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

Wei Wu$^{1,2,3}$, Yueqi Chen$^2$, Jun Xu$^2$, Xinyu Xing$^2$, Xiaorui Gong$^{1,3}$, and Wei Zou$^{1,3}$

1. School of Cyber Security, University of Chinese Academy of Sciences
2. College of Information Sciences and Technology, Pennsylvania State University
3. CAS-KLONAT, BKLONSPT, Institute of Information Engineering

27th Usenix Security Symposium
August, 16th, 2018
What are We Talking about?

• Discuss the challenge of exploit development
• Introduce an approach to facilitate exploit development
• Demonstrate how the new technique facilitate mitigation circumvention
Background

• All software contain bugs, and # of bugs grows with the increase of software complexity
  • E.g., Syzkaller/Syzbot reports 800+ Linux kernel bugs in 8 months
• Due to the lack of manpower, it is very rare that a software development team could patch all the bugs timely
  • E.g., A Linux kernel bug could be patched in a single day or more than 8 months; on average, it takes 42 days to fix one kernel bug

• The best strategy for software development team is to prioritize their remediation efforts for bug fix
  • E.g. based on its influence upon usability
  • E.g., based on its influence upon software security
  • E.g., based on the types of the bugs
  • ……. 
Background (cont.)

• Most common strategy is to fix a bug based on its exploitability
• To determine the exploitability of a bug, analysts generally have to write a working exploit, which needs
  1) Significant manual efforts
  2) Sufficient security expertise
  3) Extensive experience in target software
Crafting an Exploit for Kernel Use-After-Free

- Dangling ptr occurrence
- syscall_A(…)
- syscall_B(…)
- Freed object
- syscall_S(…)
- Heap spray
- Object carefully selected
- Proper time window to perform heap spray
- Dangling ptr dereference
- kernel panic

1. Use control over program counter (rip) to hijack control flow
2. Use the ability to write arbitrary content to arbitrary address to escalate privilege
3. …
Challenge 1: Needs Intensive Manual Efforts

• Analyze the kernel panic
• Manually track down
  1. The site of dangling pointer occurrence and the corresponding system call
  2. The site of dangling pointer dereference and the corresponding system call
Challenge 2: Needs Extensive Expertise in Kernel

- Identify all the candidate objects that can be sprayed to the region of the freed object
- Pinpoint the proper system calls that allow an analyst to perform heap spray
- Figure out the proper arguments and context for the system call to allocate the candidate objects
Challenge 3: Needs Security Expertise

- Find proper approaches to accomplish arbitrary code execution or privilege escalation or memory leakage
  - E.g., chaining ROP
  - E.g., crafting shellcode
  - …

1. Use control over program counter (rip) to perform arbitrary code execution
2. Use the ability to write arbitrary content to arbitrary address to escalate privilege
3. …
Some Past Research Potentially Tackling the Challenges

• Approaches for Challenge 1
  • Nothing I am aware of, but simply extending KASAN could potentially solve this problem

• Approaches for Challenge 2
  • [Blackhat07] [CCS’ 16] [USENIX-SEC18],...

• Approaches for Challenge 3
  • [NDSS’11] [S&P16], [S&P17],...

[CCS 16] Xu et al., From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel.
[S&P16] Shoshitaishvili et al., Sok:(state of) the art of war: Offensive techniques in binary analysis.
[Blackhat07] Sotirov, Heap Feng Shui in JavaScript
Some Past Research Potentially Tackling the Challenges

- Approaches for Challenge 1
  - Nothing I am aware of, but simply extending KASAN could potentially solve this problem

- Approaches for Challenge 2
  - [Blackhat07], [CCS’16], [USENIX-SEC16]

- Approaches for Challenge 3
  - [NDSS’11], [S&P16], [S&P17]

Problem unsolved.

[CCS 16] Xu et al., From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel.
[S&P16] Shoshitaishvili et al., Sok:(state of) the art of war: Offensive techniques in binary analysis.
[Blackhat07] Sotirov, Heap Feng Shui in JavaScript
Roadmap

• Unsolved challenges in exploitation facilitation
• Our techniques -- FUZE
• Evaluation with real-world Linux kernel vulnerabilities
• Conclusion
A Real-World Example (CVE-2017-15649)

```c
void *task1(void *unused) {
    ... 
    int err = setsockopt(fd, 0x107, 18, ..., ...);
    ...
}

void *task2(void *unused) {
    int err = bind(fd, &addr ...);
    ...
}

void loop_race() {
    ...
    while(1) {
        fd = socket(AF_PACKET, SOCK_RAW,
        htons(ETH_P_ALL));
        ...
        //create two racing threads
        pthread_create (&thread1, NULL,
        task1, NULL);
        pthread_create (&thread2, NULL,
        task2, NULL);
        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);
        close(fd);
    }
}```
A Real-World Example (CVE-2017-15649)

close(…) free node but not completely removed from the list

Head node

dangling ptr

```
void *task1(void *unused) {
  
  int err = setsockopt(fd, 0x017, 18, ..., ...);
}

void *task2(void *unused) {
  int err = bind(fd, &addr ...);
}

void loop_race() {
  
  while(1) {
    fd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
    
    // create two racing threads
    pthread_create (&thread1, NULL, task1, NULL);
    pthread_create (&thread2, NULL, task2, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    close(fd);
  }
}
```
Challenge 4: No Primitive Needed for Exploitation

Head node

Node newly crafted

Obtain an ability to write unmanageable data to unmanageable address

next
prev
next
prev
next
prev
dangling ptr

void *task1(void *unused) {
    ...
    int err = setsockopt(fd, 0x107, 18, ..., ...);
}

void *task2(void *unused) {
    int err = bind(fd, &addr ...);
}

void loop_race() {
    ...
    while(1) {
        fd = socket(AF_PACKET, SOCK_RAW,
                   htons(ETH_P_ALL));
        ...
        // create two racing threads
        pthread_create(&thread1, NULL,
                        task1, NULL);
        pthread_create(&thread2, NULL,
                        task2, NULL);
        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);
        close(fd);
    }
}
No Useful Primitive == Unexploitable??

Dangling ptr occurrence

Obtain the primitive – write unmanageable data to unmanageable region

Dangling ptr dereference

Obtain the primitive – hijack control flow (control over rip)

kernel panic

sendmsg(…)

```c
void *task1(void *unused) {
    ... 
    int err = setsockopt(fd, 0x107, 18, ...); 
}

void *task2(void *unused) {
    int err = bind(fd, &addr,...); 
}

void loop_race() {
    ... 
    while(1) {
        fd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
        ... 
        //create two racing threads
        pthread_create (&thread1, NULL, task1, NULL);
        pthread_create (&thread2, NULL, task2, NULL);
        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);
        close(fd);
    }
}
```
Roadmap

• Unsolved challenges in exploitation facilitation
• Our techniques -- FUZE
• Evaluation with real-world Linux kernel vulnerabilities
• Conclusion
FUZE – Extracting Critical Info.

• Identifying the site of dangling pointer occurrence, and that of its dereference; pinpointing the corresponding system calls
FUZE – Performing Kernel Fuzzing

• Identifying the site of dangling pointer occurrence, and that of its dereference; pinpointing the corresponding system calls

• Performing kernel fuzzing between the two sites and exploring other panic contexts (i.e., different sites where the vulnerable object is dereferenced)
FUZE – Performing Symbolic Execution

- Identifying the site of dangling pointer occurrence, and that of its dereference; pinpointing the corresponding system calls
- Performing kernel fuzzing between the two sites and exploring other panic contexts (i.e., different sites where the vulnerable object is dereferenced)
- Symbolically execute at the sites of the dangling pointer dereference

Set symbolic value for each byte
Useful primitive identification

• Unconstrained state
  • state with symbolic Instruction pointer
  • symbolic callback
• double free
  • e.g. mov rdi, uaf_obj; call kfree
• write-what-where
  • e.g. write arbitrary value write

mov rax, qword ptr[evil_ptr]
call rax

stack pivot gadget:
  xchg eax, esp; ret

SMAP disable gadget:
  mov cr4, rdi ; ret
Roadmap

• Unsolved challenges in exploitation facilitation
• Our techniques -- FUZE
• Evaluation with real-world Linux kernel vulnerabilities
• Conclusion
### Case Study

- 15 real-world UAF kernel vulnerabilities
- Only 5 vulnerabilities have demonstrated their exploitability against SMEP
- Only 2 vulnerabilities have demonstrated their exploitability against SMAP

<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>

*: discovered new dereference by fuzzing
Case Study (cont)

- FUZE helps track down useful primitives, giving us the power to
  - Demonstrate exploitability against SMEP for 10 vulnerabilities
  - Demonstrate exploitability against SMAP for 2 more vulnerabilities
  - Diversify the approaches to perform kernel exploitation
    - 5 vs 19 (SMEP)
    - 2 vs 5 (SMAP)

<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>
Discussion on Failure Cases

• Dangling pointer occurrence and its dereference tie to the same system call
• FUZE works for 64-bit OS but some vulnerabilities demonstrate its exploitability only for 32-bit OS
  • E.g., CVE-2015-3636
• Perhaps unexploitable!?
  • CVE-2017-7374 ← null pointer dereference
  • E.g., CVE-2013-7446, CVE-2017-15265 and CVE-2016-7117
Roadmap

• Unsolved challenges in exploitation facilitation
• Our techniques -- FUZE
• Evaluation with real-world Linux kernel vulnerabilities
• Conclusion
Conclusion

• Primitive identification and security mitigation circumvention can greatly influence exploitability
• Existing exploitation research fails to provide facilitation to tackle these two challenges
• Fuzzing + symbolic execution has a great potential toward tackling these challenges
• Research on exploit automation is just the beginning of the GAME! Still many more challenges waiting for us to tackle…
Thank you!

- Exploits and source code available at:
  - [https://github.com/ww9210/Linux_kernel_exploits](https://github.com/ww9210/Linux_kernel_exploits)

- Contact: wuwei@iie.ac.cn
Questions

FUZE

User space

Kernel space

syscall_A

syscall_B

syscall_C

syscall_D

syscall_E

syscall_M

FUZE