KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities

Wei Wu[1,2,3], Yueqi Chen[2], Xinyu Xing[2], Wei Zou[1,3]

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

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

28th USENIX Security Symposium

Aug 15, 2019

PennState

Background

- OS kernels are written in low-level languages C/C++
  - Linux: C
  - Windows: C and C++
- OS kernels are prone to memory corruption bugs
  - Out of Bounds Access, Use-After-Free, data race and even type confusion (in C++ components)
- Bugs in OS kernel are plenty and many of them are exploitable
- Exploit Mitigation: make exploit harder with ignorable cost
  - The cost to prove exploitability is increasing
- Exploitability: a predicate related to each bug
- A concrete "kernel exploit" could serve as a proof of exploitability

Background (cont.)

- Automatic exploit generation systems: capable of generating concrete exploits
- Automatic exploit generation systems in two steps:
  1. Identifying exploit primitives
  2. Evaluating exploit primitives
- Exploit primitive:
  - A machine state which empowers an attacker to craft an exploit (a.k.a. programming weird machine)
    - Data flow: Writing 8 bytes anywhere, write 1 byte to adjacent heap chunk etc.
    - Control flow: Control-flow hijacking
- Control-flow hijacking primitive is one of the most popular exploit primitives.

Crafting a control-flow hijacking kernel exploit

1 Adjusting syscall parameters and memory layout

2 Getting a control-flow hijacking primitive

3 Executing exploitation payload

- Step 1. Adjusting parameters of system calls and memory layout
  - [USENIX-SEC18][CCS 16]
- Step 2. Getting a control-flow hijacking primitive
  - [P0 blog][POC16]
- Step 3. Payload execution
  - [USENIX-SEC 14]

[USENIX-SEC14] Vasileios et al., ret2dir: Rethinking Kernel Isolation
[CCS 16] Xu et al., From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel.
[USENIX-SEC18] Heelan et al., Automatic Heap Layout Manipulation for Exploitation.
[P0 blog] Andrey Konovalov. Exploiting the Linux kernel via packet sockets.
[POC2016] Dong-hoon you. New reliable android kernel root exploitation techniques.

PennState

Key Step: from control-flow hijack to ROP payload execution

Getting a control-flow hijacking primitive

Kernel State S

```
gdb> info registers
  rsp: x rip: 0x41424344
  …
gdb> x/10gx $rsp
X     : ???????? ????????
X+8   : ???????? ????????
```

How to bootstrap a ROP attack? (e.g. Transition S -> S')

Executing exploitation payload (e.g. through ROP)

Kernel State S'

```
gdb> info registers
  rsp: x' rip: 0x51525354
  …
gdb> x/10gx $rsp
X'     : 41414141 41414141
X'+8   : 41414141 41414141
```

Semantic of an example ROP payload

commit_creds(prepare_kernel_cred(0))

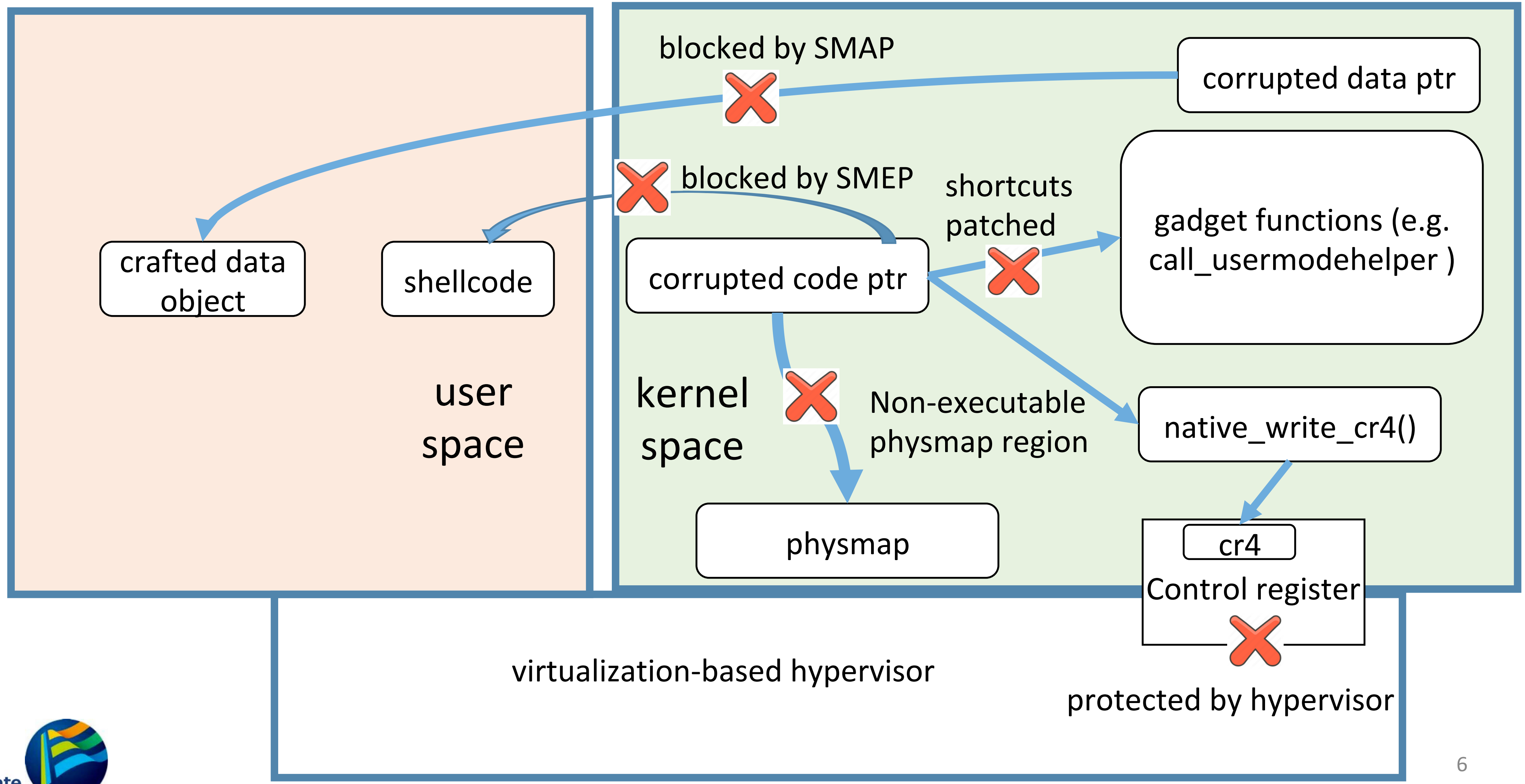
...

(fixing context and safely return to userspace)

...

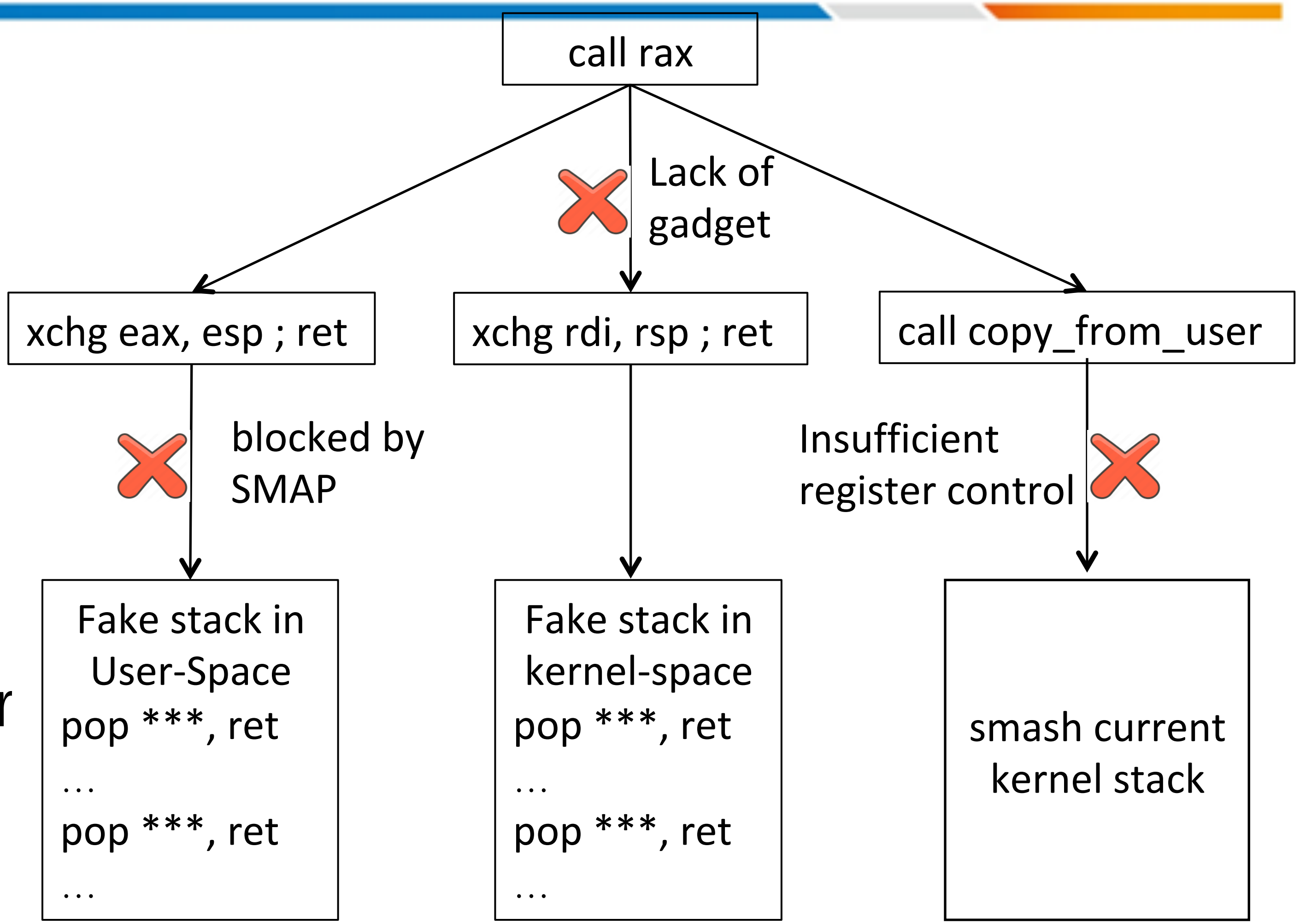
execve("/bin/sh",NULL,NULL)

Challenge 1. kernel exploit mitigations



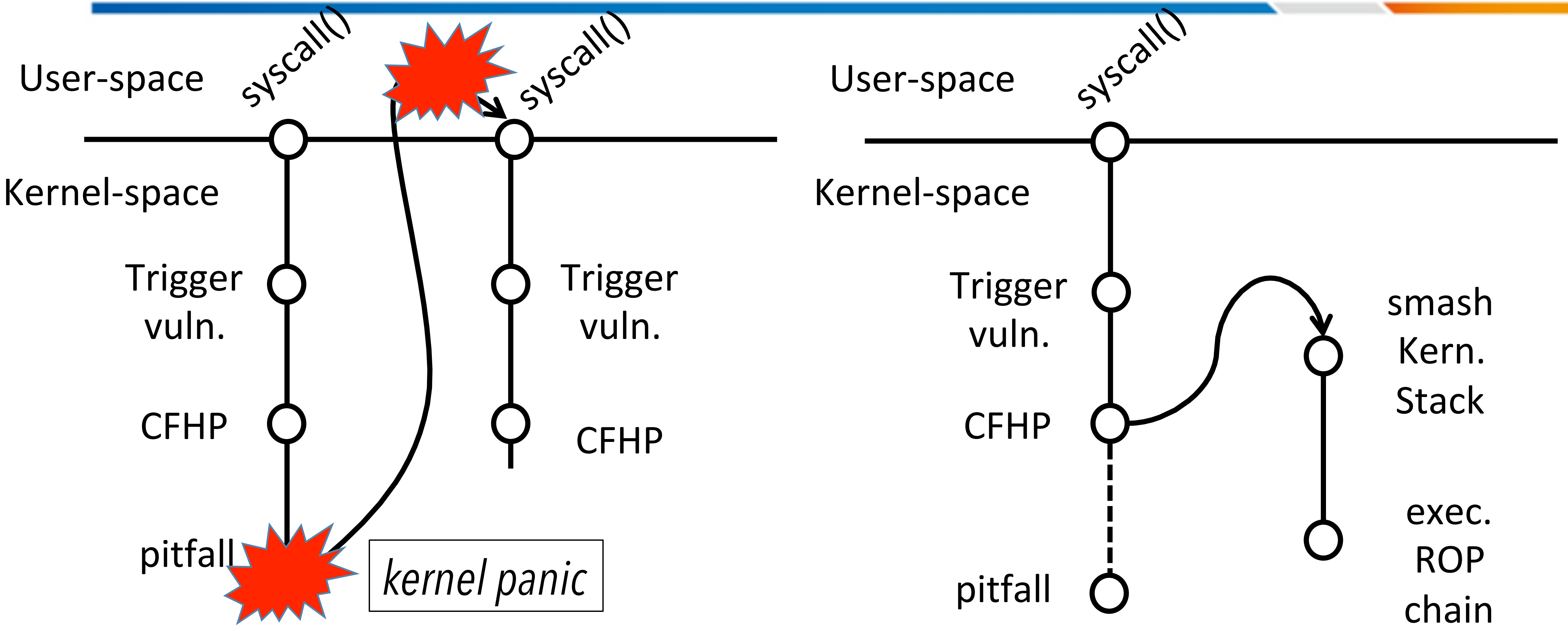
Challenge 2. ill-suited exploit primitive

- Lack of stack pivoting gadget in Linux kernel
  - traditional stack pivoting gadget blocked by SMAP because it accesses user-space memory
  - Intra-kernel stack pivoting gadget sometimes does not exist.
- Insufficient control over registers for invoking kernel functions



copy_from_user(dst, src, size)

Challenge 3. exploit path pitfall

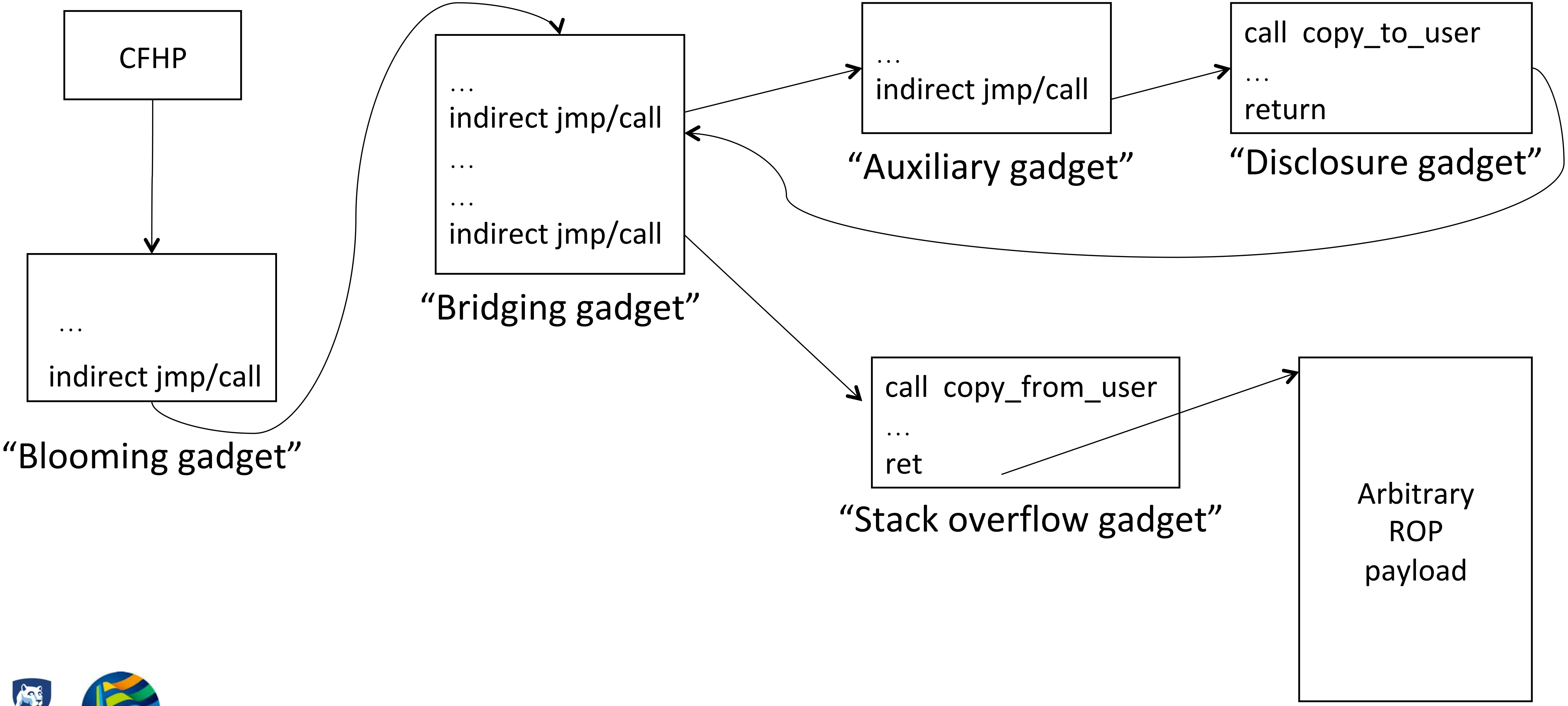


Our Solution: “single-shot” exploitation

Roadmap

- Challenges
- Our Technique
- Evaluation with real-world Linux kernel vulnerabilities
- Conclusion

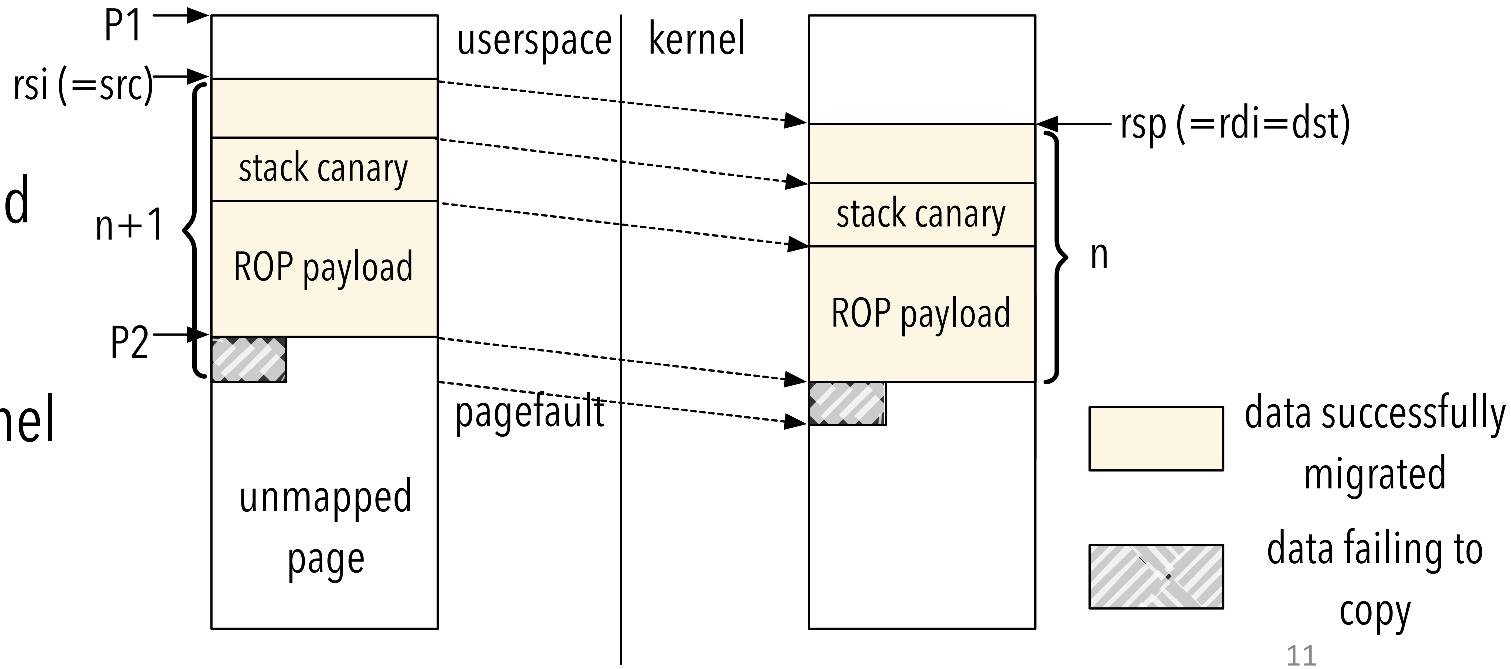
Overview of "single-shot" Exploitation



Stack smashing gadget

- copy_from_user(dst, src, size)
  - Data channel between user-space and kernel-space
  - Destination is kernel stack for 91% invocations of copy_from_user() in Linux kernel 4.15.
- Short return
  - Check for non-zero return value and returns -EFAULT
  - Short return path exists for more than 99% invocations in Linux kernel 4.15

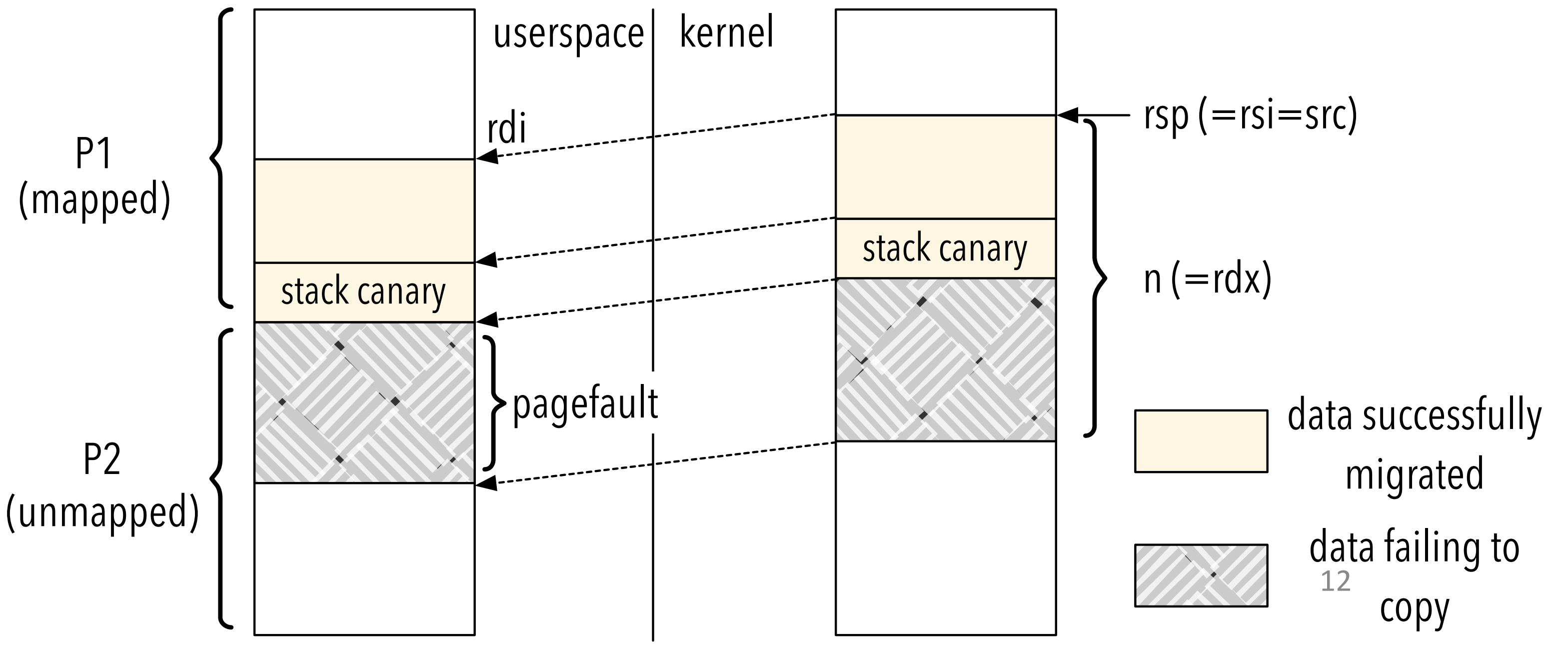
```
static long bsg_ioctl(struct file *file, unsigned int cmd,
unsigned long arg){
    struct sg_io_v4 hdr; // destination is local variable
    ...
    if (copy_from_user(&hdr, uarg, sizeof(hdr))) {
        return -EFAULT; // short return
    }
```



Bypassing stack canary: stack disclosure gadget

- copy_to_user(to, from, n)
  - Copying kernel data to user-space
  - Src is usually kernel stack (82% in 4.15)
  - Short return path exists
- Problem:
  - Caller of copy_to_user also protected by stack canary

```
SYSCALL_DEFINE2(gettimeofday, struct timeval *, tv, struct timezone *, tz){
    struct timeval ktv;
    …
    if(copy_to_user(tv, &ktv, sizeof(ktv))) {
        return -EFAULT;
    }
    …
```



Bypassing stack canary (cont.)

- Auxiliary function gadget
  - Protected by stack canary
  - controllable indirect call
- Leaking stack canary by combination of
  - Auxiliary function, and
  - Canary disclosure gadget

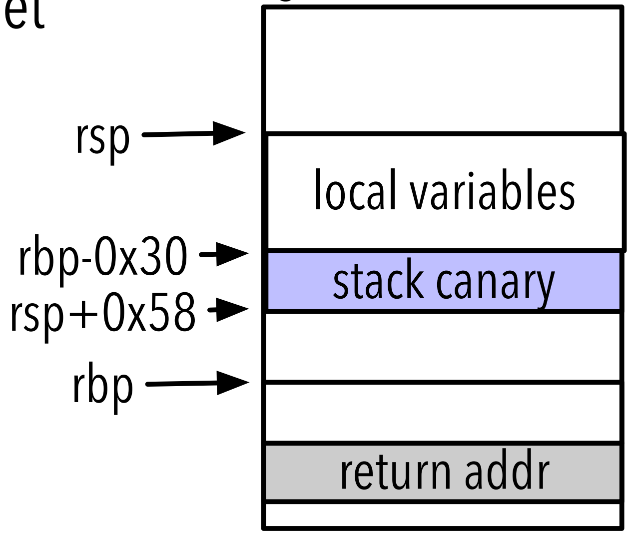
Auxiliary function gadget

```
push    rbp
mov     rbp, rsp
push    r12
...
sub     rsp, 58h
mov     rax, gs:qword_28
mov     [rbp-30h], rdi
mov     rax, [rdi]
call    rax
...
```

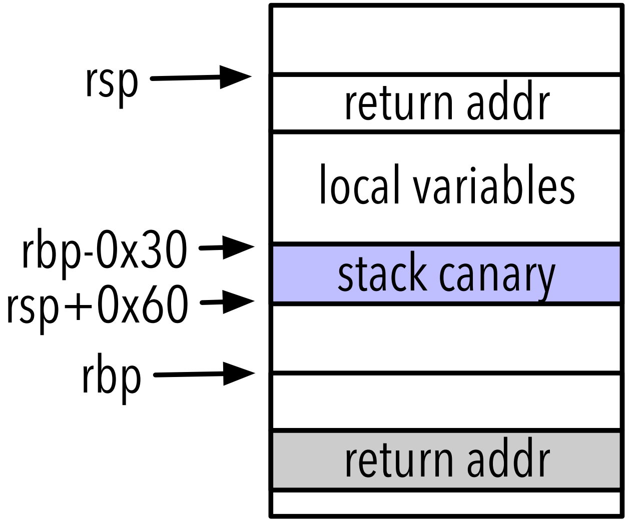
Canary disclosure gadget

```
lea     rsi, [rbp-60h]
call    _copy_to_user
test    rax, rax
jnz     fail
...
fail:
mov     rbx, FFFFFFF2h
jmp     exit
...
exit:
  mov     rcx,
[rbp-30h]
  xor     rcx,
gs:qword_28
  jnz     panic
  add     rsp, 60h
  pop     rbx
  ...
  ret
```

stack right before "**call rax**"



stack right after "**call rax**"



Enhancing register control: blooming gadget

- Linux kernel code have features of object-oriented programming
  - "self" passed as first parameter
- Blooming gadget:
  - Given register rdi is under control
  - A family of kernel functions containing an indirect call
    - target is controllable
    - three parameters of the indirect call are controllable

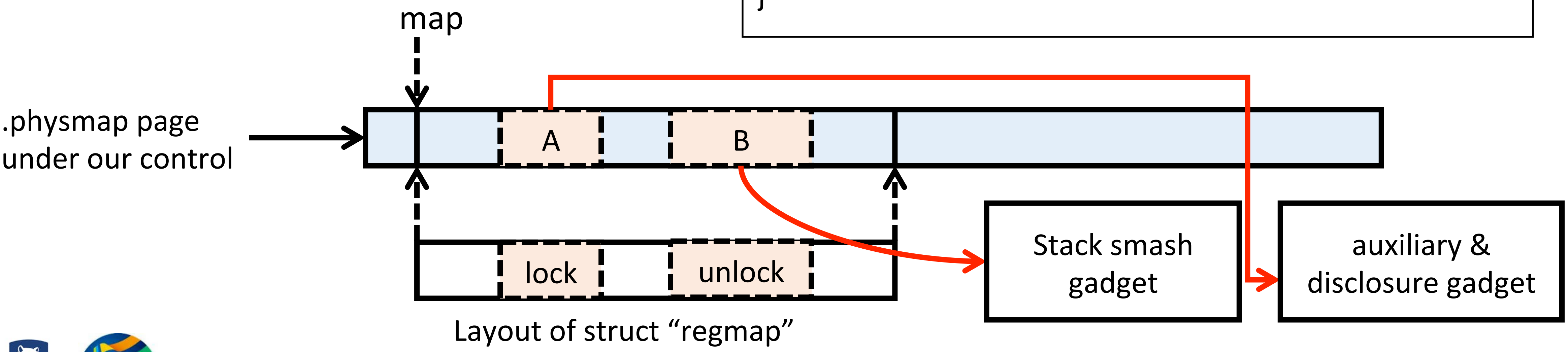
```
static void aliasing_gtt_unbind_vma(struct i915_vma *vma) {
    ...
    vma->vm->clear_range(vma->vm, vma>node.start, vma->size);
...
}
```

```
push rbp
push rbx
mov rbx,rdi
mov rax,QWORD PTR [rdi+0xa8]
mov rbp,QWORD PTR [rax+0x330]
mov rax,QWORD PTR [rdi+0xf8]
...
mov rdi,QWORD PTR [rbp+0x3f28]
mov rdx,QWORD PTR [rbx+0xd0]
mov rsi,QWORD PTR [rbx+0x8]
pop rbx
pop rbp
mov rax,QWORD PTR [rdi+0x468]
jmp rax
```

Bridging gadget

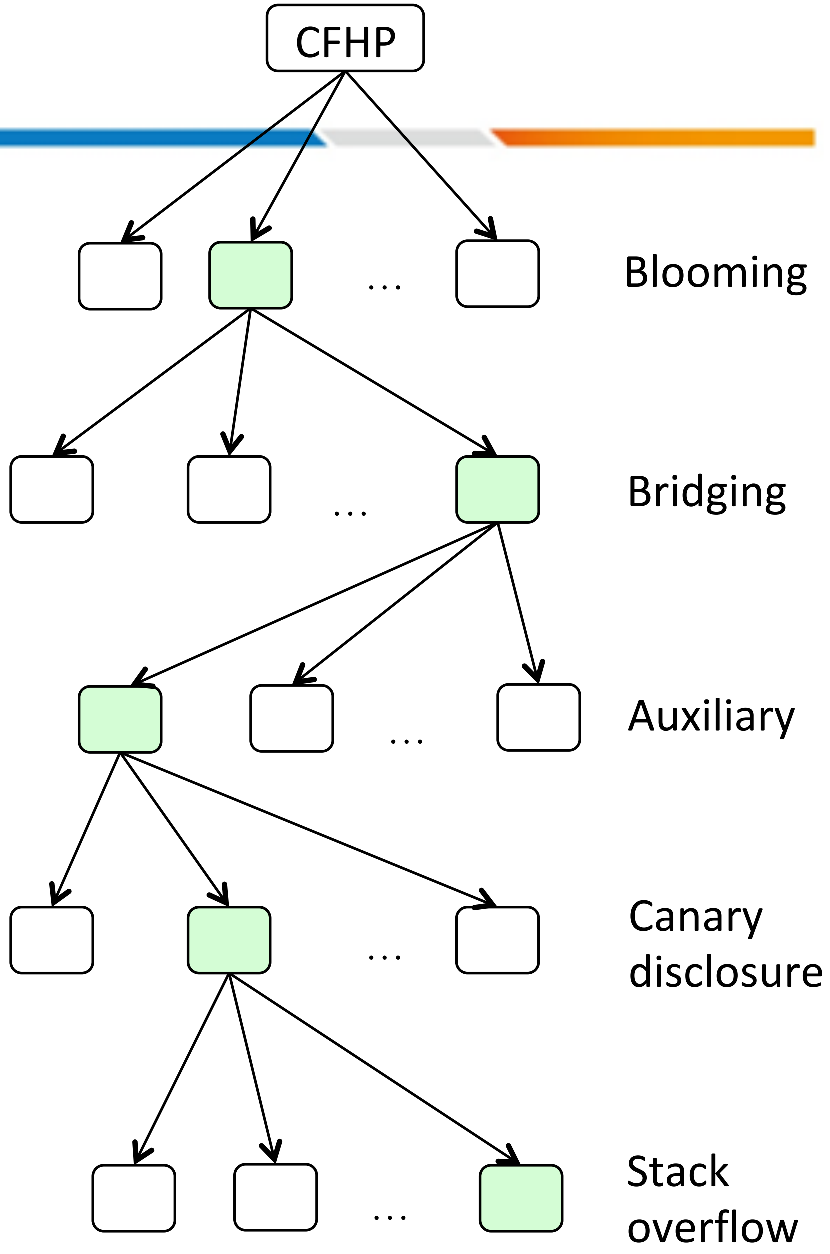
- Bridging gadget
  - Containing multiple controllable indirect calls
- Spawning two CFHPs and combining canary leak and stack smash into a single shot.

```
void regcache_mark_dirty(struct regmap *map){
  map->lock(map->lock_arg);// the 1st control-flow hijack
  map->cache_dirty=true;
  map->no_sync_defaults=true;
  map->unlock(map->lock_arg);// the 2nd control-flow hijack
}
```



Implementation

- Collecting candidate gadgets with static analysis
  - Built on IDA-Pro SDK 6.95
- Taking Exploit chain identification as a tree search problem
  - 28 workers to search different sub-trees concurrently
- Stitching gadgets with symbolic execution
  - Built on angr
  - Initialization: QEMU snapshot
  - Pruning: checking constraints satisfiability at key locations
  - State explosion mitigations:
    - Giving up after 20 steps for each stage
    - Entering a loop for less than 5 times.



Evaluation

- Test Cases:
  - 16 CVEs + 3 CTF challenges
- Comparing with previous exploit generation/hardening techniques
  - FUZE: relying on an exploit technique named "CR4 hijacking"
    - Not bypassing VMM-based hypervisor
    - Not bypassing exploitation pitfalls
  - Q : relying on stack-pivoting gadget which is not available in the kernel binary image

ID	Vulnerability type	Public exploit	Q	FUZE	KEPLER
CVE-2017-16995	OOB readwrite	✓†	✗	✗	✓
CVE-2017-15649	use-after-free	✓	✗	✓	✓
CVE-2017-10661	use-after-free	✗	✗	✗	✓
CVE-2017-8890	use-after-free	✗	✗	✗	✓
CVE-2017-8824	use-after-free	✓	✗	✓	✓
CVE-2017-7308	heap overflow	✓	✗	✗	✓
CVE-2017-7184	heap overflow	✓	✗	✗	✓
CVE-2017-6074	double-free	✓	✗	✗	✓
CVE-2017-5123	OOB write	✓†	✗	✗	✓
CVE-2017-2636	double-free	✗	✗	✗	✓
CVE-2016-10150	use-after-free	✗	✗	✗	✓
CVE-2016-8655	use-after-free	✓†	✗	✓†	✓
CVE-2016-6187	heap overflow	✗	✗	✗	✓
CVE-2016-4557	use-after-free	✗	✗	✗	✓
CVE-2017-17053	use-after-free	✗	✗	✗	✗
CVE-2016-9793	integer overflow	✗	✗	✗	✗
TCTF-credjar	use-after-free	✓†	✗	✗	✓
0CTF-knote	uninitialized use	✗	✗	✗	✓
CSAW-stringIPC	OOB read&write	✓†	✗	✗	✓

PennState

Evaluation (cont.)

- Finding exploit chain in 50 wall clock minutes
- Generating tens of thousands of exploit chains
- Hard to defeat because the gadget could not be easily removed.

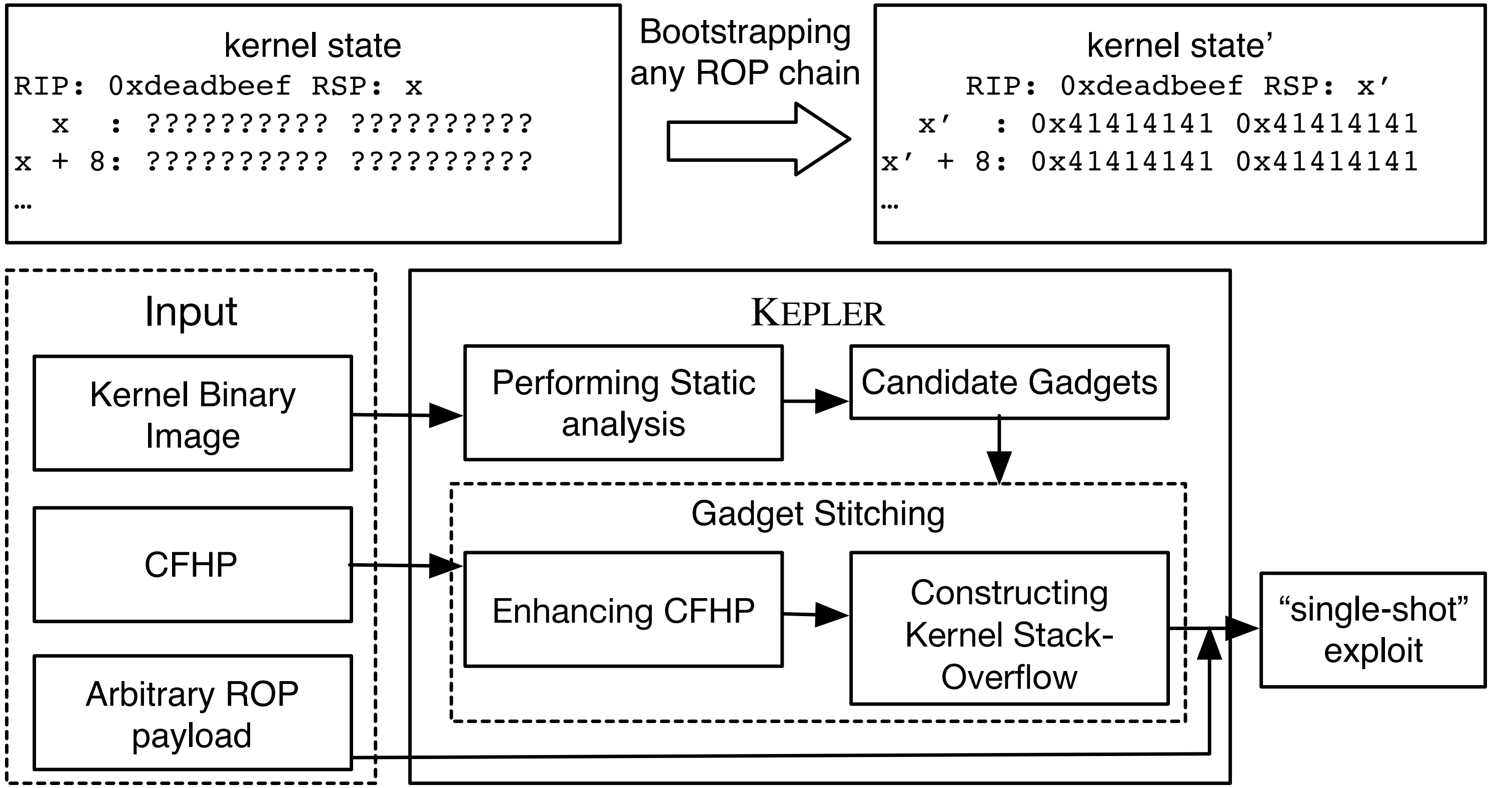
ID	Vulnerability type	G1	G2	G3	G4	First chain (min)	Total time (hour)	Total # of exploitation chains
CVE-2017-16995	OOB readwrite	41	114	27	201	45	37	29788
CVE-2017-15649	use-after-free	29	79	25	280	16	28	60207
CVE-2017-10661	use-after-free	28	78	30	301	17	25	49070
CVE-2017-8890	use-after-free	21	88	23	304	17	18	50471
CVE-2017-8824	use-after-free	63	101	35	306	50	70	164898
CVE-2017-7308	heap overflow	31	91	30	241	14	47	110176
CVE-2017-7184	heap overflow	31	95	31	254	24	37	93752
CVE-2017-6074	double-free	18	79	31	308	16	15	31436
CVE-2017-5123	OOB write	40	86	27	311	14	39	113466
CVE-2017-2636	double-free	18	89	29	289	29	19	26372
CVE-2016-10150	use-after-free	34	84	25	293	52	34	88499
CVE-2016-8655	use-after-free	18	109	32	260	15	17	47413
CVE-2016-6187	heap overflow	22	85	32	301	17	21	51954
CVE-2016-4557	use-after-free	21	80	21	295	16	37	40889
CVE-2017-17053	use-after-free	-	-	-	-	-	-	-
CVE-2016-9793	integer overflow	-	-	-	-	-	-	-
TCTF-credjar	use-after-free	35	89	25	292	25	14	82913
0CTF-knote	uninitialized use	21	89	33	318	17	36	40923
CSAW-stringIPC	OOB read&write	35	88	25	289	17	33	84414

PennState

Conclusions

- New technique: Single-shot exploitation is an effective kernel exploitation technique
  - Reduction: From "ROP is Turing Complete" to "control-flow hijacking is Turing Complete"
- New tool: Kepler is able to convert Linux kernel ROP bootstrapping task into a bounded tree-search problem and facilitate evaluation of control-flow hijacking primitive
  - Source: https://github.com/ww9210/ kepler-cfhp
- Suggestion: Kernel CFI should be deployed because other mitigations hardly stop exploitation

Thank you.


kernel state
RIP: 0xdeadbeef RSP: x
x : ?????????? ??????????
x + 8: ?????????? ??????????
…
Bootstrapping any ROP chain
kernel state'
RIP: 0xdeadbeef RSP: x'
x' : 0x41414141 0x41414141
x' + 8: 0x41414141 0x41414141
…
Input
Kernel Binary Image
CFHP
Arbitrary ROP payload
KEPLER
Performing Static analysis
Candidate Gadgets
Gadget Stitching
Enhancing CFHP
Constructing Kernel Stack-Overflow
"single-shot" exploit