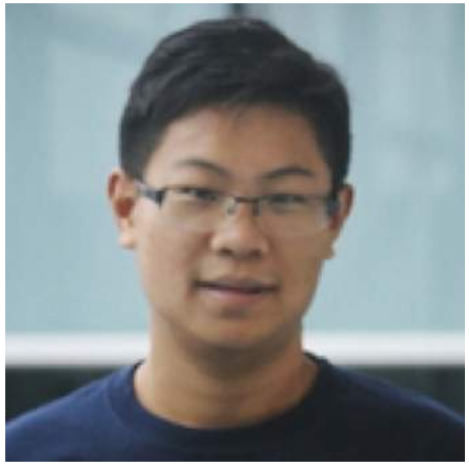
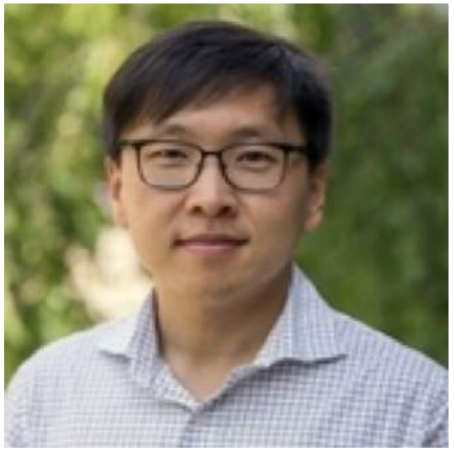

black hat
EUROPE 2019
DECEMBER 2-5, 2019
EXCEL LONDON, UK
Hands Off and Putting SLAB/SLUB Feng Shui in Blackbox
Yueqi (Lewis) Chen
#BHEU @BLACK HAT EVENTS

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

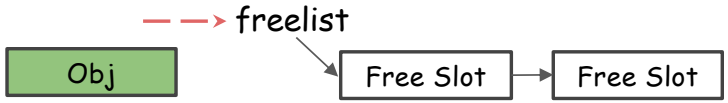
PennState

JD.COM

Working Fashion of SLAB/SLUB Allocator

A single list organizes free slots

Allocation
retrieve from the freelist head

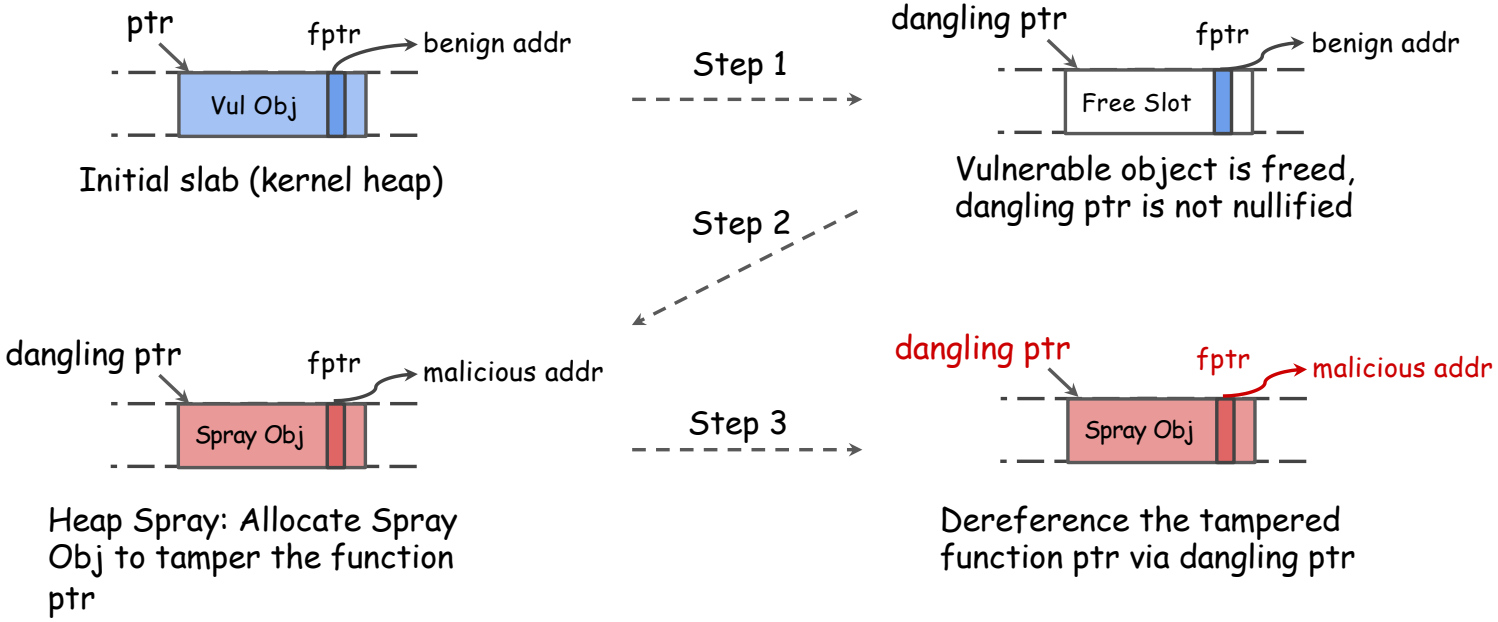


Deallocation
recycle to the freelist head

Both allocation and deallocation are at the freelist head

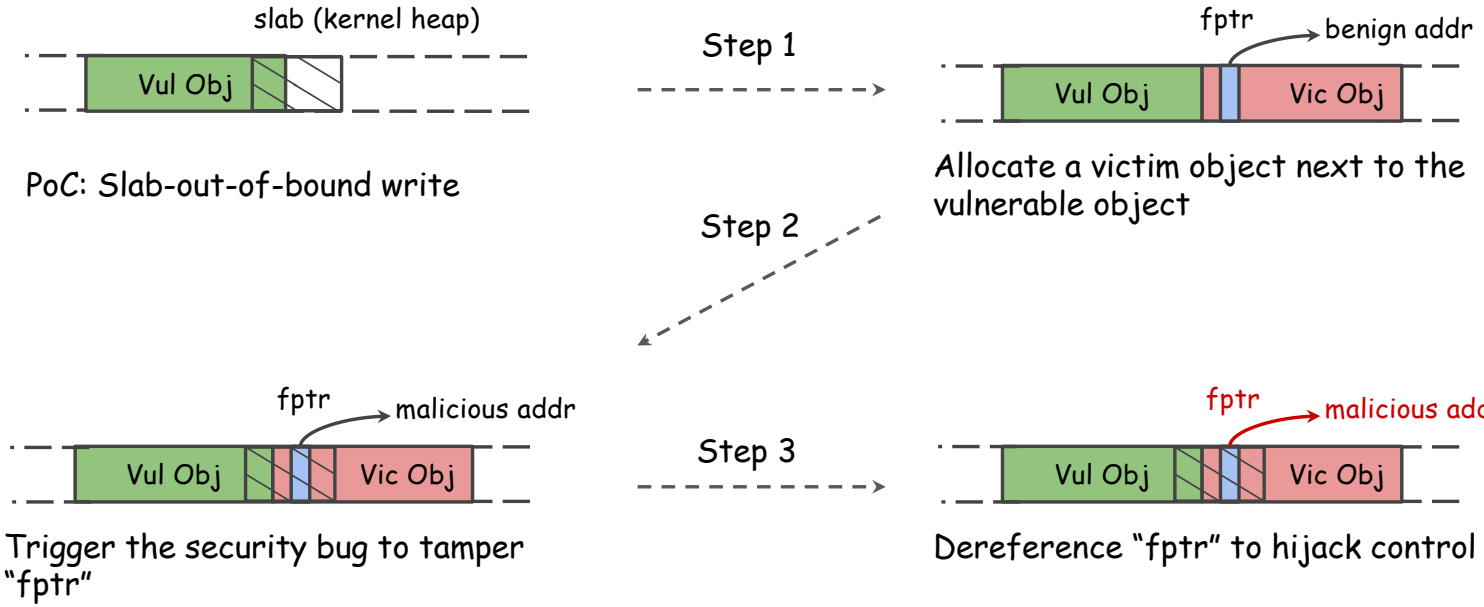
Highly simplified, not entirely correct

Workflow of Use-After-Free Exploitation



Example: Exploit A Use-After-Free in Three Steps

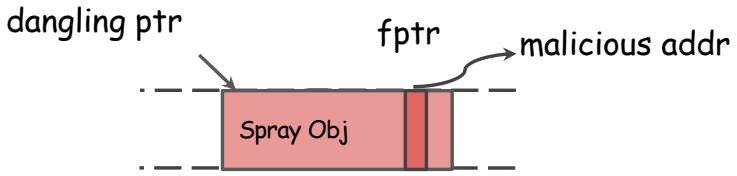
Workflow of Slab Out-of-bound Write Exploitation



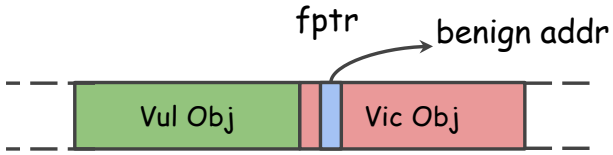
Example: Exploit A Slab Out-of-bound Write in Three Steps

Challenges of SLAB/SLUB Fengshui

1. Which kernel object is useful for exploitation
   - victim object, vulnerable object, spray object: similar size/same type to be allocated to the same cache
   - victim object encloses ptr whose offset is within corruption range
   - spray object's content is controllable



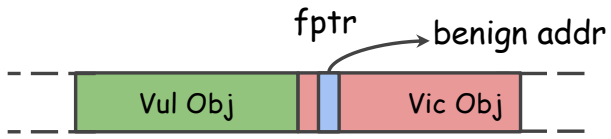
Heap Spray: Allocate **spray obj** to tamper the function ptr



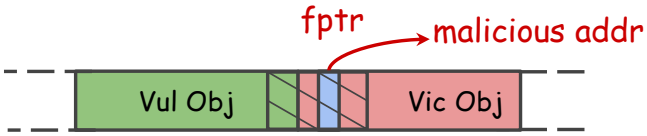
OOB: Allocate a **victim** object next to the **vulnerable** object

Challenges of SLAB/SLUB Fengshui

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



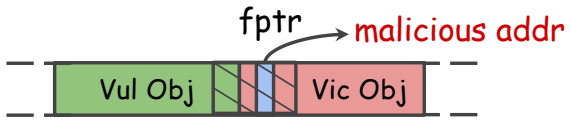
Allocate a victim object next to the vulnerable object



Dereference "fptr" to hijack control flow

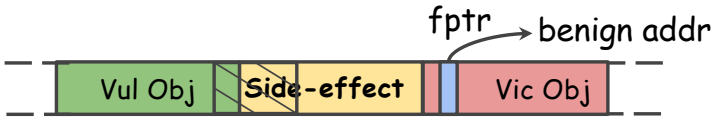
Challenges of SLAB/SLUB Fengshui

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/spray 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

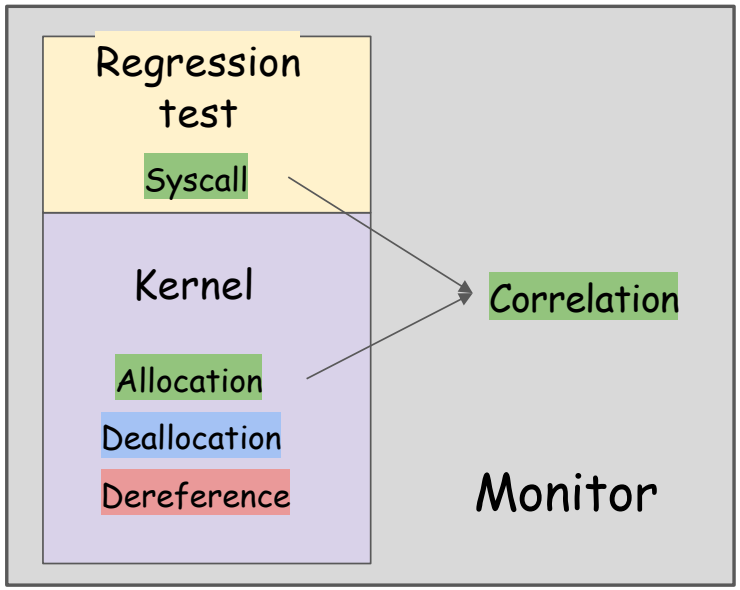
- Create an initial slab cache
- Calculate side-effect layout
- Shorten exploitation window

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

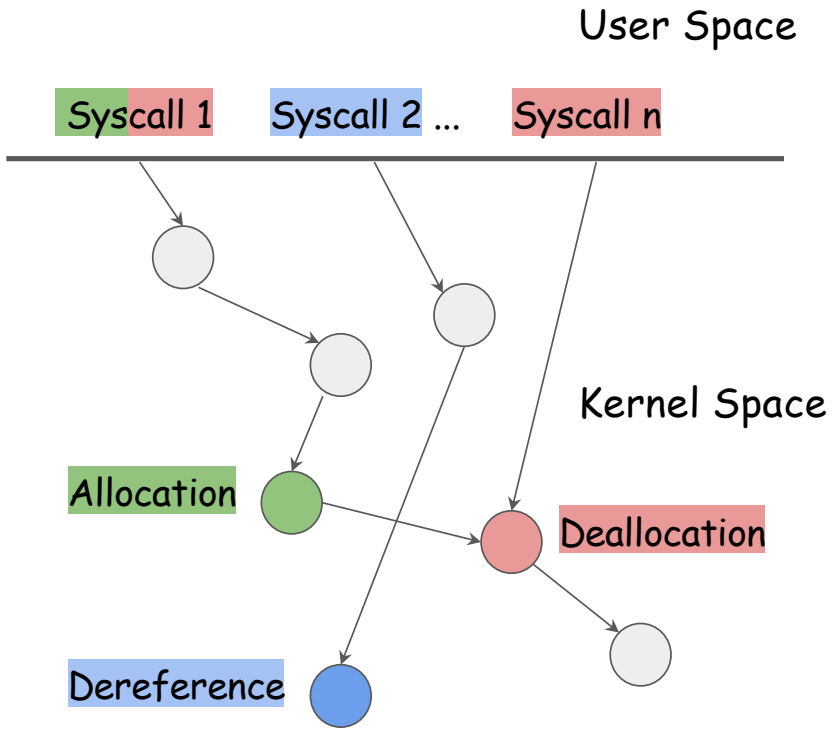
Multitask system

- 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

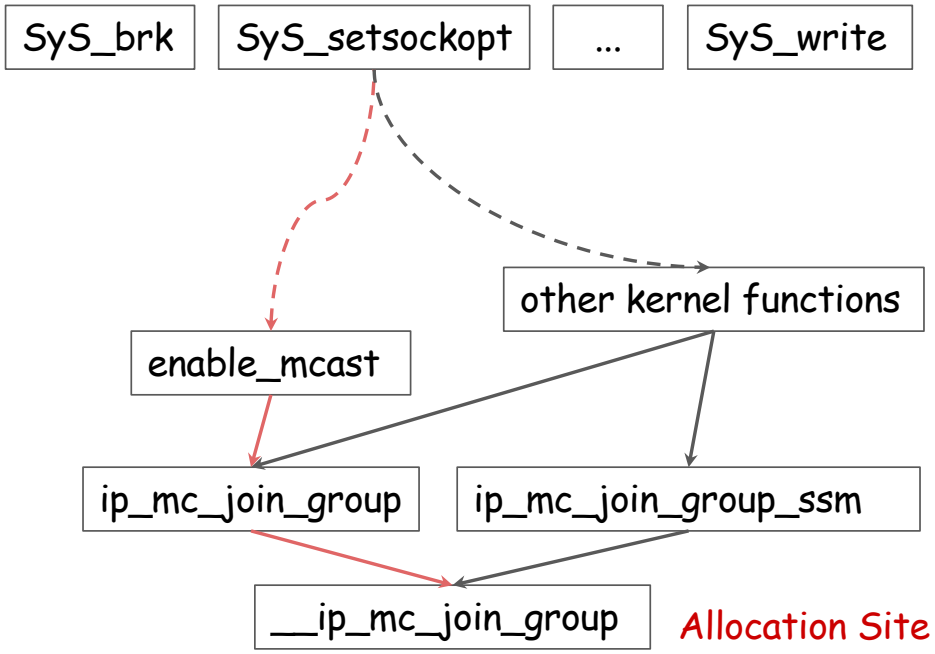
Spray Object
- most content can be controlled
- copy_from_user() migrates data from user space to kernel space

```
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

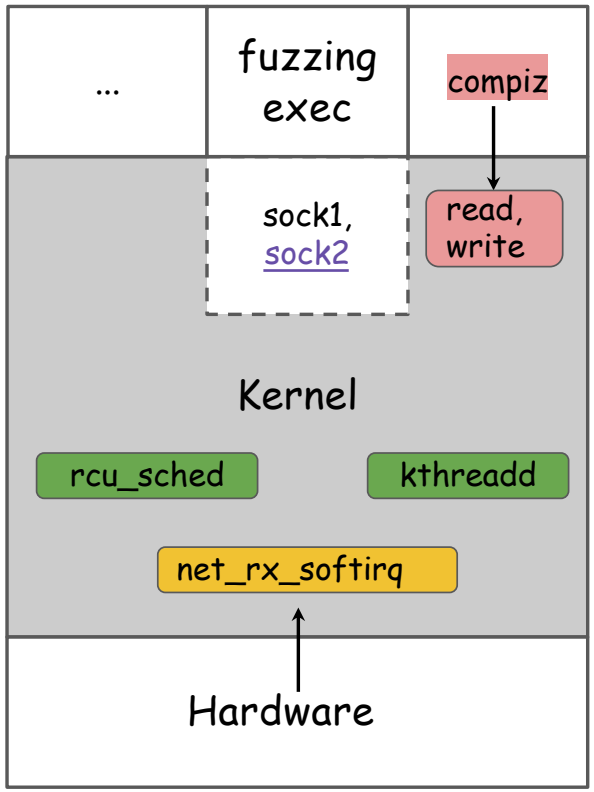
Kernel Fuzzing - Eliminate Noise

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

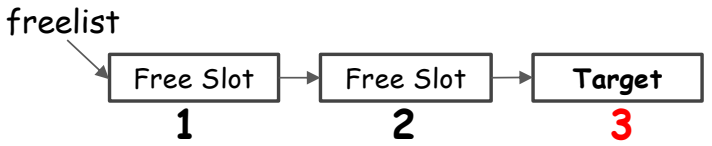
- Create an initial slab cache
- Calculate side-effect layout
- Shorten exploitation window

Layout Adjustment Involves Many Possibilities

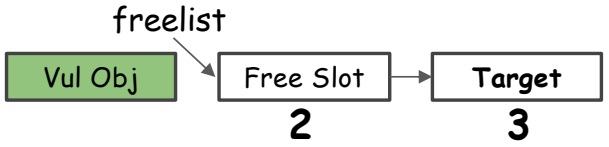
1. Desired layout depends on the vulnerability, including corruption range, control over corruption value, etc.
2. Side-effect depends on the vulnerability, including # of (de)allocations in each system call, allocation order of the vulnerable object, etc.
3. Instead of covering all possibilities case by case, we summarize them into two situations

Situation 1 - Target Slot is Unoccupied

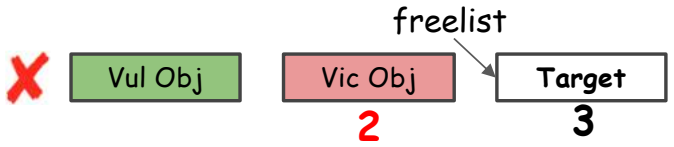
Initial State



1. Allocate the Vul Obj (1)

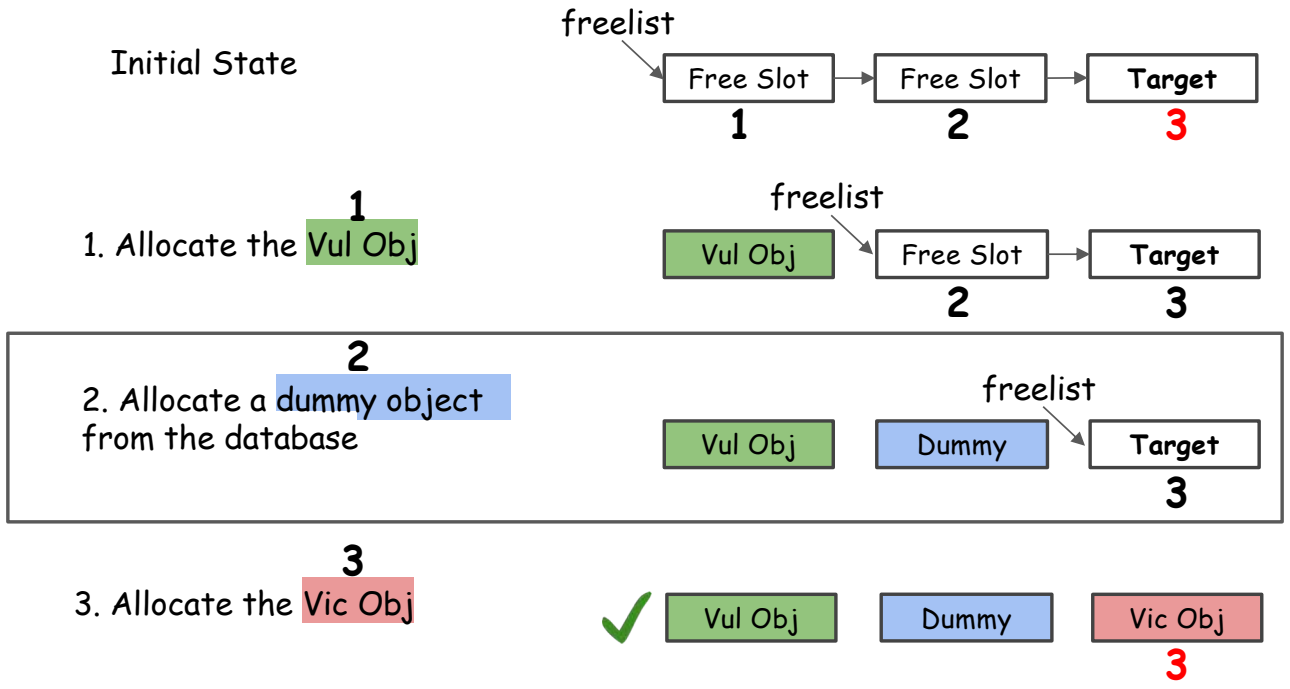


2. Allocate the Vic Obj (2)

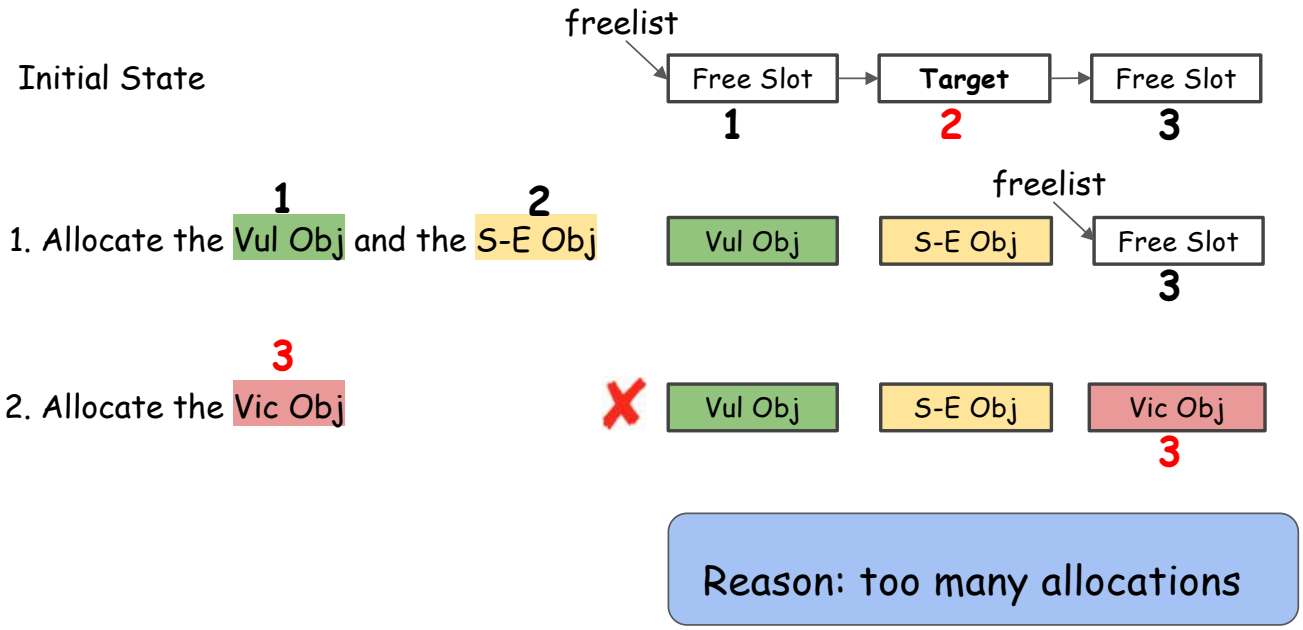


Reason: too few allocations

Situation 1 - Our Solution



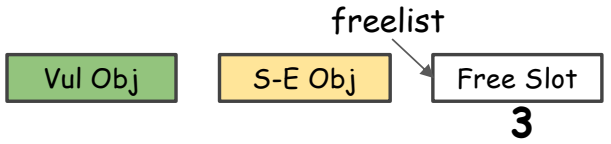
Situation 2 - Target Slot is Occupied



Situation 2 - Straightforward But Wrong Solution

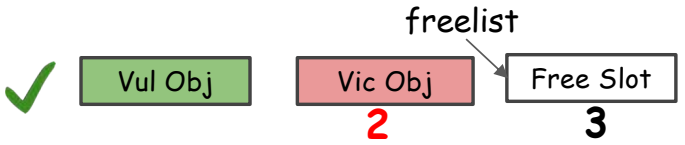
Problems with straightforward solution
- No general syscalls and arguments for deallocation
- Vul Obj can also be freed along with the S-E Obj

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



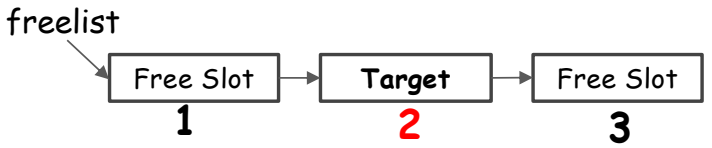
2. Deallocate the S-E Obj (-1)

3. Allocate the Vic Obj (3-1 = 2) ✓



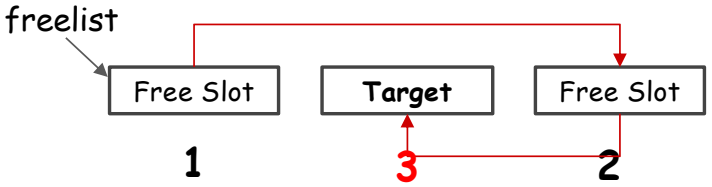
Situation 2 - Our Solution

Initial State



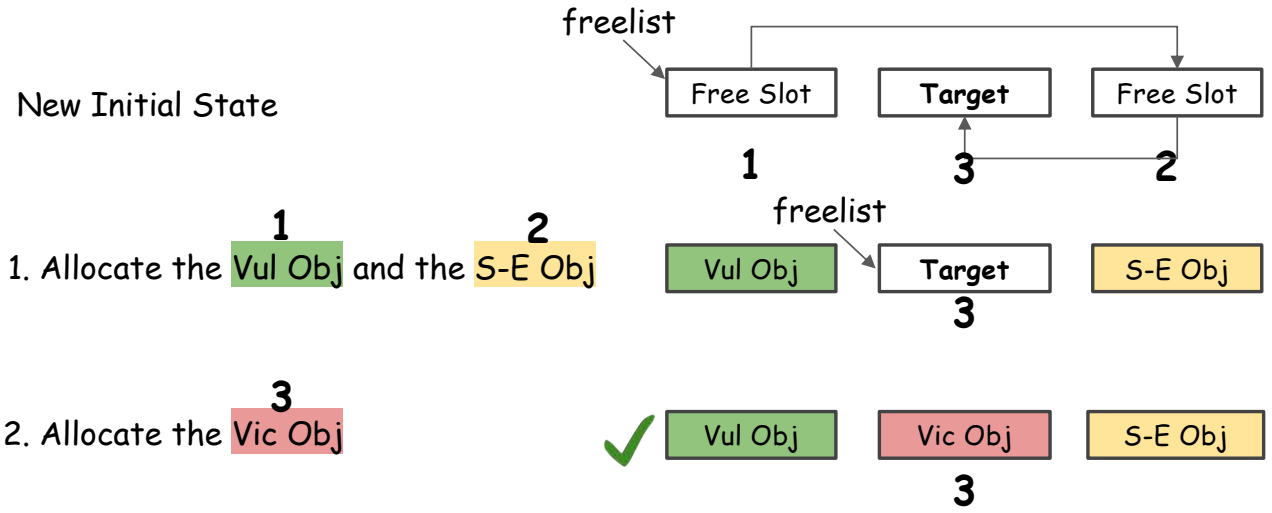
1. Allocate three dummy objects

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.)



Evaluation

- 27 kernel vulnerabilities, including UAF, Double Free, OOB
- Goal: control-flow hijacking primitive
- Succeed in 14 cases with public exploits and 3 cases without public exploits.

CVE-ID	Type	Exploitation Methods			
		I	II	III	IV
N/A[47]	OOB	5 (1★)	-	-	5 (0)
2010-2959	OOB	13 (1★)	-	-	13 (0)
2018-6555	UAF	-	1(1★)	-	-
2017-1000112	OOB	0 (1)	-	-	-
2017-2636	double free	-	0 (1)	-	-
2014-2851	UAF	-	0 (1)	-	-
2015-3636	UAF	-	3 (1)	-	2 (0)
2016-0728	UAF	-	3 (1)	-	4 (0)
2016-10150	UAF	-	3 (1)	-	-
2016-4557	UAF	-	2 (0)	-	-
2016-6187	OOB	-	-	-	6 (1)
2016-8655	UAF	-	3 (1)	-	-
2017-10661	UAF	-	3 (1)	-	-
2017-15649	UAF	-	3 (1)	-	-
2017-17052	UAF	-	0 (0)	-	-
2017-17053	double free	-	-	-	2 (1)
2017-6074	double free	-	3 (1)	12 (0)	-
2017-7184	OOB	10 (0)	-	-	10 (0)
2017-7308	OOB	14 (1)	-	-	14 (0)
2017-8824	UAF	-	3 (1)	-	-
2017-8890	double free	-	4 (1)	4 (0)	-
2018-10840	OOB	0 (0)	-	-	-
2018-12714	OOB	0 (0)	-	-	-
2018-16880	OOB	0 (0)	-	-	-
2018-17182	UAF	-	0 (0)	-	-
2018-18559	UAF	-	3(0)	-	-
2018-5703	OOB	0 (0)	-	-	-

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

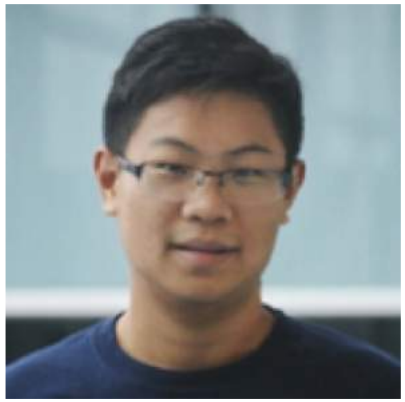
- Create an initial slab cache
  - so that slots are chained sequentially
  - defragmentation
- Calculate side-effect layout
  - ftrace logs calling to allocation/deallocation
  - analyze log to calculate layout before manipulation
- Shorten exploit window
  - to minimize influence of other kernel activities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

DEMO

Summary

1. Identifies objects useful for kernel exploitation
2. Reorganizes slab and obtains the desired layout
3. Evaluated against 27 vulnerabilities and demonstrated its effectiveness

Thank You !



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

Looking for 2020 summer internship