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/
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

Workflow of Determining Exploitability

PoC: Slab-out-of-bound write

Step 1: Allocate a victim object next to the vulnerable object

Step 2: Trigger the security bug to tamper “fptr”

Step 3: Dereference “fptr” to hijack control flow

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

![Diagram showing vulnerability object (Vul Obj) and victim object (Vic Obj) with fptr and benign addr connections.](image-url)
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

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
   - Similar size/same type to be allocated to the same cache as the vulnerable object
     - Enclose ptr whose offset is within corruption range

2. How to (de)allocate and dereference useful objects
   - System call sequence, arguments

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

Side-effect
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

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
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

```c
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

SyS_brk  SyS_setsockopt  ...  SyS_write

enable_mcast
ip_mc_join_group  ip_mc_join_group_ssm
__ip_mc_join_group

Allocation Site
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, kthread
3. Hardware interrupt net_rx_softirq
SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel

Yanci Chen
ychen@psu.edu
The Pennsylvania State University

ABSTRACT

To determine the exploitability for a kernel vulnerability, a security analyst usually has to manipulate slab and then demonstrate the capability of obtaining the control over a program counter or the capability of privilege escalation. However, this is a lengthy process and performing privilege escalation. Furthermore, this analysis typically has no clue about what objects become the victims or the spray objects. To handle these challenges, we have proposed various techniques to facilitate the slab manipulation. In this work, we model commonly-adopted slab allocation and a technique to facilitate slab manipulation.

We implement our techniques and analyze their combinations after SLAKE. We evaluate SLAKE by using 27 real kernel vulnerability data sets. We demonstrate that it could not only effectively perform kernel exploitation but also sometimes escalate the exploitability of the vulnerabilities.

Xinyu Xing
sxing@psu.edu
The Pennsylvania State University

1 INTRODUCTION

Despite extensive code review, a Linux kernel, like all other software, still inevitably contains a large number of bugs and vulnerabilities [4]. Compared with vulnerabilities in user-level applications, a vulnerability in kernel code is generally more devastating because kernel codes run 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 quickly patch all the bugs and vulnerabilities immediately. However, even though allowing anyone to contribute to code development and patching all the bugs and vulnerabilities, the current community still lacks the workforce to sift through the Linux community typically fixes the software bugs very timely. As such, the Linux community typically fixes the softw
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. Allocate the **Vul Obj**

2. Allocate the **Vic Obj**

Reason: too few allocations
Situation 1: Our Solution

Initial State

1. Allocate the **Vul Obj**

2. Allocate a **dummy object** from the database

3. Allocate the **Vic Obj**
**Situation 2: Target Slot is Occupied**

1. Allocate the **Vul Obj** and the **S-E Obj**

2. Allocate the **Vic Obj**

**Reason:** too many allocations
Situation 2: Straightforward But Wrong Solution

1. Allocate the Vul Obj and the S-E Obj

2. Deallocate the S-E Obj

3. Allocate the Vic Obj

Problems with straightforward solution:
- No general syscalls and arguments for deallocation
- can also be freed along with the Vul Obj
Situation 2: Our Solution

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** and the **S-E Obj**

2. Allocate the **Vic Obj**

```
Free Slot          Target          Free Slot
  1               3               2
freelist

Vul Obj               Target               S-E Obj
  3
Vul Obj               Vic Obj              S-E 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