Expediting Exploitability Assessment through an Exploitation Facilitation Framework

Xinyu (X.Y.) Xing

JD.com
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

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

Freed object

Heap spray

syscall_M(...)

Object carefully selected
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] [CCS16] [USENIX-SEC18]

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

[CCS16] Xu et al., 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)

```
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, 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);
}
Challenge 4: No Primitive Needed for Exploitation

Obtain an ability to write unmanageable data to unmanageable address

Node newly crafted

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

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

void loop_race() {
  ...
  while(1) {
    fd = socket(AF_PACKET, SOCK_RAW, htonl(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(...)

```
void *task1(void *unused) {
  ...
  int err = setsockopt(2, 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.

[Diagram showing User space and Kernel space with system calls syscall_A, syscall_B, and syscall_M, and a Freed object indicated]
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

Freed object

Set symbolic value for each byte
Useful Primitives for Control flow hijack

• Control flow hijack primitive
  • call rax where rax = sym. val.

• Double Free

• Memory leak
  • e.g. invocation of copy_to_user(…) with src point to a freed object

• linked list corruption
Useful Primitives for Write-what-where

• E.g., mov qword ptr [rdi], rsi

<table>
<thead>
<tr>
<th>rdi (dst)</th>
<th>rsi (src)</th>
<th>primitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbolic</td>
<td>symbolic</td>
<td>arbitrary write (qword shoot)</td>
</tr>
<tr>
<td>symbolic</td>
<td>concrete</td>
<td>write fixed value to arbitrary address</td>
</tr>
<tr>
<td>free chunk</td>
<td>any</td>
<td>write to freed object</td>
</tr>
<tr>
<td>x(concrete)</td>
<td>x(concrete)</td>
<td>self-reference structure</td>
</tr>
<tr>
<td>metadata of freed chunk</td>
<td>any</td>
<td>meta-data corruption</td>
</tr>
</tbody>
</table>

User space
Kernel space

Syscall_A
Syscall_B
Syscall_M

rdi (dst) = dst
rsi (src) = src
primitive

11/1/18 Email: xxing@ist.psu.edu
Useful Primitives != Ability to Perform Exploitation

- SMEP
- SMAP
- Hypervisor
- CFI
- KASLR
- KPTI
- heap metadata hardening
- read-only credentials
- read-only vdso
- read-only vsyscall
Exploitable Machine States

• A machine state with the ability to bypass SMEP
  • Control over rip which could redirect execution to pivot gadget -- xchg eax, esp
  • E.g., mov rax, qword ptr[evil_ptr]; call rax

• A machine state with the ability to bypass SMAP/SMEP
  • Control over rip which could redirect execution to native_write_cr4(...)
  • Also, control over rdi, rsi and rax
Roadmap

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

• 15 real-world UAF kernel vulnerabilities
• Only 5 vulnerabilities have demonstrated their exploitability against SMEP
• Only 2 vulnerabilities have demonstrated their exploitability against SMAP
Evaluation (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 performing kernel exploitation
    • 5 vs 19 (SMEP)
    • 2 vs 5 (SMAP)
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…