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.

Xin Chen 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

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 states**
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

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

Step 2
- Heap Spray: Allocate Spray Obj to tamper the function ptr

Step 3
- 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?
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., 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

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

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
   - 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

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
   - 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

2. How to (de)allocate and dereference useful objects
   - System call sequence, arguments

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

Situation 1: Target slot is unoccupied
- benign addr
- fptr
- Vul Obj
- Target
- Vic Obj

Situation 2: Target slot is occupied
- benign addr
- fptr
- Vul Obj
- Side-effect
- Vic Obj
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
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.

<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>2015-3636</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>4 (1)</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>
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

- *User Space*
  - fake object
  - shellcode
  - corrupted data ptr
  - corrupted code ptr
  - native_write_cr4()

- *Kernel Space*
  - shellcode in physmap
  - gadget functions (e.g., call_usermodehelper)
  - blocked by non-executable physmap
  - shortcuts patched
  - protected by hypervisor

- *Virtualization-based Hypervisor*
  - CR4

- *Protected by Hypervisor*
  - SMAP/PAN
  - SMEP

Note: The diagram illustrates how different kernel elements are protected or blocked by various mechanisms to mitigate security vulnerabilities.
Overview of KEPLER

Control-flow hijacking Primitive

1. Obtained through FUZE and SLAKE
2. "Fork" one hijacking into two hijackings

Bridging gadget

1st hijacking

... indirect jmp/call
... 
2nd hijacking

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

Disclosure gadget

copy_to_user();
... return;

Stack overflow gadget

copy_from_user();
... return;
Evaluation

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

<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>use-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>
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

FUZE: explore corruption capability
SLAKE: facilitate slab Fengshui
KEPLER: generally bypass mitigations

Future Work1: More types
Future Work2: Memory corruption
Future Work3: Fengshui, Payload
Future Work4: Bypass Mitigations
Future Work5

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/