Digital Whisper Technologic Papers

Promiscuous Mode Detection

מאת בניה

הקדמה

רשתות מחשבים מורכבות מצמתים (nodes[1]). צומת הוא שם כללי לרכיב המסוגל לקבל ולשלוח מידע באופן אקטיבי. הפלאפון, המחשב והראוטר שלכם- כולם נחשבים צמתים, ולכולם דבר אחד משותף המגדיר צומת - לכולם יש כרטיס רשת (NIC) שהוא רכיב חומרה המאפשר את יכולות התקשורת של הרכיב.

לכרטיס הרשת משויכת כתובת הנקראת כתובת MAC ובאמצעותה ניתן לזהות את הרכיב ולשלוח לו מידע.

תוכנת רחרחן (sniffer) היא תוכנה הנמצאת על אחד הצמתים ברשת, ומקליטה את התעבורה העוברת דרכו. רחרחנים (יקראו מעתה והלאה sniffers) יכולים לסייע במגוון משימות - לגיטימיות וזדוניות כאחד - החל בניתוח בעיות ברשת ואיתור רכיבים שאינם מתפקדים כמצופה וכלה בגניבת מידע (סיסמאות, פרטי אשראי, מידע מסווג וכו') שאינו מוצפן ובמיפוי הרשת לקראת תקיפתה.

כרטיס רשת עושה שימוש בכתובת ה-MAC הייחודית[2] שלו על מנת לסנן תעבורה שאינה מיועדת למחשב עצמו.

על מנת ש-sniffer יראה את כל התעבורה, כולל כזו שאינה מיועדת למחשב עליו הוא יושב, עליו להכניס את כרטיס הרשת למצב פרוץ (Promiscuous Mode). במצב זה, כלל התעבורה תחלוף על פני כרטיס הרשת ללא סינון ותטופל ע"י רכיבי התוכנה הנמצאים בליבת מערכת ההפעלה ואחראים לטיפול בתעבורת רשת (Kernel Network Stack).

---

[1] https://he.wikipedia.org/wiki/%D7%A6%D7%95%D7%9E%D7%AA_(%D7%A8%D7%A9%D7%AA)

[2] למעשה, כתובות MAC עשויות שלא להיות ייחודיות. להרחבה:
https://www.howtogeek.com/228286/how-is-the-uniqueness-of-mac-addresses-enforced

בשוק ישנו מגוון רחב של מוצרי הסנפה, בהדגמות של מאמר זה אשתמש בעיקר בWireshark אולם מרבית מוצרי ההסנפה האחרים רלוונטיים בדיוק באותה מידה. להלן רשימה חלקית (מתוך ויקיפדיה) של מוצרי הסנפה (כמו גם כלי ניטור, אבטחה ותקיפה) העושים שימוש ב-Promiscuous Mode:

The following applications and applications classes use promiscuous mode.

Packet Analyzer

- NetScout Sniffer
- Wireshark (formerly *Ethereal*)
- tcpdump
- OmniPeek
- Capsa
- ntop
- Firesheep

Virtual machine

- VMware's VMnet bridging
- VirtualBox bridging mode

Cryptanalysis

- Aircrack-ng
- AirSnort
- Cain and Abel

Network monitoring

- KisMAC (used for WLAN)
- Kismet

ישנה חשיבות מבחינתנו, כחוקרי רשתות או מומחי אבטחה, להכיר את הרשת ולדעת אלו רכיבים ותוכנות רצות עליה. זיהוי כרטיסי רשת פרוצים היא יכולת חשובה בסט הכלים של האנליסט. האנליסט יכול להשתמש ביכולת זו בתור כלי ניטור שיכול להוביל למציאתו של תוקף.

במאמר זה אפרט ואדגים טכניקות שונות לזיהוי כרטיסי רשת פרוצים (Promiscuous Mode). חשוב לי להדגיש, זהו אינו מדע מדויק. הניסיון לאבחן האם כרטיס רשת הוא פרוץ או לא מתבסס על תגובתו של המחשב החשוד. היעד יכול שלא להתנהג כמצופה כתוצאה ממגוון סיבות ואז אנו עשויים לקבל תוצאות שגויות.

השיטה הטובה ביותר להשיג תוצאות ברמת מהימנות גבוהה היא לשלב מספר טכניקות יחדיו- וכן, לפענח את חלק מתוצאות הבדיקות באופן ידני.

את רוב הטכניקות והניסויים ערכתי מול מכונות וירטואליות בעלי כרטיסי רשת וירטואליים. ייתכן ונראה הבדלים מסוימים בין התוצאות המוצגות במאמר לתוצאות ב"עולם האמתי".

ידע מוקדם:

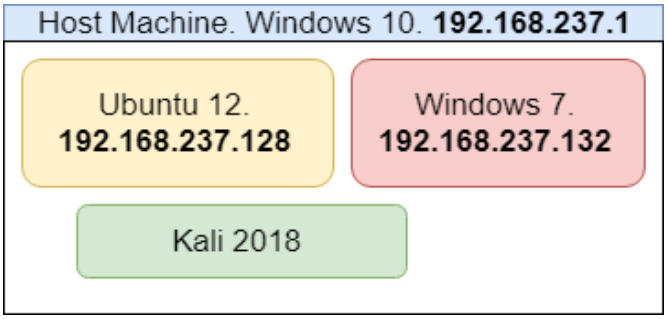
אני מניח שלקורא יש:

- הכרות בסיסית עם רכיבי רשת מקומית (כגון Hub ו-Switch).
- הבנה כללית של פרוטוקולי רשת נפוצים (ARP, IP, ICMP, DNS).
- ניסיון בסיסי עם Linux (Ubuntu).
- ניסיון תכנותי. הכרות בסיסית עם Threading.

לכל מושג חדש שאציג אצרף קישור, כך שקוראים ללא הרקע המתאים יוכלו להשלימו ולהמשיך בקריאה. לאורך המאמר מצורפים קטעי קוד ב-python, שברובם ככולם השתמשתי בספריית scapy להרכבת ושליחת פקטות. ניסיתי להסביר את הקוד כמיטב יכולתי. לקוראים שעדיין מתקשים בהבנתו, אני ממליץ לקרוא את הדוקומנטציה של הפונקציות העיקריות בהן עשיתי שימוש.

סביבת העבודה:

את הבדיקות ערכתי מול מכונת windows 7 ומכונת Ubuntu 12. חלק מהבדיקות אומתו גם מול מכונת Kali 2018.4. המחשב המארח מריץ Windows 10:



מילה על רחרחנים (sniffers):

בקווים כלליים, רחרחנים או סניפרים נחלקים לשניים- סניפרים אקטיביים ופסיביים.

סניפרים אקטיביים הם סניפרים המשנים את מבנה/קונפיגורצית הרשת באופן אקטיבי כך שהתעבורה תוזרם לתוקף. כיום רוב הרשתות מודרניות מכילות רכיב הנקרא מתג (switch) שמנתב את התעבורה באופן חכם - כך שכל מחשב יקבל רק את התעבורה שמיועדת אליו.

כלומר, גם אם התוקף מצותת לרשת וכרטיס הרשת שלו פרוץ- אם הוא לא ישנה את מבנה הרשת באופן אקטיבי- הוא יקבל רק התעבורה המיועדת אליו (או תעבורת Broadcast - המושג יוסבר בקצרה בהמשך).

סניפרים אקטיביים יכולים לבצע התקפות כגון MAC Flooding ולגרום ל-Switch להתנהג כ-Hub ולהזרים את התעבורה בכל הפורטים שלו או ARP Spoofing על מנת לזהם את טבלאות ה-ARP (ARP יוסבר בהמשך) של מחשבים ולגרום להם לדבר עם התוקף במקום אחד עם השני.

סניפרים פסיביים לעומת זאת, אינם מבצעים שינויים ברשת עליה הם יושבים. סניפרים פסיביים לדוגמה הם tcpdump ו-wireshark.

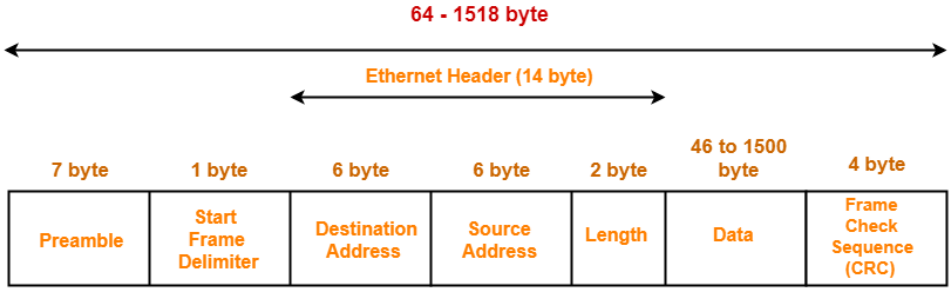
רובו של מאמר זה אינו מתמקד בזיהוי הסניפר עצמו, אלא בזיהוי קיומו של מצב פרוץ הנדרש לשני סוגי הסניפרים.

בעולם האמיתי, לעומת זאת, כיוון שמרבית הרשתות המודרניות מכילות Switch ועל מנת להאזין להן בצורה יעילה יש לבצע תחילה מניפולציה של רכיבים ברשת- מאמצי ההתגוננות מתמקדים בזיהוי התקפות אקטיביות (כגון ARP Spoofing ו-MAC Flooding) ולא בזיהוי מחשבים עם כרטיס רשת פרוץ.

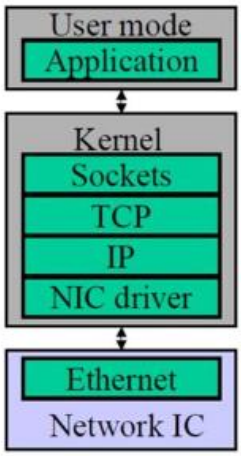
מה זה Promiscuous Mode?

על מנת להבין איך נוכל לזהות כרטיסי רשת ב-Promiscuous Mode עלינו להבין כיצד מתנהגים כרטיסי רשת פרוצים לעומת כרטיסי רשת שאינם פרוצים ואז, לבסס את הטכניקות שלנו על ההבדלים הללו.

תחילה נבין מה קורה כאשר תעבורה מגיעה למחשב. כל מידע המגיע למחשב עובר דרך כרטיס הרשת. כרטיס הרשת מוודא כי המידע בפורמט המצופה (במקרה של Ethernet - מכיל בתחילתו רצף בתים מסוים הנקרא Preamble שמסתיים בבית הנקרא SFD ומסמל את תחילת המידע[3]) ושהוא תקין[4].



IEEE 802.3 Ethernet Frame Format



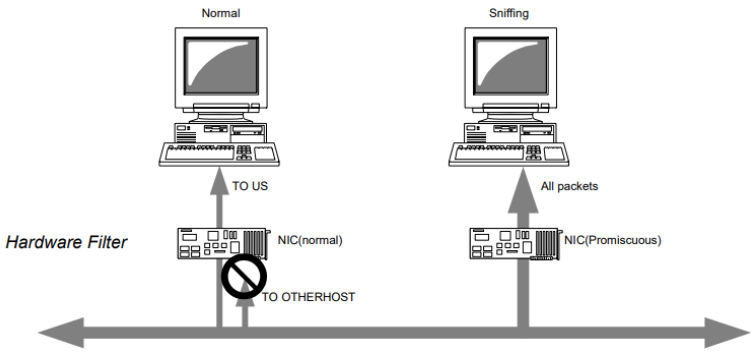
במידה והמידע תקין כרטיס הרשת מוודא שחבילת המידע מיועדת אלינו. חבילה כזו היא אחת מהאפשרויות הבאות:

1. חבילה שכתובת היעד שלה היא הכתובת MAC של כרטיס הרשת שלנו (Unicast).
2. חבילה שמיועדת למספר רכיבים (Multicast).
3. חבילה המיועדת לכל המחשבים באותה רשת (Broadcast) - ניתן לראות בכך מקרה פרטי של Multicast.

במידה וחבילה ענתה על אחת משלושת האפשרויות הללו, כרטיס הרשת ייזום פסיקת חומרה והפקטה תועבר "למעלה" אל ה-Kernel להמשך טיפול.

[3] https://en.wikipedia.org/wiki/Ethernet_frame#Preamble_and_start_frame_delimiter
[4] https://en.wikipedia.org/wiki/Frame_check_sequence

כאשר אנו מכניסים את ה-NIC שלנו למצב פרוץ - אנו בעצם מוותרים על אותו סינון חומרתי שמיועד לוודא שלא נקבל חבילות שלא מיועדות אלינו:



[http://www.securityfriday.com/promiscuous_detection_01.pdf]

כעת למעשה כל חבילה תקינה שתתקבל ע"י כרטיס הרשת, לא משנה למי היא מיועדת- תועבר הלאה. אם כך, על מנת לגלות אם למחשב יש כרטיס רשת פרוץ - (לכאורה) כל שעלינו לעשות הוא למעשה להרכיב חבילה שלא תעבור את הסינון של כרטיס הרשת אבל כן תצליח להוציא תשובה מהמחשב במידה ולא היה סינון כזה. אם למחשב יש כרטיס רשת רגיל - הוא יסנן אותה, ולא נראה תגובה לחבילה הזו. אולם אם כרטיס הרשת פרוץ - החבילה תעבור הלאה והמחשב יחזיר לנו תשובה.

ICMP PING Detection:

הרעיון בשיטה זו הוא פשוט ביותר. נשלח למחשב שאנו חושדים בו פקטת ICMP מסוג Echo request (נקרא גם ping).

הקאץ' הוא שכתובת ה-MAC של היעד מזויפת ואינה כתובת שמחשב היעד אמור לקבל, בעוד כתובת הIP של היעד היא כתובת היעד האמתית שלו.

אנחנו מתבססים על ההנחה שבמידה ואין סינון ברמת כרטיס הרשת, הפקטה תעבור ל-Kernel. ה-Kernel יניח שאם היא הגיע אליו, הרי שהבדיקה בשכבה 2 צלחה - ולכן הוא יבדוק רק את ה-Headers של השכבות העליונות. מכיוון שכתובת ה-IP מתאימה - הוא ייעתר לבקשה ונקבל בחזרה ICMP Echo Reply.

נכתוב סקריפט פשוט שממחיש את הרעיון:

```
import sys
from scapy.all import *

ICMP_ECHO_REQUEST = 8
MAX_TIMEOUT = 2

def detect_promiscuous(device_ip):
    request_packet = Ether(dst="ab:cd:ef:11:22:33")
    request_packet /= IP(dst=device_ip)
    request_packet /= ICMP(type=ICMP_ECHO_REQUEST)
```

```
    response = srp1(request_packet, timeout=MAX_TIMEOUT, iface="VMware Virtual
Ethernet Adapter for VMnet1", verbose=False)
    if response is None:
        print("Device {DEVICE_IP} is not in Promiscuous Mode.".format(DEVICE_IP
= device_ip))
    else:
        print("Device {DEVICE_IP} is in Promiscuous Mode.".format(DEVICE_IP =
device_ip))

def main():
    target_device_ip = sys.argv[1]
    detect_promiscuous(target_device_ip)

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

נסביר את הפונקציה detect_promiscuous שהיא לב התוכנית (על פי מספור השורות):

```
7   def detect_promiscuous(device_ip):
8       request_packet = Ether(dst="ab:cd:ef:11:22:33")
9       request_packet /= IP(dst=device_ip)
10      request_packet /= ICMP(type=ICMP_ECHO_REQUEST)
11      response = srp1(request_packet, timeout=MAX_TIMEOUT, iface="VMware Virtual Ethernet Adapter for VMnet1", verbose=False)
12      if response is None:
13          print("Device {DEVICE_IP} is not in promiscuous mode.".format(DEVICE_IP = device_ip))
14      else:
15          print("Device {DEVICE_IP} is in promiscuous mode.".format(DEVICE_IP = device_ip))
```

7. הפונקציה מקבלת כפרמטר את כתובת ה-IP של המכשיר שנרצה לבדוק.
8. כתובת ה-MAC של היעד תהיה כתובת פיקטיבית בכוונה. את כתובת המקור אין צורך לציין משום ש-scapy משלים זאת בעצמו.
9. נוסיף לפקטה את ה-Header של שכבת הרשת. כתובת ה-IP של היעד היא אותה כתובת שקיבלנו כפרמטר והיא כתובת הIP האמיתית של המכשיר החשוד. גם כאן, את כתובת המקור אין צורך לציין משום ש-scapy עושה זאת עבורנו.
10. נוסיף את ה-Header של ICMP. באופן דיפולטי כשיוצרים חבילת ICMP ב-Scapy היא כבר Echo Request אולם למען הבהירות נציין במפורש שאנו מעוניינים בEcho Request (שמספרה הוא 8).
11. נשלח את הפונקציה באמצעות srp1. פונקציה זו מקבלת חבילה לשליחה, ושולחת אותה ברמת שכבת הקו. לאחר מכן היא מאזינה לתשובה (אחת). במידה וחוזרת תשובה לפקטה ששלחנו, הפונקציה תחזיר אותה. במידה ולא היא תחזיר None.

לפונקציה זו שלחתי 4 פרמטרים:

1. את הפקטה אותה אני רוצה לשלוח.
2. את הזמן המקסימלי עבורו אנו מעוניין לחכות לתשובה לאחר השליחה, במקרה זה-2 שניות.
3. את הממשק דרכו אני רוצה לשלוח. בחרתי בממשק של הרשת הוירטואלית אליה מחוברות המכונות.
4. אם אני מעוניין בחיווי של סטטוס השליחה (כמה פקטות שלחתי, כמה פקטות התקבלו בחזרה וכו'...). במקרה זה בחרתי במצב שקט - ללא חיווי.

12. (עד 15) נבדוק מה קורה לאחר שהפונקציה הופעלה. אם קיבלנו תשובה, סימן שמחשב היעד ענה על הבקשה ששלחנו, למרות שהיא לא מיועדת אליו - ולכן נוכל להניח בסבירות גבוהה שכרטיס הרשת שלו פרוץ.

לעומת זאת, אם לא קיבלנו תשובה - ייתכן וכרטיס הרשת של מחשב היעד סינן את בקשת ה-Echo ששלחנו - משום שהוא אינו במצב פרוץ.

חשוב להדגיש שישנן גם סיבות אחרות לכך שלא קיבלנו תשובה, למרות שכרטיס הרשת שלו פרוץ. נדבר על חלק מסיבות אלו בהמשך.

תחילה נבחן את תגובת המכונה כשכרטיס הרשת אינו במצב פרוץ. לצורך ההדגמה הרמתי מכונת 12.04 Ubuntu (שהורדתי מ-osboxes.org) והתקנתי עליה Wireshark. המחשב המארח מריץ Windows 10. פרטי המכונה:

```
osboxes@osboxes:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:19:1d:5f
          inet addr:192.168.237.128  Bcast:192.168.237.255  Mask:255.255.255.0
```

ראשית, נבדוק את המצב הנוכחי. נחפש התייחסות למצב פרוץ בלוגים של הקרנל וניקח את הרשומה האחרונה (הכי עדכנית) שמתייחסת אליו:

```
osboxes@osboxes:~$ grep -r promiscuous /var/log/kern.log | tail -1
Feb 15 15:54:11 osboxes kernel: [ 2598.747852] device eth0 left promiscuous mode
osboxes@osboxes:~$
```

אנו רואים שכרטיס הרשת יצא ממצב פרוץ - כלומר כרגע הוא לא. אגב, אם אין פלט כלל כנראה כרטיס הרשת אינו פרוץ (כי הוא כנראה לא נכנס אליו מלכתחילה).

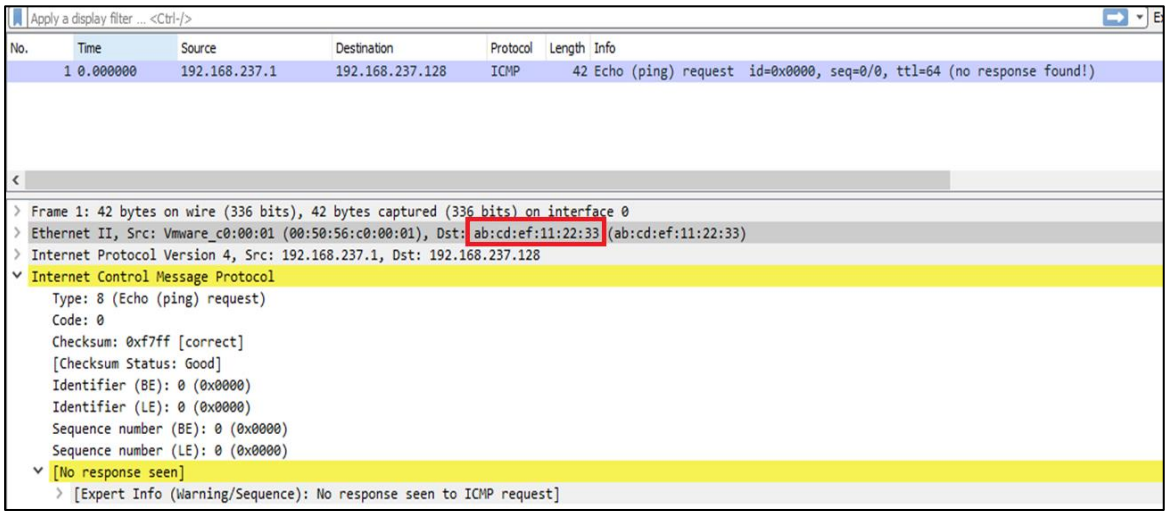
> הערה: ישנן פקודות נוספות שעושות את זה בצורה יותר אינטואיטיבית (כמו ifconfig או netstat -i, שתודגם בהמשך) אולם הן לא כל כך אמינות ולעיתים קרובות לא שיקפו את המצב כראוי (לשתיהן הייתה בעיה לזהות את המצב הפרוץ כשהוא לא השתנה דרך ה-Terminal אלא באמצעות תוכנה כמו Wireshark).

נריץ את הסקריפט שכתבנו:

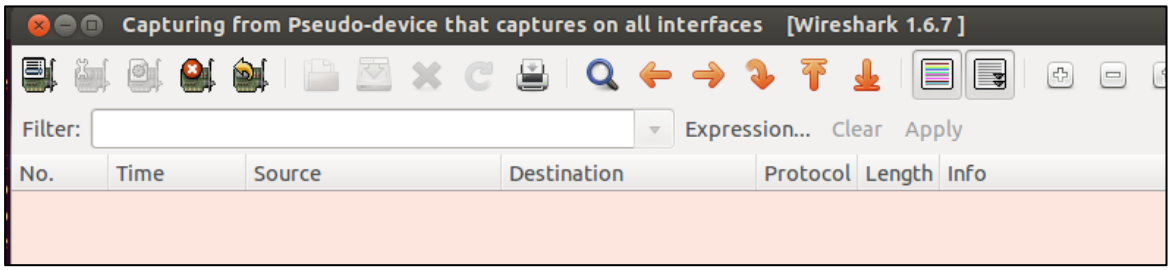
```
C:\Users\bnaya\Desktop\scripts>icmp_ping_detect.py 192.168.237.128
Device 192.168.237.128 is not in promiscuous mode.

C:\Users\bnaya\Desktop\scripts>
```

הסקריפט טוען שכרטיס הרשת אינו במצב פרוץ.

נתבונן בתקשורת של המחשב השולח ושל המכונה המקבלת. במחשב השולח:

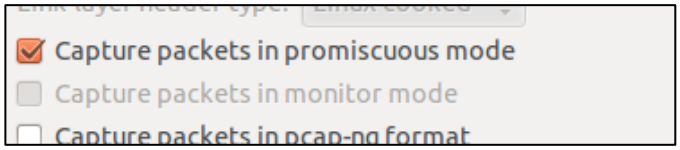


ניתן לראות שהמחשב המארח שלח את הבקשה לכתובת ה-IP של המכונה, ולכתובת MAC מזויפת ושאכן לא חזרה תשובה. במכונה עצמה אנו לא רואים כל תקשורת:



מדוע? משום שכרטיס הרשת סינן את כל התעבורה שאינה מיועדת אלינו. Wireshark מציג את כל התעבורה שעברה את כרטיס הרשת. משום שכרטיס הרשת אינו פרוץ, הרי שהפקטה ששלחנו סוננה. התנהגות זו תקינה ותואמת את ציפיותינו.

כעת נבחן את התנהגות המחשב כאשר כרטיס הרשת פרוץ. במכונת Ubuntu נריץ את wireshark במצב פרוץ תחת הרשאות root:



ואכן כרטיס הרשת נפרץ:

```
osboxes@osboxes:~$ grep -r promiscuous /var/log/kern.log | tail -1
Feb 15 16:02:13 osboxes kernel: [ 3080.134877] device eth0 entered promiscuous mode
osboxes@osboxes:~$
```

או לחלופין (אם אתם לא עובדים דרך Wireshark או Sniffer אחר שמכניס למצב פרוץ אוטומטית) נשנה את כרטיס הרשת ידנית:

```
osboxes@osboxes:~$ sudo ifconfig eth0 promisc
[sudo] password for osboxes:
osboxes@osboxes:~$ netstat -i
Kernel Interface table
Iface   MTU Met   RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-ERR TX-DRP TX-OVR Flg
eth0       1500 0     202      0      0 0        664      0      0      0 BMPRU
lo        65536 0     446      0      0 0        446      0      0      0 LRU
osboxes@osboxes:~$
```

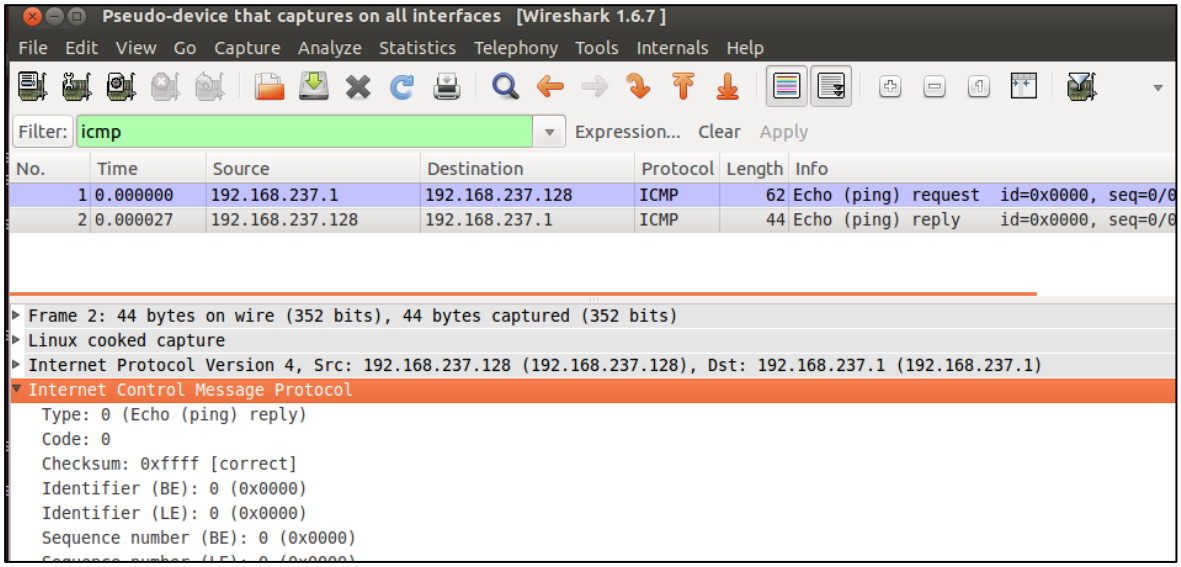
ניתן לראות שהדגל P (Promiscuous) התווסף ל-Interface.

> אגב, בהמשך להמלצה הקודמת לא להסתמך על netstat -i למרות שהיא דרך קצת יותר נוחה - היא מציגה גם את המצב Point to Point כ-"P"- מה שעשוי ליצור לא מעט בלבול. (קוד המקור של netstat)

נריץ את הסקריפט בשנית:

```
C:\Users\bnaya\Desktop\scripts>icmp_ping_detect.py 192.168.237.128
Device 192.168.237.128 is in promiscuous mode.
```

הצלחנו לזהות נכונה שכרטיס הרשת פרוץ. נתבונן בתקשורת:



הפעם התקבלה תשובת Echo Reply. המחשב הנבדק הגיב לבקשת ה-ICMP Echo שלנו למרות שמבחינת כתובת ה-MAC היא בכלל לא מיועדת אליו. ניסיתי את הסקריפט גם על מכונת Kali 2018.4 והתוצאות היו זהות.

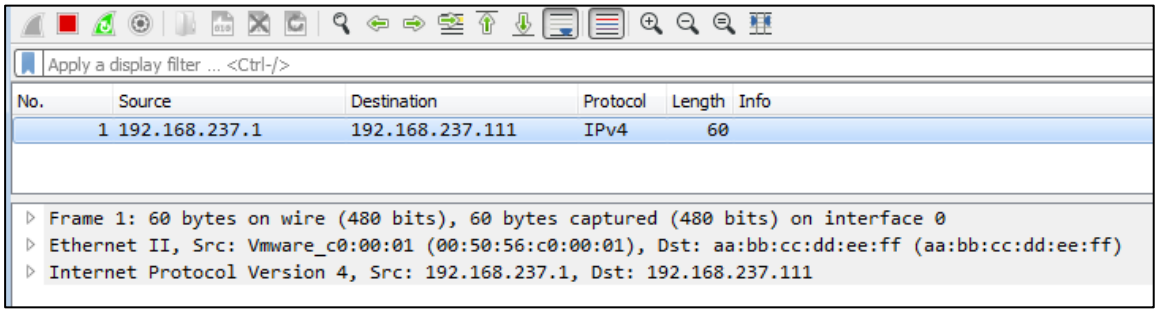
שיטה זו הייתה בשימוש די הרבה זמן ונחשבה לדרך הסטנדרטית בה מבצעים בדיקה זו, אולם כיום לא משתמשים בה ככלי בלעדי משום שבמערכות הפעלה מודרניות היא אינה מדויקת ולעיתים רבות תחזיר תוצאות שגויות (בעיקר False Negatives).

נראה דוגמה: הרמתי מכונת Windows 7 (SP1, 64 bit). כתובת ה-IP שלה היא: 192.168.237.**132**. נריץ עליה Wireshark במצב Promiscuous ונוודא שWireshark באמת מצליח לראות תקשורת שלא מיועדת אליו.

נשלח פקטת IP עם כתובות MAC ו-IP מפוברקות:

```
>>> my_packet = Ether(dst="aa:bb:cc:dd:ee:ff")
>>> my_packet /= IP(dst="192.168.237.111")
>>> sendp(my_packet, iface="VMware Virtual Ethernet Adapter for VMnet1")
.
Sent 1 packets.
>>>
```

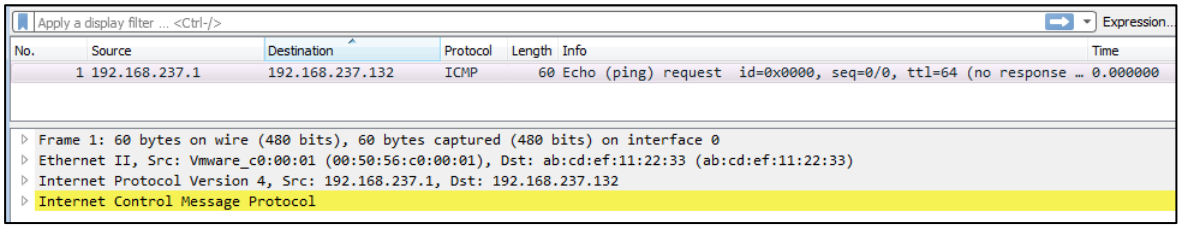
ואכן היא התקבלה במחשב המסניף:



יפה. כעת נפעיל את הסקריפט ונצפה לתפוס את המסניף:

```
C:\Users\bnaya\Desktop\scripts>icmp_ping_detect.py 192.168.237.132
Device 192.168.237.132 is not in promiscuous mode.
```

אופס... מה קורה פה? למה הסקריפט לא הצליח לזהות את המסניף? נבדוק את התקשורת כפי שהתקבלה במחשב המסניף:



ניתן לראות שה-ICMP Echo Request התקבלה באופן תקין אולם לא חזרה תשובה. מעניין.

בואו ננסה משהו בסיסי יותר. ננסה לשלוח ICMP Echo רגיל- בלי כתובות מזויפות ונראה מה קורה:

```
C:\Users\bnaya>ping 192.168.237.132

Pinging 192.168.237.132 with 32 bytes of data:
Request timed out.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 192.168.237.132:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss),
```

במחשב המסניף:

25	192.168.237.1	192.168.237.132	ICMP	74	Echo (ping) request id=0x0001, seq=17/4352, ttl=128 (no response found!)
26	Vmware_c0:00:01	Vmware_c1:80:86	ARP	60	Who has 192.168.237.132? Tell 192.168.237.1
27	Vmware_c1:80:86	Vmware_c0:00:01	ARP	42	192.168.237.132 is at 00:0c:29:c1:80:86
28	192.168.237.1	192.168.237.132	ICMP	74	Echo (ping) request id=0x0001, seq=18/4608, ttl=128 (no response found!)
29	192.168.237.1	192.168.237.132	ICMP	74	Echo (ping) request id=0x0001, seq=19/4864, ttl=128 (no response found!)
30	192.168.237.1	192.168.237.132	ICMP	74	Echo (ping) request id=0x0001, seq=20/5120, ttl=128 (no response found!)

גם כאן נכשלנו ולא קיבלנו תשובה. בואו נבין מה קורה כאן. המחשב המסניף מצליח לראות תקשורת שאינה מיועדת אליו- כלומר הוא אכן במצב פרוץ.

הוא מקבל בקשות ICMP Echo, גם בקשות המיועדות אליו וגם בקשות שאינן- ולא עונה לאף סוג.

נבדוק את ה-Firewall של Windows ונעבור על פני החוקים עד שנמצא משהו רלוונטי:

Distributed Transaction Coordinator (RPC-EPMAP)	Distributed Transaction Coo...	Private, Public	No	Allow	No	%System...	Any	Local subnet	TCP	RPC Endp.
Distributed Transaction Coordinator (TCP-In)	Distributed Transaction Coo...	Private, Public	No	Allow	No	%System...	Any	Local subnet	TCP	Any
Distributed Transaction Coordinator (TCP-In)	Distributed Transaction Coo...	Domain	No	Allow	No	%System...	Any	Any	TCP	Any
File and Printer Sharing (Echo Request - ICMPv4-In)	File and Printer Sharing	Private, Public	No	Allow	No	Any	Any	Local subnet	ICMPv4	Any
File and Printer Sharing (Echo Request - ICMPv4-In)	File and Printer Sharing	Domain	No	Allow	No	Any	Any	Any	ICMPv4	Any
File and Printer Sharing (Echo Request - ICMPv6-In)	File and Printer Sharing	Domain	No	Allow	No	Any	Any	Any	ICMPv6	Any

חוק זה מתיר תקשורת ICMPv4 בתוך הרשת הפנימית- אולם הוא מכובה ולכן המחשב לא עונה לנו.

נדליק אותו:

File and Printer Sharing (Echo Request - ICMPv4-In)	File and Printer Sharing	Private, Public	Yes	Allow	No	Any	Any	Local subnet	ICMPv4

נבדוק שיש תקשורת:

```
C:\Users\bnaya>ping 192.168.237.132

Pinging 192.168.237.132 with 32 bytes of data:
Reply from 192.168.237.132: bytes=32 time<1ms TTL=128
Reply from 192.168.237.132: bytes=32 time<1ms TTL=128
```

כעת נפעיל את הסקריפט:

```
C:\Users\bnaya\Desktop\scripts>icmp_ping_detect.py 192.168.237.132
Device 192.168.237.132 is in promiscuous mode.
```

ואכן הוא הצליח לזהות את המכונה.

נסכם את הדוגמה האחרונה:

במכונות ה-Linux שבדקנו ראינו שמתודה זו פעלה כשורה. במכונת ה-Windows - חומת האש אינה מאפשרת באופן דיפולטי תקשורת בפרוטוקול ICMP ולכן, מחשב היעד לא החזיר תשובה. הסקריפט שלנו חשב שהוא לא החזיר תשובה משום שה-NIC סינן את התעבורה כי הוא אינו פרוץ ולכן החזיר תוצאה שגויה.

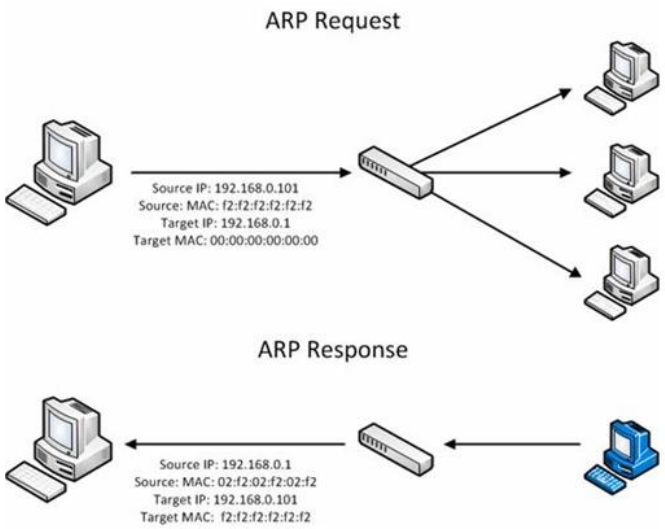
נחשוב על זה קצת יותר לעומק - לסקריפט שלנו אין שום דרך לדעת מדוע מחשב היעד לא החזיר תשובה- והוא יניח שזה משום שהוא אינו פרוץ. למעשה אנו רואים כאן שיטה מעניינת להתחמקות מפני אנטי סניפרים - באמצעות Firewall שחוסם את התקשורת החוצה.

ARP Test

שיטה זו מתבססת על עיקרון זהה לשיטה הקודמת אולם במקום לשלוח פקטת ICMP עם כתובת MAC מזויפת, נעשה אותו דבר אבל בעזרת פרוטוקול ARP[5].

פרוטוקול ARP, מסייע לנו לברר את כתובת ה-MAC של מחשבים שאיננו יודעים את כתובת ה-MAC שלהם אבל כן את כתובת ה-IP שלהם. באופן תקין נשלח חבילת בקשה (ARP Request) לכתובת Broadcast. בבקשה מציינים את כתובת ה-IP של היעד.

כלל המחשבים ברשת רואים את הבקשה אבל רק היעד שמזהה את כתובת ה-IP שלו - עונה בחזרה עם כתובת ה-MAC שלו וכך השולח יודע מהי כתובת ה-MAC שהוא אמור לפנות אליה בהמשך התקשורת. הוא שומר את הצימוד של ה-MAC וה-IP בטבלה מיוחדת (ARP Cache Table) וכשהוא ייתקל בכתובת IP בהמשך הוא ידע לאיזו כתובת פיזית לפנות:



[http://computerprojectsduff.wikia.com/wiki/File:Image0021268491809942.jpg]

[5] https://he.wikipedia.org/wiki/Address_Resolution_Protocol

בניגוד לפרוטוקול ICMP שכפי שראינו - מגיע מנוטרל בגרסאות החדשות של windows, פרוטוקול ARP הוא פרוטוקול חיוני יותר. מרבית רשתות התקשורת המודרניות המתבססות על IPv4 עושות שימוש ב-ARP[6].

חסימת פרוטוקול זה איננה מומלצת משום שהיא עשויה לפגוע ביכולות התקשורת של המחשב - מה שרוב המשתמשים לא ירצו לעשות.

מחשב שאינו עונה לבקשות ARP לא יימצא ע"י מחשבים שלא יודעים את כתובתו הפיזית, ומחשב שלא שולח בקשות ARP לא יוכל לתקשר עם מחשבים שלא שמורים אצלו ב-Cache.

> מנהלי רשת יכולים להגדיר טבלת ARP סטטית בין מחשבים באותה הרשת וכך מחשבים אלו לא יעשו שימוש בפרוטוקול ARP, אולם לשיטה זו חסרונות רבים. למשל, עבור כל מחשב חדש שיצטרף לרשת, מנהלי הרשת יצטרכו להגדיר רשומה נוספת בכל המחשבים האחרים. במרבית המקרים עדיף לאפשר את פרוטוקול ARP ולתת למחשבים למצוא את הכתובת MAC של חבריהם באופן דינאמי.

גם כאן, אנו נתבסס על ההנחה שאם כרטיס הרשת פרוץ - אז מחשב היעד יענה על כל בקשת ARP שכתובת ה-IP בה מצביעה עליו, לא משנה אם כתובת ה-MAC מיועדת אליו או לא. לעומת זאת, אם כרטיס הרשת אינו פרוץ ונשלח כתובת MAC שאינה כתובתו או שאינה Broadcast - אז הוא יסנן אותה ולא נראה כל תשובה.

נבנה סקריפט דומה:

```
9   def detect_promiscuous(device_ip):
10      arp_packet = Ether(dst="aa:bb:cc:dd:ee:ff")
11      arp_packet /= ARP(pdst= device_ip)
12      response = srp1(arp_packet, timeout=MAX_TIMEOUT, iface="VMware Virtual Ethernet Adapter for VMnet1", versbose=False)
13      if response is None:
14          print("Device {DEVICE_IP} is not promiscuous mode.".format(DEVICE_IP = device_ip))
15      else:
16          print("Device {DEVICE_IP} is in promiscuous mode.".format(DEVICE_IP = device_ip))
```

נסביר את השינויים (על פי מספר השורות):

10. השכבה הראשונה היא Ethernet, שבה אנו מציינים כתובת יעד פיזית (MAC) מזויפת.
11. על גבי שכבת ה-Ethernet נשים את שכבת ה-ARP. כתובת היעד הלוגית (IP) תהיה הכתובת האמתית של המחשב החשוד.
12. נשלח את הבקשה (במצב שקט, ללא חיווי על סטטוס השליחה) ונמתין לתשובה.
13. (עד 16): בדומה לסקריפט הקודם, אם קיבלנו תשובה - הרי שהמחשב החשוד התעלם מכך שהחבילה לא מיועדת אליו ולכן כנראה כרטיס הרשת שלו במצב פרוץ.

טוב, אז שנריץ?

[6] ב-IPv6 - ARP לא רלוונטי יותר (אבל זה נושא אחר).

הרמתי שוב את מכונת ה-Ubuntu 12.04 שלנו עם הכתובת: 192.168.237.128, כשהיא לא במצב פרוץ.

הרצתי את הסקריפט שזיהה שהיא אכן לא במצב פרוץ:

```
C:\Users\bnaya\Desktop\scripts>arp_detection.py 192.168.237.128
Device 192.168.237.128 is not promiscuous mode.
```

במחשב השולח ניתן לראות שהבקשה נשלחה כשורה:

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	Vmware_c0:00:01	aa:bb:cc:dd:ee:ff	ARP	42	Who has 192.168.237.128? Tell 192.168.237.1

ובמחשב המקבל לא רואים את בקשת ה-ARP משום שהיא סוננה ע"י ה-NIC:

Filter: Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Length	Info

נשנה למצב פרוץ:

```
osboxes@osboxes:~$ sudo ifconfig eth0 promisc
[sudo] password for osboxes:
osboxes@osboxes:~$ netstat -i
Kernel Interface table
Iface   MTU Met   RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-ERR TX-DRP TX-OVR Flg
eth0       1500 0    1675      0      0 0       1264      0      0      0 BMPRU
lo        65536 0     894      0      0 0        894      0      0      0 LRU
osboxes@osboxes:~$
```

או ב-Wireshark:

Capture packets in promiscuous mode

ונריץ שנית:

```
C:\Users\bnaya\Desktop\scripts>arp_detection.py  192.168.237.128
Device 192.168.237.128 is not promiscuous mode.
```

אופס... מה קורה פה? הסקריפט לא הצליח לזהות שהמכשיר במצב פרוץ ונתן תוצאה שגויה. המחשב השולח שלח את ההודעה כראוי:

arp

No.	Time	Source	Destination	Protocol	Length	Info
3	25.016736	Vmware_c0:00:01	aa:bb:cc:dd:ee:ff	ARP	42	Who has 192.168.237.128? Tell 192.168.237.1

והיעד קיבל את הבקשה:

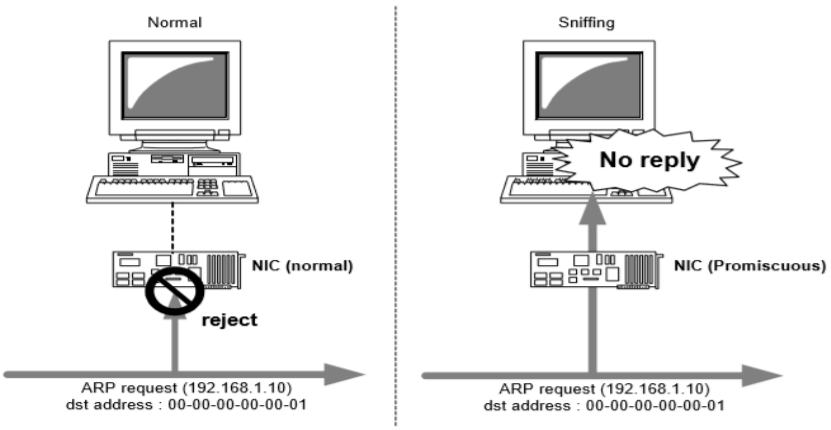
Filter: arp Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Length	Info
4	29.954186	Vmware_c0:00:01	aa:bb:cc:dd:ee:ff	ARP	60	Who has 192.168.237.128? Tell 192.168

אולם בחר שלא להגיב עליה. מדוע? לכאורה, במידה וכרטיס הרשת פרוץ- כל כתובת באשר היא תגרום ליעד להחזיר תשובה - כפי שראינו כשהשתמשנו בטכניקה של ICMP. נכון?

אז לא בדיוק. בפרוטוקול ARP, מבוצעות בדיקות נוספות על המסגרת ורק מסגרת שתעמוד בבדיקות אלו, תזכה למענה ולטיפול. בדיקות אלו מכונות בספרות המקצועית "הסינון התוכנתי" (Software Filter), וזאת על מנת להבדיל מהסינון החומרתי שמבוצע ע"י ה-NIC.

התגובה שראינו מתוארת בתרשים הבא:



[http://www.securityfriday.com/promiscuous_detection_01.pdf]

בשני המצבים לא נקבל תשובה אולם מסיבות שונות. במצב רגיל כרטיס הרשת לא יעביר את הפקטה הלאה למערכת ההפעלה (ואז גם לא נראה את הפקטה ב-Wireshark) ואילו במצב פרוץ כרטיס הרשת יעביר את הפקטה אולם המחשב עצמו יבחר שלא להגיב כי הפקטה אינה תקינה. מה נעשה?

למזלנו, הבדיקות שמבצעת מ"ה הן פחות קפדניות מהבדיקות שעורך ה-NIC. ניתן לתכנן שליחת בקשת ARP שתדחה ע"י ה-NIC במצב רגיל אבל תתקבל ותיענה ע"י מ"ה אם היא כבר הגיעה אליו. עוד נקודה שמעניין לציין היא שהבדיקות הללו שונות ממ"ה למ"ה, וייתכן ובקשות עם כתובות מסוימות ייענו על ידי מ"ה מסוימת אבל לא ע"י אחרות.

בטבלה להלן ניתן לראות בקשות עם כתובות MAC שונות למספר מערכות הפעלה (ישנות אומנם, אבל העיקרון רלוונטי גם למ"ה מודרניות) ואת תגובתן במצב רגיל ובמצב פרוץ.

Table 1. Promiscuous mode detection results using trap ARP request packets

Hardware Addresses \ Operating Systems		Windows XP		Windows Me/9x		Windows 2k/NT		Linux 2.4.x		FreeBSD 5.0	
		Norm.	Prom.	Norm.	Prom.	Norm.	Prom.	Norm.	Prom.	Norm.	Prom.
FF:FF:FF:FF:FF:FF	Br	O	O	O	O	O	O	O	O	O	O
FF:FF:FF:FF:FF:FE	B47	--	X	--	X	--	X	--	X	--	X
FF:FF:00:00:00:00	B16	--	X	--	X	X	X	--	X	--	X
FF:00:00:00:00:00	B8	--	--	--	X	--	--	--	X	--	X
01:00:00:00:00:00	Gr	--	--	--	--	--	--	--	X	--	X
01:00:5E:00:00:00	M0	--	--	--	--	--	--	--	X	--	X
01:00:5E:00:00:01	M1	O	O	O	O	O	O	O	O	O	O
01:00:5E:00:00:02	M2	--	--	--	--	--	--	--	X	--	X
01:00:5E:00:00:03	M3	--	--	--	--	--	--	--	X	--	X

O: Legal response, **X**: Illegal response, **--**: No response

[מקור]

כלל מערכות ההפעלה, גם אלו שעורכות את הבדיקות הכי פחות קפדניות - בודקות שכתובת היעד היא או הכתובת האמתית של המחשב, או שהיא מכילה Group bit דולק.

Group bit הוא הביט שמציין אם כתובת MAC היא Multicast או לא. ביט זה הוא למעשה הביט הראשון (הימני ביותר, LSB) בבית השמאלי ביותר. כך תראה כתובת שרק ה-Group bit בה דלוק:

```
00000001:00000000:00000000:00000000:00000000:00000000
```

או בבסיס הקסה-דצימלי:

```
01:00:00:00:00:00
```

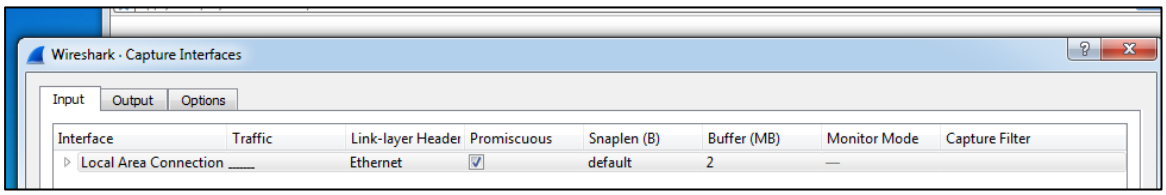
נרצה שהבדיקה שלנו תהיה כמה שיותר מהימנה, ולכן נזייף כתובת שתתאים לכמה שיותר מערכות הפעלה.

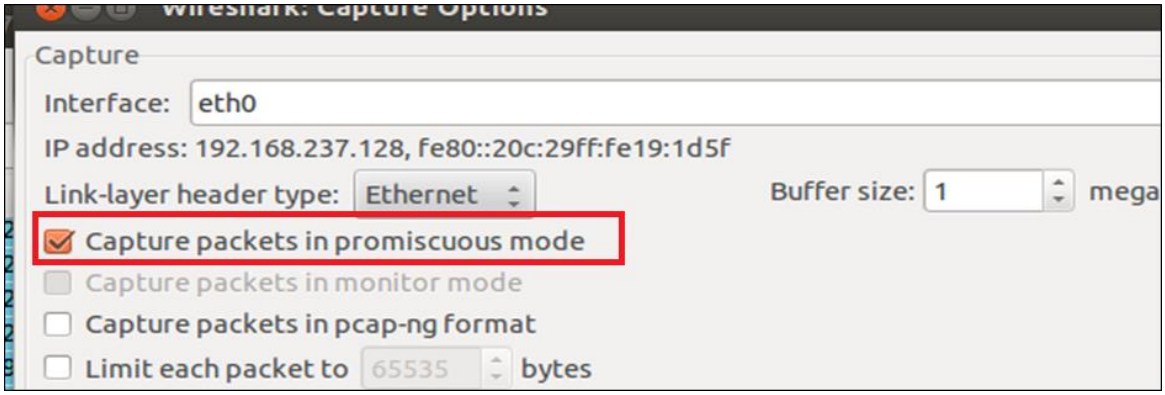
מהתבוננות בטבלה ניתן לראות שהכתובת ff:ff:ff:ff:ff:fe (ששונה רק בביט אחד מכתובת broadcast), מניבה תוצאות טובות עבור כלל מערכות ההפעלה שנבדקו.

נשנה את הסקריפט בהתאם:

```
def detect_promiscuous(device_ip):
    arp_packet = Ether(dst="ff:ff:ff:ff:ff:fe")
    arp_packet /= ARP(pdst= device_ip)
    response = srp1(arp_packet, timeout=MAX_TIMEOUT, iface="VMware Virtual Ethernet Adapter for VMnet1", verbose=False)
    if response is None:
        print("Device {DEVICE_IP} is not promiscuous mode.".format(DEVICE_IP = device_ip))
    else:
        print("Device {DEVICE_IP} is in promiscuous mode.".format(DEVICE_IP = device_ip))
```

נרים Wireshark במצב פרוץ במכונות linux ו-windows שלנו ונריץ את הסקריפט:





על מכונת ה-Ubuntu:

```
C:\Users\bnaya\Desktop\scripts>arp_detection.py 192.168.237.128
Device 192.168.237.128 is in promiscuous mode.
```

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	Vmware_c0:00:01	ff:ff:ff:ff:ff:fe	ARP	60	Who has 192.168.237.128? Tell 192.168.237.1
2	0.000018	Vmware_19:1d:5f	Vmware_c0:00:01	ARP	42	192.168.237.128 is at 00:0c:29:19:1d:5f

ועל מכונת ה-Windows:

```
C:\Users\bnaya\Desktop\scripts>arp_detection.py 192.168.237.132
Device 192.168.237.132 is in promiscuous mode.
```

Source	Destination	Protocol	Length	Info
Vmware_c0:00:01	ff:ff:ff:ff:ff:fe	ARP	60	Who has 192.168.237.132? Tell 192.168.237.1
Vmware_c1:80:86	Vmware_c0:00:01	ARP	42	192.168.237.132 is at 00:0c:29:c1:80:86

סיכום חלק זה

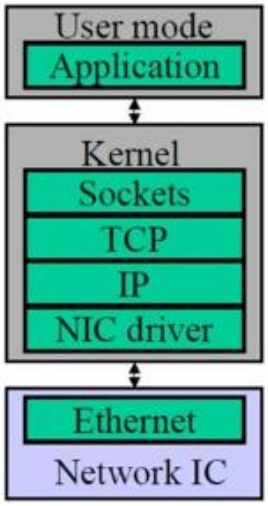
ראינו בדוגמה האחרונה את שיטת ה-ARP Request. לשיטה זו יתרון משמעותי על פני שיטת ה-ICMP. היא אמינה יותר משום שהיא מתבססת על ARP ולא על ICMP שלעיתים רבות (ב-Windows - דיפולטית) מנוטרל.

גם שיטה זו עשויה להניב תוצאות שגויות כתוצאה מהתנהגות מ"ה לאחר קבלת בקשת ה-ARP. חסרון מסוים שלה הוא שבניגוד לשיטה המתבססת על ICMP, כאן עלינו להרכיב בקשה עם כתובת יעד מיוחדת, שתצלח את הסינון התוכנתי שמבצעת מערכת ההפעלה.

במערכות הפעלה שהן Open Source ניתן לקרוא את קוד המקור ולהבין במדויק אלו בדיקות המערכת עורכת לבקשות וכך לנסח בקשה עם כתובת מתאימה. במערכות הפעלה אחרות, כמו Windows ניתן לבחון את התנהגות המערכת על סמך ניסויים ותצפיות ולקבוע אילו בקשות זוכות למענה (כלומר עוברות את הסינון התוכנתי).

בדוגמה שלנו שלחנו בקשה עם הכתובת FF:FF:FF:FF:FF:FE שלפי ניסויים קודמים שנערכו (ראו "העמקה נוספת") התקבלה ע"י מספר מערכות ההפעלה הגדול ביותר ואכן הצלחנו לזהות נכונה את מצב כרטיס הרשת של המכונות שלנו.

Latency test



כפי שהסברנו בתחילת המאמר, כאשר כרטיס רשת במצב פרוץ הוא מעביר את כלל התעבורה "למעלה", לטיפול ע"י רכיבי התוכנה של מ"ה.

אותם רכיבי תוכנה בליבת מ"ה יצטרכו להתמודד עם תעבורה לא מסוננת והם יאלצו לסנן אותה בעצמם. אנחנו נתבסס על ההנחה שמחשב שכרטיס הרשת שלו במצב פרוץ, יגיב לאט יותר לבקשות משום שה-Kernel שלו עמוס בסינון תעבורה שלא רלוונטית אליו.

בנוסף, אם על היעד רץ Sniffer שמקליט את התעבורה, ה-Kernel יצטרך להעביר כל פקטה שהתקבלה לסניפר, שרץ ב-User Mode. מעבר כזה, מ-Kernel Mode ל-User Mode הוא יחסית בזבזני ויצור עומס נוסף על היעד.

בשיטה זו, אנחנו נמדוד ערך שנקרא RTT (round-trip time). RTT הוא בעצם הזמן שלקח לפקטה שלנו לעשות את הדרך למחשב היעד ועוד הזמן שלקח לתשובה ממחשב היעד להגיע חזרה אלינו. ערך זה ישקף לנו את זמן התגובה של מחשב היעד וכך נראה כמה זמן לוקח לו להגיב לבקשות שלנו.

לאחר לקיחת RTT נציף את הרשת בתעבורה מזובלת ונבצע מדידה נוספת. במידה וכרטיס הרשת אינו פרוץ, נצפה לקבל תוצאות קרובות יחסית למה שקיבלנו לפני ההצפה. במידה וכרטיס הרשת פרוץ - נצפה שה-RTT יהיה גבוה יותר באופן משמעותי.

לשיטה זו חסרונות רבים:

- בכך שהיא מעמיסה על הרשת היא לא רק פוגעת בביצועים של כלל הרכיבים- אלא היא עשויה להתריע לתוקף באופן מאוד בולט שחושדים בו.
- היא חשופה להטיות כתוצאה ממגוון סיבות. למשל, ייתכן וכרטיס הרשת פרוץ, אולם הרשת עמוסה מאוד בכל מקרה- כך שבמדידה הראשונה, לפני ההצפה-מתקבל RTT גבוה ובמדידה השנייה, במהלך ההצפה, מתקבל RTT קרוב אליו. במצב כזה אנחנו נטעה לחשוב שכרטיס הרשת אינו פרוץ, כי לא

ראינו הבדל משמעותי ב-RTT. לחלופין, ייתכן וכרטיס הרשת אינו פרוץ, אולם היעד השתהה בתשובה למדידה השנייה כתוצאה מסיבה אחרת ואז נטעה לחשוב שכרטיס הרשת פרוץ, משום שראינו הבדל משמעותי בין המדידות, אולם לאמיתו של דבר כרטיס הרשת אינו פרוץ וההבדל נבע מסיבה אחרת.

כפי שמיד נראה, נדרשת מיומנות בקריאת והבנת הפלט המתקבל. בדיקה זו אינה אופטימלית בסביבה וירטואלית משום שכרטיסי הרשת של המכונות מסומלצים באמצעות תוכנה, כך שלמעשה אנחנו לא בוחנים כרטיסי רשת אמיתיים אלא רק את הסימלוץ שלהם. לכן את בדיקה זו ערכתי את עם מחשבים פיזיים - מחשב המריץ Windows 10 שישמש בתפקיד המזבל והמודד, ומחשבWindows 7 שישמש כנמדד.

כתבתי את פונקציית המדידה בשתי גרסאות. גרסה multithreaded וגרסת single threaded. בגרסה הראשונה יש thread שמזבל וthread שמבצע את המדידות ובגרסה השנייה ישנה לולאה ובכל איטרציה נשלח כמה מאות פקטות מזובלות ואז נבצע את המדידה. לכל אחת מהגרסאות יש יתרונות וחסרונות משלה. הסבר של ההבדלים נמצא בדוקומנטציה של הסקריפט.

להלן חלק מהקוד של גרסת ה-multi-threaded:

```
without_noise = ping_measurement(target_ip)
noise_thread = threading.Thread(target=bomb_network,args=(target_ip, NOISE_AMOUNT))
noise_thread.start()
# Let the noise thread work a bit and create a load on the target before we start to measure the RTTs
time.sleep(SLEEP_BEFORE_MEASUREMENT)
with_noise = ping_measurement(target_ip)
```

1. נתחיל בלבצע את המדידות כשהרשת שקטה - נשלח 30 בקשות ICMP Echo ונחזיר רשימה עם ה-RTT של כל אחת מהבקשות.
2. (עד 4) נגדיר thread שיבצע את הזיבול עבורנו (הפונקציה המזבלת היא bomb_network) ונריץ אותו.
5. ניתן ל-thread המזבל לרוץ קצת כדי שהיעד יוצף בזבל לפני שיתחיל לקבל את פקטות המדידה שלנו.
6. נמדוד פעם נוספת, תוך כדי שהthread המזבל רץ.

איך תיראה bomb_network, הפונקציה שמזבלת את הרשת? אנחנו יכולים לעשות משהו פשוט כמו לשלוח מחרוזת אקראית ברשת מס' פעמים גדול (נגיד 2,000 פעמים):

```
def bomb_network(device_ip, amount):
    raw_data = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalala"
    for i in xrange(amount):
        sendp(raw_data, iface=VMWARE_IFACE, verbose=False)
```

נחשוב רגע מה קורה כאשר החבילה להלן מגיעה ליעד שכרטיס הרשת שלו פרוץ:

```
1120 3.081029   6c:61:6c:61:6c:61   6c:61:6c:61:6c:61   0x6c61   72 Ethernet II
▸ Frame 1120: 72 bytes on wire (576 bits), 72 bytes captured (576 bits)
▾ Ethernet II, Src: 6c:61:6c:61:6c:61 (6c:61:6c:61:6c:61), Dst: 6c:61:6c:61:6c:61 (6c:61:6c:6
  ▸ Destination: 6c:61:6c:61:6c:61 (6c:61:6c:61:6c:61)
  ▸ Source: 6c:61:6c:61:6c:61 (6c:61:6c:61:6c:61)
    Type: Unknown (0x6c61)
▾ Data (58 bytes)
    Data: 6c616c616c616c616c616c616c616c616c616c616c616c61...
    [Length: 58]
0000  6c 61 6c 61 6c 61 6c 61  6c 61 6c 61 6c 61 6c 61   lalalala lalalala
0010  6c 61 6c 61 6c 61 6c 61  6c 61 6c 61 6c 61 6c 61   lalalala lalalala
0020  6c 61 6c 61 6c 61 6c 61  6c 61 6c 61 6c 61 6c 61   lalalala lalalala
0030  6c 61 6c 61 6c 61 6c 61  6c 61 6c 61 6c 61 6c 61   lalalala lalalala
0040  6c 61 6c 61 6c 61 6c 61                            lalalala
```

היא חולפת על פני כרטיס הרשת, למרות שהיא פגומה ונכנסת לתוך ליבת מ"ה. שם, מיד מזהים שהיא לא תואמת אף פרוטוקול נורמלי ומסלקים אותה. אז אומנם גרמנו למחשב היעד לעבוד קצת יותר קשה כי הוא צריך לסנן זבל - אבל הוא מסנן אותו במהירות וחוזר לשגרה.

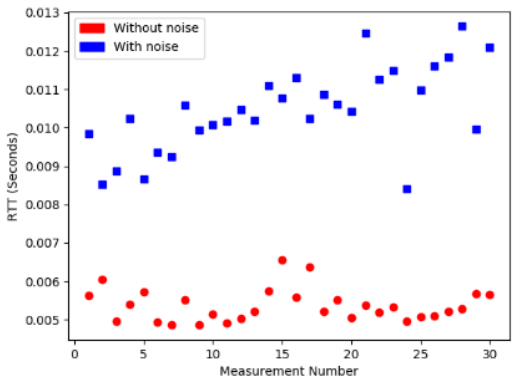
אנחנו רוצים שמחשב היעד יעבור יותר קשה. כדי לעשות זאת, נרכיב חבילת מידע שתצלח כמה שיותר "שלבים" בתוך ה-Kernel. כך, במקום לשלוח סתם מאות מחרוזות חסרות פשר שידחו מיד, נרכיב חבילות שיש להן סיכוי טוב להעמיס באופן יותר משמעותי על מחשב היעד.

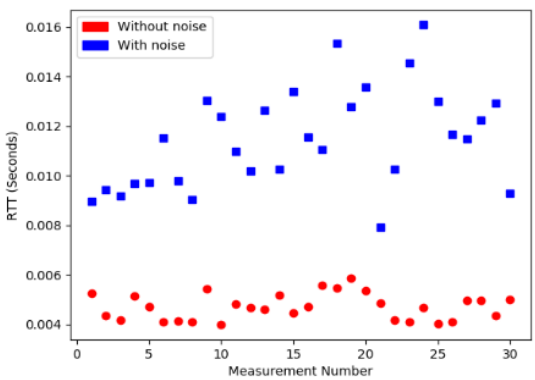
הסגמנט שנרכיב יהיה מורכב משכבת ethernet עם כתובת שונה מהכתובת של מחשב היעד - כדי שתדחה ע"י מחשבים עם כרטיס רשת רגיל. נוסיף IP Header - עם כתובת היעד האמיתית של המחשב הנבדק, כדי לצלוח את הבדיקה של שכבה 3. לאחר מכן נוסיף Header של TCP - בשלב זה הסגמנט כנראה יידחה ע"י היעד, כי הוא פונה לפורט סגור:

```
def bomb_network(device_ip, amount):
    noise_packet = Ether(dst="aa:aa:aa:aa:aa:aa")/IP(dst=device_ip)/TCP()
    for i in xrange(amount):
        sendp(noise_packet, iface=VMWARE_IFACE, verbose=False)
```

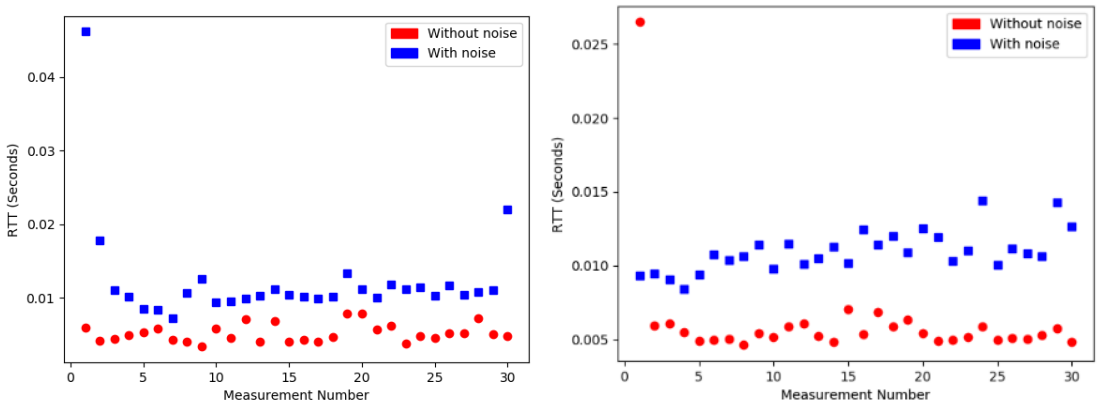
דבר נוסף שניתן לעשות הוא לפרגמנט את פקטות ה-IP כך שהיעד יצטרך להשקיע זמן וכוח עיבוד בלבצע Reassembly.

יאללה. מוכנים להריץ? להלן תוצאות של שתי הרצות במצב פרוץ, אחת עבור כל גרסה של מדידה.





ושתי הרצות כשהיעד לא פרוץ:



טוב... בואו נעשה קצת סדר בבלגן ונבין מה אנחנו רואים כאן.

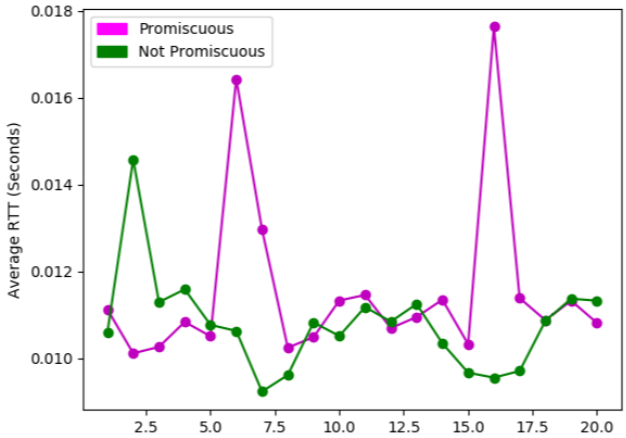
הנקודות האדומות הם ערכי ה-RTT של הפינגים שנשלחו כשהרשת הייתה שקטה והרבועים הכחולים הם ה-RTT של הפינגים שנשלחו כשהרשת הייתה רועשת.

ניתן לראות שבכל ארבעת התרשימים, מרבית הריבועים הכחולים היו מעל הנקודות האדומות, כלומר, המדידות שנלקחו כשהרעשנו את הרשת - היו גבוהות יותר מאלו שנשלחו כשהרשת הייתה שקטה - וזה הגיוני.

נשים לב שבשני התרשימים האחרונים יש חריגים סטטיסטיים (outliers) גם כחולים וגם אדומים- שצריך לזהות ולהבין האם להחשיב אותם או להיפטר מהם. אתם מצליחים לראות ההבדל משמעותי בין התרשימים שתי זוגות התרשימים?

לצערי עניתי על השאלה הזו בשלילה. כדי להיות יותר בטוח בתשובה שלי השוויתי בין ממוצעי ה-RTT של מצב פרוץ ורגיל.

נחשב את ה-RTT הממוצע של 30 מדידות - אותו ממוצע ייחשב כנתון אחד בגרף הסופי שלנו. ניקח 20 ממוצעי RTT של מצב פרוץ ו-20 של מצב רגיל, במידה ויש הבדל - נראה את זה מתבטא בגרף.



ההפרש בין העקומות הוא עדיין לא מספיק משמעותי. בהתאם להסבר התיאורטי, הייתי מצפה שבכל נקודה ונקודה על פני התרשים, העקומה הסגולה תהיה מעל הירוקה.

לאורך הדרך הזכרתי מס' דברים שאנחנו יכולים לנסות להגיע לתוצאות טובות יותר: אנחנו יכולים לנסות להעמיס על היעד בדרכים נוספות- כגון שליחת פקטות IP מפוצלות (fragments) או לחלופין, לנסות לזקק ולמצות את המיטב מהנתונים שאספנו- למשל, לנסות להתמודד עם Outliers או לחשב מדדים נוספים על הנתונים בניסיון למצוא הבדל מובהק בין המצבים. אנחנו יכולים גם לנסות לכוונן אחרת את הפרמטרים של הסקריפט - למשל, להגדיל את מספר פקטות הרעש, או את המרווחים בין המדידות.

סתם נקודה מעניינת - מכיוון שאנחנו מנסים למדוד ולהסיק מסקנות ממספרים קטנים מאוד- ישנה חשיבות מכרעת לקבלת תוצאות מדויקות במדידות- ולכן, ישנה חשיבות גם למבנה של הסקריפט ולביצועים שלו. למשל, התבוננו בפיסת הקוד הבא שאמורה לרוץ בתוך ה-Thread שאחראי להציף את הרשת:

```
for i in xrange(amount):
    noise_packet = Ether(dst="aa:aa:aa:aa:aa:aa")/IP(dst=device_ip)/TCP()
    sendp(noise_packet, iface=VMWARE_IFACE, verbose=False)
```

האם אתם מבחינים בבעייתיות שלה?
בכל איטרציה, הפקטה תורכב מחדש. הדבר בעייתי משום שהרכבת הפקטה לוקחת זמן יחסית ארוך שבמהלכו יעד עם כרטיס רשת פרוץ אינו "מופצץ" והוא פנוי להשיב על בקשות ICMP Echo של Thread אחר והוא יעשה זאת באותה מהירות שבה ישיב עליו מחשב שכרטיס הרשת שלו אינו פרוץ. כאן המקום לציין שהשימוש בפייתון באופן כללי וב-Scapy ספציפית, אינו ידוע כאופטימלי מבחינת זמן ריצה - אולם אלו נבחרו בשל פשטותן ונוחותן.

סיכום חלק זה

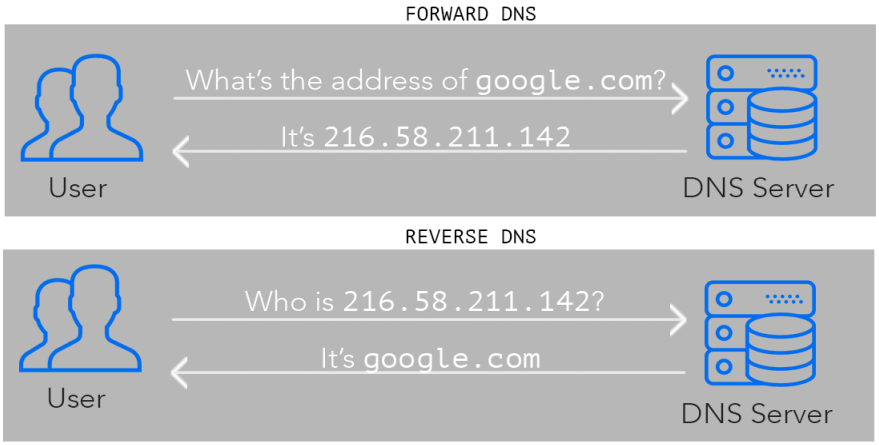
בחלק זה סקרנו את שיטת ה-Latency Test שבה ניסינו לקבוע את מצב כרטיס הרשת של היעד לפי זמן התגובה שלו.

לצערי - התוצאות לא הראו הבדל משמעותי שניתן להסתמך עליו בקביעת מצבו של כרטיס רשת. כפי שבוודאי שמתם לב, שיטה זו יוצרת תעבורה מרובה (ואורכת זמן רב) ולכן למרות שהיא מעניינת מבחינת המחקר התיאורטי, היא לא באמת ישימה לרשת אמיתית.

DNS Decoy

בניגוד לשיטות הקודמות, שהתמקדו בהתנהגות ובזמן התגובה של מ"ה - שיטה זו מתמקדת בהתנהגות תוכנת הרחרחן.

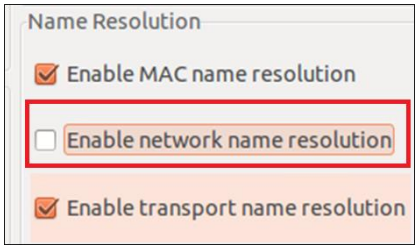
למעשה, סניפרים רבים שמוגדרים כפסיביים - אינם פסיביים לחלוטין ויוצרים תעבורה שמאפשרת לנו לזהות אותם. אותם סניפרים מנסים לתרגם כתובות IP לשמות Domain כדי להציג לנו את התקשורת בצורה ברורה יותר. הם עושים זאת ע"י שימוש בפרוטוקול DNS. בדרך כלל נשמע על DNS בהקשר של תרגום דומיינים לכתובות IP (Forward DNS). אולם ניתן להיעזר ב-DNS גם לתהליך ההפוך, שמכונה: Reverse DNS (rDNS) - כלומר, תרגום כתובות IP לדומיינים:



[מקור: https://www.leadfeeder.com/blog/what-is-reverse-dns-and-why-you-should-care]

כאשר נגדיר לסניפר לבצע תרגום שכזה - בכל פעם שהוא ייתקל בכתובת IP לא מוכרת - הוא ישלח בקשה לשרת DNS עם כתובת ה-IP שהוא ראה. במידה והשרת יחזיר תשובה - הוא יציג את הדומיין שחזר במקום את כתובת ה-IP.

ב-Wireshark ניתן להגדיר את ביצוע התרגום דרך מסך ה-Capture Options:



ב-tcpdump התהליך מתבצע באופן דיפולטי (ניתן לבטל אותו באמצעות הדגל **-n**).

אנחנו ננסה לנצל את מה שלמדנו ולזייף תעבורה עם כתובות מזויפות ובכך "לפתות" את הרחרחן לחשוף את עצמו.

במידה ונראה Reverse DNS Lookups על אותן כתובות, נדע שיש לנו רחרחן ברשת. הקוד המלא קצת ארוך ולכן אסביר רק את הפונקציה העיקרית בתוכנית:

```
27  def detect_promiscuous():
28      # Get pseudo random generated address
29      fake_ip_address = get_fake_ip()
30
31      # Build Fake request to a none-existing web server
32      request_packet = Ether(dst=FAKE_MAC)
33      request_packet /= IP(dst=fake_ip_address)
34      request_packet /= TCP(sport=randint(1025, 65535), dport=80, flags="S")
35      sendp(request_packet, iface=VM_INTERFACE, verbose=False)
36
37      # Filter the results to get only dns requests
38      dns_packets = sniff(filter="udp and dst port 53", timeout=MAX_TIMEOUT, iface=VM_INTERFACE)
39      sniffing_hosts = get_sniffing_hosts(dns_packets, fake_ip_address)
40
41      if len(sniffing_hosts) == 0:
42          print "There is no sniffer in the network."
43      else:
44          print "Found {SNIFFERS_NUMBER} sniffer(s) in the network:".format(SNIFFERS_NUMBER=len(sniffing_hosts))
45          for host in sniffing_hosts:
46              print "\tMAC: {MAC_ADDRESS}. IP: {IP_ADDRESS}".format(MAC_ADDRESS=host[0], IP_ADDRESS=host[1])
```

27. שימו לב שבניגוד לסקריפטים הקודמים, כאן אנחנו לא מציינים את הכתובת של המחשב שאנו חושדים בו - משום שאנחנו לא מייעדים את הבקשות למחשב ספציפי, אלא פשוט שולחים בקשות ברשת ובודקים מי מגיב.
29. הפונקציה get_fake_ip מחזירה את הכתובת IP שבה נשתמש. אנחנו צריכים לג'נרט כתובת שונה בכל פעם כי לאחר שהרחרחנים גילו מה הדומיין של כתובת ספציפית, או לחלופין, הבינו שהם לא יקבלו תשובה משרת ה-DNS - הם מפסיקים לנסות. לכן, אם נזייף תקשורת לאותה כתובת קבועה - בפעם הראשונה הרחרחנים ברשת ינסו לרזלב (לבצע Resolution) אותה, אבל לאחר הניסיון הראשון לא נראה יותר בקשות rDNS.
32. (עד 35) בניית ושליחת חבילה לכתובת המזויפת.
38. נסניף את הרשת ונקבל פקטות udp (DNS רץ מעל UDP) שמכוונות לפורט 53 (כלומר בקשות DNS).
39. הפונקציה get_sniffing_hosts סורקת את רשימת הפקטות שהתקבלו, עבור כל בקשת DNS היא מוודאת שמדובר ב-rDNS ושהכתובת שמנסים לתרגם היא אותה כתובת מזוייפת שיצרנו. אם הבקשה עונה לכל התנאים- המחשב ששלח אותה מריץ סניפר ונוסיף את כתובתו לרשימת הכתובות המסניפות.
41. (עד 46) הדפסת הכתובות שחזרו מ-get_sniffing_hosts (או הדפסה שלא נמצא מחשב מסניף).

הרצתי את הסקריפט גם על tcpdump (במצב דיפולטי) וגם על Wireshark (עם האופציה של Name Resolution מאופשרת):

```
root@osboxes:/home/osboxes# tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
```

שני הסניפרים ניסו לרזלב את הכתובות המזויפות שנשלחו ונלכדו ב-Anti Sniffer שבנינו:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.100.102.10	3.3.3.3	TCP	60	13658 > http [SYN] Seq=0 Win=8192 Len=0
2	0.328739	osboxes.local	192.168.237.1	DNS	80	Standard query PTR 3.3.3.3.in-addr.arpa
3	5.066500	osboxes.local	192.168.237.1	DNS	80	Standard query PTR 3.3.3.3.in-addr.arpa

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.100.102.10	4.4.4.4	TCP	54	50895 → http(80) [SYN] Seq=0 Win=8192 Len=0
2	0.001352	192.168.237.128	192.168.237.1	DNS	80	Standard query 0xcc23 PTR 4.4.4.4.in-addr.arpa
3	5.019982	192.168.237.128	192.168.237.1	DNS	80	Standard query 0xcc23 PTR 4.4.4.4.in-addr.arpa

והפלט:

```
C:\Users\bnaya\Desktop\scripts>dns_detection.py
Found 1 sniffer(s) in the network:
        MAC: 00:0c:29:0c:6b:c8. IP: 192.168.237.128
```

סיכום חלק זה

בחלק זה סקרנו את שיטת ה-DNS Decoy. זייפנו תעבורה עם כתובת שלא הוקצתה לאף מחשב ובדקנו אם מישהו שולח בקשות rDNS על אותה כתובת מזויפת. לשיטה זו יתרון מסוים על פני קודמותיה - אנחנו לא צריכים לבדוק בנפרד כל מחשב ברשת - אלא ניתן לבדוק בבת אחת את כלל המחשבים שחשופים לתעבורה המזויפת. עם זאת, היא מסתמכת על התנהגות תוכנת הרחרחן, ולעיתים קרובות לא תצליח לגלות מחשבים מסניפים - פשוט משום שתוכנת הרחרחן שרצה עליהם הוגדרה לא לבצע Resolution לכתובות IP.

סיכום

במאמר סקרנו והדגמנו את ארבעת הטכניקות המרכזיות בהן עושים שימוש אנטי-סניפרים המנסים לקבוע את מצבו של כרטיס רשת. שתי הטכניקות הראשונות שראינו, עושות שימוש בפרוטוקולי ARP ו-ICMP ומתבססות על העיקרון של שליחת חבילות שאמורות להידחות על ידי כרטיס רשת רגיל אבל להתקבל ולהיענות על ידי כרטיס רשת פרוץ.

ראינו שישנם הבדלים בין מערכות הפעלה שונות וכן ראינו שאנו עשויים לאבחן את מצבו של כרטיס רשת פרוץ באופן שגוי בגלל חוסר תגובה מצידו (למשל בגלל Firewall).

לאחר מכן סקרנו את שיטת הlatency בה הנחנו שנצליח לאבחן את מצבי כרטיס הרשת על סמך הבדלים בזמן התגובה של מחשב היעד. לא הצלחנו לעשות זאת במסגרת המאמר.

בטכניקה האחרונה, זייפנו תעבורה מכתובות פיקטיביות ובדקנו אם תוכנת הרחרחן תנסה לשלוח שאילתות rDNS על אותן כתובות.

נקודה שמעניין לציין היא שכמעט בכל הטכניקות שראינו (למעט latency), השגיאות היחידות הן מסוג False Negatives, כלומר, חוסר הצלחה בזיהוי של מצב פרוץ. אולם אם סקריפט אומר שכרטיס רשת הוא פרוץ - כמעט בוודאות הוא אכן כזה.

יש עוד הרבה מה להרחיב בנושא ולצערי לא הספקתי לגעת בנושאים מעניינים כמו טכניקות נוספות לזיהוי מצב פרוץ או טכניקות אנטי-גילוי והתחמקות של סניפרים. באופן אישי, הייתי רוצה להתעמק במימושים מוצלחים לשיטת ה-Latency ולהבין מדוע לא הצלחנו להגיע לתוצאות משמעותיות. כמו כן, יהיה מעניין לעבור על הקוד של הקרנל של לינוקס האחראי לטיפול ב-ARP ולבחון את בדיקות התקינות שמבצעת מ"ה על המסגרת.

על המחבר

בניה, בן 19, סטודנט למדעי המחשב באוניברסיטה הפתוחה. מתעניין בפיתוח תוכנה, בתקשורת ובאבטחת מידע. ניתן לפנות אליי לכל שאלה: bnayaYoo@gmail.com. כלל הסקריפטים שהופיעו במסמך, בתוספת תיעוד קצר ותיקונים נמצאים בקישור הבא:

https://github.com/bnaya19/PromiscuousModeDetection/tree/master/detection%20scripts

מקורות לתמונות ולתרשימים

- https://en.wikipedia.org/wiki/Promiscuous_mode
- https://www.cubrid.org/blog/understanding-tcp-ip-network-stack
- https://www.gatevidyalay.com/ethernet-ethernet-frame-format
- http://www.just.edu.jo/~tawalbeh/nyit/incs745/presentations/Sniffers.pdf
- https://slideplayer.com/slide/9510821/
- https://www.leadfeeder.com/blog/what-is-reverse-dns-and-why-you-should-care/

קישורים להעמקה נוספת

- http://www.securityfriday.com/promiscuous_detection_01.pdf
- http://hadmernok.hu/132_27_nagyd_1.pdf
- http://hadmernok.hu/132_28_nagyd_2.pdf
- https://pdfs.semanticscholar.org/e71d/fb37402252ef66072518720fc217891e1bd4.pdf
- http://www.lsv.fr/~goubault/SECI-02/Final/actes-seci02/pdf/008-Abdelallahelhadj.pdf
- https://www.iasj.net/iasj?func=fulltext&aId=29687
- https://gupea.ub.gu.se/bitstream/2077/1159/1/Nr_3_DS.pdf
- http://www.lsv.fr/~goubault/SECI-02/Final/actes-seci02/pdf/008-Abdelallahelhadj.pdf
- https://shodhganga.inflibnet.ac.in/bitstream/10603/108363/9/09_chapter%203.pdf