black hat EUROPE 2019

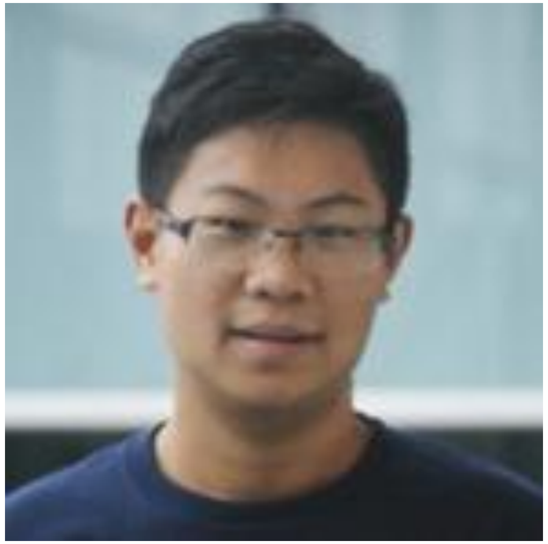
DECEMBER 2-5, 2019

EXCEL LONDON, UK

Hands Off and Putting SLAB/SLUB Feng Shui in Blackbox

Yueqi (Lewis) Chen

Who We Are



Yueqi Chen @Lewis_Chen_
- Ph.D. Student, Pennsylvania State University
- **Looking for 2020 Summer internship**

Xinyu Xing
- Assistant Professor, Pennsylvania State University
- Visiting Scholar, JD.com

Jimmy Su
- Senior Director, JD Security Research Center in Silicon Valley

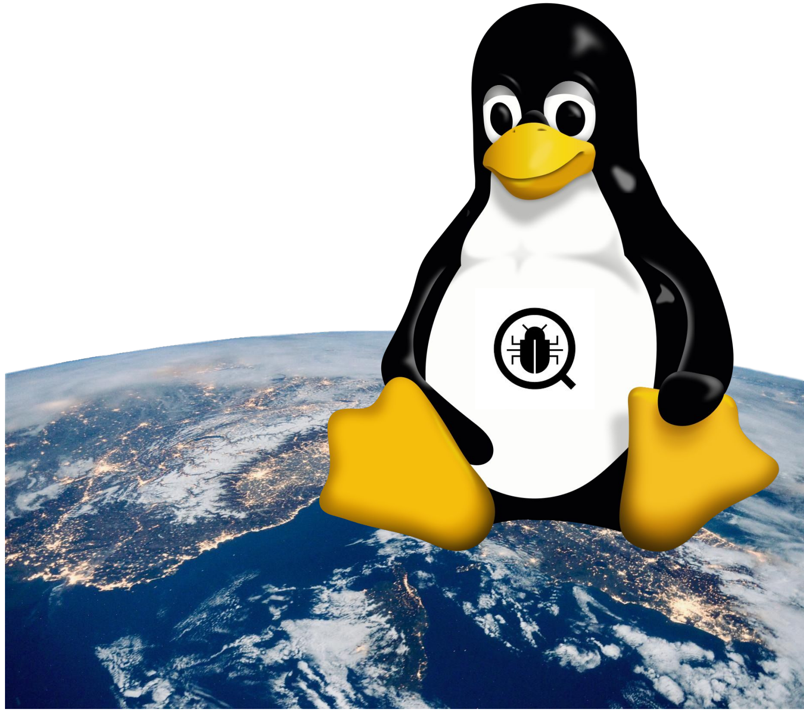
PennState

JD.COM

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

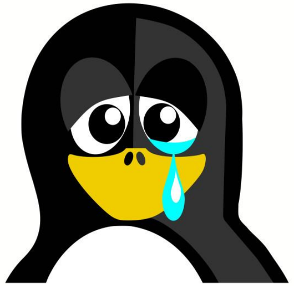


[1] SLTS project, https://lwn.net/Articles/749530/
[2] "Syzbot and the Tale of Thousand Kernel Bugs" - Dmitry Vyukov, Google

Harsh Reality: Cannot Patch All Bugs Immediately

Google Syzbot[3] , on Nov 24th

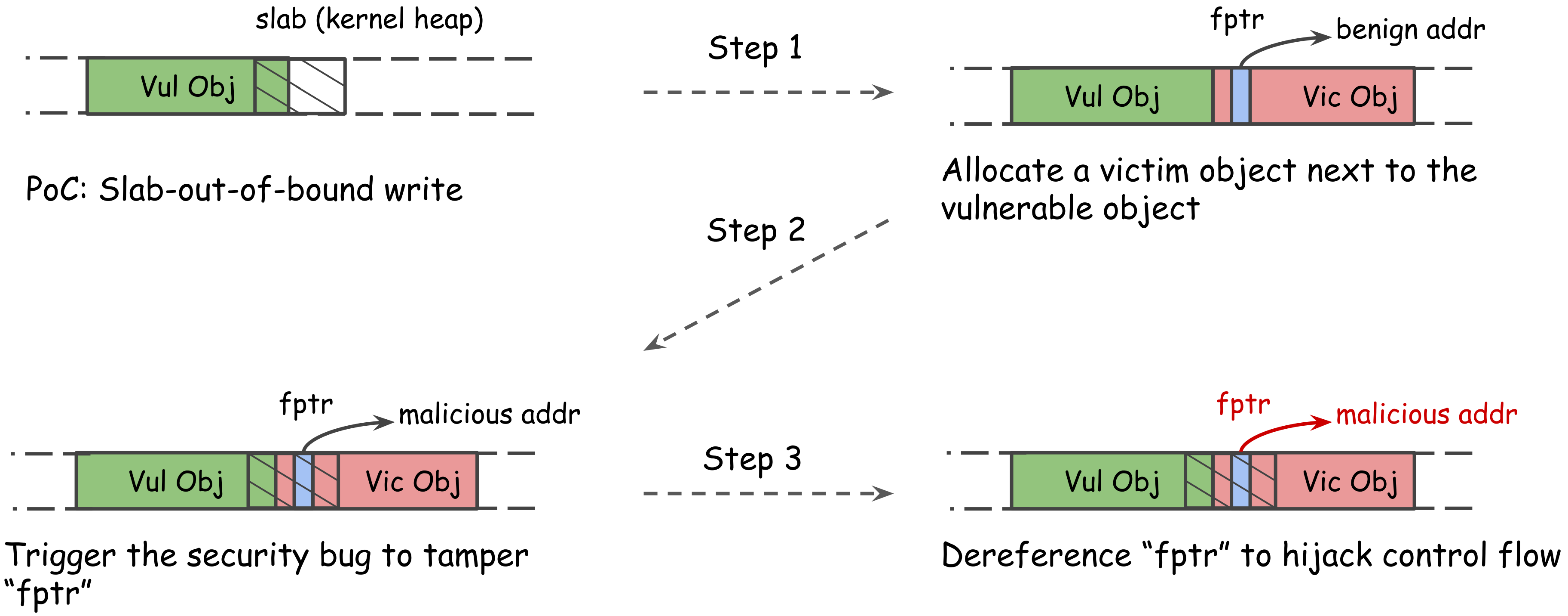
- 459 not fixed, 92 fix pending, 55 in moderation
- # of bug reports increases 200 bugs/month



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

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

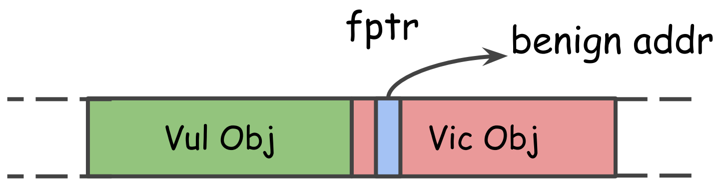
Workflow of Determine Exploitability



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

Challenges of Developing Exploits

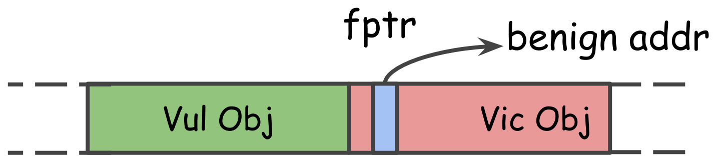
1. Which kernel object is useful for exploitation
   - similar size/same type to be allocated to the same cache as the vulnerable object
   - e.g, enclose ptr whose offset is within corruption range



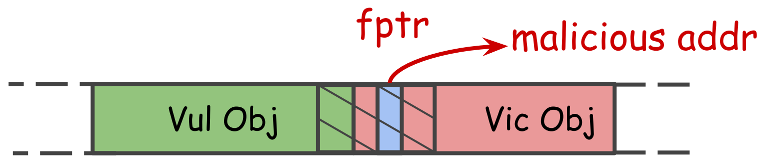
Allocate a **victim** object next to the **vulnerable** object

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



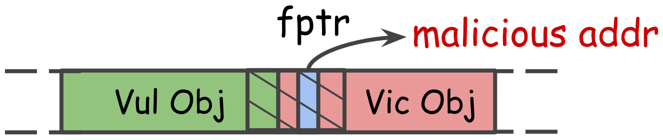
Allocate a victim object next to the vulnerable object



Dereference "fptr" to hijack control flow

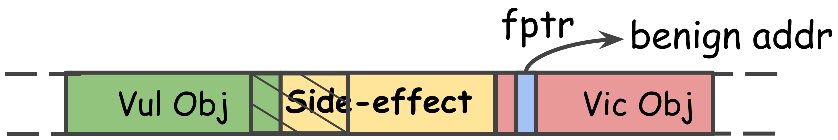
Challenges of Developing Exploits

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)

Part III: Tricks

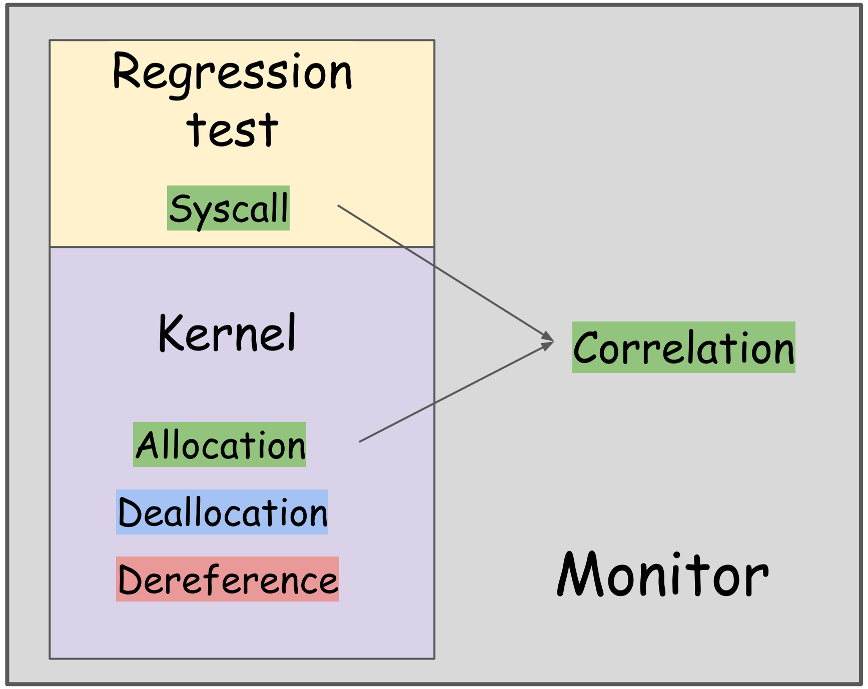
- Create an initial slab cache
- Calculate side-effect layout
- Shorten exploitation window

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

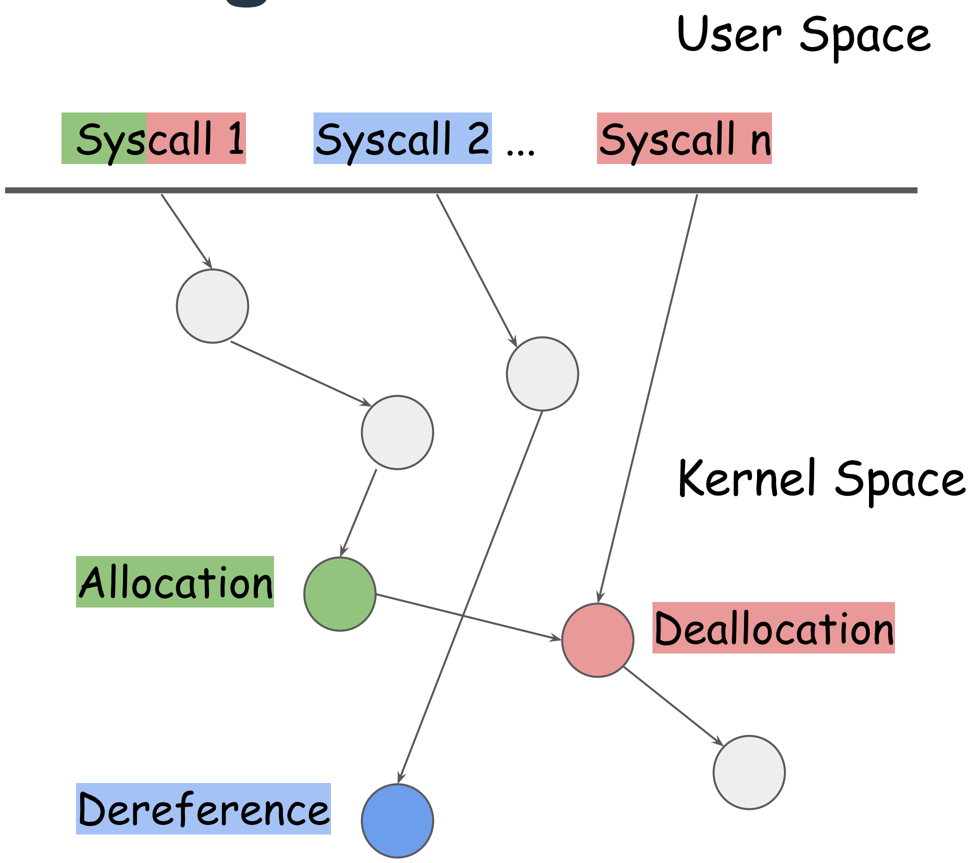
Multitask system

- 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

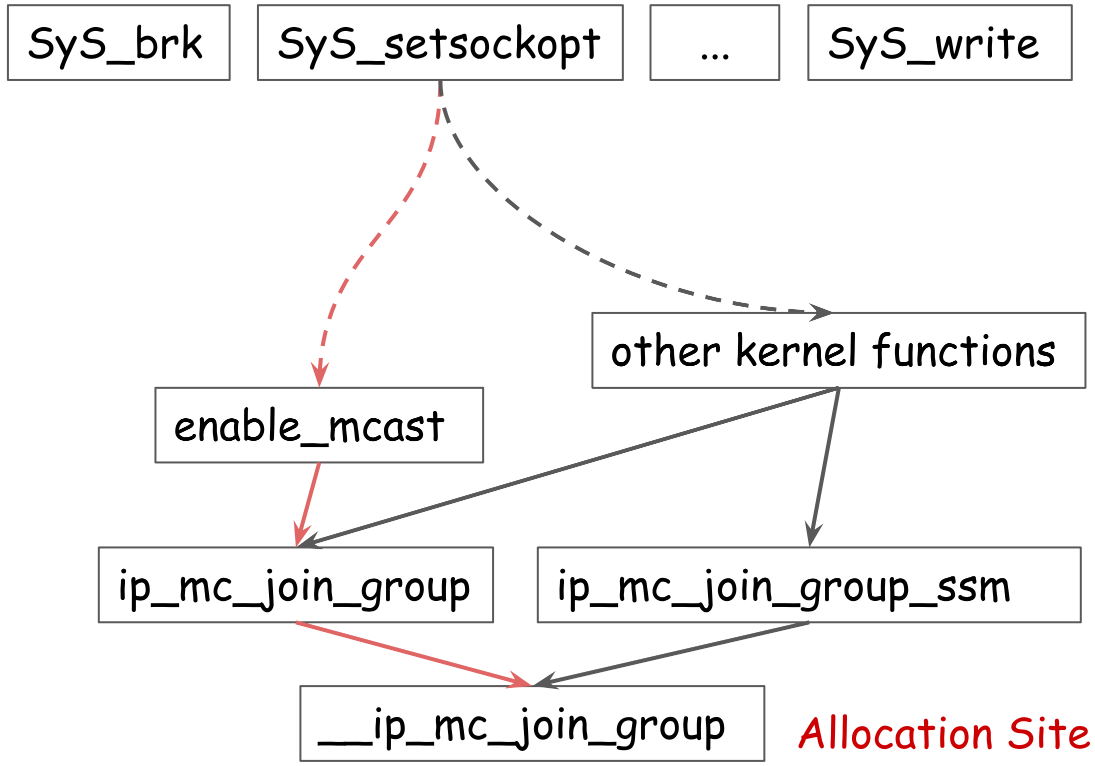
Spray Object
- most content can be controlled
- copy_from_user() migrates data from user space to kernel space

```
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



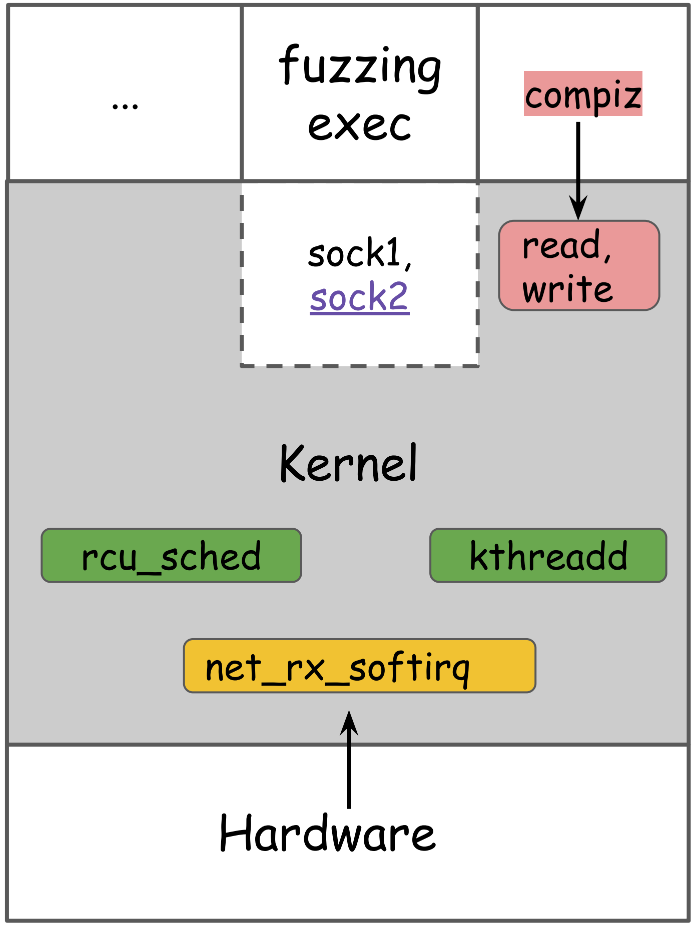
Kernel Call Graph

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

	Static Analysis	Kernel Fuzzing		
	Victim/Spray Object	Victim Object (alloc/dealloc/deref)	Spray Object	Avg. time (min)
Total	124/4	75/20/29	4	2

# of identified objects/syscalls (v4.15, defnoconfig + 32 other modules)

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)

Part III: Tricks

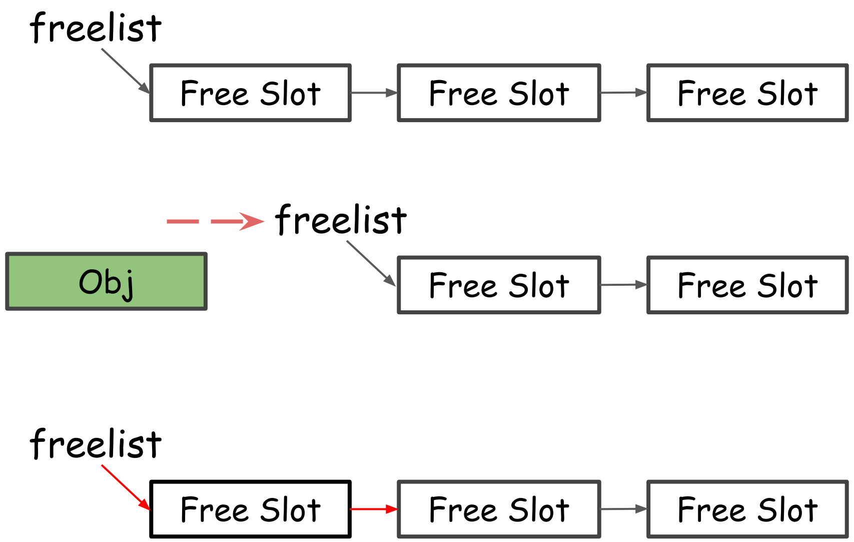
- Create an initial slab cache
- Calculate side-effect layout
- Shorten exploitation window

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

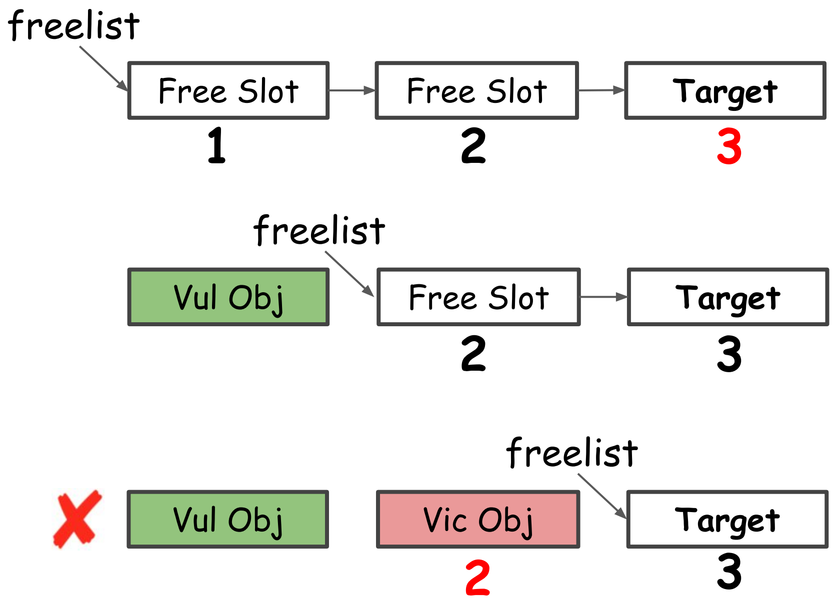
Initial State

1

1. Allocate the Vul Obj

2

2. Allocate the Vic Obj



Reason: too few allocations

Situation 1 - Our Solution

Initial State

1

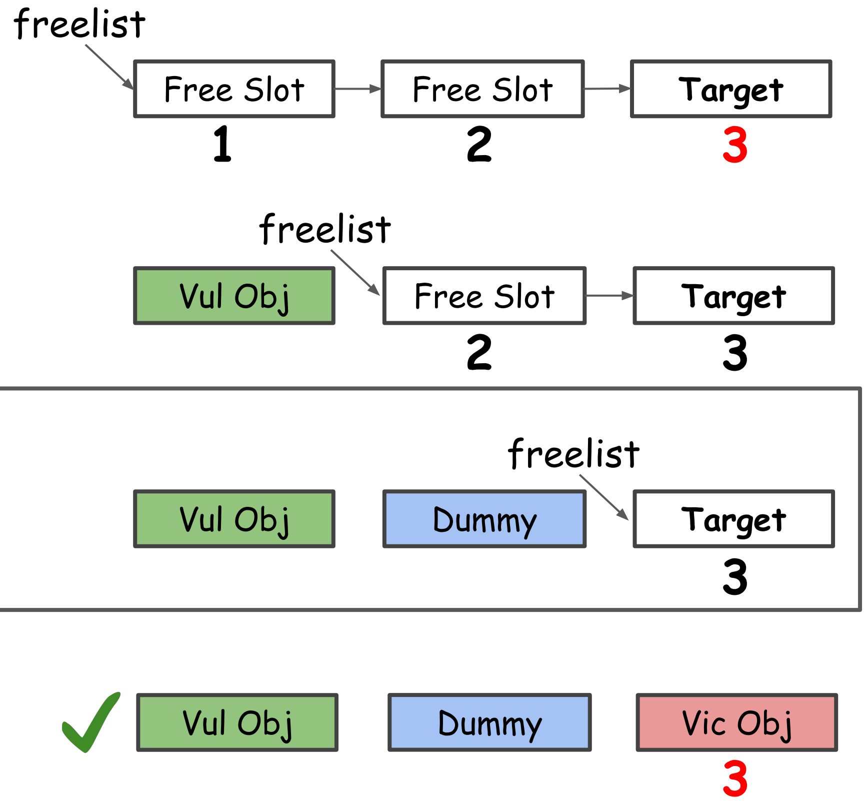
1. Allocate the Vul Obj

2

2. Allocate a dummy object from the database

3

3. Allocate the Vic Obj

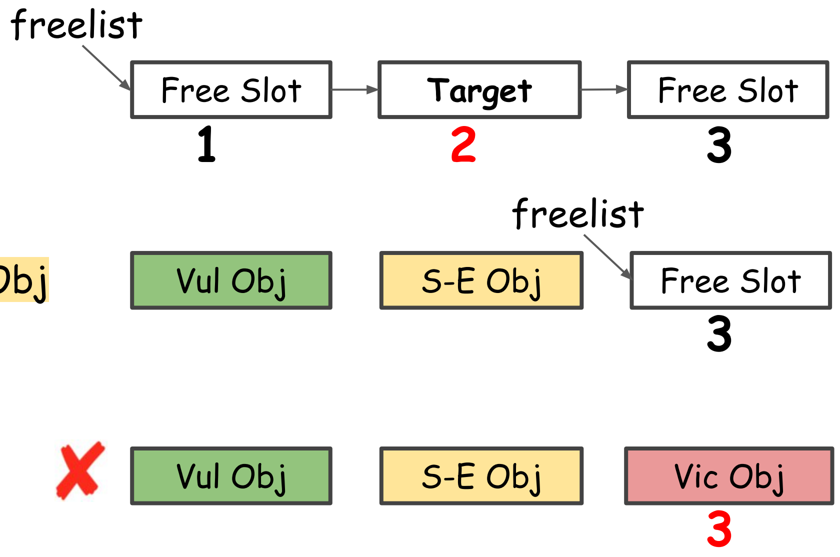


Situation 2 - Target Slot is Occupied

Initial State

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

2. Allocate the Vic Obj (3)



Reason: too many allocations

Situation 2 - Straightforward But Wrong Solution

Initial State

freelist → Free Slot (1) → Target (2) → Free Slot (3)

1. Al

Problems with straightforward solution

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

-1

2. Deallocate the S-E Obj

Vul Obj | freelist → Target (2) → Free Slot (3)

3-1 = 2

3. Allocate the Vic Obj

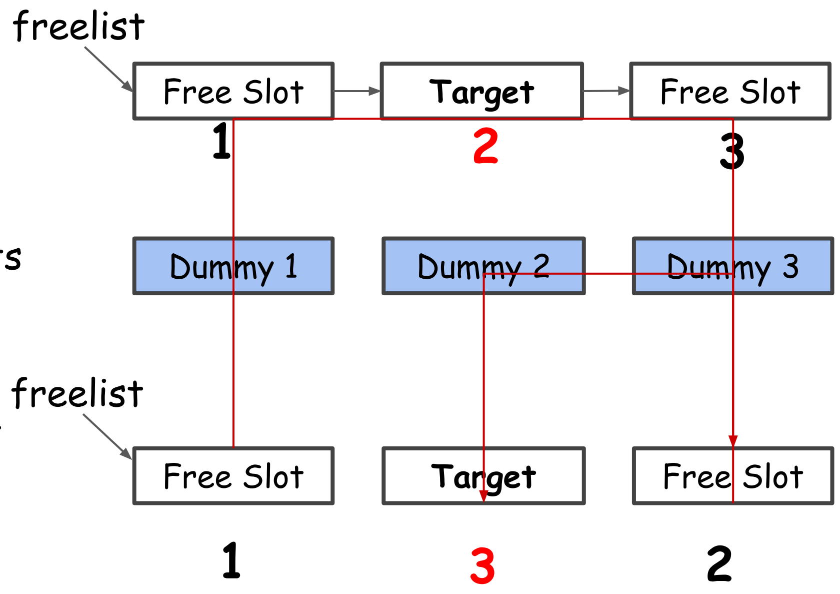
✓ Vul Obj | Vic Obj (2) | freelist → Free Slot (3)

Situation 2 - Our Solution

Initial State

1. Allocate three dummy objects

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



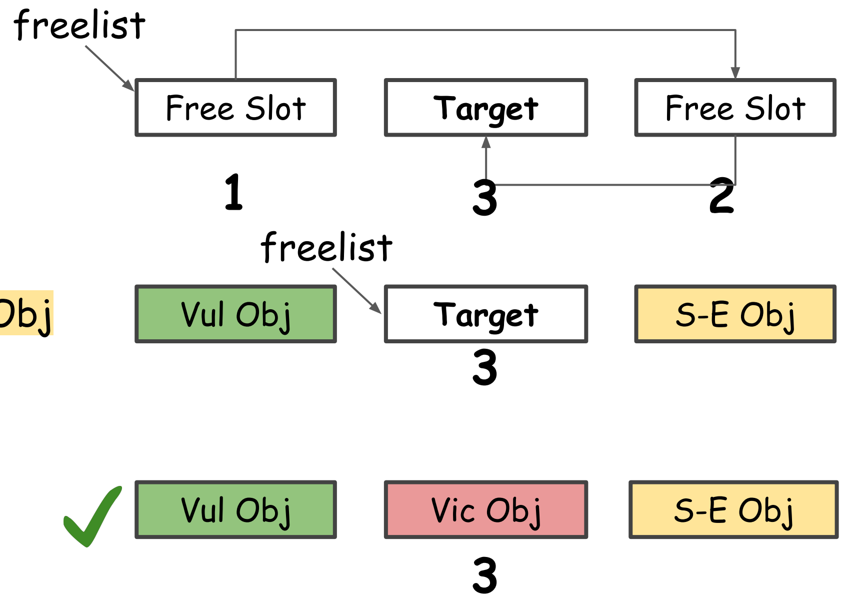
Our solution is to reorganize the freelist, switching the target slot's order from 2nd to 3rd

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.

Diversify the ways to exploitation

9 cases with NO public exploits

- 3 successful cases
- 25 unique exploits in total

Potentially escalate 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

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)

➡ Part III: Tricks

- Create an initial slab cache
- Calculate side-effect layout
- Shorten exploitation window

Tricks

- Create an initial slab cache
  - so that slots are chained sequentially
  - defragmentation
- Calculate side-effect layout
  - ftrace logs calling to allocation/deallocation
  - analyze log to calculate layout before manipulation
- Shorten exploit window
  - to minimize influence of other kernel activities on layout
  - put critical operation after defragmentation

Summary & Conclusion

Summary:

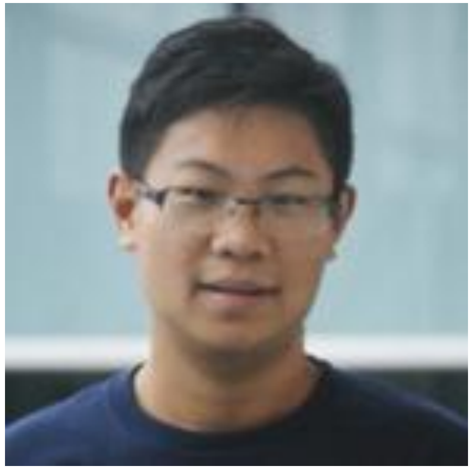
1. Identifies objects useful for kernel exploitation
2. Reorganizes slab and obtains the desired layout

Conclusion:

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

DEMO

Thank You !



Yueqi Chen
Twitter: @Lewis_Chen_
Email: ychen@ist.psu.edu
Personal Page: http://www.personal.psu.edu/yxc431

Looking for 2020 summer internship