Deep-Kernel Treasure Hunt

Finding exploitable structures in the Linux kernel

About Me

Yudai Fujiwara

- Security researcher at Ricerca Security, Inc
- Binary exploitation (pwn)
- Twitter: @ptrYudai

What is this talk about?

The challenge:

How do you find *exploit-friendly* structures for heap-related bugs?

The goal:

To find dynamic and *exploit-friendly* structures allocated in a program

The outcome:

Discovered more than 1k *exploit-friendly* structures from the Linux

A Bug is NOT Exploitable Itself

Redis is an in-memory database that persists on disk. A specially crafted Lua script executing in Redis can trigger a heap overflow in the cjson library, and result with heap corruption and potentially remote code execution. The problem exists in all versions of Redis with Lua scripting support, starting from 2.6, and affects only authenticated and authorized users. The problem is fixed in versions 7.0.12, 6.2.13, and 6.0.20.

CVE-2022-24834

Heap buffer overflow in PDF in Google Chrome prior to 118.0.5993.70 allowed a remote attacker who convinced a user to engage in specific user interactions to potentially exploit heap corruption via a crafted PDF file. (Chromium security severity: Medium)

CVE-2023-5474

A Bug is NOT Exploitable Itself

Redis is an in-memory database that persists on disk. A specially crafted Lua script executing in Redis can trigger a heap overflow in the cjson library, and result with heap corruption and potentially remote code execution. The problem exists in all versions of Redis with Lua scripting support, starting from 2.6, and affects only authenticated and authorized us 20.

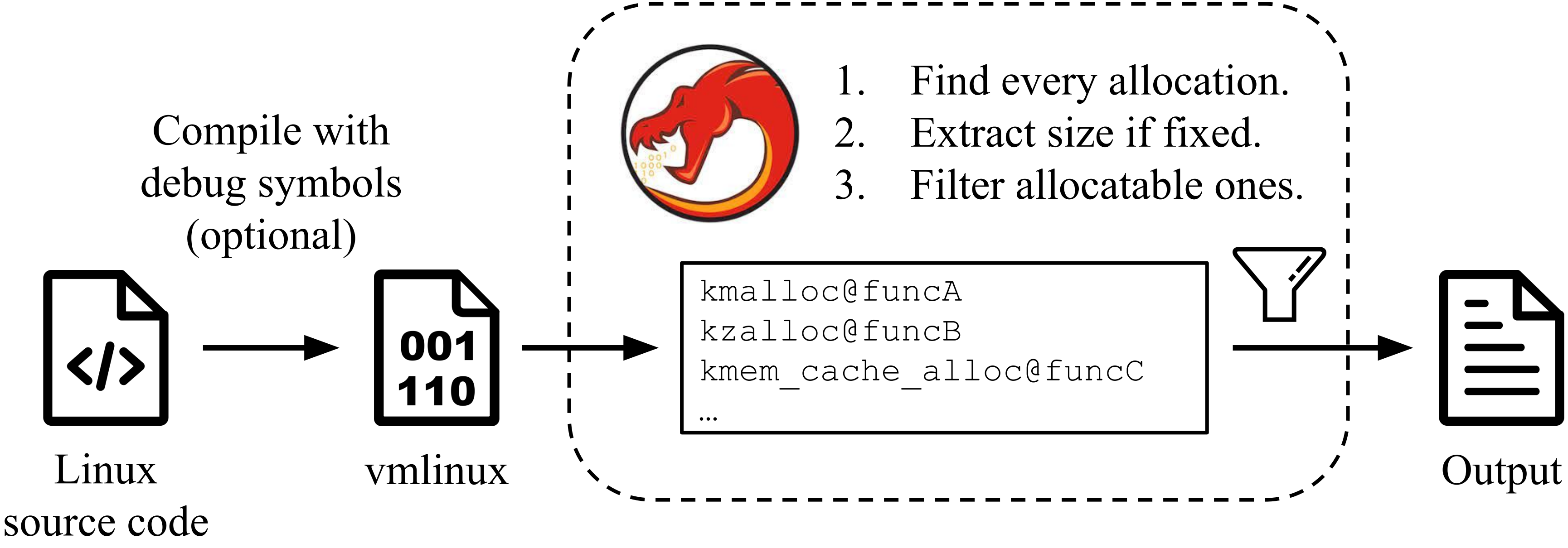
Exploit = Bug + *Useful* Structures

Heap buffer overflow in PDF in Google Chrome prior to 118.0.5993.70 allowed a remote attacker who convinced a user to engage in specific user interactions to potentially exploit heap corruption via a crafted PDF file. (Chromium security severity: Medium)

CVE-2023-5474

Proposal: MALTIES

MALTIES; Malloc Tracker for Identifying Exploitable Structures



The Target

- Linux kernel 6.1.52
- Structures dynamically allocated on kmalloc-xxx (slab caches)

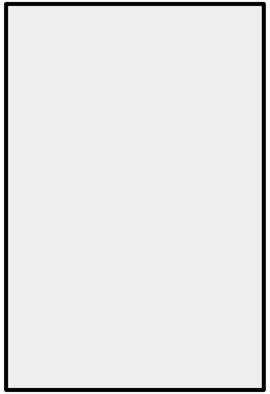
kmalloc-8k	384	388	8192	4
kmalloc-4k	1293	1296	4096	8
kmalloc-2k	2640	2640	2048	16
kmalloc-1k	2729	2752	1024	32
kmalloc-512	14874	14976	512	32
kmalloc-256	8695	8704	256	32
kmalloc-192	12869	13083	192	21
kmalloc-128	2619	2752	128	32
kmalloc-96	4196	5292	96	42
kmalloc-64	20602	21184	64	64
kmalloc-32	17719	18048	32	128
kmalloc-16	26663	33024	16	256
kmalloc-8	13663	13824	8	512

Background and Motivation

Heap in the Kernel

Heap - used to store data for drivers and the kernel

- `kmalloc` carves out a certain size of chunk
- `kfree` marks a specific chunk as *freed*
- They generally share the same heap

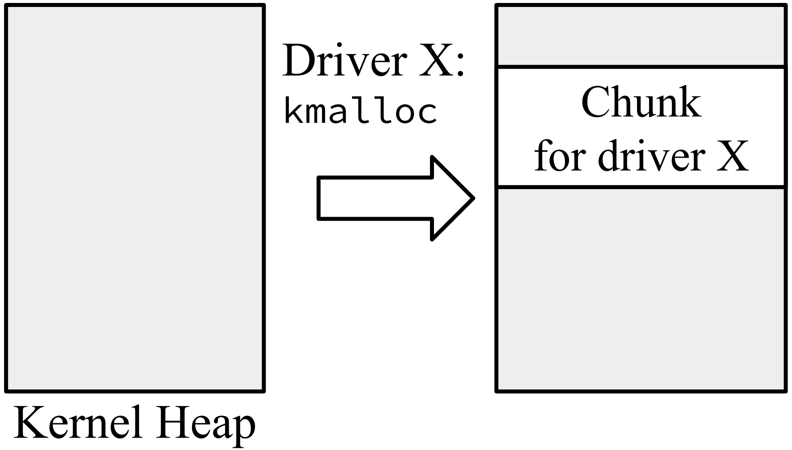


Kernel Heap

Heap in the Kernel

Heap - used to store data for drivers and the kernel

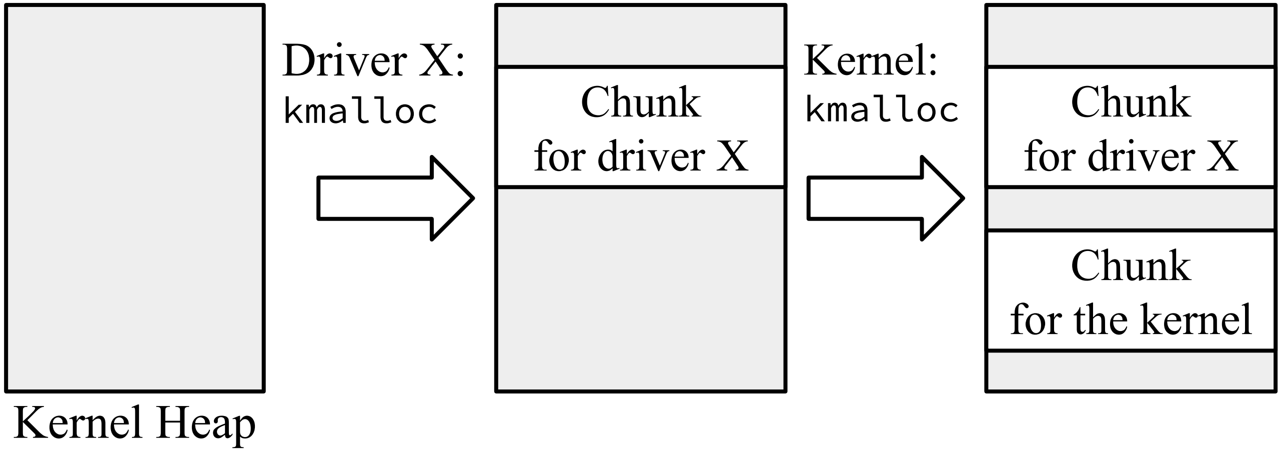
- **`kmalloc` carves out a certain size of chunk**
- `kfree` marks a specific chunk as *freed*
- They generally share the same heap



Heap in the Kernel

Heap - used to store data for drivers and the kernel

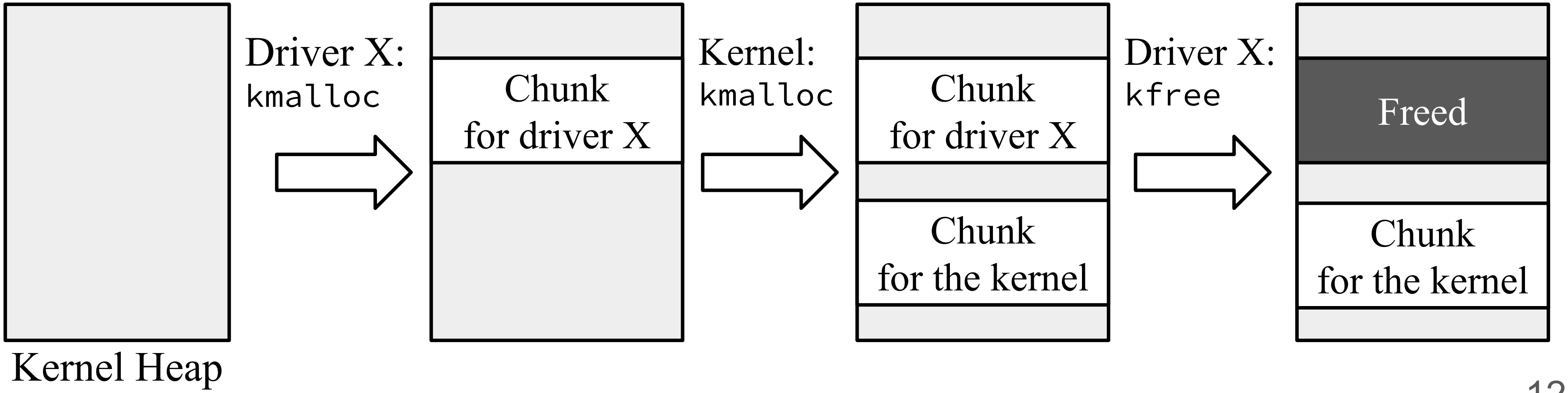
- **`kmalloc` carves out a certain size of chunk**
- `kfree` marks a specific chunk as *freed*
- They generally share the same heap



Heap in the Kernel

Heap - used to store data for drivers and the kernel

- `kmalloc` carves out a certain size of chunk
- **`kfree` marks a specific chunk as *freed***
- They generally share the same heap

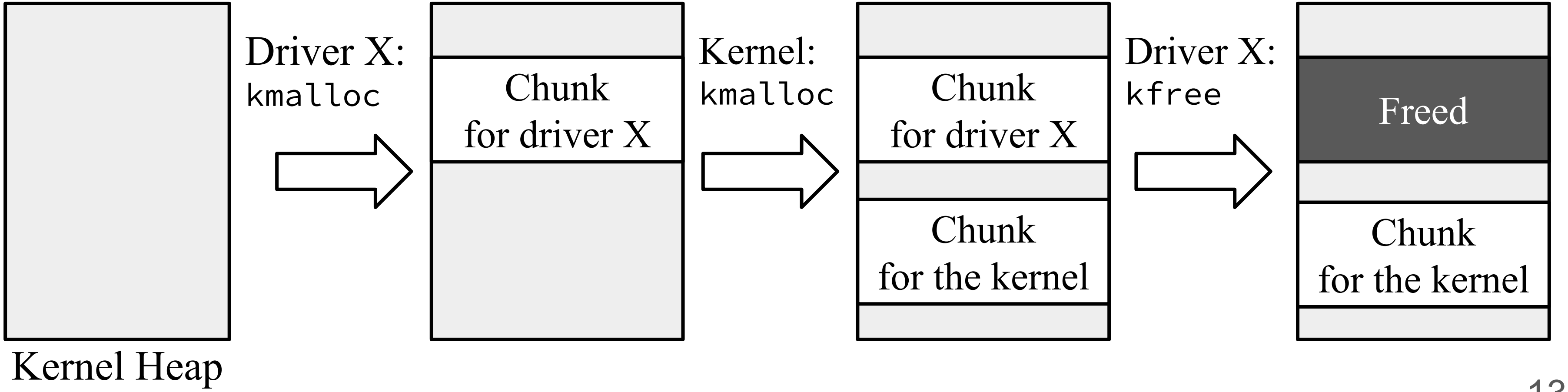


Heap in the Kernel

Heap - used to store data for drivers and the kernel

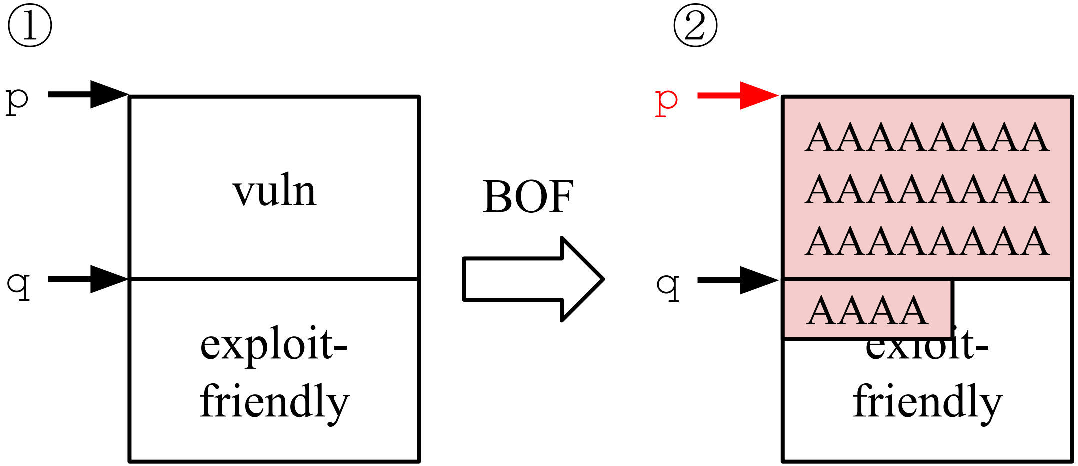
- `kmalloc` carves out a certain s
- `kfree` marks a specific chunk as *fr*
- **They generally share the same heap**

Heap bugs in a driver can affect the entire kernel



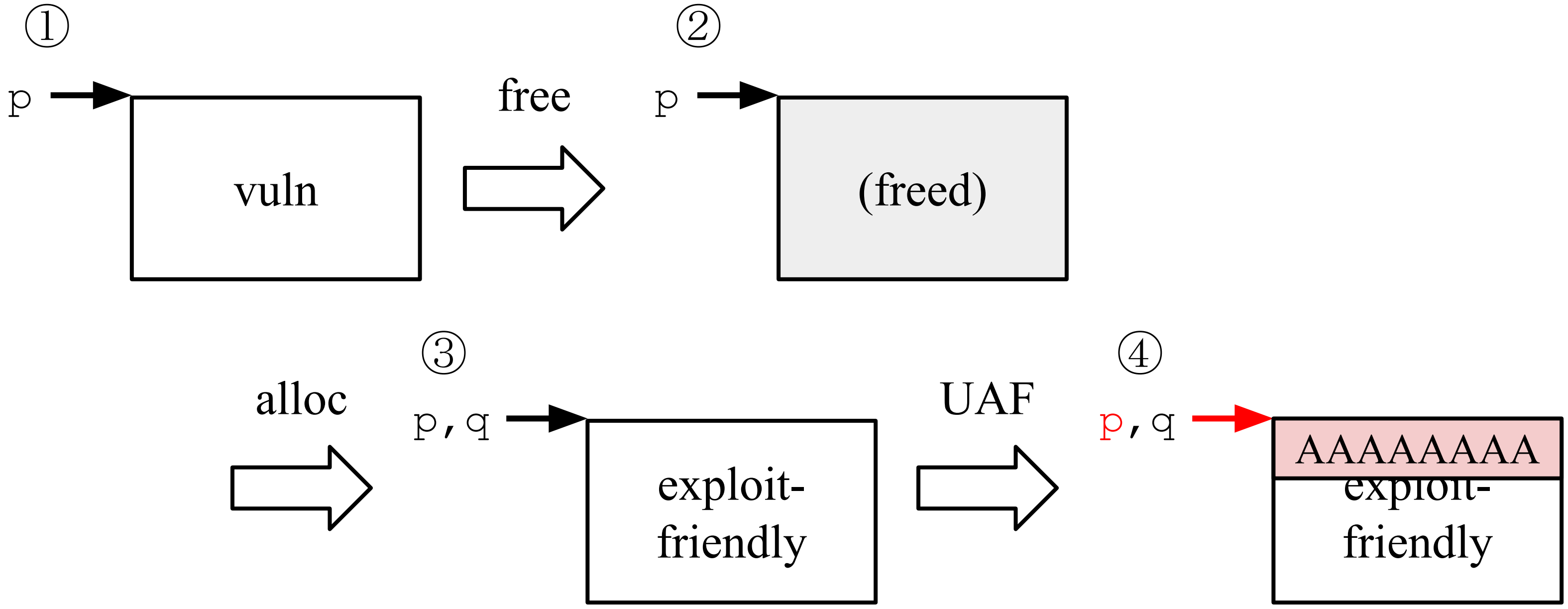
Heap Buffer Overflow

- Out-of-bounds memory access
- Exploitable if an *exploit-friendly* chunk follows the vuln chunk



Use-after-Free

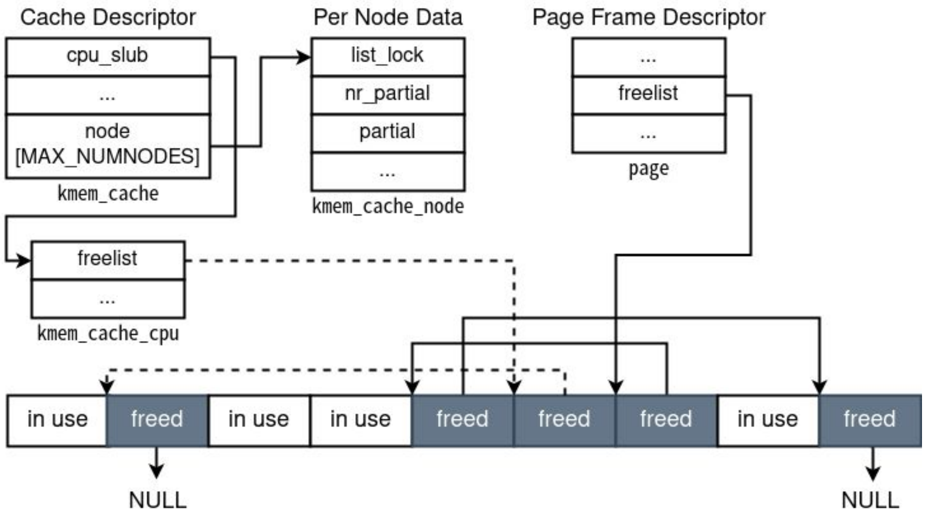
- Memory access in a freed chunk (dangling pointer)
- Exploitable if an *exploit-friendly* chunk is allocated on the vuln chunk



SLUB

SLUB - heap manager used in the Linux kernel

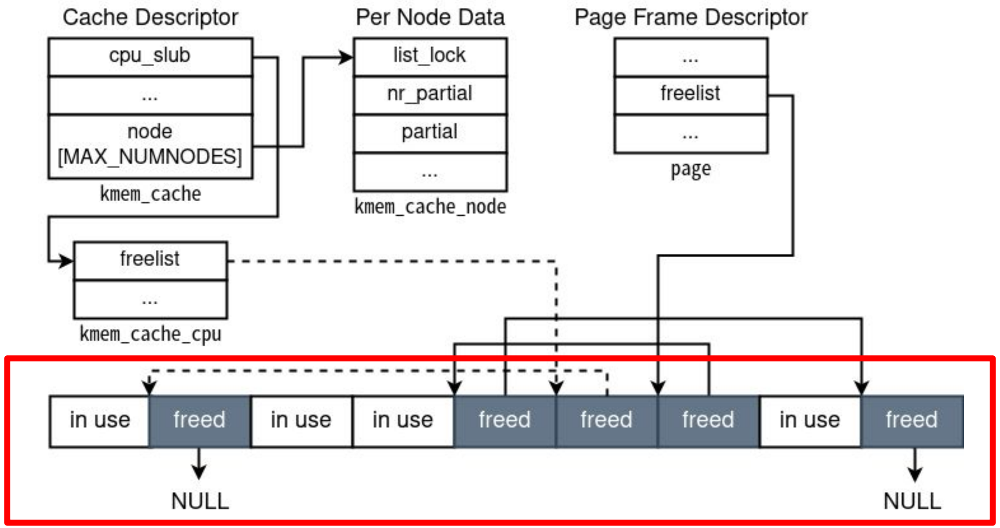
- Uses a different page frame for each size of allocation
- Manages freed chunks (cache) with singly-linked lists



SLUB

SLUB - heap manager used in the Linux kernel

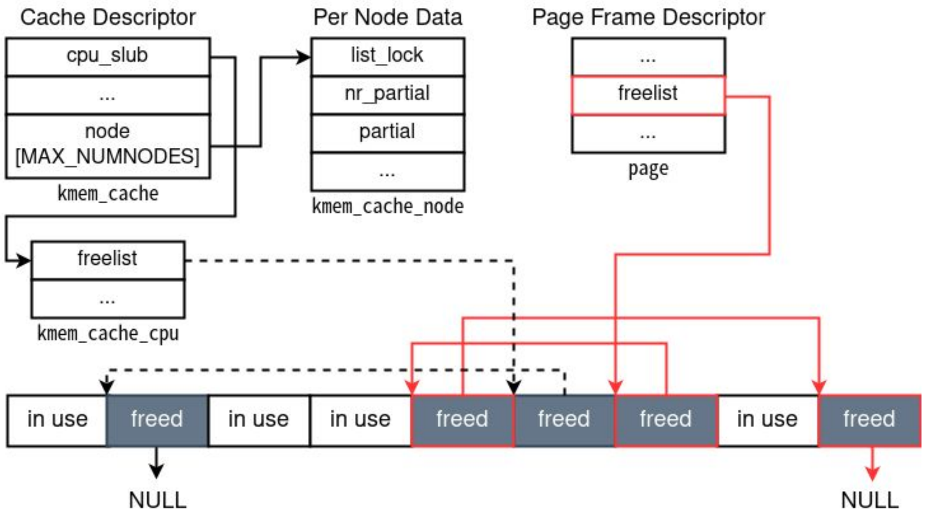
- **Uses a different page frame for each size of allocation**
- Manages freed chunks (cache) with singly-linked lists



SLUB

SLUB - heap manager used in the Linux kernel

- Uses a different page frame for each size of allocation
- **Manages freed chunks (cache) with singly-linked lists**



Exploit-Friendly Structures

- Can control the allocation from user-land
- With exploitable or controllable members
  - i.e., Function pointers, Blob data, etc
- Structure of a size similar to that of the vulnerable chunk
  - kmalloc-8, kmalloc-16, kmalloc-32, …, kmalloc-4k, kmalloc-8k

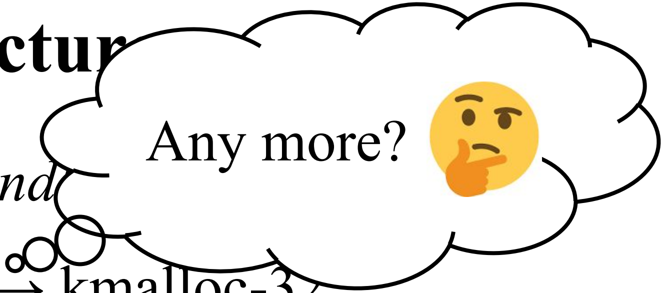
Well-Known Structures

Examples of *exploit-friendly* structures:

- `seq_operations` → kmalloc-32
- `tty_struct` → kmalloc-1024
- `setxattr` → arbitrary data and size, but is freed immediately
- `msg_msg` → arbitrary data and size ranging from 0x31 to 0x1000

i.e., Open `/dev/ptmx` to allocate a `tty_struct` structure

Well-Known Structur



Examples of *exploit-friend*

- `seq_operations` → kmalloc-32
- `tty_struct` → kmalloc-1024
- `setxattr` → arbitrary data and size, but is freed immediately
- `msg_msg` → arbitrary data and size ranging from 0x31 to 0x1000

i.e., Open `/dev/ptmx` to allocate a `tty_struct` structure

Related Works

Related Works

- Dynamic analysis
  - [dyn-1] Instrument `kmalloc` to detect runtime allocations
- Static analysis
  - [stat-1] Utilize LLVM to spot all `kmalloc` calls in the source code
  - [stat-2] Analyze the Linux source code to locate all structures
  - [stat-3] Analyze the debug info of vmlinux to identify all structures

Challenges

- Size accuracy
  - Determine the precise size of the allocation
- Coverage accuracy
  - Locate all allocations through `kmalloc`
- Allocatability
  - Verify whether the structure can be allocated from user-land
- Exploitability
  - Check if the structure has exploitable or controllable data

Introducing MALTIES

MALTIES

MALTIES; Malloc Tracker for Identifying Exploitable Structures

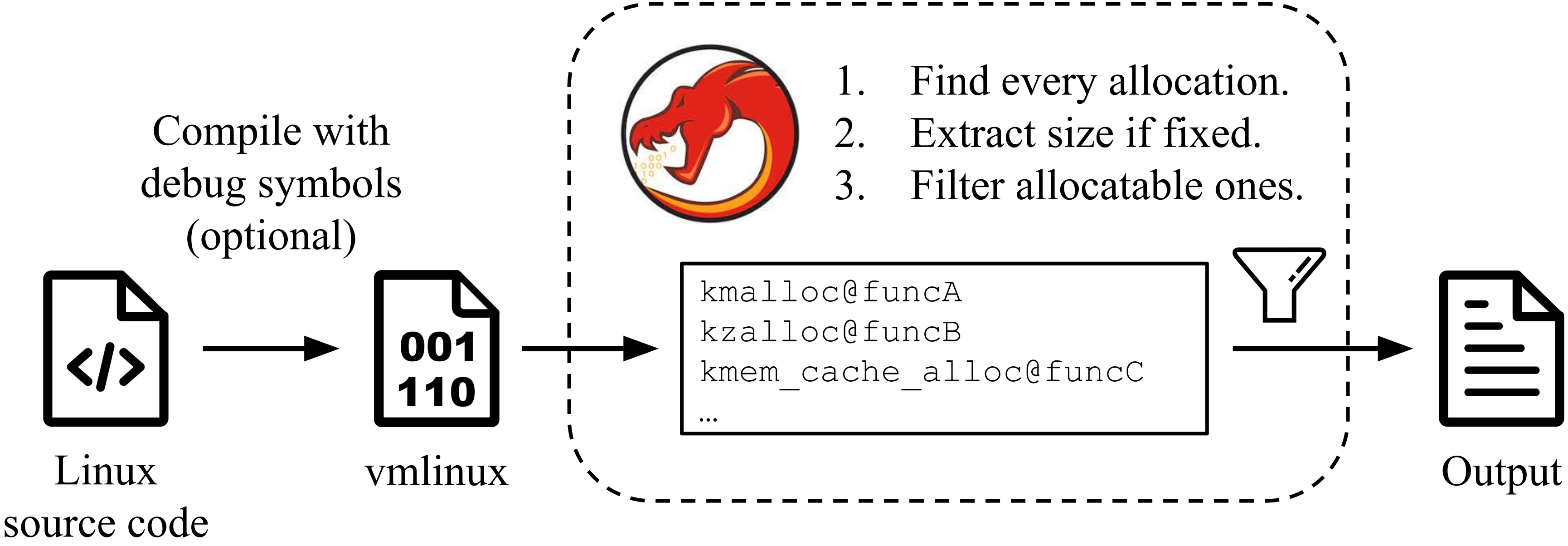
- MALTIES finds:
  - Dynamically allocated structures
  - Size of each allocation
  - Possible paths to reach the allocation
- MALTIES extracts:
  - Structures likely allocatable from user-land.
  - Structures likely exploitable. (optional)

Comparison

Method	Size accuracy	Coverage accuracy	Allocatability check	Exploitablity check	Support binary-only
dyn-1	✔	✗	✔	✗	✔
stat-1	✔	✔	✗	✗	✗
stat-2	✔	△	✗	✗	✗
stat-3	✔	△	✗	✗	✗
MALTIES	✔	✔	△	△	✔

Algorithm

MALTIES is written as a Ghidra script



Why Ghidra?

- Open source
- Powerful disassembler and decompiler
- Binary-oriented
- Good coverage
- Simplifies the process of locating function references

Algorithm

```
void fun_XXXX() {
  size_t lVarA = 0x110;
  …
  lVarB = __kmalloc(lVarA, param_2 | 0x100);
  …
}

void fun_YYYY(long param_1) {
  …
  lVarD = __kmalloc(param_1, 0xdc0)
  …
}
```

Algorithm

① Locate all allocations such as kmalloc, kmalloc_trace, etc

```
void fun_XXXX() {
   size_t lVarA = 0x110;
   …
   lVarB = __kmalloc(lVarA, param_2 | 0x100);
   …
}

void fun_YYYY(long param_1) {
   …
   lVarD = __kmalloc(param_1, 0xdc0)
   …
}
```

Algorithm

② Propagate variables to identify size

```
void fun_XXXX() {
   size_t lVarA = 0x110;
   …
   lVarB = __kmalloc(lVarA, param_2 | 0x100);
   …
}

void fun_YYYY(long param_1)
   …
   lVarD = __kmalloc(param_1, 0xdc0)
   …
}
```

size: 0x110

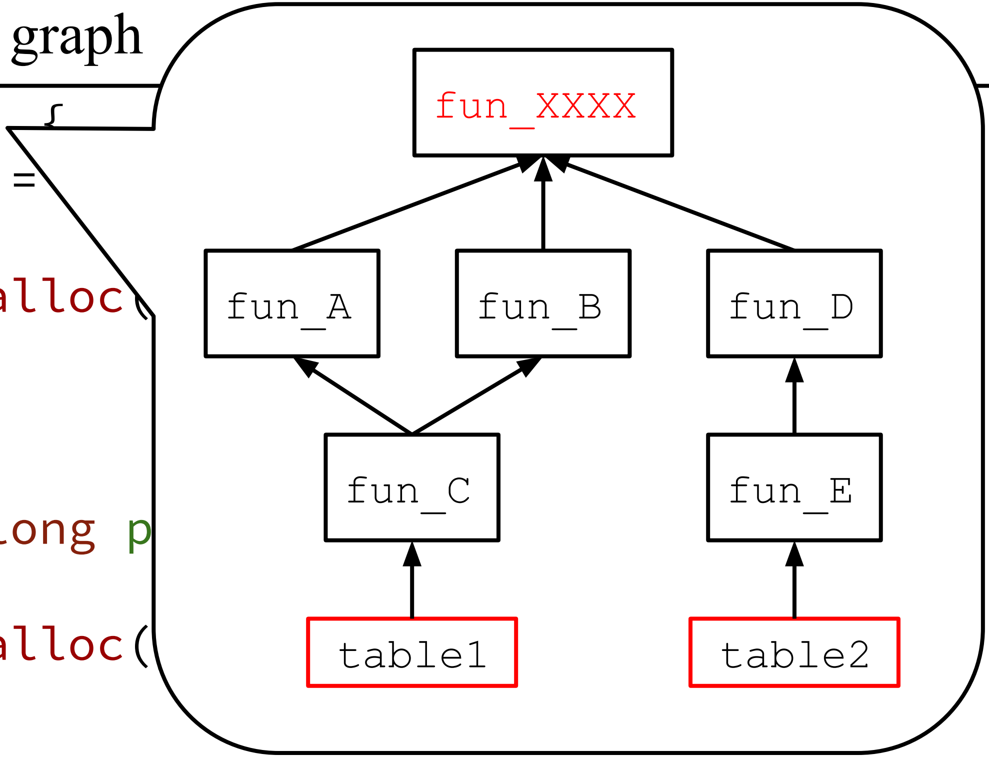
size: variable

Algorithm

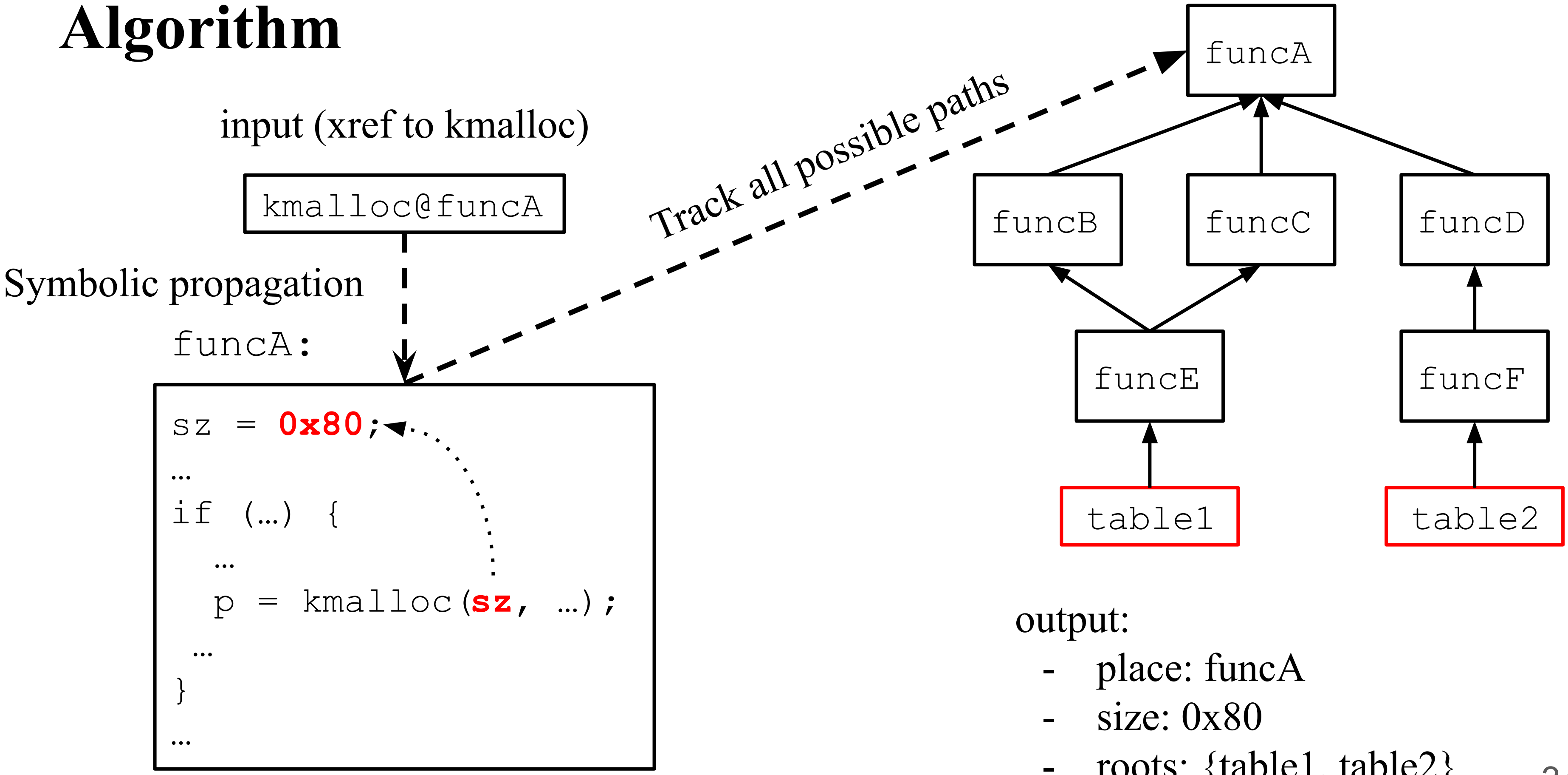
③ Traverse control flow graph

```
void fun_XXXX() {
   size_t lVarA =
   …
   lVarB = __kmalloc(
   …
}

void fun_YYYY(long p
   …
   lVarD = __kmalloc(
   …
}
```



Algorithm


input (xref to kmalloc)
kmalloc@funcA
Symbolic propagation
Track all possible paths
funcA:
sz = 0x80;
…
if (…) {
…
p = kmalloc(sz, …);
…
}
…
funcA
funcB
funcC
funcD
funcE
funcF
table1
table2
output:
- place: funcA
- size: 0x80
- roots: {table1, table2}

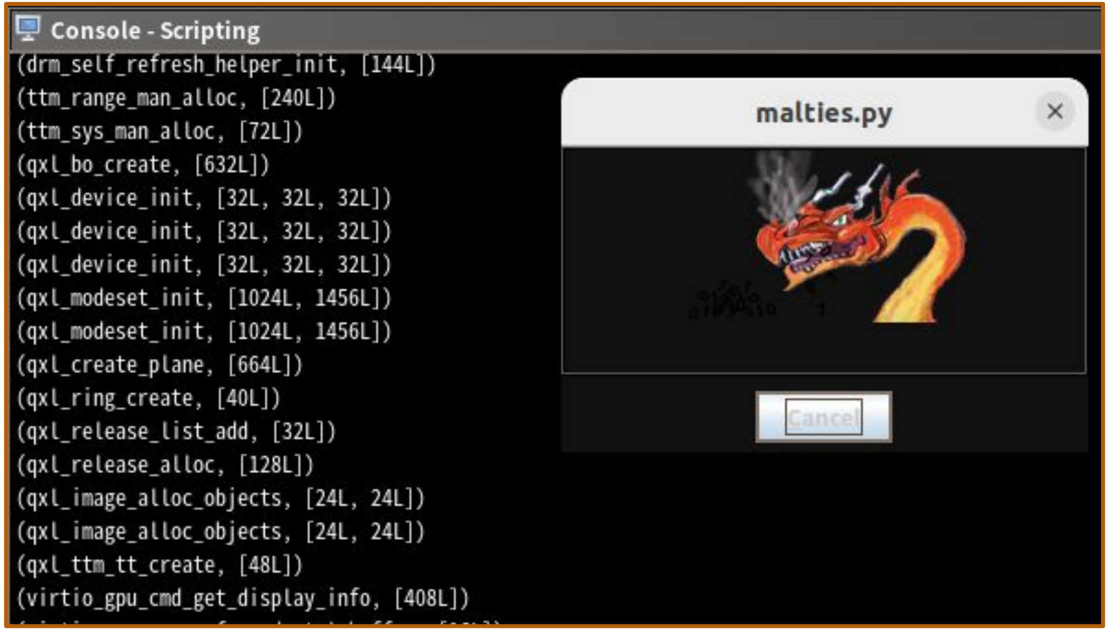
Result

Discovered Structures

Found a total of 1133 structures (excluding kmem_cache)

- signalfd_ctx
- tty_ldisc
- sock_fprog_kern
- inotify_event_info
- sem_undo_list
- sk_filter

and many more……



Example 1: kmalloc-8

```c
struct signalfd_ctx {
        sigset_t sigmask;
};
```

signalfd_ctx

- Allocatable?
  - Call signalfd or signalfd4 to allocate
  - Close fd and wait for RCU to free
- Exploit-friendly?
  - Readable from user-land (Read /proc/self/fdinfo/<fd>)
  - Writable from user-land (Call sigdelsetmask and signotset)

Example 2: kmalloc-256

`shmid_kernel`

- Allocatable?
  - Call `shmget` to allocate
  - Call `shmctl` with `IPC_RMID` to free

```
struct shmid_kernel /* private to the kernel */
{
	struct kern_ipc_perm	shm_perm;
	struct file		*shm_file;
	unsigned long		shm_nattch;
	unsigned long		shm_segsz;
	time64_t		shm_atim;
	time64_t		shm_dtim;
	time64_t		shm_ctim;
	struct pid		*shm_cprid;
	struct pid		*shm_lprid;
	struct ucounts		*mlock_ucounts;

	/*
	 * The task created the shm object, for
	 * task_lock(shp->shm_creator)
	 */
	struct task_struct	*shm_creator;

	/*
	 * List by creator. task_lock(->shm_crea
	 * If list_empty(), then the creator is 
	 */
	struct list_head	shm_clist;
	struct ipc_namespace	*ns;
} __randomize_layout;
```

Example 2: kmalloc-256

`shmid_kernel`

- Exploit-friendly?
  - `ns` has a pointer to `init_ipc_ns`
    - ➢ Useful for bypassing FGKASLR
  - `shm_creator` has a pointer to `current`
    - ➢ Useful for overwriting `cred`
  - `shm_perm` has a function pointer

```
struct shmid_kernel /* private to the kernel */
{
        struct kern_ipc_perm    shm_perm;
        struct file             *shm_file;
        unsigned long           shm_nattch;
        unsigned long           shm_segsz;
        time64_t                shm_atim;
        time64_t                shm_dtim;
        time64_t                shm_ctim;
        struct pid              *shm_cprid;
        struct pid              *shm_lprid;
        struct ucounts          *mlock_ucounts;

        /*
         * The task created the shm object, for
         * task_lock(shp->shm_creator)
         */
        struct task_struct      *shm_creator;

        /*
         * List by creator. task_lock(->shm_crea
         * If list_empty(), then the creator is
         */
        struct list_head        shm_clist;
        struct ipc_namespace    *ns;
} __randomize_layout;
```

Example 3: variable size

`sem_array`

- Allocatable?
  - Call `semget` to allocate
    - ➢ Set `nsems` to control size
- Exploit-friendly?
  - Elements of `sems` are controllable
  - `sem_perm` has a function pointer

```
struct sem_array {
        struct kern_ipc_perm    sem_perm;
        time64_t                sem_ctime;
        struct list_head        pending_alter;

        struct list_head        pending_const;

        struct list_head        list_id;
        int                     sem_nsems;
        int                     complex_count;
        unsigned int            use_global_lock;

        struct sem              sems[];
} __randomize_layout;
```

Future Work

- Indirect branches
  - Ghidra cannot find some xrefs of indirect branches
    - ➢ i.e., Dynamically allocated function table
- Variable sizes
  - The size of allocation is often variable
  - Requires *precise* (=sound) DFA to calculate the range of possible sizes

Conclusion

- Discovered >1k *exploit-friendly* structures in the Linux
  - Structures that can be allocated from user-land
  - Structures that have exploitable/controllable data fields
- MALTIES is designed for general purpose
  - Not only for the Linux kernel but also for user-land applications
  - Applicable to close-sourced and large software

Takeaways

- Exploiting heap-related bugs
  - Requires *exploit-friendly* structures
- Vulnerabilities in the kernel space
  - Easier to exploit
  - More critical
  - Do not easily install drivers
- Ghidra script
  - A strong tool to easily write complicated algorithms

Questions?

Yudai Fujiwara

Work: yudaif@ricsec.co.jp

Personal: ptr.yudai@gmail.com

Twitter: @ptrYudai

Special thanks:

CODEBLUE staffs

Yuichi Sugiyama (Ricerca Security, Inc.)