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

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

Figure 3.1: seL4’s proof chain.

[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

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

Vulnerability Exploitation - State Machine’s Perspective

The State Machine of A Vulnerable Software

\[
\begin{align*}
\{x=5\} & \quad x += 10; \\
\{x=15\} &
\end{align*}
\]

Good Machine of the Software

Weird Machine of the Vulnerability

Viewpoint: Exploitation is programming weird machine

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

![Diagram showing exploitability and ground-truth exploitability](image-url)
Our Previous Works in OS Kernel

- Memory Corruption
- Fengshui, Payload
- Bypass Mitigations
- Repair Corruption

**FUZE:** explore capability

**SLAKE:** facilitate Fengshui

**KEPLER & ELOISE:** bypass mitigations

Known exploit path — Unknown exploit path
Part I

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

USENIX Security 2018
Workflow of Use-After-Free Exploitation

Step 1
Vulnerable object is freed, dangling ptr is not nullified

Step 2
Vulnerable object is freed, dangling ptr is not nullified

Step 3
Dereference the tampered function ptr via dangling ptr

Heap Spray: Allocate Spray Obj to tamper the function ptr
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

<table>
<thead>
<tr>
<th>CVE-ID</th>
<th># of public exploits</th>
<th># of generated exploits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SMEP</td>
<td>SMAP</td>
</tr>
<tr>
<td>2017-17053</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2017-15649</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2017-15265</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2017-10661</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2017-8890</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2017-8824</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2017-7374</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2016-10150</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2016-8655</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2016-7117</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2016-4557</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2016-0728</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2015-3636</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2014-2851</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2013-7446</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Overall</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

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
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

PoC: Slab-out-of-bound write

Step 1
Allocate a victim object next to the vulnerable object

Step 2
Trigger the security bug to tamper “fptr”

Step 3
Dereference “fptr” to hijack control flow

Vul Obj
Vic Obj

slab (kernel heap)

benign addr

malicious addr

fptr
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

<table>
<thead>
<tr>
<th>CVE-ID</th>
<th>Type</th>
<th>Exploitation Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>I</td>
</tr>
<tr>
<td>N/A[47]</td>
<td>OOB</td>
<td>5 (1*)</td>
</tr>
<tr>
<td>2010-2959</td>
<td>OOB</td>
<td>13 (1*)</td>
</tr>
<tr>
<td>2018-6555</td>
<td>UAF</td>
<td>-</td>
</tr>
<tr>
<td>2017-1000112</td>
<td>OOB</td>
<td>0 (1)</td>
</tr>
<tr>
<td>2017-2636</td>
<td>double free</td>
<td>-</td>
</tr>
<tr>
<td>2014-2851</td>
<td>UAF</td>
<td>-</td>
</tr>
<tr>
<td>2015-5636</td>
<td>UAF</td>
<td>-</td>
</tr>
<tr>
<td>2016-0728</td>
<td>UAF</td>
<td>-</td>
</tr>
<tr>
<td>2016-10150</td>
<td>UAF</td>
<td>-</td>
</tr>
<tr>
<td>2016-4557</td>
<td>UAF</td>
<td>-</td>
</tr>
<tr>
<td>2016-6187</td>
<td>OOB</td>
<td>-</td>
</tr>
<tr>
<td>2016-8655</td>
<td>UAF</td>
<td>-</td>
</tr>
<tr>
<td>2017-10661</td>
<td>UAF</td>
<td>-</td>
</tr>
<tr>
<td>2017-15649</td>
<td>UAF</td>
<td>-</td>
</tr>
<tr>
<td>2017-17052</td>
<td>UAF</td>
<td>-</td>
</tr>
<tr>
<td>2017-17053</td>
<td>double free</td>
<td>-</td>
</tr>
<tr>
<td>2017-6074</td>
<td>double free</td>
<td>-</td>
</tr>
<tr>
<td>2017-7184</td>
<td>OOB</td>
<td>10 (0)</td>
</tr>
<tr>
<td>2017-7308</td>
<td>OOB</td>
<td>14 (1)</td>
</tr>
<tr>
<td>2017-8824</td>
<td>UAF</td>
<td>3 (1)</td>
</tr>
<tr>
<td>2017-8890</td>
<td>double free</td>
<td>-</td>
</tr>
<tr>
<td>2018-10840</td>
<td>OOB</td>
<td>0 (0)</td>
</tr>
<tr>
<td>2018-12714</td>
<td>OOB</td>
<td>0 (0)</td>
</tr>
<tr>
<td>2018-16880</td>
<td>OOB</td>
<td>0 (0)</td>
</tr>
<tr>
<td>2018-17182</td>
<td>UAF</td>
<td>-</td>
</tr>
<tr>
<td>2018-18559</td>
<td>UAF</td>
<td>-</td>
</tr>
<tr>
<td>2018-5703</td>
<td>OOB</td>
<td>0 (0)</td>
</tr>
</tbody>
</table>

- 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

- Blocked by SMAP/PAN:
  - Fake object
  - Corrupted data ptr

- Blocked by SMEP/PXN:
  - Shellcode
  - Corrupted code ptr
  - Native_write_cr4()

- Shellcode in physmap:
  - Blocked by non-executable physmap
  - Shortcuts patched
  - Protected by hypervisor

- gadget functions (e.g., call_usermodehelper)

User Space

Kernel Space

Virtualization-based Hypervisor
Overview of KEPLER

Control-flow hijacking Primitive (ip control)

1st hijacking

... indirect jmp/call
... indirect jmp/call
... indirect jmp/call

Bridging gadget

copy_to_user();
... return;

Disclosure gadget

2nd hijacking

copy_from_user();
... return;

Stack overflow gadget

Obtained through FUZE and SLAKE

“Fork” one hijacking into two hijackings

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

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

<table>
<thead>
<tr>
<th>ID</th>
<th>Vulnerability type</th>
<th>Public exploit</th>
<th>KEPLER</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2017-16995</td>
<td>OOB readwrite</td>
<td>✓ †</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2017-15649</td>
<td>use-after-free</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2017-10661</td>
<td>use-after-free</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2017-8890</td>
<td>use-after-free</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2017-8824</td>
<td>use-after-free</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2017-7308</td>
<td>heap overflow</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2017-7184</td>
<td>heap overflow</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2017-6074</td>
<td>double-free</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2017-5123</td>
<td>OOB write</td>
<td>✓ †</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2017-2636</td>
<td>double-free</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-10150</td>
<td>use-after-free</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-8655</td>
<td>usec-after-free</td>
<td>✓ †</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-6187</td>
<td>heap overflow</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-4557</td>
<td>use-after-free</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2017-17053</td>
<td>use-after-free</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>CVE-2016-9793</td>
<td>integer overflow</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>TCTF-credjar</td>
<td>use-after-free</td>
<td>✓ †</td>
<td>✓</td>
</tr>
<tr>
<td>0CTF-knote</td>
<td>uninitialized use</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>CSAW-stringIPC</td>
<td>OOB read&amp;write</td>
<td>✓ †</td>
<td>✓</td>
</tr>
</tbody>
</table>

- 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

- Memory Corruption
- Fengshui, Payload
- Bypass Mitigations
- Repair Corruption

 FUZE: explore capability
 SLAKE: facilitate Fengshui
 KEPLER & ELOISE: bypass mitigations
Future Work - Continue the Escalation

- Repair Corruption
- Bypass Mitigations
- Exploring FUZE capability
- SLAKE to facilitate Fengshui
- KEPLER & ELOISE: bypass mitigations

Future work 1: Repairing
Future work 2: More types, e.g., in KOOBE
Future work 3: More payload injection, e.g., in leak-kptr
Future work 4: Stabilize the exploit
Future work 5: Prove unexploitable
Future Work - Extend the Framework

- Extend the Framework
- Memory Corruption
- Fengshui
- Payload Bypass Mitigations
- Repair Corruption

Future work 6: Exploit multiple unexploitable bugs
Future work 7: Append more exploitation phases, e.g., load malicious modules for APT attack
Future work 8: More vulnerability types other than memory corruption, e.g., semantic bugs
Future Work - Build Better Mitigations

- Future work 9: Study the generality and severity of an exploitation approach
- Future work 10: Quantify the security improvement of a proposed mitigation

News: We did some preliminary exploration in our recent CCS paper
Thank You!

Contact

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