Escalate Exploitability for More Secure Software Systems

Yueqi Chen

Advisor: Xinyu Xing
The Pennsylvania State University
October 9th, 2020

To Secure Software Systems is Important, Especially Today


Cyberwar between nations

Info leaking of enterprises

Crimes against individuals

Approaches Towards More Secure Software Systems

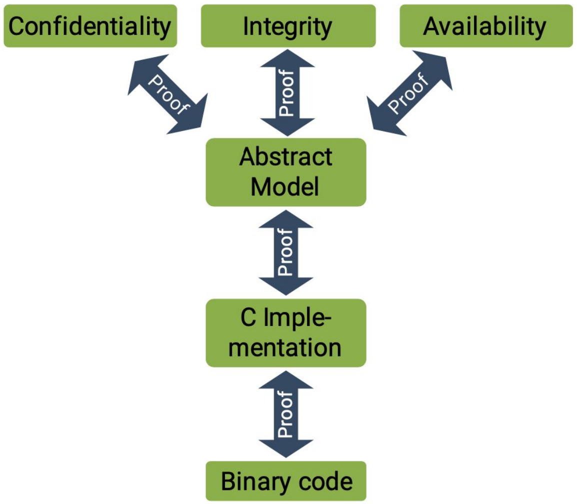


Figure 3.1: seL4's proof chain.

Approach 1: Formal verification

- E.g., seL4
- Proof between C implementation and binary code

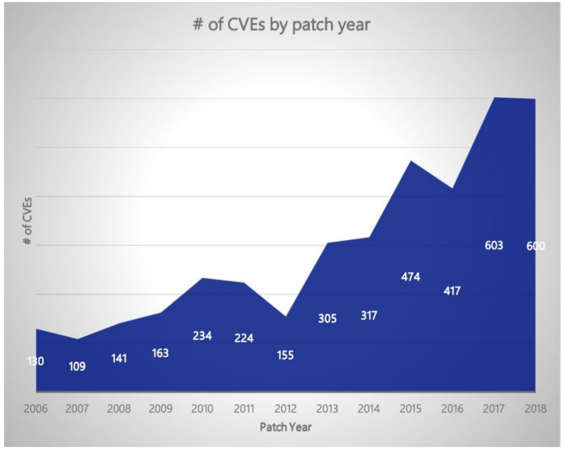
Critical Problems

- Clearly define trust/threat model
- Correctly write the underlying specifications

Exploitability is the key concept

[1] The source of Figure 3.1: Gernot Heiser, "The seL4 Microkernel - An Introduction"

Approaches Towards More Secure Software Systems



Approach 2: Eradicate all security bugs

- E.g., code auditing, static analysis, fuzzing, etc.
- The # of CVEs increases by year

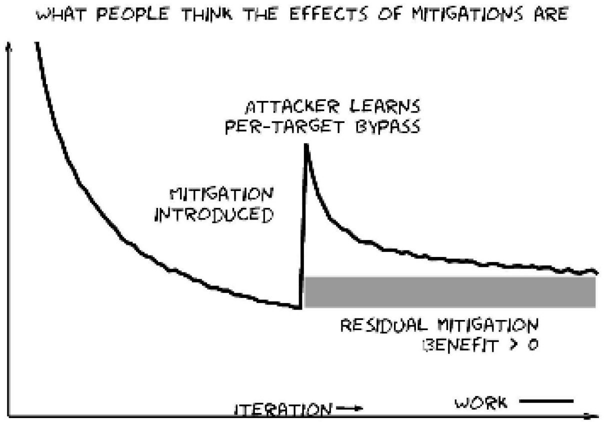
Critical Problems

- Prioritize the bug patching
- Get rid of incomplete/incorrect patch

Exploitability is the key concept

[2] The source of the figure: Matt Miller, MSRC, “Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape”

Approaches Towards More Secure Software Systems



Approach 3: Software systems guard themselves

- E.g., control flow restrictions, partitioning
- False estimate of benefit / cost

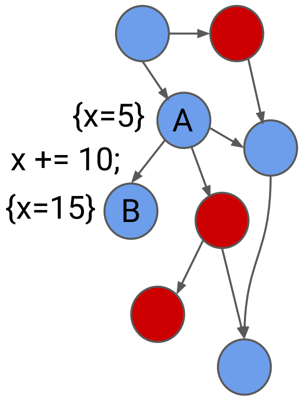
Critical Problems

- Justify for mitigations proposal
- Quantify the security improvement

Exploitability is the key concept

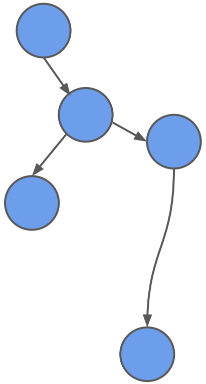
[3] The source of the figure: halvar.flake (Thomas Dullien), “Before you ship a security mitigation”

Vulnerability Exploitation - State Machine's Perspective



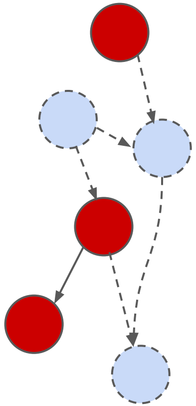
The State Machine of
A Vulnerable Software

=



Good Machine of
the Software

+

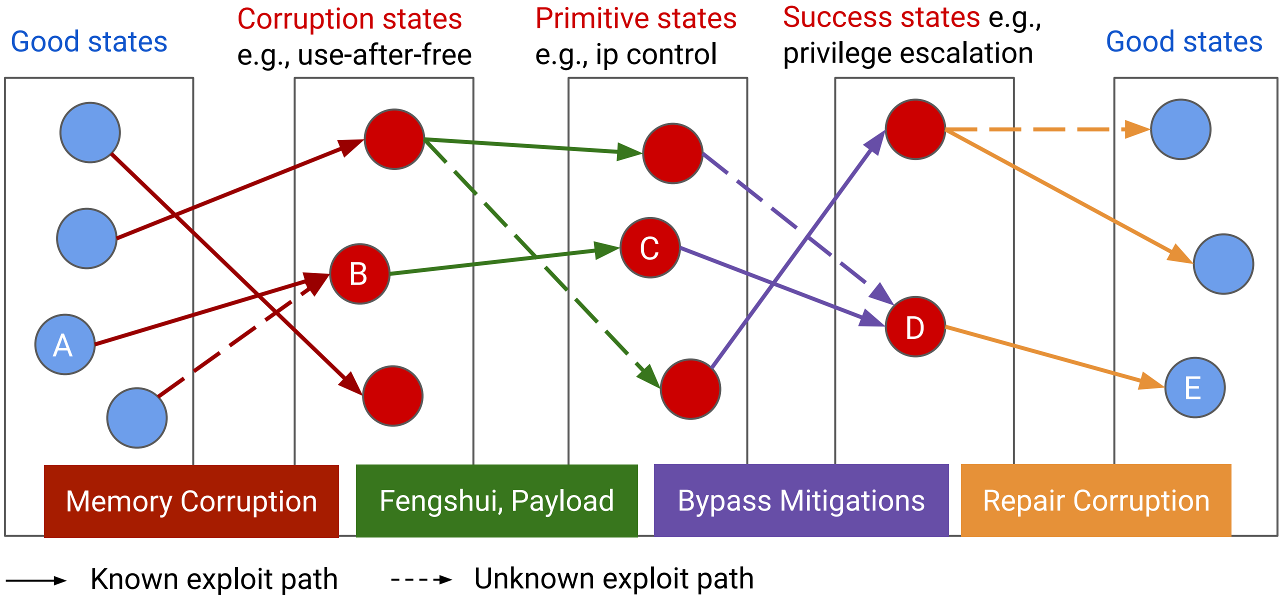


Weird Machine of
the Vulnerability

Viewpoint: Exploitation is programming weird machine

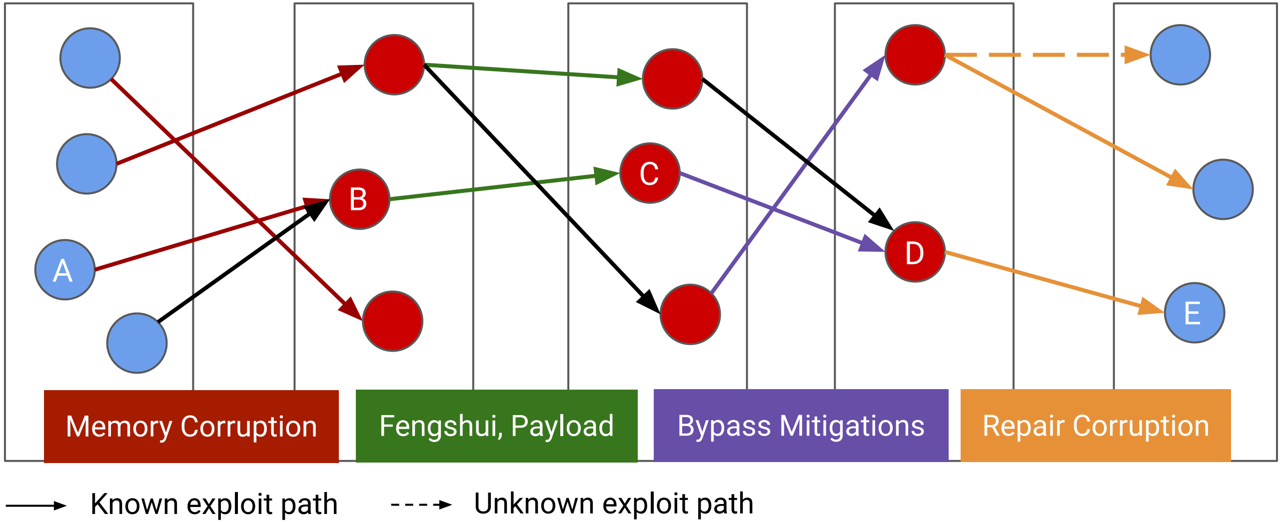
[4] Thomas Dullien, "Weird Machines, Exploitability, and Provable Unexploitability"

Our View of Exploit Development



Exploitability: whether there is a path from "left" to "right" (e.g., A→ B → C → D → E)
Ground-truth Exploitability: known + unknown exploit paths
Escalate exploitability: "solidate" unknown exploit paths

Our Previous Works in OS Kernel



FUZE:
explore capability

SLAKE:
facilitate Fengshui

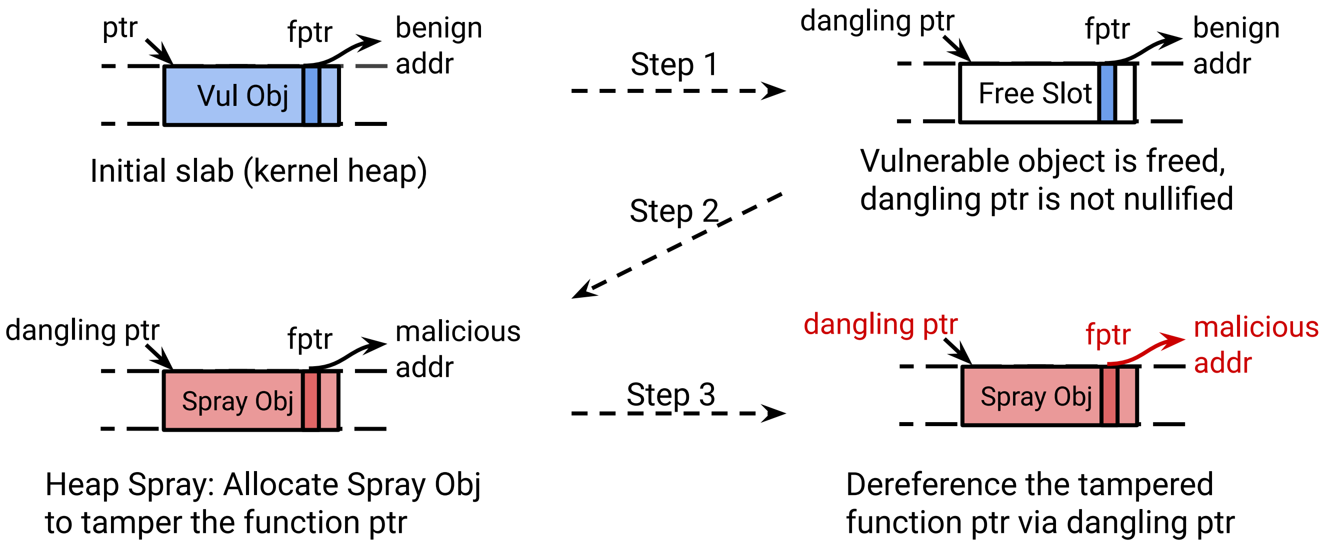
KEPLER & ELOISE:
bypass mitigations

Part I

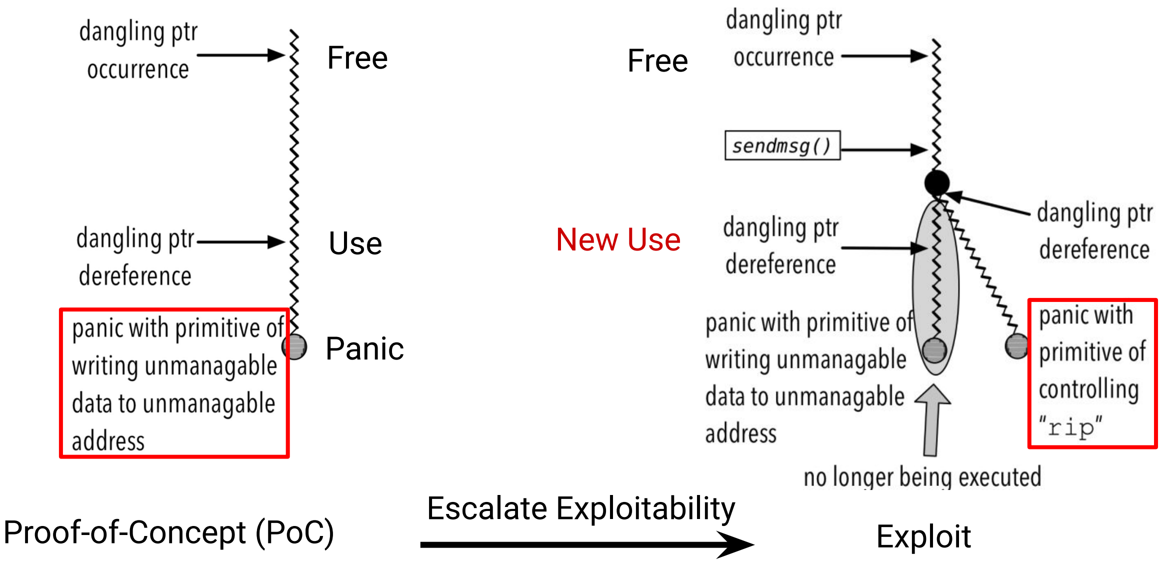
FUZE: Towards Facilitating Exploit Generation For Kernel Use-After-Free Vulnerabilities

USENIX Security 2018

Workflow of Use-After-Free Exploitation



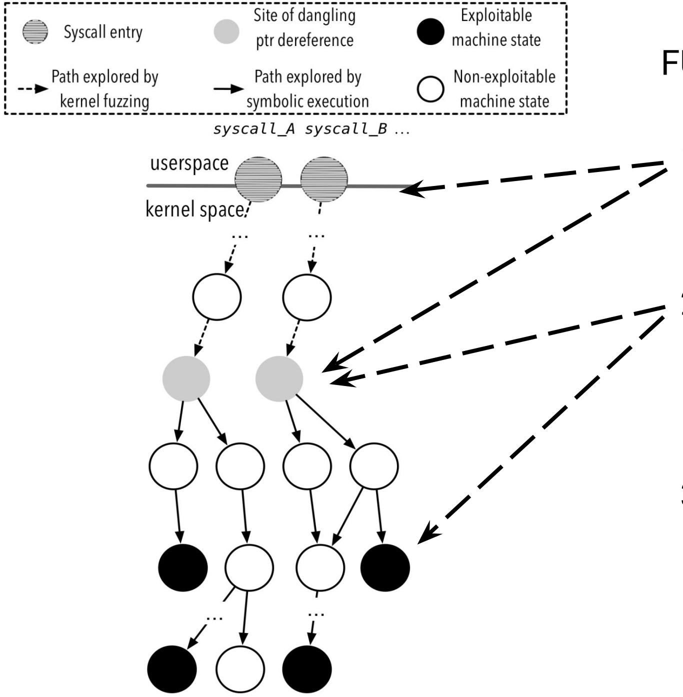
Challenges of Use-After-Free Exploitation



Challenges:

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 to make good use of the primitive?

Overview of FUZE



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., IP control, arbitrary read/write) can be obtained

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

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.

- 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

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

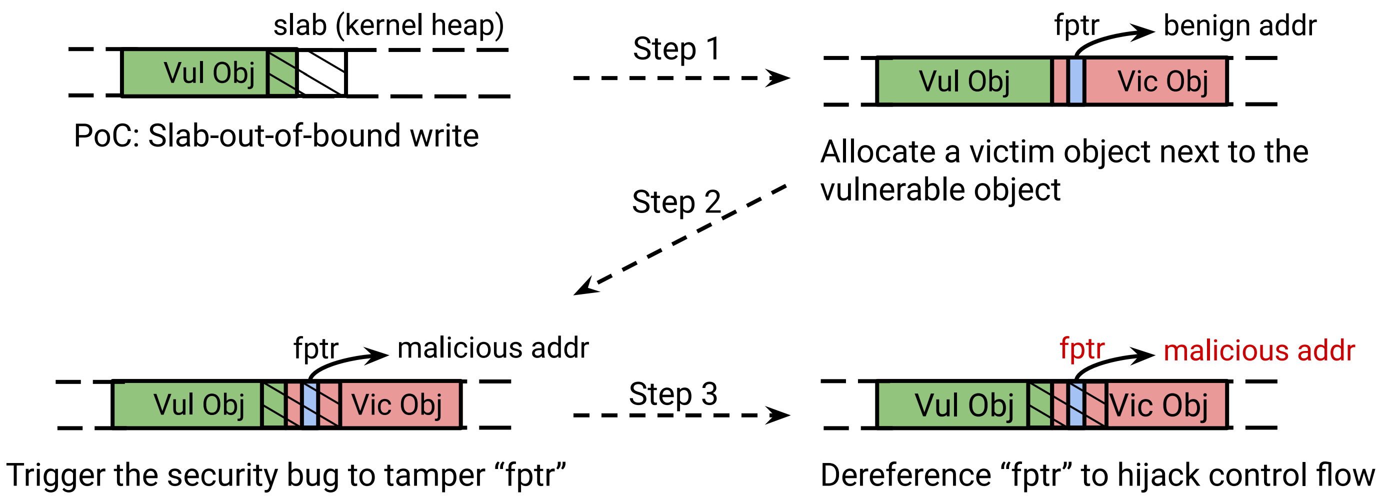
- For Use-After-Free vulnerabilities, new uses indicate more memory corruption capability
- More memory corruption capability escalates the exploitability

Part II

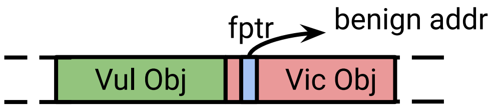
SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel

ACM CCS 2019

Workflow of Slab Out-of-bound Write Exploitation



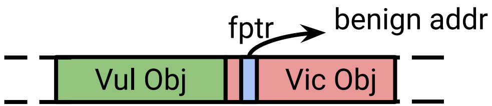
Shared Challenges of Slab Vulnerability Exploitation



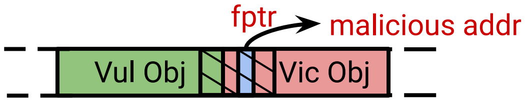
1. The **victim** object and **vulnerable** object are allocated to the same slab
2. The **vulnerable** object encloses a function pointer or other sensitive data

1. Which kernel object is useful for exploitation

Shared Challenges of Slab Vulnerability Exploitation



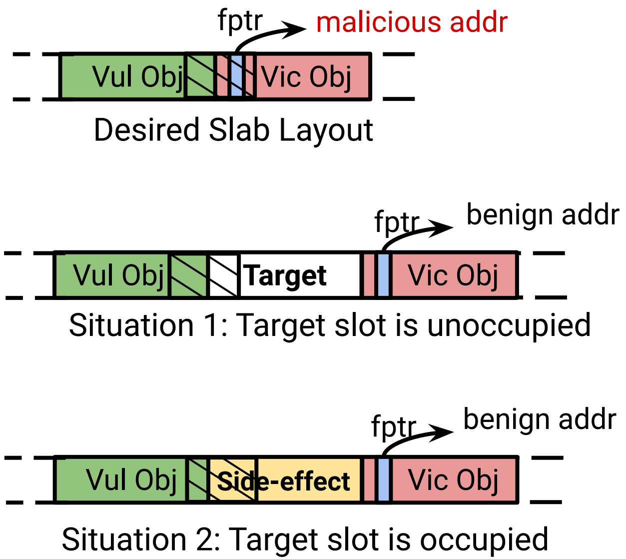
Allocate a victim object next to the vulnerable object



Dereference “fptr” to hijack control flow

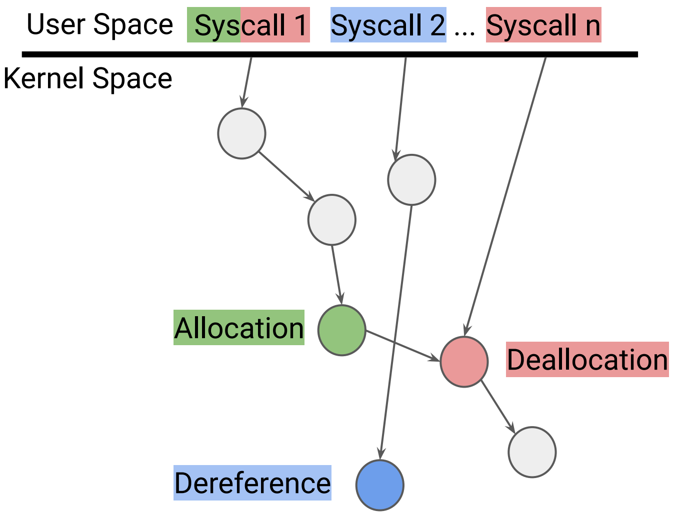
1. Which kernel object is useful for exploitation
2. How to (de)allocate and dereference useful objects

Shared 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

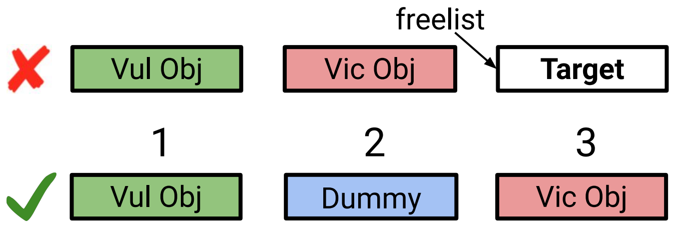
Overview of SLAKE - Resolving Challenge 1&2



Build a kernel object database via

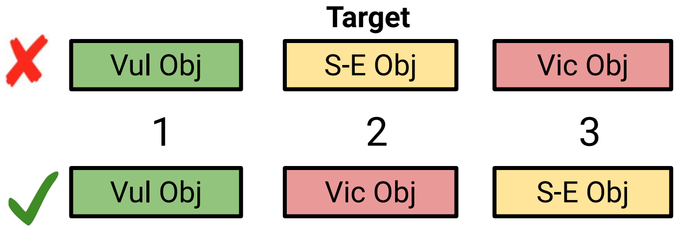
- Static Analysis to identify useful objects, sites of interest (allocation, deallocation, dereference), potential system calls
- Fuzzing Kernel to confirm System calls and complete arguments

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 the 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

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)	-	-	-

- 27 kernel vulnerabilities, including UAF, Double Free, OOB
- SLAKE obtains control-flow hijacking primitive in 15 cases with public exploits and 3 cases without public exploits.

Summary of SLAKE

Assumption - same as FUZE

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

Takeaway

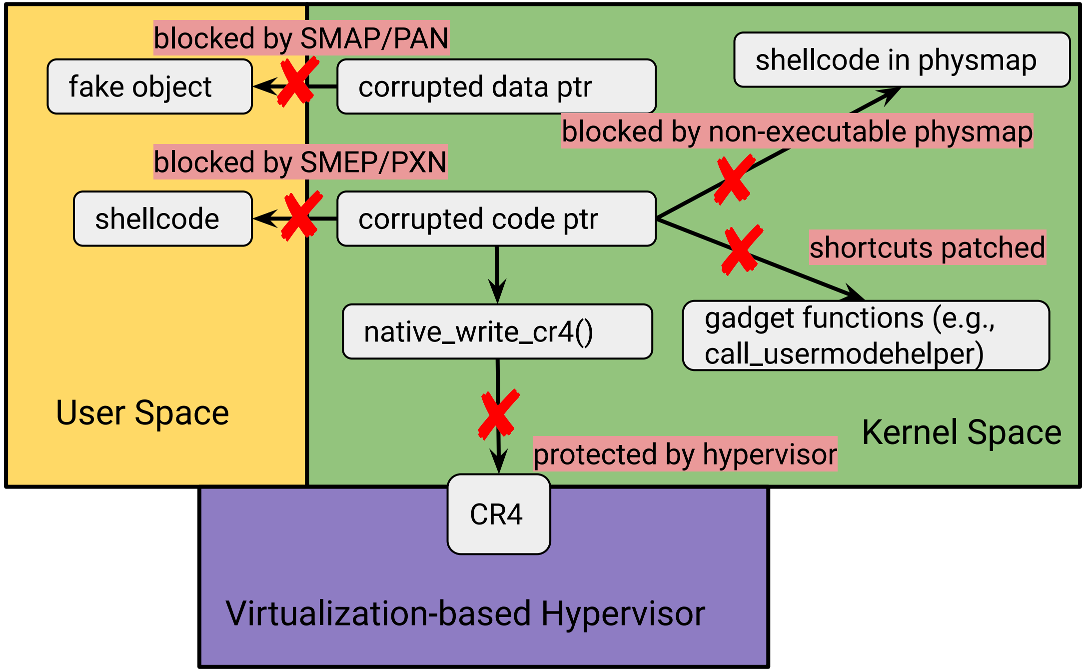
- More useful kernel objects and systematic Fengshui approach can bridge the gap between memory corruption and primitives
- Filling the gap not only diversifies the ways of performing kernel exploitation but also potentially escalates exploitability.

Part III

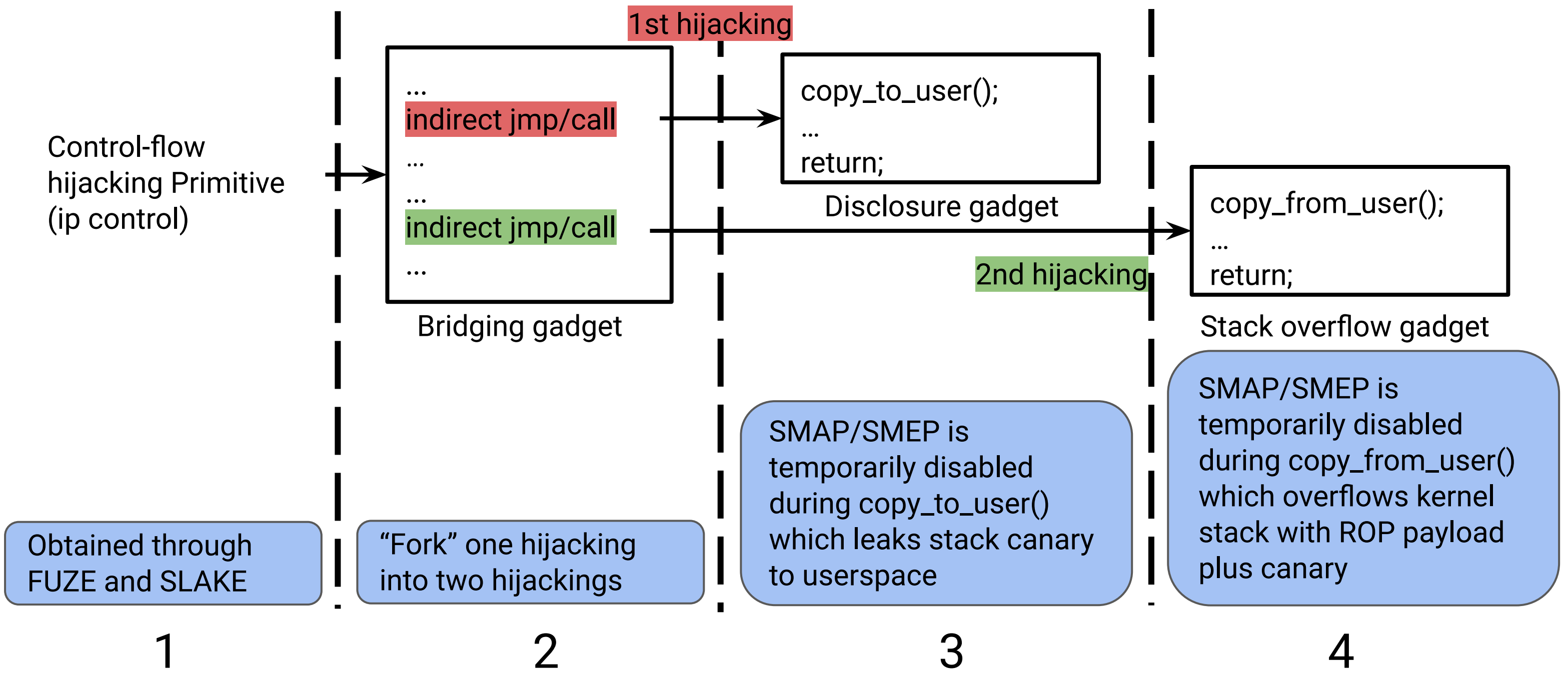
KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities

USENIX Security 2019

Mitigations in the Linux Kernel



Overview of KEPLER



Evaluation

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	✓†	✓

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

Summary of KEPLER

Assumption

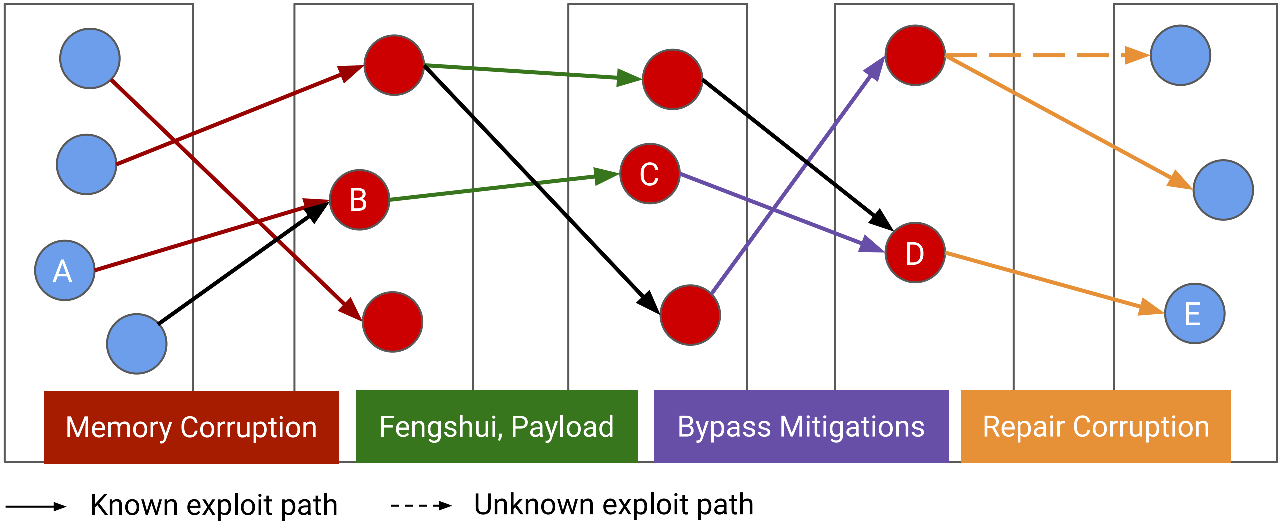
- KASLR can be bypassed via hardware side-channels
- Control flow hijacking primitive can be gained via FUZE/SLAKE
- SMAP/SMEP, stack canary, STATIC_USERMODEHELPER_PATH, non-executable physmap, hypervisor based cr4 protection are enabled mitigations

Takeaway

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

Contributions & Future Work

Contributions



FUZE:
explore capability

SLAKE:
facilitate Fengshui

KEPLER & ELOISE:
bypass mitigations

Future Work - Continue the Escalation

Future Work - Extend the Framework

Future Work - Build Better Mitigations

Thank You!

Contact

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