samodzielne skanowanie całych podsieci. Skrypt powłoki *bash*, opisany w tej recepturze, automatycznie wywołuje w pętli polecenie `ping` dla każdego z adresów podsieci `/24` CIDR, podanej jako argument wywołania. Dzięki zastosowaniu skryptu powłoki *bash* możesz uniknąć mozolnego, ręcznego testowania każdego z adresów IP z podanego zakresu i szybko wykonać takie zadanie.

# Skanowanie sieci na warstwie 3. przy użyciu programu Scapy

Scapy to narzędzie, które pozwala użytkownikowi na przygotowywanie i wstrzykiwanie do sieci odpowiednio spreparowanych pakietów. Przykładowo możesz wykorzystać ten program do tworzenia żądań protokołu ICMP, wstrzykiwania ich do sieci i analizowania nadchodzących odpowiedzi. W tej recepturze pokażemy, w jaki sposób możesz używać programu Scapy do wykrywania zdalnych hostów poprzez skanowanie sieci na warstwie 3.

## Przygotuj się

Zastosowanie programu Scapy do wykrywania hostów na warstwie 3. nie wymaga żadnego środowiska testowego, ponieważ bardzo wiele systemów działających w sieci Internet domyślnie odpowiada na żądania *ICMP Echo*. Z drugiej jednak strony, jeżeli nie znasz dokładnie przepisów prawa i regulacji, którym podlegasz w tym zakresie, rekomendowanym rozwiązaniem będzie zdecydowanie przeprowadzanie skanowania wyłącznie we własnym, dedykowanym środowisku testowym. Aby można było za pomocą programu Scapy przeprowadzić skanowanie na warstwie 3., w testowanej sieci musi działać przynajmniej jeden system, który będzie odpowiadał na żądania *ICMP Echo*. W przedstawionym przykładzie wykorzystujemy kombinację systemów Linux i Windows. Więcej szczegółowych informacji na temat instalacji i konfiguracji tych systemów w naszym środowisku testowym znajdziesz w rozdziale 1., w recepturach „Instalacja systemu Metasploitable2" oraz „Instalacja systemu Windows". Oprócz tego w tej recepturze będziemy używać edytorów tekstu, takich jak VIM czy Nano, do napisania w języku Python skryptów skanujących i zapisania ich w systemie plików. Więcej szczegółowych informacji na temat pisania skryptów znajdziesz w rozdziale 1., w recepturze „Praca z edytorami tekstu VIM i Nano".

## Jak to zrobić?

Aby przy użyciu programu Scapy wysyłać żądania *ICMP Echo* do zdalnych hostów, musimy rozpocząć pracę od zbudowania pakietów na poszczególnych warstwach. Dobrą zasadą przy budowaniu pakietów jest przechodzenie kolejno przez poszczególne warstwy modelu OSI. Aby wygenerować żądanie *ICMP Echo*, musimy połączyć warstwę protokołu IP z żądaniem ICMP. Rozpocznij od wykonania polecenia `scapy`, które uruchomi interaktywną konsolę programu Scapy, a następnie przypisz obiekt `IP` do zmiennej.

```
root@KaliLinux:~# scapy
Welcome to Scapy (2.2.0)
>>> ip = IP()
>>> ip.display()
###[ IP ]###
   version= 4
   ihl= None
   tos= 0x0
   len= None
   id= 1
   flags=
   frag= 0
   ttl= 64
   proto= ip
   chksum= None
   src= 127.0.0.1
   dst= 127.0.0.1
   \options\
```

W przedstawionym przykładzie funkcja `display` została użyta do wyświetlenia domyślnej konfiguracji atrybutów obiektu po przypisaniu go do zmiennej `ip`. Domyślnie obiekt `IP` jest skonfigurowany do wysyłania i odbierania pakietów z adresu pętli zwrotnej `127.0.0.1`. Aby w programie Scapy zmienić wartość dowolnie wybranego atrybutu obiektu, powinieneś wpisać nazwę atrybutu w formacie [*nazwa_obiektu*].[*nazwa_atrybutu*], a następnie wykonać przypisanie, wpisując znak równości i żądaną wartość atrybutu. W naszym przypadku chcemy zmienić docelowy adres IP na adres systemu, do którego chcemy wysłać żądanie ICMP, tak jak to zostało przedstawione na listingu poniżej:

```
>>> ip.dst = "172.16.36.135"
>>> ip.display()
###[ IP ]###
   version= 4
   ihl= None
   tos= 0x0
   len= None
   id= 1
   flags=
   frag= 0
   ttl= 64
```

```
    proto= ip
    chksum= None
    src= 172.16.36.180
    dst= 172.16.36.135
    \options\
```

Po przypisaniu nowej wartości do atrybutu reprezentującego docelowy adres IP możesz zweryfikować zmiany, wywołując ponownie funkcję `display()`. Zwróć uwagę, że jeżeli docelowy adres IP zmieni się na inną wartość niż domyślna, jako źródłowy adres IP zostanie automatycznie wpisany adres IP domyślnego, lokalnego interfejsu sieciowego. Skoro atrybuty obiektu `IP` zostały już pomyślnie zmodyfikowane, możemy rozpocząć tworzenie kolejnej warstwy naszego pakietu, czyli warstwy ICMP, którą przypiszemy do osobnej zmiennej, tak jak to zostało zaprezentowane poniżej.

```
>>> ping = ICMP()
>>> ping.display()
###[ ICMP ]###
   type= echo-request
   code= 0
   chksum= None
   id= 0x0
   seq= 0x0
```

W przedstawionym przykładzie obiekt `ICMP` został przypisany do zmiennej `ping`, a następnie funkcja `display()` została użyta do wyświetlenia domyślnej konfiguracji atrybutów ICMP. Aby wykonać żądanie *ICMP Echo*, wystarczą nam domyślne wartości atrybutów. Skoro mamy już gotowe obie warstwy, możemy je złożyć w jeden pakiet i przygotować do wysłania. W programie Scapy poszczególne warstwy można składać, oddzielając ich nazwy od siebie znakami prawego ukośnika, tak jak to zostało przedstawione na listingu poniżej.

```
>>> ping_request = (ip/ping)
>>> ping_request.display()
###[ IP ]###
   version= 4
   ihl= None
   tos= 0x0
   len= None
   id= 1
   flags=
   frag= 0
   ttl= 64
   proto= icmp
   chksum= None
   src= 172.16.36.180
   dst= 172.16.36.135
   \options\
###[ ICMP ]###
      type= echo-request
      code= 0
```

```
      chksum= None
      id= 0x0
      seq= 0x0
```

Po przypisaniu złożonych warstw do nowej zmiennej za pomocą funkcji `display()` możemy wyświetlić cały pakiet. Opisany powyżej proces składania warstw pakietu jest często nazywany kapsułkowaniem datagramów (ang. *datagram encapsulation*). Teraz, skoro poszczególne warstwy zostały złożone w jeden pakiet, nasze żądanie jest gotowe do wysłania w sieć, co w programie Scapy możemy zrobić za pomocą funkcji `sr1()`.

```
>>> ping_reply = sr1(ping_request)
..Begin emission:
.........Finished to send 1 packets.
...*
Received 15 packets, got 1 answers, remaining 0 packets
>>> ping_reply.display()
###[ IP ]###
   version= 4L
   ihl= 5L
   tos= 0x0
   len= 28
   id= 62577
   flags=
   frag= 0L
   ttl= 64
   proto= icmp
   chksum= 0xe513
   src= 172.16.36.135
   dst= 172.16.36.180
   \options\
###[ ICMP ]###
      type= echo-reply
      code= 0
      chksum= 0xffff
      id= 0x0
      seq= 0x0
###[ Padding ]###
         load= '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
         x00\x00\x00\x00'
```

W przedstawionym przykładzie funkcja `sr1()` zostaje przypisana do zmiennej `ping_reply`, co powoduje wywołanie funkcji i przekazanie do zmiennej wyników jej działania. Po otrzymaniu odpowiedzi zawartość zmiennej `ping_reply` jest wyświetlana za pomocą funkcji `display()`. Zauważ, że pakiet zawierający odpowiedź został nadesłany z hosta, do którego przesłaliśmy żądanie, a docelowym adresem IP dla odpowiedzi jest adres naszego systemu Kali Linux. Co więcej, warto również zauważyć, że nadesłana odpowiedź to pakiet typu *ICMP Reply*. Bazując na podanym przykładzie, możemy dojść do wniosku, że taki proces wysyłania żądań i otrzymywania odpowiedzi ICMP jest całkiem funkcjonalny, ale kiedy spróbujesz zastosować go dla nieaktywnego adresu IP, szybko zauważysz, gdzie leży problem.

```
>>> ip.dst = "172.16.36.136"
>>> ping_request = (ip/ping)
>>> ping_reply = sr1(ping_request)
.Begin emission:
.....................................................................
.....................................................................
.......... Finished to send 1 packets ..............................
.....................................................................
*** {Pozostałe wiersze zostały pominięte} ***
```

Większość wyników działania zaprezentowanych w przykładzie została celowo pominięta, ale w rzeczywistości kolejne wiersze będą wyświetlane dopóty, dopóki nie przerwiesz całego procesu, naciskając kombinację klawiszy *Ctrl+C*. Jeżeli podczas wywołania funkcji `sr1()` nie zdefiniujesz maksymalnego czasu, po którym jej działanie zostanie zakończone (ang. *timeout*), funkcja będzie nasłuchiwać aż do momentu, gdy nadejdzie odpowiedź na przesłane żądanie. Inaczej mówiąc, jeżeli host o podanym adresie IP nie istnieje bądź jest nieaktywny, działanie funkcji nie zostanie zakończone. Planując wykorzystanie tej funkcji w skryptach, powinieneś zawsze pamiętać o ustawieniu odpowiedniej wartości atrybutu `timeout`, tak jak to zostało przedstawione poniżej.

```
>>> ping_reply = sr1(ping_request, timeout=1)
.Begin emission:
.....................................................................
.....................................................................
Finished to send 1 packets.
.................................
Received 3982 packets, got 0 answers, remaining 1 packets
```

Gdy podasz w wierszu polecenia argument `timeout`, czyli drugi argument wywołania funkcji `sr1()`, proces zakończy działanie, jeżeli po upływie określonego czasu odpowiedź na przesłane żądanie nie nadejdzie. W przykładzie przedstawionym poniżej funkcja `sr1()` została użyta do wysłania żądania ICMP do nieistniejącego hosta i kiedy odpowiedź nie nadchodzi, kończy działanie po upływie jednej sekundy. W przykładach omawianych do tej pory przypisywaliśmy funkcje do zmiennych, tworząc obiekty, którymi można było później manipulować. Warto jednak zauważyć, że takie funkcje mogą być wywoływane bezpośrednio, bez wcześniejszego przypisywania do zmiennych, co zostało pokazane w przykładzie poniżej.

```
>>> answer = sr1(IP(dst="172.16.36.135")/ICMP(),timeout=1)
.Begin emission:
...*Finished to send 1 packets.

Received 5 packets, got 1 answers, remaining 0 packets
>>> response.display()
###[ IP ]###
   version= 4L
   ihl= 5L
   tos= 0x0
   len= 28
```

```
   id= 62578
   flags=
   frag= 0L
   ttl= 64
   proto= icmp
   chksum= 0xe512
   src= 172.16.36.135
   dst= 172.16.36.180
   \options\
###[ ICMP ]###
      type= echo-reply
      code= 0
      chksum= 0xffff
      id= 0x0
      seq= 0x0
###[ Padding ]###
         load=
'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

W przedstawionym przykładzie wszystko to, co do tej pory robiliśmy w czterech wierszach, zostało skompresowane do jednego polecenia, w którym funkcje są wywoływane bezpośrednio. Zauważ, że jeżeli żądanie ICMP zostanie wysłane do adresu IP, który nie prześle odpowiedzi w czasie określonym przez atrybut `timeout`, to późniejsze wywołanie takiego obiektu spowoduje wygenerowanie wyjątku. W przykładzie poniżej widzimy, że ponieważ nie otrzymaliśmy żadnej odpowiedzi na przesłane żądanie, zmienna `answer` nie została zainicjowana i próba wyświetlenia jej zawartości kończy się niepowodzeniem.

```
>>> answer = sr1(IP(dst="83.166.169.231")/ICMP(),timeout=1)
Begin emission:
.....................................Finished to send 1 packets.
.....................................................................
.....................
Received 1180 packets, got 0 answers, remaining 1 packets
>>> answer.display()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'display'
```

Jeżeli wiemy, jak wyglądają odpowiedzi dla istniejących i nieistniejących hostów, możemy w języku Python napisać skrypt, który będzie wysyłał żądania ICMP do wielu adresów IP. Pętla w skrypcie przechodzi przez wszystkie wartości ostatniego oktetu adresu IP, wysyła żądanie ICMP dla kolejnych wartości, analizuje odpowiedzi dla poszczególnych wywołań funkcji `sr1()` i na ich podstawie wyświetla wyniki działania.

```
#!/usr/bin/python

import logging
logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
from scapy.all import *
```

```
if len(sys.argv) != 2:
   print "Usage - ./pinger.py [/24 network address]"
   print "Example - ./pinger.py 172.16.36.0"
   print "Example will perform an ICMP scan of the 172.16.36.0/24 range"
   sys.exit()

address = str(sys.argv[1])
prefix = address.split('.')[0] + '.' + address.split('.')[1] + '.' +
address.split('.')[2] + '.'

for addr in range(1,254):
   answer=sr1(ARP(pdst=prefix+str(addr)),timeout=1,verbose=0)
   if answer == None:
      pass
   else:
      print prefix+str(addr)
```

Pierwszy wiersz skryptu wskazuje lokalizację interpretera języka Python, dzięki czemu skrypt może zostać wykonany bez konieczności podawania tej informacji w wierszu wywołania. Następnie skrypt importuje wszystkie funkcje Scapy oraz definiuje poziomy logowania, co pozwala na wyeliminowanie niepotrzebnych elementów w wynikach działania programu. Drugi blok kodu zawiera instrukcję warunkową, sprawdzającą, czy w wierszu poleceń została podana odpowiednia liczba argumentów wywołania skryptu. Jeżeli nie, na ekranie wyświetlana jest krótka informacja, składająca się z opisu składni, przykładu wywołania oraz określenia przeznaczenia skryptu. Dalej znajduje się pojedynczy wiersz kodu, w którym argument wywołania skryptu zostaje przypisany do zmiennej `address`. Jej wartość jest następnie wykorzystywana do wyodrębnienia podciągu znaków reprezentującego prefiks sieci. Na przykład jeżeli w zmiennej `address` przechowywany jest adres `192.168.11.0`, to do zmiennej `prefix` zostanie przypisana wartość `192.168.11.`. W ostatnim bloku kodu umieszczona została pętla `for`, która realizuje właściwe skanowanie. Pętla przechodzi kolejno przez wartości od `0` do `254` i w każdej iteracji wartość licznika pętli jest dołączana do prefiksu sieci. W naszym przykładzie przedstawionym wcześniej żądanie ICMP zostanie rozesłane do wszystkich hostów o adresach IP od `192.168.11.0` do `192.168.11.254`. Jeżeli dany host odeśle odpowiedź, na ekranie wyświetlony zostanie jego adres IP, wskazujący, że host o takim adresie jest aktywny. Po zapisaniu skryptu w lokalnym katalogu na dysku możesz spróbować uruchomić go z poziomu okna terminala, wpisując w wierszu wywołania kropkę, prawy ukośnik i nazwę skryptu, tak jak to zostało przedstawione w przykładzie poniżej.

```
root@KaliLinux:~# ./pinger.py
Usage - ./pinger.py [/24 network address]
Example - ./pinger.py 172.16.36.0
Example will perform an ICMP scan of the 172.16.36.0/24 range
root@KaliLinux:~# ./pinger.py 172.16.36.0
172.16.36.2
172.16.36.1
172.16.36.132
172.16.36.135
```

Jeżeli skrypt zostanie uruchomiony bez żadnych argumentów wywołania, na ekranie wyświetlony zostanie wspomniany wcześniej opis sposobu użycia, z którego wynika, że poprawne uruchomienie skryptu wymaga podania jednego argumentu, reprezentującego adres /24 sieci przeznaczonej do skanowania. W naszym przykładzie skrypt został użyty do skanowania sieci 172.16.36.0. Podobnie jak pokazywaliśmy w poprzednich przykładach, możesz bez trudu przekierować wyniki działania skryptu do pliku na dysku. Aby to zrobić, powinieneś na końcu wiersza wywołania skryptu dodać znak większości (>) i nazwę pliku, w którym mają zostać zapisane dane, tak jak to zostało przedstawione poniżej.

```
root@KaliLinux:~# ./pinger.py 172.16.36.0 > output.txt
root@KaliLinux:~# ls output.txt
output.txt
root@KaliLinux:~# cat output.txt
172.16.36.1
172.16.36.2
172.16.36.132
172.16.36.135
```

Po zakończeniu działania skryptu i zapisaniu wyników działania w pliku możesz użyć polecenia ls do sprawdzenia, czy plik wynikowy został utworzony, a następnie za pomocą polecenia cat wyświetlić zawartość tego pliku. W razie potrzeby możesz łatwo zmodyfikować skrypt tak, aby wysyłał żądania tylko do hostów, których adresy IP znajdują się w pliku tekstowym podanym jako argument wywołania skryptu. By to zrobić, pętla for musi odczytywać kolejne adresy IP z pliku. Przykład takiego skryptu został zaprezentowany poniżej.

```
#!/usr/bin/python

import logging
logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
from scapy.all import *

if len(sys.argv) != 2:
   print "Usage - ./pinger.py [filename]"
   print "Example - ./pinger.py iplist.txt"
   print "Example will perform an ICMP ping scan of the IP addresses listed in
   iplist.txt"
   sys.exit()

filename = str(sys.argv[1])
file = open(filename,'r')

for addr in file:
   ans=sr1(IP(dst=addr.strip())/ICMP(),timeout=1,verbose=0)
   if ans == None:
      pass
   else:
      print addr.strip()
```

Jedyną większą różnicą pomiędzy tym skryptem a jego poprzednią wersją jest to, że zamiast adresu sieci podczas wywołania skryptu musisz podać nazwę pliku tekstowego, zawierającego listę adresów IP do sprawdzenia. Podobnie jak w poprzednich przypadkach, wynikiem działania skryptu jest lista adresów IP hostów, które na przesłane żądania *ICMP Echo* odpowiedziały pakietami *ICMP Reply*.

```
root@KaliLinux:~# ./pinger.py
Usage - ./pinger.py [filename]
Example - ./pinger.py iplist.txt
Example will perform an ICMP ping scan of the IP addresses listed in iplist.txt
root@KaliLinux:~# ./pinger.py iplist.txt
172.16.36.1
172.16.36.2
172.16.36.132
172.16.36.135
```

Wyniki działania skryptu możesz przekierować do pliku tekstowego w dokładnie taki sam sposób jak poprzednio. Aby to zrobić, powinieneś na końcu wiersza wywołania skryptu dodać znak większości (>) i nazwę pliku, w którym mają zostać zapisane dane. Przykład takiego rozwiązania został przedstawiony poniżej.

```
root@KaliLinux:~# ./pinger.py iplist.txt > output.txt
root@KaliLinux:~# ls output.txt
output.txt
root@KaliLinux:~# cat output.txt
172.16.36.1
172.16.36.2
172.16.36.132
172.16.36.135
```

## Jak to działa?

W tej recepturze przeprowadzaliśmy skanowanie sieci na warstwie 3. za pomocą programu Scapy, w którym budowaliśmy pakiety składające się z warstwy protokołu IP oraz dołączonego do niej żądania ICMP. Warstwa protokołu IP powoduje, że taki pakiet może być routowany poza sieć lokalną, a żądanie ICMP powoduje wymuszenie nadesłania odpowiedzi przez zdalny system. Wykorzystując taką technikę w skryptach języka Python, możemy przeprowadzać szybkie skanowanie wielu systemów czy nawet całych podsieci.

# Skanowanie sieci na warstwie 3. przy użyciu programu Nmap

Nmap jest jednym z najbardziej rozbudowanych i uniwersalnych narzędzi skanujących dostępnych w systemie Kali Linux. Z tego powodu nie powinno chyba nikogo dziwić, że Nmap ma również wbudowane mechanizmy pozwalające na skanowanie sieci z wykorzystaniem protokołu ICMP. W tej recepturze pokażemy, w jaki sposób możesz używać tego programu do wykrywania zdalnych hostów poprzez skanowanie sieci na warstwie 3.

## Przygotuj się

Zastosowanie skanera Nmap do wykrywania hostów na warstwie 3. nie wymaga żadnego środowiska testowego, ponieważ bardzo wiele systemów działających w sieci Internet domyślnie odpowiada na żądania *ICMP Echo*. Z drugiej jednak strony, jeżeli nie znasz dokładnie przepisów prawa i regulacji, którym podlegasz w tym zakresie, rekomendowanym rozwiązaniem będzie zdecydowanie przeprowadzanie skanowania wyłącznie we własnym, dedykowanym środowisku testowym. Aby można było za pomocą programu Nmap przeprowadzić skanowanie na warstwie 3., w testowanej sieci musi działać przynajmniej jeden system, który będzie odpowiadał na żądania *ICMP Echo*. W przedstawionym przykładzie wykorzystujemy kombinację systemów Linux i Windows. Więcej szczegółowych informacji na temat instalacji i konfiguracji tych systemów w naszym środowisku testowym znajdziesz w rozdziale 1., w recepturach „Instalacja systemu Metasploitable2" oraz „Instalacja systemu Windows".

## Jak to zrobić?

Nmap jest inteligentnym narzędziem, które samo adaptuje się do rodzaju skanowania i w zależności od potrzeb automatycznie wybiera skanowanie na warstwie 2., 3. czy 4. Przykładowo jeżeli w wierszu wywołania programu użyjemy opcji `-sn` do skanowania adresów IP znajdujących się poza siecią lokalną, Nmap automatycznie zacznie wysyłać żądania ICMP i oczekiwać na odpowiedzi ze zdalnych hostów. Aby przeprowadzić skanowanie ICMP pojedynczego hosta, użyj w wierszu poleceń opcji `-sn` i jako argument wywołania podaj adres IP badanego hosta, tak jak to zostało przedstawione poniżej.

```
root@KaliLinux:~# nmap -sn 74.125.228.1

Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-16 23:05 EST
Nmap scan report for iad23s05-in-f1.1e100.net (74.125.228.1)
Host is up (0.00013s latency).
Nmap done: 1 IP address (1 host up) scanned in 0.02 seconds
```

Wyniki działania tego polecenia pokazują, czy dany host jest aktywny, oraz dostarczają paru dodatkowych informacji o przeprowadzonym skanie. Warto również zauważyć, że w wynikach skanowania oprócz adresu IP pojawia się również nazwa badanego systemu (aby ją uzyskać,

Nmap „odpytuje" odpowiednie serwery DNS). Jak już wspominaliśmy w jednej z poprzednich receptur, programu Nmap możesz również używać do skanowania całych zakresów adresów IP, podając je w wierszu wywołania w notacji z myślnikiem. Nmap jest programem pracującym domyślnie na wielu wątkach i wykorzystującym wiele procesów działających równolegle, dzięki czemu charakteryzuje się dużą szybkością skanowania. Przykład wywołania programu Nmap został przedstawiony poniżej.

```
root@KaliLinux:~# nmap -sn 74.125.228.1-255

Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-16 23:14 EST
Nmap scan report for iad23s05-in-f1.1e100.net (74.125.228.1)
Host is up (0.00012s latency).
Nmap scan report for iad23s05-in-f2.1e100.net (74.125.228.2)
Host is up (0.0064s latency).
Nmap scan report for iad23s05-in-f3.1e100.net (74.125.228.3)
Host is up (0.0070s latency).
Nmap scan report for iad23s05-in-f4.1e100.net (74.125.228.4)
Host is up (0.00015s latency).
Nmap scan report for iad23s05-in-f5.1e100.net (74.125.228.5)
Host is up (0.00013s latency).
Nmap scan report for iad23s05-in-f6.1e100.net (74.125.228.6)
Host is up (0.00012s latency).
Nmap scan report for iad23s05-in-f7.1e100.net (74.125.228.7)
Host is up (0.00012s latency).
Nmap scan report for iad23s05-in-f8.1e100.net (74.125.228.8)
Host is up (0.00012s latency).
*** { Pozostałe wiersze zostały pominięte} ***
```

W powyższym przykładzie Nmap został użyty do przeskanowania całej podsieci /24. Dla uproszczenia analizy większość wyników działania została celowo pominięta. Analizując ruch sieciowy za pomocą programu Wireshark, możesz zauważyć, że skanowanie kolejnych adresów IP nie przebiega sekwencyjnie (co widać na rysunku załączonym poniżej). Jest to kolejny dowód na wielowątkową naturę programu Nmap i jego możliwości szybkiego skanowania dużych podsieci.

| No. | Destination | Protocol | Info |
|---|---|---|---|
| 85 | 74.125.228.2 | ICMP | Echo (ping) request id=0x0620, seq=0/0, ttl=52 |
| 86 | 74.125.228.3 | ICMP | Echo (ping) request id=0x3507, seq=0/0, ttl=50 |
| 87 | 74.125.228.4 | ICMP | Echo (ping) request id=0xa375, seq=0/0, ttl=44 |
| 88 | 74.125.228.5 | ICMP | Echo (ping) request id=0xc693, seq=0/0, ttl=45 |
| 89 | 74.125.228.6 | ICMP | Echo (ping) request id=0x2f9b, seq=0/0, ttl=56 |
| 90 | 74.125.228.7 | ICMP | Echo (ping) request id=0xfa75, seq=0/0, ttl=43 |

Programu Nmap możesz również używać do skanowania hostów, których adresy IP znajdują się na liście w pliku tekstowym. Aby to zrobić, powinieneś w wierszu wywołania polecenia dodać opcję -iL, po której następuje nazwa pliku zawierającego listę adresów IP do skanowania.

```
root@KaliLinux:~# cat iplist.txt
74.125.228.13
74.125.228.28
74.125.228.47
74.125.228.144
74.125.228.162
74.125.228.211
root@KaliLinux:~# nmap -iL iplist.txt -sn

Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-16 23:14 EST
Nmap scan report for iad23s05-in-f13.1e100.net (74.125.228.13)
Host is up (0.00010s latency).
Nmap scan report for iad23s05-in-f28.1e100.net (74.125.228.28)
Host is up (0.0069s latency).
Nmap scan report for iad23s06-in-f15.1e100.net (74.125.228.47)
Host is up (0.0068s latency).
Nmap scan report for iad23s17-in-f16.1e100.net (74.125.228.144)
Host is up (0.00010s latency).
Nmap scan report for iad23s18-in-f2.1e100.net (74.125.228.162)
Host is up (0.0077s latency).
Nmap scan report for 74.125.228.211
Host is up (0.00022s latency).
Nmap done: 6 IP addresses (6 hosts up) scanned in 0.04 seconds
```

W przykładzie przedstawionym powyżej w bieżącym katalogu roboczym znajdował się plik tekstowy zawierający listę sześciu adresów IP. Nazwa tego pliku została przekazana jako argument wywołania programu Nmap, który rozpoczął skanowanie znajdujących się na niej adresów IP.

## Jak to działa?

Nmap wykonuje skanowanie sieci na warstwie 3. poprzez rozsyłanie na adresy IP z podanego zakresu żądań *ICMP Echo*. Ponieważ Nmap jest narzędziem działającym wielowątkowo, żądania ICMP są rozsyłane na wiele adresów IP jednocześnie, dzięki czemu użytkownik szybko otrzymuje wyniki skanowania całego zakresu. Jak już wspominaliśmy, Nmap potrafi się automatycznie adaptować do rodzaju skanowania i zakresu adresów IP, dzięki czemu wykorzystuje skanowanie ICMP na warstwie 3. tylko wtedy, kiedy skanowanie ARP nie przyniesie oczekiwanych rezultatów. Podobnie jeżeli okaże się, że ani skanowanie ICMP, ani ARP nie pozwoli na wykrywanie aktywnych hostów, Nmap automatycznie rozpocznie skanowanie sieci na warstwie 4.

# Skanowanie sieci na warstwie 3. przy użyciu programu fping

`fping` to narzędzie, które jest bardzo podobne do popularnego i dobrze znanego polecenia `ping`, ale oferuje również kilka dodatkowych możliwości, których nie znajdziemy w poleceniu `ping`. To właśnie dzięki tym dodatkowym mechanizmom program `fping` może być bez żadnych modyfikacji używany jako w pełni funkcjonalny skaner. W tej recepturze pokażemy, w jaki sposób możesz wykorzystywać to narzędzie do wykrywania hostów poprzez skanowanie sieci na warstwie 3.

## Przygotuj się

Zastosowanie polecenia `fping` do wykrywania hostów na warstwie 3. nie wymaga żadnego środowiska testowego, ponieważ bardzo wiele systemów działających w sieci Internet domyślnie odpowiada na żądania *ICMP Echo*. Z drugiej jednak strony, jeżeli nie znasz dokładnie przepisów prawa i regulacji, którym podlegasz w tym zakresie, rekomendowanym rozwiązaniem będzie zdecydowanie przeprowadzanie skanowania wyłącznie we własnym, dedykowanym środowisku testowym. Aby można było za pomocą polecenia `fping` przeprowadzić skanowanie na warstwie 3., w testowanej sieci musi działać przynajmniej jeden system, który będzie odpowiadał na żądania *ICMP Echo*. W przedstawionym przykładzie wykorzystujemy kombinację systemów Linux i Windows. Więcej szczegółowych informacji na temat instalacji i konfiguracji tych systemów w naszym środowisku testowym znajdziesz w rozdziale 1., w recepturach „Instalacja systemu Metasploitable2" oraz „Instalacja systemu Windows".

## Jak to zrobić?

Jak już wspominaliśmy, polecenie `fping` jest bardzo podobne do polecenia `ping`, z tym że ma kilka dodatkowych funkcji. Możesz go używać w taki sam sposób do wysyłania żądań ICMP do pojedynczych adresów IP i sprawdzania, czy host o danym adresie jest aktywny. Aby to zrobić, powinieneś podać adres IP sprawdzanego hosta jako argument wywołania polecenia `fping`.

```
root@KaliLinux:~# fping 172.16.36.135
172.16.36.135 is alive
```

W przeciwieństwie do standardowego polecenia `ping`, polecenie `fping` po otrzymaniu odpowiedzi z badanego hosta przestaje wysyłać kolejne żądania *ICMP Echo* i wyświetla na ekranie informację, że taki host jest aktywny. Domyślnie jeżeli po wysłaniu czterech kolejnych żądań *ICMP Echo* host o podanym adresie IP nadal nie odpowiada, polecenie `fping` zakończy działanie i wyświetli na ekranie informację, że host jest nieosiągalny (ang. *Host Unreachable*).

```
root@KaliLinux:~# fping 172.16.36.136
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.136
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
```

```
172.16.36.136
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.136
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.136
172.16.36.136 is unreachable
```

Jeżeli chcesz zmienić domyślną liczbę wysyłanych żądań ICMP, powinieneś dodać w wierszu wywołania polecenia opcję `-c` i umieścić po niej liczbę całkowitą, reprezentującą nową liczbę żądań do wysłania.

```
root@KaliLinux:~# fping 172.16.36.135 -c 1
172.16.36.135 : [0], 84 bytes, 0.67 ms (0.67 avg, 0% loss)

172.16.36.135 : xmt/rcv/%loss = 1/1/0%, min/avg/max = 0.67/0.67/0.67
root@KaliLinux:~# fping 172.16.36.136 -c 1

172.16.36.136 : xmt/rcv/%loss = 1/0/100%
```

Kiedy używasz polecenia `fping` w taki sposób, wyniki jego działania są nieco bardziej złożone, a ich zrozumienie wymaga chwili zastanowienia. W wynikach działania polecenia znajdziesz między innymi adres IP badanego hosta, liczbę wysłanych żądań (parametr `xmt`), liczbę otrzymanych odpowiedzi (parametr `rcv`) oraz wyrażoną w procentach wartość reprezentującą liczbę utraconych pakietów. W przedstawionym przykładzie pierwszy host jest aktywny i odpowiada na żądania, o czym świadczy informacja o rozmiarze odebranej odpowiedzi (84 bajty) i czasie, po jakim odpowiedź została odebrana (0,67 ms). O tym, czy dany host jest online, możesz się również przekonać, sprawdzając procent utraconych pakietów. Wartość 100% oznacza, że z badanego hosta nie otrzymano żadnej odpowiedzi.

Choć polecenie `ping` jest najczęściej używanym narzędziem do badania hostów w sieci, to jednak nie ma takich możliwości jak `fping`, które ma wbudowane mechanizmy pozwalające na skanowanie wielu hostów jednocześnie. Aby za pomocą tego polecenia przeskanować wybrany zakres adresów IP, powinieneś w wierszu wywołania polecenia umieścić opcję `-g`, a po niej wpisać pierwszy i ostatni adres IP żądanego zakresu:

```
root@KaliLinux:~# fping -g 172.16.36.1 172.16.36.4
172.16.36.1 is alive
172.16.36.2 is alive
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.3
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.3
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.3
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.3
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.4
```

```
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.4
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.4
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.4
172.16.36.3 is unreachable
172.16.36.4 is unreachable
```

Używając opcji `-g`, możesz również podawać zakresy adresów IP w notacji CIDR, a polecenie `fping` automatycznie przeliczy zakresy i wygeneruje listę adresów IP do sprawdzenia:

```
root@KaliLinux:~# fping -g 172.16.36.0/24
172.16.36.1 is alive
172.16.36.2 is alive
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.3
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.4
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.5
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.6
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.7
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.8
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.9
*** {Pozostałe wiersze zostały pominięte} ***
```

Nie sposób również nie wspomnieć o tym, że polecenie `fping` może być wykorzystywane do sprawdzania hostów, których adresy IP zostały zapisane w pliku tekstowym. Aby skorzystać z takiej możliwości, powinieneś w wierszu wywołania polecenia umieścić opcję `-f` i po niej podać nazwę pliku, w którym znajduje się lista adresów IP do przeskanowania:

```
root@KaliLinux:~# fping -f iplist.txt
172.16.36.2 is alive
172.16.36.1 is alive
172.16.36.132 is alive
172.16.36.135 is alive
172.16.36.180 is alive
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.203
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.203
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.203
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.203
```

```
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.205
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.205
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.205
ICMP Host Unreachable from 172.16.36.180 for ICMP Echo sent to
172.16.36.205
172.16.36.203 is unreachable
172.16.36.205 is unreachable
172.16.36.254 is unreachable
```

## Jak to działa?

Polecenie `fping` przeprowadza skanowanie sieci z wykorzystaniem protokołu ICMP dokładnie w taki sam sposób, w jaki robią to inne narzędzia, o których mówiliśmy już wcześniej w tym rozdziale. Dla każdego wejściowego adresu IP polecenie `fping` wysyła jedno lub więcej żądań *ICMP Echo* i na podstawie otrzymanych odpowiedzi dokonuje identyfikacji aktywnych hostów. Dodając w wierszu wywołania odpowiednie opcje, możesz użyć polecenia `fping` do skanowania całych zakresów adresów IP bądź wielu adresów IP, które zostały zapisane w pliku tekstowym. Dzięki takiemu mechanizmowi polecenie `fping` jest w pełni funkcjonalnym skanerem i nie musisz już dodatkowo tworzyć skryptów powłoki, tak jak to robiliśmy w przypadku polecenia `ping`.

# Skanowanie sieci na warstwie 3. przy użyciu programu hping3

Jeszcze bardziej uniwersalnym narzędziem, którego można używać do wykrywania hostów w sieci, jest polecenie `hping3`. Jego przewaga nad poleceniem `fping` polega na tym, że wykorzystuje ono wiele różnych technik wykrywania hostów, ale jest zdecydowanie mniej użyteczne jako skaner, ponieważ umożliwia skanowanie tylko jednego hosta naraz. Na szczęście taką wadę możemy łatwo zniwelować, pisząc odpowiedni skrypt powłoki. W tej recepturze pokażemy, w jaki sposób możesz używać polecenia `hping3` do wykrywania hostów poprzez skanowanie sieci na warstwie 3.

## Przygotuj się

Zastosowanie polecenia `hping3` do wykrywania hostów na warstwie 3. nie wymaga żadnego środowiska testowego, ponieważ bardzo wiele systemów działających w sieci Internet domyślnie odpowiada na żądania *ICMP Echo*. Z drugiej jednak strony, jeżeli nie znasz dokładnie przepisów prawa i regulacji, którym podlegasz w tym zakresie, rekomendowanym rozwiązaniem

będzie zdecydowanie przeprowadzanie skanowania wyłącznie we własnym, dedykowanym środowisku testowym. Aby można było za pomocą polecenia `hping3` przeprowadzić skanowanie na warstwie 3., w testowanej sieci musi działać przynajmniej jeden system, który będzie odpowiadał na żądania *ICMP Echo*. W przedstawionym przykładzie wykorzystujemy kombinację systemów Linux i Windows. Więcej szczegółowych informacji na temat instalacji i konfiguracji tych systemów w naszym środowisku testowym znajdziesz w rozdziale 1., w recepturach „Instalacja systemu Metasploitable2" oraz „Instalacja systemu Windows". Oprócz tego w tej recepturze będziemy używać edytorów tekstu, takich jak VIM czy Nano, do napisania skryptów skanujących i zapisania ich w systemie plików. Więcej szczegółowych informacji na temat pisania skryptów znajdziesz w rozdziale 1., w recepturze „Praca z edytorami tekstu VIM i Nano".

## Jak to zrobić?

Polecenie `hping3` to potężne narzędzie, oferujące ogromną liczbę opcji i wiele trybów pracy, które umożliwia wykrywanie hostów poprzez skanowanie sieci zarówno na warstwie 3., jak i 4. Aby za pomocą tego polecenia przeprowadzić proste sprawdzenie pojedynczego hosta, powinieneś w wierszu wywołania podać jego adres IP, a następnie ustawić tryb skanowania na ICMP, tak jak to zostało przedstawione poniżej.

```
root@KaliLinux:~# hping3 172.16.36.1 --icmp
HPING 172.16.36.1 (eth1 172.16.36.1): icmp mode set, 28 headers + 0 data bytes
len=46 ip=172.16.36.1 ttl=64 id=41835 icmp_seq=0 rtt=0.3 ms
len=46 ip=172.16.36.1 ttl=64 id=5039 icmp_seq=1 rtt=0.3 ms
len=46 ip=172.16.36.1 ttl=64 id=54056 icmp_seq=2 rtt=0.6 ms
len=46 ip=172.16.36.1 ttl=64 id=50519 icmp_seq=3 rtt=0.5 ms
len=46 ip=172.16.36.1 ttl=64 id=47642 icmp_seq=4 rtt=0.4 ms
^C
--- 172.16.36.1 hping statistic ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 0.3/0.4/0.6 ms
```

W przedstawionym przykładzie działanie polecenia zostało przerwane naciśnięciem przez użytkownika kombinacji klawiszy *Ctrl+C*. Podobnie jak to miało miejsce w przypadku standardowego polecenia `ping`, polecenie `hping3`, działające w trybie ICMP, wysyła żądania w niekończącej się pętli, o ile liczba wysyłanych pakietów nie zostanie ograniczona poprzez ustawienie odpowiedniej opcji. Jeżeli chcesz zmienić domyślną liczbę wysyłanych żądań ICMP, powinieneś dodać w wierszu wywołania polecenia opcję `-c` i umieścić po niej liczbę całkowitą, reprezentującą nową liczbę żądań do wysłania.

```
root@KaliLinux:~# hping3 172.16.36.1 --icmp -c 2
HPING 172.16.36.1 (eth1 172.16.36.1): icmp mode set, 28 headers + 0 data bytes
len=46 ip=172.16.36.1 ttl=64 id=40746 icmp_seq=0 rtt=0.3 ms
len=46 ip=172.16.36.1 ttl=64 id=12231 icmp_seq=1 rtt=0.5 ms

--- 172.16.36.1 hping statistic ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.3/0.4/0.5 ms
```

Polecenie hping3 nie potrafi skanować wielu hostów jednocześnie, jednak możemy w prosty sposób zniwelować tę niedogodność poprzez napisanie odpowiedniego skryptu powłoki. Aby to zrobić, musimy najpierw zidentyfikować różnice w wynikach działania polecenia dla aktywnego hosta oraz w sytuacji, kiedy sprawdzany host nie odpowiada na żądania. W przykładzie powyżej możemy zobaczyć, jak wyglądają wyniki działania programu dla aktywnego hosta. Teraz musimy uruchomić polecenie hping3 dla adresu IP, który nie jest przypisany do żadnego działającego hosta:

```
root@KaliLinux:~# hping3 172.16.36.4 --icmp -c 2
HPING 172.16.36.4 (eth1 172.16.36.4): icmp mode set, 28 headers + 0 data bytes

--- 172.16.36.4 hping statistic ---
2 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
```

Przeglądając i porównując oba wyniki, możemy znaleźć ciąg znaków, który pozwoli nam na odróżnienie działających hostów od tych, które nie są włączone. Łatwo zauważyć, że w wynikach działania polecenia hping3 rozmiar otrzymanej odpowiedzi (parametr len) jest wyświetlany tylko i wyłącznie dla aktywnych hostów. Znając tę właściwość, możemy za pomocą polecenia grep wyodrębnić z wyników działania polecenia hping3 wiersze zawierające ciąg znaków len. Aby sprawdzić nasze założenia, za pomocą potokowania połączymy teraz polecenie hping3 z poleceniem grep. Zakładając, że ciąg znaków len rzeczywiście pojawia się tylko w przypadku otrzymania odpowiedzi na żądanie ICMP, polecenie przedstawione poniżej powinno wyświetlić wyłącznie wiersze zawierające informacje o aktywnych hostach:

```
root@KaliLinux:~# hping3 172.16.36.1 --icmp -c 1; hping3 172.16.36.4 --icmp -c 1 |
grep "len"
HPING 172.16.36.1 (eth1 172.16.36.1): icmp mode set, 28 headers + 0 data bytes
len=46 ip=172.16.36.1 ttl=64 id=63974 icmp_seq=0 rtt=0.2 ms

--- 172.16.36.1 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.2/0.2/0.2 ms

--- 172.16.36.4 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
```

Pomimo że otrzymaliśmy mniej więcej taki wynik, o jaki nam chodziło, nietrudno zauważyć, że polecenie grep nie zostało w tym przypadku zastosowane zbyt efektywnie. Ponieważ polecenie hping3 wyświetla wyniki w dosyć specyficzny sposób, efektywne filtrowanie wierszy za pomocą polecenia grep jest mocno utrudnione. Spróbujemy zatem osiągnąć zamierzony cel w nieco inny sposób. Zamiast bezpośrednio filtrować dane z wyjścia polecenia hping3, przekierujemy je do pliku, a dopiero potem spróbujemy użyć polecenia grep. Aby to zrobić, zapiszemy wyniki działania obu używanych wcześniej poleceń ping3 w pliku o nazwie *handle.txt*:

```
root@KaliLinux:~# hping3 172.16.36.1 --icmp -c 1 >> handle.txt

--- 172.16.36.1 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms
root@KaliLinux:~# hping3 172.16.36.4 --icmp -c 1 >> handle.txt

--- 172.16.36.4 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
root@KaliLinux:~# cat handle.txt
HPING 172.16.36.1 (eth1 172.16.36.1): icmp mode set, 28 headers + 0 data bytes
len=46 ip=172.16.36.1 ttl=64 id=56022 icmp_seq=0 rtt=0.4 ms
HPING 172.16.36.4 (eth1 172.16.36.4): icmp mode set, 28 headers + 0 data bytes
```

Choć nasza próba zakończyła się tylko częściowym sukcesem, ponieważ nie wszystkie dane z wyjścia poleceń `hping3` zostały zapisane w pliku, to jednak możemy zauważyć, że to, co się „zapisało", w zupełności wystarczy nam do przygotowania w pełni funkcjonalnego skryptu powłoki — w pliku tekstowym zapisywane są tylko wiersze wyników powiązane z aktywnymi hostami, zawierające wiele informacji, łącznie z adresem IP. Aby przekonać się, czy takie podejście sprawdzi się w praktyce, spróbujemy teraz przygotować nieco bardziej złożone polecenie, które będzie przechodziło w pętli przez kolejne adresy IP zakresu `/24`, dla każdego z adresów wywoływało polecenie `hping3` i przekazywało wyniki jego działania do pliku *handle.txt*.

```
root@KaliLinux:~# for addr in $(seq 1 254); do hping3 172.16.36.$addr
--icmp -c 1 >> handle.txt & done

--- 172.16.36.2 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 6.6/6.6/6.6 ms

--- 172.16.36.1 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 55.2/55.2/55.2 ms

--- 172.16.36.8 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
*** {Pozostałe wiersze zostały pominięte} ***
```

Po wykonaniu takiego polecenia możemy się przekonać, że wyniki jego działania są bardzo obszerne (fragment zamieszczony w powyższym przykładzie został celowo przycięty) i zawierają wiele dodatkowych wierszy, które nie są zapisywane w pliku na dysku. Pamiętaj jednak, że powodzenie całego przedsięwzięcia nie zależy od tego, jak bardzo „gadatliwa" jest nasza pętla, ale od tego, czy będziemy w stanie z pliku wynikowego wyodrębnić interesujące nas dane.

```
root@KaliLinux:~# ls
Desktop handle.txt pinger.sh
root@KaliLinux:~# grep len handle.txt
len=46 ip=172.16.36.2 ttl=128 id=7537 icmp_seq=0 rtt=6.6 ms
```

```
len=46 ip=172.16.36.1 ttl=64 id=56312 icmp_seq=0 rtt=55.2 ms
len=46 ip=172.16.36.132 ttl=64 id=47801 icmp_seq=0 rtt=27.3 ms
len=46 ip=172.16.36.135 ttl=64 id=62601 icmp_seq=0 rtt=77.9 ms
```

Po zakończeniu działania pętli skanującej możesz za pomocą polecenia `ls` sprawdzić, czy plik tekstowy został utworzony zgodnie z oczekiwaniami, a następnie przy użyciu polecenia `grep` możesz spróbować wyodrębnić z niego wszystkie wiersze zawierające ciąg znaków `len`. Wyniki działania tego polecenia dadzą nam listę wierszy zawierających informacje o aktywnych hostach. W tym momencie pozostało nam już tylko wyodrębnić z poszczególnych wierszy same adresy IP, a następnie połączyć to wszystko w jeden funkcjonalny skrypt. Sposób „wycinania" adresów IP został przedstawiony poniżej.

```
root@KaliLinux:~# grep len handle.txt
len=46 ip=172.16.36.2 ttl=128 id=7537 icmp_seq=0 rtt=6.6 ms
len=46 ip=172.16.36.1 ttl=64 id=56312 icmp_seq=0 rtt=55.2 ms
len=46 ip=172.16.36.132 ttl=64 id=47801 icmp_seq=0 rtt=27.3 ms
len=46 ip=172.16.36.135 ttl=64 id=62601 icmp_seq=0 rtt=77.9 ms
root@KaliLinux:~# grep len handle.txt | cut -d " " -f 2
ip=172.16.36.2
ip=172.16.36.1
ip=172.16.36.132
ip=172.16.36.135
root@KaliLinux:~# grep len handle.txt | cut -d " " -f 2 | cut -d "=" -f 2
172.16.36.2
172.16.36.1
172.16.36.132
172.16.36.135
```

Łącząc ze sobą za pomocą potoku wiele poleceń `cut`, możemy z poszczególnych wierszy tekstu wyodrębnić adresy IP. Mając zatem już sprawdzony sposób na skanowanie wielu hostów naraz i odpowiednie przetwarzanie wyników, możemy nasze rozwiązanie zaimplementować w skrypcie powłoki. Przykład takiego skryptu został przedstawiony na listingu poniżej.

```
#!/bin/bash

if [ "$#" -ne 1 ]; then
echo "Usage - ./ping_sweep.sh [/24 network address]"
echo "Example - ./ping_sweep.sh 172.16.36.0"
echo "Example will perform an ICMP ping sweep of the 172.16.36.0/24 network and
output to an output.txt file"
exit
fi

prefix=$(echo $1 | cut -d '.' -f 1-3)

for addr in $(seq 1 254); do
hping3 $prefix.$addr—icmp -c 1 >> handle.txt;
done

grep len handle.txt | cut -d " " -f 2 | cut -d "=" -f 2 >> output.txt
rm handle.txt
```

W pierwszym wierszu skryptu zdefiniowana jest lokalizacja interpretera powłoki *bash*. Dalej znajduje się blok kodu, którego zadaniem jest ustalenie, czy w wierszu wywołania skryptu został podany odpowiedni argument wywołania. Jest to realizowane poprzez proste sprawdzenie, czy liczba argumentów wywołania jest różna od 1. Jeżeli oczekiwany argument wywołania nie został podany, na ekranie wyświetlany jest opis sposobu użycia i skrypt kończy działanie. W opisie działania możemy znaleźć informację, że argumentem wywołania skryptu powinien być adres podsieci `/24`. Kolejny wiersz skryptu wyodrębnia z podanego argumentu wywołania prefiks adresu sieci. Na przykład jeżeli podany adres to `192.168.11.0`, do zmiennej `prefix` przypisany zostanie adres `192.168.11`. Następnie polecenie `hping3` jest wywoływane w pętli dla każdego z adresów IP zakresu `/24`, a wyniki działania są zapisywane w pliku *handle.txt*.

Po zakończeniu działania pętli z pliku wyników za pomocą polecenia `grep` wyodrębniane są wiersze zawierające odpowiedzi z aktywnych hostów, które następnie są „przepuszczane" przez wiele potokowanych poleceń `cut`, wyodrębniających z nich same adresy IP. Otrzymana lista adresów IP jest zapisywana w pliku *output.txt*, a niepotrzebny, tymczasowy plik *handle.txt* jest usuwany z katalogu. Po zapisaniu skryptu w lokalnym katalogu na dysku możesz spróbować uruchomić go z poziomu okna terminala, wpisując w wierszu wywołania kropkę, prawy ukośnik i nazwę skryptu, tak jak to zostało przedstawione w przykładzie poniżej.

```
root@KaliLinux:~# ./ping_sweep.sh
Usage - ./ping_sweep.sh [/24 network address]
Example - ./ping_sweep.sh 172.16.36.0
Example will perform an ICMP ping sweep of the 172.16.36.0/24 network and output to
an output.txt file
root@KaliLinux:~# ./ping_sweep.sh 172.16.36.0

--- 172.16.36.1 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms

--- 172.16.36.2 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.5/0.5/0.5 ms

--- 172.16.36.3 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
*** {Pozostałe wiersze zostały pominięte} ***
```

Po zakończeniu działania skryptu i zapisaniu wyników działania w pliku *output.txt* możesz użyć polecenia `ls` do sprawdzenia, czy plik wynikowy został utworzony, a następnie za pomocą polecenia `cat` wyświetlić zawartość tego pliku.

```
root@KaliLinux:~# ls output.txt
output.txt
root@KaliLinux:~# cat output.txt
172.16.36.1
172.16.36.2
```

```
172.16.36.132
172.16.36.135
172.16.36.253
```

Po uruchomieniu skryptu na ekranie nadal wyświetlanych jest tak samo wiele informacji, jak to miało miejsce, kiedy wywoływaliśmy pierwszą pętlę „z ręki", ale od tej chwili lista aktywnych hostów już nam nie umknie, ponieważ za każdym razem zostanie bezpiecznie zapisana w pliku tekstowym na dysku.

## Jak to działa?

Aby użyć polecenia `hping3` do wykrywania hostów w dużych zakresach adresów IP, musieliśmy wprowadzić nieco nowych rozwiązań. W przedstawionej recepturze utworzyliśmy skrypt powłoki *bash*, który w pętli, za pomocą programu `hping3`, wysyłał żądania ICMP do kolejnych adresów IP. Poprawne działanie skryptu było możliwe, ponieważ udało nam się wcześniej zidentyfikować unikatowy ciąg znaków, za pomocą którego byliśmy w stanie odróżnić w skrypcie odpowiedź aktywnego hosta od braku odpowiedzi na przesłane żądanie. Dzięki zapisywaniu kolejnych wyników działania polecenia `hping3` i następnie filtrowaniu ich przy użyciu poleceń `grep` i `cut`, mogliśmy utworzyć efektywny skrypt, który pozwala na skanowanie całych zakresów adresów IP i zapisuje listę adresów IP aktywnych hostów w pliku tekstowym na dysku.

# Skanowanie sieci na warstwie 4. przy użyciu programu Scapy

Istnieje bardzo wiele metod wykrywania hostów z wykorzystaniem skanowania sieci na warstwie 4., które może być realizowane zarówno za pomocą protokołu **UDP** (ang. *User Datagram Protocol*), jak i **TCP** (ang. *Transmission Control Protocol*). Scapy to narzędzie, które pozwala użytkownikowi na tworzenie odpowiednio spreparowanych żądań przy użyciu obu wymienionych protokołów, co w połączeniu ze skryptami w języku Python pozwala na stworzenie wielu bardzo przydatnych narzędzi. W tej recepturze pokażemy, w jaki sposób możesz używać programu Scapy do wykrywania zdalnych hostów poprzez skanowanie sieci na warstwie 4. z wykorzystaniem protokołów TCP i UDP.

## Przygotuj się

Zastosowanie programu Scapy do wykrywania hostów na warstwie 4. nie wymaga żadnego środowiska testowego, ponieważ bardzo wiele systemów działających w sieci Internet domyślnie odpowiada na żądania TCP czy UDP. Z drugiej jednak strony, jeżeli nie znasz dokładnie przepisów prawa i regulacji, którym podlegasz w tym zakresie, rekomendowanym rozwiązaniem będzie zdecydowanie przeprowadzanie skanowania wyłącznie we własnym, dedykowanym

środowisku testowym. Aby można było za pomocą programu Scapy przeprowadzić skanowanie na warstwie 4., w środowisku testowym musi działać przynajmniej jeden system, który będzie odpowiadał na żądania TCP i (lub) UDP. Najlepszym kandydatem będzie system, na którym działa co najmniej jedna usługa TCP i co najmniej jedna usługa UDP. W przedstawionym przykładzie wykorzystujemy kombinację systemów Linux i Windows. Więcej szczegółowych informacji na temat instalacji i konfiguracji tych systemów w naszym środowisku testowym znajdziesz w rozdziale 1., w recepturach „Instalacja systemu Metasploitable2" oraz „Instalacja systemu Windows". Oprócz tego w tej recepturze będziemy używać edytorów tekstu, takich jak VIM czy Nano, do napisania skryptów skanujących i zapisania ich w systemie plików. Więcej szczegółowych informacji na temat pisania skryptów znajdziesz w rozdziale 1., w recepturze „Praca z edytorami tekstu VIM i Nano".

## Jak to zrobić?

Aby sprawdzić, czy badany host nadeśle odpowiedź z ustawioną flagą `RST`, możemy za pomocą pakietu Scapy wysłać do niego pakiet TCP z flagą `ACK`. W przykładzie przedstawionym poniżej pakiet z flagą `ACK` jest wysyłany do portu TCP 80, na którym zazwyczaj działają serwery WWW i inne usługi protokołu HTTP. W naszym przypadku na porcie 80 badanego hosta działa serwer Apache. Podobnie jak w poprzednich przykładach z programem Scapy, nasz pakiet musimy zbudować warstwa po warstwie, rozpoczynając od warstwy protokołu IP, co zostało przedstawione na listingu poniżej.

```
root@KaliLinux:~# scapy
Welcome to Scapy (2.2.0)
>>> i = IP()
>>> i.display()
###[ IP ]###
   version= 4
   ihl= None
   tos= 0x0
   len= None
   id= 1
   flags=
   frag= 0
   ttl= 64
   proto= ip
   chksum= None
   src= 127.0.0.1
   dst= 127.0.0.1
   \options\
>>> i.dst="172.16.36.135"
>>> i.display()
###[ IP ]###
   version= 4
   ihl= None
   tos= 0x0
```

```
    len= None
    id= 1
    flags=
    frag= 0
    ttl= 64
    proto= ip
    chksum= None
    src= 172.16.36.180
    dst= 172.16.36.135
    \options\
```

Najpierw utworzyliśmy nowy obiekt `IP`, który został przypisany do zmiennej `i`, a następnie odpowiednio ustawiliśmy adres IP docelowego hosta. Zwróć uwagę, że jeżeli docelowy adres IP zmieni się na inną wartość niż domyślna, jako źródłowy adres IP zostanie automatycznie wpisany adres IP domyślnego, lokalnego interfejsu sieciowego. Następnym krokiem będzie utworzenie warstwy TCP naszego pakietu. Aby to zrobić, wykonaj polecenia przedstawione poniżej.

```
>>> t = TCP()
>>> t.display()
###[ TCP ]###
   sport= ftp_data
   dport= http
   seq= 0
   ack= 0
   dataofs= None
   reserved= 0
   flags= S
   window= 8192
   chksum= None
   urgptr= 0
   options= {}
>>> t.flags='A'
>>> t.display()
###[ TCP ]###
   sport= ftp_data
   dport= http
   seq= 0
   ack= 0
   dataofs= None
   reserved= 0
   flags= A
   window= 8192
   chksum= None
   urgptr= 0
   options= {}
```

Na początek utworzyliśmy nowy obiekt `TCP`, który został przypisany do zmiennej `t`. Zauważ, że po utworzeniu obiektu jego port przeznaczenia (parametr `dport`) jest domyślnie ustawiany na wartość `http` (czyli na port 80). Dalej zmieniamy ustawienia flag TCP pakietu z domyślnej

S (SYN) na A (ACK). Na koniec pozostaje nam już tylko połączyć poszczególne warstwy w jeden pakiet, tak jak to zostało przedstawione poniżej.

```
>>> request = (i/t)
>>> request.display()
###[ IP ]###
   version= 4
   ihl= None
   tos= 0x0
   len= None
   id= 1
   flags=
   frag= 0
   ttl= 64
   proto= tcp
   chksum= None
   src= 172.16.36.180
   dst= 172.16.36.135
   \options\
###[ TCP ]###
      sport= ftp_data
      dport= http
      seq= 0
      ack= 0
      dataofs= None
      reserved= 0
      flags= A
      window= 8192
      chksum= None
      urgptr= 0
      options= {}
```

Gotowy pakiet zostaje przypisany do zmiennej request. Teraz możemy go już wysłać za pomocą funkcji sr1() i na podstawie otrzymanej odpowiedzi określić status badanego hosta.

```
>>> response = sr1(request)
Begin emission:
.......Finished to send 1 packets.
....*
Received 12 packets, got 1 answers, remaining 0 packets
>>> response.display()
###[ IP ]###
   version= 4L
   ihl= 5L
   tos= 0x0
   len= 40
   id= 0
   flags= DF
   frag= 0L
   ttl= 64
```

```
   proto= tcp
   chksum= 0x9974
   src= 172.16.36.135
   dst= 172.16.36.180
   \options\
###[ TCP ]###
      sport= http
      dport= ftp_data
      seq= 0
      ack= 0
      dataofs= 5L
      reserved= 0L
      flags= R
      window= 0
      chksum= 0xe21
      urgptr= 0
      options= {}
###[ Padding ]###
         load= '\x00\x00\x00\x00\x00\x00'
```

Zwróć uwagę, że system zdalny odpowiada na żądanie pakietem TCP z ustawioną flagą `RST`, o czym świadczy litera `R` w atrybucie `flags`. Poprzez bezpośrednie wywoływanie odpowiednich funkcji cały proces budowania pakietu, a następnie jego wysyłania i odbierania odpowiedzi można zmieścić w jednym poleceniu:

```
>>> response = sr1(IP(dst="172.16.36.135")/TCP(flags='A'))
.Begin emission:
................Finished to send 1 packets.
....*
Received 22 packets, got 1 answers, remaining 0 packets
>>> response.display()
###[ IP ]###
   version= 4L
   ihl= 5L
   tos= 0x0
   len= 40
   id= 0
   flags= DF
   frag= 0L
   ttl= 64
   proto= tcp
   chksum= 0x9974
   src= 172.16.36.135
   dst= 172.16.36.180
   \options\
###[ TCP ]###
      sport= http
      dport= ftp_data
      seq= 0
```

```
      ack= 0
      dataofs= 5L
      reserved= 0L
      flags= R
      window= 0
      chksum= 0xe21
      urgptr= 0
      options= {}
###[ Padding ]###
         load= '\x00\x00\x00\x00\x00\x00'
```

Wiemy już, jak wygląda odpowiedź na żądanie TCP z ustawioną flagą ACK, wysłane na otwarty port działającego hosta, spróbujemy więc teraz wysłać podobny pakiet na zamknięty port działającego hosta i zobaczymy, czy odpowiedzi różnią się od siebie.

```
>>> response = sr1(IP(dst="172.16.36.135")/TCP(dport=1111,flags='A'))
.Begin emission:
.........Finished to send 1 packets.
....*
Received 15 packets, got 1 answers, remaining 0 packets
>>> response.display()
###[ IP ]###
   version= 4L
   ihl= 5L
   tos= 0x0
   len= 40
   id= 0
   flags= DF
   frag= 0L
   ttl= 64
   proto= tcp
   chksum= 0x9974
   src= 172.16.36.135
   dst= 172.16.36.180
   \options\
###[ TCP ]###
      sport= 1111
      dport= ftp_data
      seq= 0
      ack= 0
      dataofs= 5L
      reserved= 0L
      flags= R
      window= 0
      chksum= 0xa1a
      urgptr= 0
      options= {}
###[ Padding ]###
         load= '\x00\x00\x00\x00\x00\x00'
```

W tym żądaniu numer docelowego portu TCP został zmieniony z 80 na 1111 (to numer portu, na którym nie działa żadna usługa). Zwróć jednak uwagę, że zarówno w przypadku otwartego portu, jak i zamkniętego nadesłane odpowiedzi są identyczne. Inaczej mówiąc, niezależnie od tego, czy skanowany port jest otwarty, czy zamknięty, działający system odpowiada pakietem TCP z ustawioną flagą RST. Warto również zauważyć, że jeżeli podobne żądanie wyślemy na adres IP, który nie jest przypisany do działającego systemu, nie otrzymamy żadnej odpowiedzi. Możemy to łatwo sprawdzić, odpowiednio zmieniając w naszym żądaniu docelowy adres IP:

```
>>> response = sr1(IP(dst="172.16.36.136")/TCP(dport=80,flags='A'),timeout=1)
Begin emission:
.....................................................................
.....................................................................
......Finished to send 1 packets.
.....................
Received 3559 packets, got 0 answers, remaining 1 packets
```

Podsumowując, udało nam się udowodnić, że pakiet TCP z ustawioną flagą ACK, wysłany na dowolny port działającego hosta, niezależnie od jego stanu, zawsze spowoduje odesłanie odpowiedzi z ustawioną flagą RST, ale jeżeli wyślemy taki pakiet na adres IP, który nie jest powiązany z działającym hostem, nie otrzymamy żadnej odpowiedzi. To naprawdę znakomita wiadomość, ponieważ oznacza to, że możemy przeprowadzić wykrywanie hostów w dużych podsieciach poprzez interakcję z pojedynczym portem na każdym z systemów. Korzystając z pakietu Scapy w połączeniu ze skryptem w języku Python, możemy szybko przechodzić w pętli przez kolejne adresy IP podsieci /24, na wybrany port każdego z nich wysyłać jeden pakiet TCP z ustawioną flagą ACK, a następnie na podstawie odsyłanych odpowiedzi (bądź ich braku) tworzyć listę aktywnych adresów IP.

```
#!/usr/bin/python

import logging
logging.getLogger("scapy.runtime").setLevel(logging.ERROR)
from scapy.all import *

if len(sys.argv) != 2:
    print "Usage - ./ACK_Ping.py [/24 network address]"
    print "Example - ./ACK_Ping.py 172.16.36.0"
    print "Example will perform a TCP ACK ping scan of the 172.16.36.0/24 range"
sys.exit()

address = str(sys.argv[1])
prefix = address.split('.')[0] + '.' + address.split('.')[1] + '.' +
address.split('.')[2] + '.'

for addr in range(1,254):
   response =
sr1(IP(dst=prefix+str(addr))/TCP(dport=80,flags='A'),timeout=1,verbose=0)
   try:
```

```
        if int(response[TCP].flags) == 4:
            print prefix+str(addr)
    except:
        pass
```

Przykładowy skrypt przedstawiony powyżej jest dosyć prosty. Pętla `for` przechodzi przez kolejne wartości ostatniego oktetu adresu IP i dla każdego z adresów wysyła na port `80` pakiet TCP z ustawioną flagą `ACK`. Następnie skrypt analizuje otrzymaną odpowiedź, sprawdzając, czy po dokonaniu konwersji flag na liczbę całkowitą otrzymamy wartość `4` (co odpowiada ustawionej fladze `RST`). Jeżeli pakiet ma ustawioną flagę `RST`, skrypt wyświetla na ekranie adres IP hosta, który nadesłał odpowiedź. Jeżeli z danego adresu nie zostanie odebrana żadna odpowiedź, interpreter języka Python nie będzie w stanie sprawdzić wartości zmiennej reprezentującej odpowiedź, ponieważ nie zostanie do niej wcześniej przypisana żadna wartość. W takiej sytuacji generowany jest wyjątek, po którego napotkaniu skrypt przechodzi do kolejnej iteracji pętli. Wynikiem działania skryptu jest lista adresów IP aktywnych hostów działających w badanej podsieci. Po zapisaniu skryptu w lokalnym katalogu na dysku możesz spróbować uruchomić go z poziomu okna terminala, wpisując w wierszu wywołania kropkę, prawy ukośnik i nazwę skryptu, tak jak to zostało przedstawione w przykładzie poniżej.

```
root@KaliLinux:~# ./ACK_Ping.py
Usage - ./ACK_Ping.py [/24 network address]
Example - ./ACK_Ping.py 172.16.36.0
Example will perform a TCP ACK ping scan of the 172.16.36.0/24 range
root@KaliLinux:~# ./ACK_Ping.py 172.16.36.0
172.16.36.1
172.16.36.2
172.16.36.132
172.16.36.135
```

Podobna metoda wykrywania hostów może być użyta do skanowania sieci na warstwie 4. z wykorzystaniem protokołu UDP. Aby sprawdzić, czy uda nam się wykryć hosta za pomocą protokołu UDP, musimy znaleźć sposób na to, by przy użyciu odpowiednio spreparowanego pakietu UDP wymusić odpowiedź hosta, niezależnie od tego, czy na badanym porcie UDP działa jakakolwiek usługa. Przygotowania rozpoczniemy od zbudowania odpowiedniego pakietu za pomocą programu Scapy:

```
root@KaliLinux:~# scapy
Welcome to Scapy (2.2.0)
>>> i = IP()
>>> i.dst = "172.16.36.135"
>>> u = UDP()
>>> request = (i/u)
>>> request.display()
###[ IP ]###
   version= 4
   ihl= None
   tos= 0x0
   len= None
```

```
   id= 1
   flags=
   frag= 0
   ttl= 64
   proto= udp
   chksum= None
   src= 172.16.36.180
   dst= 172.16.36.135
   \options\
###[ UDP ]###
      sport= domain
      dport= domain
      len= None
      chksum= None
```

Zauważ, że domyślnie port źródłowy i docelowy obiektu UDP jest ustawiony na usługę **DNS** (ang. *Domain Name System*). DNS to usługa sieciowa działająca na porcie 53, która pozwala na rozwiązywanie nazw domenowych i ich zamianę na adresy IP. Jednak wysłanie pakietu w obecnej postaci nie ułatwi nam zbytnio sprawdzenia, czy docelowy adres IP jest przypisany do działającego hosta. Aby się o tym przekonać, możesz wykonać polecenie przedstawione poniżej.

```
>>> reply = sr1(request,timeout=1,verbose=1)
Begin emission:
Finished to send 1 packets.

Received 7 packets, got 0 answers, remaining 1 packets
```

Jak widać, bez względu na fakt, że w naszym przypadku docelowy adres IP należy do aktywnego hosta, na wysłane żądanie nie otrzymaliśmy żadnej odpowiedzi. Paradoksalnie brak odpowiedzi jest spowodowany tym, że na docelowym systemie działa usługa DNS. Co ciekawe, czasami znacznie bardziej efektywnym sposobem wykrywania hostów w sieci może być wysyłanie pakietów UDP na porty, na których nie działają żadne usługi (zakładając, że ruch ICMP nie jest blokowany przez zaporę sieciową). Dzieje się tak, ponieważ bardzo wiele usług sieciowych jest skonfigurowanych tak, aby odpowiadać tylko na pakiety o ściśle określonej zawartości. Teraz spróbujemy ponownie wysłać nasz pakiet, ale tym razem na inny port UDP, który nie jest używany.

```
>>> u.dport = 123
>>> request = (i/u)
>>> reply = sr1(request,timeout=1,verbose=1)
Begin emission:
Finished to send 1 packets.

Received 5 packets, got 1 answers, remaining 0 packets
>>> reply.display()
###[ IP ]###
   version= 4L
   ihl= 5L
   tos= 0xc0
   len= 56
```

```
    id= 62614
    flags=
    frag= 0L
    ttl= 64
    proto= icmp
    chksum= 0xe412
    src= 172.16.36.135
    dst= 172.16.36.180
    \options\
###[ ICMP ]###
       type= dest-unreach
       code= port-unreachable
       chksum= 0x9e72
       unused= 0
###[ IP in ICMP ]###
          version= 4L
          ihl= 5L
          tos= 0x0
          len= 28
          id= 1
          flags=
          frag= 0L
          ttl= 64
          proto= udp
          chksum= 0xd974
          src= 172.16.36.180
          dst= 172.16.36.135
          \options\
###[ UDP in ICMP ]###
             sport= domain
             dport= ntp
             len= 8
             chksum= 0x5dd2
```

Po zmianie portu docelowego na 123 i ponownym wysłaniu pakietu otrzymaliśmy odpowiedź wskazującą, że badany port jest nieosiągalny (ang. *port unreachable*). Jeżeli sprawdzisz źródłowy adres IP otrzymanej odpowiedzi, przekonasz się, że została wysłana z hosta, do którego przesłaliśmy wcześniej nasze początkowe żądanie. Otrzymanie takiej odpowiedzi jest wystarczającym dowodem pozwalającym na potwierdzenie, że dany adres IP należy do aktywnego hosta. Niestety zdarza się jednak, że niektóre hosty w takiej sytuacji nie odsyłają żadnych odpowiedzi. Efektywność opisanej techniki zależy w dużej mierze od tego, jakie systemy próbujesz skanować i jak są skonfigurowane. Z tego względu wykrywanie hostów za pomocą protokołu UDP jest często znacznie trudniejsze niż w sytuacji, kiedy używamy protokołu TCP, a już z całą pewnością nie jest to tak proste jak wysyłanie pojedynczego pakietu TCP z ustawioną jedną flagą. Jeżeli na danym porcie działa określona usługa UDP, to bardzo często do efektywnego skanowania niezbędne będzie wysyłanie pakietów dostosowanych do wymogów takiej usługi.

Na szczęście istnieje całkiem sporo bardzo rozbudowanych narzędzi przeznaczonych do skanowania z wykorzystaniem protokołu UDP, które pozwalają na wysyłanie różnych rodzajów żądań UDP, dostosowanych do wielu różnych usług sieciowych, i wykrywanie hostów na podstawie otrzymanych odpowiedzi.

## Jak to działa?

W przedstawionych przykładach wykorzystywaliśmy skanowanie przy użyciu zarówno protokołu TCP, jak i UDP. Za pomocą pakietu Scapy budowaliśmy odpowiednio spreparowane pakiety, których używaliśmy do wykrywania hostów w sieci. W przypadku protokołu TCP tworzyliśmy pakiety z ustawioną flagą `ACK`, które następnie wysyłaliśmy na wybrany port poszczególnych systemów. Kiedy w odpowiedzi na takie żądanie otrzymywaliśmy pakiet z ustawioną flagą `RST`, uzyskiwaliśmy potwierdzenie, że host o danym adresie IP jest aktywny. W przypadku protokołu UDP wysyłaliśmy puste żądania UDP na arbitralnie wybrany port zdalnego hosta w nadziei, że uda nam się wymusić wysłanie odpowiedzi informujących, że port docelowy jest nieosiągalny (ang. *ICMP Port Unreachable*). Otrzymanie takiej odpowiedzi było dla nas potwierdzeniem, że host o danym adresie IP jest aktywny. Każda z opisanych technik może być wykorzystywana w skryptach języka Python, pozwalających na skanowanie wielu hostów lub nawet całych podsieci.

# Skanowanie sieci na warstwie 4. przy użyciu programu Nmap

W arsenale różnych technik skanowania wbudowanych w programie Nmap znajdziesz również opcję pozwalającą na wykrywanie hostów poprzez skanowanie hostów na warstwie 4. W tej recepturze pokażemy, w jaki sposób możesz użyć programu Nmap do skanowania sieci na warstwie 4. z wykorzystaniem protokołów TCP i UDP.

## Przygotuj się

Zastosowanie programu Nmap do wykrywania hostów na warstwie 4. nie wymaga żadnego środowiska testowego, ponieważ bardzo wiele systemów działających w sieci Internet domyślnie odpowiada na żądania TCP czy UDP. Z drugiej jednak strony, jeżeli nie znasz dokładnie przepisów prawa i regulacji, którym podlegasz w tym zakresie, rekomendowanym rozwiązaniem będzie zdecydowanie przeprowadzanie skanowania wyłącznie we własnym, dedykowanym środowisku testowym.

Aby można było za pomocą programu Nmap przeprowadzić skanowanie na warstwie 4., w środowisku testowym musi działać przynajmniej jeden system, który będzie odpowiadał na żądania TCP i (lub) UDP. Najlepszym kandydatem będzie system, na którym działa co najmniej jedna

usługa TCP i co najmniej jedna usługa UDP. W przedstawionym przykładzie wykorzystujemy kombinację systemów Linux i Windows. Więcej szczegółowych informacji na temat instalacji i konfiguracji tych systemów w naszym środowisku testowym znajdziesz w rozdziale 1., w recepturach „Instalacja systemu Metasploitable2" oraz „Instalacja systemu Windows".

## Jak to zrobić?

Nmap ma wiele opcji pozwalających na wykrywanie hostów poprzez skanowanie sieci z wykorzystaniem protokołów TCP i UDP. Mechanizm wykrywania hostów za pomocą skanowania UDP jest już wstępnie skonfigurowany i wykorzystuje unikatowe ładunki niezbędne do wywołania odpowiedzi z niektórych mniej „rozmownych" usług sieciowych. Aby przeprowadzić skanowanie UDP, powinieneś w wierszu wywołania Nmapa dodać opcję `-PU`, po której musisz wpisać numer portu do testowania:

```
root@KaliLinux:~# nmap 172.16.36.135 -PU53 -sn

Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-11 20:11 EST
Nmap scan report for 172.16.36.135
Host is up (0.00042s latency).
MAC Address: 00:0C:29:3D:84:32 (VMware)
Nmap done: 1 IP address (1 host up) scanned in 0.13 seconds
This UDP discovery scan can also be modified to perform a scan of a
sequential range by using dash notation. In the example provided, we will
scan the entire 172.16.36.0/24 address range:
root@KaliLinux:~# nmap 172.16.36.0-255 -PU53 -sn

Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-17 06:33 EST
Nmap scan report for 172.16.36.1
Host is up (0.00020s latency).
MAC Address: 00:50:56:C0:00:08 (VMware)
Nmap scan report for 172.16.36.2
Host is up (0.00018s latency).
MAC Address: 00:50:56:FF:2A:8E (VMware)
Nmap scan report for 172.16.36.132
Host is up (0.00037s latency).
MAC Address: 00:0C:29:65:FC:D2 (VMware)
Nmap scan report for 172.16.36.135
Host is up (0.00041s latency).
MAC Address: 00:0C:29:3D:84:32 (VMware)
Nmap scan report for 172.16.36.180
Host is up.
Nmap scan report for 172.16.36.254
Host is up (0.00015s latency).
MAC Address: 00:50:56:EB:E1:8A (VMware)
Nmap done: 256 IP addresses (6 hosts up) scanned in 3.91 seconds
```

W podobny sposób możesz użyć Nmapa do skanowania za pomocą protokołu UDP wielu adresów IP, których lista jest przechowywana w pliku. W przykładzie przedstawionym poniżej lista adresów IP znajduje się w pliku tekstowym o nazwie *iplist.txt*, zapisanym w bieżącym katalogu roboczym.

```
root@KaliLinux:~# nmap -iL iplist.txt -sn -PU53

Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-17 06:36 EST
Nmap scan report for 172.16.36.2
Host is up (0.00015s latency).
MAC Address: 00:50:56:FF:2A:8E (VMware)
Nmap scan report for 172.16.36.1
Host is up (0.00024s latency).
MAC Address: 00:50:56:C0:00:08 (VMware)
Nmap scan report for 172.16.36.135
Host is up (0.00029s latency).
MAC Address: 00:0C:29:3D:84:32 (VMware)
Nmap scan report for 172.16.36.132
Host is up (0.00030s latency).
MAC Address: 00:0C:29:65:FC:D2 (VMware)
Nmap scan report for 172.16.36.180
Host is up.
Nmap scan report for 172.16.36.254
Host is up (0.00021s latency).
MAC Address: 00:50:56:EB:E1:8A (VMware)
Nmap done: 6 IP addresses (6 hosts up) scanned in 0.31 seconds
```

Choć w wynikach działania programu możemy zobaczyć, że wykrytych zostało sześć hostów, nie oznacza to wcale, że wszystkie sześć hostów zostało znalezionych za pomocą skanowania UDP. Oprócz wysyłania żądań na port UDP o numerze 53 Nmap może automatycznie używać dowolnych innych technik skanowania, jakie mogą się okazać skuteczne do wykrywania hostów o podanych adresach IP. Choć zastosowanie opcji `-sn` uniemożliwia Nmapowi przeprowadzenie skanowania portów TCP, to jednak nie powoduje również ograniczenia aktywności Nmapa do naszych żądań UDP. Choć w praktyce nie istnieje żaden efektywny sposób na realizację wyłącznie takiego zadania, to jednak w razie potrzeby możesz sprawdzić, które hosty zostały wykryte przy użyciu żądań UDP, poprzez przechwytywanie i analizowanie ruchu sieciowego za pomocą programów takich jak Wireshark czy TCPdump. Warto również wspomnieć, że programu Nmap możesz używać do skanowania sieci z wykorzystaniem pakietów TCP z ustawioną flagą `ACK` w podobny sposób, w jaki to robiliśmy z programem Scapy. Aby przeprowadzić wykrywanie hostów za pomocą pakietów z flagą `ACK`, powinieneś w wierszu wywołania programu Nmap dodać opcję `-PA` i wpisać po niej numer portu, na który taki pakiet ma być wysłany.

```
root@KaliLinux:~# nmap 172.16.36.135 -PA80 -sn

Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-11 20:09 EST
Nmap scan report for 172.16.36.135
Host is up (0.00057s latency).
MAC Address: 00:0C:29:3D:84:32 (VMware)
Nmap done: 1 IP address (1 host up) scanned in 0.21 seconds
```

Metody wykrywania hostów za pomocą pakietów TCP z ustawioną flagą ACK możesz również używać do skanowania całych zakresów adresów IP lub adresów IP, których lista jest przechowywana w pliku tekstowym:

```
root@KaliLinux:~# nmap 172.16.36.0-255 -PA80 -sn

Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-17 06:46 EST
Nmap scan report for 172.16.36.132
Host is up (0.00033s latency).
MAC Address: 00:0C:29:65:FC:D2 (VMware)
Nmap scan report for 172.16.36.135
Host is up (0.00013s latency).
MAC Address: 00:0C:29:3D:84:32 (VMware)
Nmap scan report for 172.16.36.180
Host is up.
Nmap done: 256 IP addresses (3 hosts up) scanned in 3.43 seconds
root@KaliLinux:~# nmap -iL iplist.txt -PA80 -sn

Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-17 06:47 EST
Nmap scan report for 172.16.36.135
Host is up (0.00033s latency).
MAC Address: 00:0C:29:3D:84:32 (VMware)
Nmap scan report for 172.16.36.132
Host is up (0.00029s latency).
MAC Address: 00:0C:29:65:FC:D2 (VMware)
Nmap scan report for 172.16.36.180
Host is up.
Nmap done: 3 IP addresses (3 hosts up) scanned in 0.31 seconds
```

## Jak to działa?

Technika używana przez program Nmap do wykrywania hostów poprzez skanowanie sieci z wykorzystaniem protokołu TCP opiera się na tych samych podstawowych zasadach co program Scapy. Nmap wysyła serię pakietów TCP z ustawioną flagą ACK na wybrane porty badanych systemów, próbuje wymusić tym samym odesłanie przez system zdalny odpowiedzi z ustawioną flagą RST, które wskazywałyby, że dany system jest aktywny. Jednak do skanowania sieci z wykorzystaniem protokołu UDP Nmap używa już nieco innej techniki niż program Scapy. Zamiast polegać na odpowiedziach *ICMP Port Unreachable*, które mogą być niejednoznaczne lub blokowane, Nmap próbuje wykrywać zdalne hosty poprzez wysyłanie odpowiednio spreparowanych żądań, dostosowanych do poszczególnych usług sieciowych, i oczekiwanie, że tak dostosowane pakiety spowodują odesłanie odpowiedzi.

# Skanowanie sieci na warstwie 4. przy użyciu programu hping3

W jednej z poprzednich receptur w tym rozdziale używaliśmy polecenia `hping3` do skanowania sieci na warstwie 3. z wykorzystaniem protokołu ICMP. Warto jednak zauważyć, że polecenie `hping3` umożliwia również wykrywanie hostów za pomocą protokołów TCP i UDP. Jak już jednak wspominaliśmy wcześniej, program `hping3` został zaprojektowany do skanowania pojedynczych hostów, stąd aby „zamienić go" w pełnowymiarowe, efektywne narzędzie skanujące, będziemy musieli się posłużyć małym skryptem. W tej recepturze pokażemy, w jaki sposób za pomocą programu `hping3` możesz przeprowadzić skanowanie sieci na warstwie 4. z wykorzystaniem protokołów TCP i UDP.

## Przygotuj się

Zastosowanie polecenia `hping3` do wykrywania hostów na warstwie 4. nie wymaga żadnego środowiska testowego, ponieważ bardzo wiele systemów działających w sieci Internet domyślnie odpowiada na żądania TCP czy UDP. Z drugiej jednak strony, jeżeli nie znasz dokładnie przepisów prawa i regulacji, którym podlegasz w tym zakresie, rekomendowanym rozwiązaniem będzie zdecydowanie przeprowadzanie skanowania wyłącznie we własnym, dedykowanym środowisku testowym. Aby można było za pomocą polecenia `hping3` przeprowadzić skanowanie na warstwie 4., w środowisku testowym musi działać przynajmniej jeden system, który będzie odpowiadał na żądania TCP i (lub) UDP. Najlepszym kandydatem będzie system, na którym działa co najmniej jedna usługa TCP i co najmniej jedna usługa UDP. W przedstawionym przykładzie wykorzystujemy kombinację systemów Linux i Windows. Więcej szczegółowych informacji na temat instalacji i konfiguracji tych systemów w naszym środowisku testowym znajdziesz w rozdziale 1., w recepturach „Instalacja systemu Metasploitable2" oraz „Instalacja systemu Windows". Oprócz tego w tej recepturze będziemy używać edytorów tekstu, takich jak VIM czy Nano, do napisania skryptów skanujących i zapisania ich w systemie plików. Więcej szczegółowych informacji na temat pisania skryptów znajdziesz w rozdziale 1., w recepturze „Praca z edytorami tekstu VIM i Nano".

## Jak to zrobić?

W przeciwieństwie do sytuacji z programem Nmap, korzystanie z polecenia `hping3` pozwala na jednoznaczną identyfikację hostów, które zostały wykryte za pomocą skanowania UDP. Dzięki zastosowaniu w wierszu wywołania opcji `--udp` polecenie `hping3` wysyła na podany adres docelowy żądania UDP, próbując wymusić na systemie zdalnym odesłanie odpowiedzi.

```
root@KaliLinux:~# hping3 --udp 172.16.36.132
HPING 172.16.36.132 (eth1 172.16.36.132): udp mode set, 28 headers + 0 data bytes
ICMP Port Unreachable from ip=172.16.36.132 name=UNKNOWN
status=0 port=2792 seq=0
```

```
ICMP Port Unreachable from ip=172.16.36.132 name=UNKNOWN
status=0 port=2793 seq=1
ICMP Port Unreachable from ip=172.16.36.132 name=UNKNOWN
status=0 port=2794 seq=2
^FICMP Port Unreachable from ip=172.16.36.132 name=UNKNOWN
status=0 port=2795 seq=3
^C
--- 172.16.36.132 hping statistic ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 1.8/29.9/113.4 ms
```

Jak widać w powyższym przykładzie, proces skanowania został przerwany przez użytkownika naciśnięciem kombinacji klawiszy *Ctrl+C*. Kiedy polecenie `hping3` działa w trybie UDP, proces wykrywania hostów działa w nieskończonej pętli dopóty, dopóki nie zostanie przerwany przez użytkownika lub dopóki nie zostanie osiągnięty limit wysyłanych pakietów, który może być opcjonalnie zdefiniowany w wierszu wywołania polecenia. Aby zdefiniować maksymalną liczbę wysyłanych pakietów, powinieneś w wierszu wywołania polecenia dodać opcję `-c`, po której należy wpisać żądaną liczbę pakietów.

```
root@KaliLinux:~# hping3 --udp 172.16.36.132 -c 1
HPING 172.16.36.132 (eth1 172.16.36.132): udp mode set, 28 headers + 0 data bytes
ICMP Port Unreachable from ip=172.16.36.132 name=UNKNOWN
status=0 port=2422 seq=0

--- 172.16.36.132 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 104.8/104.8/104.8 ms
```

Choć polecenie `hping3` nie oferuje mechanizmów pozwalających na skanowanie wielu hostów jednocześnie, ograniczenie to możemy łatwo zniwelować za pomocą prostego skryptu powłoki *bash*. Aby to zrobić, musimy najpierw zidentyfikować różnice pomiędzy odpowiedziami, jakie otrzymujemy dla aktywnych i nieaktywnych hostów. Odpowiedź dla aktywnego hosta możemy zobaczyć w przykładzie powyżej, więc pozostało nam już tylko wypróbować takie samo polecenie dla adresu IP, który nie jest powiązany z żadnym działającym hostem:

```
root@KaliLinux:~# hping3 --udp 172.16.36.131 -c 1
HPING 172.16.36.131 (eth1 172.16.36.131): udp mode set, 28 headers + 0 data bytes

--- 172.16.36.131 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
```

Dokładnie analizując obie odpowiedzi, możemy znaleźć unikatowy ciąg znaków, pozwalający na zidentyfikowanie aktywnych hostów, który będziemy mogli wyodrębniać z wyników działania za pomocą polecenia `grep`. Kiedy przyjrzysz się dokładnie naszym przykładom, z pewnością zauważysz, że fraza `ICMP Port Unreachable` pojawia się w wynikach działania tylko wtedy, gdy ze zdalnego hosta została odebrana odpowiedź na przesłane żądanie. Znamy tę właściwość, możemy więc wyświetlić informacje o aktywnych hostach, wyodrębniając za pomocą polecenia

grep wiersze zawierające słowo Unreachable. Aby przetestować efektywność takiego rozwiązania, spróbujemy teraz połączyć dwa poprzednie polecenia i następnie przesłać wyniki ich działania do polecenia grep. Przyjmując, że nasze założenie jest prawdziwe, w rezultacie wykonania takiego złożonego polecenia powinniśmy otrzymać tylko dane powiązane z aktywnym hostem.

```
root@KaliLiniux:~# hping3 --udp 172.16.36.132 -c 1; hping3 --udp
172.16.36.131 -c 1 | grep "Unreachable"
HPING 172.16.36.132 (eth1 172.16.36.132): udp mode set, 28 headers + 0 data bytes
ICMP Port Unreachable from ip=172.16.36.132 name=UNKNOWN
status=0 port=2836 seq=0

--- 172.16.36.132 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 115.2/115.2/115.2 ms

--- 172.16.36.131 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
```

Pomimo że otrzymaliśmy mniej więcej taki wynik, o jaki nam chodziło, nietrudno zauważyć, że polecenie grep nie zostało w tym przypadku zastosowane zbyt efektywnie. Ponieważ polecenie hping3 wyświetla wyniki w dosyć specyficzny sposób, efektywne filtrowanie wierszy za pomocą polecenia grep jest mocno utrudnione, spróbujemy zatem osiągnąć zamierzony efekt w nieco inny sposób. Zamiast bezpośrednio filtrować dane z wyjścia polecenia hping3, przekierujemy je do pliku, a dopiero potem spróbujemy użyć polecenia grep. Aby to zrobić, zapiszemy wyniki działania obu używanych wcześniej poleceń ping3 w pliku o nazwie *handle.txt*:

```
root@KaliLinux:~# hping3 --udp 172.16.36.132 -c 1 >> handle.txt

--- 172.16.36.132 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 28.6/28.6/28.6 ms
root@KaliLinux:~# hping3 --udp 172.16.36.131 -c 1 >> handle.txt

--- 172.16.36.131 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
root@KaliLinux:~# ls
Desktop handle.txt
root@KaliLinux:~# cat handle.txt
HPING 172.16.36.132 (eth1 172.16.36.132): udp mode set, 28 headers + 0 data bytes
ICMP Port Unreachable from ip=172.16.36.132 name=UNKNOWN
status=0 port=2121 seq=0
HPING 172.16.36.131 (eth1 172.16.36.131): udp mode set, 28 headers + 0 data bytes
```

Choć nasza próba zakończyła się tylko częściowym sukcesem, ponieważ nie wszystkie dane z wyjścia poleceń hping3 zostały zapisane w pliku, to jednak możemy zauważyć, że to, co się „zapisało”,

w zupełności wystarczy nam do przygotowania w pełni funkcjonalnego skryptu powłoki — w pliku tekstowym zapisywane są tylko wiersze wyników powiązane z aktywnymi hostami, zawierające wiele informacji, łącznie z adresem IP. Aby sprawdzić, czy takie podejście sprawdzi się w praktyce, spróbujemy teraz przygotować nieco bardziej złożone polecenie, które będzie przechodziło w pętli przez kolejne adresy IP zakresu /24, dla każdego z adresów wywoływało polecenie `hping3` i przekazywało wyniki jego działania do pliku *handle.txt*.

```
root@KaliLinux:~# for addr in $(seq 1 254); do hping3 --udp
172.16.36.$addr -c 1 >> handle.txt; done

--- 172.16.36.1 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms

--- 172.16.36.2 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms

--- 172.16.36.3 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
*** {Pozostałe wiersze zostały pominięte} ***
```

Po wykonaniu takiego polecenia możemy się przekonać, że wyniki jego działania są bardzo obszerne (fragment zamieszczony w powyższym przykładzie został celowo przycięty) i zawierają wiele dodatkowych wierszy, które nie są zapisywane w pliku na dysku. Pamiętaj jednak, że powodzenie całego przedsięwzięcia nie zależy od tego, jak bardzo „gadatliwa" jest nasza pętla, ale od tego, czy będziemy w stanie z pliku wynikowego wyodrębnić interesujące nas dane, tak jak to zostało przedstawione w przykładzie poniżej.

```
root@KaliLinux:~# ls
Desktop handle.txt
root@KaliLinux:~# grep Unreachable handle.txt
ICMP Port Unreachable from ip=172.16.36.132 HPING 172.16.36.133 (eth1
172.16.36.133): udp mode set, 28 headers + 0 data bytes
ICMP Port Unreachable from ip=172.16.36.135 HPING 172.16.36.136 (eth1
172.16.36.136): udp mode set, 28 headers + 0 data bytes
```

Po zakończeniu działania pętli skanującej możesz za pomocą polecenia `ls` sprawdzić, czy plik tekstowy został utworzony zgodnie z oczekiwaniami, a następnie przy użyciu polecenia `grep` możesz spróbować wyodrębnić z niego wszystkie wiersze zawierające ciąg znaków `Unreachable`. Wyniki działania tego polecenia dadzą nam listę wierszy zawierających informacje o aktywnych hostach, wykrytych za pomocą skanowania UDP. W tym momencie pozostało nam już tylko wyodrębnić z poszczególnych wierszy same adresy IP, a następnie połączyć to wszystko w jeden funkcjonalny skrypt. Sposób „wycinania" adresów IP został przedstawiony poniżej.

```
root@KaliLinux:~# grep Unreachable handle.txt
```

```
ICMP Port Unreachable from ip=172.16.36.132 HPING 172.16.36.133 (eth1 172.16.36.133):
udp mode set, 28 headers + 0 data bytes
ICMP Port Unreachable from ip=172.16.36.135 HPING 172.16.36.136 (eth1 172.16.36.136):
udp mode set, 28 headers + 0 data bytes
root@KaliLinux:~# grep Unreachable handle.txt | cut -d " " -f 5
ip=172.16.36.132
ip=172.16.36.135
root@KaliLinux:~# grep Unreachable handle.txt | cut -d " " -f 5 | cut -d "=" -f 2
172.16.36.132
172.16.36.135
```

Łącząc ze sobą za pomocą potoku wiele poleceń `cut`, możemy z poszczególnych wierszy tekstu wyodrębnić adresy IP aktywnych hostów. Skoro mamy już sprawdzony sposób na skanowanie wielu hostów naraz i odpowiednie przetwarzanie wyników, możemy nasze rozwiązanie zaimplementować w skrypcie powłoki. Przykład takiego skryptu został przedstawiony na listingu poniżej.

```
#!/bin/bash

if [ "$#" -ne 1 ]; then
   echo "Usage - ./udp_sweep.sh [/24 network address]"
   echo "Example - ./udp_sweep.sh 172.16.36.0"
   echo "Example will perform a UDP ping sweep of the 172.16.36.0/24
   network and output to an output.txt file"
   exit
fi

prefix=$(echo $1 | cut -d '.' -f 1-3)

for addr in $(seq 1 254); do
   hping3 $prefix.$addr --udp -c 1 >> handle.txt;
done

grep Unreachable handle.txt | cut -d " " -f 5 | cut -d "=" -f 2 >> output.txt
rm handle.txt
```

W pierwszym wierszu skryptu zdefiniowana jest lokalizacja interpretera powłoki *bash*. Dalej znajduje się blok kodu, którego zadaniem jest sprawdzenie, czy w wierszu wywołania skryptu został podany odpowiedni argument wywołania. Jest to realizowane poprzez proste sprawdzenie, czy liczba argumentów wywołania jest różna od `1`. Jeżeli oczekiwany argument wywołania nie został podany, na ekranie wyświetlany jest opis sposobu użycia i skrypt kończy działanie. W opisie działania możemy znaleźć informację, że argumentem wywołania skryptu powinien być adres podsieci `/24`. Kolejny wiersz skryptu wyodrębnia z podanego argumentu wywołania prefiks adresu sieci. Na przykład jeżeli podany adres to `192.168.11.0`, do zmiennej `prefix` przypisany zostanie adres `192.168.11`. Następnie polecenie `hping3` jest wywoływane w pętli dla każdego z adresów IP zakresu `/24`, a wyniki działania są zapisywane w pliku *handle.txt*. Po zakończeniu działania pętli z pliku wyników za pomocą polecenia `grep` wyodrębniane są wiersze zawierające odpowiedzi z aktywnych hostów, które następnie są „przepuszczane"

przez wiele potokowanych poleceń cut, wyodrębniających z nich same adresy IP. Otrzymana lista adresów IP jest zapisywana w pliku *output.txt*, a niepotrzebny tymczasowy plik *handle.txt* jest usuwany z katalogu.

```
root@KaliLinux:~# ./udp_sweep.sh
Usage - ./udp_sweep.sh [/24 network address]
Example - ./udp_sweep.sh 172.16.36.0
Example will perform a UDP ping sweep of the 172.16.36.0/24 network and output to an
output.txt file

root@KaliLinux:~# ./udp_sweep.sh 172.16.36.0

--- 172.16.36.1 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms

--- 172.16.36.2 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms

--- 172.16.36.3 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms

*** {Pozostałe wiersze zostały pominięte} ***

root@KaliLinux:~# ls output.txt
output.txt
root@KaliLinux:~# cat output.txt
172.16.36.132
172.16.36.135
172.16.36.253
```

Po uruchomieniu skryptu na ekranie nadal wyświetlanych jest tak samo wiele informacji, jak to miało miejsce, kiedy wywoływaliśmy pierwszą pętlę „z ręki", ale od tej chwili lista aktywnych hostów już nam nie umknie, ponieważ za każdym razem zostanie bezpiecznie zapisana w pliku tekstowym na dysku. Polecenia hping3 możesz również używać do wykrywania hostów z wykorzystaniem protokołu TCP. Co więcej, tryb TCP jest dla polecenia hping3 domyślnym trybem skanowania, używanym w sytuacji, kiedy w wierszu polecenia nie zostanie zdefiniowany inny tryb działania, a argumentem wywołania jest po prostu adres IP.

```
root@KaliLinux:~# hping3 172.16.36.132
HPING 172.16.36.132 (eth1 172.16.36.132): NO FLAGS are set, 40 headers + 0 data bytes
len=46 ip=172.16.36.132 ttl=64 DF id=0 sport=0 flags=RA seq=0 win=0 rtt=3.7 ms
len=46 ip=172.16.36.132 ttl=64 DF id=0 sport=0 flags=RA seq=1 win=0 rtt=0.7 ms
len=46 ip=172.16.36.132 ttl=64 DF id=0 sport=0 flags=RA seq=2 win=0 rtt=2.6 ms
^C

--- 172.16.36.132 hping statistic ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.7/2.3/3.7 ms
```

W podobny sposób do tego, w jaki tworzyliśmy skrypt powłoki *bash* dla skanowania UDP, możemy utworzyć skrypt pozwalający na wykrywanie hostów w podsieciach /24 za pomocą protokołu TCP. Najpierw musimy znaleźć unikatowy ciąg znaków, który pozwoli na odróżnienie wyników działania dla aktywnych i nieaktywnych hostów. Aby to zrobić, musimy wykonać polecenia przedstawione poniżej.

```
root@KaliLinux:~# hping3 172.16.36.132 -c 1
HPING 172.16.36.132 (eth1 172.16.36.132): NO FLAGS are set, 40 headers + 0 data bytes
len=46 ip=172.16.36.132 ttl=64 DF id=0 sport=0 flags=RA seq=0 win=0 rtt=3.4 ms

--- 172.16.36.132 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 3.4/3.4/3.4 ms
root@KaliLinux:~# hping3 172.16.36.131 -c 1
HPING 172.16.36.131 (eth1 172.16.36.131): NO FLAGS are set, 40 headers + 0 data bytes

--- 172.16.36.131 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
```

W tym przypadku unikatowym ciągiem znaków może być `length`, który pojawia się wyłącznie w wynikach działania polecenia `hping3` dla aktywnych hostów. Podobnie jak poprzednio, możemy napisać skrypt, który będzie zapisywał wyniki działania w tymczasowym pliku tekstowym na dysku, a następnie za pomocą poleceń `grep` i `cut` wyodrębniał listę adresów IP aktywnych hostów.

```
#!/bin/bash

if [ "$#" -ne 1 ]; then
   echo "Usage - ./tcp_sweep.sh [/24 network address]"
   echo "Example - ./tcp_sweep.sh 172.16.36.0"
   echo "Example will perform a TCP ping sweep of the 172.16.36.0/24
   network and output to an output.txt file"
   exit
fi

prefix=$(echo $1 | cut -d '.' -f 1-3)

for addr in $(seq 1 254); do
   hping3 $prefix.$addr -c 1 >> handle.txt;
done

grep len handle.txt | cut -d " " -f 2 | cut -d "=" -f 2 >> output.txt
rm handle.txt
```

Przedstawiony skrypt działa w bardzo podobny sposób do skryptu, który utworzyliśmy dla skanowania UDP. Jedyne różnice to polecenie wykonywane w pętli, ciąg znaków wyszukiwany poleceniem `grep` oraz sposób wyodrębniania adresu IP. Po uruchomieniu skrypt tworzy plik tekstowy o nazwie *output.txt*, w którym znajduje się lista adresów IP aktywnych hostów wykrytych podczas skanowania TCP.

```
root@KaliLinux:~# ./tcp_sweep.sh
Usage - ./tcp_sweep.sh [/24 network address]
Example - ./tcp_sweep.sh 172.16.36.0
Example will perform a TCP ping sweep of the 172.16.36.0/24 network and output to an
output.txt file
root@KaliLinux:~# ./tcp_sweep.sh 172.16.36.0

--- 172.16.36.1 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms

--- 172.16.36.2 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.6/0.6/0.6 ms

--- 172.16.36.3 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms

*** {Pozostałe wiersze zostały pominięte} ***
```

Po zakończeniu działania skryptu i zapisaniu wyników działania w pliku *output.txt*, możesz użyć polecenia `ls` do sprawdzenia, czy plik wynikowy został utworzony, a następnie za pomocą polecenia `cat` wyświetlić zawartość tego pliku, tak jak to zostało przedstawione w przykładzie poniżej.

```
root@KaliLinux:~# ls output.txt
output.txt
root@KaliLinux:~# cat output.txt
172.16.36.1
172.16.36.2
172.16.36.132
172.16.36.135
172.16.36.253
```

## Jak to działa?

W przykładach omawianych w tej recepturze używaliśmy polecenia `hping3` do skanowania UDP i wykrywania hostów w sieci na podstawie odpowiedzi *ICMP Host Unreachable*. Polecenie to było również używane do wykrywania hostów za pomocą skanowania TCP typu *null flag*. Podczas wykrywania hostów z wykorzystaniem protokołu UDP serie żądań były wysyłane do arbitralnie wybranych portów zdalnych hostów w nadziei na wymuszenie odesłania odpowiedzi. W przypadku skanowania TCP do portu `0` zdalnych hostów wysyłaliśmy żądania, w których nie były ustawione żadne flagi. W omawianym przykładzie powodowało to odesłanie przez zdalnego hosta odpowiedzi z ustawionymi flagami `ACK+RST`. W obu przykładach utworzyliśmy skrypty powłoki *bash*, umożliwiające skanowanie całych podsieci bądź wielu adresów IP, których lista była przechowywana w pliku tekstowym na dysku.

# Skorowidz

## A



## B



## C



## D

## E



## F



## H



## I



## J



## K

## L



## Ł



## M



## N



## O



## P

## R



## S

## Ś



## T



## U



## V



## W

## X



## Z



## Ż

# Skanowanie sieci z Kali Linux
## Receptury

W dzisiejszych czasach większość systemów informatycznych na świecie podłączona jest do Internetu. To rozwiązanie ma jedną wadę — dowolna osoba może spróbować przełamać zabezpieczenia sieci i uzyskać nieuprawniony dostęp do danych przetwarzanych w Twoich systemach. Jak temu zaradzić? Odpowiedź znajdziesz w tej książce.

Kali Linux to specjalna dystrybucja systemu Linux, która ułatwia skanowanie sieci pod kątem bezpieczeństwa. Zawarty w niej zestaw narzędzi pozwala na analizę najczęstszych problemów, a co za tym idzie, dzięki niej możesz błyskawicznie uszczelnić Twój system, tak aby przełamanie zabezpieczeń nie było prostym zadaniem. Wszystko, co musisz zrobić, zostało tu przedstawione w formie receptur. Sięgnij po tę książkę i przekonaj się, jak wykrywać hosty dostępne w sieci, skanować sieć za pomocą narzędzi arping, Nmap lub NetDiscover oraz odkrywać otwarte porty. Dowiedz się, w jaki sposób możesz zdalnie zidentyfikować uruchomiony system operacyjny, przeprowadzać ataki DoS (ang. *Denial of Service*) oraz testować aplikacje sieciowe. To doskonała lektura, która w rękach wprawnego administratora pozwoli na zdecydowane zwiększenie bezpieczeństwa sieci. Warto w nią zainwestować!

**Sięgnij po tę książkę i:**

- poznaj narzędzia dostępne w dystrybucji Kali Linux
- zlokalizuj hosty dostępne w Twojej sieci
- zidentyfikuj otwarte porty
- poznaj atak typu DoS
- zwiększ bezpieczeństwo Twojej sieci

**Justin Hutchens** — starszy konsultant ds. bezpieczeństwa w GuidePoint Security. Posiadacz wielu certyfikatów związanych z bezpieczeństwem (m.in. CISSP, OSCP, eMAPT, eWPT). Na co dzień zajmuje się przeprowadzaniem testów penetracyjnych. Pisze książki oraz artykuły poświęcone bezpieczeństwu sieci.

**Poznaj najlepsze przepisy na zwiększenie bezpieczeństwa Twojej sieci!**

[PACKT] open source PUBLISHING community experience distilled

Helion

35747 numer katalogowy
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

ISBN 978-83-283-0987-6

Informatyka w najlepszym wydaniu

cena: 69,00 zł