# Facilitating Kernel UAF Vulnerability Exploitation: an idea

韩天硕

# Table of Content

# Overview

UAF vulnerability pattern:

- UAF chunk size
- certain bytes readable
- certain bytes writable

# Overview

Given an UAF vul pattern, I want to:

- identify candidate objects that can be sprayed to the UAF region
- figure out syscalls with proper arguments to perform the spray
- find proper approach to exploit

# Example1 OCTF 2021 Final - kernote

**UAF vulnerability pattern:**

- UAF size: one kmalloc-32 chunk
- readable bytes: none
- writable bytes: the first 8 bytes

**exploitation:** ***ldt_struct***

# Example1

## OCTF 2021 Final - kernote

```c
struct ldt_struct {
	/*
	 * Xen requires page-aligned LDTs with special permissions.  This is
	 * needed to prevent us from installing evil descriptors such as
	 * call gates.  On native, we could merge the ldt_struct and LDT
	 * allocations, but it's not worth trying to optimize.
	 */
	struct desc_struct	*entries;
	unsigned int		nr_entries;

	/*
	 * If PTI is in use, then the entries array is not mapped while we're
	 * in user mode.  The whole array will be aliased at the addressed
	 * given by ldt_slot_va(slot).  We use two slots so that we can allocate
	 * and map, and enable a new LDT without invalidating the mapping
	 * of an older, still-in-use LDT.
	 *
	 * slot will be -1 if this LDT doesn't have an alias mapping.
	 */
	int			slot;
};
```

```c
SYSCALL_DEFINE3(modify_ldt, int , func , void __user * , ptr ,
		unsigned long , bytecount)
{
	int ret = -ENOSYS;

	switch (func) {
	case 0:
		ret = read_ldt(ptr, bytecount);
		break;
	case 1:
		ret = write_ldt(ptr, bytecount, 1);
		break;
	case 2:
		ret = read_default_ldt(ptr, bytecount);
		break;
	case 0x11:
		ret = write_ldt(ptr, bytecount, 0);
		break;
	}
	/*
	 * The SYSCALL_DEFINE() macros give us an 'unsigned long'
	 * return type, but tht ABI for sys_modify_ldt() expects
	 * 'int'.  This cast gives us an int-sized value in %rax
	 * for the return code.  The 'unsigned' is necessary so
	 * the compiler does not try to sign-extend the negative
	 * return codes into the high half of the register when
	 * taking the value from int->long.
	 */
	return (unsigned int)ret;
}
```

# Example1

OCTF 2021 Final - kernote

```
static int read_ldt(void __user *ptr, unsigned long bytecount)
{
        struct mm_struct *mm = current->mm;
        unsigned long entries_size;
        int retval;

  ......

        if (copy_to_user(ptr, mm->context.ldt->entries, entries_size)) {
                retval = -EFAULT;
                goto out_unlock;
        }

  ......
```

# Example1 OCTF 2021 Final - kernote

two solves during the contest, both unintended:

- *@Balsn* gave a 1-day exp, unrelated to the vulnerability at all
- *@Organizor* gave a kernel ROP solve, too delicate and complicated to reproduce

conclusion: Identifying proper structs is vital for UAF exploitation

# Example1 OCTF 2021 Final - kernote

*bitbucket.org*

なお、検証にはLinux Kernel 4.19.98を使いました。



**Q1: where do these structs come from?**

# Example2 CVE-2021-26708

## UAF vulnerability pattern:

- UAF size: one kmalloc-64 chunk
- readable bytes: none
- writable bytes: 4 bytes at offset 40

# Example2 CVE-2021-26708

```c
/* one msg_msg structure for each message */
struct msg_msg {
        struct list_head m_list;
        long m_type;
        size_t m_ts;            /* message text size */
        struct msg_msgseg *next;
        void *security;
        /* the actual message follows immediately */
};
```

**the pointer *security* is at offset 40, and it will be passed to kfree() resulting to arbitrary free, and can be transformed into UAF**

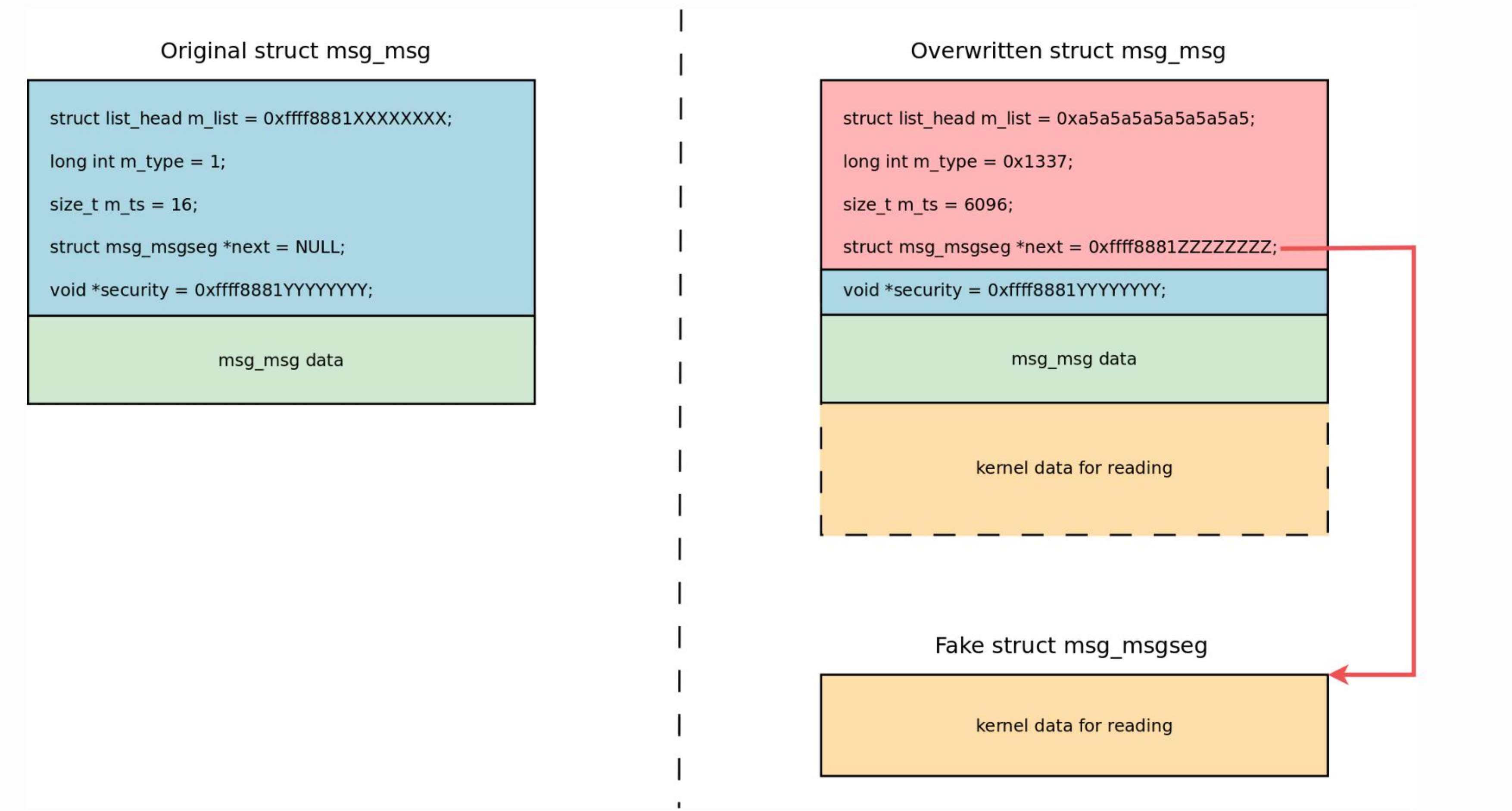
# Example2 CVE-2021-26708

## New UAF vulnerability pattern:

- UAF size: one kmalloc-64 chunk
- readable bytes: none
- writable bytes: the first 40 bytes

# Example2 CVE-2021-26708

## arbitrary read:



# Example2 CVE-2021-26708

execution path around msg_msg struct can be extended, achieving both arbitrary read and write (corCTF 2021 Finals - Fire of Salvation)

Q2: finding such execution path needs extensive expertise in kernel and intensive manual efforts. Can it be automatized?

# Example3 QWB2021 - notebook

## UAF vulnerability pattern:

- UAF size: one kmalloc-0x400 chunk
- readable bytes: the first 0x60 bytes
- writable bytes: the first 0x60 bytes

# Example3

QWB2021 - notebook

a fancy function, available in kernels who enable multi-cpu support:

```
static void work_for_cpu_fn(struct work_struct *work)
{
    struct work_for_cpu *wfc = container_of(work, struct work_for_cpu, work);

    wfc->ret = wfc->fn(wfc->arg);
}
```

after compilation:

```
static void work_for_cpu_fn(size_t * args)
{
    args[6] = ((size_t (*) (size_t)) (args[4](args[5]));
}
```

# Example3 QWB2021 - notebook

perfect for us to execute *commit_creds(prepare_kernel_cred(0))*

- hijack *ioctl()* of a tty struct to *work_for_cpu_fn()*
- set tty_struct[4]=*prepare_kernel_cred*, tty_struct[5]=0
- invoke *ioctl()* and the new generated cred struct will be placed at tty_struct[6], and we can read it
- do the trick again to invoke *commit_creds()*

# Example3 QWB2021 - notebook

**conclusion: execution path of exploit is not limited to syscall sequences, sometimes can be delicate.**

**Q3: Is it possible to identify more such fancy functions?**

# Possible Solution

Q1: where do these structs come from?

- manually create an UAF chunk
- trace kmalloc()
- fuzz the kernel until the UAF chunk is taken

# Possible Solution

Q2: finding such execution path needs extensive expertise in kernel and intensive manual efforts. Can it be automatized?

one possible solution...

if we can identify all control flow related to the struct, we can symbolicly execute them until we meet instructions like

- call/jmp SYMBOLIC_VALUE ------ control flow hijack
- mov [SYMBOLIC_VALUE], rsi ------ arbitrary read
- .....

# Possible Solution

Q3: Is it possible to identify more such fancy functions?

heuristic rules

# Related work


- Wu, W., Chen, Y., Xu, J., Xing, X., Gong, X., & Zou, W. (2018). FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. Proceedings of the 27th USENIX Security Symposium, 781–797.
- Chen, Y., & Xing, X. (2019). Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. Proceedings of the ACM Conference on Computer and Communications Security, 1707–1722. https://doi.org/10.1145/3319535.3363212

# Thanks!