Vulnerability Exploitability Assessment and Mitigation Design Defects in Linux Kernel

Yueqi Chen
The Pennsylvania State University
October 19th, 2019

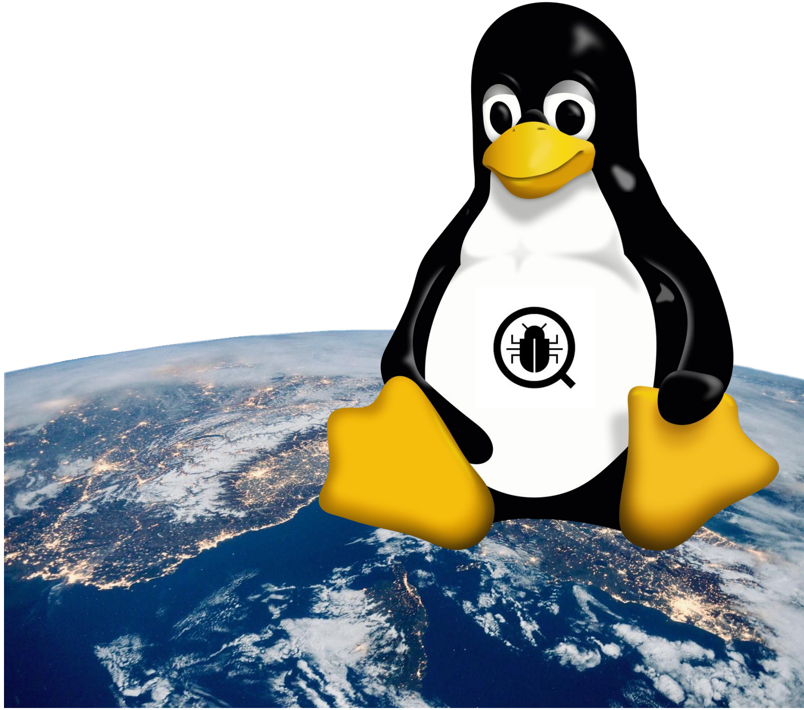
Linux Kernel is Security-critical But Buggy

"Civilization runs on Linux"[1]

- 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

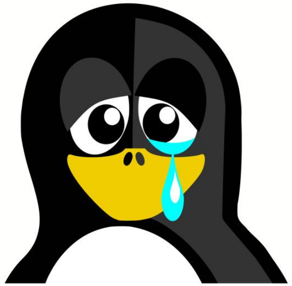


[1] SLTS project, https://lwn.net/Articles/749530/

Harsh Reality: Lack of Workforce for Patching Rapidly

Google Syzbot[2] , on Oct 7th

- 552 not fixed, 103 fix pending, 85 in moderation
- # of bug reports increases 200 bugs/month



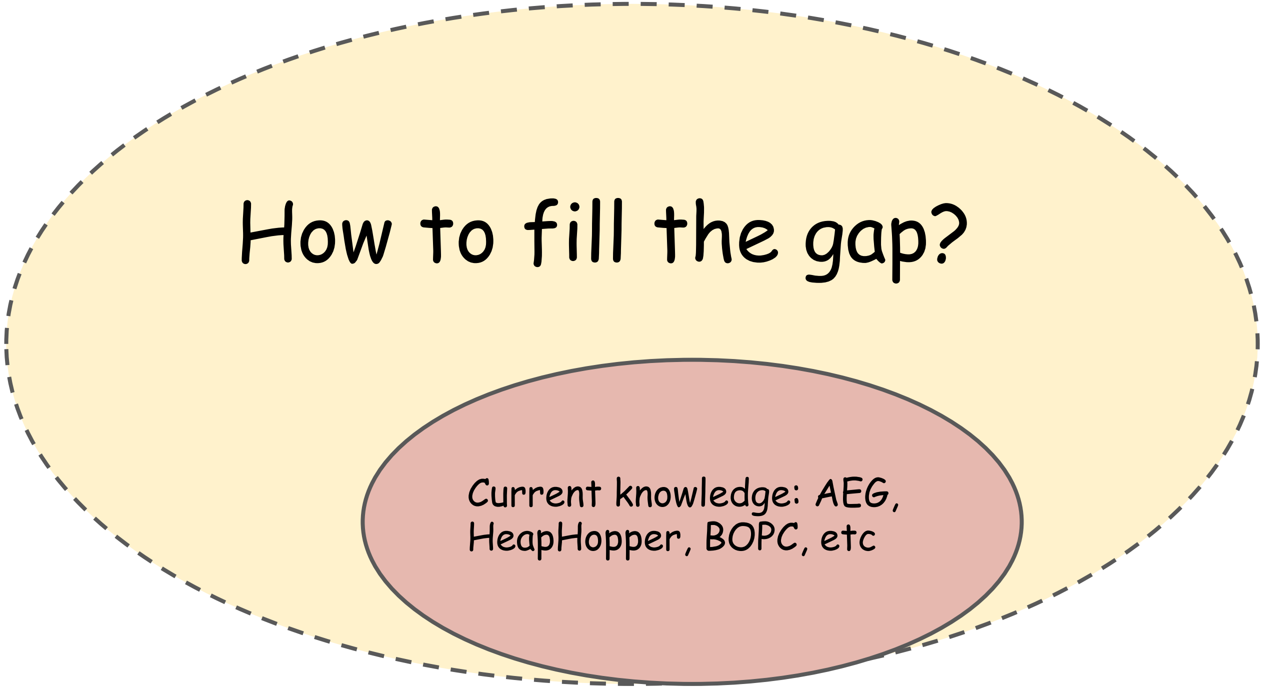
Practical solutions to minimize the damage

- prioritize patching of security bugs based on **exploitability**

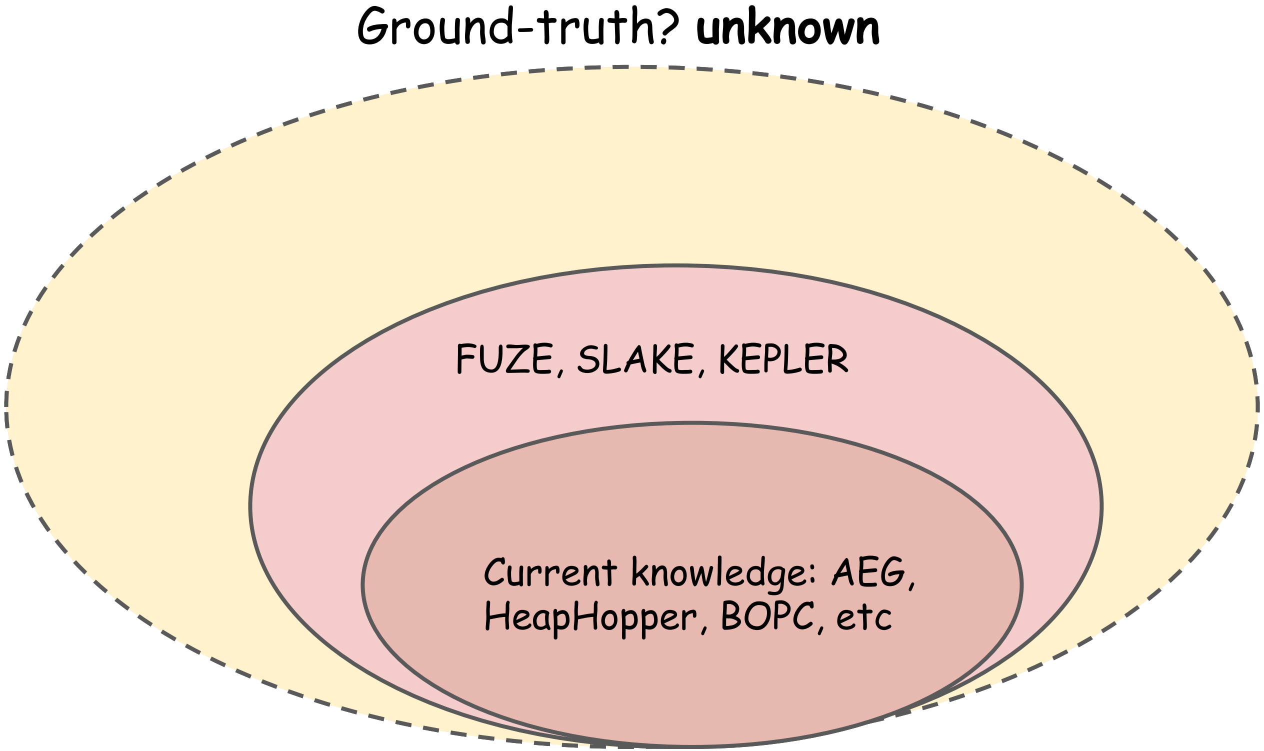
[2] syzbot https://syzkaller.appspot.com/upstream

The state-of-the-art of Exploitability Assessment

Ground-truth? **unknown**



Our Idea: Escalating Exploitability is the First Step



Our Works

FUZE [2] - Explore Capability and Identify Primitives

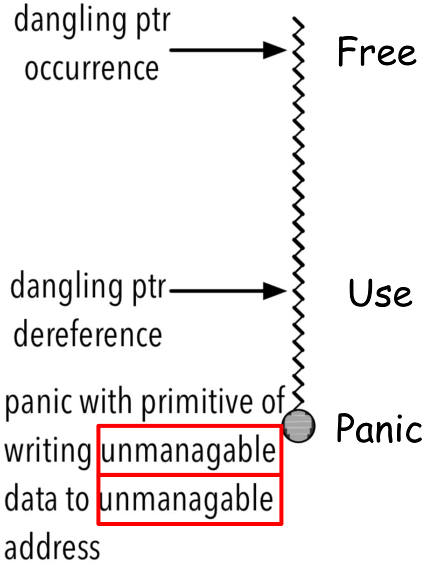
SLAKE [3] - Systematically Manipulate Slab Layout

KEPLER [4] - Bypass Almost All Default Mitigations

[2] FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities, USENIX Sec 2018
[3] SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel, ACM CCS 2019
[4] KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities, USENIX Sec 2019

Park I. FUZE

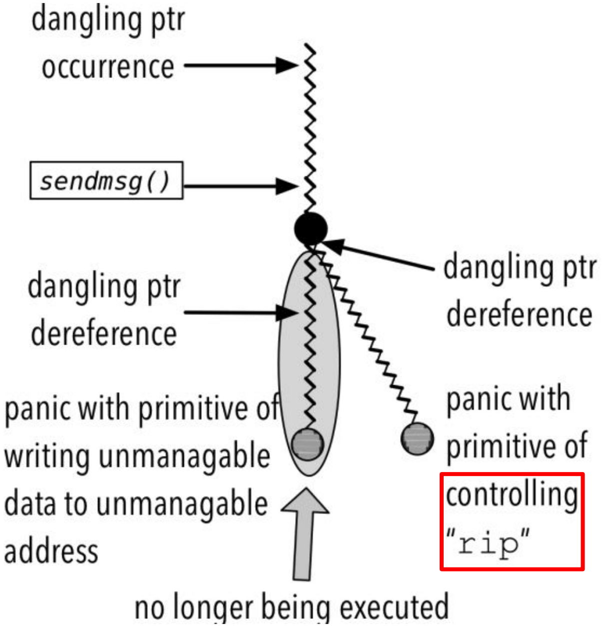
Use-After-Free: Proof-of-Concept (PoC) vs. Exploit



Free

Magic One: Heap spray

Magic Two: New Use



PoC: panic kernel without demonstrating exploitation

Proof-of-Concept (PoC)

Exploit: new use demonstrating exploitability (e.g., RIP control, arbitrary write/read)

Exploit

Magic One: Heap Spray

"kmalloc-256"

1. Free Object A

2. Heap Spray: Allocate Object B many times

3. Use Object A (B)

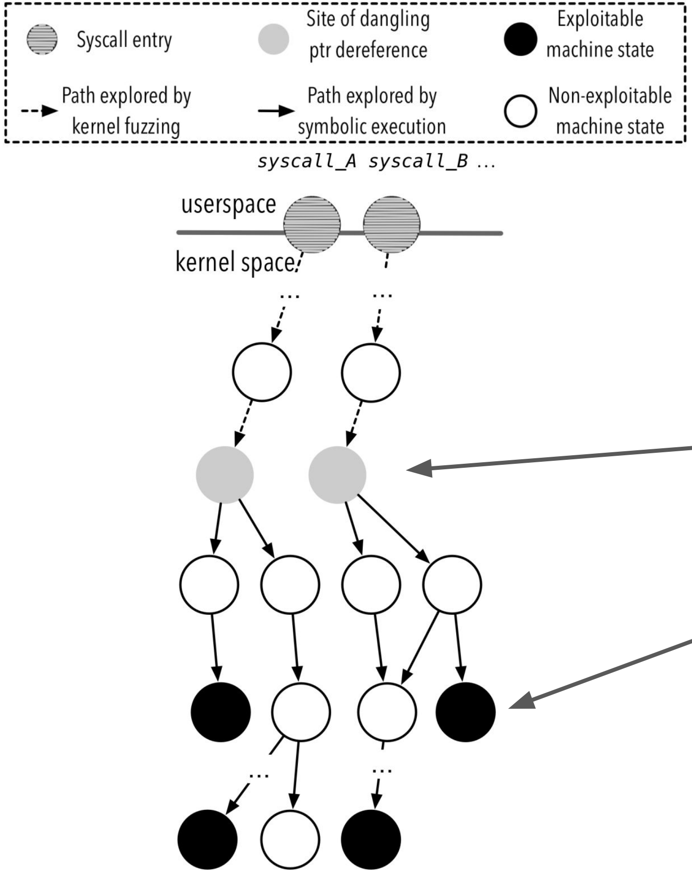
SLAB/SLUB allocator is shared; cache is shared.

Key Idea: Use content of Object B to tamper content of Object A.

Common heap spray syscalls: add_key(), sendm[m]sg(), msgsnd. etc.

FUZE's contribution: compute the content of Object B,
e.g., key for add_key(), msg for sendm[m]sg, msg()

Magic Two: New Use



FUZE's contribution:

Kernel Fuzzing - explore new dereference sites

Symbolic Execution - identify exploitable machine state

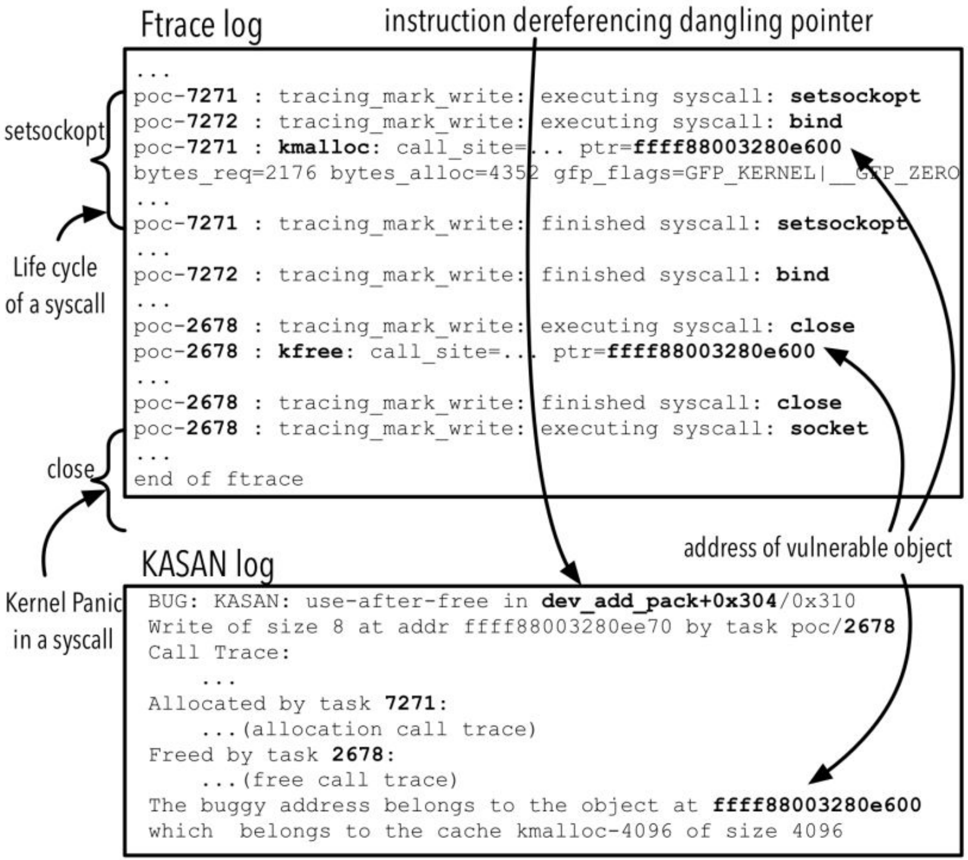
Technical Details - Under-Context Kernel Fuzzing

```
PoC_wrapper(){ // PoC wrapping function
    ...
    syscallA(...); // free site
    return; // instrumented statement
    syscallB(...); // dangling pointer
        ↪ dereference site
    ...
}
```

Build “free” Context

Kick in kernel fuzzing

Technical Details - Exploitable Machine State Identification



After fuzzing panics kernel,
set freed slot as symbolic value,
continue kernel execution symbolically
until either

1. $RIP is symbolic value
   or
2. src/dst operand of MOV is symbolic value

Evaluation

CVE-ID	# of public exploits		# of generated exploits	
	SMEP	SMAP	SMEP	SMAP
2017-17053	0	0	1	0
2017-15649	0	0	3	2
2017-15265	0	0	0	0
2017-10661	0	0	2	0
2017-8890	1	0	1	0
2017-8824	0	0	2	2
2017-7374	0	0	0	0
2016-10150	0	0	1	0
2016-8655	1	1	1	1
2016-7117	0	0	0	0
2016-4557	1	1	4	0
2016-0728	1	0	3	0
2015-3636	0	0	0	0
2014-2851	1	0	1	0
2013-7446	0	0	0	0
Overall	5	2	19	5

Table 4: Exploitability comparison with and without `FUZE`.

CVE-ID	Fuzzing		Symbolic Execution		
	Time	# of syscalls	Min # of BBL	Max # of BBL	Ave # of BBL
2017-17053	NA	NA	6	18	13
2017-15649	26 m	433	4	39	21
2017-15265	NA	NA	4	5	5
2017-10661	2 m	26	7	14	11
2017-8890	139 m	448	13	86	48
2017-8824	99 m	63	2	33	23
2017-7374	NA	NA	NA	NA	NA
2016-10150	NA	NA	1	1	1
2016-8655	1m	448	4	27	14
2016-7117	NA	NA	1	1	1
2016-4557	1 m	133	3	48	29
2016-0728	1 m	7	21	31	26
2015-3636	NA	NA	NA	NA	NA
2014-2851	146 m	1203	1	5	3
2013-7446	209 m	448	1	2	1

Table 5: The Efficiency of fuzzing and symbolic execution.

Take Away

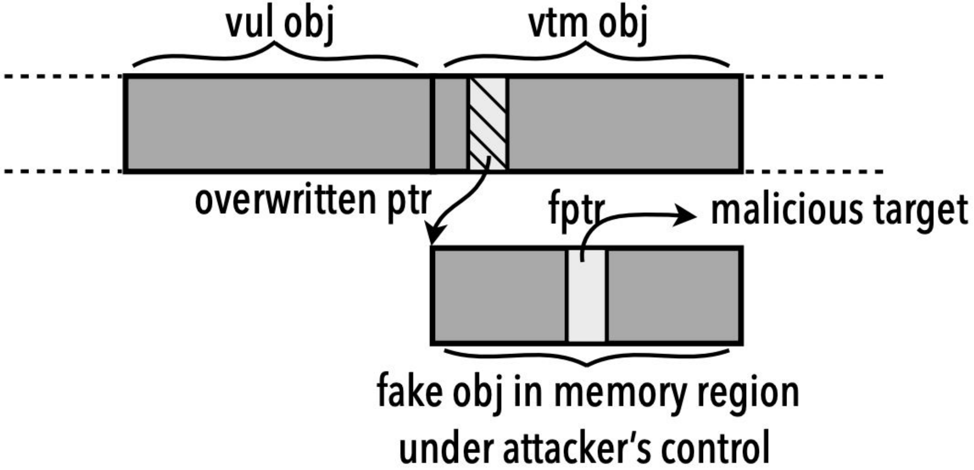
1. PoC doesn’t expose all capability.
2. Exploring capability can escalate exploitability.

FUZE is the **1st** paper exploring capability of vulnerability in Linux kernel

Park II. SLAKE

Challenges Facing Slab-based Vuln Exploitability Assessment

Example:
exploitation through Slab
Out-of-Bound (OOB) Write:



1. Which object is useful for exploitation?
2. How to (de)allocate and dereference the object?
3. How to manipulate slab layout?

Victim/Spray Objects Are Useful

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

struct file {
        ...
        // data object pointer
        const struct file_operations *f_op;
        ...
}

file->f_op->llseek(...); // indirect call
```

Victim Object
For hijacking control flow

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

Spray Object
For Tampering object/function pointer

Solution: Identify statically through type definitions and usage patterns

Evaluation

SLAKE's contributions:

1. Static analysis to collect candidate structure types
2. Kernel fuzzing to identify syscalls and corresponding parameters to (de)allocate and dereference victim objects and spray objects

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

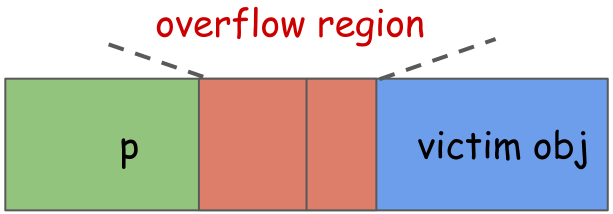
# of identified objects/syscalls (v4.15, defnoconfig + 32 modules)

Side Effect of Syscalls on Slab Layout

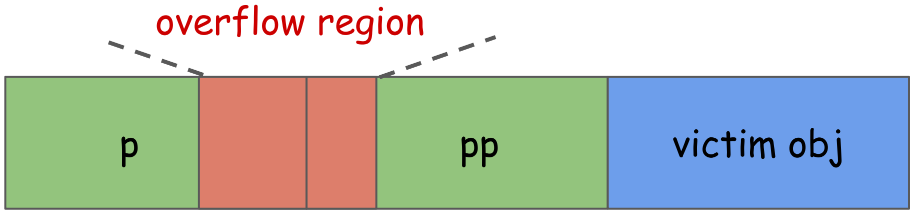
Side effect: (de)allocation of objects except from victim/spray objects.

```
static int xfrm_alloc_replay_state_esn (...)
{
        struct xfrm_replay_state_esn *p, *pp;
        ...
        p = kzalloc(klen, GFP_KERNEL);

        // side effect: unexpected allocation
        pp = kzalloc(klen, GFP_KERNEL);
        ...
}
```

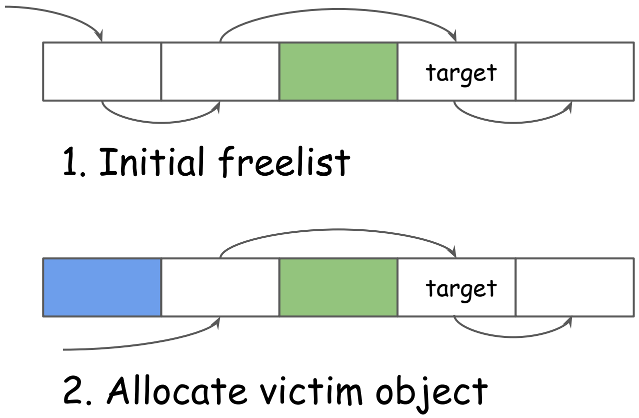


Desired Layout

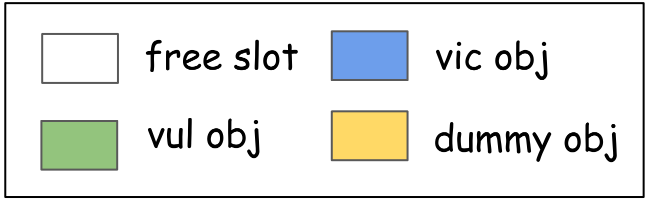


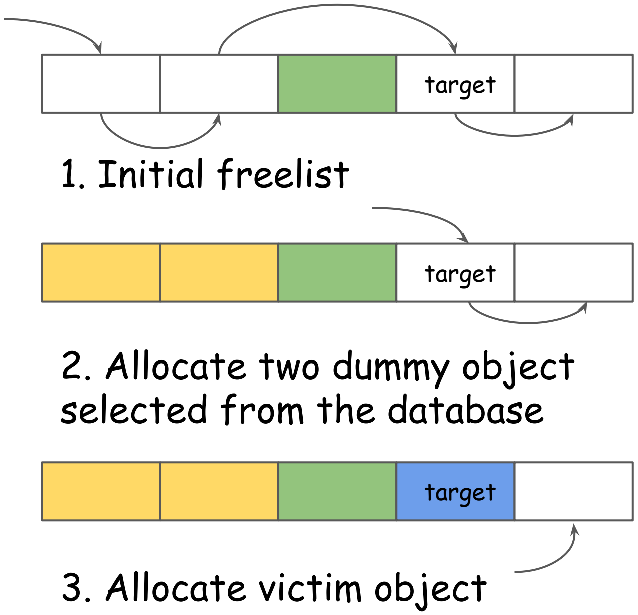
Side-effect Layout --> exploit fails

Adjust Unoccupied Slots by Sliding Freelist



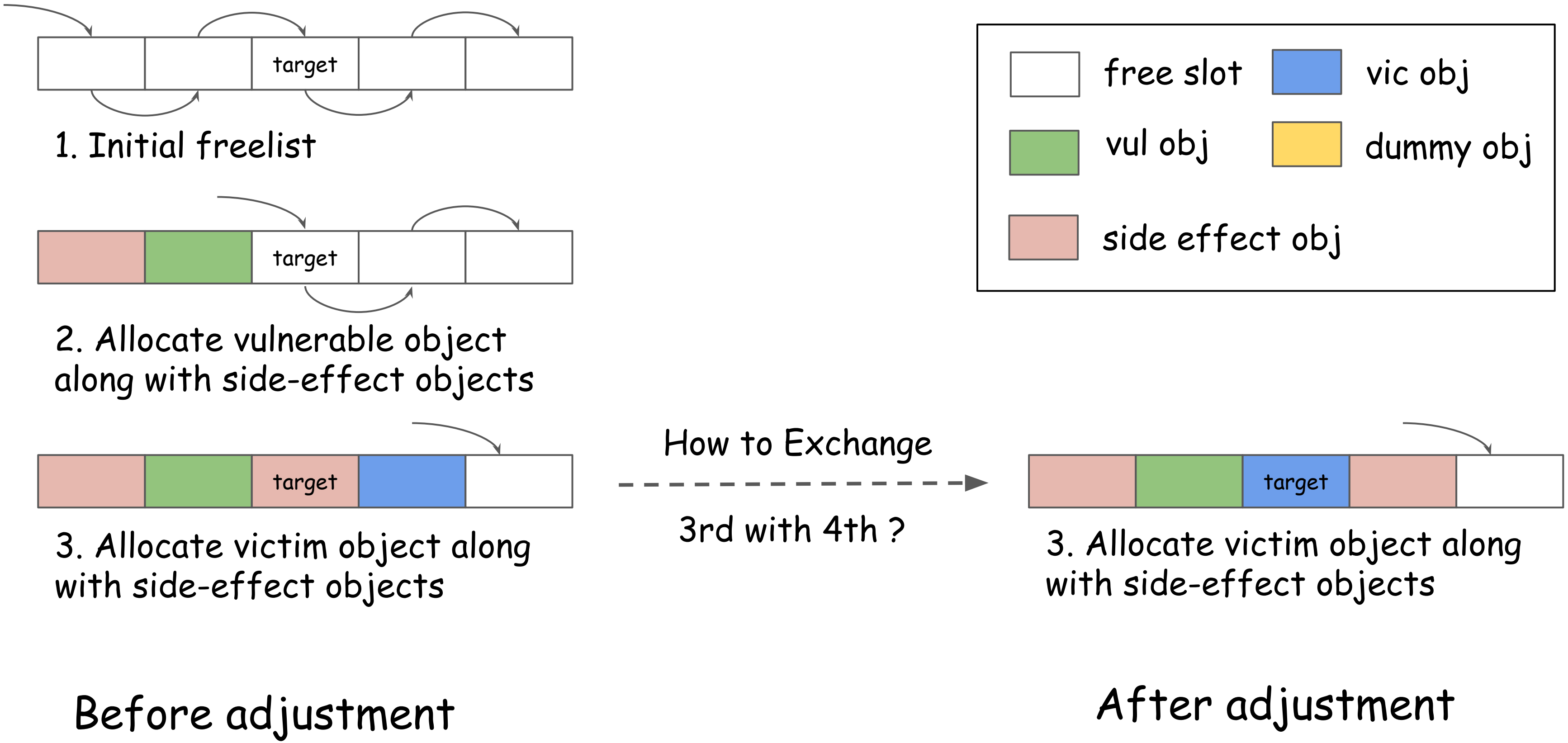
Before adjustment



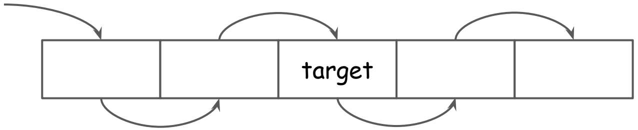


After adjustment

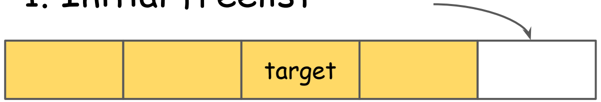
Reorganize Occupied Slots by shuffling freelist



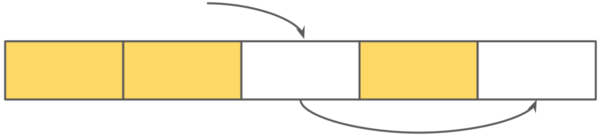
Reorganize Occupied Slots by shuffling freelist (cont.)



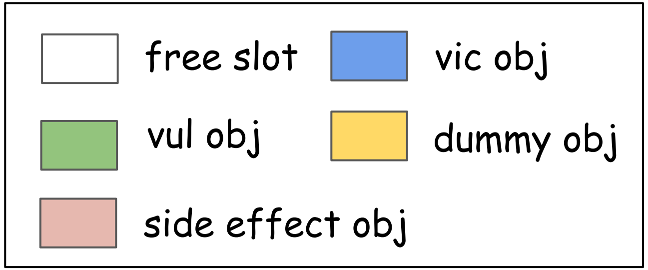
1. Initial freelist

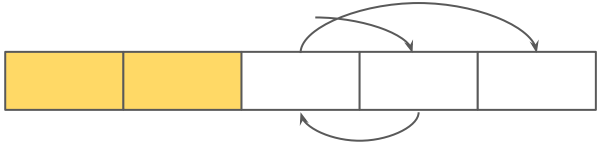


2. Allocate four dummy objects

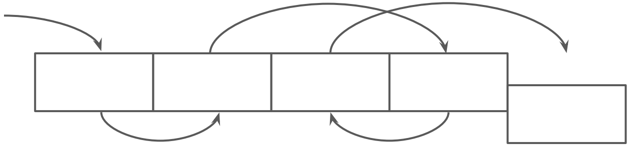


3. Free 3rd dummy objects





4. Free 4th dummy objects



5. Free 2th, 1st dummy objects (new initial list)

6. Repeat: allocate vulnerable and victim objects along with side-effect objects

After adjustment

Evaluation

27 vulnerability (largest evaluation set so far): 26 CVEs + 1 Wild =
13 UAF + 4 Double Free + 10 Slab Out-of-bound Write

18 vulns: public exploits

9 vulns: **NO** public exploits

15 vulns: 8 additional
unique exploits on avg.

Diversify

9 vulns: failure cases

3 vulns: 25 unique
exploits in toal.

Escalate

6 vulns: PoC manifests limited capability
(e.g., corruption inside object, no control
over tampering value)

3 vulns: special caches

Take Away

1. Build a database for kernel object and systematically perform Fengshui can empower the capability of developing working exploits
2. SLAKE is able to escalate exploitability and benefit its assessment for Linux kernel bugs

SLAKE is the **1st** paper comprehensively bridging memory corruption to control flow hijacking

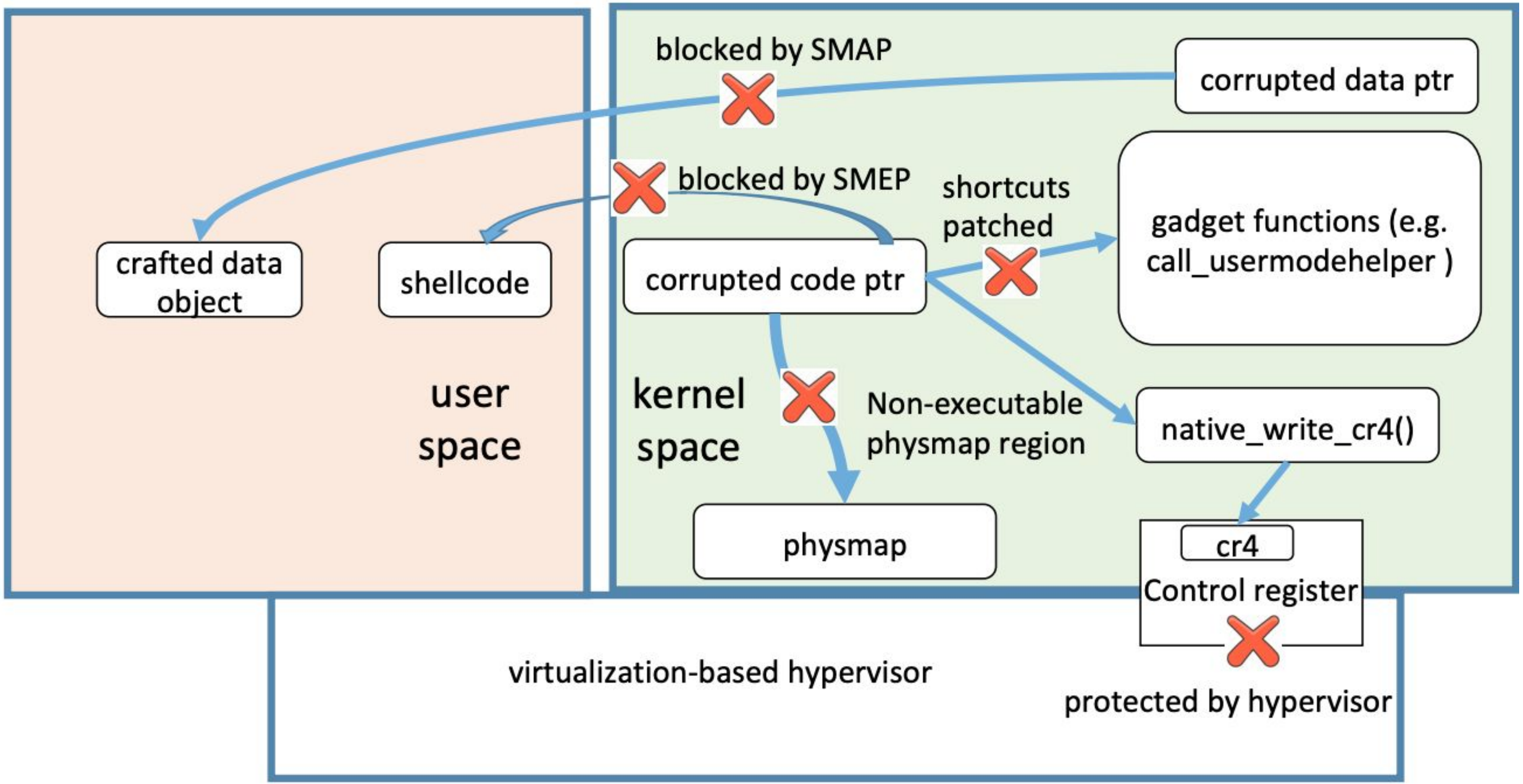
Park III. KEPLER

Both FUZE and SLAKE Assume:

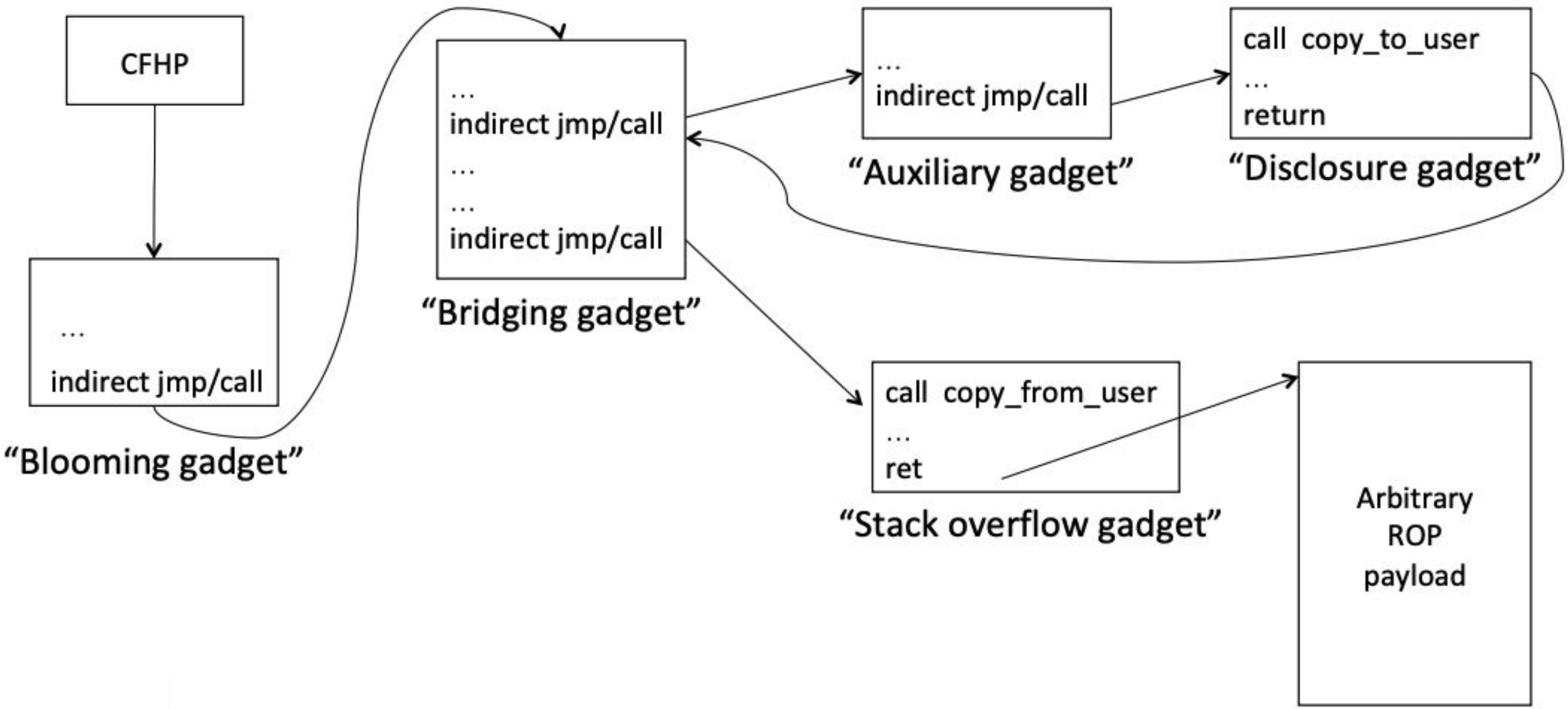
Control-flow Hijacking = exploitable

Is this assumption reasonable?
Can kernel mitigation defeat control flow hijacking?

Mitigations in Linux Kernel

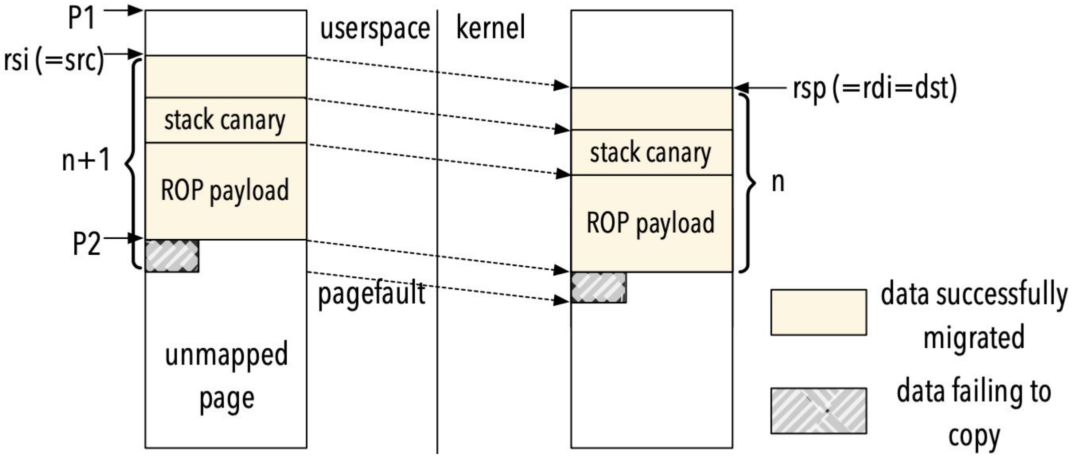


Bypassing Mitigations



Stack Overflow Gadget

```
static long bsg_ioctl(struct file *file, unsigned int cmd,
unsigned long arg){
   struct sg_io_v4 hdr; // destination is local variable
   …
   if (copy_from_user(&hdr, uarg, sizeof(hdr))) {
      return -EFAULT; // short return
   }
```

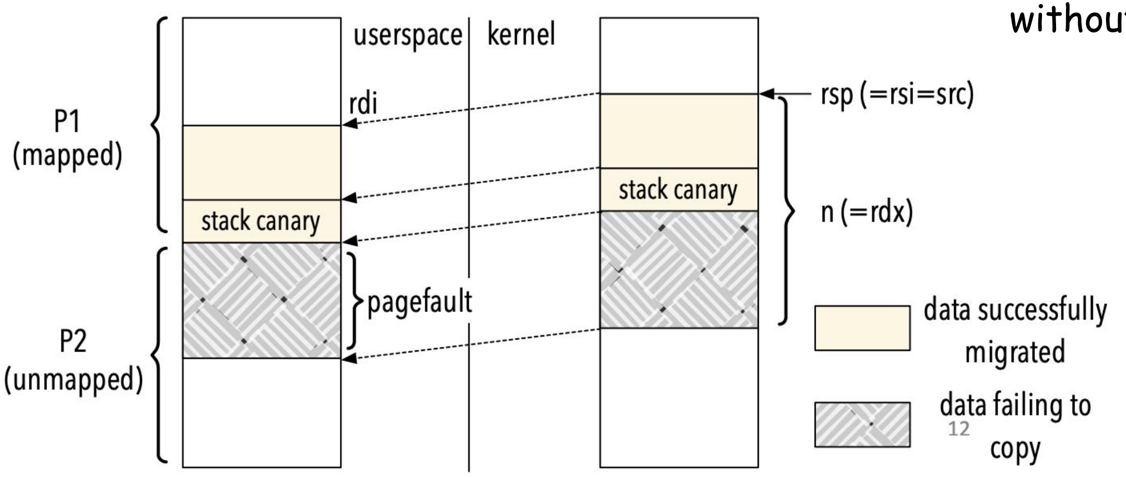


Stack overflow gadget :
Copy ROP payload to kernel stack

Question: how to disclose stack canary?

Stack Disclosure Gadget

```
SYSCALL_DEFINE2(gettimeofday, struct timeval *, tv, struct
timezone *, tz){
   struct timeval ktv;
   …
   if(copy_to_user(tv, &ktv, sizeof(ktv))) {
      return -EFAULT;
   }
   …
```



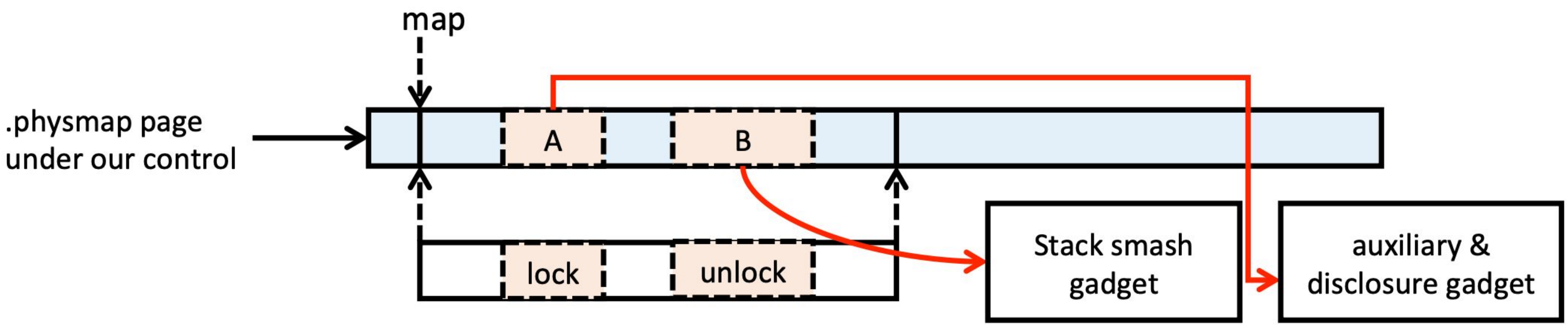
Stack Disclosure Gadget:
Copy stack canary to userland

Question: How to hijack control flow twice without BUG_ON() or panicking kernel?

Bridging Gadget

```
void regcache_mark_dirty(struct regmap *map){
  map->lock(map->lock_arg);// the 1st control-flow hijack
  map->cache_dirty=true;
  map->no_sync_defaults=true;
  map->unlock(map->lock_arg);// the 2nd control-flow hijack
}
```

Bridging Gadget:
Spawning two control-flow hijacking and combing canary leak and stack smash



Evaluation

ID	Vulnerability type	Public exploit	Q	FUZE	KEPLER	G1	G2	G3	G4	First chain (min)	Total time (hour)	Total # of exploitation chains
CVE-2017-16995	OOB readwrite	✓†	✗	✗	✓	41	114	27	201	45	37	29788
CVE-2017-15649	use-after-free	✓	✗	✓	✓	29	79	25	280	16	28	60207
CVE-2017-10661	use-after-free	✗	✗	✗	✓	28	78	30	301	17	25	49070
CVE-2017-8890	use-after-free	✗	✗	✗	✓	21	88	23	304	17	18	50471
CVE-2017-8824	use-after-free	✓	✗	✓	✓	63	101	35	306	50	70	164898
CVE-2017-7308	heap overflow	✓	✗	✗	✓	31	91	30	241	14	47	110176
CVE-2017-7184	heap overflow	✓	✗	✗	✓	31	95	31	254	24	37	93752
CVE-2017-6074	double-free	✓	✗	✗	✓	18	79	31	308	16	15	31436
CVE-2017-5123	OOB write	✓†	✗	✗	✓	40	86	27	311	14	39	113466
CVE-2017-2636	double-free	✗	✗	✗	✓	18	89	29	289	29	19	26372
CVE-2016-10150	use-after-free	✗	✗	✗	✓	34	84	25	293	52	34	88499
CVE-2016-8655	use-after-free	✓†	✗	✓†	✓	18	109	32	260	15	17	47413
CVE-2016-6187	heap overflow	✗	✗	✗	✓	22	85	32	301	17	21	51954
CVE-2016-4557	use-after-free	✗	✗	✗	✓	21	80	21	295	16	37	40889
CVE-2017-17053	use-after-free	✗	✗	✗	✗	-	-	-	-	-	-	-
CVE-2016-9793	integer overflow	✗	✗	✗	✗	-	-	-	-	-	-	-
TCTF-credjar	use-after-free	✓†	✗	✗	✓	35	89	25	292	25	14	82913
0CTF-knote	uninitialized use	✗	✗	✗	✓	21	89	33	318	17	36	40923
CSAW-stringIPC	OOB read&write	✓†	✗	✗	✓	35	88	25	289	17	33	84414

1. 16 CVEs + 3 CTF challenges
2. Tens of thousands of exploit chains in 50 wall clock minutes
3. Hard to defeat because the gadget could not be easily removed
4. CVE-2017-17053, CVE-2016-9793 ?

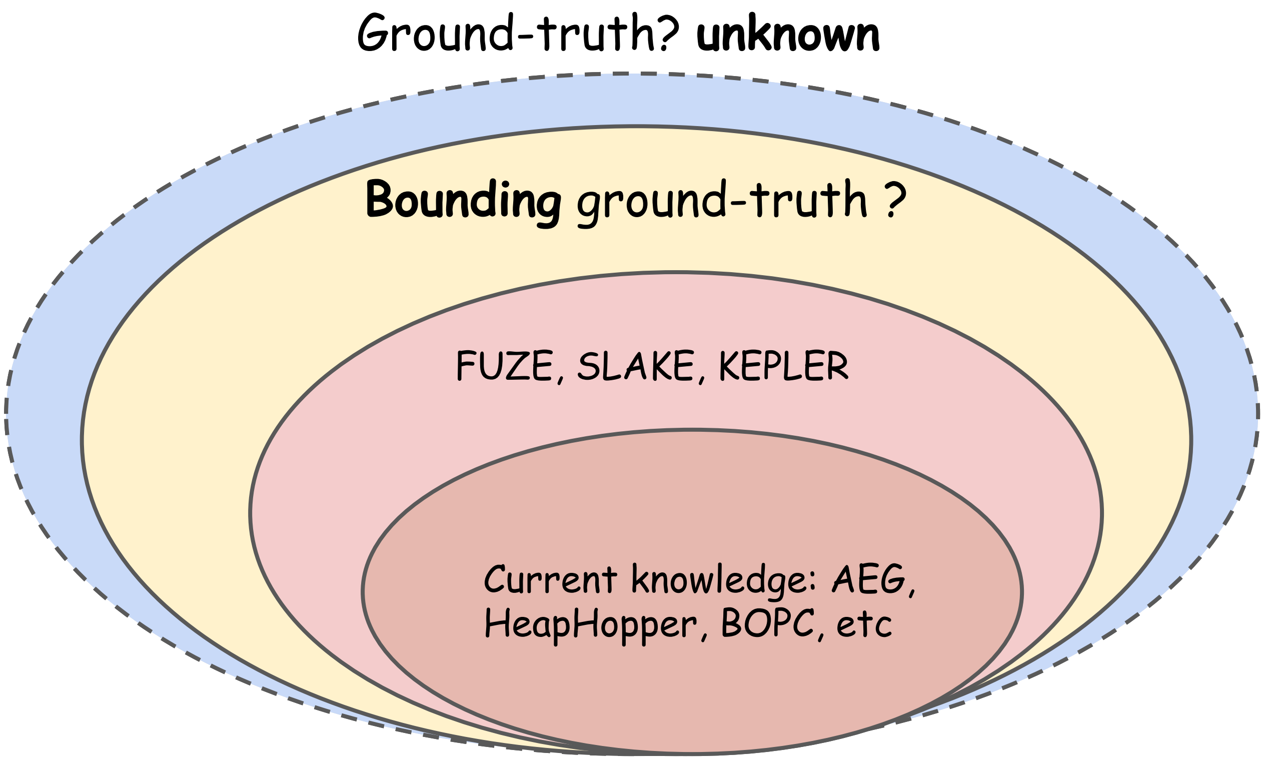
Take Away

1. Control-flow hijacking ≈ exploitable
2. Practical Kernel CFI should be designed and deployed

KEPLER is the **1st** paper considering mitigations when evaluating control-flow hijacking primitive

Conclusion & Future Work

I. Escalating Exploitability Towards Ground-truth



II. Attack-Mitigation-New Attack Circle

Shellcode injection

✘ W^X (DEP)

Stack pivot to user land (ret2usr)

✘ SMPE

Execute code in physmap (ret2dir)

✘ non-executable physmap

Fake object in userland

✘ SMAP (PAN)

Call_usermodehelper

✘ whitelist

Tamper cr4 to disable SMEP/SMAP

✘ VMM-base hypervisor

KEPLER

- Control Flow Integrity (CFI)?

Yet another exploit

- ???

1. Jump out of "Attack-Mitigation-Attack Circle"
2. Proactively Secure Systems

Thank You

Wechat

Twitter: @Lewis_Chen_
Email: ycchen@ist.psu.edu
Personal Page:
http://www.personal.psu.edu/yxc431/
Misc: Looking for 2020 summer internship