Exploiting null-derefs

...

Doing the impossible in the Linux kernel

(Very very slowly)

Google Project Zero

Seth Jenkins

Agenda

Google

01

whoami

Seth Jenkins

- Information Security Engineer at Google Project Zero
- Primarily Linux and Android kernel research
- Focusing on LPE’s (get to root without a password!)
- Love to turn bad bugs into good exploits

02

The Bug

- The edge-case of the empty process
- A process usually has VMAs (virtual memory areas) that describe the virtual memory used by the process
- …but VMAs are not required by Linux for a “valid” process

```
root@syzkaller:~# cd /proc/1250
root@syzkaller:/proc/1250# ls
arch_status  cgroup      coredump_filter     environ  gid_map             loginuid   mountinfo   ns          oom_score_adj  projid_map  sessionid     stack   syscall          timerslack_ns
attr         clear_refs  cpu_resctrl_groups  exe      io                  map_files  mounts      numa_maps   pagemap        root        setgroups     stat    task             uid_map
autogroup    cmdline     cpuset              fd       ksm_merging_pages   maps       mountstats  oom_adj     patch_state    sched       smaps         statm   timens_offsets   wchan
auxv         comm        cwd                 fdinfo   limits              mem        net         oom_score   personality    schedstat   smaps_rollup  status  timers
root@syzkaller:/proc/1250# cat maps
root@syzkaller:/proc/1250# cat numa_maps
root@syzkaller:/proc/1250# cat smaps
root@syzkaller:/proc/1250# cat smaps_rollup
Killed
root@syzkaller:/proc/1250#
```

Kernel Oops

```
[  236.561299] BUG: kernel NULL pointer dereference, address: 0000000000000000
[  236.562503] #PF: supervisor read access in kernel mode
[  236.563499] #PF: error_code(0x0000) - not-present page
[  236.564379] PGD 0 P4D 0
[  236.564865] Oops: 0000 [#1] PREEMPT SMP NOPTI
[  236.565685] CPU: 6 PID: 1270 Comm: cat Tainted: G        W          6.0.0+ #16
[  236.567001] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.16.2-debian-1.16.2-1 04/01/2014
[  236.568297] RIP: 0010:show_smaps_rollup+0x1fb/0x310
[  236.568731] Code: c0 74 1c 48 39 28 73 90 48 3b 68 08 0f 82 c1 00 00 00 4d 8b 6d 10 4d 85 ed 0f 85 79 ff f
 ef
[  236.570332] RSP: 0018:ffffc90003a03c80 EFLAGS: 00010246
[  236.570809] RAX: ffff888104490440 RBX: ffff888104490440 RCX: 0000000000000000
[  236.571471] RDX: 0000000000000000 RSI: 0000000000000100 RDI: 00000000ffffffff
[  236.572158] RBP: 0000000000000000 R08: ffffc90003a03d30 R09: 0000000000001000
[  236.572824] R10: 0000000000000000 R11: 0000000000000000 R12: ffff88812545f2c0
[  236.573507] R13: 0000000000000000 R14: 0000000000000000 R15: ffff8881044904b8
[  236.574170] FS:  00007fab98d0e480(0000) GS:ffff88842fb80000(0000) knlGS:0000000000000000
[  236.574910] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  236.575452] CR2: 0000000000000000 CR3: 0000000125fe2000 CR4: 0000000000350ee0
[  236.576096] Call Trace:
[  236.576317]  <TASK>
[  236.576513]  seq_read_iter+0x122/0x450
[  236.576832]  seq_read+0xa3/0xd0
[  236.577103]  vfs_read+0xa1/0x280
[  236.577403]  ? handle_mm_fault+0xae/0x290
[  236.577768]  ksys_read+0x63/0xe0
[  236.578066]  do_syscall_64+0x3a/0x90
[  236.578403]  entry_SYSCALL_64_after_hwframe+0x63/0xcd
[  236.578892] RIP: 0033:0x7fab986db910
```

What happened?

`mm->mmap` points to the first vma for the associated task.
If a process has no vma's...

`priv->mm->mmap == NULL`

Null-deref sends kernel to oops path which calls `make_task_dead`, ending the task
Another boring bug...

```
static int show_smaps_rollup(struct seq_file *m, void *v)
{
        ...
        priv->task = get_proc_task(priv->inode);
        if (!priv->task)
                return -ESRCH;
        mm = priv->mm;
        if (!mm || !mmget_not_zero(mm)) {
                ret = -ESRCH;
                goto out_put_task;
        }
        ...
        ret = mmap_read_lock_killable(mm);
        ...
        for (vma = priv->mm->mmap; vma;) {
                ...
                vma = vma->vm_next;
        }
        show_vma_header_prefix(m, priv->mm->mmap->vm_start,
                               last_vma_end, 0, 0, 0, 0);
        ...
        mmap_read_unlock(mm);
out_put_mm:
        mmput(mm);
out_put_task:
        put_task_struct(priv->task);
        ...
}
```

Submit and fix...

author	Seth Jenkins <sethjenkins@google.com> 2022-10-27 11:36:52 -0400
committer	Greg Kroah-Hartman <gregkh@linuxfoundation.org> 2022-10-29 10:12:58 +0200
commit	33fc9e26b7cb39f0d4219c875a2451802249c225 (patch)
tree	6cecd047ea52b6e8621ec87dcad7c23471132b5d /fs/proc/task_mmu.c
parent	b9d8cbe90a0f27f2ec4f6f32e7faf86282eb4d7d (diff)
download	linux-33fc9e26b7cb39f0d4219c875a2451802249c225.tar.gz

mm: /proc/pid/smaps_rollup: fix no vma's null-deref

```
Commit 258f669e7e88 ("mm: /proc/pid/smaps_rollup: convert to single value
seq_file") introduced a null-deref if there are no vma's in the task in
show_smaps_rollup.

Fixes: 258f669e7e88 ("mm: /proc/pid/smaps_rollup: convert to single value seq_file")
Signed-off-by: Seth Jenkins <sethjenkins@google.com>
Reviewed-by: Alexey Dobriyan <adobriyan@gmail.com>
Tested-by: Alexey Dobriyan <adobriyan@gmail.com>
Signed-off-by: Greg Kroah-Hartman <gregkh@linuxfoundation.org>
```

Diffstat (limited to 'fs/proc/task_mmu.c')

-rw-r--r-- fs/proc/task_mmu.c 2

1 files changed, 1 insertions, 1 deletions

```
diff --git a/fs/proc/task_mmu.c b/fs/proc/task_mmu.c
index d9c07eecd7872..c3b76746cce85 100644
--- a/fs/proc/task_mmu.c
+++ b/fs/proc/task_mmu.c
@@ -951,7 +951,7 @@ static int show_smaps_rollup(struct seq_file *m, void *v)
                vma = vma->vm_next;
        }

-       show_vma_header_prefix(m, priv->mm->mmap->vm_start,
+       show_vma_header_prefix(m, priv->mm->mmap ? priv->mm->mmap->vm_start : 0,
                               last_vma_end, 0, 0, 0, 0);
        seq_pad(m, ' ');
        seq_puts(m, "[rollup]\n");
```

Oh no, what have I done?!?

Isn’t it great that kernel code can’t be unexpectedly aborted like userland can?

If a task could get a “signal” in the middle of a syscall and the syscall code suddenly ends without cleanup, that’d lead to so many bugs!

A horrible realization...

oops + `make_task_dead` does exactly this...

```
static int show_smaps_rollup(struct seq_file *m, void *v)
{
	...
	priv->task = get_proc_task(priv->inode);
	if (!priv->task)
		return -ESRCH;
	mm = priv->mm;
	if (!mm || !mmget_not_zero(mm)) {
		ret = -ESRCH;
		goto out_put_task;
	}
	...
	ret = mmap_read_lock_killable(mm);
	...
	for (vma = priv->mm->mmap; vma;) {
		...
		vma = vma->vm_next;
	}
	show_vma_header_prefix(m, priv->mm->mmap->vm_start,
				last_vma_end, 0, 0, 0, 0);
	...
	mmap_read_unlock(mm);
out_put_mm:
	mmput(mm);
out_put_task:
	put_task_struct(priv->task);
	...
}
```

(not shown) seq file fdget and mutex

Task refcount increment

mm refcount increment

mm read lock

dereference

mm read unlock

mm refcount decrement

Task refcount decrement

```
static int show_smaps_rollup(struct seq_file *m, void *v)
{
	...
	priv->task = get_proc_task(priv->inode);
	if (!priv->task)
		return -ESRCH;
	mm = priv->mm;
	if (!mm || !mmget_not_zero(mm)) {
		ret = -ESRCH;
		goto out_put_task;
	}
	...
	ret = mmap_read_lock_killable(mm);
	...
	for (vma = priv->mm->mmap; vma;) {
		...
		vma = vma->vm_next;
	}
	show_vma_header_prefix(m, priv->mm->mmap->vm_start,
			       last_vma_end, 0, 0, 0, 0);
	...
	mmap_read_unlock(mm);
out_put_mm:
	mmput(mm);
out_put_task:
	put_task_struct(priv->task);
	...
}
```

(not shown) seq file fdget and mutex

Task refcount increment

mm refcount increment

mm read lock

Null dereference

Goto `make_task_dead`

Everything else skipped!!

03

Refcount Attacks

Spurious refcount increments and decrements can be exploitable issues

Over-decrement -> freeing object while references still exist a.k.a UAF

Over-increment -> repeated over-increment can cause refcount overflow, after which refcount increment+decrement can free object also UAF (except for saturating refcounts)


Use-after-frees everywhere.

Oops effects:

1. The `struct file` associated `seq_file’s` mutex will forever be locked Useless
2. The associated `struct file` will have a reference permanently held if `fdget` took a reference Useless; future refcount leaks require locking seq_file mutex
3. The `task struct` associated with the `smaps_rollup` file (aka the no-vma task) will have a refcount leaked Useless; Uses saturating refcount
4. The `mm_struct’s` `mm_users` refcount associated with the no-vma task will be leaked Useful?
5. The `mm_struct’s` mmap lock will be permanently readlocked Useless

Overflowing the refcount

Trigger a Linux kernel oops enough times to overflow the refcount; ~2^{32} times!

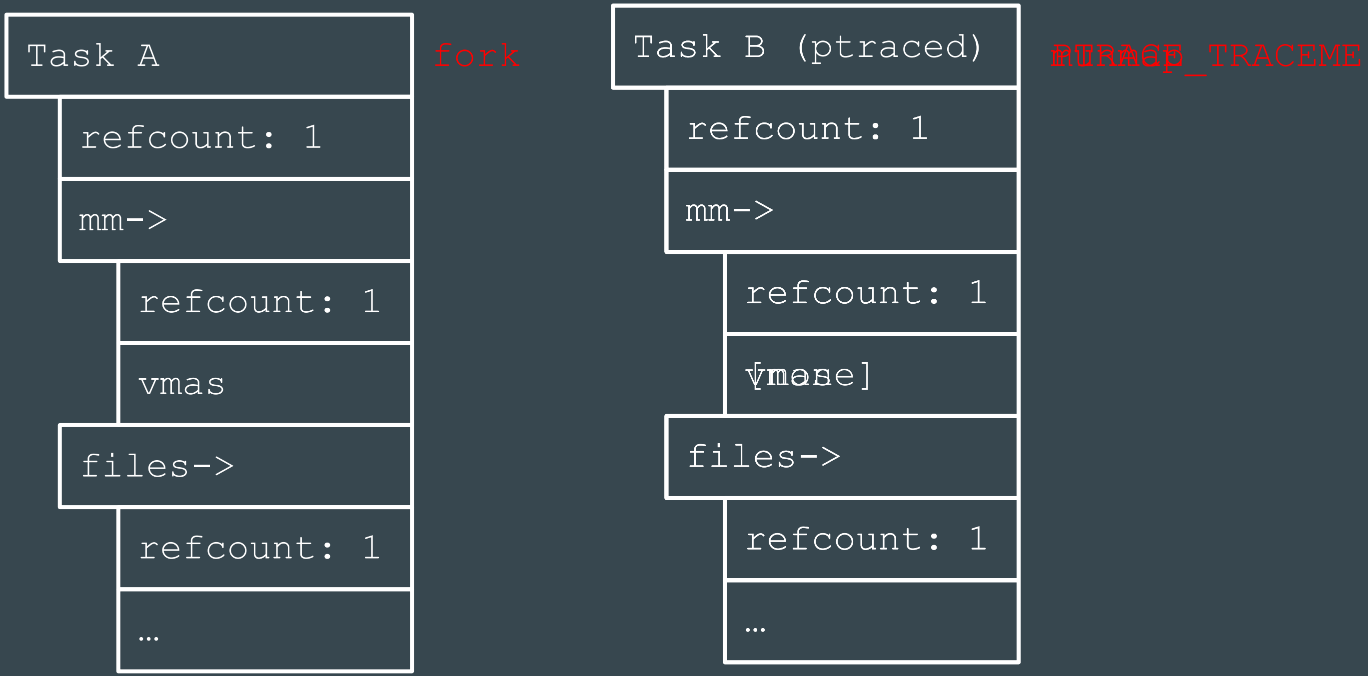
- We must be able to trigger the oops without causing memory leaks
  - We must destroy every opened `smaps_rollup` struct file
    - `smaps_rollup` must be `read` from a single-threaded process so `fdget` doesn't take a refcount
    - `fd` must be closed after oops (the associated `seq_file` mutex is permanently locked). This happens automatically when make_task_dead tears down the fdtable
  - We leak the no-vma task refcount which is a "memory leak"
    - But the "leak" only happens on the first `smaps_rollup` read. The future refcount leaks are on the same "memory leaked" task
  - We leak the `mm` refcount which is a "memory leak"
    - But it's what we're trying to overflow anyway

Timing

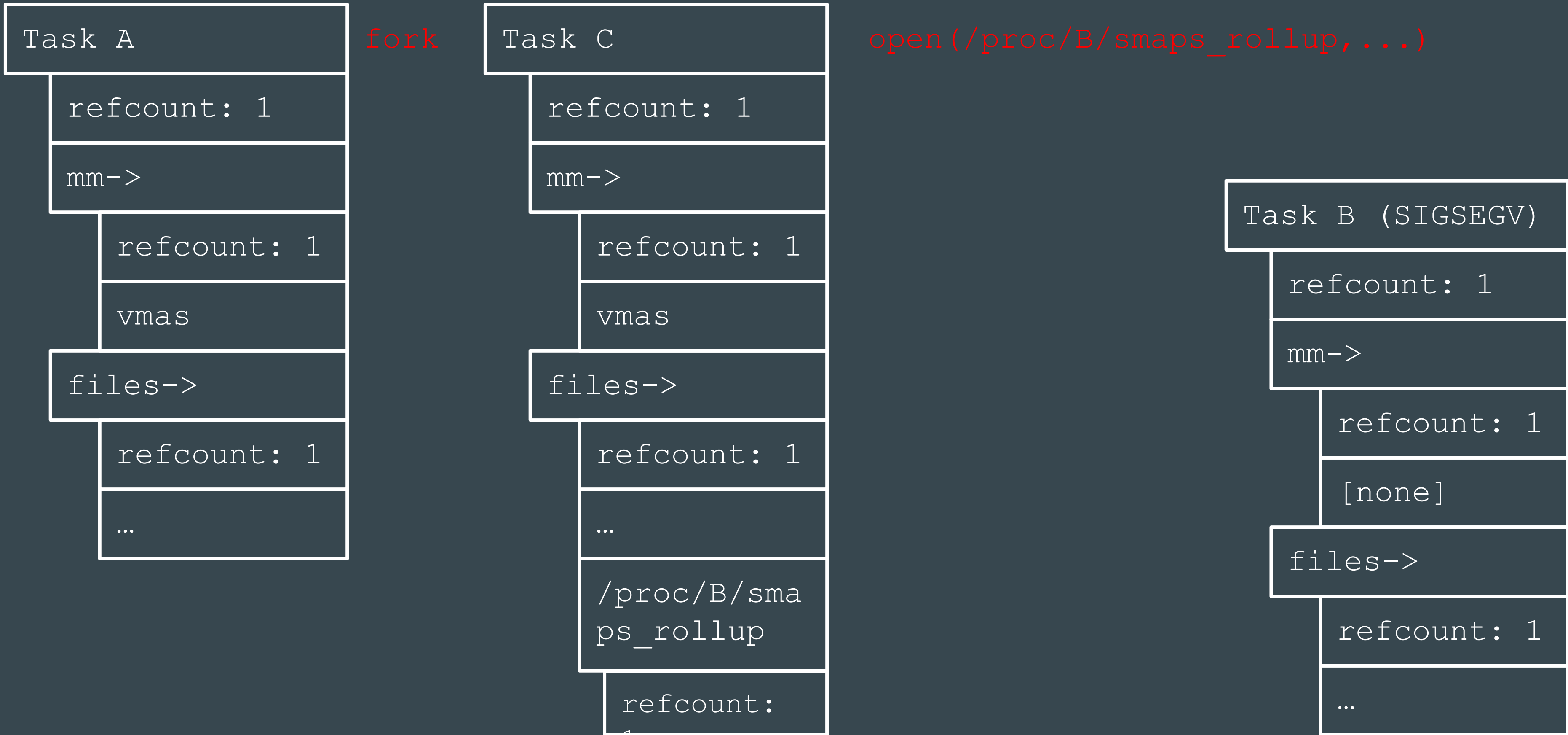
We must be able to trigger the oops **quickly** (2^{32} times is a lot of times)

	1 oops	**2^{32} oops's (multithreaded)**
Serial console (Qemu)	~45 ms	2+ years
GUI (vanilla Debian)	~.5 ms	~8 days

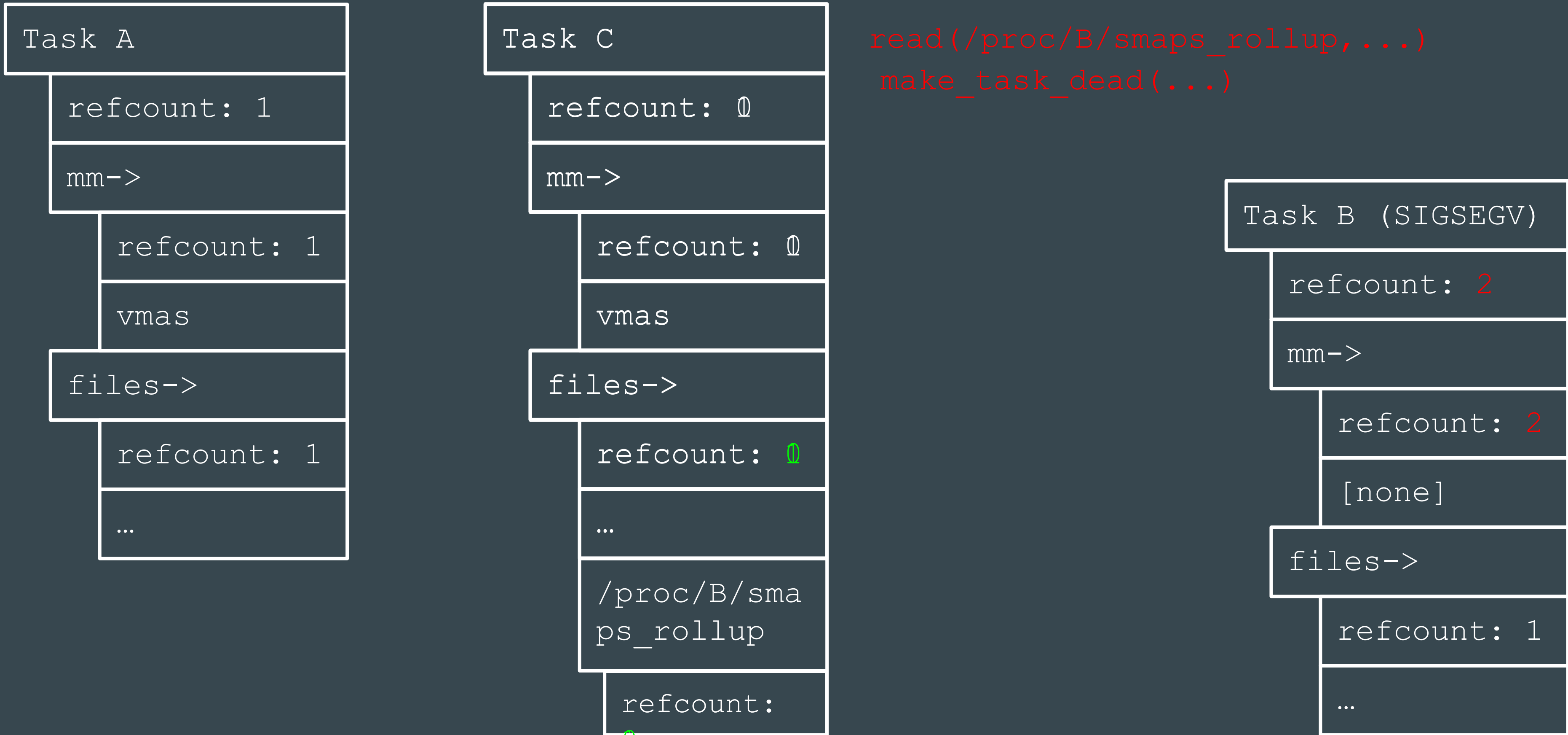
Overflowing the refcount



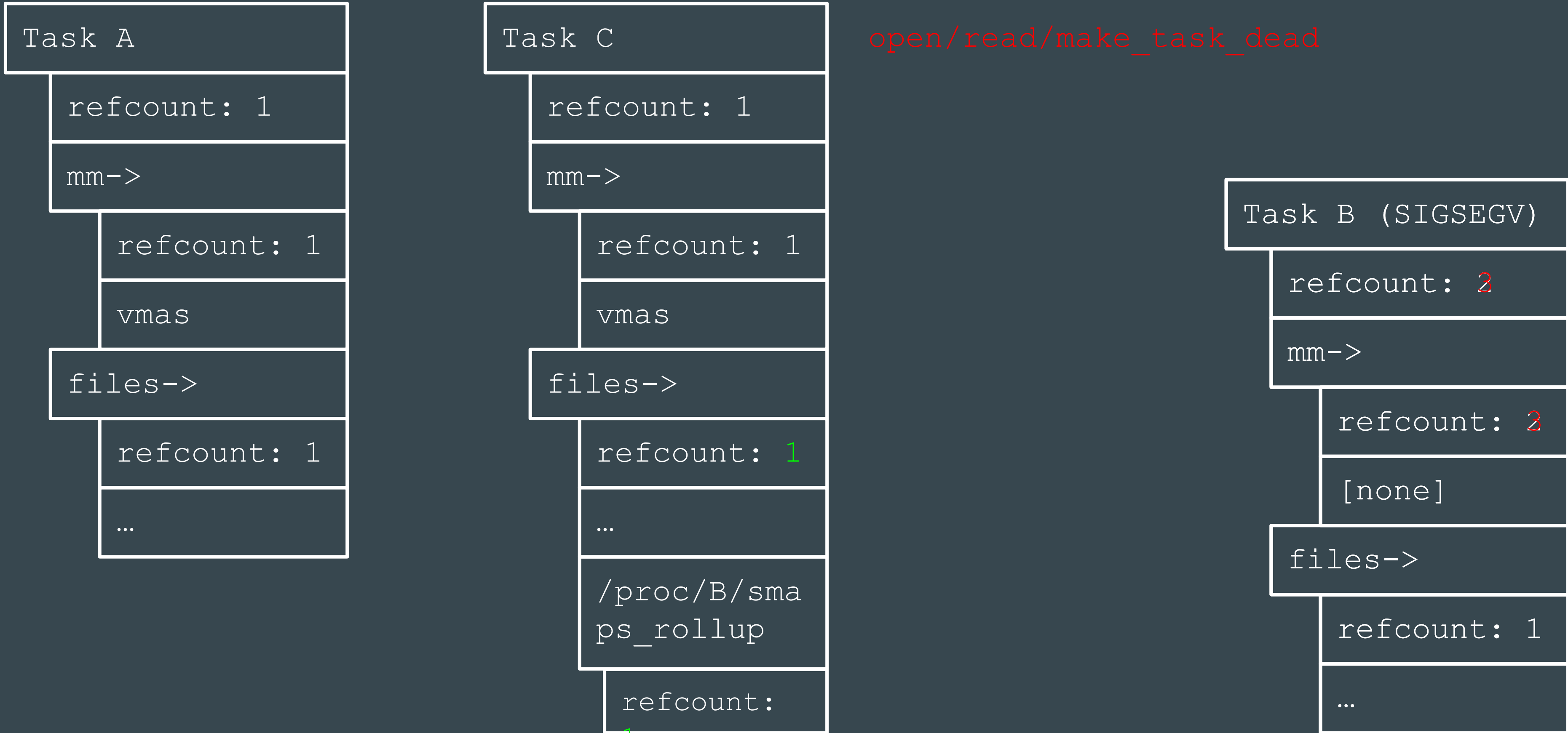
Overflowing the refcount



Overflowing the refcount



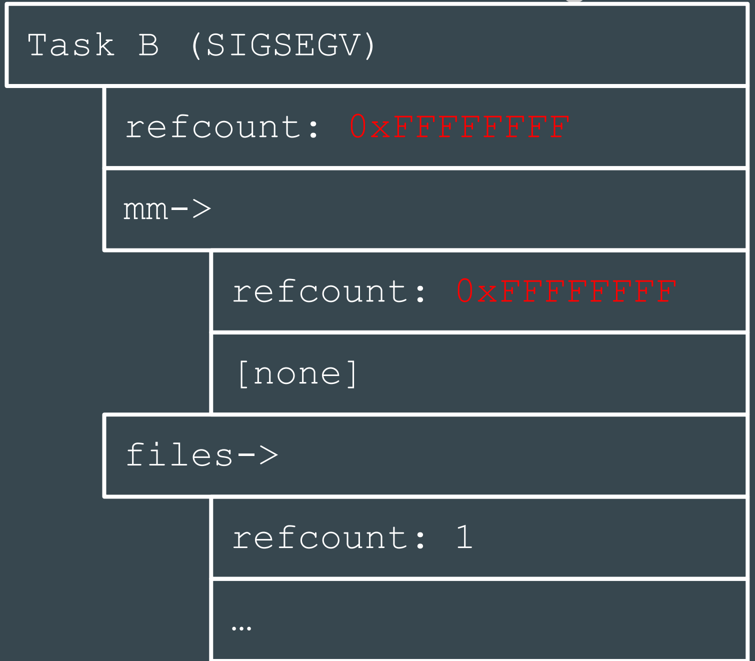
Overflowing the refcount



Overflowing the refcount

One small hiccup…

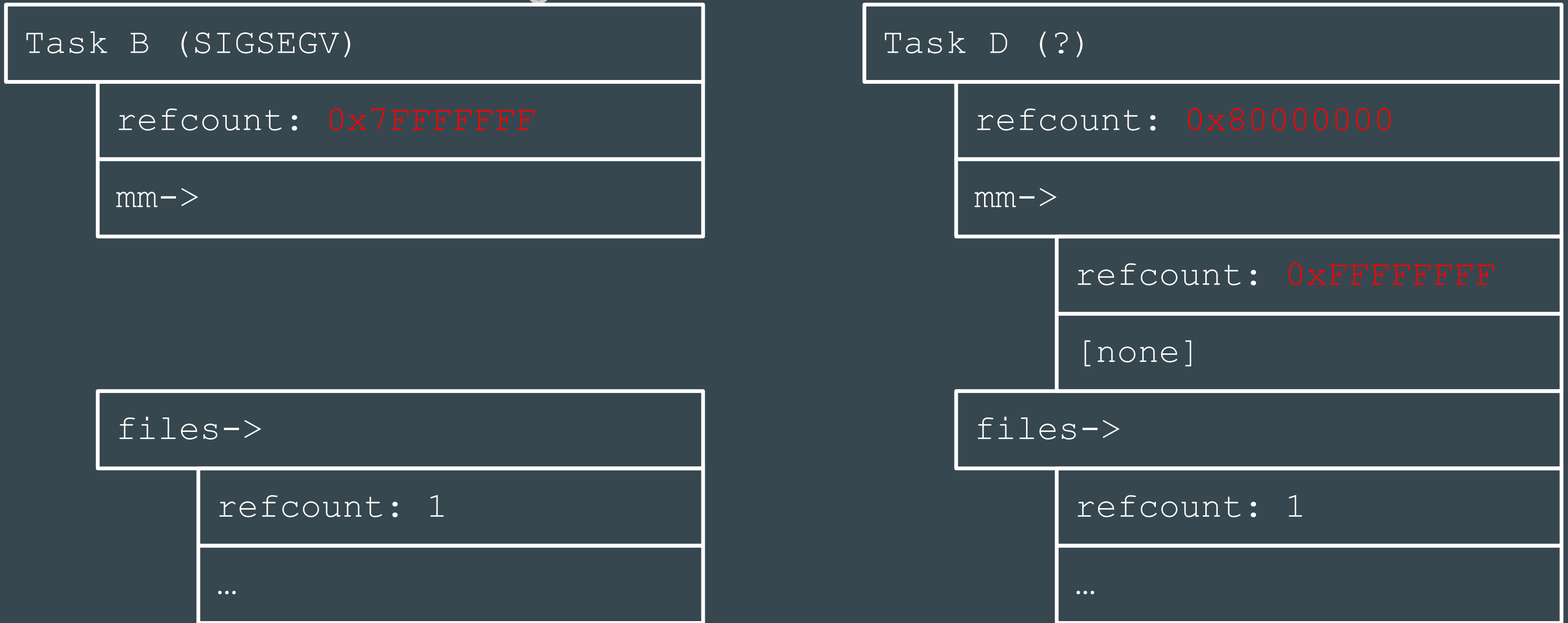
Task refcount is saturating…does that matter? We can avoid it…



Overflowing the refcount

One small hiccup...

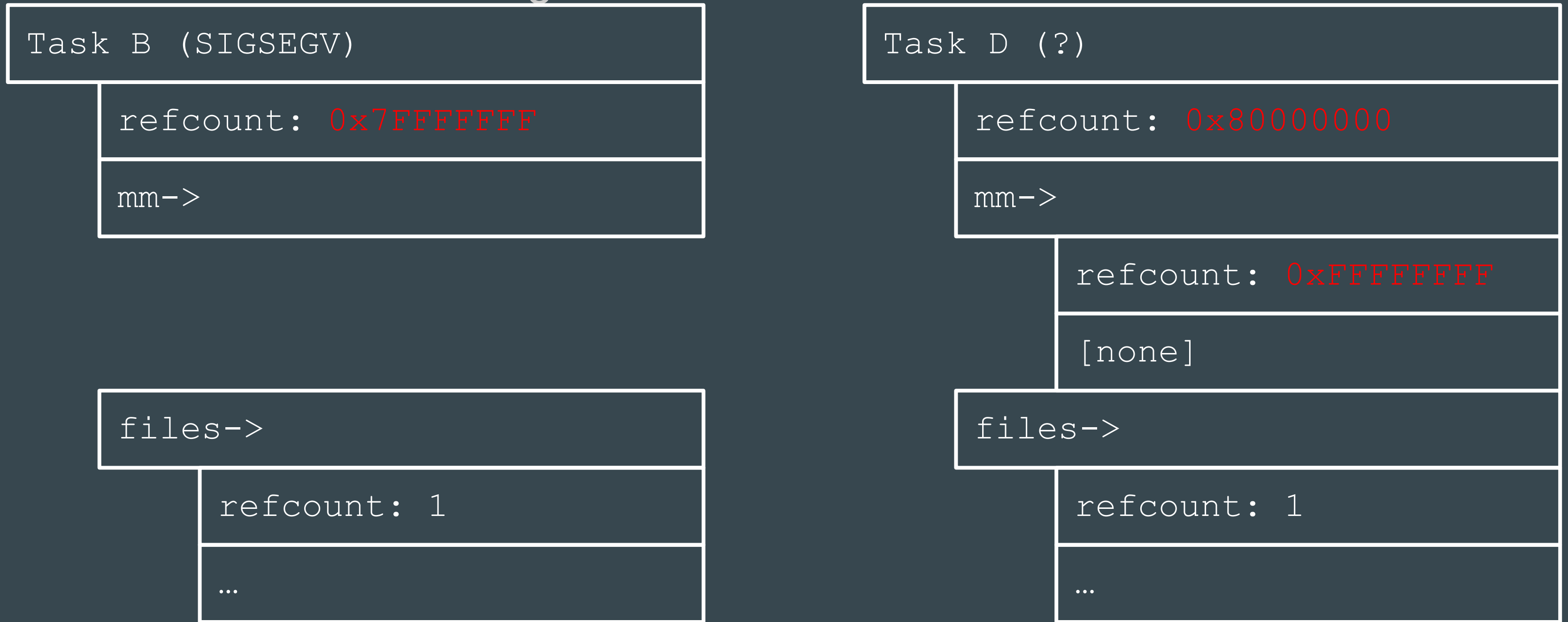
Task refcount is saturating...does that matter? We can avoid it...



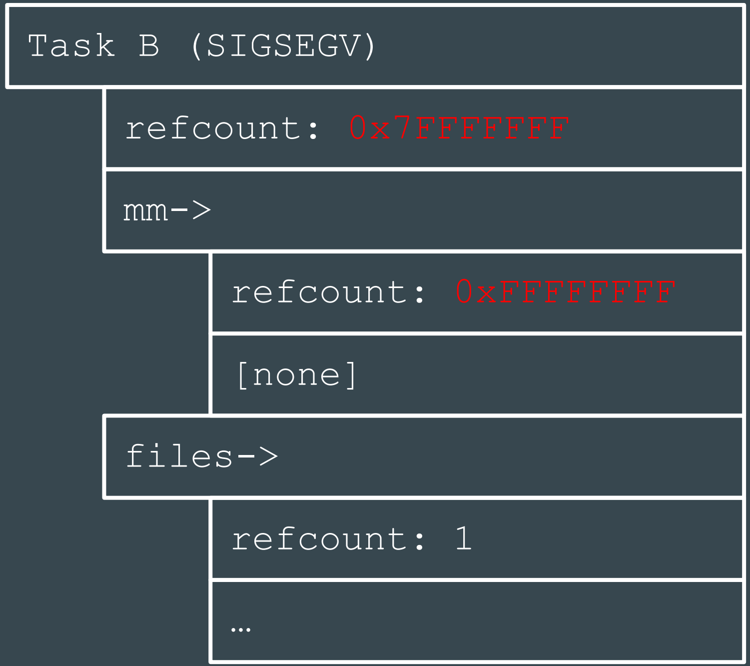
Overflowing the refcount

One small hiccup...

Task refcount is saturating...does that matter? We can avoid it...



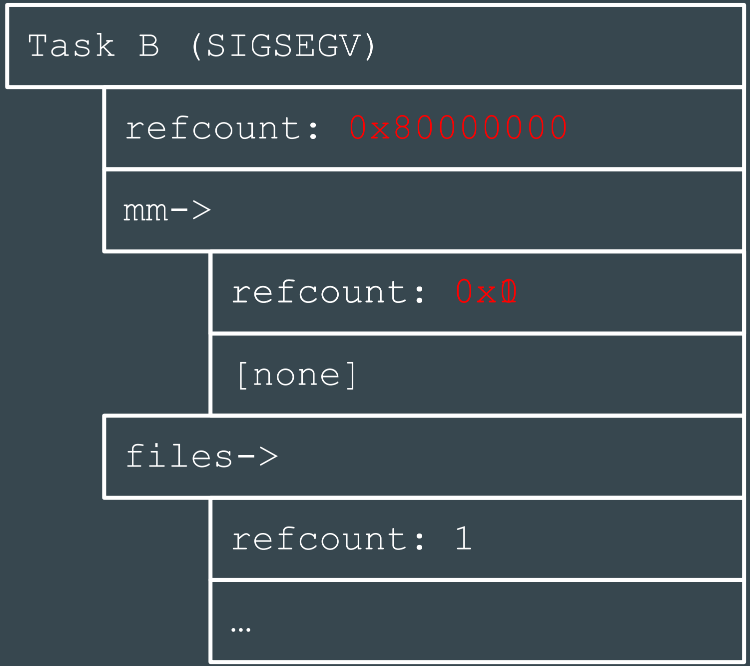
Overflowing the refcount



Overflowing the refcount

In order to free the `mm`, we `mmget` and `mmput`

E.g. Task A `open`'s and `close`'s `/proc/B/mem`



04

Getting to root

On Linux kernel after `64591e8605 ("mm: protect free_pgtables with mmap_lock write lock in exit_mmap"),exit_mmap` takes the mmap write lock. (5.17+)

Still exploitable, but need to take advantage of unintended concurrence of `__mmput` calls to cause a double free

...it's a pain.

Thankfully, on the version of Ubuntu I was looking at, the mmap lock is *not* taken in write mode.

`mm` gets freed all the way, yay!

What is an mm?

The mm tracks (among other things) the virtual memory layout of the process

- Virtual Memory Areas
- Location of .text/.data etc.
- Mutexing

mm’s also come from their own kernel slab cache

UAF Exploit strategies

- Cross-cache?
- Create arb-rw?
- Other classic UAF exploit?
- Easiest strategy - replace the mm with an mm for a more privileged process

    Attacker task will share an mm with a privileged process

    What attacker task? The task which previously had no vma’s

➔ More generic exploit strategy - replace a **boring** version of an object with a **highly privileged** version of that same type of object

Replacement strategy

Current status:

- The mm was just freed
- The attacker task is frozen in segfault tracing stop

What next?

- Reclaim mm by execve'ing passwd from a lot of processes, spraying new privileged mm structs.
  - Since the mm was allocated a long time (8 days!) ago, the slab containing the mm probably isn't going to be the per-cpu active slab
- Allocate enough processes to drain the percpu slab freelist, and allocate from the percpu partial lists.

A problematic process

Now I have an attacker task that:

- Previously had no vma's
- Is segfaulted
- Has zero understanding of the virtual memory layout it's in

...what do I do with such a task?

Open `/proc/pid/mem` from another attacker task!

ProcFS crash course

Each process on Linux has a directory with files that describe and allow interactions with the respective process.

`show_smaps_rollup` is one of these files.

`mem` is another per-process ProcFS file that when opened represents the virtual memory of the process.

Virtual memory can be “selected” with `lseek`, then read/written using `read(2)` and `write(2)`

Opening the `mem` of a child process from the parent is allowed by `ptrace` Yama as long as uid’s/gids are the same, SELinux allows `ptrace`, and...

A problematic process

...if the process is dumpable.

The process will not be dumpable, since it is a SUID process's mm.

However, a process can always open its own mem

We don't have memory rw, but we can read/write the task's registers (thanks ptrace)

```
struct mm_struct *mm_access(struct task_struct *task, unsigned int mode)
{
        ...
        if (mm && mm != current->mm && !ptrace_may_access(task, mode))
        ...
}
...
static int __ptrace_may_access(struct task_struct *task, unsigned int mode)
{
        ...
        tcred = __task_cred(task);
        if (uid_eq(caller_uid, tcred->euid) &&
          uid_eq(caller_uid, tcred->suid) &&
          uid_eq(caller_uid, tcred->uid)  &&
          gid_eq(caller_gid, tcred->egid) &&
          gid_eq(caller_gid, tcred->sgid) &&
          gid_eq(caller_gid, tcred->gid))
                goto ok;
        if (ptrace_has_cap(tcred->user_ns, mode))
                goto ok;
        ...
ok:
        ...
        if (mm &&
          ((get_dumpable(mm) != SUID_DUMP_USER) &&
           !ptrace_has_cap(mm->user_ns, mode)))
          return -EPERM;

        return security_ptrace_access_check(task, mode);
}
```

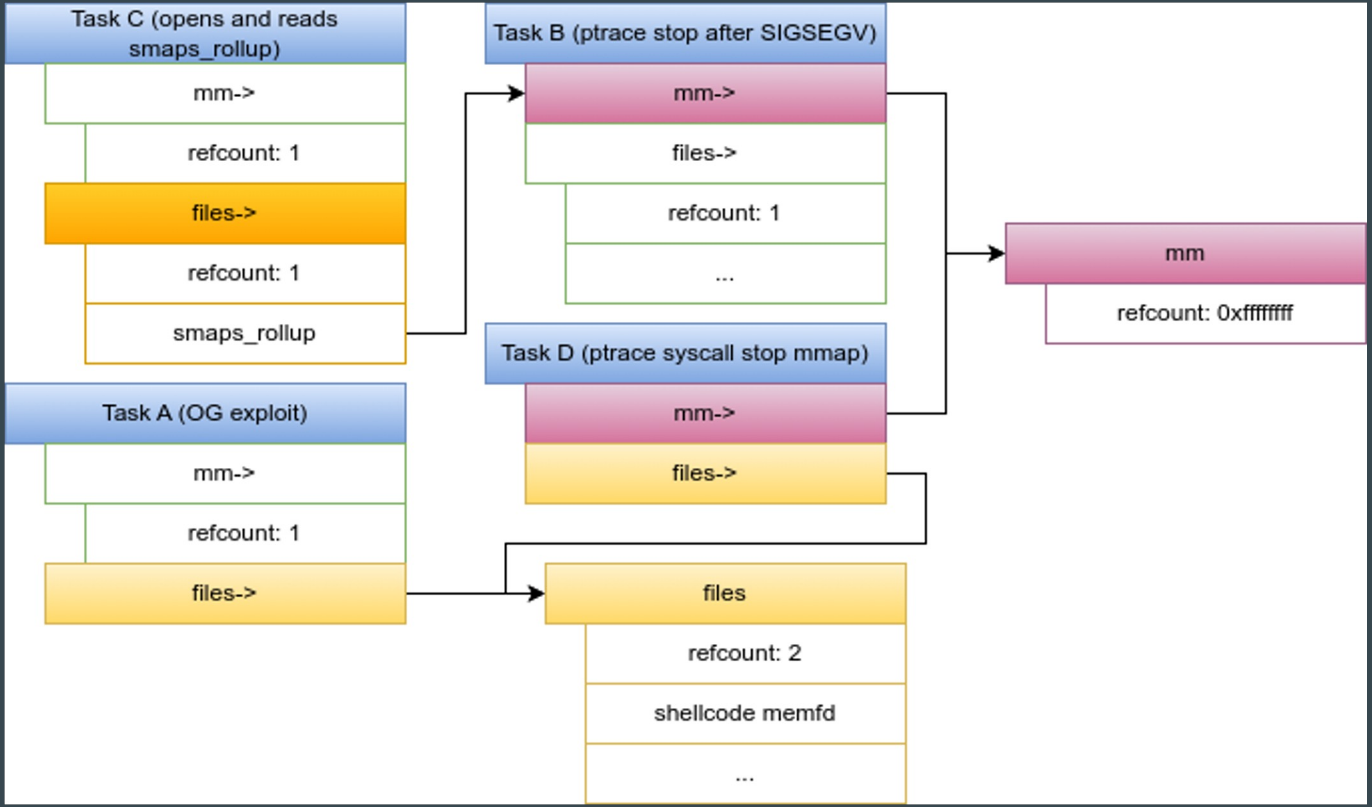
A problematic process

This task is in a nigh unrunnable state...

So let's use another task (in practice just use task D we made to split task refcounts):

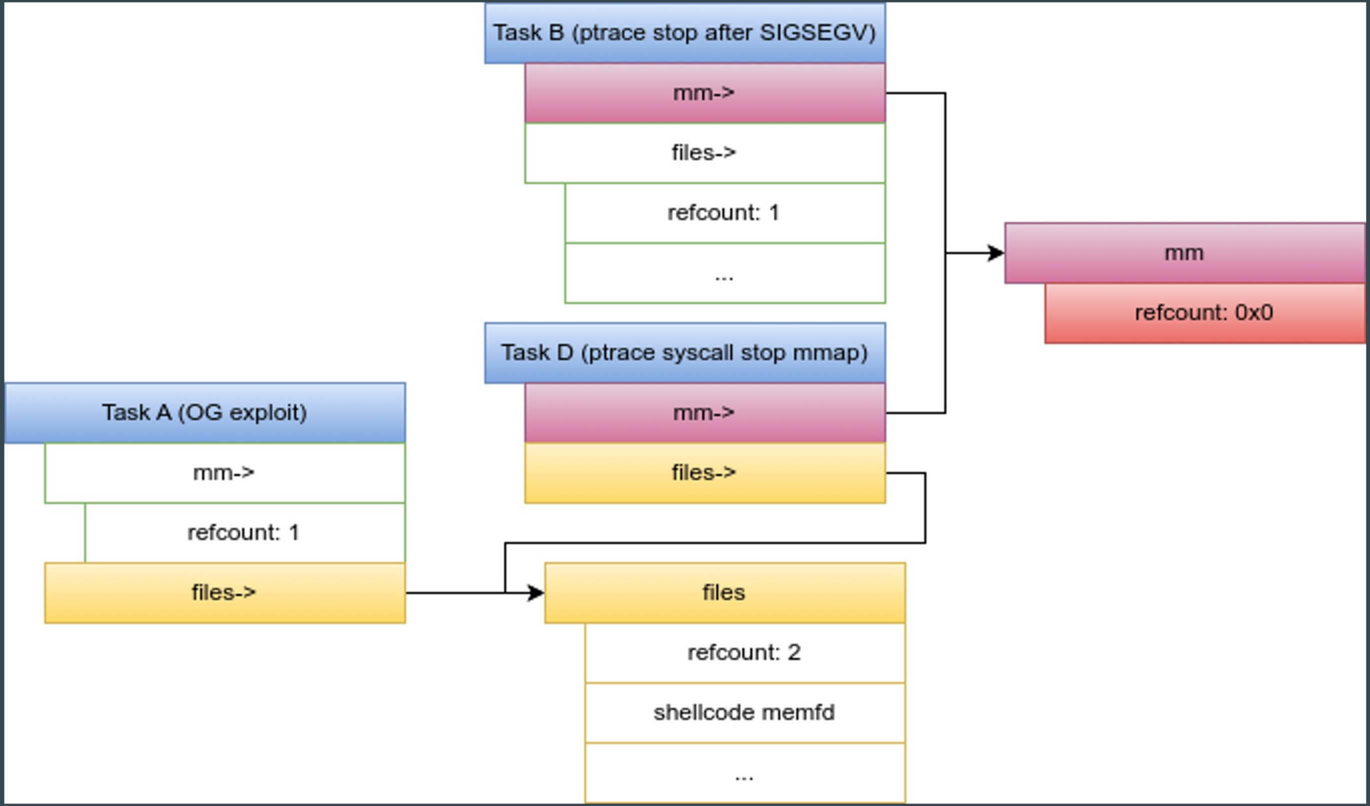
- Shares the `mm` with the other process which `munmap`'d
- But instead of being SEGFAULT'd after `munmap`...
- We ptrace syscall stop on syscall entry to an mmap of shellcode
- Wait for `mm` free and reclaim
- Release the process into the `mmap` syscall, mapping shellcode into the privileged/attacker process's virtual memory
- This process can open `/proc/self/mem` and share it with the parent process

A recap



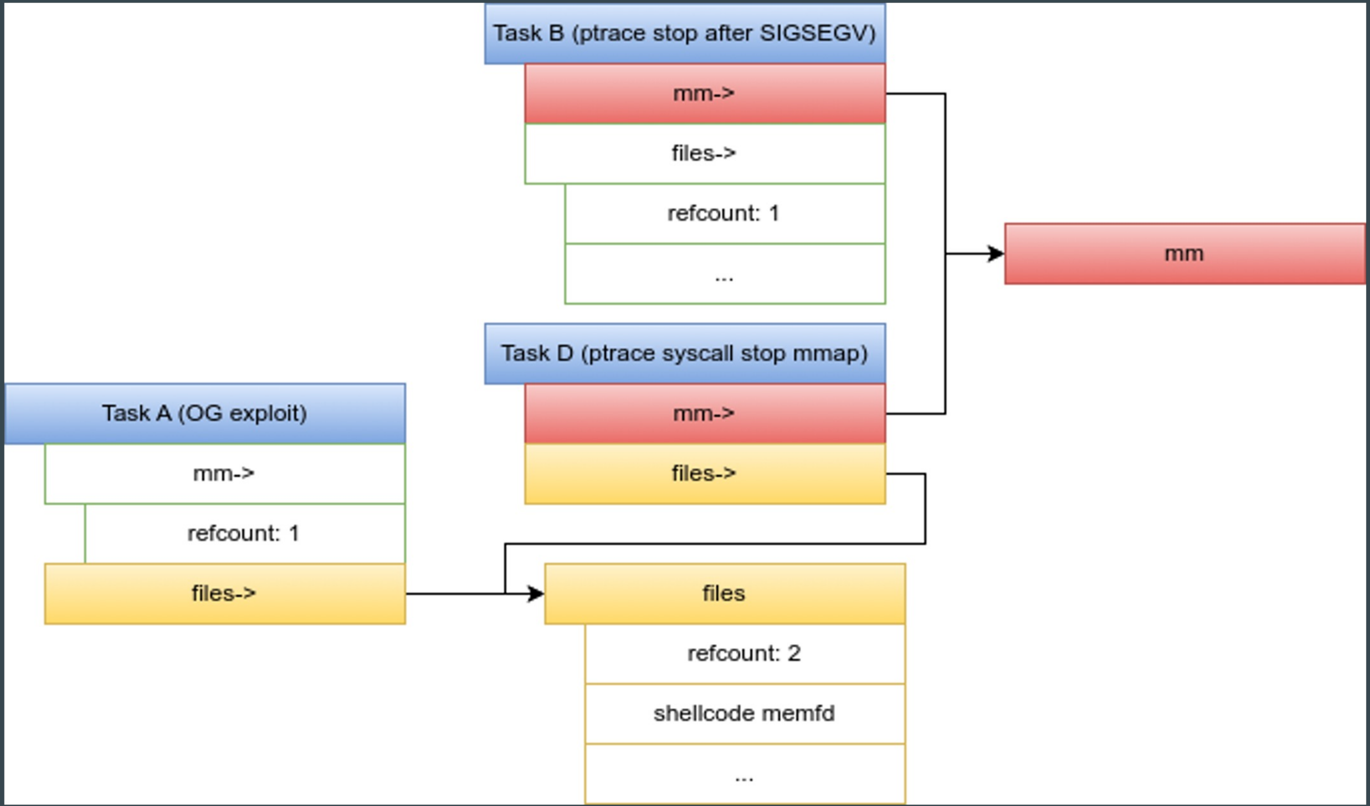
Task C (opens and reads smaps_rollup)
mm->
refcount: 1
files->
refcount: 1
smaps_rollup
Task A (OG exploit)
mm->
refcount: 1
files->
Task B (ptrace stop after SIGSEGV)
mm->
files->
refcount: 1
...
Task D (ptrace syscall stop mmap)
mm->
files->
files
refcount: 2
shellcode memfd
...
mm
refcount: 0xffffffff

A recap



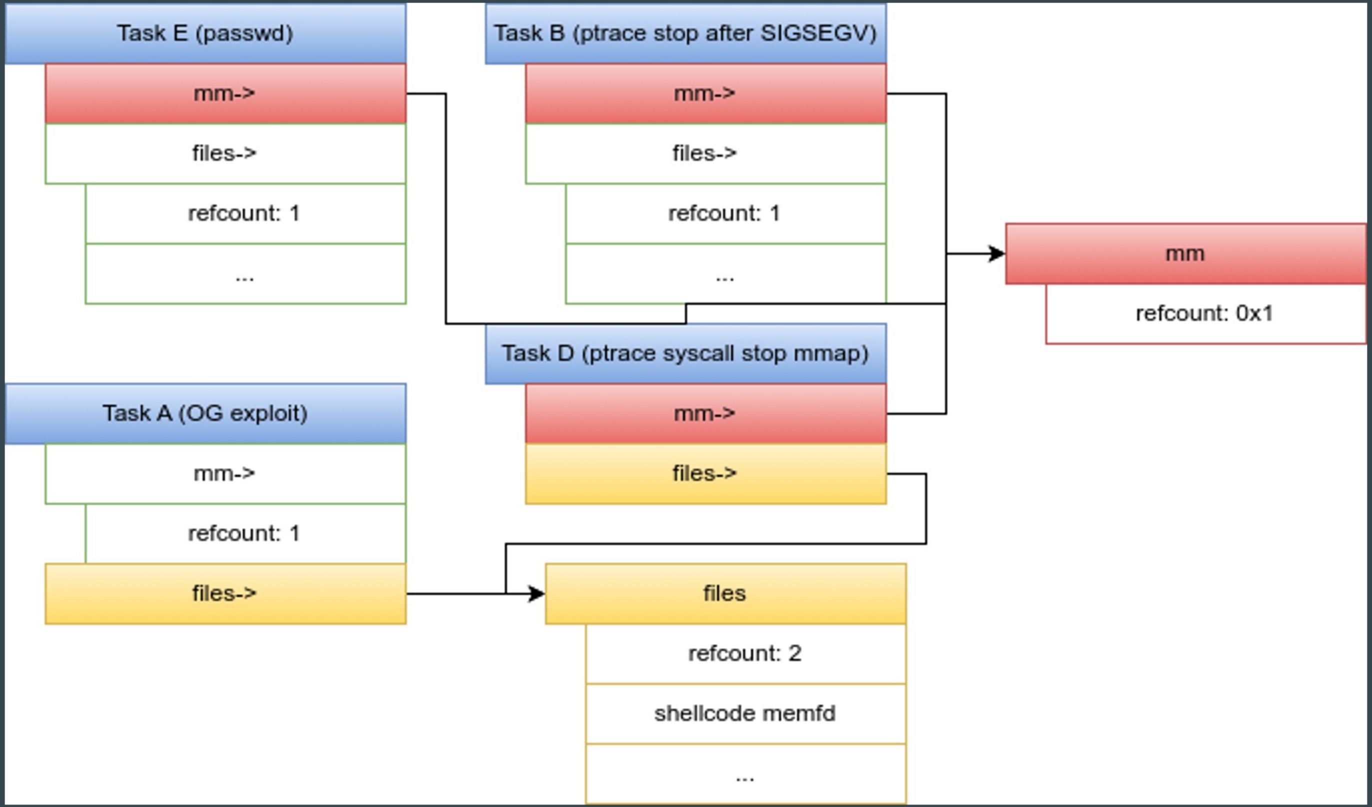
Task B (ptrace stop after SIGSEGV)
mm->
files->
refcount: 1
...
mm
refcount: 0x0
Task D (ptrace syscall stop mmap)
mm->
files->
Task A (OG exploit)
mm->
refcount: 1
files->
files
refcount: 2
shellcode memfd
...

A recap

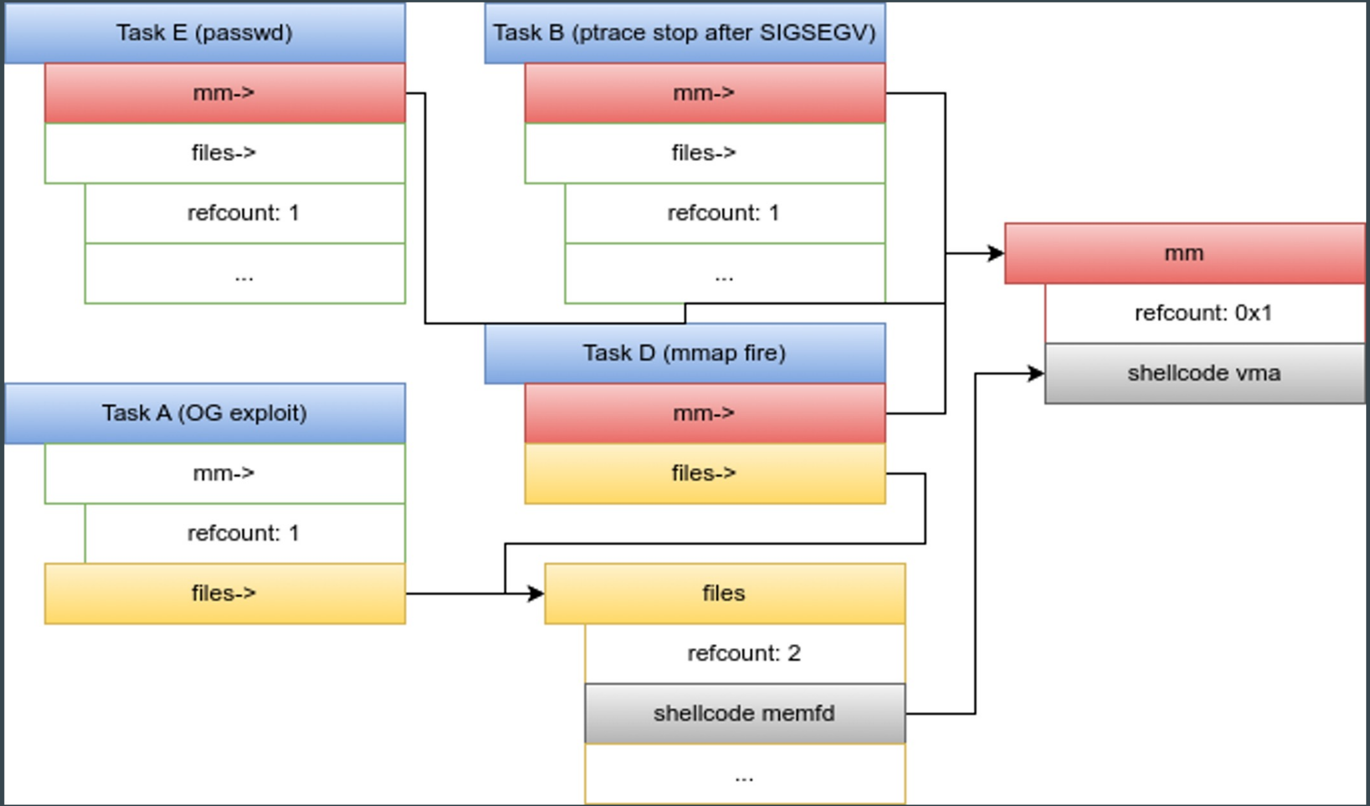


Task B (ptrace stop after SIGSEGV)
mm->
files->
refcount: 1
...
mm
Task D (ptrace syscall stop mmap)
mm->
files->
Task A (OG exploit)
mm->
refcount: 1
files->
files
refcount: 2
shellcode memfd
...

A recap

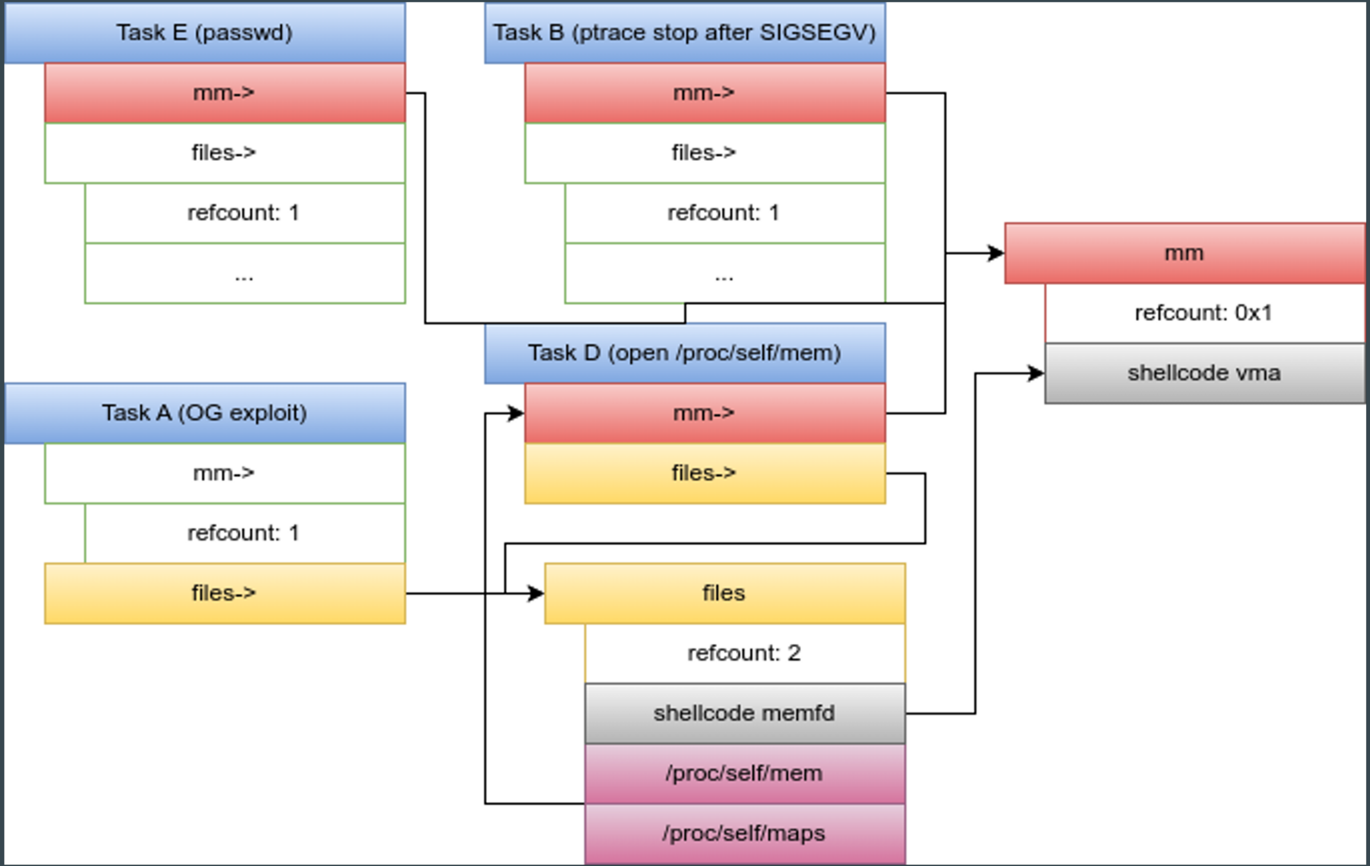

Task E (passwd)
mm->
files->
refcount: 1
...
Task B (ptrace stop after SIGSEGV)
mm->
files->
refcount: 1
...
mm
refcount: 0x1
Task D (ptrace syscall stop mmap)
mm->
files->
Task A (OG exploit)
mm->
refcount: 1
files->
files
refcount: 2
shellcode memfd
...

A recap



Task E (passwd)
mm->
files->
refcount: 1
...
Task B (ptrace stop after SIGSEGV)
mm->
files->
refcount: 1
...
mm
refcount: 0x1
shellcode vma
Task D (mmap fire)
mm->
files->
Task A (OG exploit)
mm->
refcount: 1
files->
files
refcount: 2
shellcode memfd
...

A recap



Task E (passwd)
mm->
files->
refcount: 1
...
Task B (ptrace stop after SIGSEGV)
mm->
files->
refcount: 1
...
mm
refcount: 0x1
shellcode vma
Task D (open /proc/self/mem)
mm->
files->
Task A (OG exploit)
mm->
refcount: 1
files->
files
refcount: 2
shellcode memfd
/proc/self/mem
/proc/self/maps

Final Stage: Getting to root

Now I have read/write to `/proc/E/mem` aka RW of `passwd`'s virtual memory)

No shortage of ways to complete the exploit...

While `passwd` is hung in the prompt:

- `nop` sled the entire binary image
- Append shellcode that `setuid`'s and `execve`'s `/bin/sh`
- Send POSIX signal to `passwd` that causes signal handler to fire, jumping into binary image.

```
nopriv@syzkaller:~$ whoami
nopriv
nopriv@syzkaller:~$ GLIBC_TUNABLES=glibc.pthread.rseq=0 ./poc
Creating shellcode region
Trying to turn off console spam
sh: 1: cannot create /proc/sys/kernel/printk: Permission denied
Waiting for unmap
Issuing traceme
D raising SIGSTOP
C raising SIGSTOP
pid1: 1070
pid2: 1069
child1 iterations: 8
child2 iterations: 8
Waiting for overflow to complete...Performing 8 iterations on child 0 with pid 1071
Performing 8 iterations on child 1 with pid 1072
Overflow complete, freeing mm
sh: 1: cannot create /proc/sys/kernel/printk: Permission denied
Reclaiming mm
Attempting mmap
mmap return value: 0x7f21012a7000
Modifying RIP and refiring DD pid is 1069
Recieved stop signal 5 from D, RIP is 0x7f21012a706c
siginfo data: 5 0 128 (nil)
passwd maps:
556b1b600000-556b1b60c000 r-xp 00000000 08:10 11372                          /usr/bin/passwd
556b1b80c000-556b1b80d000 r--p 0000c000 08:10 11372                          /usr/bin/passwd
556b1b80d000-556b1b80f000 rw-p 0000d000 08:10 11372                          /usr/bin/passwd
556b1b9b8000-556b1b9d9000 rw-p 00000000 00:00 0                              [heap]

7f2101084000-7f2101085000 rw-p 0000d000 08:10 2941                           /lib/x86_64-linux-gnu/libpam.so.0.83.1
7f2101085000-7f21010a8000 r-xp 00000000 08:10 2876                           /lib/x86_64-linux-gnu/ld-2.24.so
7f210129d000-7f21012a3000 rw-p 00000000 00:00 0
7f21012a7000-7f21012a8000 rwxp 00000000 00:01 1025                           /memfd:shellcode (deleted)
7f21012a8000-7f21012a9000 r--p 00023000 08:10 2876                           /lib/x86_64-linux-gnu/ld-2.24.so
7f21012a9000-7f21012aa000 rw-p 00024000 08:10 2876                           /lib/x86_64-linux-gnu/ld-2.24.so
7f21012aa000-7f21012ab000 rw-p 00000000 00:00 0
7ffeddf81000-7ffeddfa2000 rw-p 00000000 00:00 0                              [stack]
7ffeddfbb000-7ffeddfbf000 r--p 00000000 00:00 0                              [vvar]
7ffeddfbf000-7ffeddfc1000 r-xp 00000000 00:00 0                              [vdso]
hollowing passwd
write 0
write 1
write 2
write 3
write 4
write 5
write 6
write 7
write 8
write 9
write 10
write 11
Killing passwd's
# whoami
root
#
```

From: Jann Horn <jann@thejh.net>
To: kernel-hardening@lists.openwall.com, linux-kernel@vger.kernel.org
Cc: Andrew Morton <akpm@linux-foundation.org>,
HATAYAMA Daisuke <d.hatayama@jp.fujitsu.com>,
Vitaly Kuznetsov <vkuznets@redhat.com>,
Baoquan He <bhe@redhat.com>,
Masami Hiramatsu <masami.hiramatsu.pt@hitachi.com>
Subject: [RFC] kernel/panic: place an upper limit on number of oopses
Date: Tue, 12 Jan 2016 20:25:45 +0100 [thread overview]
Message-ID: <1452626745-31708-1-git-send-email-jann@thejh.net> (raw)

To prevent an attacker from turning a mostly harmless oops into an exploitable issue using a refcounter wraparound caused by repeated oopsing, limit the number of oopses.

I have not experimentally verified whether the attack I describe in the comment works, but I don't see why it wouldn't. (f_count increments through fget() use atomic_long_inc_not_zero(), but get_file() just does a normal increment and is e.g. used by dup_fd().)

This approach is strictly inferior to PAX_REFCOUNT, but as long as that's not upstreamed and turned on by default, it might make sense to at least use this patch.

Opinions?

Signed-off-by: Jann Horn <jann@thejh.net>

```
---
 kernel/panic.c | 28 ++++++++++++++++++++++++++++
 1 file changed, 28 insertions(+)
```

Subject: [PATCH] exit: Put an upper limit on how often we can oops
Date: Mon, 7 Nov 2022 21:13:17 +0100   [thread overview]
Message-ID: <20221107201317.324457-1-jannh@google.com> (raw)

Many Linux systems are configured to not panic on oops; but allowing an attacker to oops the system **really** often can make even bugs that look completely unexploitable exploitable (like NULL dereferences and such) if each crash elevates a refcount by one or a lock is taken in read mode, and this causes a counter to eventually overflow.

The most interesting counters for this are 32 bits wide (like open-coded refcounts that don't use refcount_t). (The ldsem reader count on 32-bit platforms is just 16 bits, but probably nobody cares about 32-bit platforms that much nowadays.)

So let's panic the system if the kernel is constantly oopsing.

The speed of oopsing 2^32 times probably depends on several factors, like how long the stack trace is and which unwinder you're using; an empirically important one is whether your console is showing a graphical environment or a text console that oopses will be printed to.
In a quick single-threaded benchmark, it looks like oopsing in a vfork() child with a very short stack trace only takes ~510 microseconds per run when a graphical console is active; but switching to a text console that oopses are printed to slows it down around 87x, to ~45 milliseconds per run.
(Adding more threads makes this faster, but the actual oops printing happens under &die_lock on x86, so you can maybe speed this up by a factor of around 2 and then any further improvement gets eaten up by lock contention.)

It looks like it would take around 8-12 days to overflow a 32-bit counter with repeated oopsing on a multi-core X86 system running a graphical environment; both me (in an X86 VM) and Seth (with a distro kernel on normal hardware in a standard configuration) got numbers in that ballpark.

12 days aren't *that* short on a desktop system, and you'd likely need much longer on a typical server system (assuming that people don't run graphical desktop environments on their servers), and this is a *very* noisy and violent approach to exploiting the kernel; and it also seems to take orders of magnitude longer on some machines, probably because stuff like EFI pstore will slow it down a ton if that's active.

Signed-off-by: Jann Horn <jannh@google.com>

Project Zero

News and updates from the Project Zero team at Google

Thursday, January 19, 2023

Exploiting null-dereferences in the Linux kernel

Posted by Seth Jenkins, Project Zero

For a fair amount of time, null-deref bugs were a highly exploitable kernel bug class. Back when the kernel was able to access userland memory without restriction, and userland programs were still able to map the zero page, there were many easy techniques for exploiting null-deref bugs. However with the introduction of modern exploit mitigations such as SMEP and SMAP, as well as `mmap_min_addr` preventing unprivileged programs from mmap'ing low addresses, null-deref bugs are generally not considered a security issue in modern kernel versions. This blog post provides an exploit technique demonstrating that treating these bugs as universally innocuous often leads to faulty evaluations of their relevance to security.

Conclusions

Innocuous bugs can be exploitable for subtle reasons. Identify side-effects of even “obviously” unexploitable bugs.

Ensure that the oops limit is merged down into distros (Ubuntu has it, CentOS not yet, others?)

OR redefine null-derefs (et. al) as a security-relevant bug-class...

Outside-the-box thinking is particularly valuable for exploit strategies and can make even the impossible possible.

Questions?