Breaking the Cube

A gentle introduction

Warmin’

Trollin’

Profit

Warmin

Don’t worry I’ll run it in a VM!

Virtualization

“Virtualization, in computing, refers to the act of creating a virtual (rather than actual) version of something, including but not limited to a virtual computer hardware platform, operating system (OS), storage device, or computer network resources.”

2 types of virtualization

- Hardware-based Virtual Machine (HVM)
- Para-virtualization

HVM

- Uses Intel VT-x in order to provide isolation
- Introduces a “new” ring mode, un/privileged mode
- Privileged instructions must be emulated by the Hypervisor
  - Great attack surface, parsing x86 is tedious
- Memory mode has to be decided by VMM
  - Huge headache to code
- Device security must be enumerated
  - Attacks against VT-d, etc

Paravirtualization

- First introduced by Xen
- Modify the Guest OS and remove all privileged instructions
- Kinda better performance than HVM due to MMU
- In Linux mainline since 2.6.23
- Hyper-V calls it “enlightened VM”
    - During boot of a Windows guest machine it can detect it is virtualized by Hyper-V and thus become “enlightened”

Guest-Host comm

- How does a VM asks for more memory ?
- How does a VM perform I/O operations?
- How does a VM perform a context switch?

Guest-Host comm

- HVMs offer a “trap-based” interface
  - Implemented as a bitarray of events in the Hypervisors
  - Later Intel + AMD offered “Virtualization exceptions” for custom operations
- Paravirtualization offers hypercalls
  - Special address range to jump into to cause a trap

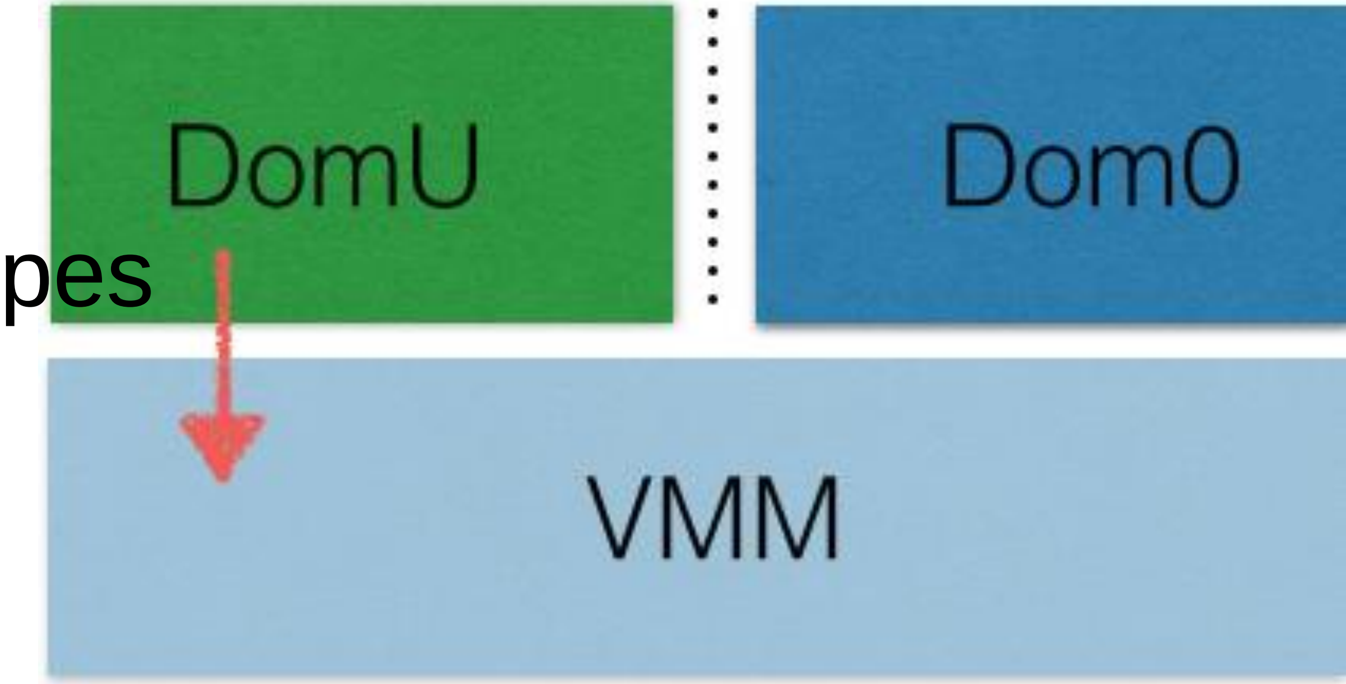
Rollin’

2 main types of escapes

- Golden Escapes
  - Direct escape from an unprivileged VM to the hypervisor
- Chained escapes
  - Exploitation of several bugs in order to reach the hypervisor
  - May it be logic or a memory corruption

Golden Escapes

- Direct escape from Guest to the Hypervisor
- Either by faulty hypercalls or by bad interpretation of an event
  - Use your imagination
- These are GOLDEN escapes
- I only know one public



XSA-7

Golden Escapes

- Kinda famous one
- Originally found somewhere in 2k6 (CVE-2006-044)
- Caused by improper understanding of the #GP
  - Intel said one thing, AMD said another
  - Xen only implemented AMD’s for both CPUs
- Resulted in a golden escape from DomU <-> Hypervisor

Golden Escapes

- On 64b only 48bits are used for the address space
  - This gives us only a maximum of 256 terabyte of RAM :(
- If you look at a typical 64b address you’d see that bits 48-63 are all the same as bit 47
  - For ex 0x**ffff**8000deadbeef
- Intel made this on purpose to stop from programmers from using those bits as special flags

Golden Escapes

- Intel called this range of valid addresses “canonical”
  - Meaning we can use it
  - 0x0000000000000000-0x00007fffffffffff
  - 0xffff800000000000-0xfffffffffffffff
- What happens if we try use (execute from) a non-canonical address then?

Listen Tight

A #GP fault occurs

Golden Escapes

- But this is where the magic comes
- On AMD the #GP occurs while the machine is in **guest mode**
- On Intel the #GP occurs while the machine is in **hypervisor mode**

Golden Escapes

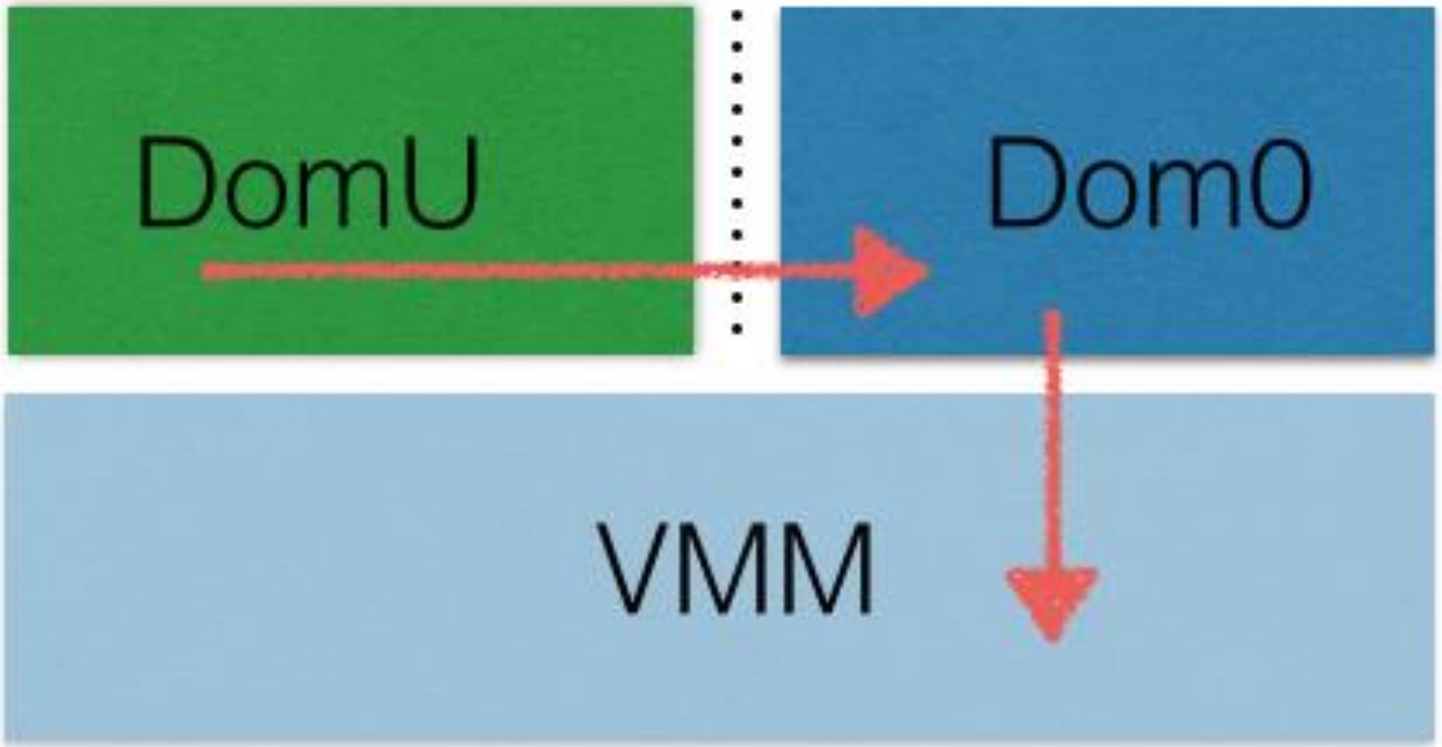
- While the #GP occurs, Xen restores all of it’s registers from the stack
- If we’re in Hypervisor mode, we’d get the **Hypervisor stack**
- But…
- If we’re in Guest mode, we’d get the **Guest stack**

That’s Gold

Golden Escapes

- Allocate a page in a non-canonical address
- Create simple shellcode in a non-canonical-address-1 (valid one)
  - So when we return from the syscall we’d generate a #GP
- Setup fake stack that the hypervisor will restore from
- Profit

Chained Escapes



Chained Escapes

- Kinda main topic :(
  - I’d love to find more golden escapes :X
- They’re usually composed by 2-3 bugs
  - Infoleak – read sensitive struct info from the hypervisor
  - Corruption or a logic bug on guest <-> dom0
  - Corruption or a logic bug on dom0 <-> hypervisor
    - Excluding driver domains as qemu is cheating

XSA-105

Chained Escapes

- HVM needs to be able to emulate privileged instructions
- LIDT, LGDT, INVLPG, LMSW
- The code which handles it resides in a huge case
- What could go wrong ?

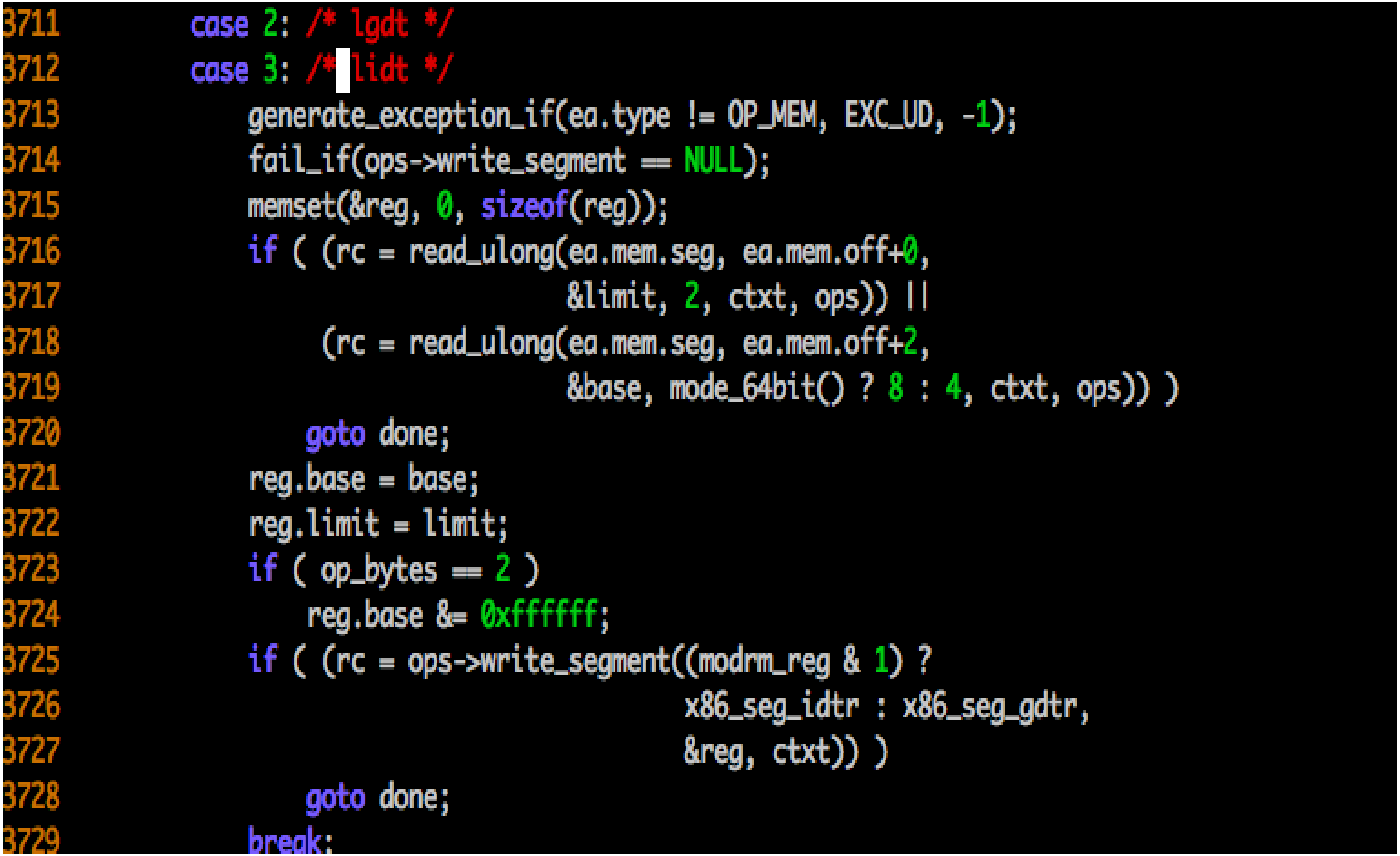
Chained Escapes

- LIDT = Load a new Interrupt Descriptor Table
- LGDT = Load a new Global Descriptor Table
- INVPLG = Cache flush, invalidate the page cache
- LMSW = Overwrite CR0

All of these are **sensitive** instructions

Parsing x86 is easy right ?

Chained Escapes



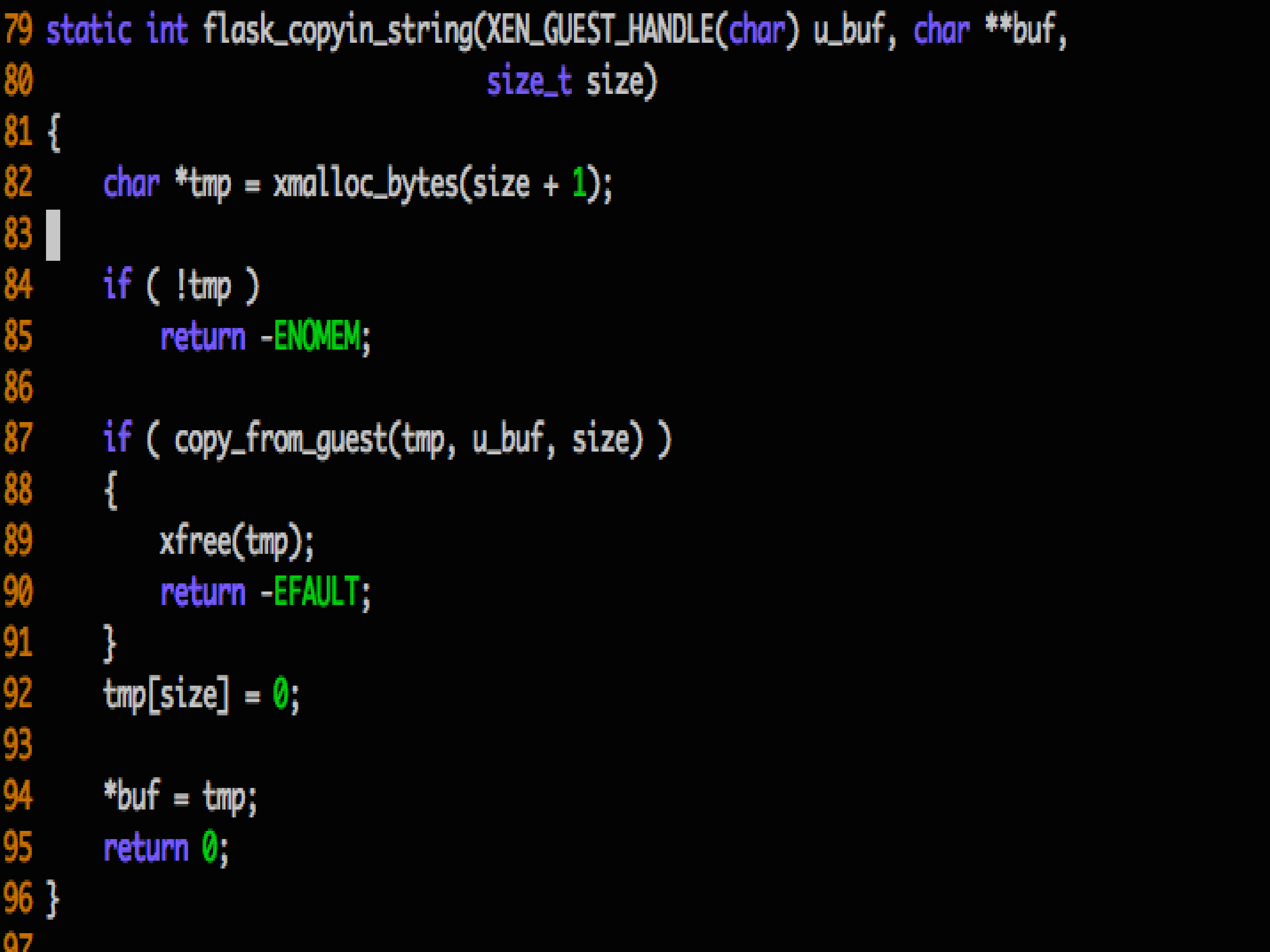
```
        case 2: /* lgdt */
        case 3: /* lidt */
            generate_exception_if(ea.type != OP_MEM, EXC_UD, -1);
            fail_if(ops->write_segment == NULL);
            memset(&reg, 0, sizeof(reg));
            if ( (rc = read_ulong(ea.mem.seg, ea.mem.off+0,
                                  &limit, 2, ctxt, ops)) ||
                 (rc = read_ulong(ea.mem.seg, ea.mem.off+2,
                                  &base, mode_64bit() ? 8 : 4, ctxt, ops)) )
                goto done;
            reg.base = base;
            reg.limit = limit;
            if ( op_bytes == 2 )
                reg.base &= 0xffffff;
            if ( (rc = ops->write_segment((modrm_reg & 1) ?
                                          x86_seg_idtr : x86_seg_gdtr,
                                          &reg, ctxt)) )
                goto done;
            break;
```

No privilege checks!

Chained Escapes

1. Kinda generic attaq:
    1. Create a custom IDT table in usermode
    2. Cause an invalid opcode to be generated
    3. Replace the opcode to be LIDT
        1. Be quick enough to do so
2. Profit

XSA-84



```c
static int flask_copyin_string(XEN_GUEST_HANDLE(char) u_buf, char **buf,
                               size_t size)
{
    char *tmp = xmalloc_bytes(size + 1);

    if ( !tmp )
        return -ENOMEM;

    if ( copy_from_guest(tmp, u_buf, size) )
    {
        xfree(tmp);
        return -EFAULT;
    }
    tmp[size] = 0;

    *buf = tmp;
    return 0;
}
```

Chained Escapes

- Kinda golden escape, but not a full one
- Finalizing this one needs another vulnerability
    - Other than an infoleak
    - And tmem internals..
- However this code is accessible directly using a hypercall
    - FLASK_SETBOOL hypercall

Chained Escapes

1. Try to allocate 4gb of memory
2. Xmalloc allocation add +1 by default
3. Profit from zero allocation

Profit

Profit

- I’d like to take things a bit to the next stage
- INSERT_HERE_PROGRAM_ANAL_BUZZ
- Below are some ideas I would like to try in the next several months

Anal

- I’d like to try to get Xen with Address Sanitizer
- This means modifying the TMEM implementation
- Sounds easy? It depends

From the xen-devel-ml:

"**Until** TMEM has **gained production maturity**, the Xen.org security team intends [..] to handle these and future TMEM vulnerabilities in public, **as if they were normal non-security-related bugs.**

We therefore intend that currently-known vulnerabilities will be publicly disclosed on the xen-devel mailing list, as normal bug reports, at the expiry of the XSA-15 embargo. In the meantime the list below may be helpful. "

Anal

- I’d like to get the Xen project compiled with Clang and have the ability to search for paths in it
- Maybe even use joern on it and model a few bugs
- I used ctags + cscope + grep in my research, until learning the whole tree to find the appropriate calls
  - Lots of indirect struct initialization, too many defines
- I’d like to be able to get basic graphing abilities
  - I’m using IDA originally, with my own compiled Xen
  - Doesn’t work for Guest <-> Host stuff..Lots of headaches

Anal

- Finally, I’d like to be able to set constraints on Xen and ask some solver to solve them for me

- Z3, yachs, stp, whatever

- Kinda gets all previous ideas into one project…

- Help is welcomed! :p I can share 0days

Thanks for listening

Questions?