Kernel-Hack-Drill: Environment For Developing Linux Kernel Exploits

Alexander Popov

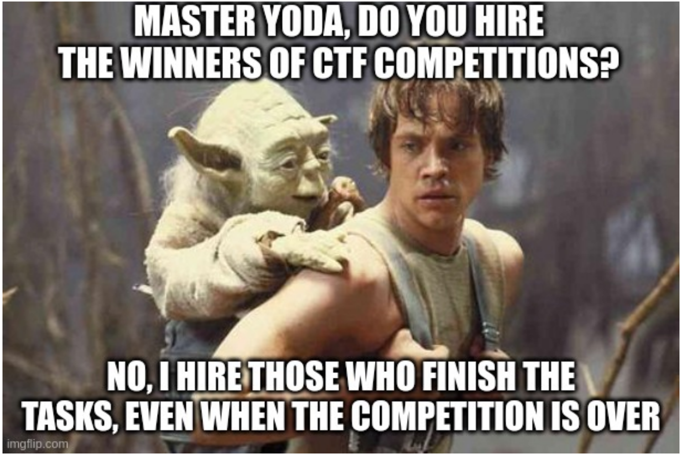
positive technologies

ZER0CON

April 10, 2025

Teaser

Teaser


MASTER YODA, DO YOU HIRE THE WINNERS OF CTF COMPETITIONS?
NO, I HIRE THOSE WHO FINISH THE TASKS, EVEN WHEN THE COMPETITION IS OVER
imgflip.com

Who Am I

- Alexander Popov
- Linux kernel developer since 2012
- Maintainer of some free software projects
- Principal Security Researcher and Head of
  Open Source Program Office at positive technologies

- Conference speaker:
  Zer0Con, OffensiveCon, H2HC, Nullcon Goa, Linux Security Summit, Still Hacking Anyway, HITB, Positive Hack Days, ZeroNights, HighLoad++, Open Source Summit, OS Day, Linux Plumbers...
  a13xp0p0v.github.io/conference_talks

Agenda

1. The bug collision story
2. About **CVE-2024-50264**
3. A new approach to exploiting it
4. How **kernel-hack-drill** helped to achieve this

How It Began

- I first found and exploited a bug in `AF_VSOCK` in 2021:

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

  a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html
- In spring 2024, I was fuzzing the kernel with a customized syzkaller
- I found another bug in `AF_VSOCK` in April 2024
- I minimized the reproducer, disabled KASAN and

  got instant null-ptr-deref in a kernel worker
- Postponed this bug

Bug Collision

- I decided to look at this bug again in autumn 2024
- Results were promising but then...

Bug Collision

- I decided to look at this bug again in autumn 2024
- Results were promising but then...
- Got bug collision with Hyunwoo Kim (@v4bel) and Wongi Lee (@qwerty)
- They disclosed this bug as **CVE-2024-50264** and used it at kernelCTF
- Their patch turned my PoC into null-ptr-deref

Diffstat (limited to 'net/vmw_vsock')

-rw-r--r-- net/vmw_vsock/virtio_transport_common.c 1

1 files changed, 1 insertions, 0 deletions

```
diff --git a/net/vmw_vsock/virtio_transport_common.c b/net/vmw_vsock/virtio_transport_common.c
index ccbd2bc0d2109a..fc5666c8298f7b 100644
--- a/net/vmw_vsock/virtio_transport_common.c
+++ b/net/vmw_vsock/virtio_transport_common.c
@@ -1109,6 +1109,7 @@ void virtio_transport_destruct(struct vsock_sock *vsk)
 	struct virtio_vsock_sock *vvs = vsk->trans;
 
 	kfree(vvs);
+	vsk->trans = NULL;
 }
 EXPORT_SYMBOL_GPL(virtio_transport_destruct);
```

Continue Anyway

- The exploit strategy by `@v4bel` and `@qwerty` looked very complicated

  github.com/google/security-research/pull/145/files

- I had some different ideas and decided to continue my research anyway

- I chose **Ubuntu Server 24.04** with a fresh OEM/HWE kernel (**v6.11**) as the target

Viktor Vasnetsov: The Knight at the Crossroads (1878)

CVE-2024-50264

- The bug was introduced in August 2016 (commit 06a8fc78367d, Linux v4.8)
- Race condition in `AF_VSOCK` sockets between `connect()` and a POSIX signal
- `CONFIG_USER_NS` is not required
- UAF on `virtio_vsock_sock` object (`kmalloc-96`)
- Memory corruption: UAF write in a kernel worker
- It has a lot of nasty limitations for the exploitation
    - **The worst bug** for the exploitation that I've ever seen

Reproducing CVE-2024-50264: Immortal Signal Handler

- @v4bel & @qwerty used SIGKILL
- My fuzzer found another approach, which amazed me

```
struct sigevent sev = {};
timer_t race_timer = 0;
sev.sigev_notify = SIGEV_SIGNAL; /* Notification type */
sev.sigev_signo = 33; /* Secret NPTL Signal (see nptl(7)) */
ret = timer_create(CLOCK_MONOTONIC, &sev, &race_timer);
```

- Native POSIX Threads Library makes internal use of signal 33
- Syscall wrappers and glibc functions **hide this signal** from applications
- So I can use `timer_settime()` for `race_timer`
  - It gives **control of timing**: at which moment signal should interrupt `connect()`
  - It is invisible for the exploit process and **doesn't kill it**

CVE-2024-50264: Code Performing UAF Write

- This function is called **in kworker** after `virtio_vsock_sock` is freed

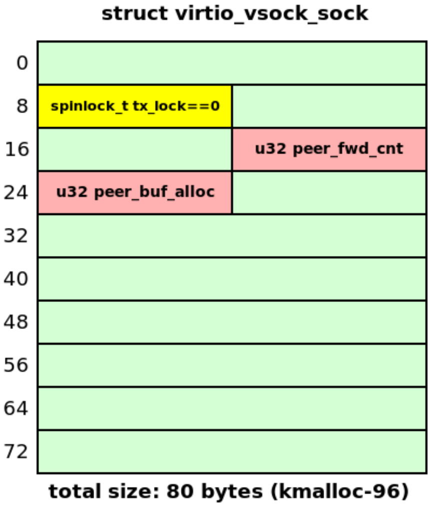
```
static bool virtio_transport_space_update(struct sock *sk,
                                          struct sk_buff *skb)
{
    struct virtio_vsock_hdr *hdr = virtio_vsock_hdr(skb);
    struct vsock_sock *vsk = vsock_sk(sk);
    struct virtio_vsock_sock *vvs = vsk->trans;        /* ptr to freed object */
    bool space_available;

    if (!vvs)
        return true;

    spin_lock_bh(&vvs->tx_lock); /* proceed if 4 bytes are zero (UAF write non-zero to lock) */
    vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc); /* UAF write 4 bytes */
    vvs->peer_fwd_cnt = le32_to_cpu(hdr->fwd_cnt);     /* UAF write 4 bytes */
    space_available = virtio_transport_has_space(vsk); /* UAF read, not interesting */
    spin_unlock_bh(&vvs->tx_lock);                     /* UAF write, restore 4 zero bytes */
    return space_available;
}
```

- There is no pointer dereference in freed object

CVE-2024-50264: UAF Write



UAF Write: Data Control

- About `virtio_vsock_sock.peer_buf_alloc` value control from userspace:

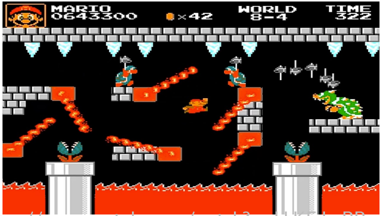
```
/* Increase the range for the value that we want to write during UAF: */
uaf_val_limit = 0x1lu; /* can't be zero */
setsockopt(vsock1, PF_VSOCK, SO_VM_SOCKETS_BUFFER_MIN_SIZE,
           &uaf_val_limit, sizeof(uaf_val_limit));
uaf_val_limit = 0xfffffffflu;
setsockopt(vsock1, PF_VSOCK, SO_VM_SOCKETS_BUFFER_MAX_SIZE,
           &uaf_val_limit, sizeof(uaf_val_limit));

/* Set the 4-byte value that we want to write during UAF: */
setsockopt(vsock1, PF_VSOCK, SO_VM_SOCKETS_BUFFER_SIZE,
           &uaf_val, sizeof(uaf_val));
```

- About `virtio_vsock_sock.peer_fwd_cnt` value control from userspace:
    - It represents the number of bytes pushed through `vsock` using `sendmsg()/recvmsg()`
    - Zero by default (4 zero bytes)

Not So Fast: CVE-2024-50264 Limitations

1. Vulnerable `virtio_vsock_sock` client object is allocated together with the server one
   - It's bad for cross-cache attack



https://www.youtube.com/watch?v=hbKEdmPPxy4

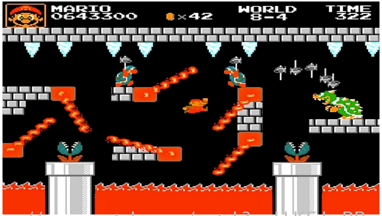
Not So Fast: CVE-2024-50264 Limitations

1. Vulnerable `virtio_vsock_sock` client object is allocated together with the server one
   - It's bad for cross-cache attack
2. Reproducing this race condition is very unstable

https://www.youtube.com/watch?v=hbKEdmPPxy4

Not So Fast: CVE-2024-50264 Limitations

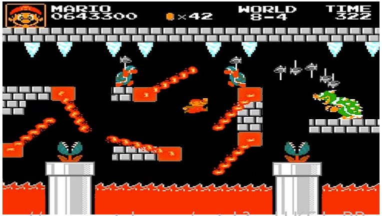
1. Vulnerable `virtio_vsock_sock` client object is allocated together with the server one
   - It's bad for cross-cache attack
2. Reproducing this race condition is very unstable
3. UAF write happens in kworker within few µs after `kfree()`



https://www.youtube.com/watch?v=hbKEdmPPxy4

Not So Fast: CVE-2024-50264 Limitations

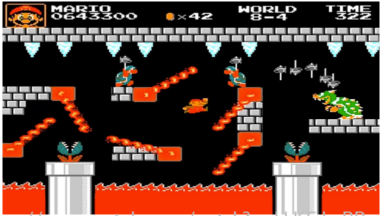
1. Vulnerable `virtio_vsock_sock` client object is allocated together with the server one
    - It's bad for cross-cache attack
2. Reproducing this race condition is very unstable
3. UAF write happens in kworker within few µs after `kfree()`
4. Null-ptr-deref happens in kworker right after UAF write



https://www.youtube.com/watch?v=hbKEdmPPxy4

Not So Fast: CVE-2024-50264 Limitations

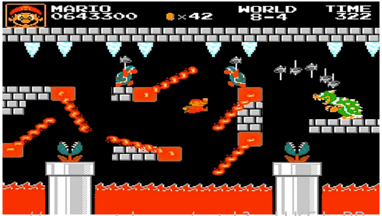
1. Vulnerable `virtio_vsock_sock` client object is allocated together with the server one
   - It's bad for cross-cache attack
2. Reproducing this race condition is very unstable
3. UAF write happens in kworker within few µs after `kfree()`
4. Null-ptr-deref happens in kworker right after UAF write
5. If this kernel oops is avoided, another null-ptr-deref happens in kworker after `VSOCK_CLOSE_TIMEOUT` (8 sec)



https://www.youtube.com/watch?v=hbKEdmPPxy4

Not So Fast: CVE-2024-50264 Limitations

1. Vulnerable `virtio_vsock_sock` client object is allocated together with the server one
    - It's bad for cross-cache attack
2. Reproducing this race condition is very unstable
3. UAF write happens in kworker within few `µs` after `kfree()`
4. Null-ptr-deref happens in kworker right after UAF write
5. If this kernel oops is avoided, another null-ptr-deref happens in kworker after `VSOCK_CLOSE_TIMEOUT` (8 sec)
6. Kworker hangs if `virtio_vsock_sock.tx_lock` is not zero



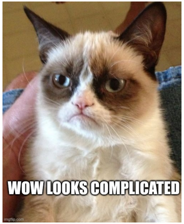
https://www.youtube.com/watch?v=hbKEdmPPxy4

Not So Fast: CVE-2024-50264 Limitations

Challenge

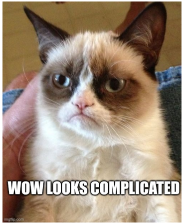
`Now you can see why this was the worst bug for exploitation I had ever seen`

1. Large-scale BPF JIT Spray populating a significant portion of the physical memory



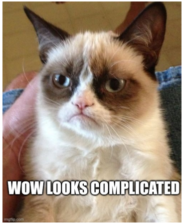
Exploit Strategy of @v4bel & @qwerty

1. Large-scale BPF JIT Spray populating a significant portion of the physical memory
2. SLUBStick technique github.com/IAIK/SLUBStick
   - Using timing side channel to determine number of objects in the active slab
   - Allocating the `virtio_vsock_sock` client and server objects in different slabs
   - It's possible by making them the last and first objects in slabs



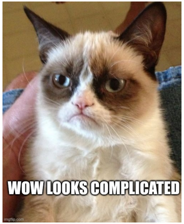
Exploit Strategy of @v4bel & @qwerty

1. Large-scale BPF JIT Spray populating a significant portion of the physical memory
2. SLUBStick technique github.com/IAIK/SLUBStick
   - Using timing side channel to determine number of objects in the active slab
   - Allocating the `virtio_vsock_sock` client and server objects in different slabs
   - It's possible by making them the last and first objects in slabs
3. Dirty Pagetable technique yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html
   - Cross-allocator attack reclaiming slab with UAF object for Page Table Entry
   - UAF write to PTE to make it possibly point a BPF JIT region

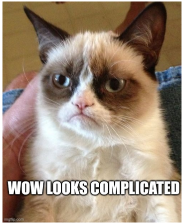


Exploit Strategy of @v4bel & @qwerty

1. Large-scale BPF JIT Spray populating a significant portion of the physical memory
2. SLUBStick technique github.com/IAIK/SLUBStick
   - Using timing side channel to determine number of objects in the active slab
   - Allocating the `virtio_vsock_sock` client and server objects in different slabs
   - It's possible by making them the last and first objects in slabs
3. Dirty Pagetable technique yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html
   - Cross-allocator attack reclaiming slab with UAF object for Page Table Entry
   - UAF write to PTE to make it possibly point a BPF JIT region
4. Inserting the privilege escalation payload into BPF code



Exploit Strategy of @v4bel & @qwerty



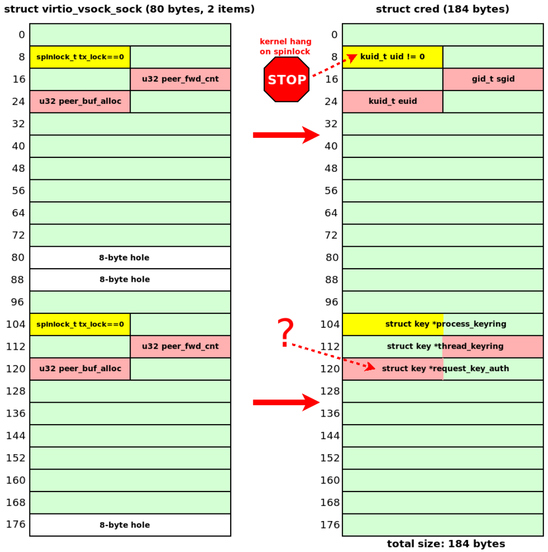
1. Large-scale BPF JIT Spray populating a significant portion of the physical memory
2. SLUBStick technique github.com/IAIK/SLUBStick
   - Using timing side channel to determine number of objects in the active slab
   - Allocating the `virtio_vsock_sock` client and server objects in different slabs
   - It's possible by making them the last and first objects in slabs
3. Dirty Pagetable technique yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html
   - Cross-allocator attack reclaiming slab with UAF object for Page Table Entry
   - UAF write to PTE to make it possibly point a BPF JIT region
4. Inserting the privilege escalation payload into BPF code
5. Socket communication to trigger the privesc payload

My First Ideas on Exploit Strategy

- Try UAF write to some kernel object
- Should I search kernel objects in `kmalloc-96`?
- No! Ubuntu Server 24.04 has:
  - `CONFIG_RANDOM_KMALLOC_CACHES=y`
  - `CONFIG_SLAB_BUCKETS=y`
  - `CONFIG_SLUB_CPU_PARTIAL=y`
- I will try cross-cache attack

Possible Target for UAF Write: struct cred

Target for UAF Write: struct cred (No Way)

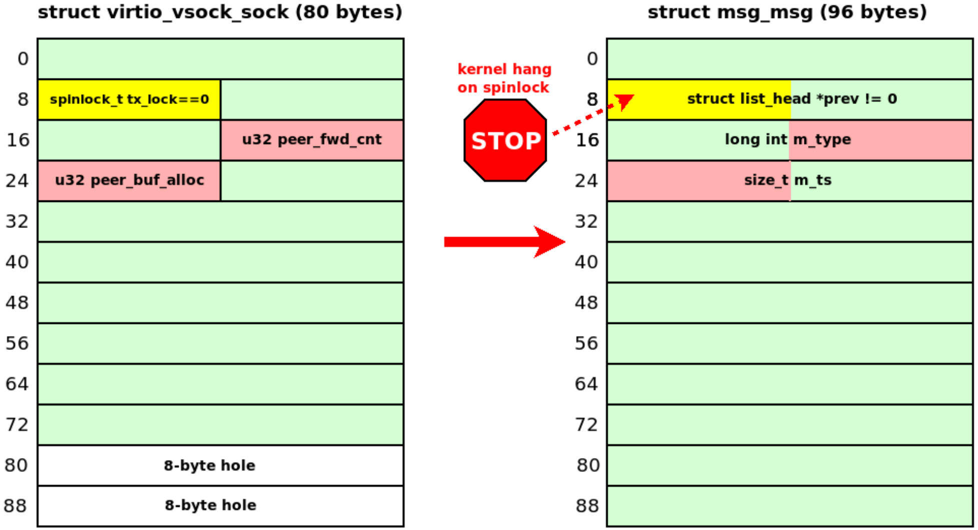


Target for UAF Write: struct msg_msg

- Why? Because I like it
- I first used it as a UAF target object in 2021

  a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html
- It was a novel approach back then
- I decided to create something new again

virtio_vsock_sock vs msg_msg



Bypassing the Unwanted msg_msg.m_list Corruption

- msg_msg.m_list.prev would be interpreted as non-null tx_lock
- virtio_transport_space_update() would hang in spin_lock_bh()
- Need to initialize msg_msg.m_list.prev **after** the UAF write
- Can we postpone placing msg_msg in the message queue?
- Yes!

Spray msg_msg Allowing m_list Corruption (Novel Technique?)

1. Fill the message queue almost completely before sending the target `msg_msg`
   - The message queue size is `MSGMNB` (16384 bytes)
   - Send 2 clogging messages of of 8191 bytes each
   - 2 bytes left in the queue, don't call `msgrcv()`

https://www.youtube.com/watch?v=0XVCz6nekJc

Spray msg_msg Allowing m_list Corruption (Novel Technique?)

1. Fill the message queue almost completely before sending the target `msg_msg`
   - The message queue size is `MSGMNB` (16384 bytes)
   - Send 2 clogging messages of of 8191 bytes each
   - 2 bytes left in the queue, don't call `msgrcv()`
2. Spray target `msg_msg` objects
   - Call the `msgsnd()` syscall in separate pthreads
   - Kernel allocates target `msg_msg` and `msgsnd()` blocks

https://www.youtube.com/watch?v=0XVCz6nekJc

Spray msg_msg Allowing m_list Corruption (Novel Technique?)

1. Fill the message queue almost completely before sending the target `msg_msg`
   - The message queue size is `MSGMNB` (16384 bytes)
   - Send 2 clogging messages of of 8191 bytes each
   - 2 bytes left in the queue, don't call `msgrcv()`
2. Spray target `msg_msg` objects
   - Call the `msgsnd()` syscall in separate pthreads
   - Kernel allocates target `msg_msg` and `msgsnd()` blocks
3. Perform UAF write, corrupt `msg_msg.m_list` as you want

https://www.youtube.com/watch?v=0XVCz6nekJc

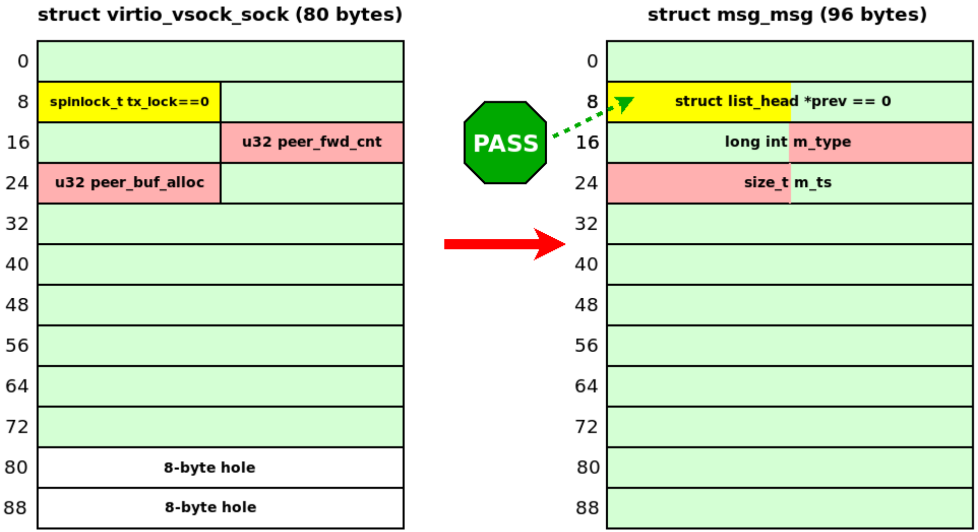
Spray msg_msg Allowing m_list Corruption (Novel Technique?)

1. Fill the message queue almost completely before sending the target msg_msg
   - The message queue size is MSGMNB (16384 bytes)
   - Send 2 clogging messages of of 8191 bytes each
   - 2 bytes left in the queue, don't call msgrcv()

https://www.youtube.com/watch?v=0XVCz6nekJc

2. Spray target msg_msg objects
   - Call the msgsnd() syscall in separate pthreads
   - Kernel allocates target msg_msg and msgsnd() blocks
3. Perform UAF write, corrupt msg_msg.m_list as you want
4. Perform msgrcv() for clogging messages
   - Now the kernel can add sprayed msg_msg to the queue
   - The **kernel fixes** the **corrupted** msg_msg.m_list pointers!

virtio_vsock_sock vs msg_msg



Nice Trick, What's Next?

1. I managed to overwrite `msg_msg.m_ts` and make kernel fix up `msg_msg.m_list`
   - This technique is also useful for **blind overwriting of** `msg_msg`
   - No kernel infoleak is needed — the kernel will restore the corrupted pointers

Nice Trick, What's Next?

1. I managed to overwrite `msg_msg.m_ts` and make kernel fix up `msg_msg.m_list`
   - This technique is also useful for **blind overwriting of** `msg_msg`
   - No kernel infoleak is needed — the kernel will restore the corrupted pointers
2. To use this trick, I needed to perform cross-cache attack
   - `virtio_vsock_sock` lives in one of 16 `kmalloc-rnd-?-96` slab caches (`CONFIG_RANDOM_KMALLOC_CACHES`)
   - `msg_msg` lives in `msg_msg-96` slab cache (`CONFIG_SLAB_BUCKETS`)

Nice Trick, What's Next?

1. I managed to overwrite `msg_msg.m_ts` and make kernel fix up `msg_msg.m_list`
   - This technique is also useful for **blind overwriting of** `msg_msg`
   - No kernel infoleak is needed — the kernel will restore the corrupted pointers
2. To use this trick, I needed to perform cross-cache attack
   - `virtio_vsock_sock` lives in one of 16 `kmalloc-rnd-?-96` slab caches (`CONFIG_RANDOM_KMALLOC_CACHES`)
   - `msg_msg` lives in `msg_msg-96` slab cache (`CONFIG_SLAB_BUCKETS`)
3. **Problems:**
   - I needed to learn how cross-cache attacks work on the latest Ubuntu kernel
   - Testing exploit primitives together with this crazy race condition was **painful**

Solution That Makes Researcher's Life Easier

Unstable race condition creating problems?

`Use a testing ground for developing the exploit primitives!`

Kernel Hack Drill

- Open-source project: github.com/a13xp0p0v/kernel-hack-drill
- Provides test environment for developing the Linux kernel exploit primitives you need
- Includes a good step-by-step setup guide in the README (kudos to the contributors!)
- A bit similar to github.com/hacktivesec/KRWX, but
  - Much simpler
  - Contains interesting PoC exploits

https://www.pngall.com/wp-content/uploads/4/Drill-Machine-PNG-Free-Download.png

Kernel Hack Drill Contents: Kernel Module

1. `drill_mod.c`
   - A small Linux kernel module
   - Provides `/proc/drill_act` file as a simple interface to userspace
   - Contains nice vulnerabilities that you control
2. `drill.h`
   - Header file describing the `drill_mod.ko` interface
3. `drill_test.c`
   - Userspace test for `drill_mod.ko`
   - It also passes if `CONFIG_KASAN=y`

```c
#define DRILL_N 10240
#define DRILL_ITEM_SIZE 95

struct drill_item_t {
    unsigned long foobar;
    void (*callback)(void);
    char data[]; /* C99 flexible array */
};

enum drill_act_t {
    DRILL_ACT_NONE = 0,
    DRILL_ACT_ALLOC = 1,
    DRILL_ACT_CALLBACK = 2,
    DRILL_ACT_SAVE_VAL = 3,
    DRILL_ACT_FREE = 4,
    DRILL_ACT_RESET = 5
};
```

Kernel Hack Drill Contents: PoC Exploits

1. drill_uaf_callback.c
   - UAF exploit invoking a callback in the freed drill_item_t struct
   - Performs control flow hijack and gains LPE

https://www.printables.com/model/78077-drill-guide

Kernel Hack Drill Contents: PoC Exploits

1. `drill_uaf_callback.c`
   - UAF exploit invoking a callback in the freed `drill_item_t` struct
   - Performs control flow hijack and gains LPE
2. `drill_uaf_write_msg_msg.c`
   - UAF exploit writing data to the freed `drill_item_t` struct
   - Performs a cross-cache attack, overwrites `msg_msg.m_ts`
   - Enables out-of-bounds read of the kernel memory

https://www.printables.com/model/78077-drill-guide

Kernel Hack Drill Contents: PoC Exploits

1. `drill_uaf_callback.c`
   - UAF exploit invoking a callback in the freed `drill_item_t` struct
   - Performs control flow hijack and gains LPE
2. `drill_uaf_write_msg_msg.c`
   - UAF exploit writing data to the freed `drill_item_t` struct
   - Performs a cross-cache attack, overwrites `msg_msg.m_ts`
   - Enables out-of-bounds read of the kernel memory
3. `drill_uaf_write_pipe_buffer.c`
   - UAF exploit writing data to the freed `drill_item_t` struct
   - Performs cross-cache attack, overwrites `pipe_buffer.flags`
   - Implements the Dirty Pipe attack and gains LPE

https://www.printables.com/model/78077-drill-guide

Kernel Hack Drill Contents: PoC Exploits

1. drill_uaf_callback.c
   - UAF exploit invoking a callback in the freed drill_item_t struct
   - Performs control flow hijack and gains LPE
2. drill_uaf_write_msg_msg.c
   - UAF exploit writing data to the freed drill_item_t struct
   - Performs a cross-cache attack, overwrites msg_msg.m_ts
   - Enables out-of-bounds read of the kernel memory
3. drill_uaf_write_pipe_buffer.c
   - UAF exploit writing data to the freed drill_item_t struct
   - Performs cross-cache attack, overwrites pipe_buffer.flags
   - Implements the Dirty Pipe attack and gains LPE
4. More PoC exploits will come soon!

https://www.printables.com/model/78077-drill-guide

Cross-Cache Attack in Kernel Hack Drill

Standard cross-cache procedure, see the code: kernel-hack-drill/drill_uaf_write_msg_msg.c

Cross-Cache Attack in Kernel Hack Drill

Standard cross-cache procedure, see the code: kernel-hack-drill/drill_uaf_write_msg_msg.c

1. Collect the needed info in `/sys/kernel/slab`: `cpu_partial=120`, `objs_per_slab=42`

Cross-Cache Attack in Kernel Hack Drill

Standard cross-cache procedure, see the code: kernel-hack-drill/drill_uaf_write_msg_msg.c

1. Collect the needed info in `/sys/kernel/slab`: `cpu_partial=120`, `objs_per_slab=42`
2. Create a new active slab: allocate `objs_per_slab` objects

Cross-Cache Attack in Kernel Hack Drill

Standard cross-cache procedure, see the code: kernel-hack-drill/drill_uaf_write_msg_msg.c

1. Collect the needed info in `/sys/kernel/slab`: `cpu_partial=120`, `objs_per_slab=42`
2. Create a new active slab: allocate `objs_per_slab` objects
3. Allocate `(objs_per_slab * cpu_partial)` objects for the partial list

Cross-Cache Attack in Kernel Hack Drill

Standard cross-cache procedure, see the code: kernel-hack-drill/drill_uaf_write_msg_msg.c

1. Collect the needed info in `/sys/kernel/slab`: `cpu_partial=120`, `objs_per_slab=42`
2. Create a new active slab: allocate `objs_per_slab` objects
3. Allocate `(objs_per_slab * cpu_partial)` objects for the partial list
4. Create the vulnerable slab: allocate `objs_per_slab` objects

Standard cross-cache procedure, see the code: kernel-hack-drill/drill_uaf_write_msg_msg.c

1. Collect the needed info in `/sys/kernel/slab`: `cpu_partial=120`, `objs_per_slab=42`
2. Create a new active slab: allocate `objs_per_slab` objects
3. Allocate `(objs_per_slab * cpu_partial)` objects for the partial list
4. Create the vulnerable slab: allocate `objs_per_slab` objects
5. Obtain dangling reference to the vulnerable object for UAF

Cross-Cache Attack in Kernel Hack Drill

Standard cross-cache procedure, see the code: kernel-hack-drill/drill_uaf_write_msg_msg.c

1. Collect the needed info in `/sys/kernel/slab`: `cpu_partial=120`, `objs_per_slab=42`
2. Create a new active slab: allocate `objs_per_slab` objects
3. Allocate `(objs_per_slab * cpu_partial)` objects for the partial list
4. Create the vulnerable slab: allocate `objs_per_slab` objects
5. Obtain dangling reference to the vulnerable object for UAF
6. Create a new active slab: allocate `objs_per_slab` objects

Cross-Cache Attack in Kernel Hack Drill

Standard cross-cache procedure, see the code: kernel-hack-drill/drill_uaf_write_msg_msg.c

1. Collect the needed info in `/sys/kernel/slab`: `cpu_partial=120`, `objs_per_slab=42`
2. Create a new active slab: allocate `objs_per_slab` objects
3. Allocate `(objs_per_slab * cpu_partial)` objects for the partial list
4. Create the vulnerable slab: allocate `objs_per_slab` objects
5. Obtain dangling reference to the vulnerable object for UAF
6. Create a new active slab: allocate `objs_per_slab` objects
7. Free the slab with UAF object: free `(objs_per_slab * 2 - 1)` objects before the last one

Cross-Cache Attack in Kernel Hack Drill

Standard cross-cache procedure, see the code: kernel-hack-drill/drill_uaf_write_msg_msg.c

1. Collect the needed info in `/sys/kernel/slab`: `cpu_partial=120`, `objs_per_slab=42`
2. Create a new active slab: allocate `objs_per_slab` objects
3. Allocate `(objs_per_slab * cpu_partial)` objects for the partial list
4. Create the vulnerable slab: allocate `objs_per_slab` objects
5. Obtain dangling reference to the vulnerable object for UAF
6. Create a new active slab: allocate `objs_per_slab` objects
7. Free the slab with UAF object: free `(objs_per_slab * 2 - 1)` objects before the last one
8. Clean up the partial list: free one of each `objs_per_slab` objects in the reserved slabs

Cross-Cache Attack in Kernel Hack Drill

Standard cross-cache procedure, see the code: kernel-hack-drill/drill_uaf_write_msg_msg.c

1. Collect the needed info in `/sys/kernel/slab`: `cpu_partial=120`, `objs_per_slab=42`
2. Create a new active slab: allocate `objs_per_slab` objects
3. Allocate `(objs_per_slab * cpu_partial)` objects for the partial list
4. Create the vulnerable slab: allocate `objs_per_slab` objects
5. Obtain dangling reference to the vulnerable object for UAF
6. Create a new active slab: allocate `objs_per_slab` objects
7. Free the slab with UAF object: free `(objs_per_slab * 2 - 1)` objects before the last one
8. Clean up the partial list: free one of each `objs_per_slab` objects in the reserved slabs
9. Reclaim the page with UAF object: spray target objects

Cross-Cache Attack in Kernel Hack Drill

Standard cross-cache procedure, see the code: kernel-hack-drill/drill_uaf_write_msg_msg.c

1. Collect the needed info in `/sys/kernel/slab`: `cpu_partial=120`, `objs_per_slab=42`
2. Create a new active slab: allocate `objs_per_slab` objects
3. Allocate `(objs_per_slab * cpu_partial)` objects for the partial list
4. Create the vulnerable slab: allocate `objs_per_slab` objects
5. Obtain dangling reference to the vulnerable object for UAF
6. Create a new active slab: allocate `objs_per_slab` objects
7. Free the slab with UAF object: free `(objs_per_slab * 2 - 1)` objects before the last one
8. Clean up the partial list: free one of each `objs_per_slab` objects in the reserved slabs
9. Reclaim the page with UAF object: spray target objects
10. Exploit UAF

Debugging Cross-Cache Attack: Kernel Patch

```
diff --git a/ipc/msgutil.c b/ipc/msgutil.c
@@ -64,6 +64,7 @@ static struct msg_msg *alloc_msg(size_t len)
        msg = kmem_buckets_alloc(msg_buckets, sizeof(*msg) + alen, GFP_KERNEL);
        if (msg == NULL)
                return NULL;
+       printk("msg_msg 0x%lx\n", (unsigned long)msg);

        msg->next = NULL;
        msg->security = NULL;

diff --git a/mm/slub.c b/mm/slub.c
@@ -3140,6 +3140,7 @@ static void __put_partials(struct kmem_cache *s, struct slab *partial_slab)
        while (slab_to_discard) {
                slab = slab_to_discard;
                slab_to_discard = slab_to_discard->next;
+               printk("__put_partials: cache 0x%lx slab 0x%lx\n", (unsigned long)s, (unsigned long)slab);

                stat(s, DEACTIVATE_EMPTY);
                discard_slab(s, slab);
```

- `__put_partials()` calls `discard_slab()`, which moves the slab to the page allocator

Debugging Cross-Cache Attack: Console Output and GDB

- Legend: kernel log, stdout, GDB session (with bata24/gef)

```
[   49.755740] drill: kmalloc'ed item 5081 (0xffff8880068878a0, size 95)

[+] current_n: 5082 (next for allocating)
4) obtain dangling reference from use-after-free bug
[+] uaf_n: 5081

gef> slab-contains 0xffff8880068878a0
[+] Wait for memory scan
slab: 0xffffea00001a21c0
kmem_cache: 0xffff88800384e800
base: 0xffff888006887000
name: kmalloc-rnd-14-96  size: 0x60  num_pages: 0x1

[   51.371255] __put_partials: cache 0xffff88800384e800 slab 0xffffea00001a21c0
[   51.463570] msg_msg 0xffff8880068878a0
```

- The `drill_item_t` object `0xffff8880068878a0` in slab `0xffffea00001a21c0` is reallocated as `msg_msg`

In My Humble Opinion

Cross-Cache Attack: Adoption to AF_VSOCK Exploit

- The vulnerable `virtio_vsock_sock` client object is allocated together with the server one
- It is harmful for the attack (**Limitation #1**):
  - Not closing server vsock prevents complete freeing of UAF slab
  - Closing server `vsock` breaks UAF
- How can we cope with it?
  - @v4bel and @qwerty used the **SLUBStick** technique

Cross-Cache Attack: Adoption to AF_VSOCK Exploit

- The vulnerable `virtio_vsock_sock` client object is allocated together with the server one
- It is harmful for the attack (**Limitation #1**):
  - Not closing server vsock prevents complete freeing of UAF slab
  - Closing server `vsock` breaks UAF
- How can we cope with it?
  - @v4bel and @qwerty used the **SLUBStick** technique
  - **My idea**: what if we hit `connect()` with a signal **very early**?

Race Conditions Are Awful/Awesome

I used one more race condition to exploit the main race condition

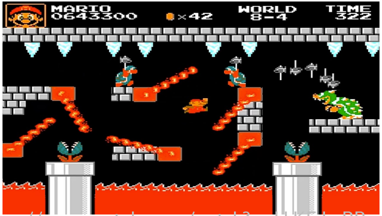
1. Hit `vsock connect()` with the "immortal" signal 33 after 10000 ns
2. Check whether the race condition succeeded:
   - The `connect()` syscall should return "Interrupted system call"
   - Connecting to server `vsock` from another test client `vsock` should succeed
3. If that is true, only a single vulnerable `vsock` was created
4. ~~Limitation #1~~ (paired object creation) is bypassed
5. Cool, the cross-cache attack for `vsock` is unlocked!

AF_VSOCK Exploit Speedrun

- This smart testing of `signal` vs `connect()` state also made the exploit **much faster**
  - The UAF write can now be triggered **once per second** instead of ~~once per several minutes~~
  - ~~**Limitation #2**~~ (unstable race condition) is mitigated
  - ~~**Limitation #5**~~ (kworker oops in 8 sec) is bypassed
- To counter ~~**Limitation #4**~~ (kworker oops just after UAF), I used one more race condition
  - Idea by @v4bel and @qwerty
  - Call `listen()` for vulnerable `vsock` just after `connect()` provoking UAF
  - If we are lucky, `listen()` executes before UAF-kworker and prevents null-ptr-deref
  - This is the main source of instability of the whole exploit 🙁

Not So Fast: CVE-2024-50264 Limitations

1. ~~Vulnerable `virtio_vsock_sock` client object is allocated together with the server one~~
2. ~~Reproducing this race condition is very unstable~~
3. UAF write happens in kworker within few µs after `kfree()`
4. ~~Null-ptr-deref happens in kworker right after UAF write~~
5. ~~If this kernel oops is avoided, another null-ptr-deref happens in kworker after `VSOCK_CLOSE_TIMEOUT` (8 sec)~~
6. Kworker hangs if `virtio_vsock_sock.tx_lock` is not zero



https://www.youtube.com/watch?v=hbKEdmPPxy4

Not So Fast: Cross-Cache Attack is Too Late

- UAF write in kworker happens within few µs after `kfree(virtio_vsock_sock)`
- The cross-cache attack is too slow

Not So Fast: Cross-Cache Attack is Too Late

- UAF write in kworker happens within few µs after `kfree(virtio_vsock_sock)`
- The cross-cache attack is too slow
- To deal with ~~**Limitation #3**~~, I also used a well-known technique by **Jann Horn**
  googleprojectzero.blogspot.com/2022/03/racing-against-clock-hitting-tiny.html
- Hit kworker with a timer interrupt that has **a lot of** `epoll` watches registered for `timerfd`

Not So Fast: Cross-Cache Attack is Too Late

- UAF write in kworker happens within few `µs` after `kfree(virtio_vsock_sock)`
- The cross-cache attack is too slow
- To deal with ~~**Limitation #3**~~, I also used a well-known technique by **Jann Horn**
  googleprojectzero.blogspot.com/2022/03/racing-against-clock-hitting-tiny.html
- Hit kworker with a timer interrupt that has **a lot of** `epoll` watches registered for `timerfd`
  1. Call `timerfd_create(CLOCK_MONOTONIC, 0)`

Not So Fast: Cross-Cache Attack is Too Late

- UAF write in kworker happens within few µs after `kfree(virtio_vsock_sock)`
- The cross-cache attack is too slow
- To deal with ~~**Limitation #3**~~, I also used a well-known technique by **Jann Horn**
  googleprojectzero.blogspot.com/2022/03/racing-against-clock-hitting-tiny.html
- Hit kworker with a timer interrupt that has **a lot of** `epoll` watches registered for `timerfd`
  1. Call `timerfd_create(CLOCK_MONOTONIC, 0)`
  2. Create 8 forks, call `dup()` 100 times for `timertfd` in each fork

Not So Fast: Cross-Cache Attack is Too Late

- UAF write in kworker happens within few µs after `kfree(virtio_vsock_sock)`
- The cross-cache attack is too slow
- To deal with ~~**Limitation #3**~~, I also used a well-known technique by **Jann Horn**
  googleprojectzero.blogspot.com/2022/03/racing-against-clock-hitting-tiny.html
- Hit kworker with a timer interrupt that has **a lot of** `epoll` watches registered for `timerfd`
  1. Call `timerfd_create(CLOCK_MONOTONIC, 0)`
  2. Create 8 forks, call `dup()` 100 times for `timertfd` in each fork
  3. Call `epoll_create()` 500 times in each fork, register `epoll fd` for each duplicated `fd`

Not So Fast: Cross-Cache Attack is Too Late

- UAF write in kworker happens within few µs after `kfree(virtio_vsock_sock)`
- The cross-cache attack is too slow
- To deal with ~~**Limitation #3**~~, I also used a well-known technique by **Jann Horn**
  googleprojectzero.blogspot.com/2022/03/racing-against-clock-hitting-tiny.html
- Hit kworker with a timer interrupt that has **a lot of** `epoll` watches registered for `timerfd`
  1. Call `timerfd_create(CLOCK_MONOTONIC, 0)`
  2. Create 8 forks, call `dup()` 100 times for `timertfd` in each fork
  3. Call `epoll_create()` 500 times in each fork, register `epoll fd` for each duplicated `fd`
  4. Don't exceed `/proc/sys/fs/epoll/max_user_watches` ($8 \times 100 \times 500 < 446976$)

Not So Fast: Cross-Cache Attack is Too Late

- UAF write in kworker happens within few µs after `kfree(virtio_vsock_sock)`
- The cross-cache attack is too slow
- To deal with ~~**Limitation #3**~~, I also used a well-known technique by **Jann Horn**
  googleprojectzero.blogspot.com/2022/03/racing-against-clock-hitting-tiny.html
- Hit kworker with a timer interrupt that has **a lot of** `epoll` watches registered for `timerfd`
  1. Call `timerfd_create(CLOCK_MONOTONIC, 0)`
  2. Create 8 forks, call `dup()` 100 times for `timertfd` in each fork
  3. Call `epoll_create()` 500 times in each fork, register `epoll fd` for each duplicated `fd`
  4. Don't exceed `/proc/sys/fs/epoll/max_user_watches` ($8 \times 100 \times 500 < 446976$)
  5. Shoot into kworker setting the proper timeout:
     `timerfd_settime(timerfd, TFD_TIMER_CANCEL_ON_SET, &retard_tmo, NULL)`

Achieved msg_msg Out-Of-Bounds Read

- vsock UAF changes the msg_msg data size from 48 bytes to 8192 (MSGMAX)
- Cool, msgrcv() performs out-of-bounds read of kernel memory
- What does infoleak provide?
  - A kernel address 0xffffffff8233cfa0
  - GDB shows that it is pointer to socket_file_ops()
  - Which kernel object stores it? It's struct file!
  - It contains f_cred pointer, which also leaked
- This infoleak works with high probability

What's Next?

The most interesting / difficult part of the research

`Then I needed arbitrary address writing`
`for privilege escalation.`
`I wanted to implement data-only attack`
`without control flow hijacking.`

How About Dirty Page Table Attack?

- Good description: yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html
- Attacking page tables requires knowing the physical address of kernel text/heap

How About Dirty Page Table Attack?

- Good description: yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html
- Attacking page tables requires knowing the physical address of kernel text/heap
- How about bruteforcing?
    - No, I can trigger UAF around 5 times before the kworker dies — not enough

How About Dirty Page Table Attack?

- Good description: yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html
- Attacking page tables requires knowing the physical address of kernel text/heap
- How about bruteforcing?
  - No, I can trigger UAF around 5 times before the kworker dies — not enough
- How about a KASLR infoleak from `msg_msg` out-of-bounds read?
  - Ok, let's give it a try!

Physical Versus Virtual KASLR

- VM run #1

```
gef> ksymaddr-remote
[+] Wait for memory scan
0xffffffff98400000 T _text

gef> v2p 0xffffffff98400000
Virt: 0xffffffff98400000 -> Phys: 0x57400000
```

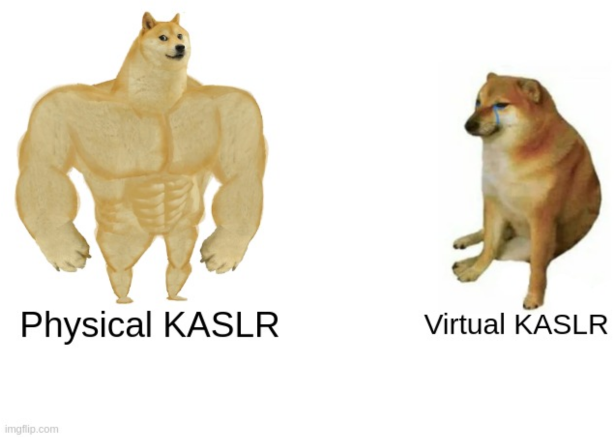
- VM run #2

```
gef> ksymaddr-remote
[+] Wait for memory scan
0xffffffff81800000 T _text

gef> v2p 0xffffffff81800000
Virt: 0xffffffff81800000 -> Phys: 0x18600000
```

- Virtual address minus physical address:
  - VM run #1: 0xffffffff98400000 − 0x57400000 = 0xffffffff41000000
  - VM run #2: 0xffffffff81800000 − 0x18600000 = 0xffffffff69200000
- 0xffffffff41000000 != 0xffffffff69200000
- Sorry, leaking the virtual KASLR offset **doesn't help** against the physical KASLR

Physical KASLR Versus Virtual KASLR



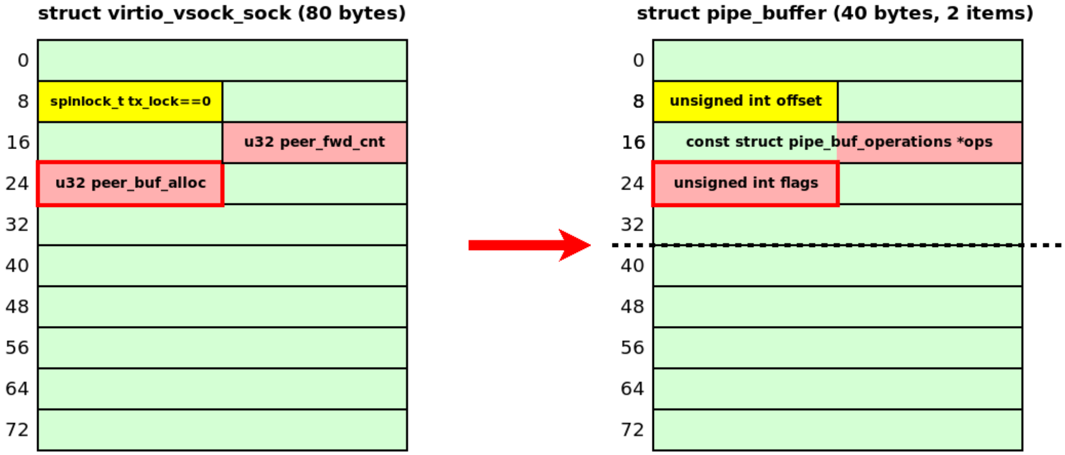
Still Needed to Invent Arbitrary Address Writing Primitive

1. Dirty Page Table Attack?
   - Requires page allocator feng-shui to leak the kernel physical address
   - No, would be too complicated

2. Turn UAF write to some kernel object into arbitrary address writing?
   - Not so easy... Exhausting!
   - Looked through dozens of different kernel objects
   - Read dozens of kernel exploit write-ups
   - Tried Kernel Exploitation Dashboard by Eduardo Vela & KernelCTF team
   - Then focused on `pipe_buffer` kernel object

Target for UAF Write: struct pipe_buffer

- We can make `pipe_buffer`s of similar size with `virtio_vsock_sock`:
  - Reallocate the write end of the pipe
  - `fcntl(pipe_fd[1], F_SETPIPE_SZ, PAGE_SIZE * 2);`
  - The object size becomes: `2 * sizeof(struct pipe_buffer) = 80`
  - Suitable for `kmalloc-96`, like `virtio_vsock_sock`
- Attacker-controlled bytes of `vsock` UAF write change `pipe_buffer.flags`
- It's the original **Dirty Pipe attack** by Max Kellermann dirtypipe.cm4all.com
- Even doesn't need an infoleak
- **One shot, wow, let's try!**


struct virtio_vsock_sock (80 bytes)
0
8
spinlock_t tx_lock==0
16
u32 peer_fwd_cnt
24
u32 peer_buf_alloc
32
40
48
56
64
72
struct pipe_buffer (40 bytes, 2 items)
0
8
unsigned int offset
16
const struct pipe_buf_operations *ops
24
unsigned int flags
32
40
48
56
64
72

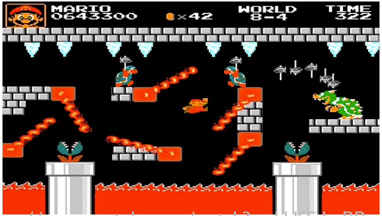
First of All, Drill!

- Created a Dirty Pipe prototype in kernel-hack-drill
- See the code: [kernel-hack-drill/drill_uaf_write_pipe_buffer.c]
    - Performs cross-cache attack: reclaims `drill_item_t` as `pipe_buffers`
    - Exploits UAF write to `drill_item_t` struct:
        - ⋆ Controlled bytes at offset 24
    - Attacker-controlled bytes modify `pipe_buffer.flags`
    - Implements the Dirty Pipe attack
    - LPE in one shot without infoleak

https://www.pngall.com/wp-content/uploads/4/Drill-Machine-PNG-Free-Download.png

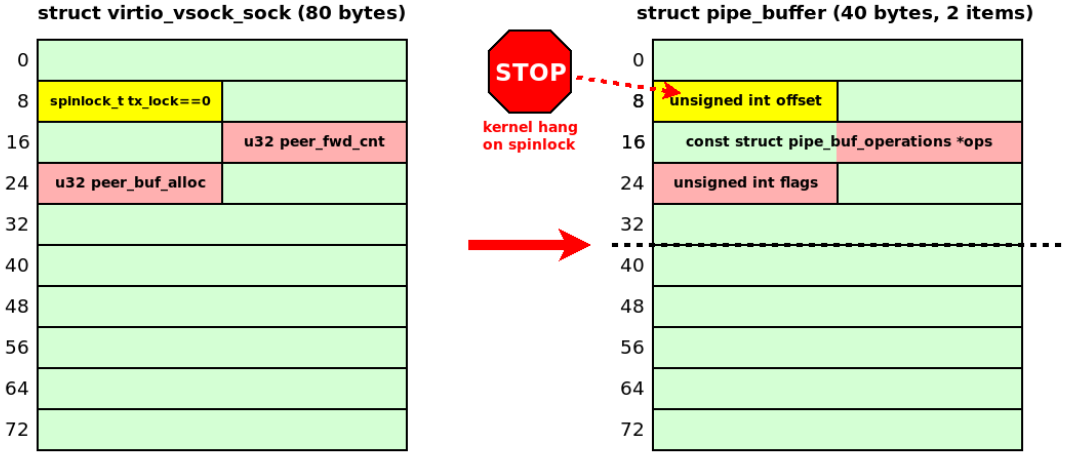
Not So Fast: CVE-2024-50264 Limitations

1. ~~Vulnerable `virtio_vsock_sock` client object is allocated together with the server one~~
2. ~~Reproducing this race condition is very unstable~~
3. ~~UAF write happens in kworker within few µs after `kfree()`~~
4. ~~Null-ptr-deref happens in kworker right after UAF write~~
5. ~~If this kernel oops is avoided, another null-ptr-deref happens in kworker after `VSOCK_CLOSE_TIMEOUT` (8 sec)~~
6. Kworker hangs if `virtio_vsock_sock.tx_lock` is not zero



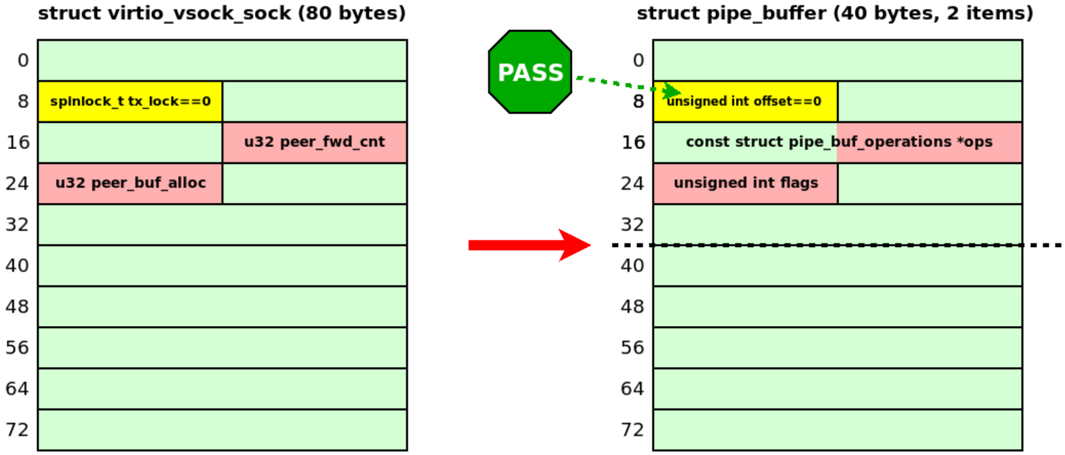
https://www.youtube.com/watch?v=hbKEdmPPxy4

Target for UAF Write: struct pipe_buffer



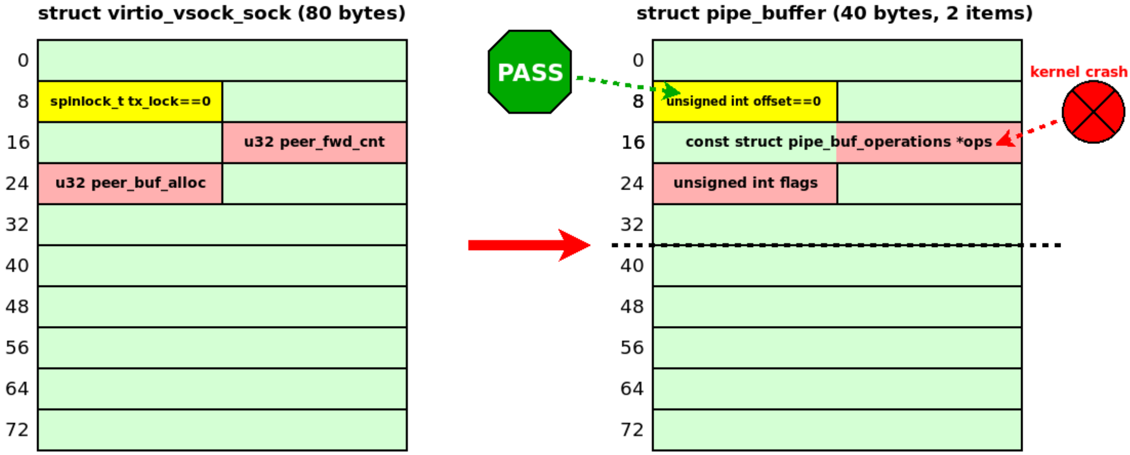
Target for UAF Write: struct pipe_buffer

I can do `splice()` from file to pipe starting from zero `offset` to bypass ~~Limitation #6~~!



Target for UAF Write: struct pipe_buffer (No Way)

Oh no, `pipe_buffer.ops` gets corrupted by 4 zero bytes of `peer_fwd_cnt`!



Target for UAF Write: struct pipe_buffer (No Way)

- Oh no, `pipe_buffer.ops` gets corrupted by 4 zero bytes of `peer_fwd_cnt`!
  - Changing peer_fwd_cnt requires sending data through vsock
  - But successful `vsock connect()` makes the UAF impossible
  - No way to execute the original Dirty Pipe attack 🙁

Target for UAF Write: struct pipe_buffer (No Way)

- Oh no, pipe_buffer.ops gets corrupted by 4 zero bytes of peer_fwd_cnt!
  - Changing peer_fwd_cnt requires sending data through vsock
  - But successful vsock connect() makes the UAF impossible
  - No way to execute the original Dirty Pipe attack 🙁
- **Suddenly I got a bright idea**

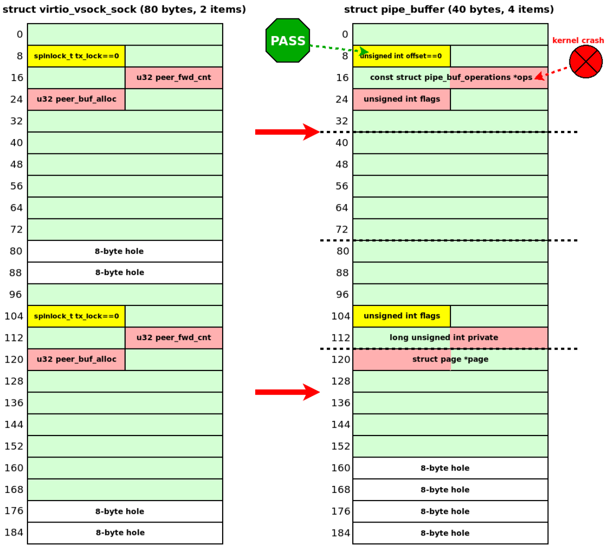
What If?

New hope

What if I allocate 4 pipe_buffers in kmalloc-192?

Target for UAF Write: Four pipe_buffers

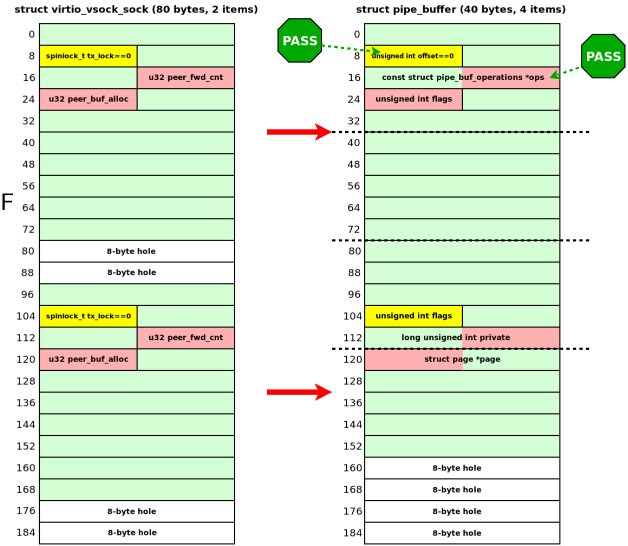
- Oh no, `pipe_buffer.ops` is corrupted by 4 zero bytes!



Target for UAF Write: Four pipe_buffers

- Oh no, `pipe_buffer.ops` is corrupted by 4 zero bytes!
- The kernel crashes if I read from the pipe
- **Idea**: I discarded the first `pipe_buffer` **before** UAF
- In that case the bad `pipe_buffer.ops` isn't used!
- How to do it without changing `offset`:

```
splice(pipe_fds[i][0], NULL,
  temp_pipe_fd[1], NULL, 1, 0);
read(temp_pipe_fd[0],
  pipe_data_to_read, 1);
```

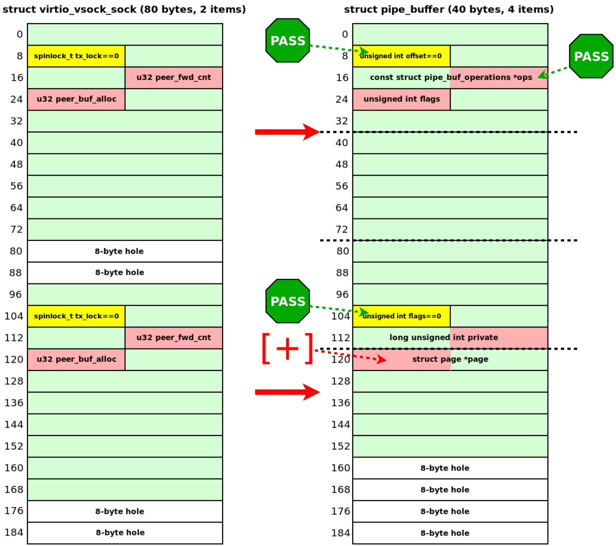


Target for UAF Write: Four pipe_buffers

- Made flags of pipe_buffer #3 zero by using splice() from file

```
splice(temp_file_fd, &file_offset,
  pipe_fds[i][1], NULL, 1, 0);
```

- [+] Corrupted pipe_buffer.page! **YES!**
- kernel-hack-drill helped to develop it



Last Revenge From Physical KASLR

- We don't know where the kernel text is inside `vmemmap`
- We can't point `pipe_buffer.page` to kernel code 🙁
- Let's shoot to the leaked `struct cred` in the kernel heap
- I can calculate the offset of `struct page` poniting to `cred`:

```
#define STRUCT_PAGE_SZ 64lu
#define PAGE_ADDR_OFFSET(addr) (((addr & 0xfffffffflu) >> 12) * STRUCT_PAGE_SZ)
uaf_val = PAGE_ADDR_OFFSET(cred_addr);
```

- Don't need to know the `vmemmap_base`!
  - **[!] I overwrite only 4 lower bytes of** `pipe_buffer.page`
- Randomized `vmemmap_base` address has only 2 random bits in lower bytes

Bruteforce 2 Bits

- In case of fail reading from pipe simply returns "Bad address"
- In case of success reading from pipe gives `struct cred` contents

- Finally, I write zero pipe, overwrite `euid` and `egid`, and I AM ROOT

Demo Time

Conclusion

- Bug collision is painful
- But finishing the research anyway is rewarding
- Try my open source project
  github.com/a13xp0p0v/kernel-hack-drill
- **kernel-hack-drill** is a useful testing environment
  for Linux kernel security researchers
- Contributors are always welcome!

Thanks 감사합니다

Enjoy the conference!

Contacts:

a13xp0p0v

a13xp0p0v@tuta.io

Blog: a13xp0p0v.github.io

Channel: t.me/linkersec