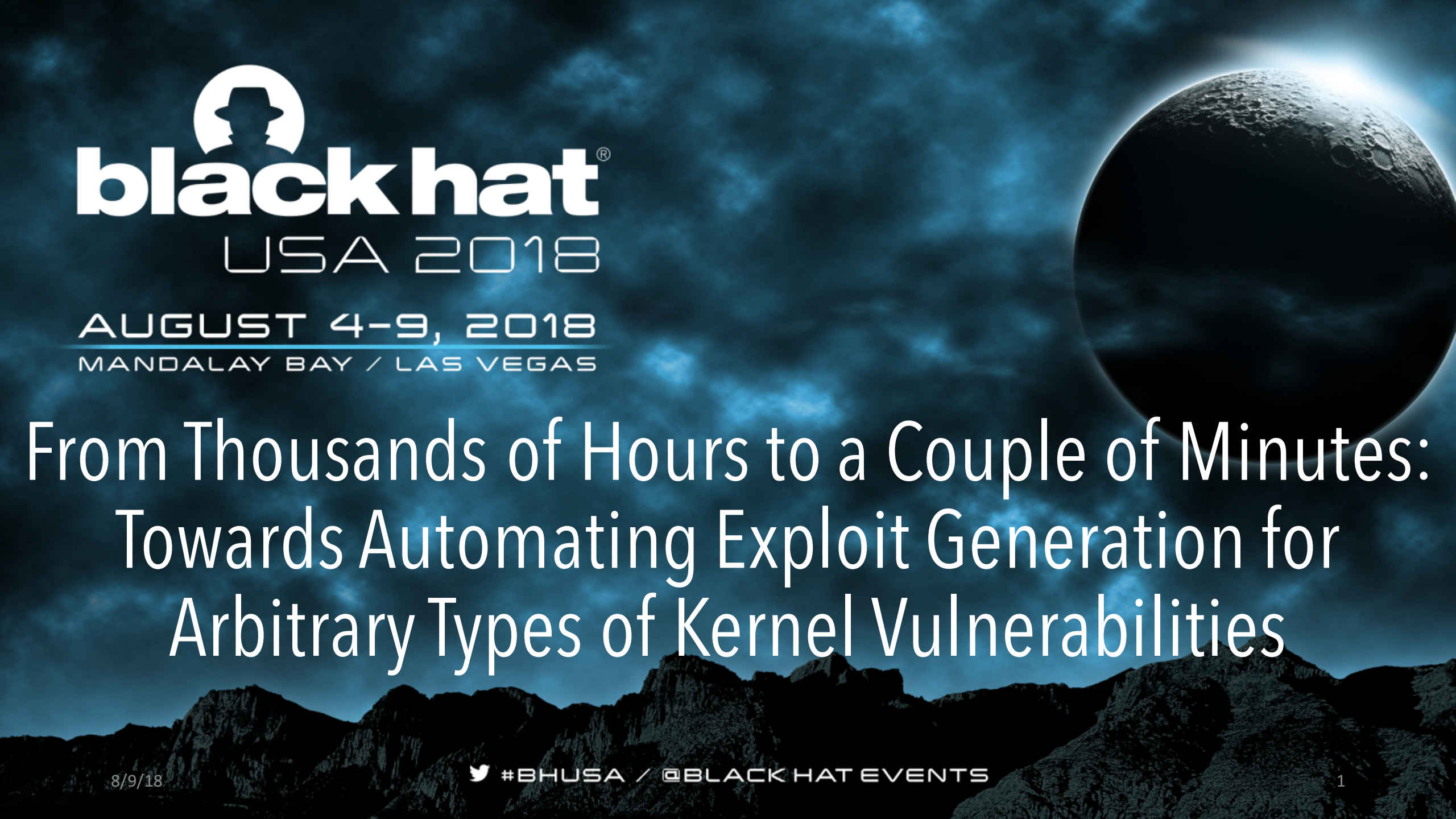


From Thousands of Hours to a Couple of Minutes: Towards Automating Exploit Generation for Arbitrary Types of Kernel Vulnerabilities

Who are We?

- Wei Wu @wu_xiao_wei
  - Visiting scholar at JD.com
    - Conducting research on software security in Enterprise Settings
  - Visiting Scholar at Penn State University
    - Vulnerability analysis
    - Memory forensics
    - Malware dissection
    - Reverse engineering
    - Symbolic execution
    - Static analysis
  - Final year PhD candidate at UCAS
    - Knowledge-driven vulnerability analysis
  - Co-founder of CTF team Never Stop Exploiting.(2015)
    - ctftime 2017 ranking 4th team in China
  - I am on market.

NSA Codebreaker Challenge

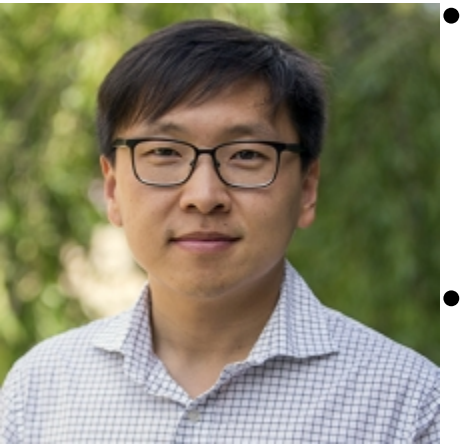
University
Carnegie Mellon University
Lafayette College
University of Hawaii
Pennsylvania State University
Georgia Institute of Technology

China

Position	Country position	Name	Points	Events
21	1	eee	365.576	16
24	2	A*0*E	332.972	6
27	3	0ops	278.460	18
30	4	Never Stop Exploiting	247.073	9
34	5	Azure Assassin Alliance	235.876	25

Who are We? (cont)

Xinyu Xing



- Visiting scholar at JD.com
  - Conducting research on software and hardware security in Enterprise Settings
- Assistant Professor at Penn State University
  - Advising PhD students and conducting many research projects on
    - Vulnerability identification
    - Vulnerability analysis
    - Exploit development facilitation
    - Memory forensics
    - Deep learning for software security
    - Binary analysis
    - …

Jimmy Su

- Head of JD security research center
  - Vulnerability identification and exploitation in Enterprise Settings
  - Red Team
  - JD IoT device security assessments
  - Risk control
  - Data security
  - Container security

What are We Talking about?

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

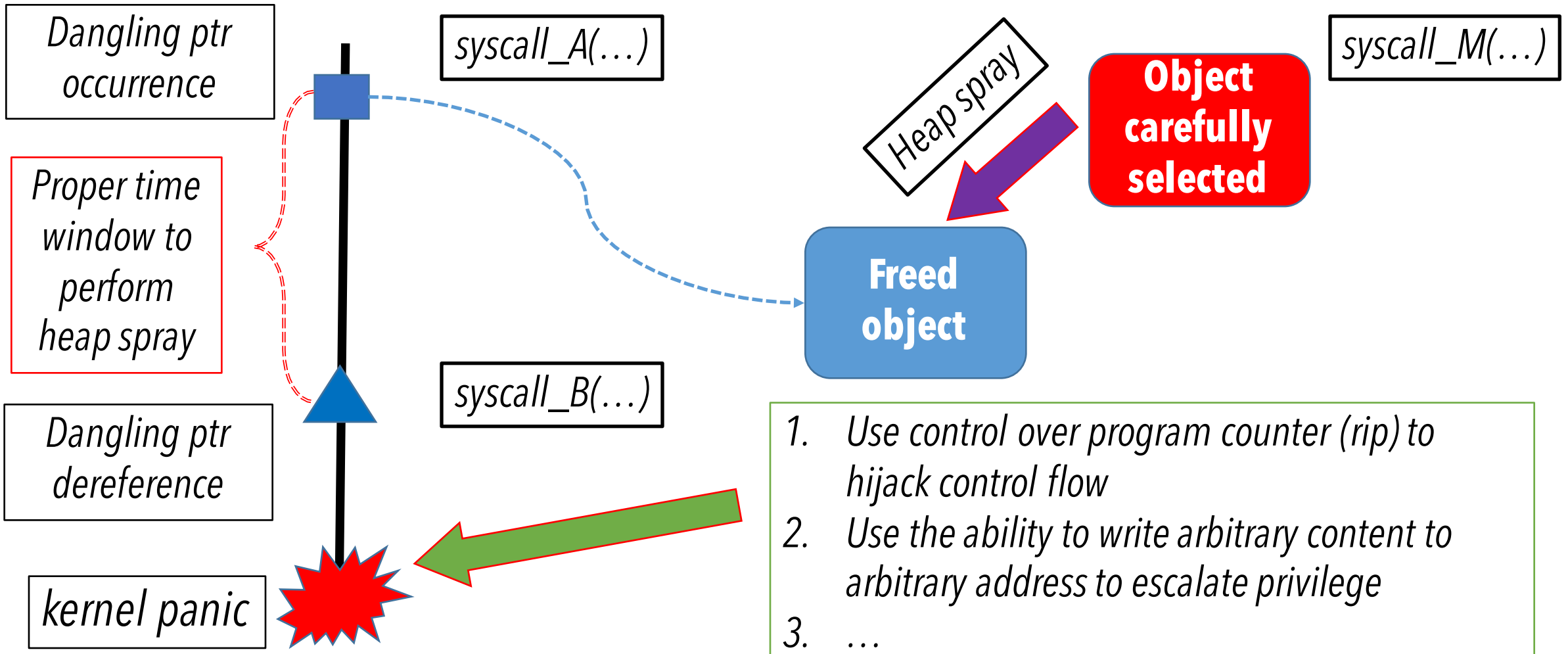
Background

- 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
- In the blue team, you need to "attack" with exploitable vulnerabilities
- 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

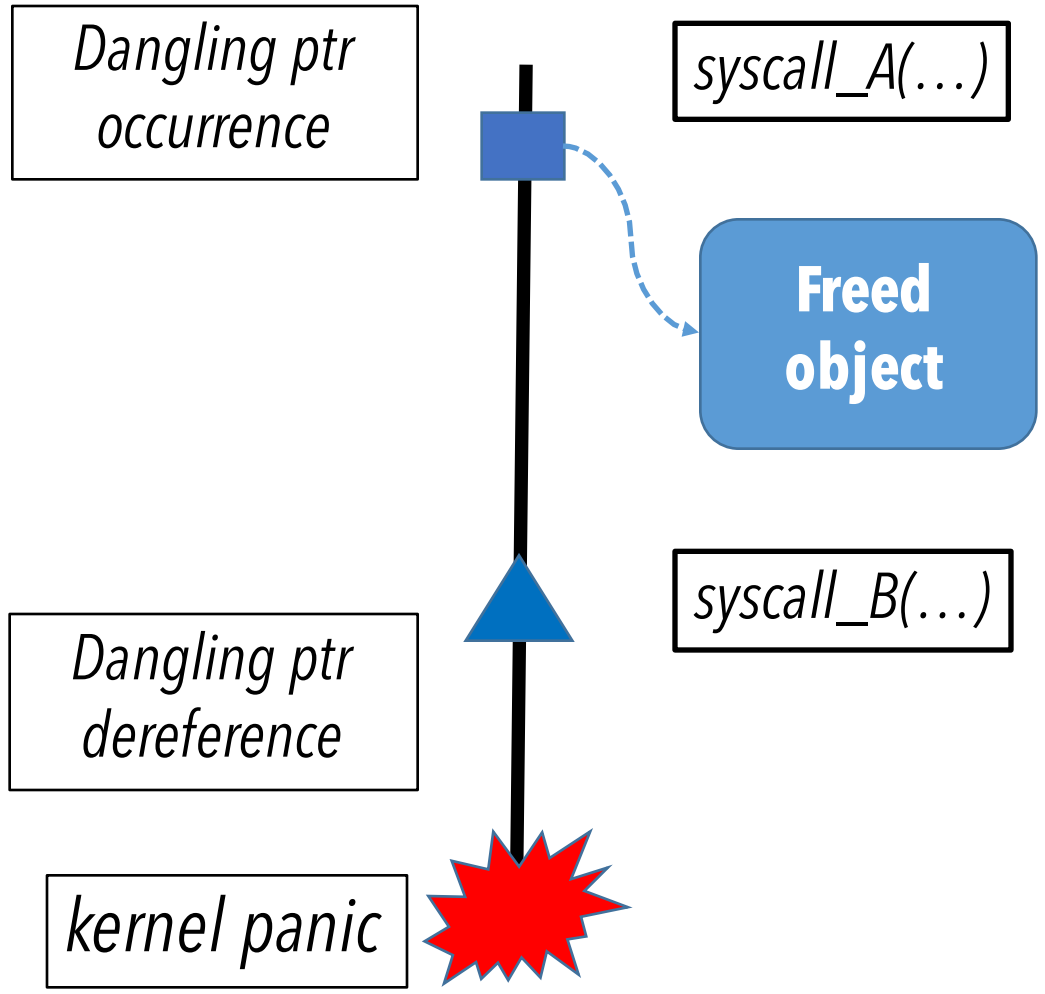
Crafting an Exploit for Kernel Use-After-Free



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. *…*

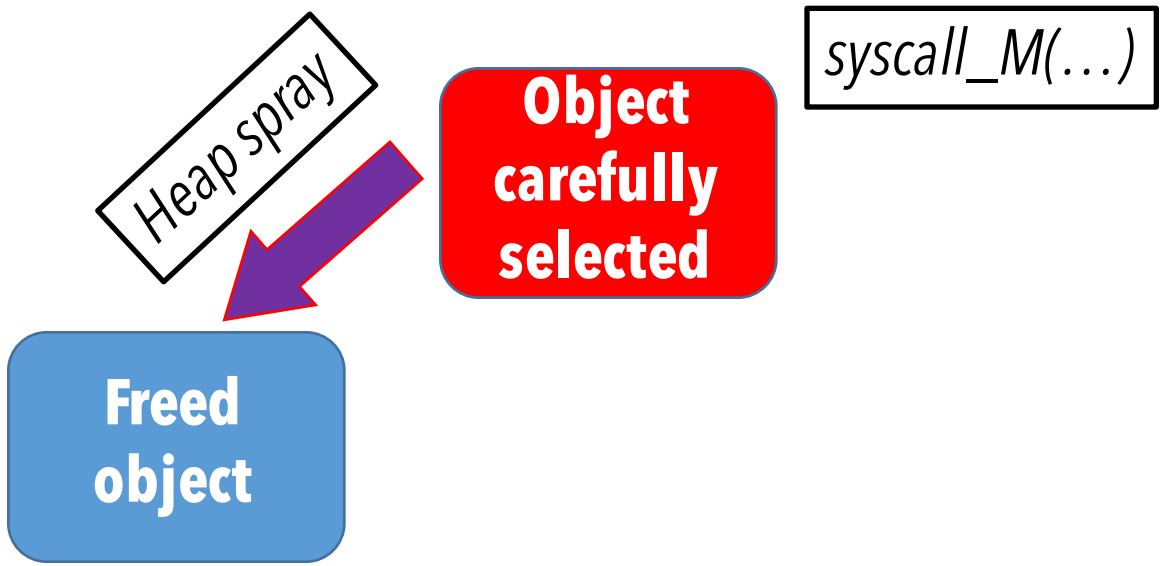
Challenge 1: Needs Intensive Manual Efforts

- 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



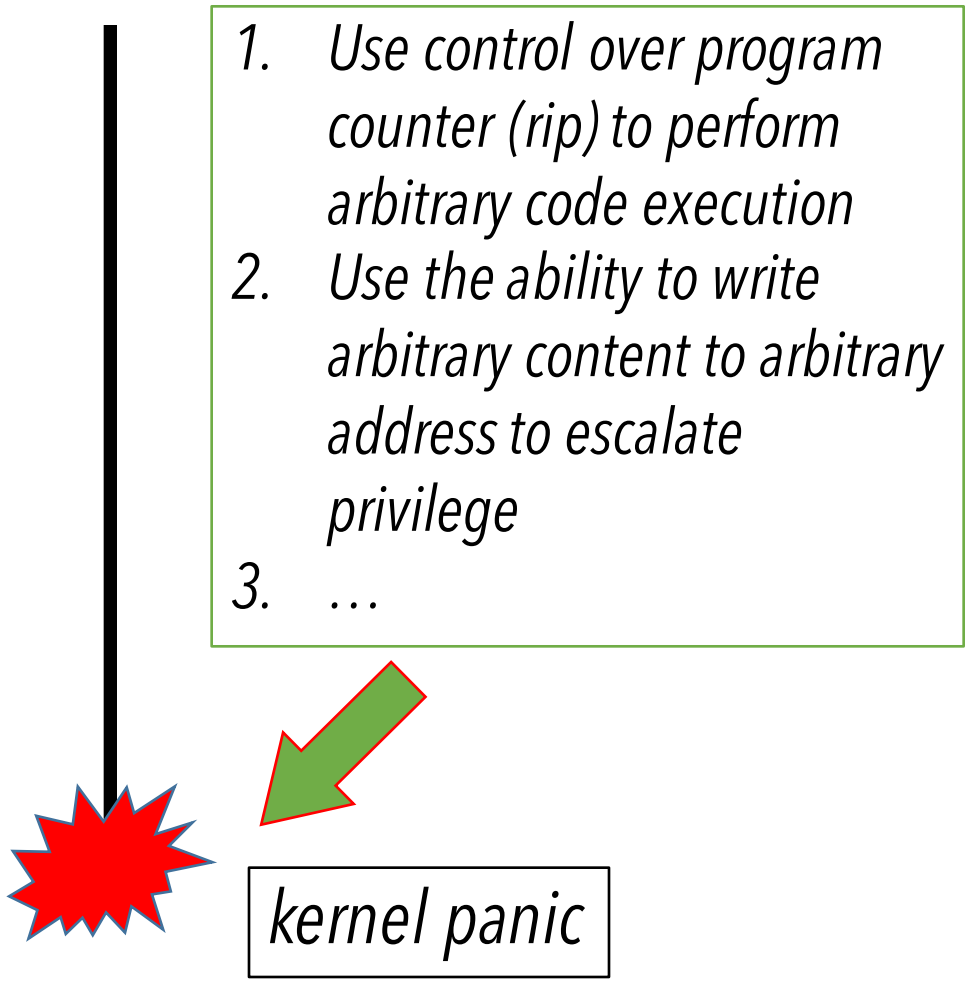
Challenge 2: Needs Extensive Expertise in Kernel

- 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
  - E.g., fixing critical data
  - …



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] [Blackhat15] [USENIX-SEC18]
- Approaches for Challenge 3
  - [NDSS'11] [S&P16], [S&P17]

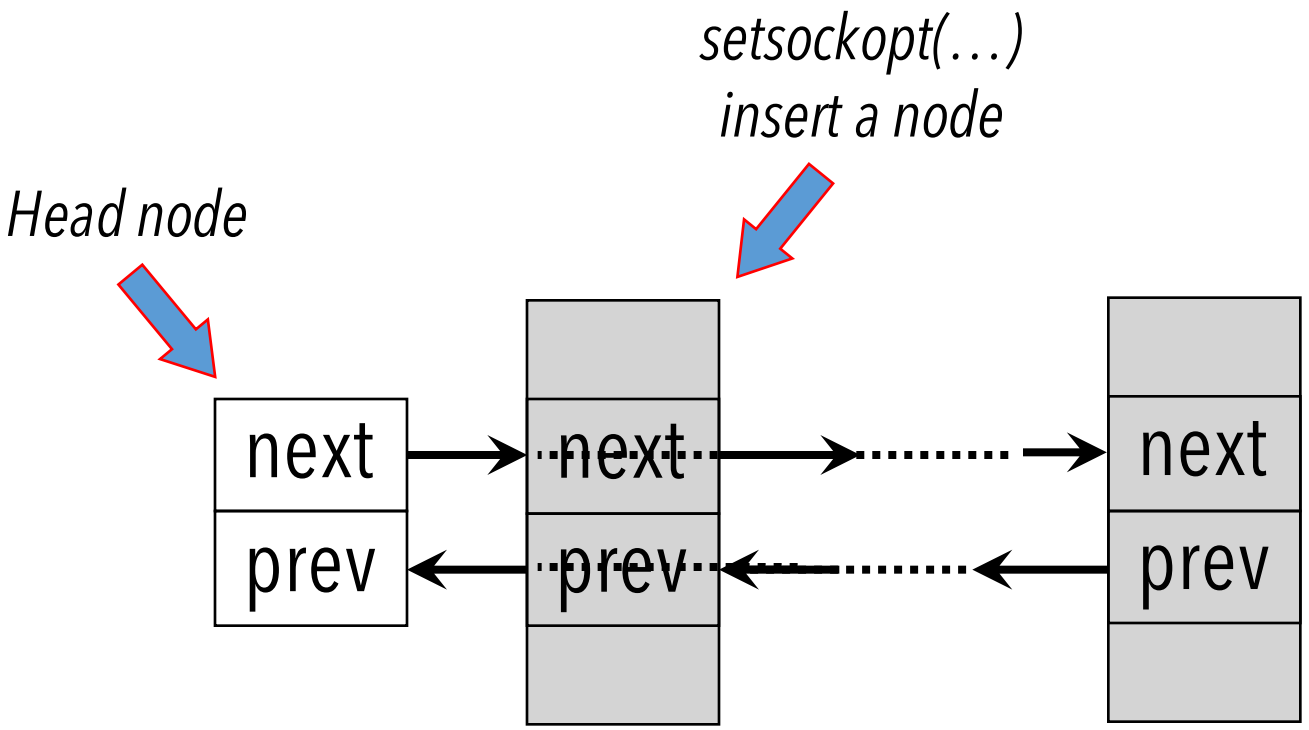
[NDSS11] Avgerinos et al., AEG: Automatic Exploit Generation.
[Blackhat 15] Xu et al., Ah! Universal android rooting is back.
[S&P16] Shoshitaishvili et al., Sok:(state of) the art of war: Offensive techniques in binary analysis.
[USENIX-SEC18] Heelan et al., Automatic Heap Layout Manipulation for Exploitation.
[S&P17] Bao et al., Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits.
[Blackhat07] Sotirov, Heap Feng Shui in JavaScript


UNSOLVED

Roadmap

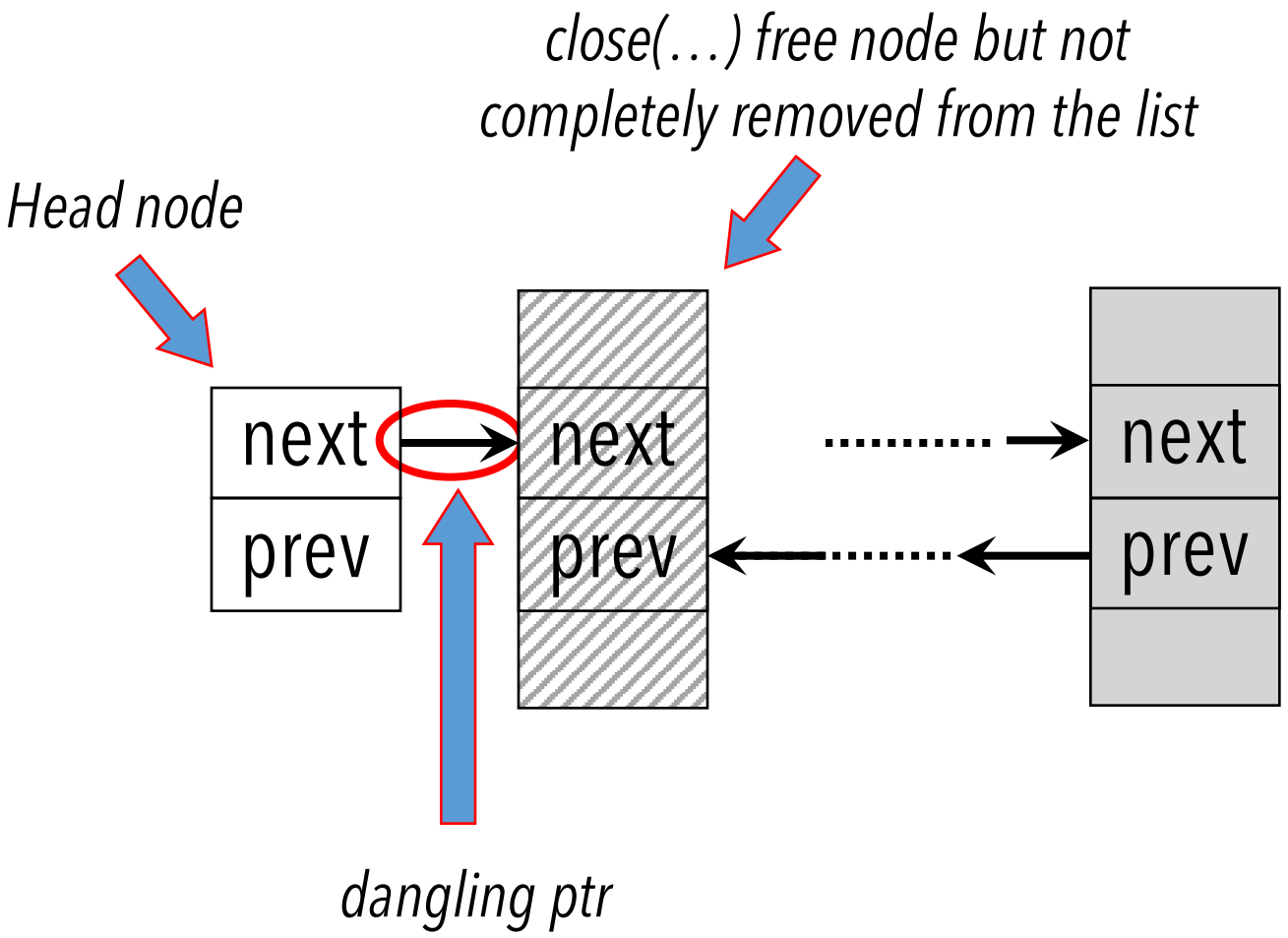
- Unsolved challenges in exploitation facilitation
- Our techniques -- FUZE
- Evaluation with real-world Linux kernel vulnerabilities
- Conclusion

A Real-World Example (CVE-2017-15649)



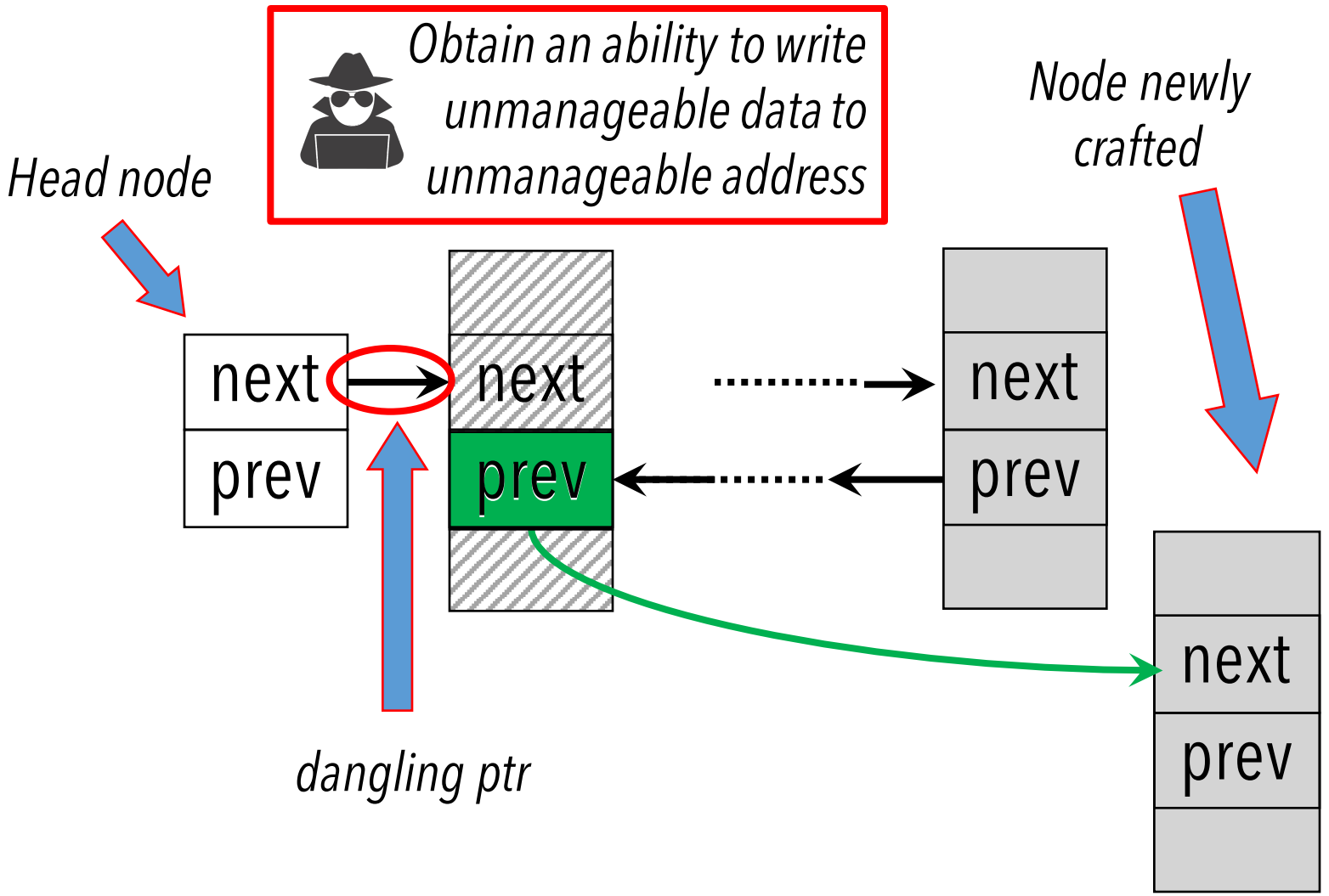

```
1  void *task1(void *unused) {
2    ...
3    int err = setsockopt(fd, 0x107, 18,
         ↪ ..., ...);
4  }
5
6  void *task2(void *unused) {
7    int err = bind(fd, &addr, ...);
8  }
9
10 void loop_race() {
11   ...
12   while(1) {
13     fd = socket(AF_PACKET, SOCK_RAW,
           ↪ htons(ETH_P_ALL));
14     ...
15     //create two racing threads
16     pthread_create (&thread1, NULL,
           ↪ task1, NULL);
17     pthread_create (&thread2, NULL,
           ↪ task2, NULL);
18
19     pthread_join(thread1, NULL);
20     pthread_join(thread2, NULL);
21
22     close(fd);
23   }
24 }
```

A Real-World Example (CVE 2017-15649)



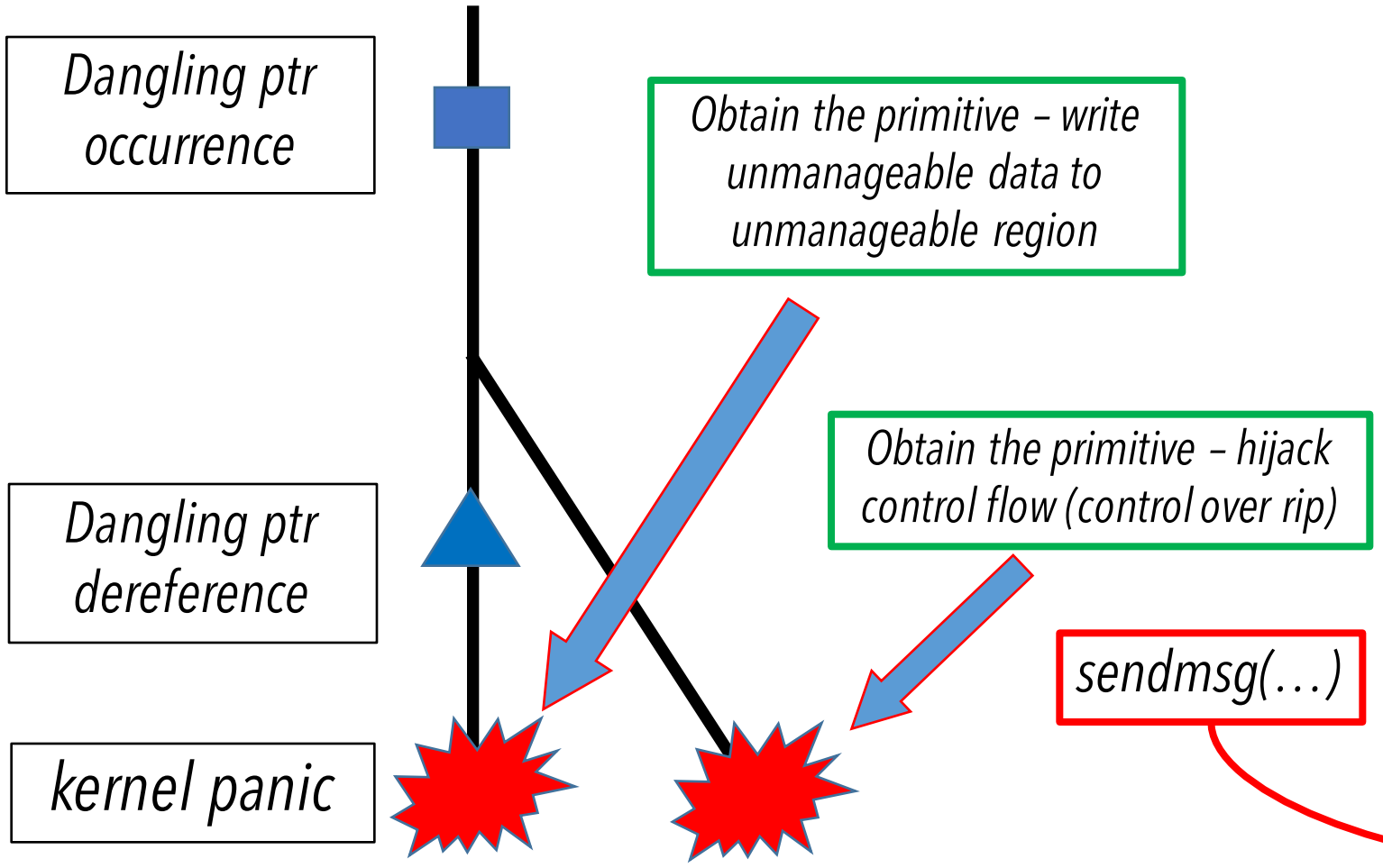

```
1  void *task1(void *unused) {
2    ...
3    int err = setsockopt(fd, 0x107, 18,
         ↪ ..., ...);
4  }
5
6  void *task2(void *unused) {
7    int err = bind(fd, &addr, ...);
8  }
9
10 void loop_race() {
11   ...
12   while(1) {
13     fd = socket(AF_PACKET, SOCK_RAW,
           ↪ htons(ETH_P_ALL));
14     ...
15     //create two racing threads
16     pthread_create (&thread1, NULL,
           ↪ task1, NULL);
17     pthread_create (&thread2, NULL,
           ↪ task2, NULL);
18
19     pthread_join(thread1, NULL);
20     pthread_join(thread2, NULL);
21
22     close(fd);
23   }
24 }
```

Challenge 4: No Primitive Needed for Exploitation




```
1  void *task1(void *unused) {
2    ...
3    int err = setsockopt(fd, 0x107, 18,
         ↪ ..., ...);
4  }
5
6  void *task2(void *unused) {
7    int err = bind(fd, &addr, ...);
8  }
9
10 void loop_race() {
11   ...
12   while(1) {
13     fd = socket(AF_PACKET, SOCK_RAW,
           ↪ htons(ETH_P_ALL));
14     ...
15     //create two racing threads
16     pthread_create (&thread1, NULL,
           ↪ task1, NULL);
17     pthread_create (&thread2, NULL,
           ↪ task2, NULL);
18
19     pthread_join(thread1, NULL);
20     pthread_join(thread2, NULL);
21
22     close(fd);
23   }
24 }
```

No Useful Primitive == Unexploitable??



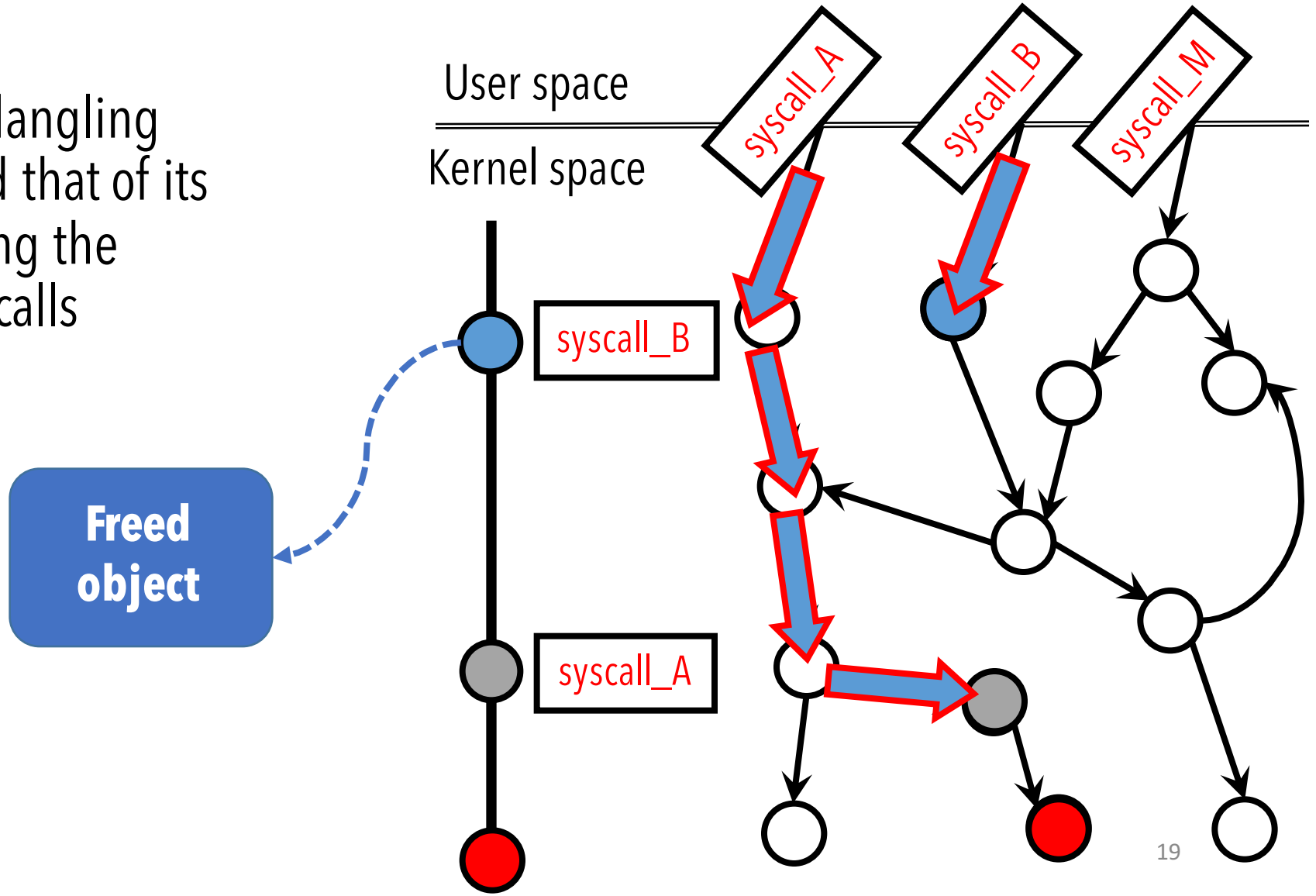

```
1  void *task1(void *unused) {
2    ...
3    int err = setsockopt(fd, 0x107, 18,
       ↪ ..., ...);
4  }
5
6  void *task2(void *unused) {
7    int err = bind(fd, &addr, ...);
8  }
9
10 void loop_race() {
11   ...
12   while(1) {
13     fd = socket(AF_PACKET, SOCK_RAW,
         ↪ htons(ETH_P_ALL));
14     ...
15     //create two racing threads
16     pthread_create (&thread1, NULL,
         ↪ task1, NULL);
17     pthread_create (&thread2, NULL,
         ↪ task2, NULL);
18
19     pthread_join(thread1, NULL);
20     pthread_join(thread2, NULL);
21
22     close(fd);
23   }
24 }
```

Roadmap

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

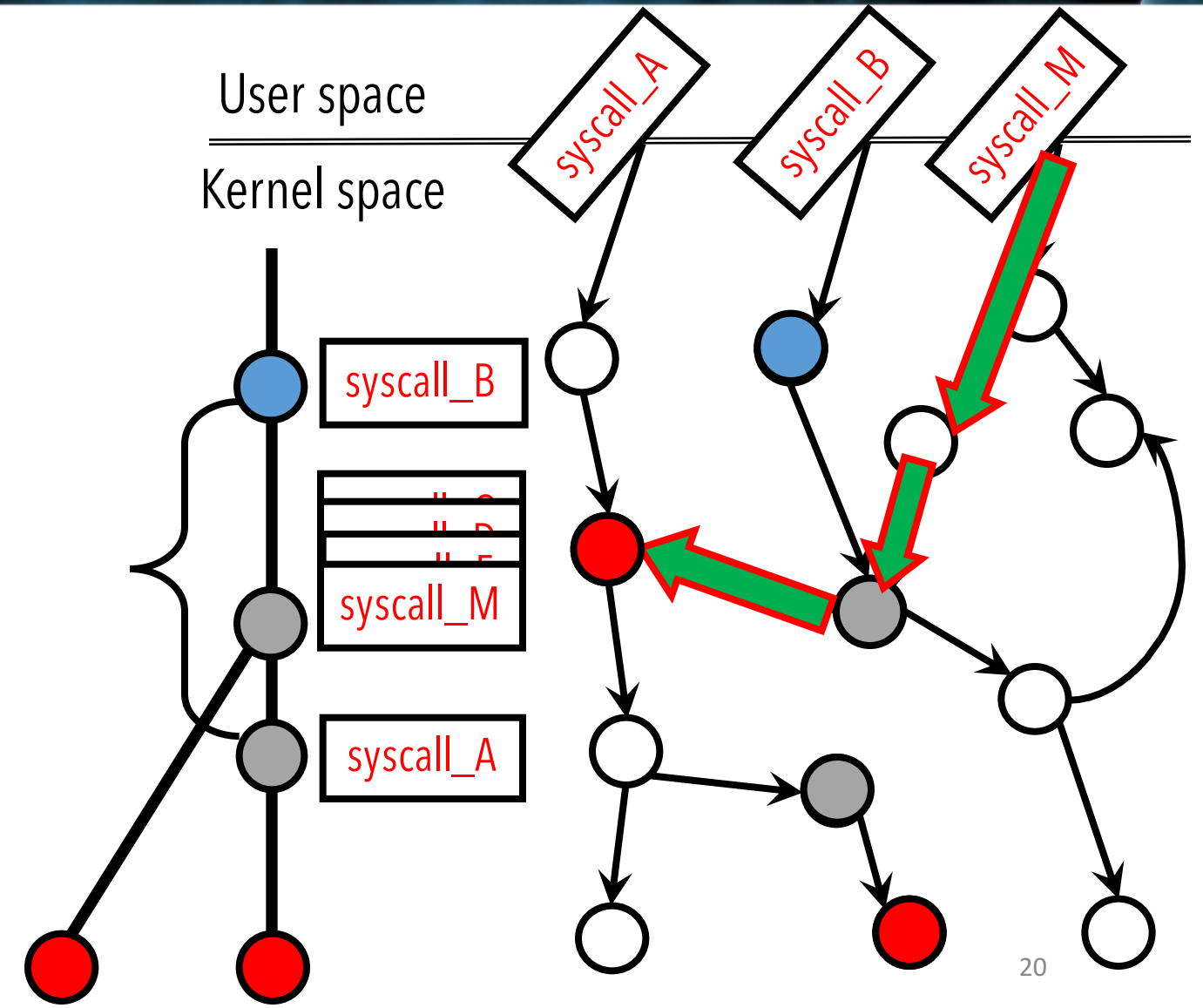
FUZE – Extracting Critical Info.

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



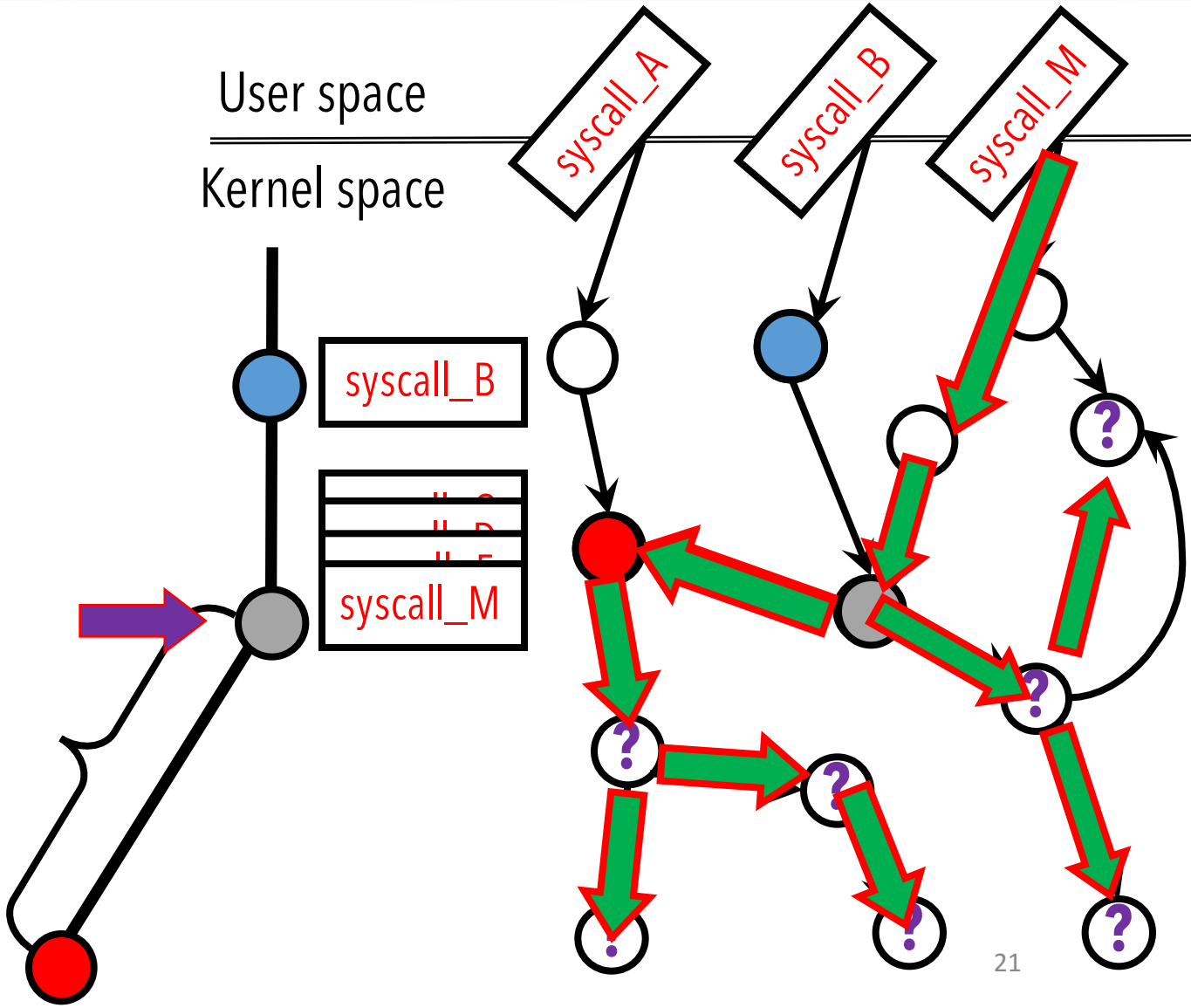
FUZE – Performing Kernel Fuzzing

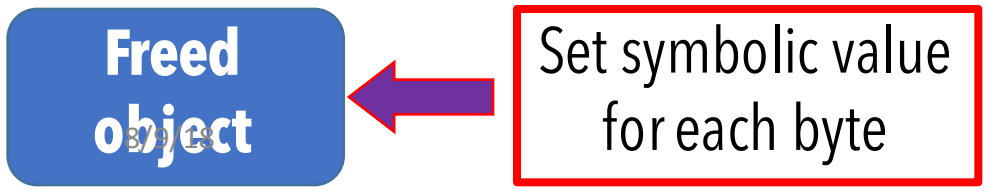
- Identifying the site of dangling pointer occurrence, and that of its dereference; pinpointing the corresponding system calls
- Performing kernel fuzzing between the two sites and exploring other panic contexts (i.e., different sites where the vulnerable object is dereferenced)



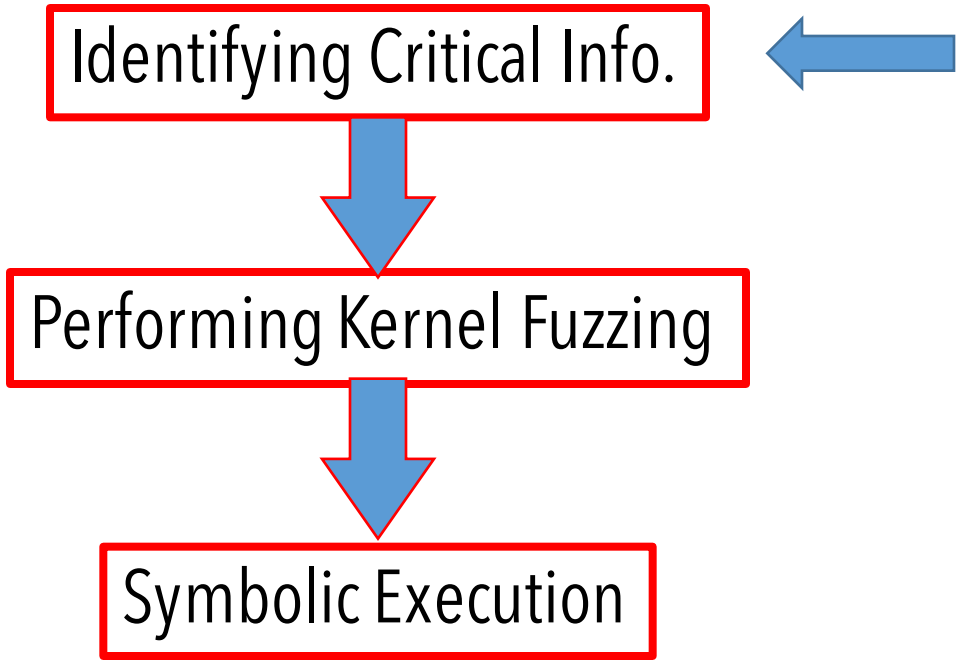
FUZE – Performing Symbolic Execution

- 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



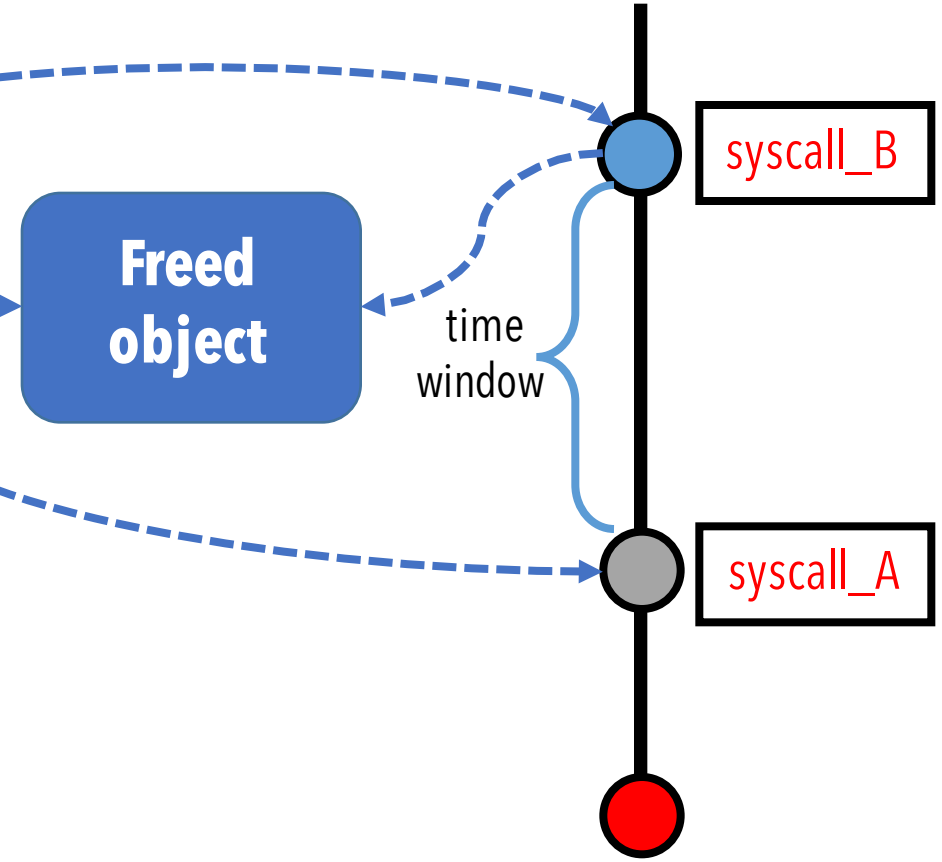


Crafting Working Exploits Step by Step



Critical Information Extraction

- Goal: identifying following critical information
  - Vulnerable object
  - Free site
  - Dereference site
  - Syscalls in PoC tied to corresponding free and dereference
  - Time window between free and dereference
- Methodology:
  - Instrument the PoC with ftrace and generate ftrace log
  - instrument kernel with KASAN
  - Combining both ftrace and KASAN log for analysis



Critical Information Extraction (cont)

- Goal: identifying following critical information
  - Vulnerable object
  - Free site
  - Dereference site
  - Syscalls in PoC tied to corresponding free and dereference
  - Time window between free and dereference
- Methodology:
  - Instrument the PoC with ftrace[1] and generate ftrace log
  - instrument kernel with KASAN[2]
  - Combining both ftrace and KASAN log for analysis

Unique ID for each syscall in PoC

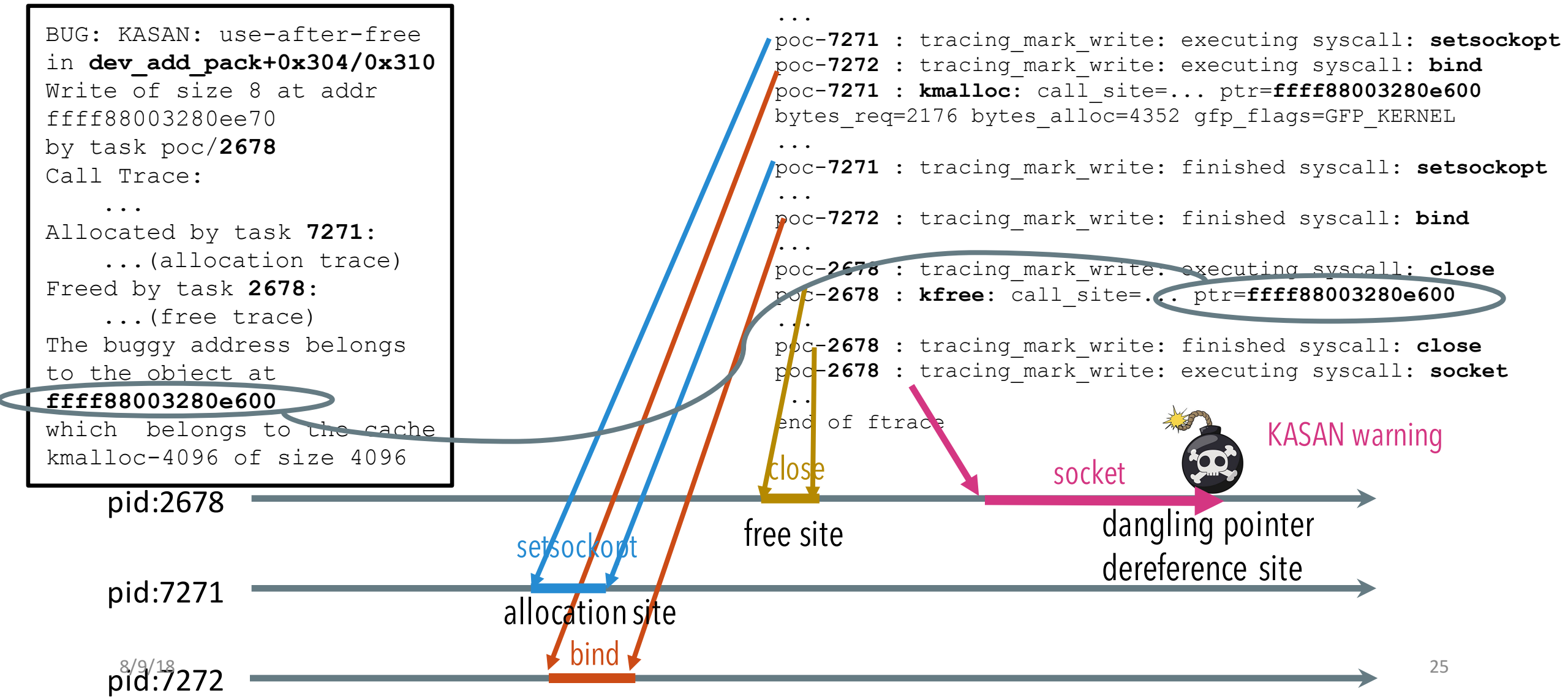
```
void *task1(void *unused) {
  ...
  write_ftrace_marker(1);
  int err = setsockopt(...);
  write_ftrace_marker(1);
}
void *task2(void *unused) {
  write_ftrace_marker(2);
  int err = bind(...);
  write_ftrace_marker(2);
}
…
void loop_race(){
  ...
}
int main(){
  ftrace_kmem_trace_enable();
  loop_race();
}
```

[1] ftrace. https://www.kernel.org/doc/Documentation/trace/ftrace.txt
[2] kasan. https://github.com/google/kasan/wiki

Critical Information Extraction (cont)


```
BUG: KASAN: use-after-free
in dev_add_pack+0x304/0x310
Write of size 8 at addr
ffff88003280ee70
by task poc/2678
Call Trace:
    ...
Allocated by task 7271:
    ...(allocation trace)
Freed by task 2678:
    ...(free trace)
The buggy address belongs
to the object at
ffff88003280e600
which  belongs to the cache
kmalloc-4096 of size 4096
```

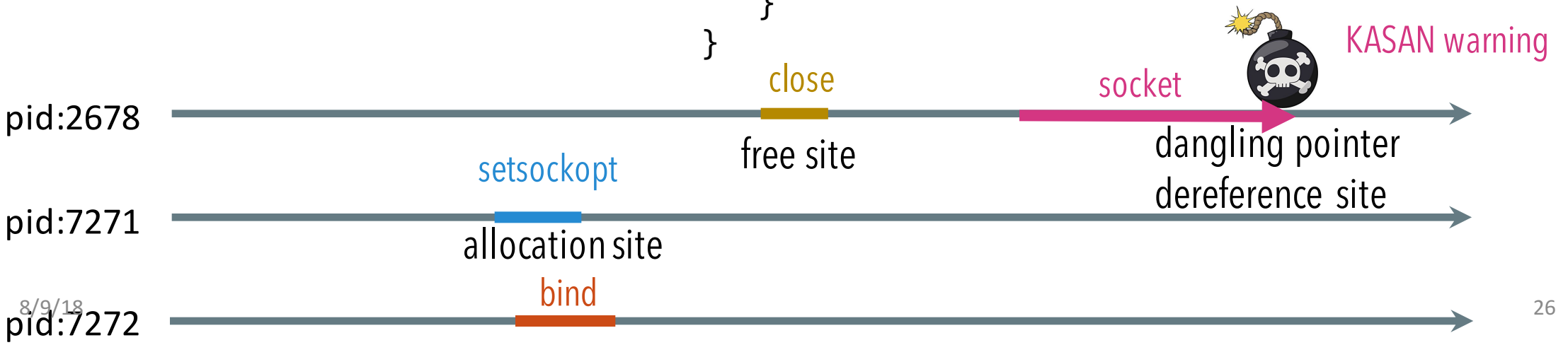
```
...
poc-7271 : tracing_mark_write: executing syscall: setsockopt
poc-7272 : tracing_mark_write: executing syscall: bind
poc-7271 : kmalloc: call_site=... ptr=ffff88003280e600
bytes_req=2176 bytes_alloc=4352 gfp_flags=GFP_KERNEL
...
poc-7271 : tracing_mark_write: finished syscall: setsockopt
...
poc-7272 : tracing_mark_write: finished syscall: bind
...
poc-2678 : tracing_mark_write: executing syscall: close
poc-2678 : kfree: call_site=... ptr=ffff88003280e600
...
poc-2678 : tracing_mark_write: finished syscall: close
poc-2678 : tracing_mark_write: executing syscall: socket
...
end of ftrace
```



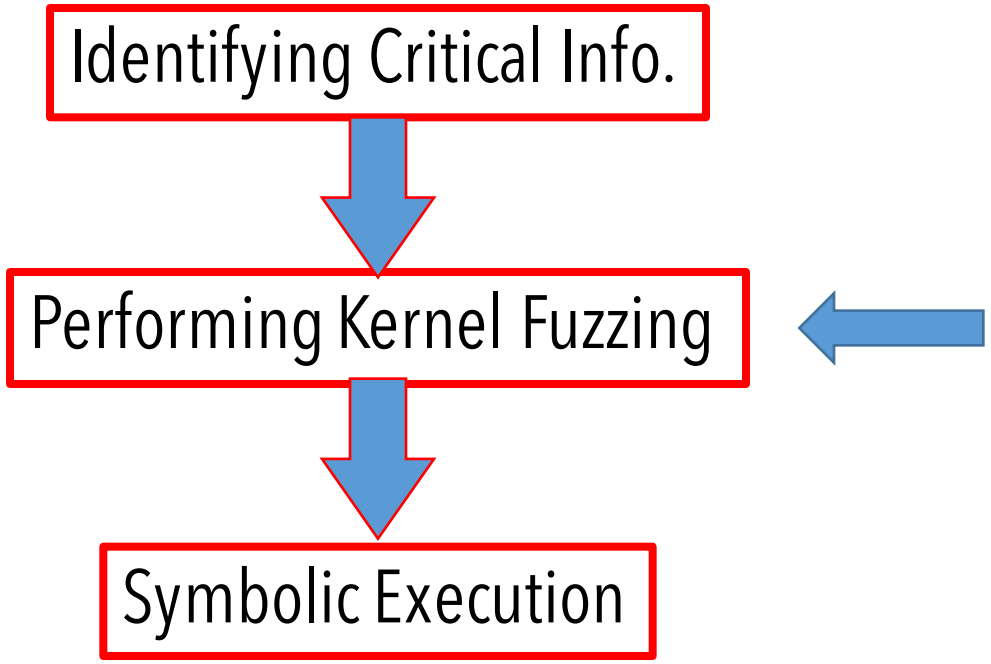
Critical Information Extraction (cont)


```
void *task1(void *unused) {
  ...
  int err = setsockopt(fd,
0x107, 18, ..., ...);
}
void *task2(void *unused) {
  int err = bind(fd, &addr,
...);
}
```

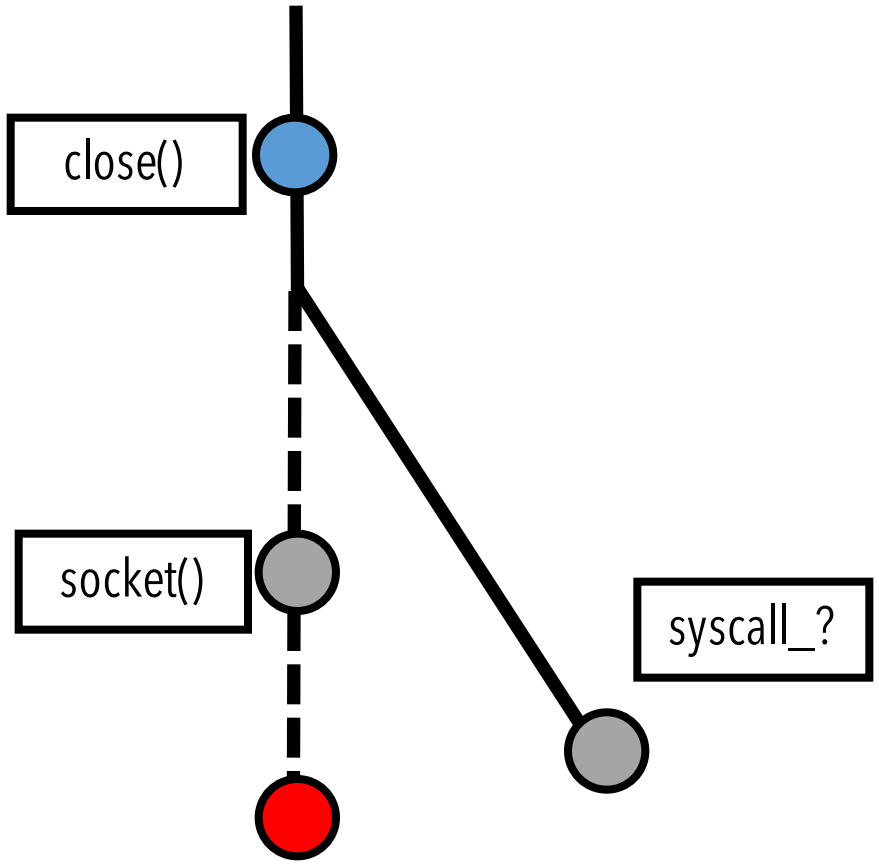
```
void loop_race() {
   ...
   while(1) {
     fd = socket(AF_PACKET, SOCK_RAW,
htons(ETH_P_ALL));
     ...
     pthread_create (&thread1, NULL, task1, NULL);
     pthread_create (&thread2, NULL, task2, NULL);
     pthread_join(thread1, NULL);
     pthread_join(thread2, NULL);
     close(fd);
   }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



Crafting Working Exploits Step by Step



Kernel Fuzzing

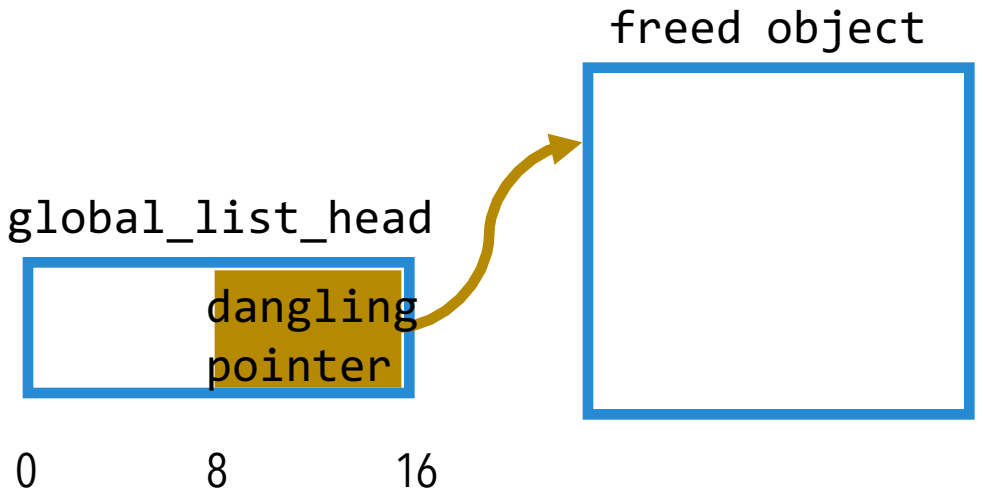



```
poc_wrapper(){
 /* PoC wrapping function */
    ...
    socket();//dereference site
    while(true){ // Race condition
        ...
        threadA(...);
        threadB(...);
        ...
        close(); //free site
        /* instrumented statements */
        if (!ioctl(...)) // interact with
a kernel module
            return;
    }
}
poc_wrapper();
fuzzing();
```

Kernel Module for Dangling Pointer Identification

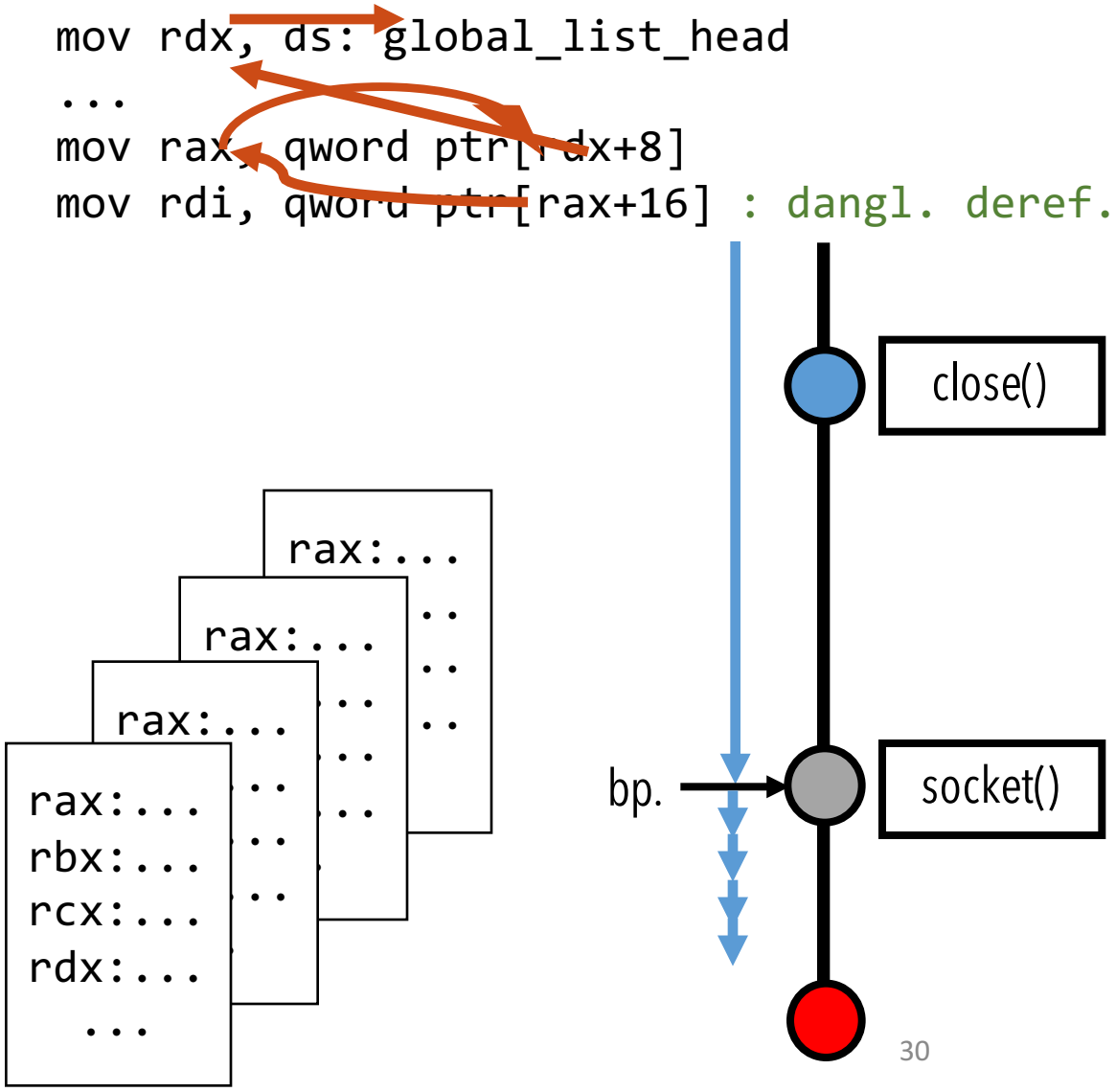
- Identifying dangling pointer through the global variable pertaining to vulnerable object
  - Setting breakpoint at syscall tied to the dangling pointer dereference
  - Executing PoC program and triggering the vulnerability
  - Debugging the kernel step by step and recording dataflow (all registers)
  - Tracking down global variable (or current task_struct) through backward dataflow analysis
  - Recording the base address the global variable (or current task_struct) and the offset corresponding to the freed object

```
mov rdx, ds: global_list_head
...
mov rax, qword ptr[rdx+8]
mov rdi, qword ptr[rax+16] : dangl. deref.
```



Kernel Module for Dangling Pointer Identification (cont)

- Identifying dangling pointer through the global variable pertaining to vulnerable object
  - Setting breakpoint at syscall tied to the dangling pointer dereference
  - Executing PoC program and triggering the vulnerability
  - Debugging the kernel step by step and recording dataflow (all registers)
  - Tracking down global variable (or current task_struct) through backward dataflow analysis
  - Recording the base address the global variable (or current task_struct) and the offset corresponding to the freed object



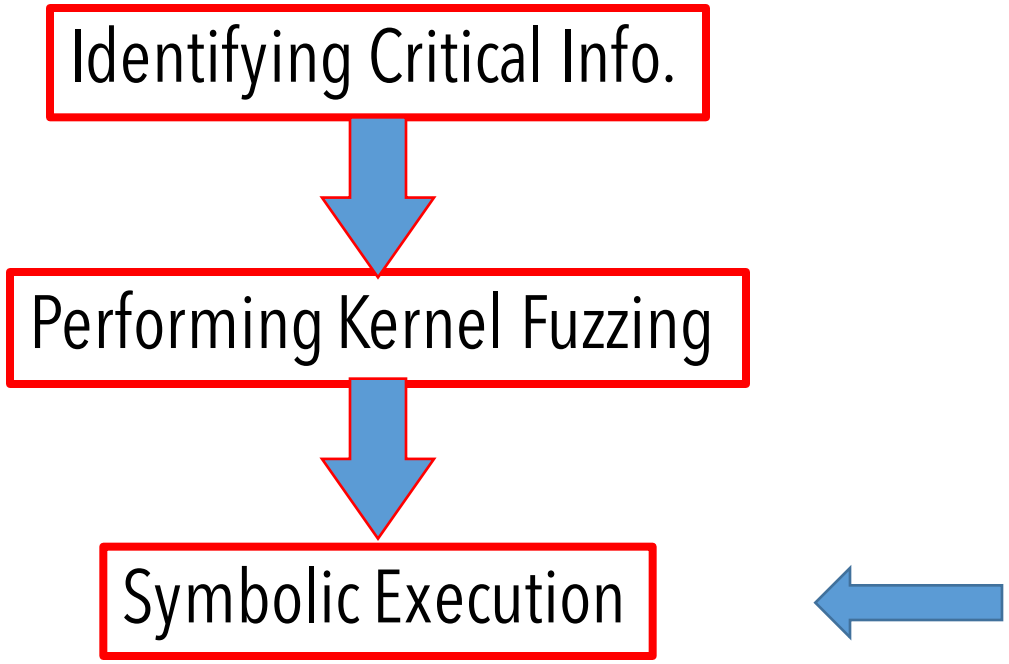
Kernel Fuzzing(cont)

- Reusing syzkaller[1] to performing kernel fuzzing after a dangling pointer is identified
  - generate syz-executor which invoke poc_wrapper first
- enable syscalls that potentially dereference the vulnerable object
  - "enable_syscalls"
- transfer variables that appears in the PoC into the interface
  - e.g. file descriptors

```
poc_wrapper();
fuzzing();
```

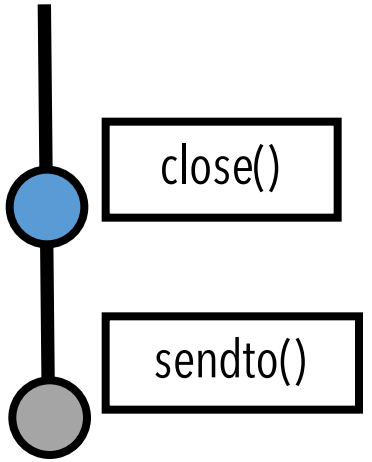

[1] syzkaller – kernel fuzzer. https://github.com/google/syzkaller

Crafting Working Exploits Step by Step



Symbolic Execution

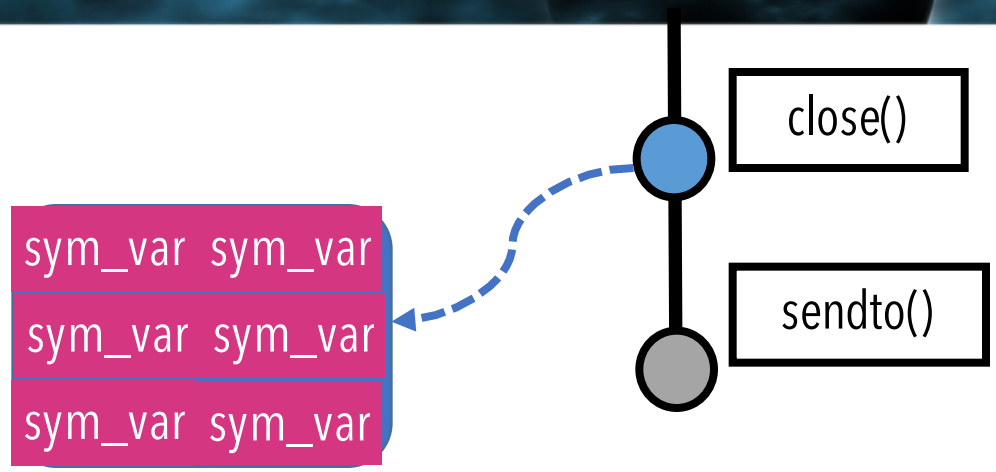
- Symbolic execution for kernel is challenging.
  - How to model and emulate interrupts?
  - How to handling multi-threading?
  - How to emulate hardware device?
- Our goal: use symbolic execution for identifying exploitable primitives
- We can opt-in angr[1] for kernel symbolic execution from a concrete state
  - single thread
  - no interrupt
  - no context switching



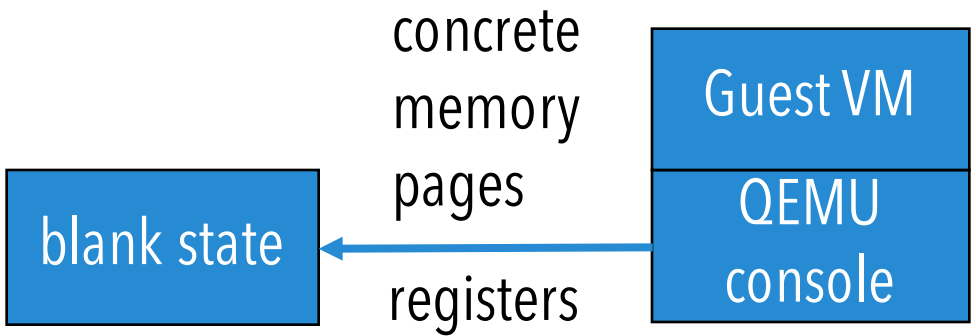
[1] angr: http://angr.io/

Symbolic Execution

- Symbolic Execution initialization
  - Setting conditional breakpoint at the dangling pointer dereference site
  - Running the PoC program to reach the dangling pointer dereference site
  - Migrating the memory/register state to a blank state
  - Setting freed object memory region as symbolic
  - Starting symbolic execution!
- Challenges:
  - State(path) explosion
  - Determining exploitable primitive
  - Handle symbolic read/write




```
for i in range(uaf_object_size):
    sym_var = state.se.BVS("uaf_byte"+str(i), 8)
    state.memory.store(uaf_object_base+i,sym_var)
```

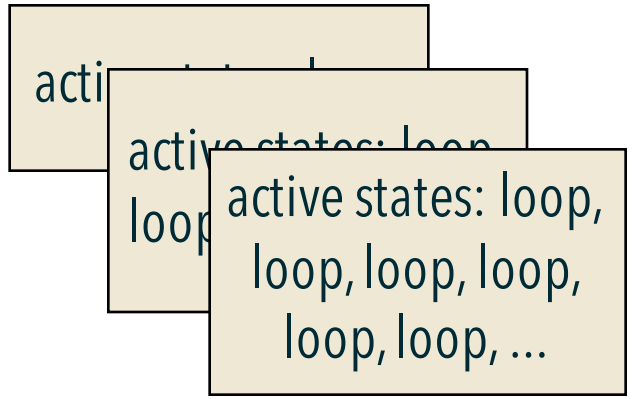


State(Path) Explosion

Memory consumption ≈ number_of_states * size_of_each_state

- Our design already mitigates state explosion by starting from the first dereference site
  - no syscall issues
  - no user input issues
- However, if a byte from the freed object is used in a branch condition, path explosion occurs.
- Workarounds:
  - limiting the time of entering a loop.
  - limiting the total length of a path.
  - copy concrete memory page on demand
  - kernel function summary

```
mov edx, dword ptr[freed obj]
loop:
...
inc ecx
cmp ecx, edx
jne loop (0xffffffff81abcdef)
...
```




```
for state in simgr.active:
    if detect_loop(state, 5):
        simgr.remove(state)

for state in simgr.active:
    if len(state.history) > 200:
        simgr.remove(state)
```

Useful primitive identification

- Unconstrained state
  - state with symbolic Instruction pointer
  - symbolic callback
- double free
  - e.g. `mov rdi, uaf_obj; call kfree`
- memory leak
  - invocation of copy_to_user with src point to a freed object
  - syscall return value

Code fragment related to an exploit primitive of CVE-2017-15649

```
if (ptype->id_match)
    return ptype->id_match(ptype, skb->sk)
```

Code fragment related to an exploit primitive of CVE-2017-17053

```
...
kfree(ldt); // ldt is already freed
```

Code fragment related to an exploit primitive of CVE-2017-8824

```
case 127...191:
    return ccid_hc_rx_getsockopt(dp-
>dccps_hc_rx_ccid, sk, optname, len, (u32
__user *)optval, optlen)
```

Useful primitive identification(cont)

- write-what-where
  - mov qword ptr [rdi], rsi

rdi (destination)	rsi (source)	primitive
symbolic	symbolic	arbitrary write (qword shoot)
symbolic	concrete	write fixed value to arbitrary address
free chunk	any	write to freed object
x(concrete)	x(concrete)	self-reference structure
metadata of freed chunk	any	meta-data corruption

From Primitive to Exploitation

- When you found a cute exploitation technique, why not make it reusable?
- Each technique can be implemented as state plugins to angr.
- Exploit technique database
  - Control flow hijack attacks:
    - pivot-to-user
    - turn-off-smap and ret-to-user
    - set_rw() page permission modification
    - …
  - Double free attacks
    - auxiliary victim object
    - loops in free pointer linked list
  - memory leak attacks
    - leak sensitive information (e.g. credentials)
  - write-what-where attacks
    - heap metadata corruption
    - function-pointer-hijack
    - vdso-hijack
    - credential modification
    - …

From Primitive to Exploitation: SMEP bypass

- Solution: ROP
  - stack pivot to userspace [1]

```
If simgr.unconstrained:
    for ucstate in simgr.unconstrained:
        try_pivot_and_rop_chain(ucstate)
```

control flow hijack primitive

```
mov rax, qword ptr[evil_ptr]
call rax
```

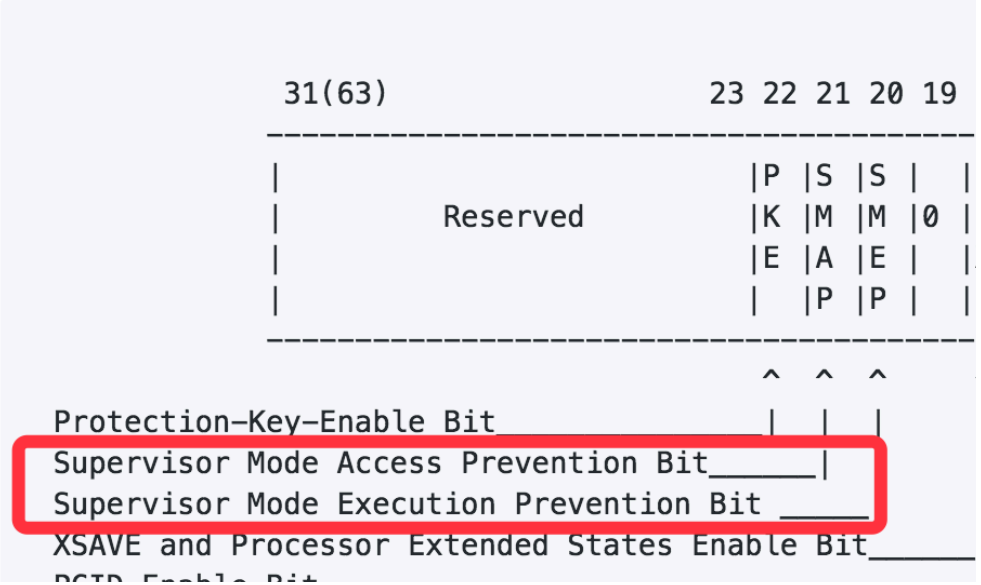
stack pivot gadget

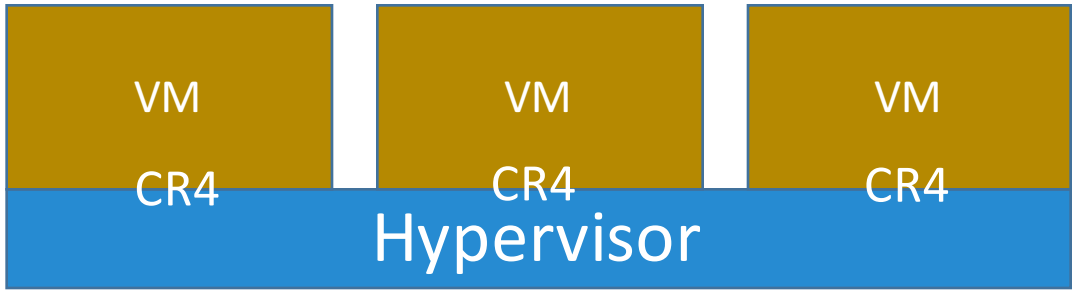
```
xchg eax, esp; ret
```

[1] Linux Kernel ROP – Ropping your way to # (Part 2)
https://www.trustwave.com/Resources/SpiderLabs-Blog/Linux-Kernel-ROP---Ropping-your-way-to---(Part-2)

From Primitive to Exploitation: SMAP bypass

- Solution: using two control flow hijack primitives to clear SMAP bit (21th) in CR4 and land in shellcode
  - 1st --- > mov cr4, rdi ; ret
  - 2nd --- > shellcode
- limitation
  - can not bypass hypervisor that protects control registers
- Universal Solution: kernel space ROP
  - bypass all mainstream mitigations.





Extra Symbolization

- Goal: enhance the ability to find useful primitives
- Observation: we can use a ROP/JOP gadget to control an extra register and explore more state space
- Approach:
  - forking states with additional symbolic register upon symbolic states
  - We may explore more states by adding extra symbolic registers

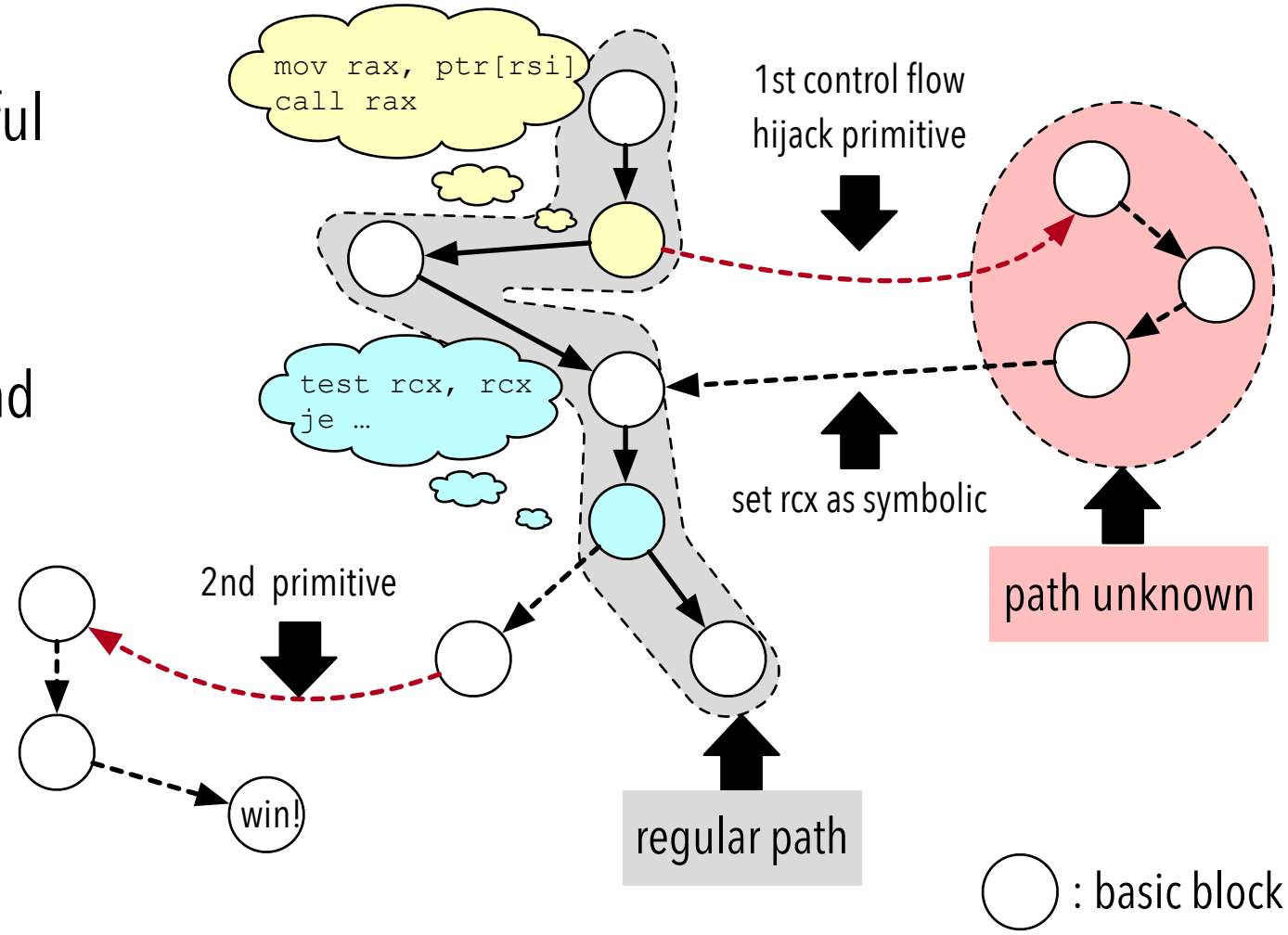


Figure: Identifying two control flow hijack primitive for CVE-2017-15649

From Primitive to Exploitation: post-exploit fix

- Sometimes we get control flow hijack primitive in interrupt context.
  - avoiding double fault: keep writing to your ROP payload page to keep it mapped in
- Some syscall (e.g. execve) checks current execution context (e.g. via reading preempt_count) and decides to panic upon unmatched context.

```
BUG_ON(in_interrupt());
```

```
-----------[ cut here ]-----------
kernel BUG at linux/mm/vmalloc.c:1394!
```

- Solution: fixing preempt_count before invoking `execve("/bin/sh", NULL, NULL)`

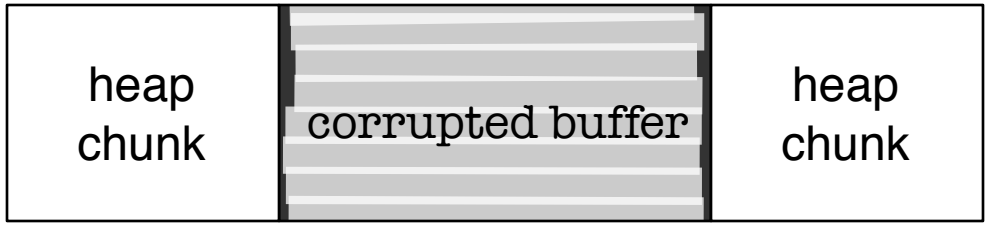
Symbolic Read/Write


```
t0
  mov rdi, QWORD PTR [corrupted_buffer]
t1
  mov rax, QWORD PTR [rdi]
t2
```

rdi: xxx

t0

rdi → symbolic_qword



t1

rax → symbolic_qword

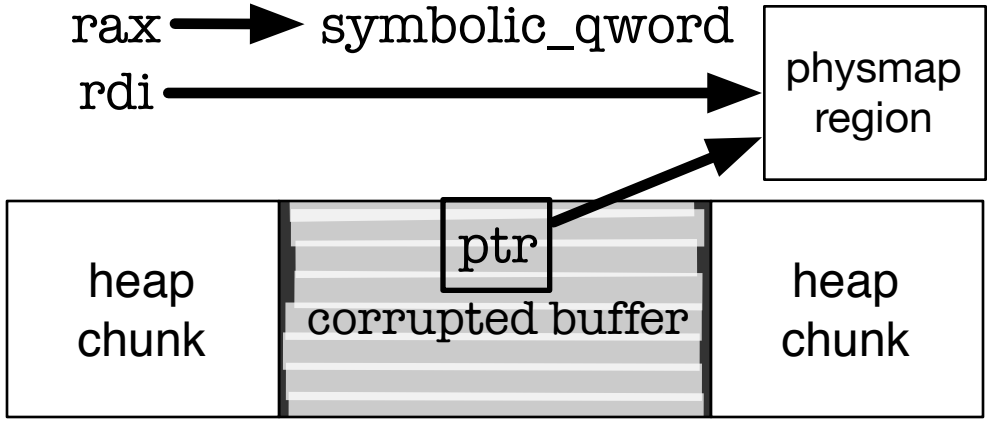
rdi → ???

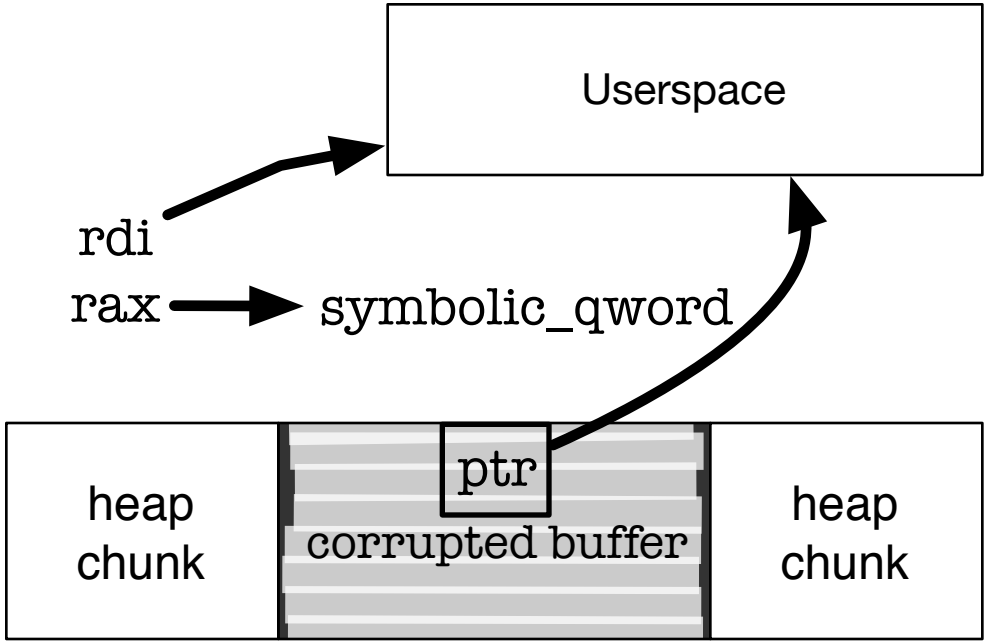
t2

Symbolic read/write concretization strategy

- Concretize the symbolic address to pointing a region under our control
  - no SMAP: entire userspace
  - with SMAP but no KASLR: physmap region
  - with SMAP and KASLR: … need a leak first

```
mov rdi, QWORD PTR [corrupted_buffer]
mov rax, QWORD PTR [rdi]
```





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

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

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 performin kernel exploitation
    - 5 vs 19 (SMEP)
    - 2 vs 5 (SMAP)

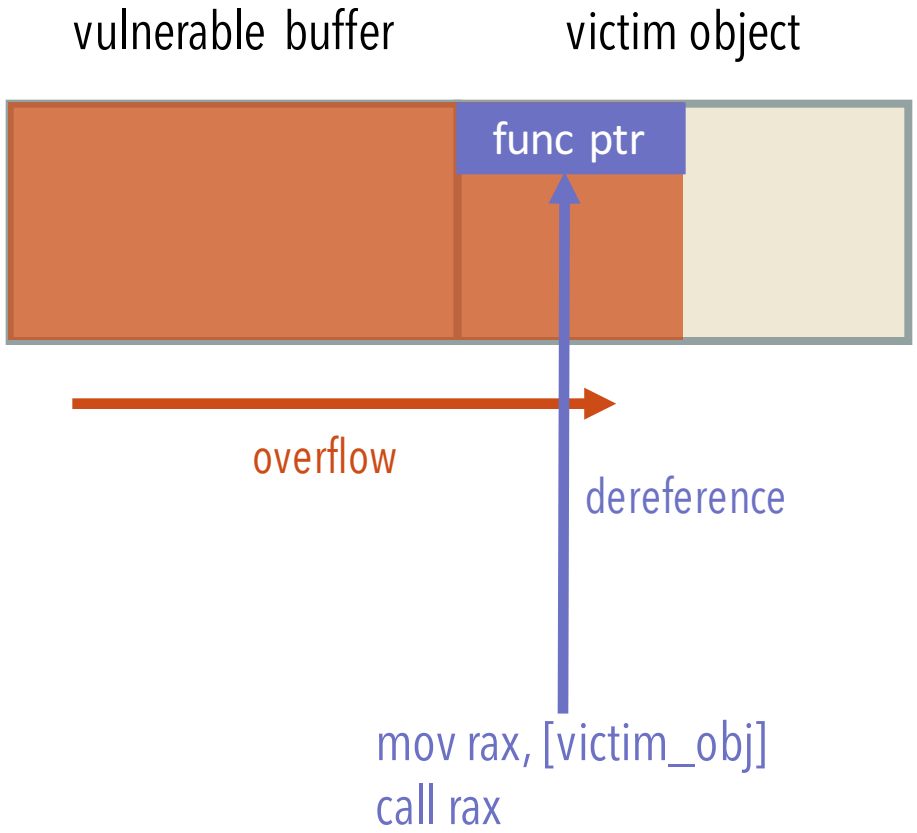
CVE-ID	# of public exploits		# of generated exploits	
	SMEP	SMAP	SMEP	SMAP
2017-17053	0	0	1	0
2017-15649	0	0	3	2
2017-15265	0	0	0	0
2017-10661	0	0	2	0
2017-8890	1	0	1	0
2017-8824	0	0	2	2
2017-7374	0	0	0	0
2016-10150	0	0	1	0
2016-8655	1	1	1	1
2016-7117	0	0	0	0
2016-4557	1	1	4	0
2016-0728	1	0	3	0
2015-3636	0	0	0	0
2014-2851	1	0	1	0
2013-7446	0	0	0	0
overall	5	2	19	5

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

What about heap overflow

- Heap overflow is similar to use-after-free:
  - a victim object can be controlled by attacker by:
    - heap spray (use-after-free)
    - overflow (or memory overlap incurred by corrupted heap metadata)
- Heap overflow exploitation in three steps:
  1) Understanding the heap overflow
     off-by-one? arbitrary length? content controllable?
  2) Find a suitable victim object and place it after the vulnerable buffer
     automated heap layout[1]
  3) Dereference the victim object for exploit primitives



[1] Heelan et al. Automatic Heap Layout Manipulation for Exploitation. USENIX Security 2018.

JD.COM Introduction

236.5 million

Largest retailer in China,
online or offline shoppers

$37.5bn

Third largest internet company
in the world by revenue in 2016

First e-commerce company to use commercial drone delivery

Massive Scale

700 Million

June Sales Event **Items** Sold

Massive Scale

236.5M
active customer accounts

120K
active third-party vendors on JD platform

120K
full-time employees

1.59B
full-time orders fulfilled in 2016

Usage Scenarios

- Bug prioritization
  - Focus limited resources to fix bugs with working exploits
- APT detection
  - Use generated exploits to generate fingerprints for APT detection
- Exploit generation for Red Team
  - Supply Red Team with a supply of new exploits

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…

Q&A

- Acknowledgement:
  - Yueqi Chen
  - Jun Xu
  - Xiaorui Gong
  - Wei Zou

- Exploits available at:
  - https://github.com/ww9210/Linux_kernel_exploits
- Contact: wuwei@iie.ac.cn

Q&A

- Acknowledgement:
  - Yueqi Chen
  - Jun Xu
  - Xiaorui Gong
  - Wei Zou

- Exploits available at:
  - https://github.com/ww9210/Linux_kernel_exploits