SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel

Yueqi (Lewis) Chen, Xinyu Xing

The Pennsylvania State University

ACM CCS 2019
Nov 14th

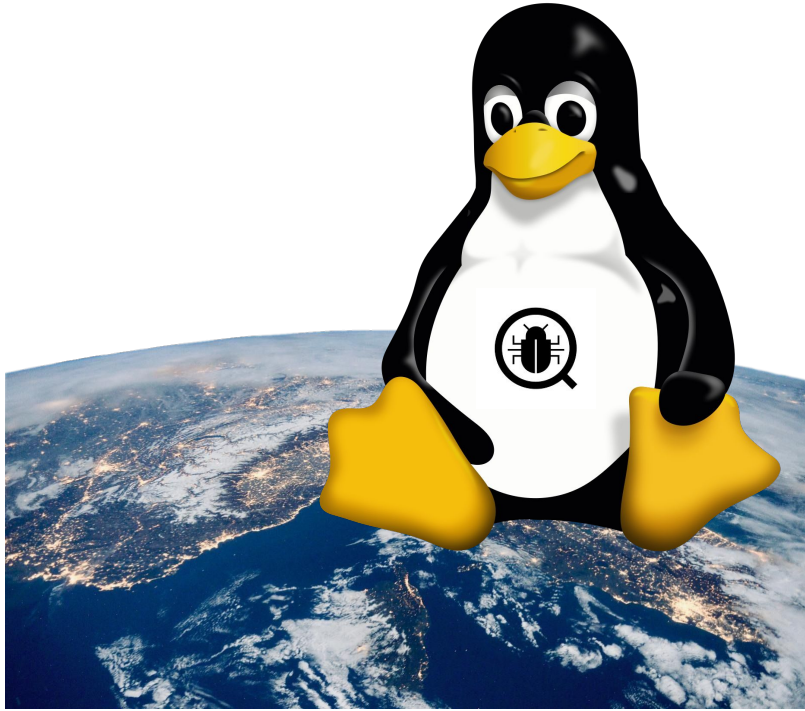
Linux Kernel is Security-critical But Buggy

"Civilization runs on Linux"[1][2]

- Android (2e9 users)
- cloud servers, desktops
- cars, transportation
- power generation
- nuclear submarines, etc.

Linux kernel is buggy

- 631 CVEs in two years (2017, 2018)
- 4100+ official bug fixes in 2017

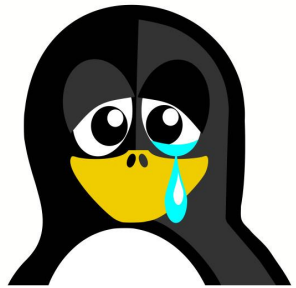


[1] SLTS project, https://lwn.net/Articles/749530/
[2] "Syzbot and the Tale of Thousand Kernel Bugs" - Dmitry Vyukov, Google

Harsh Reality: Cannot Patch All Bugs Immediately

Google Syzbot[3] , on Nov 14th

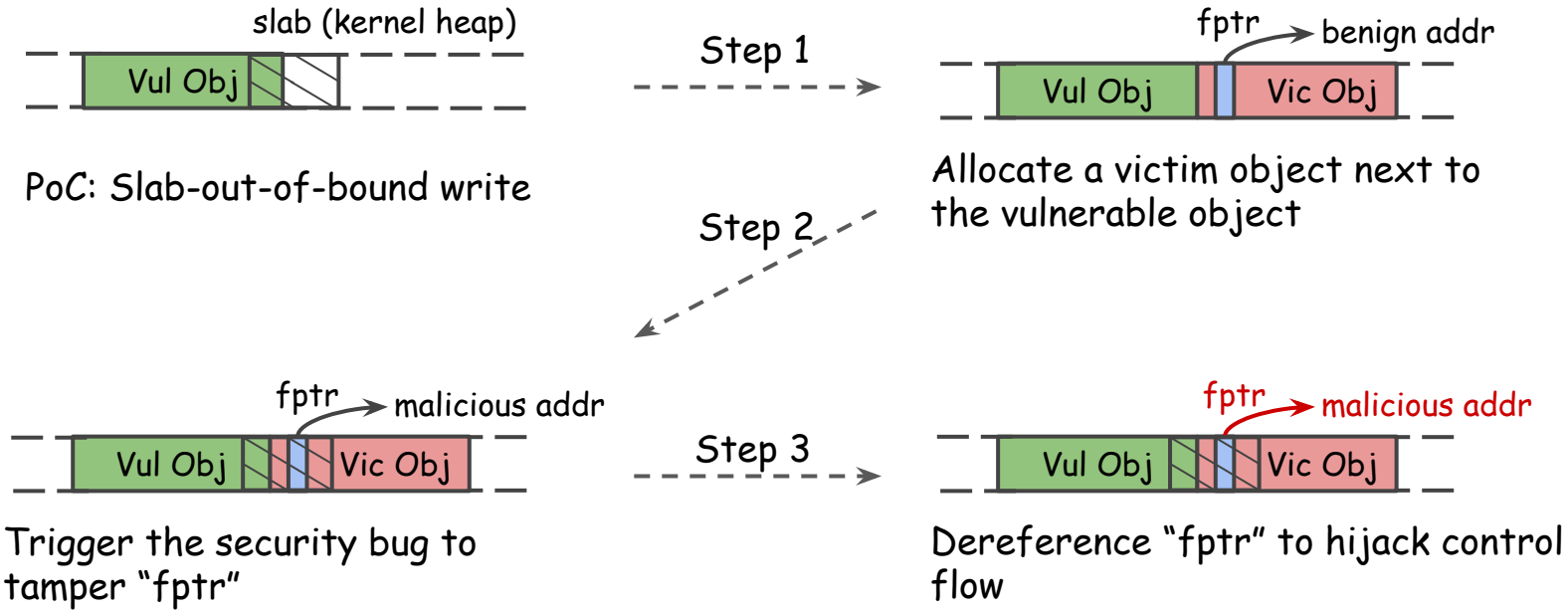
- 487 not fixed, 106 fix pending, 51 in moderation
- # of bug reports increases 200 bugs/month



Practical solution to minimize the damage: prioritize patching of security bugs based on **exploitability**

[3] syzbot https://syzkaller.appspot.com/upstream

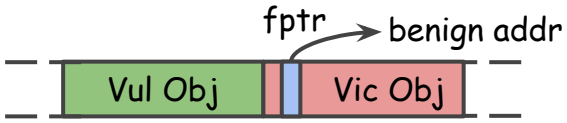
Workflow of Determining Exploitability



Example: Exploit A Slab Out-of-bound Write in Three Steps

Challenges of Developing Exploits

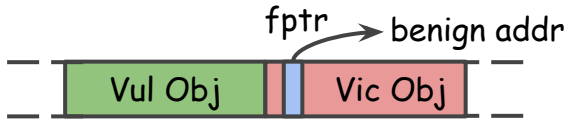
1. Which kernel object is useful for exploitation
   - similar size/same type to be allocated to the same cache as the vulnerable object
   - e.g, enclose ptr whose offset is within corruption range



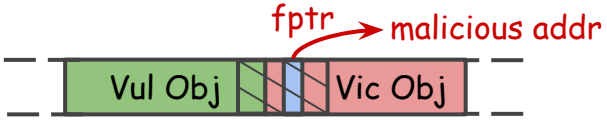
Allocate a **victim** object next to the **vulnerable** object

Challenges of Developing Exploits

1. Which kernel object is useful for exploitation
2. How to (de)allocate and dereference useful objects
   - System call sequence, arguments



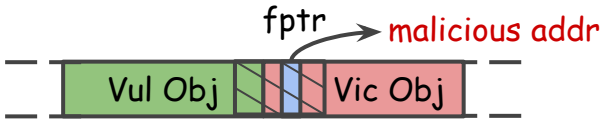
Allocate a victim object next to the vulnerable object



Dereference "fptr" to hijack control flow

Challenges of Developing Exploits

1. Which kernel object is useful for exploitation
2. How to (de)allocate and dereference useful objects
3. How to manipulate slab to reach desired layout
   - unexpected (de)allocation along with vulnerable/victim object makes side-effect to slab layout



Desired Slab Layout

Situation 1: Target slot is unoccupied

Situation 2: Target slot is occupied

Roadmap

Part I: Build A Kernel Object Database

- Include the kernel objects useful for exploitation and system calls and arguments that (de)allocate and dereference them (Challenge 1&2)

Part II: Adjust Slab Layout Systematically

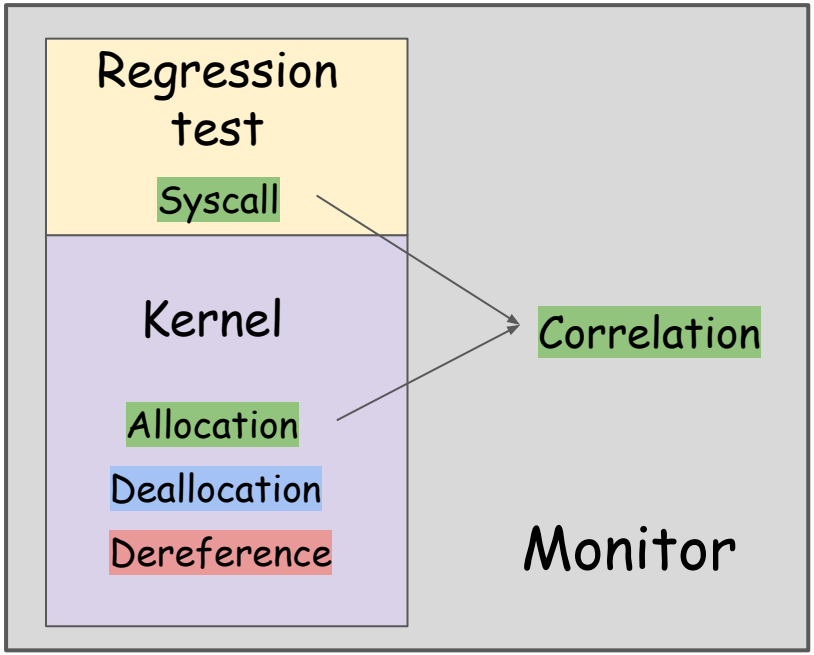
- Deal with unoccupied/occupied situations respectively (Challenge 3)

A Straightforward Solution to Challenges 1&2

Run kernel regression test

Monitor (de)allocation, dereference of objects in kernel

Correlate the object's operations to the system calls



This solution can't be directly applied to kernel.

Problems With the Straightforward Solution

Huge codebase

- # of objects is large while not all of them are useful
  e.g., in a running kernel, 109, 000 objects and 846, 000 pointers[4]
- Over 300 system calls with various combinations of arguments
- Complex runtime context and dependency between system calls

Asynchronous mechanism

- e.g, Read-Copy-Update (RCU) callback, dereference is registered first and triggered after a grace period

Multitask system

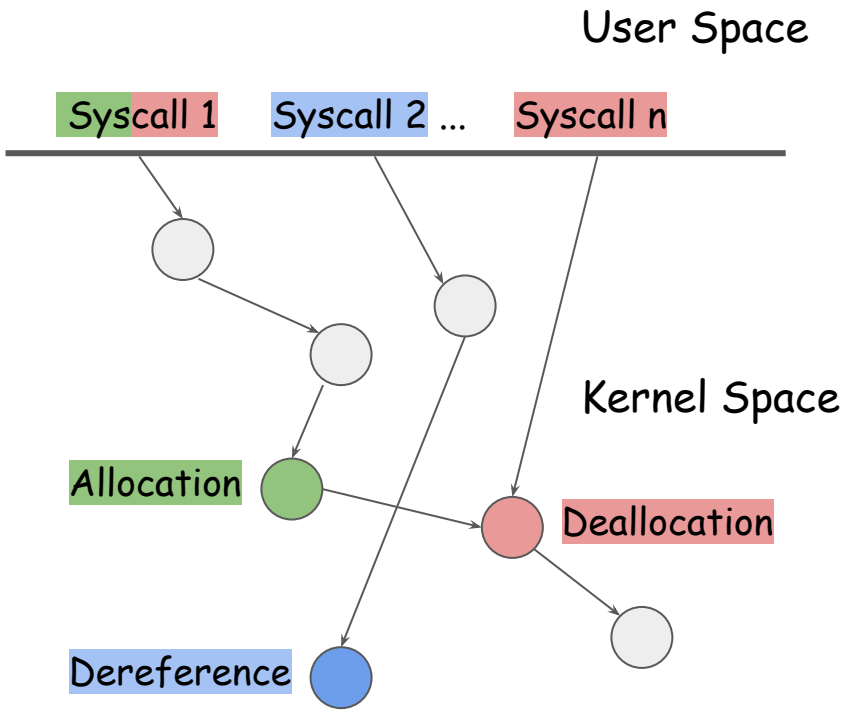
- Noise: other user-space processes, kernel threads, and hardware interrupts can also (de)allocate and dereference objects

[4] Back to the Whiteboard: a Principled Approach for the Assessment and Design of Memory Forensic Techniques, USENIX Security '19

Overview - Our Solution to Challenge 1&2

- Static Analysis to identify useful objects, sites of interest (allocation, deallocation, dereference), potential system calls

- Fuzzing Kernel to confirm system calls and complete arguments



Kernel Call Graph

Static Analysis - Useful Objects and Sites of Interest

Victim Object
- enclose a function pointer or a data object pointer
- once written, the adversaries can hijack control flow

Dereference Site
- indirect call
- asynchronous callback

```
struct file_operations {
        ...
        int (*llseek)(struct file*, loff_t, int);
        ...
}

struct file {
        ...
        const struct file_operations *f_op;
        ...
}

file->f_op->llseek(...);

kfree_rcu(...);
```

Static Analysis - Useful Objects and Sites of Interest

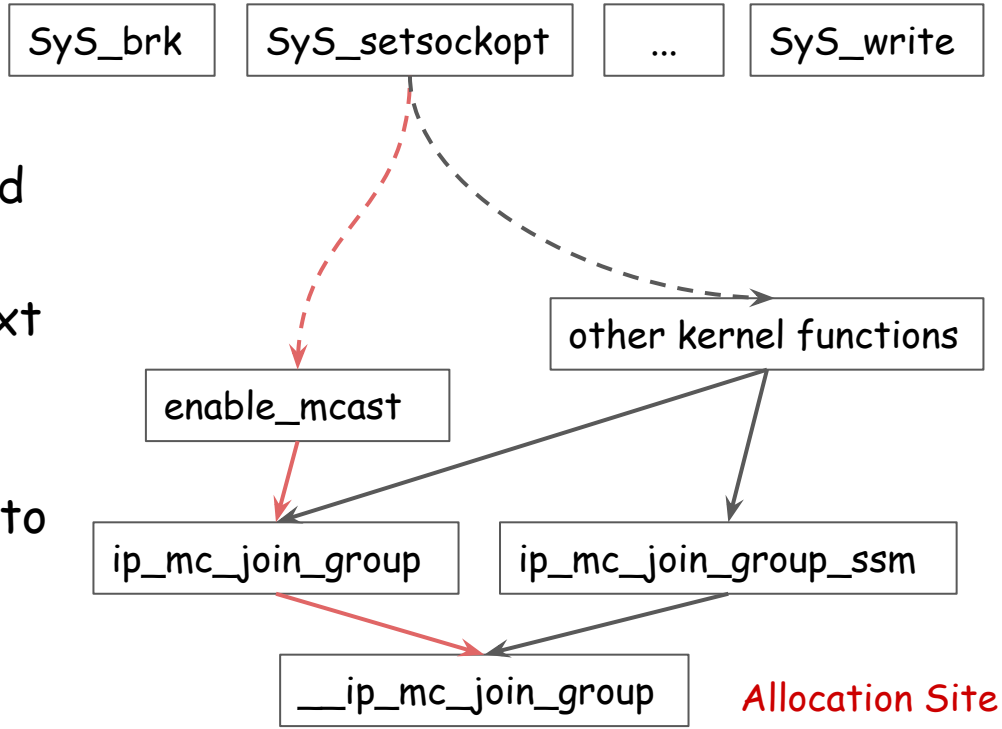
Spray Object
- most content can be controlled
- copy_from_user() migrates data from user space to kernel space

```
SYSCALL_DEFINE5(add_key, …, const void __user*,
        _payload, ...)
{
        …
        void* payload = kmalloc(plen, GFP_KERNEL);
        copy_from_user(payload, _payload, plen);
        …
}
```

Static Analysis - Potential System Calls

Reachable analysis over a customized type-matching kernel call graph

- delete function nodes in .init.text section
- delete call edges between independent modules according to KConfig
- add asynchronous callbacks to the graph



Kernel Call Graph

Kernel Fuzzing - Eliminate Noise

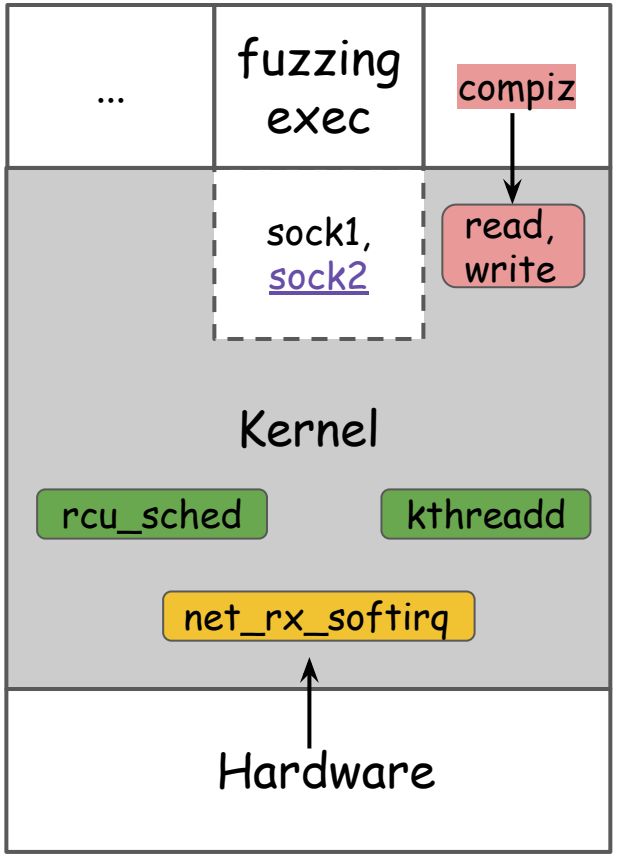
Instrument checking at sites of interest to eliminate following noises:

Source 1:

Objects of the same type from fuzzing executor sock2

Source 2:

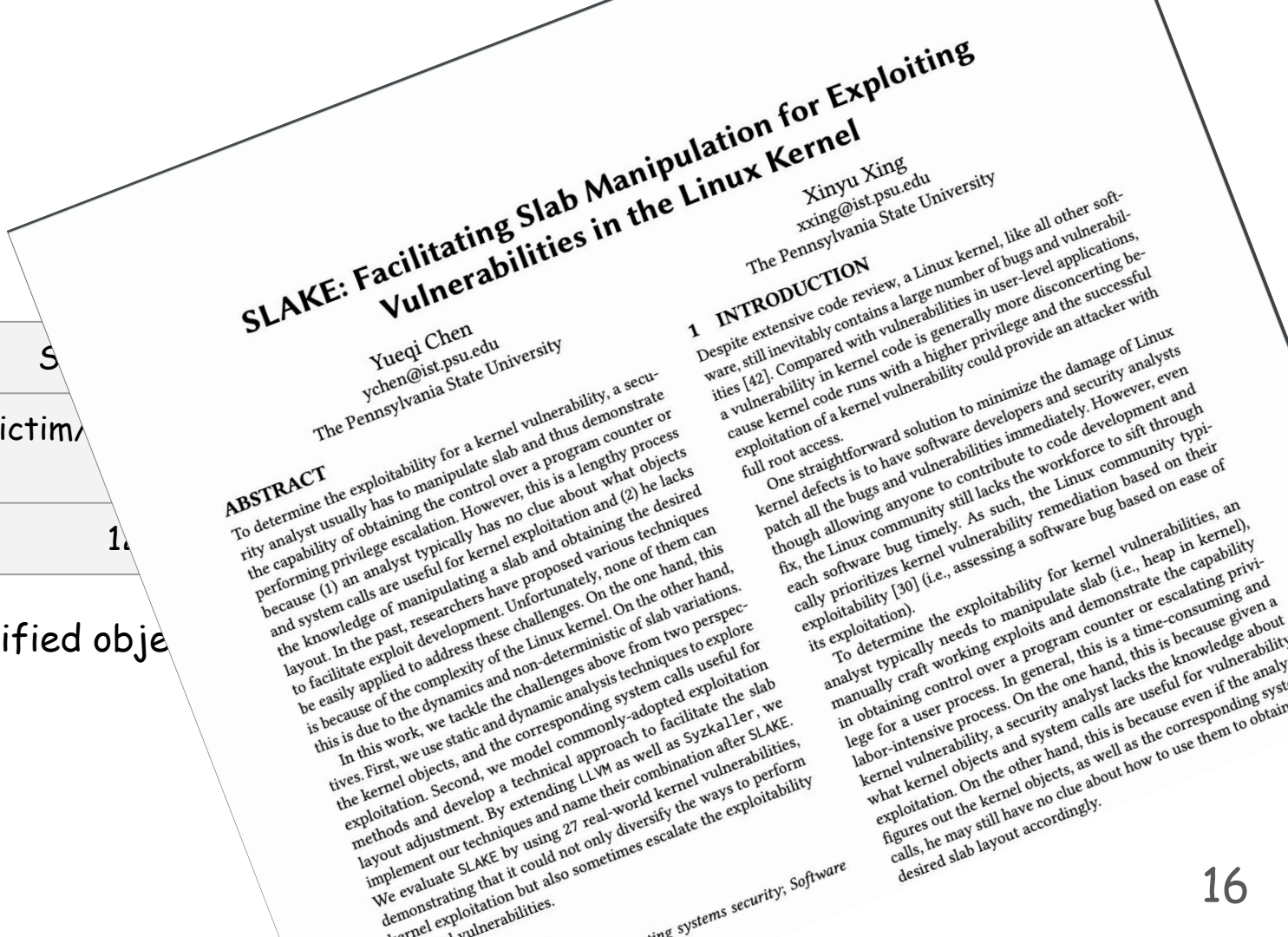
1. Other processes' syscalls read, write
2. Kernel threads rcu_sched kthreadd
3. Hardware interrupt net_rx_softirq



Evaluation

	S
	Victim/
Total	1

# of identified obje



SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel

Yueqi Chen
ychen@ist.psu.edu
The Pennsylvania State University

Xinyu Xing
xxing@ist.psu.edu
The Pennsylvania State University

ABSTRACT

To determine the exploitability for a kernel vulnerability, a security analyst usually has to manipulate slab and thus demonstrate the capability of obtaining the control over a program counter or performing privilege escalation. However, this is a lengthy process because (1) an analyst typically has no clue about what objects and system calls are useful for kernel exploitation and (2) he lacks the knowledge of manipulating a slab and obtaining the desired layout. In the past, researchers have proposed various techniques to facilitate exploit development. Unfortunately, none of them can be easily applied to address these challenges. On the one hand, this is because of the complexity of the Linux kernel. On the other hand, this is due to the dynamics and non-deterministic of slab variations. In this work, we tackle the challenges above from two perspectives. First, we use static and dynamic analysis techniques to explore the kernel objects, and the corresponding system calls useful for exploitation. Second, we model commonly-adopted exploitation methods and develop a technical approach to facilitate the slab layout adjustment. By extending LLVM as well as Syzkaller, we implement our techniques and name their combination after SLAKE. We evaluate SLAKE by using 27 real-world kernel vulnerabilities, demonstrating that it could not only diversify the ways to perform kernel exploitation but also sometimes escalate the exploitability

1 INTRODUCTION

Despite extensive code review, a Linux kernel, like all other software, still inevitably contains a large number of bugs and vulnerabilities [42]. Compared with vulnerabilities in user-level applications, a vulnerability in kernel code is generally more disconcerting because kernel code runs with a higher privilege and the successful exploitation of a kernel vulnerability could provide an attacker with full root access.

One straightforward solution to minimize the damage of Linux kernel defects is to have software developers and security analysts patch all the bugs and vulnerabilities immediately. However, even though allowing anyone to contribute to code development and fix, the Linux community still lacks the workforce to sift through each software bug timely. As such, the Linux community typically prioritizes kernel vulnerability remediation based on their exploitability [30] (i.e., assessing a software bug based on ease of its exploitation).

To determine the exploitability for kernel vulnerabilities, an analyst typically needs to manipulate slab (i.e., heap in kernel), manually craft working exploits and demonstrate the capability in obtaining control over a program counter or escalating privilege for a user process. In general, this is a time-consuming and labor-intensive process. On the one hand, this is because given a kernel vulnerability, a security analyst lacks the knowledge about what kernel objects and system calls are useful for vulnerability exploitation. On the other hand, this is because even if the analyst figures out the kernel objects, as well as the corresponding system calls, he may still have no clue about how to use them to obtain desired slab layout accordingly.

ing systems security, Software

Roadmap

Part I: Build A Kernel Object Database

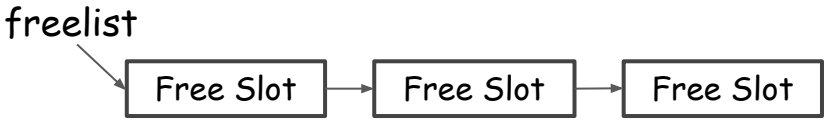
- Include the kernel objects useful for exploitation and system calls and arguments that (de)allocate and dereference them (Challenge 1&2)

Part II: Adjust Slab Layout Systematically

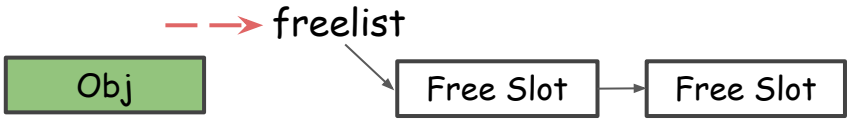
- Deal with unoccupied/occupied situations respectively (Challenge 3)

Working Fashion of SLAB/SLUB allocator

A single list organizes free slots



Allocation
retrieve from the freelist head

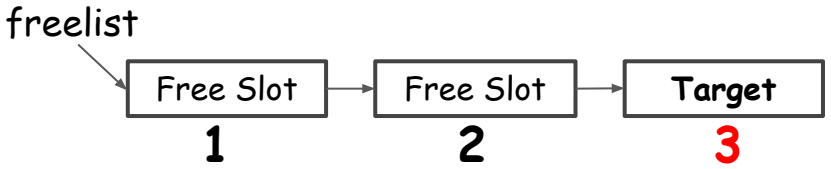


Deallocation
recycle to the freelist head

Both allocation and deallocation are at the freelist head

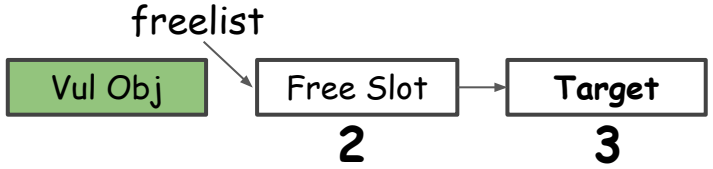
Situation 1: Target Slot is Unoccupied

Initial State



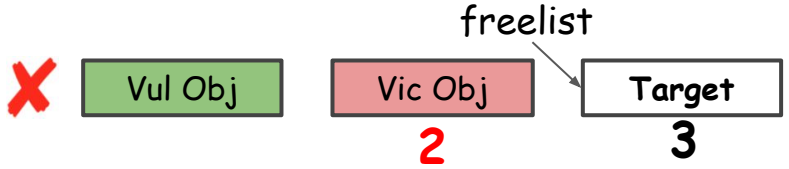
1

1. Allocate the Vul Obj



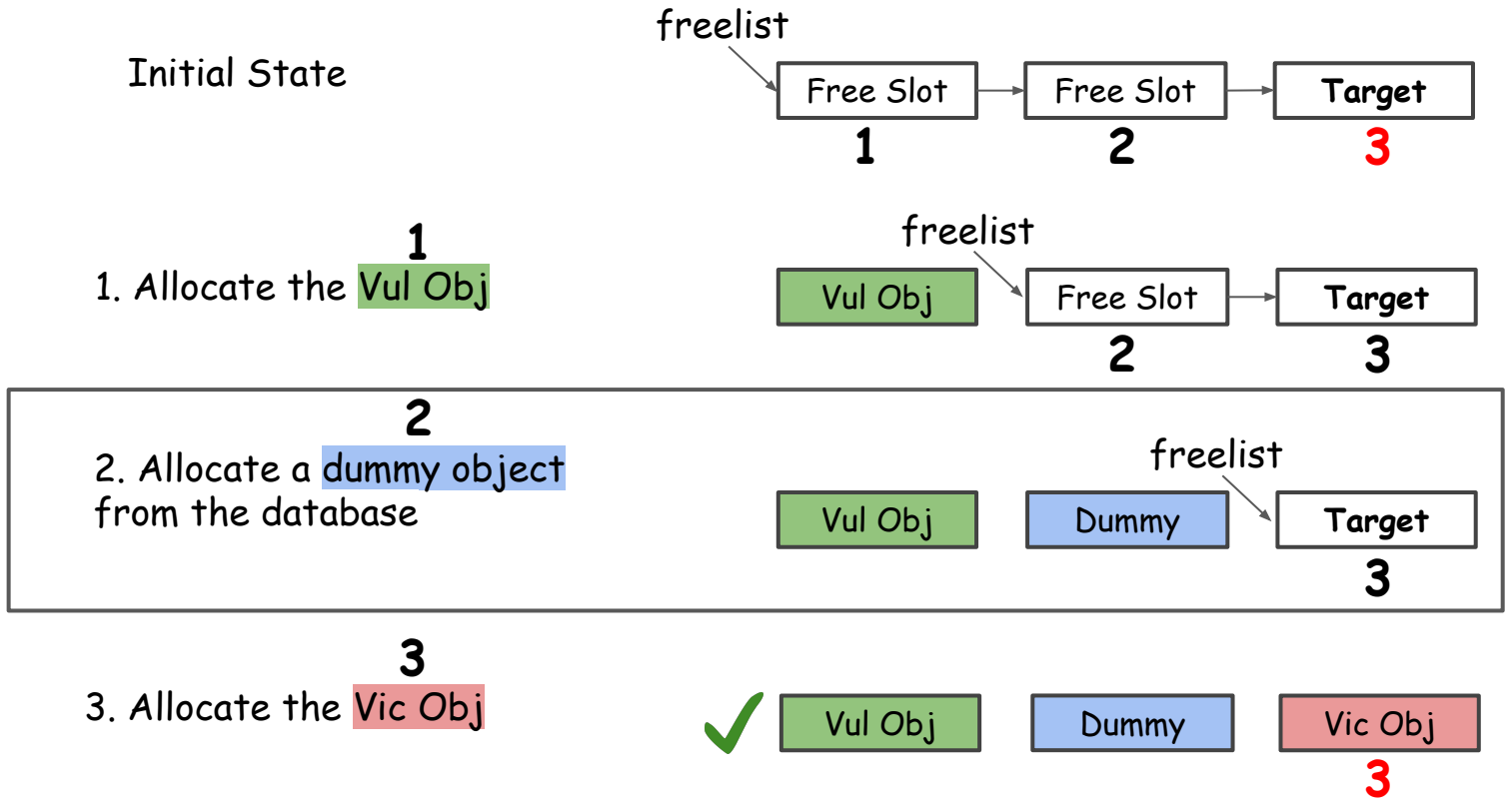
2

2. Allocate the Vic Obj

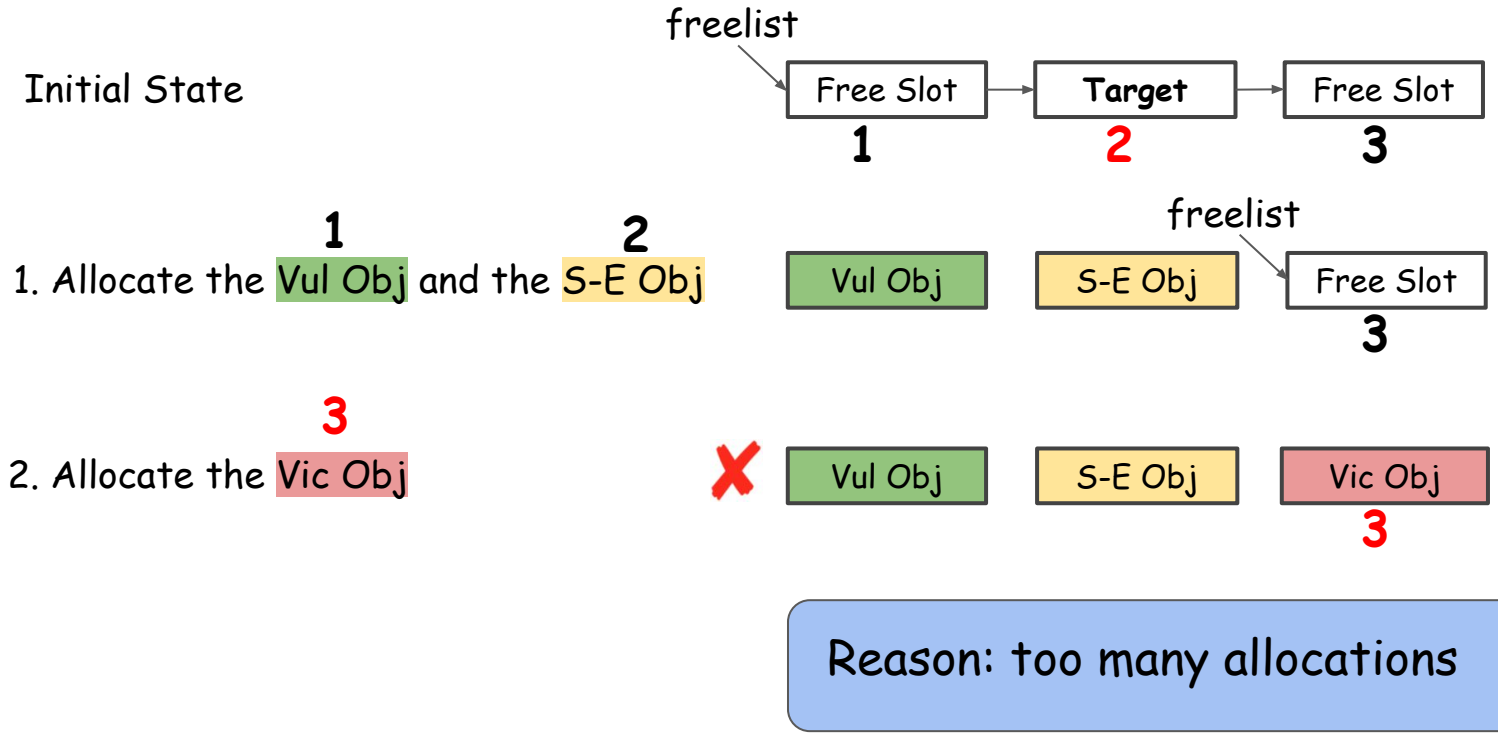


Reason: too few allocations

Situation 1: Our Solution

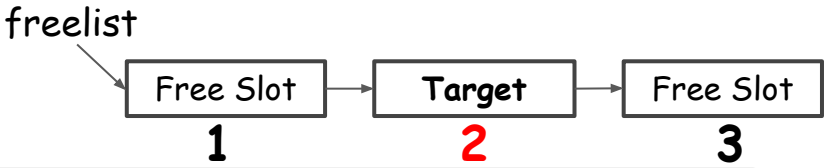


Situation 2: Target Slot is Occupied



Situation 2: Straightforward But Wrong Solution

Initial State



Problems with straightforward solution
- No general syscalls and arguments for deallocation
- Vul Obj can also be freed along with the S-E Obj

1. Al

-1

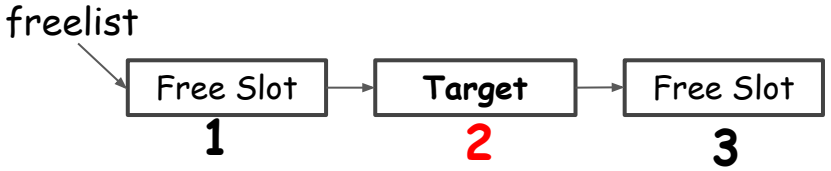
2. Deallocate the S-E Obj

3-1 = 2

3. Allocate the Vic Obj

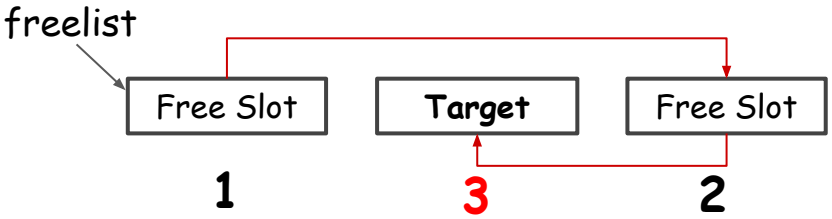
Situation 2: Our Solution

Initial State



1. Allocate three dummy objects

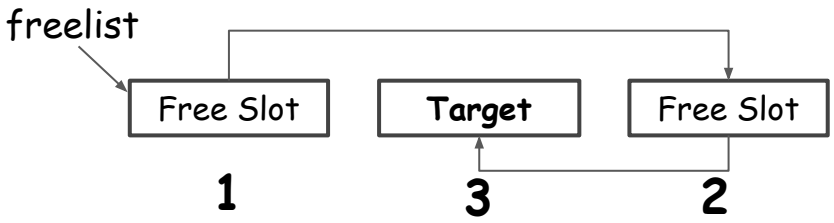
2. Deallocate the dummy object in the order **2nd**, **3rd**, **1st**



Our solution is to reorganize the freelist, switching the target slot's order from 2nd to 3rd

Situation 2: Our Solution (cont.)

New Initial State



1. Allocate the Vul Obj (1) and the S-E Obj (2)

2. Allocate the Vic Obj (3)

Evaluation Set

27 vulnerabilities (the largest evaluation set so far)

- 26 CVEs, 1 Wild
- 13 UAF, 4 Double Free, 10 Slab Out-of-bound Write
- 18 with public exploits, 9 with NO public exploits

Evaluation Results

18 cases with public exploits
- 15 successful cases
- 8 additional unique exploits on avg.

SLAKE diversifies the ways to exploitation

9 cases with NO public exploits
- 3 successful cases
- 25 unique exploits in total

SLAKE potentially escalates exploitability

Evaluation Results (cont.)

9 failure cases

- 6 cases, PoC manifests limited capability

Future work: continue exploring more capability of security bugs

- 3 cases, vulnerability is in special caches

Future work: include more modules for analysis

Summary & Conclusion

SLAKE

1. Identifies objects useful for kernel exploitation
2. Reorganizes slab and obtains the desired layout

SLAKE is able to

1. Empower the capability of developing working exploits
2. Potentially escalate exploitability and benefit its assessment for Linux kernel bugs

Thank You

Code & Data

https://github.com/chenyueqi/SLAKE

Contact

Twitter: @Lewis_Chen_

Email: ychen@ist.psu.edu

Personal Page: http://www.personal.psu.edu/yxc431/

Misc: **Looking for 2020 summer internship**