fortenforge

ROP Guide

Return-oriented programming (ROP) is one of the most powerful and useful binary exploitation techniques we know. The concept–chaining together existing snippets of assembly instructions to achieve an objective–is not particularly difficult to understand, but building a successful ROP chain in practice often requires piecing together many different considerations that inexperienced pwners are less familiar with.

32-bit vs. 64-bit

Most of the servers and deszktops that we use today run on x86-64, an assembly instruction set released by AMD in 2000 (for this reason, it's sometimes referred to as AMD64 or x64). As the name indicates, it is a 64-bit instruction set, meaning that the registers and memory addresses are 64 bits = 8 bytes long. It's an extension of an older instruction set called x86, which was written by Intel. x86 is a 32-bit instruction set–meaning that the registers and memory addresses are 32-bits = 4 bytes long.

Binaries can be compiled as 64-bit or 32-bit. (That's why sometimes when you're downloading a piece of software you get to choose between a 64-bit and 32-bit version of the same application). Computers that are running x86-64 can usually also run 32-bit binaries. Thus, one of the first steps when exploiting a program is to determine if it's 32-bit or 64-bit. This can be done with the `file` command:

```
$ file again
again: ELF 64-bit LSB  executable, x86-64, version 1 (GNU/Linux),
statically linked, for GNU/Linux 2.6.32,
BuildID[sha1]=e311338e0e24ec622ec060b34a19d8328a61aa2c, not stripped
```

This binary is 64-bit.

```
$ file ropi
ropi: ELF 32-bit LSB  executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.24, not
stripped
```

...and this one is 32-bit.

There are a number of differences between 32-bit and 64-bit binaries that affect how you construct your ROP chain. The first and most obvious difference is the word-size. In 64-bit binaries you often use the function `p64` to pack addresses as strings. These strings will be 8 characters (64 bits) long. In 32-bit binaries you should use `p32` instead, which produces strings of length 4 (32 bits).

The other major difference is in the calling conventions. In 32-bit binaries, arguments are pushed onto the stack, right underneath the return address. This makes it relatively easy to pass arguments to a function in a ROP chain:

```
+-----------------+
|    function     |
+-----------------+
|    (4 bytes)    |  <-- padding
+-----------------+
|      arg1       |
+-----------------+
|      arg2       |
+-----------------+
```

Recall that the padding is where the program would return to *after* calling `function` (it's the return address of `function`). Since arguments go under a function's return address, we put `arg1` and `arg2` below the padding. Each of the boxes in the diagram must be 4 bytes = 32-bits long.

In contrast, in x86-64, arguments are passed via *registers*, specifically in the order of: `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`. (If a function has more than six arguments, all further ones are passed via the stack).

So, in x86-64, if you want to call a function with two arguments, you need to leverage a couple gadgets which will allow you to get the arguments into `rdi` and `rsi`. Specifically, we'll need a `pop rdi ; ret` gadget and a `pop rsi ; ret` gadget. Our ROP chain will look like:

```
+----------------+
|    pop rdi     |
+----------------+
|      arg1      |
+----------------+
|    pop rsi     |
+----------------+
|      arg2      |
+----------------+
|    function    |
+----------------+
```

Make sure you understand what's happening here: first the program will go to the `pop rdi` gadget and pop the next thing on the stack into `rdi`, namely `arg1`. Then the program will return to the `pop rsi` gadget and then pop `arg2` into `rsi`. Finally, the program will return to `function` with `rdi = arg1` and `rsi = arg2` as desired.

Statically-linked vs. Dynamically-linked

When you use a library function in your code and compile it, where does the library code end up in your binary? Well, if you *statically-link* the library with your program, then the functions in the library will appear in the binary itself. This can often lead to big, bloated binaries. Thus, it's often preferred that libraries are *dynamically-linked* with your program. Namely, when the program is run, a special process called the *linker* will copy the contents of the library into the program's memory space.

If a program is statically linked with a library, you can figure out the addresses of the library functions outside of a debugger using something like `readelf`. If a program is dynamically linked, then library function addresses are only determined at runtime, so you need a debugger to determine them. (And if ASLR is on, then those addresses will change every time you run the program! More on that in the next section).

You can use `file` to determine if your binary is statically or dynamically linked.

```
[again]> file again
again: ELF 64-bit LSB  executable, x86-64, version 1 (GNU/Linux),
statically linked, for GNU/Linux 2.6.32,
BuildID[sha1]=e311338e0e24ec622ec060b34a19d8328a61aa2c, not stripped
```

This binary is statically linked.

```
[ropi]> file ropi
ropi: ELF 32-bit LSB  executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.24, not
stripped
```

...and this one is dynamically linked.

In general, ROP chains become easier if your program is statically linked. Since your binary is bigger, the program is usually rich in valuable gadgets that you can chain together to do almost anything. If the `system` library function is used anywhere in the program and it's statically linked, then the `system` function will exist in the binary. If you can call it with the argument `"/bin/sh"`, you'll have your shell.

If `system` is not present and the binary is statically linked, you'll need to resort to making a syscall to `execve`, a low-level "syscall" that the operating system provides which does essentially the same thing as `system`.

Syscalls are called differently than regular functions, and they're called differently in x86 vs x86-64. In x86, you make a syscall using the `int 0x80` assembly instruction. You put the syscall number in `eax` (the number for `execve` is 11 = 0xb) and arguments go in `ebx`, `ecx`, `edx`, `esi` and `edi` in that order. Here's what a 32-bit ROP chain might look like:

```
+----------------+
|     popexx     |  <-- this is pop edx ; pop ecx ; pop ebx ; ret
+----------------+
|     0 (edx)    |
+----------------+
|     0 (ecx)    |
+----------------+
| /bin/sh (ebx)  |  <-- not the string "/bin/sh", but a ptr. to it
+----------------+
| ret2 (popeax)  |  <-- this is pop eax ; ret
+----------------+
|   0xb (eax)    |
+----------------+
|    int 0x80    |
+----------------+
```

This ROP chain executes `execve("/bin/sh", NULL, NULL)`. Note that we actually need both `ecx` and `edx` to be 0, since the second and third arguments of `execve` must be NULL if we want to spawn a shell.

In x86-64, you make a syscall using the `syscall` assembly instruction (makes sense right?). Again, you put the syscall number in `rax` (the number for `execve` is 59 = 0x3b) and arguments go in `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9` in that order. Here's what a 64-bit ROP chain might look like:

```
+----------------+
|     poprxx     |  <-- this is pop rdx ; pop rcx ; pop rbx ; ret
+----------------+
|     0 (rdx)    |
+----------------+
|     0 (rcx)    |
+----------------+
| /bin/sh (rbx)  |
+----------------+
| ret2 (poprax)  |  <-- this is pop rax ; ret
+----------------+
|  0x3b (rax)    |
+----------------+
|    syscall     |
+----------------+
```

ALSR on vs. ASLR off

After NX, the second most important buffer overflow mitigation is address-space layout randomization, or ASLR. ASLR is a property of the machine rather than the binary. There is a machine flag, which if set, will turn ASLR on, and if unset will turn ASLR off. In a CTF, the organizers will generally tell you if ASLR is on or off on their remote servers.

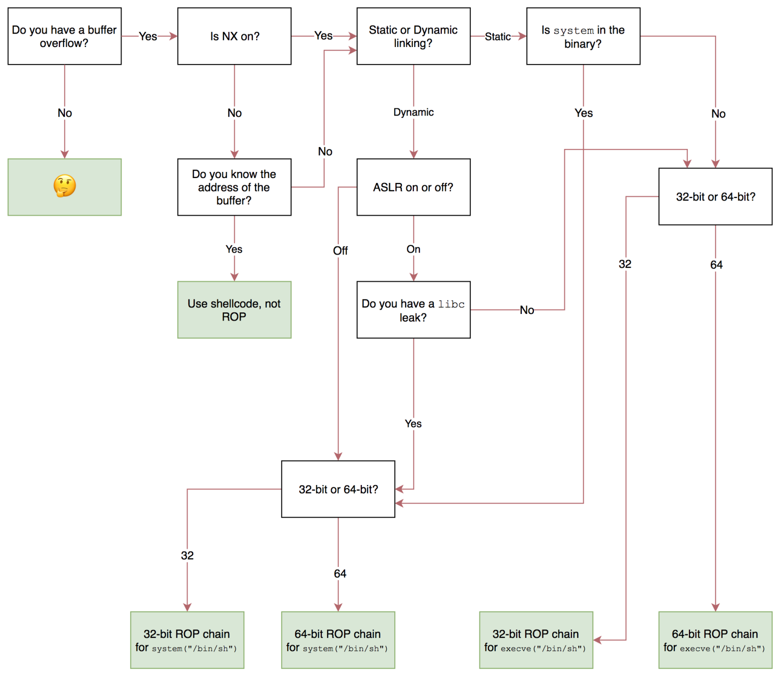
ASLR randomly arranges the memory layout of as much of the program as it can. However, it often can't randomize everything. Unless your program has something called PIE enabled (which we'll talk about later), the address of `main` and any other functions the program defines are fixed, and hardcoded to the binary. However, if your program is dynamically linked, ASLR *can* randomize the address of all the libraries that are linked with your program.

Specifically, every time you run your program, your libraries will be at positioned at a different offset in the memory space. So, even if you determine the address of `system` in the library using gdb, that address will change the next time you run the program.

There are really only two ways of dealing with ASLR: first, see if you can leverage the ROP gadgets present in the program itself to call `execve`, or second, see if you can leak an address of libc. If you can leak an address, then you can figure out where `system` is relative to that address, and then call `system("/bin/sh")`.

Flowchart

I compiled all the above information in to one nifty flowchart which you can find here. However I want to echo again the caution I gave in class: no one actually uses a flowchart to figure out what kind of ROP exploit they need to launch. It's far more useful to just understand conceptually what each of the modifications do, and then reason out the exploit path yourself.


Do you have a buffer overflow?
Yes
No
🤔
Is NX on?
Yes
No
Do you know the address of the buffer?
Yes
No
Use shellcode, not ROP
Static or Dynamic linking?
Static
Dynamic
Is system in the binary?
Yes
No
ASLR on or off?
Off
On
Do you have a libc leak?
No
Yes
32-bit or 64-bit?
32
64
32-bit or 64-bit?
32
64
32-bit ROP chain for system("/bin/sh")
64-bit ROP chain for system("/bin/sh")
32-bit ROP chain for execve("/bin/sh")
64-bit ROP chain for execve("/bin/sh")

How do I find the address of a function in a binary?

In pwndbg, you can do `print foo`. With `readelf` you can do:

`readelf --symbol <binary> | grep foo.`

How do I find the address of a string in a binary?

In pwndbg, you can do `search "bar"`. With ROPgadget you can do:

`ROPgadget --binary <binary> --string "bar"`.

Why are the above two methods different?

Function names don't actually exist in the process's memory space. They exist as *symbols* in the binary, which are essentially useful names for various addresses that binaries keep around for debugging or linking. You can search memory for the name of a function, since it won't be there. Instead you have to ask `readelf` to print out the symbols in a binary.

In contrast, strings do exist in a process's memory space, so you can find them using pwndbg's `search` command. (However, the *names* of string variables are symbols, so you'd find those using `readelf`).

How do I find out what libraries a binary is using?

Use `ldd`, a command line tool:

```
$ ldd use_leak
linux-vdso.so.1 =>  (0x00007ffff7ffd000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7c14000)
/lib64/ld-linux-x86-64.so.2 (0x0000555555554000)
```

Most often, you're concerned with `libc.so.6` which is libc.

How do I find ROP gadgets?

Use ROPgadget, a command line tool:

```
$ ROPgadget --binary use_leak
Gadgets information
============================================================
0x0000000000400532 : adc byte ptr [rax], ah ; jmp rax
0x000000000040052e : adc dword ptr [rbp - 0x41], ebx ; push rax ; adc
byte ptr [rax], ah ; jmp rax
0x00000000004006af : add bl, dh ; ret
  .
  .
  .
```

For large binaries (like libc), ROPgadget takes a while to run, so it's often useful to save its output in a file, and then grep for what you want:

```
$ ROPgadget --binary use_leak > rop.txt
$ cat rop.txt | grep "pop rdi ; ret"
0x00000000004006a3 : pop rdi ; ret
```

How do I determine the distance between the return address and the buffer I'm overflowing?

Here's the process that I've found most reliable:

1. First run pwndbg on the binary: `pwndbg <binary>`
2. Break at `main` : `b main`
3. Run the binary: `r`
4. Disassemble the function where the overflow happens: `disassemble foo`
5. Set a breakpoint right after the call to user input: `b *foo+45`
6. Continue: `c`
7. When prompted for user input, enter a few A's: `AAAAA`
8. When you hit the breakpoint, run `stack 50`
9. Examine the stack. The address right under `rbp` should be your return address. The address which contains a bunch of A's should be your buffer.
10. Compute the distance between the two: `distance 0x40abcd 0x40abff`

What things are and aren't affected by ASLR?

In short, ALSR always randomizes the locations of shared libraries, the stack, the heap, and some other sections you don't know about. It usually does not randomize the .text section of a program which contains all of the user-defined functions.