Facilitating Linux Kernel Heap Vulnerability Exploitation: an idea

韩天硕

Table of Content

Overview

Kernel heap vulnerability form:

- double free
- use after free
- out of bound read / write
- .....

Overview

Kernel heap vulnerability capability model:

- the chunk size
- readable/writable bytes within(around) the chunk

Overview

General pattern to exploit a heap vul model:

- identify an useful kernel object from kernel code space
- allocate such an object to region under control
- untilizing an execution path(related to the object) to finish exploition / help with exploitation

Overview

General patterns to exploit a heap vul model:

- identify an useful kernel object from kernel code space
- allocate such an object to region under control
- untilizing an execution path(related to the object) to finish exploition / help with exploitation

all are unsolved challenges

Overview

I want to build a database/tool that, given a vul model, can indicate me: what can be done with such a model

more specifically:

- list candidate objects, that can fit in the evil chunk size and may be useful for exploitation
- syscall sequences to allocate such useful objects
- list the objects' capability and syscall sequences to trigger

Example1 OCTF 2021 Final - kernote

```c
void* kernote_ioctl(void* d, long cmd, long arg){
    ....
    void* ptr;
    void* note[0x10];

    if(cmd == CMD_ADD){

        note[arg] = kmalloc(0x20);

    }else if(cmd == CMD_SELECT){

        ptr = note[arg];

    }else if(cmd == CMD_WRITE){

        copy_from_user(ptr, arg, 8);

    }else if(cmd == CMD_DELE){

        kfree(note[arg]);

    }
    .....
}
```

Example1 OCTF 2021 Final - kernote

vulnerability model:

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

exploitation: *ldt_struct*

Example1

0CTF 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 and its useful execution path is vital for kernel heap 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。

- はじめに
- Leak/AAR/AAW/RIP制御に使える構造体一覧
  - shm_file_data
  - seq_operations
  - msg_msg (+user-supplied data)
  - subprocess_info
  - cred
  - file
  - timerfd_ctx
  - tty_struct
- 任意データ書き込み/Heap Sprayに使える構造体
  - msg_msg
  - setxattr
  - sendmsg
- 参考文献

Q1: where do these structs and execution path come from?

Example2 CVE-2021-26708

vulnerability model:

- chunk 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

Combining with the vul other capabilities, the vulnerability pattern can be transformed:

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

Example2

CVE-2021-26708

Conclusion: some execution path does not follow general exploitation paradigm, but may work at specific vul context

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

Example3 QWB2021 - notebook

vulnerability model:

- 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 1: execution path of exploit is not limited to syscall sequences, sometimes can be subtle.

conclusion 2: overread is as important as overwrite.

Q3: by taking extra dimensions of vul capability into consideration, is it possible to identify more exploit paradigm?

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
- Chen, Y., Lin, Z., & Xing, X. (2020). A Systematic Study of Elastic Objects in Kernel Exploitation. Proceedings of the ACM Conference on Computer and Communications Security

Related Work: SLAKE

SLAKE: identify kernel objects that enclose at least one function pointer, then filter out objects that may be utilized for RIP hijacking

- the object can be allocated from user land
- the function pointer can be called from user land

Related Work: SLAKE

e.g.

```c
struct seq_operations {
        void * (*start) (struct seq_file *m, loff_t *pos);
        void (*stop) (struct seq_file *m, void *v);
        void * (*next) (struct seq_file *m, void *v, loff_t *pos);
        int (*show) (struct seq_file *m, void *v);
};
```

```c
// allocate
int fd = open("/proc/self/stat");

// trigger
read(fd, buf, 1);
```

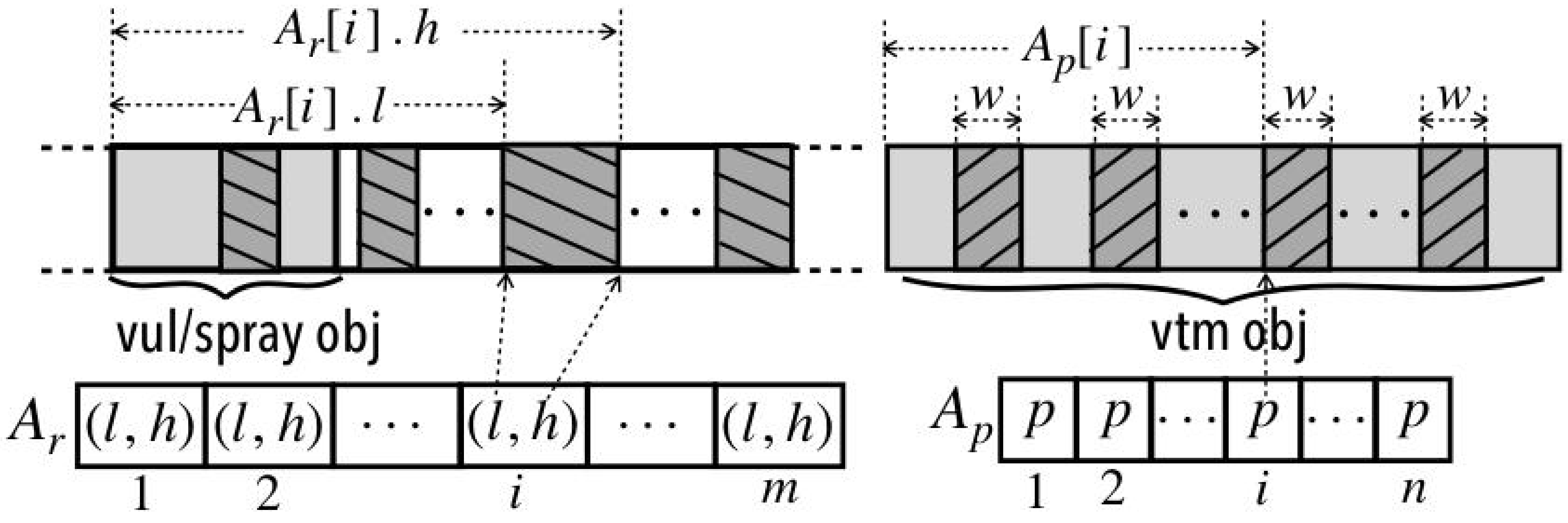
Related Work: SLAKE

SLAKE answered such a question: I have a heap chunk overwrite primitive and I want to do a RIP hijacking, how can I do it?

The identified useful kernel objects are stored in a database, and will be paired with vul capability model

Related Work: SLAKE

SLAKE vul capability model



(a) Vulnerability capability (b) Kernel object property

Related Work: ELOISE

ELOISE: focuses on elastic kernel objects, identify kernel objects and execution path for kernel space memory leaking

ELOISE answerd such a question: I have a kernel heap chunk overwrite primitive and I want to do a kernel space memory leaking, how can I do it?

Related Work: ELOISE

e.g.

```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;
}
```

Unsolved challgenges & thinking

1. there is a great number of exploitation paradigms, besides RIP hijacking and kernel space memory leaking
   - reading kernel space memory to a readable region(besides transfer them directly to user space)
   - kernel space memory overwrite
   - arbitrary kfree()
   - leaking not by elastic objects
   - critical kernel data corruption
   - spraying data into kernel heap chunk
   - .....

digging the kernel according to known exploitation paradigms can be inefficient and unscaleble

Unsolved challgenges & thinking

2. the vul capability model is inadequate to fully describe a vul’s capability

- chunk size
- writable bytes

- chunk size
- chunk number I can control
- times I can control for the same chunk
- is the chunk at same address
- writable bytes
- times I can write
- readble bytes
- times I can read
- .....

stronger the vul capability indicates more possible exploitation path

Unsolved challgenges & thinking

3. in many situations, I only know the vul capability at hand, I have no idea what to do next

CVE-2021-26708 vulnerability model:

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

- RIP hijacking is impossible
- kernel memory leaking is unlikely to work
- what I do with this primitive?

knowing what can be done with a vul primitive can greatly indicate how to perform exploit

Possible Solution

loose the constraint

SLAKE and ELOISE digs the kernel strictly subject to constraints, to make sure the found objects and syscall sequences follows the desired exploitation paradigm

Possible Solution

mining the kernel not according to known exploitation paradigms, but by considering value and reachability

- value: heuristicly definning valuable operations
- reachability: filter out operations can be triggerd by syscall sequences from user land

Possible Solution

value:

1. definning valuable operations: memory read/write, call/jmp, critical kernel functions like *kfree()*, ...
2. mark controllable bytes as taint source / symbolic value
3. checking if valuable operations can be tainted / controlled by symbolic values

Possible Solution

reachability:

1. identify all valuable operation sites in execution path related to the kernel object
2. fuzz syscalls, until valuable operation sites reached

Possible Solution

e.g. find the arbitrary free() primitive in CVE-2021-26708

```
/* 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 */
};
```

1. mark all fields in this struct as taint source
2. fuzz the kernel until found kfree() is tainted by *security*

Possible Solution

store identified kernel objects, syscalls and corresponding capability in a database, and give an algorithm that retrieve information from the database. Given a vul model such a database can:

- greatly indicate exploitation path by knowing all possible operations
- give indications on how to implement an eploitation path

Thanks!