Towards Exploitability Assessment for Linux Kernel Vulnerabilities

Yueqi (Lewis) Chen

Advisor: Xinyu Xing

The Pennsylvania State University

Nov 22th, 2019

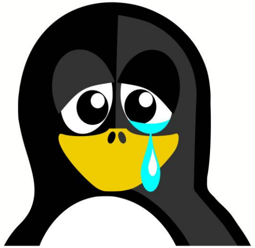
Vulnerability Exploitation Research in Decades

2008	Return-oriented Programming: Exploitation without Code Injection
2009	Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities
2011	AEG: Automatic Exploit Generation
2016	DARPA hosted the Cyber Grand Challenge (CGC)

The community shows continued enthusiasm in vulnerability exploitation. Why?

Reasons for Studying Vulnerability Exploitation

1. Prioritize the Patching of Bugs
   a. Linux kernel is security-critical but buggy
      i. Android (2e9 users), cloud servers, nuclear submarines, etc.
      ii. 631 CVEs (2017, 2018), 4100+ official bug fixes (2017)
   b. Harsh Reality: cannot patch all bugs immediately
      i. Google Syzbot on Nov 25th: 458 not fixed, 94 fix pending, 53 in moderation
      ii. # of bug reports increases 200 bugs/month

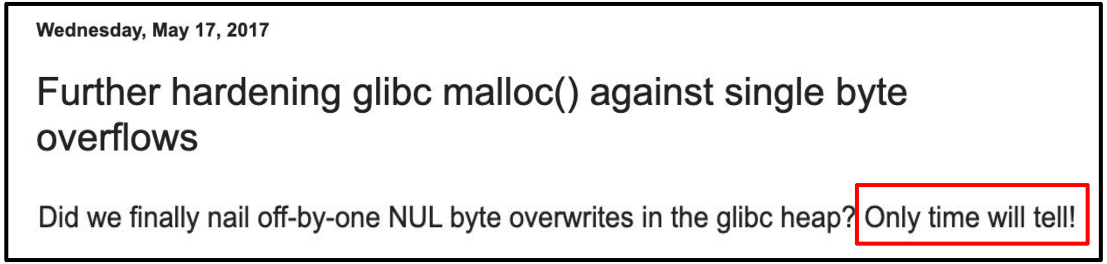


Practical solution to minimize the damage: prioritize patching of security bugs based on **exploitability**

Reasons for Studying Vulnerability Exploitation (cont.)

2. Evaluate the effectiveness of defenses

Does the new defense successfully invalidate attacks ?

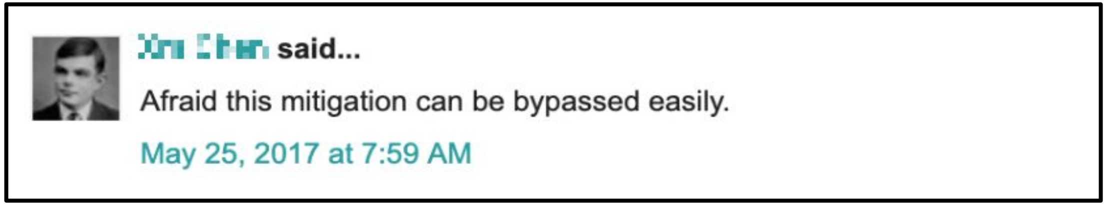


Wednesday, May 17, 2017

Further hardening glibc malloc() against single byte overflows

Did we finally nail off-by-one NUL byte overwrites in the glibc heap? Only time will tell!

The adversaries know the answer best.



[redacted] said...

Afraid this mitigation can be bypassed easily.

May 25, 2017 at 7:59 AM

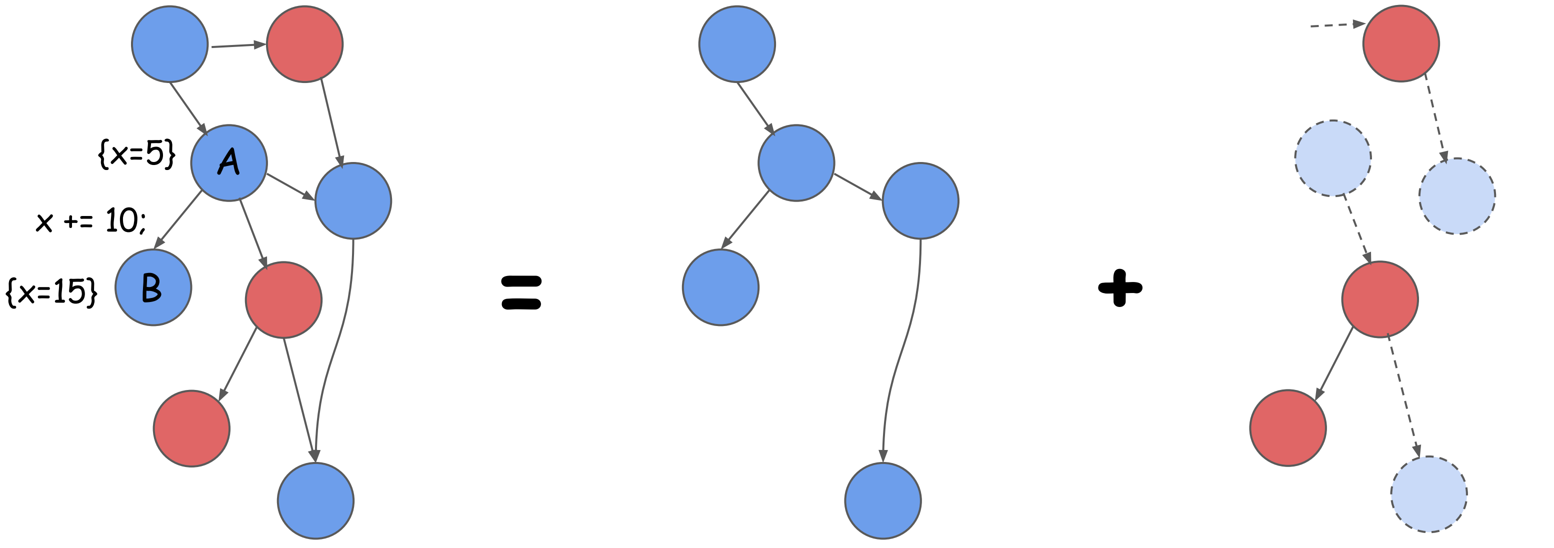
Reasons for Studying Vulnerability Exploitation (cont.)

3. Penetration testing
4. Enterprise security risk early warning

...

How to interpret exploitation and exploitability?

Vulnerability Exploitation from State Machine's Perspective



State Machine of A Vulnerable Software

Good states of Software

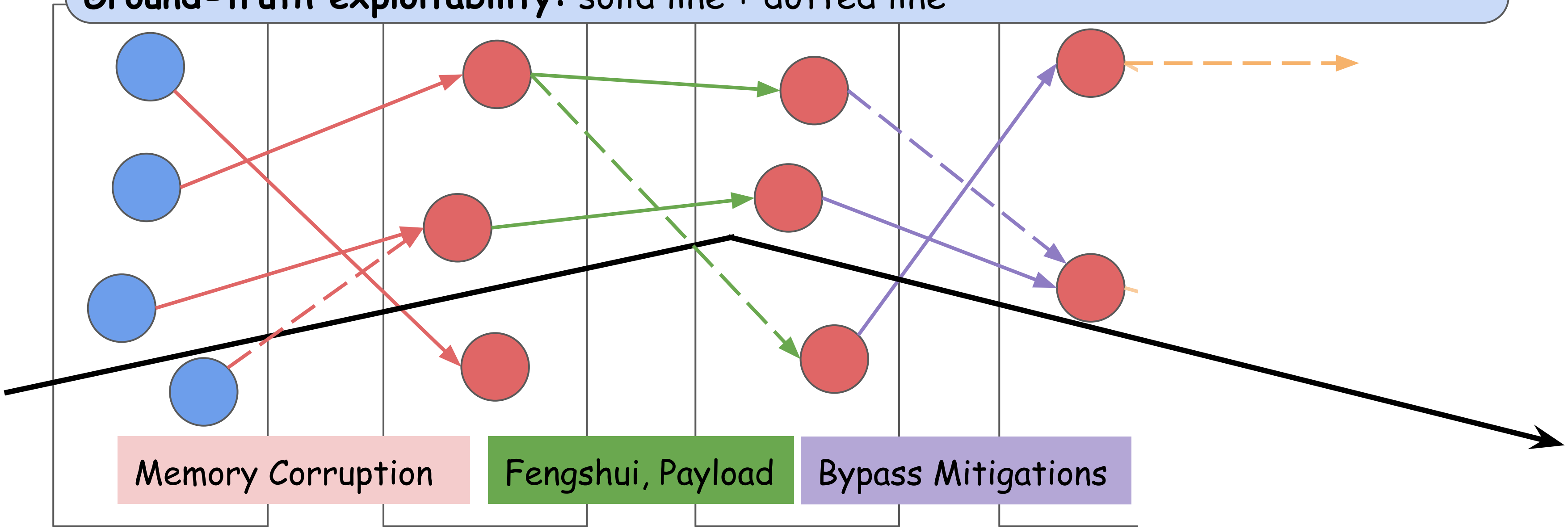
Weird states of Vulnerability

Exploitation is programming weird machine

[1] Thomas Dullien, "Weird machines, exploitability, and provable unexploitability."

Our View of Exploit Development

Exploitability: a property describing whether there is a path from "left" to "right"
Known exploitability: solid line;
Ground-truth exploitability: solid line + dotted line



Memory Corruption

Fengshui, Payload

Bypass Mitigations

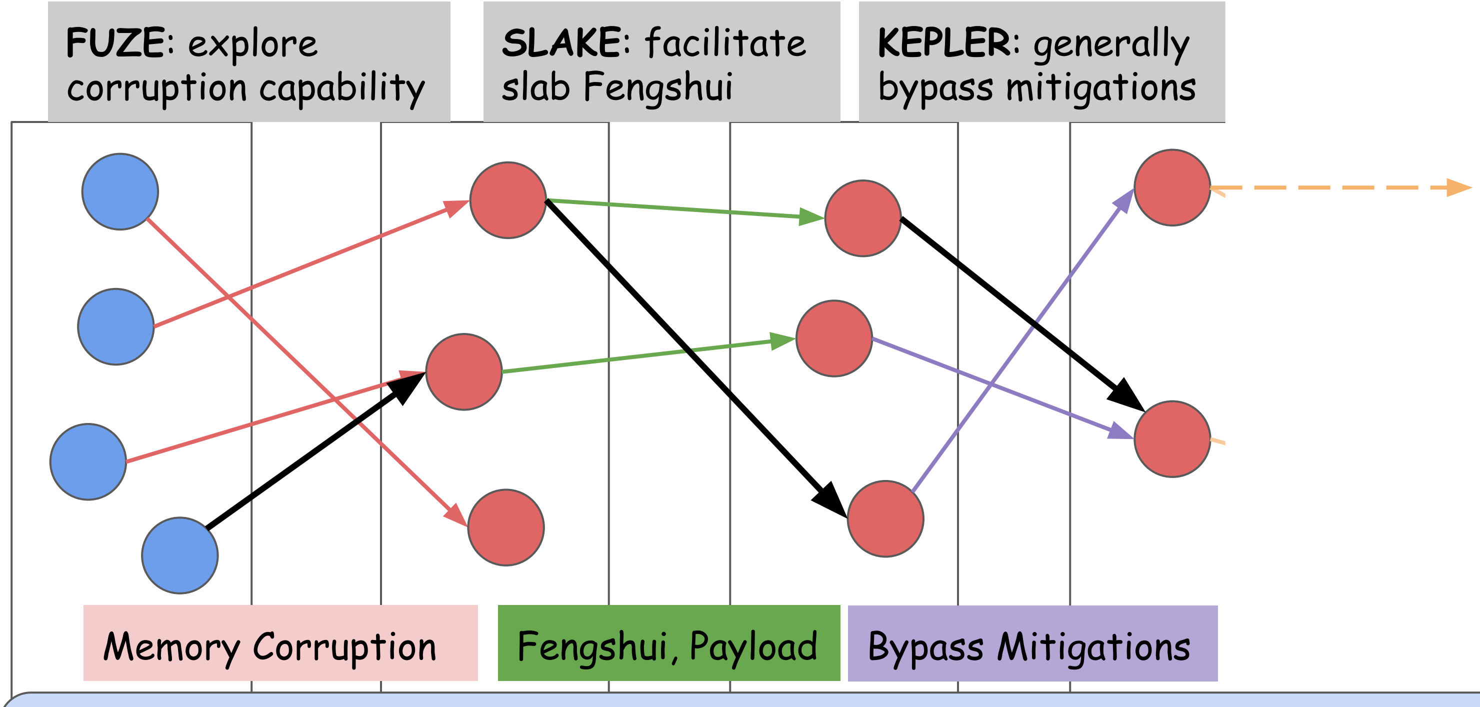
Good states

Corruption states
e.g., use-after-free

Primitive states
e.g., control-flow
hijacking

Success st
e.g., privilege
escalation

Our Works in the Linux Kernel



FUZE: explore corruption capability

SLAKE: facilitate slab Fengshui

KEPLER: generally bypass mitigations

Memory Corruption

Fengshui, Payload

Bypass Mitigations

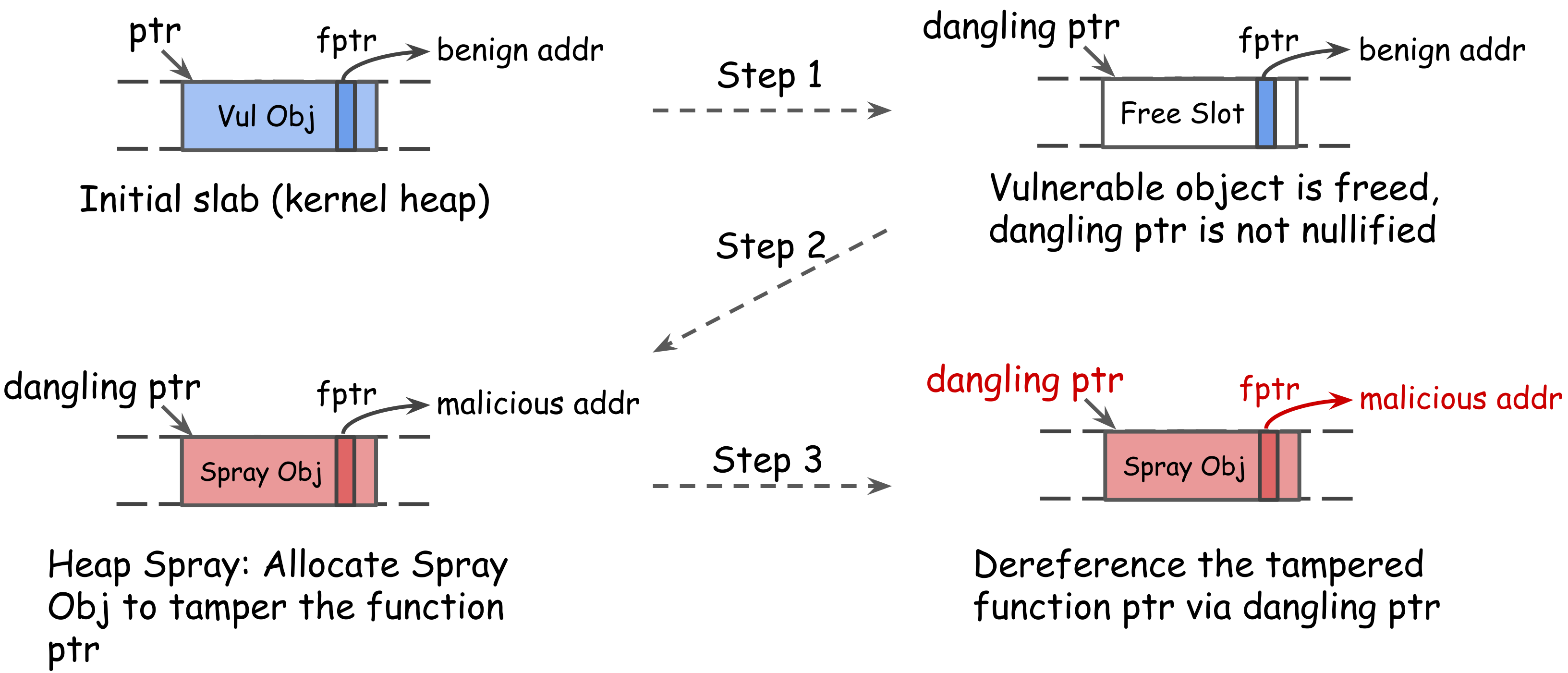
Key idea: Escalate exploitability (solidate dotted lines and connect more paths) towards ground-truth for more sound assessment

Park I

FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities

USENIX Security 18

Workflow of Use-After-Free Exploitation



Initial slab (kernel heap)

Vulnerable object is freed, dangling ptr is not nullified

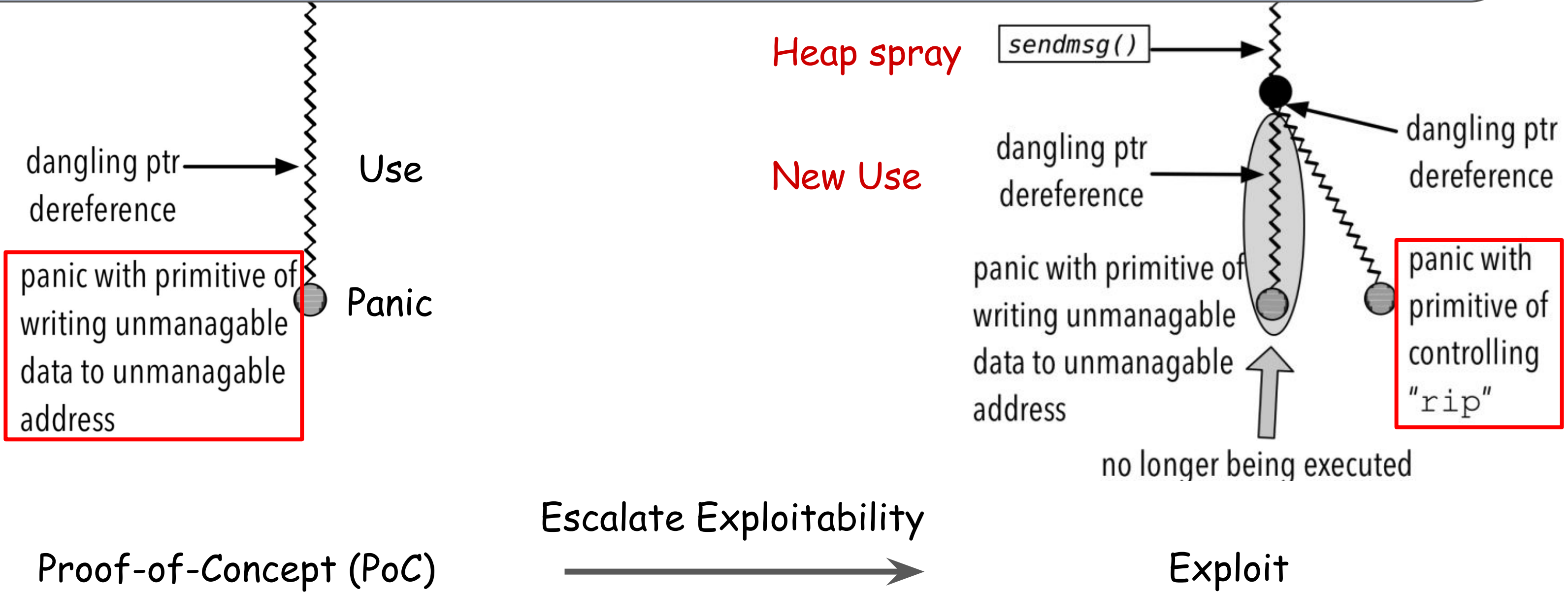
Heap Spray: Allocate Spray Obj to tamper the function ptr

Dereference the tampered function ptr via dangling ptr

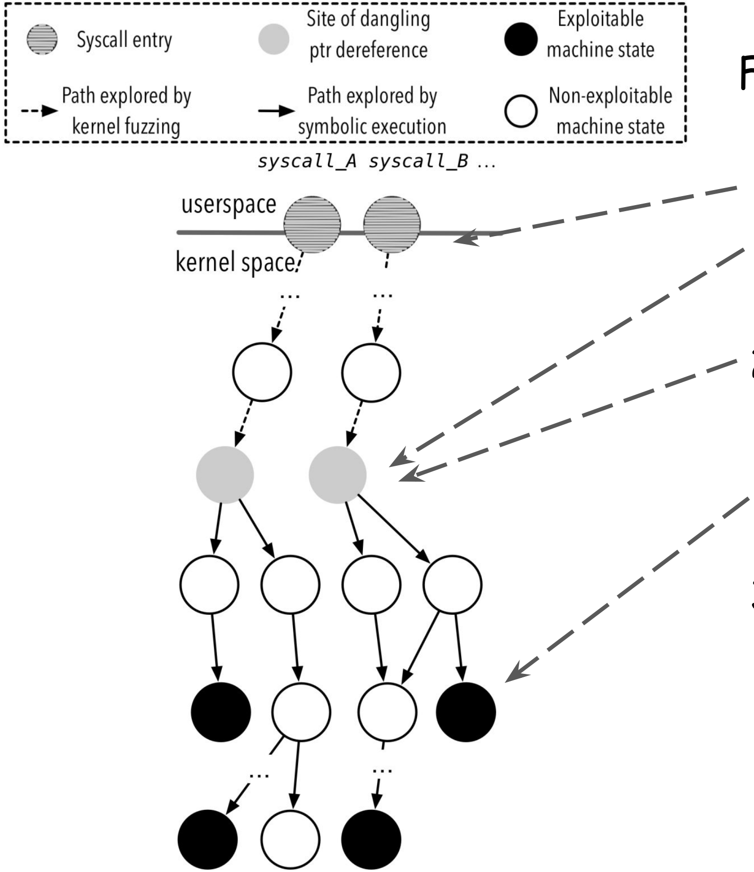
Example: Exploit A Use-After-Free in Three Steps

Challenges of Use-After-Free Exploitation

1. What are the system calls and arguments to reach new use sites?
2. Does the new use site provide useful primitives for exploitation?
3. What is the content of spray object?


dangling ptr dereference
Use
panic with primitive of writing unmanagable data to unmanagable address
Panic
Heap spray
sendmsg()
New Use
dangling ptr dereference
dangling ptr dereference
panic with primitive of writing unmanagable data to unmanagable address
panic with primitive of controlling "rip"
no longer being executed
Escalate Exploitability
Proof-of-Concept (PoC)
Exploit

Overview of FUZE


Syscall entry
Site of dangling ptr dereference
Exploitable machine state
Path explored by kernel fuzzing
Path explored by symbolic execution
Non-exploitable machine state
syscall_A syscall_B ...
userspace
kernel space
...
...
...
...

FUZE's contributions:

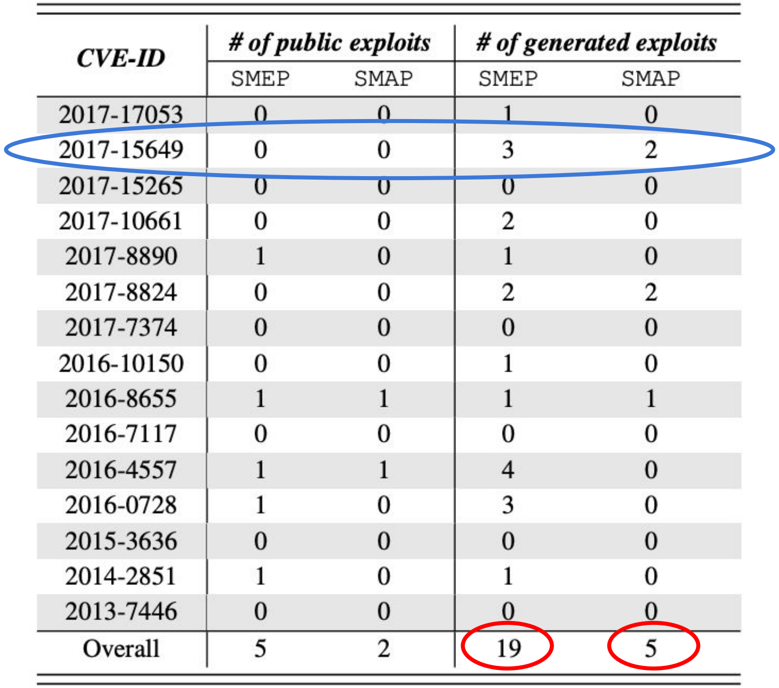
1. Kick in kernel fuzzing to explore new use sites after freeing the vulnerable object

2. Symbolically execute the kernel from the new use sites to check if useful primitives (e.g., RIP control, arbitrary read/write) can be obtained

3. Solve conjunction of path constraints towards primitives and constraints for primitives (e.g., function pointer == the malicious address) to calculate the content of spray object

Evaluation

- 15 kernel UAF vulnerabilities as evaluation set
- FUZE escalated exploitability of 7 vulnerabilities
- The new use sites found by FUZE generate 12 additional exploits bypassing SMEP and 3 additional exploits bypassing SMAP
- Example: CVE-2017-15649



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.

Summary of FUZE

Assumption

- KASLR can be bypassed given hardware side-channels
- Control flow hijacking, arbitrary read/write primitive indicate exploitable machine state
- From PoC program, system calls for freeing object, addr/size of freed object can be learned via debugging tools (e.g., KASAN)

Takeaway

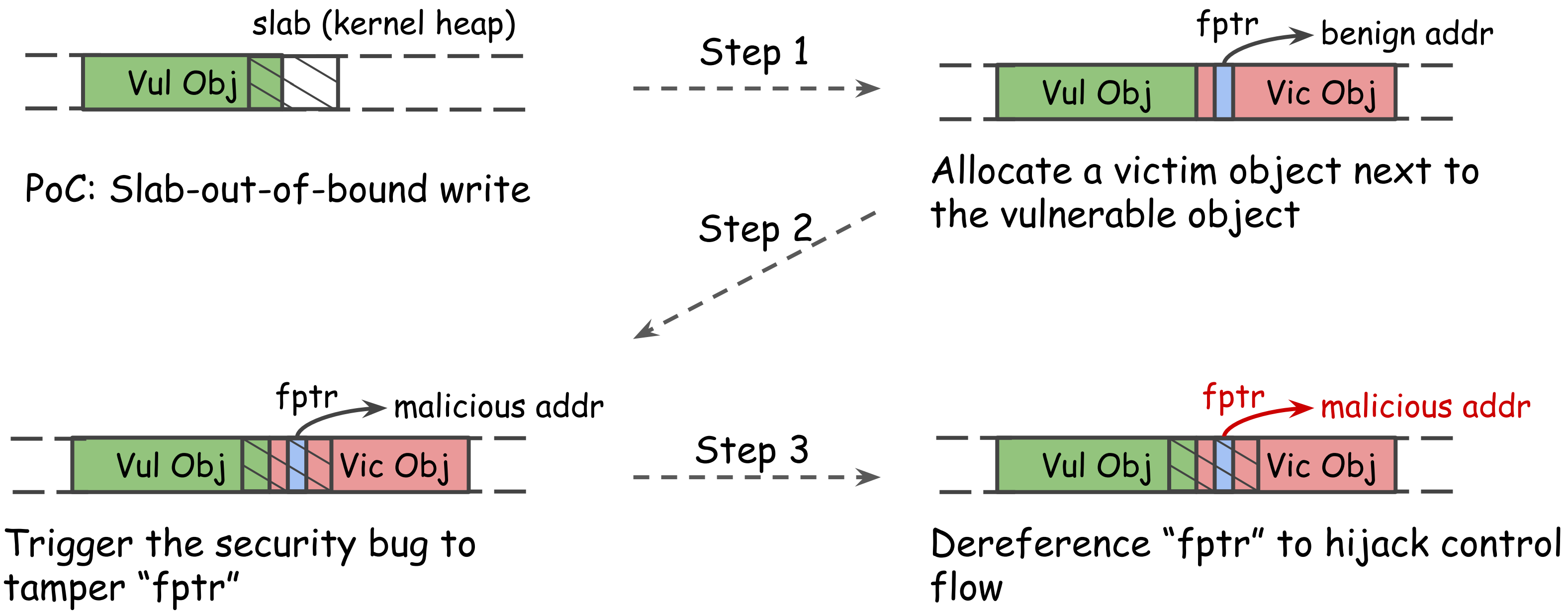
- For Use-After-Free vulnerabilities, new uses indicate more memory corruption capability
- More memory corruption capability escalates the exploitability

Park II

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

ACM CCS 19

Workflow of Slab Out-of-bound Write Exploitation



PoC: Slab-out-of-bound write

Allocate a victim object next to the vulnerable object

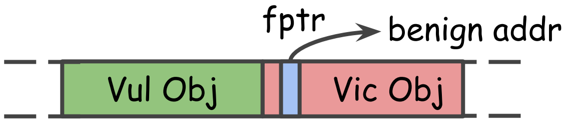
Trigger the security bug to tamper "fptr"

Dereference "fptr" to hijack control flow

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

Common Challenges of Slab Vulnerability Exploitation

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

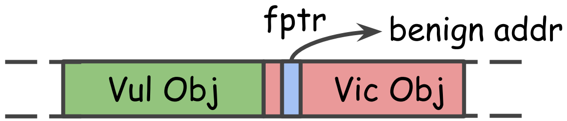


Allocate a **victim** object next to the **vulnerable** object

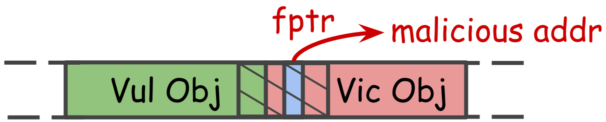
Common Challenges of Slab Vulnerability Exploitation

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



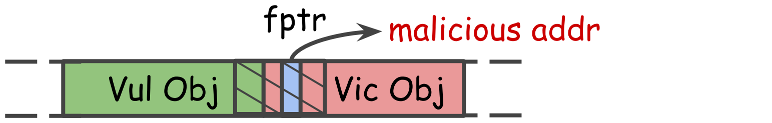
Dereference “fptr” to hijack control flow

Common Challenges of Slab Vulnerability Exploitation

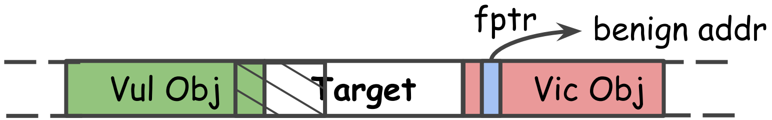
1. Which kernel object is useful for exploitation

2. How to (de)allocate and dereference useful objects

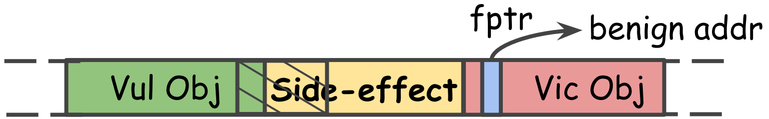
3. How to manipulate slab to reach desired layout
   - unexpected (de)allocation along with vulnerable/victim object makes side-effect to slab layout



Desired Slab Layout



Situation 1: Target slot is unoccupied

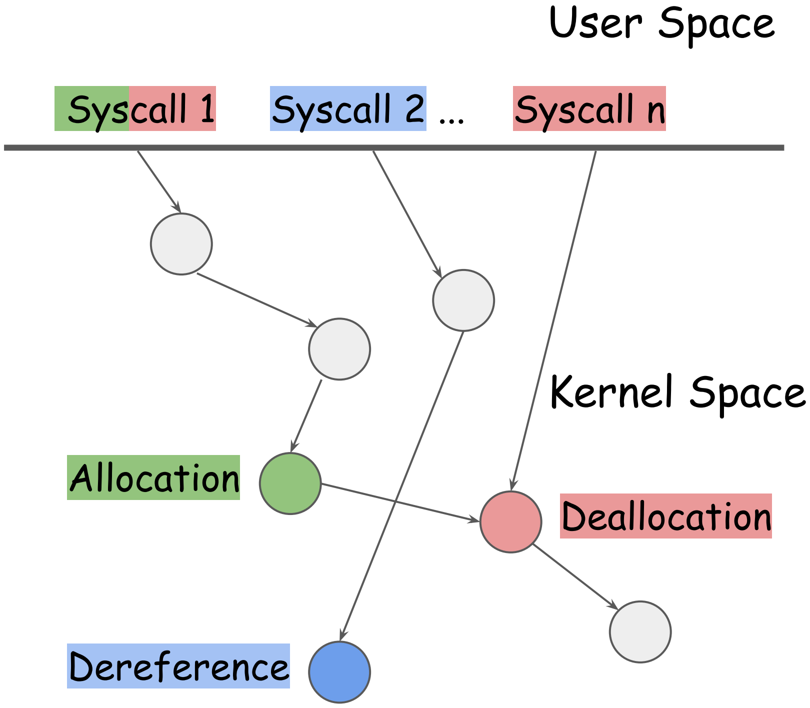


Situation 2: Target slot is occupied

Overview of SLAKE - Resolving Challenge 1&2

SLAKE builds a kernel object database via

- Static Analysis to identify useful objects, sites of interest (allocation, deallocation, dereference), potential system calls
- Fuzzing Kernel to confirm System calls and complete arguments

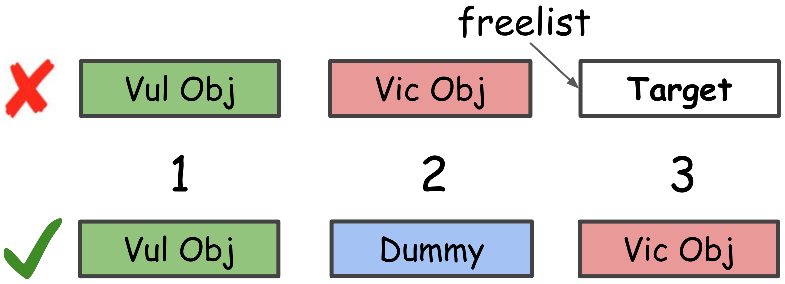


Kernel Call Graph

Overview of SLAKE - Resolving Challenge 3

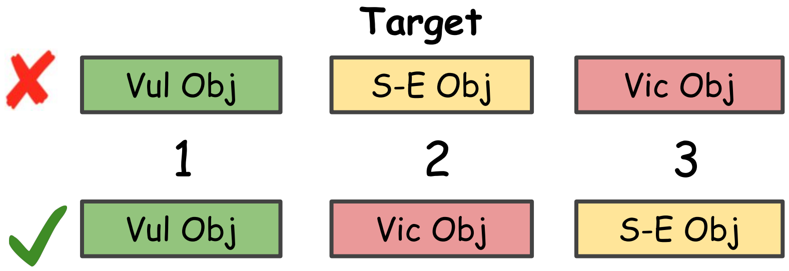
Situation 1: Target slot is unoccupied

- 2 allocations while the order of target slot is 3rd
- add one more allocation of Dummy before Vic Obj



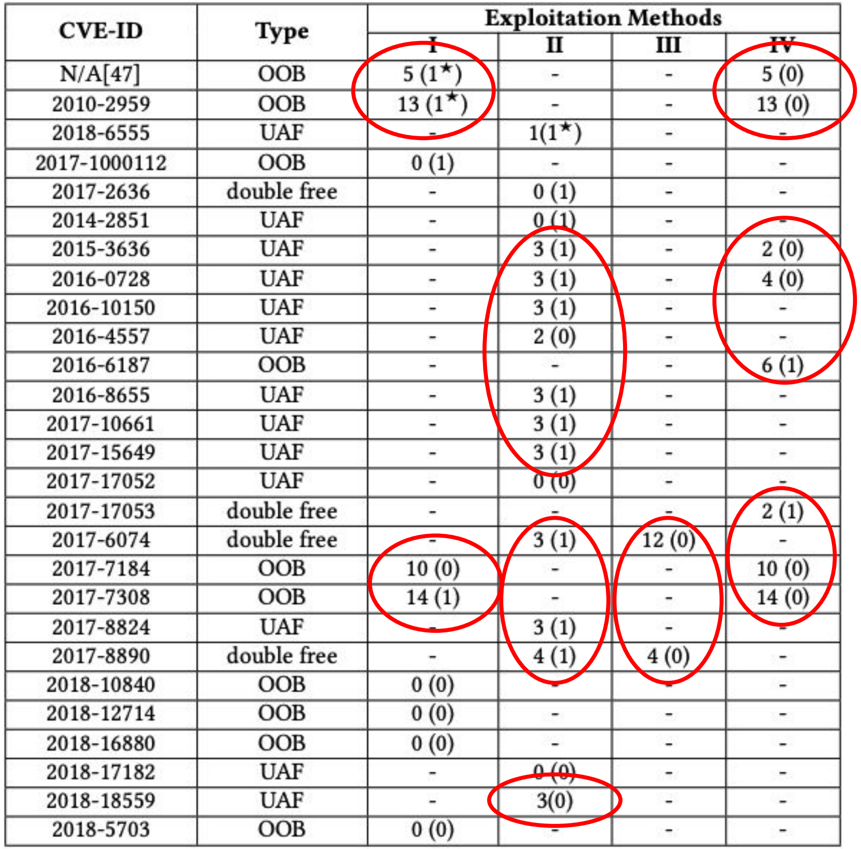
Situation 2: Target slot is occupied

- side-effect object possesses the target
- switch the order of slots holding S-E Obj and Vic Obj in the freelist



Evaluation

- 27 kernel vulnerabilities, including UAF, Double Free, OOB
- SLAKE obtains control-flow hijacking primitive 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)	-	-	-

Summary of SLAKE

Assumption

- KASLR can be bypassed given hardware side-channel
- Partial corruption capability can be learned from PoC program via debugging tools (e.g., GDB, KASAN)
- Control flow hijacking primitive indicates exploitable machine state

Takeaway

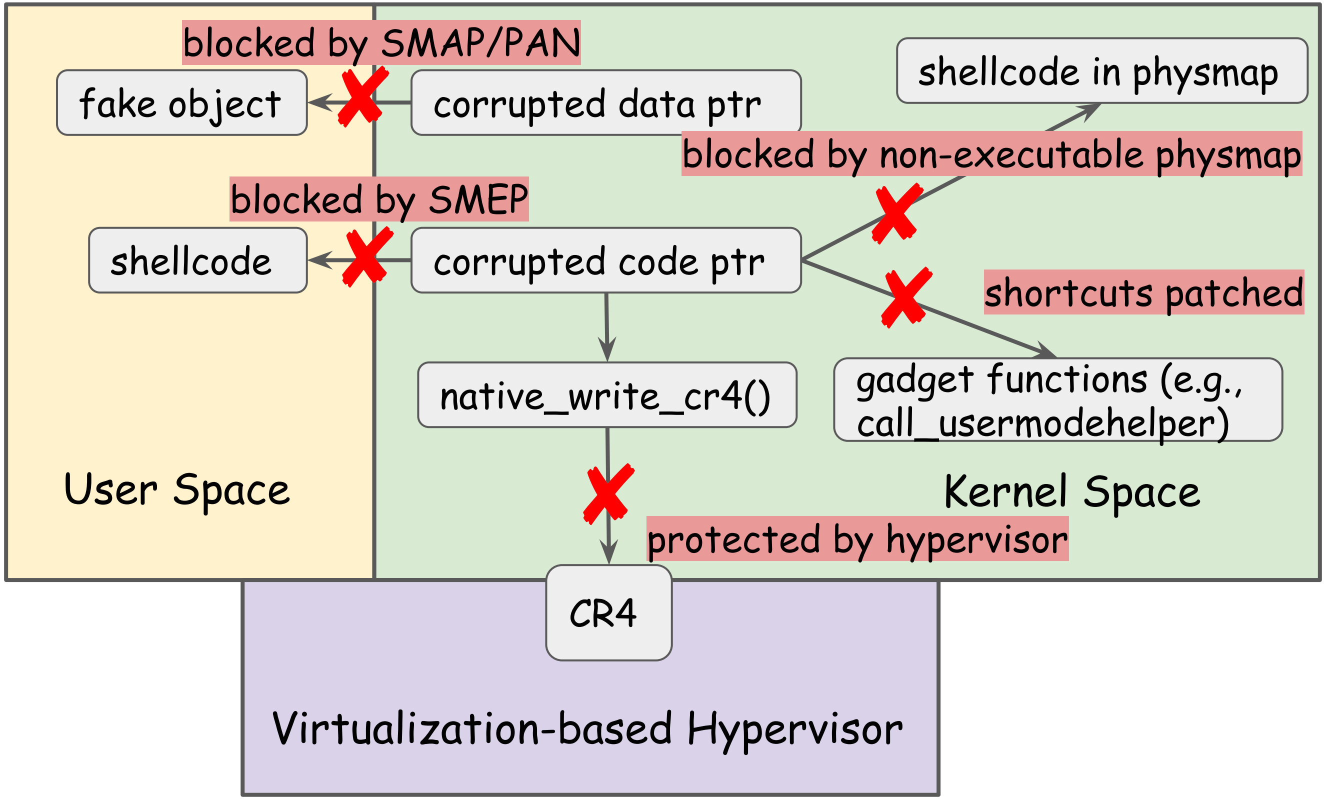
- More useful kernel objects and systematic fengshui approach can bridge the gap between memory corruption and primitives
- Filling the gap not only diversifies the ways of performing kernel exploitation but also potentially escalates exploitability.

Park III

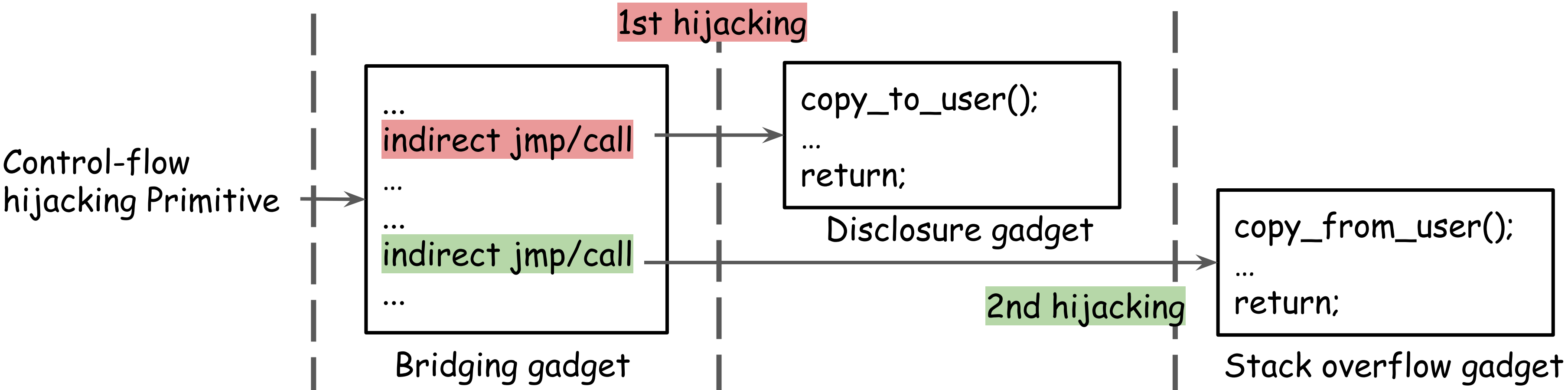
KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities

USENIX Security 19

Mitigations in Linux Kernel



Overview of KEPLER



Control-flow hijacking Primitive

1st hijacking

2nd hijacking

```
…
indirect jmp/call
…
…
indirect jmp/call
…
```

Bridging gadget

```
copy_to_user();
…
return;
```

Disclosure gadget

```
copy_from_user();
…
return;
```

Stack overflow gadget

Obtained through FUZE and SLAKE

1

"Fork" one hijacking into two hijackings

2

SMAP/SMEP is temporarily disabled during copy_to_user() which leaks stack canary to userspace

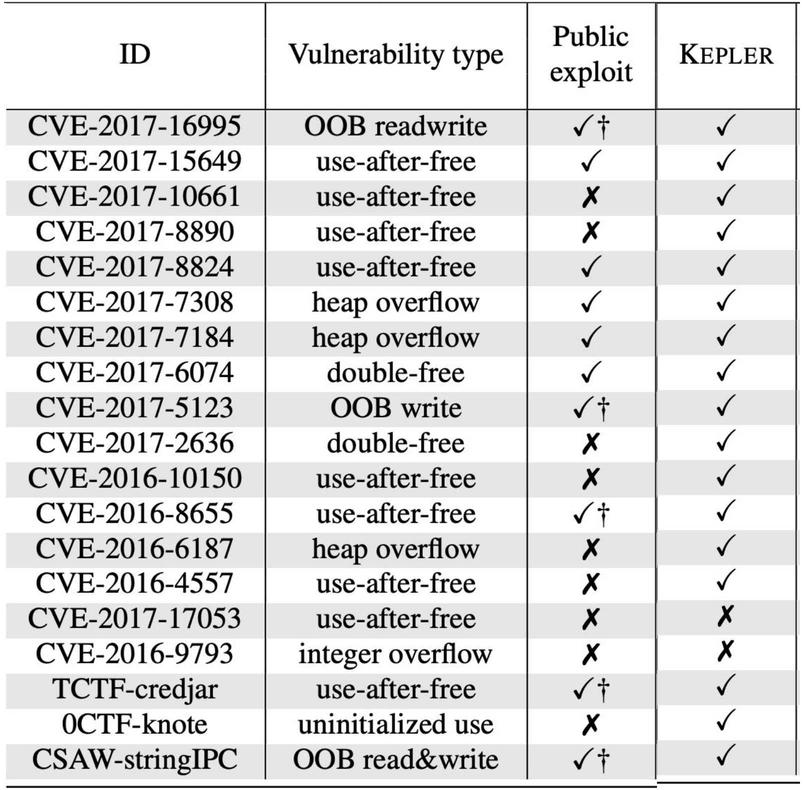
3

SMAP/SMEP is temporarily disabled during copy_from_user() which overflows kernel stack with ROP payload plus canary

4

Evaluation

- 16 CVEs + 3 CTF challenges as evaluation set
- KEPLER bypasses mitigations using control-flow hijacking primitives in 17 vulnerabilities



ID	Vulnerability type	Public exploit	KEPLER
CVE-2017-16995	OOB readwrite	✓†	✓
CVE-2017-15649	use-after-free	✓	✓
CVE-2017-10661	use-after-free	✗	✓
CVE-2017-8890	use-after-free	✗	✓
CVE-2017-8824	use-after-free	✓	✓
CVE-2017-7308	heap overflow	✓	✓
CVE-2017-7184	heap overflow	✓	✓
CVE-2017-6074	double-free	✓	✓
CVE-2017-5123	OOB write	✓†	✓
CVE-2017-2636	double-free	✗	✓
CVE-2016-10150	use-after-free	✗	✓
CVE-2016-8655	use-after-free	✓†	✓
CVE-2016-6187	heap overflow	✗	✓
CVE-2016-4557	use-after-free	✗	✓
CVE-2017-17053	use-after-free	✗	✗
CVE-2016-9793	integer overflow	✗	✗
TCTF-credjar	use-after-free	✓†	✓
0CTF-knote	uninitialized use	✗	✓
CSAW-stringIPC	OOB read&write	✓†	✓

Summary of KEPLER

Assumption

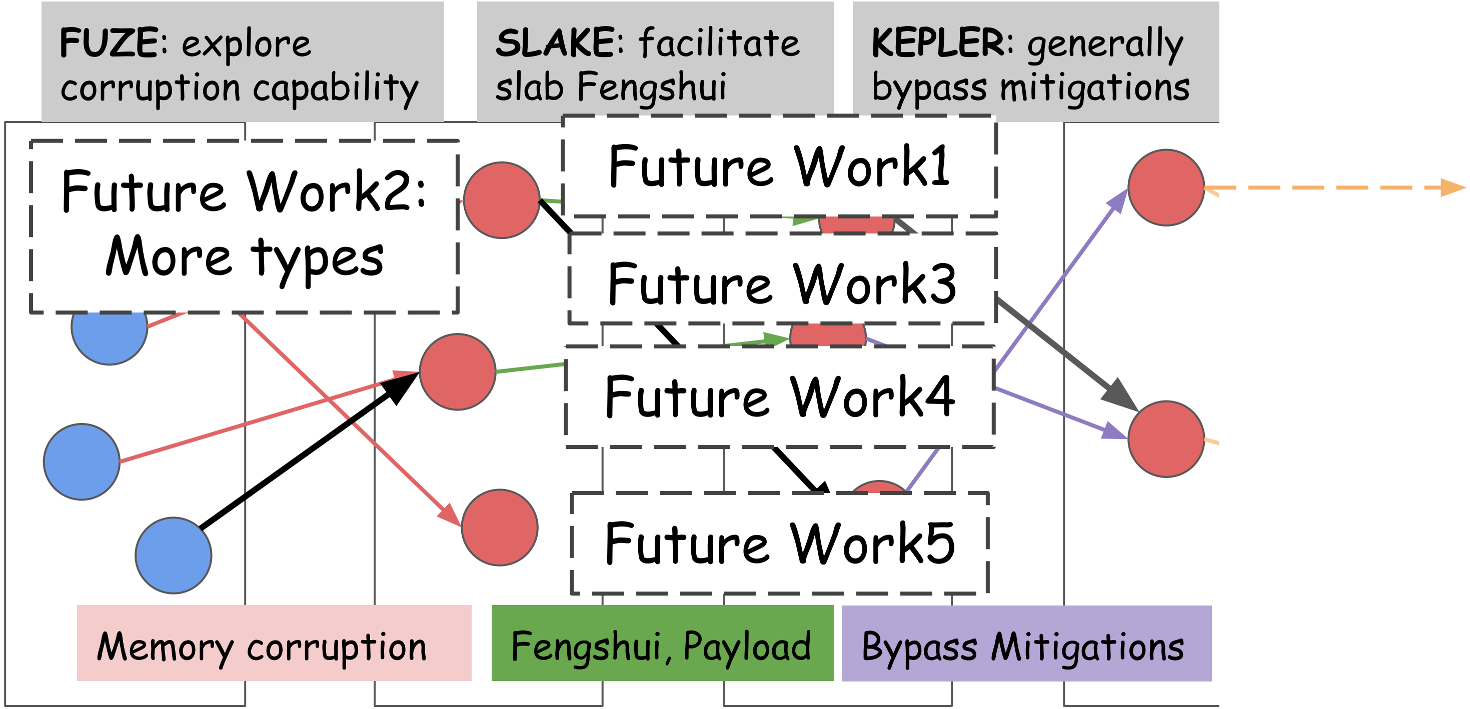
- KASLR can be bypassed via hardware side-channels
- Control flow hijacking primitive can be gained via FUZE/SLAKE
- SMAP/SMEP, stack canary, STATIC_USERMODEHELPER_PATH, non-executable physmap, hypervisor based cr4 protection are enabled mitigations.

Takeaway

- Given control-flow hijacking primitives, KEPLER bypasses default mitigations in Linux distros
- Bypassing mitigations escalates exploitability

Summary & Future Work

Our View of Exploit Development



1. Reduce the human effort in developing exploitation for Linux kernel
2. Escalate exploitability for more sound assessment and towards ground-truth

Thank You

Contact

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