# Return oriented programming (ROP)

## Summary

In this exercise, we will learn about return oriented programming in Ubuntu Linux and about relevant defensive measures for mitigation of associated risks.



Figure 1-www.ransomizer.com

## Prerequisites

* Setup an Ubuntu VM as outlined in the VM setup instructions on Blackboard
* Install cmake using

$ sudo apt-get install cmake

* Download rp++ using

$ git clone https://github.com/0vercl0k/rp.git

* Build rp++ using

$ cd rp ; mkdir build ; cd build && cmake .. && make

## Background

* Return-oriented programming, or ROP, is an exploit technique that allows code execution even in the presence of defenses such as
  + Non-executable memory (W ^ X and DEP)
  + Code integrity checks
* Instead of returning to lib-c functions or injecting shell code on the stack
  + Manipulate the stack to return into a short assembly instruction sequence
  + These sequences, or gadgets, are already located in executable memory sections available in
    - binary itself
    - libraries linked to the binary
* Gadgets are versatile and can
  + Load/Store data between registers and memory
    - Load constants
    - Load from memory
    - Store (write) to memory
  + Perform arithmetic and logic computations
    - Add, Subtract, Multiply
    - XOR, AND, OR, NOT
    - Shift and Rotate
  + Manipulate control flow of programs
    - Perform Unconditional/Conditional jumps
    - Call system and library functions
* As a language, libraries of ROP gadget sequences can even be Turing complete
  + I.e., could be used to emulate a Turing machine, or a formal model for a system of rules, states and transitions that performs a computing function
  + Assumes suitable gadgets exist in executable memory
* If we manipulate the stack appropriately, we can generate Turing complete code by using a sequence of returns into gadgets

## Details

### Locating ROP gadgets

* Inspect binaries for "RET"-like codes and the immediate instructions that precede them
  + 0xC3-RET, near return (version used in the System V ABI)
  + 0xC2-RET imm16, near return with stack unwind
  + 0xCB-RET, far return
  + 0xCA-RET imm16, far return with stack unwind
* We will use rp++ for locating gadgets
  + rp++ is is a full-cpp written tool that aims to find ROP sequences in PE/Elf/Mach-O x86/x64 binaries and supports x86\_64 bit architecture and Intel syntax queries
* For implementing shellcode, we will need the following gadgets in "/lib/x86\_64-linux-gnu/libc-2.27.so"

Table 1-Libc gadgets for implementing shellcode

|  |  |
| --- | --- |
| **Gadget sequence** | **Offset** |
| pop rdi; ret; |  |
| pop rsi; ret; |  |
| pop rdx; ret; |  |
| xor rax, rax; ret; |  |
| mov eax, 0x3b; syscall; ret; |  |

* Use rp++ to locate offsets for the above gadgets in "/lib/x86\_64-linux-gnu/libc-2.27.so"
  + Generate a database to search using

$ rp-lin-x64 --file=/lib/x86\_64-linux-gnu/libc-2.27.so --rop=1 --unique > lib.x86\_64-linux-gnu.libc-2.27.so.txt

* + Search the generated "lib.x86\_64-linux-gnu.libc-2.27.so.txt" for the sequences listed in the table above and capture the offsets in the column to the right
* It is also useful locate the following two byte sequences in "/lib/x86\_64-linux-gnu/libc-2.27.so"

Table 2-Libc byte sequences for implementing shellcode

|  |  |
| --- | --- |
| **Byte sequence** | **Offset** |
| 0x0000000000000000 |  |
| "/bin/sh" |  |

* Use rp++ to locate offsets for the above byte sequences in "/lib/x86\_64-linux-gnu/libc-2.27.so"
  + Search "/lib/x86\_64-linux-gnu/libc-2.27.so" for a null string using the following command and capture the offset in the table above

$ rp-lin-x64 --file=/lib/x86\_64-linux-gnu/libc-2.27.so --search-hexa="\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"

* + Search "/lib/x86\_64-linux-gnu/libc-2.27.so" for "/bin/sh" string using the following command and capture the offset in the table above

$ rp-lin-x64 --file=/lib/x86\_64-linux-gnu/libc-2.27.so --search-hexa="/bin/sh\x00"

### Chaining ROP gadgets

* Recall syscall’s calling convention from our shellcode exercise and the Sys-V ABI, AMD64 (see 3.2.3 and A.2.1)
  + place 1st argument in rdi
  + place 2nd argument in rsi
  + place 3rd argument in rdx
  + place number of the syscall in rax
    - For execve, syscall number is 0x3b, or 59

$ cat /usr/include/x86\_64-linux-gnu/asm/unistd\_64.h | grep execve

* + Execute syscall CPU instruction
  + On completion, rax contains the status result of the syscall
* Recall execve’s calling convention from our shellcode exercise and man execve
  + execve(filename, argv[], envp[])
  + To get a bash shell, we will pass in
    - Filename (rdi)=address of "/bin/sh" byte sequence
    - argv[] (rsi)=address of 0x0 byte sequence
    - envp[] (rdx)=address of 0x0 byte sequence
    - 0x3b (rax)=syscall number
* Using the ball-peen hammer style of stack-smashing, our stack layout will look as follows

Table 3-Rop chain for execve syscall shellcode

|  |  |
| --- | --- |
| **Location** | **Description** |
| Old rbp | Valid, dummy rbp |
| Return fptr (gadget 1) | Address of 1st gadget (pop rdi; ret;) |
| arg | Address of "/bin/sh" |
| Gadget 2 | Address of 2nd gadget (pop rsi; ret;) |
| Arg | Address of 0x0 |
| Gadget 3 | Address of 3rd gadget (pop rdx; ret;) |
| Arg | Address of 0x0 |
| Gadget 4 | Address of 4th gadget (xor rax, rax; ret;) |
| Gadget 5 | Address of 5th gadget (mov eax, 0x3b; syscall; ret; |

### Example shellcode injection using ROP

* Prerequisites
  + Vulnerability that allows attacker to control stack frame prior to function returning
    - We will use our previously discussed buffer-overflow vulnerability and the ball-peen hammer style of stack-frame corruption
  + Suitable gadgets are available in memory to implement our shellcode using arc injection
  + ASLR and stack protections are turned off for demonstration purposes
  + execstack is turned off (*disabled to prevent code injection*)
* Download the project into a local sandbox

$ git clone <https://gitlab.com/underpantsgnomes/softwaresecurity/ropexploit>

* Import the project into eclipse and build it
* The exploit takes two arguments and its usage is

$ ./ropexploit <oldrbp> <libcbase>

* Create a debug configuration for the demonstration with dummy placeholders for the two arguments, <oldrbp> and <libcbase>, and specify your ~/.gdbinit
* Use a breakpoint and run it to locate libc’s base address during execution
  + Use the gdb command "(gdb) info proc mapping" to query mappings
  + In the Table below, the resulting value for libc’s base address would be 0x7ffff77e0000

Table 4-Example use of gdb to obtain libc's base address

|  |
| --- |
| (gdb) info proc mapping  process 64980  Mapped address spaces:  Start Addr End Addr Size Offset objfile  0x555555554000 0x555555557000 0x3000 0x0 /home/reub/sandbox/rop/Debug/rop  0x555555756000 0x555555757000 0x1000 0x2000 /home/reub/sandbox/rop/Debug/rop  0x555555757000 0x555555758000 0x1000 0x3000 /home/reub/sandbox/rop/Debug/rop  0x7ffff77e0000 0x7ffff79c7000 0x1e7000 0x0 /lib/x86\_64-linux-gnu/libc-2.27.so  ...  (gdb) |

* Continue the program to obtain the <oldrbp> value from the console when vulnFunction is executed
* Set a breakpoint on the return command in vulnFunction and rerun the demo with the appropriate <oldrbp> and <libcbase> arguments
* Enable instruction-stepping mode and step in to the first gadget
* Step through the sequence of gadgets up until the syscall instruction
  + Watch the rdi, rsi, rdx, and rax registers change as you step through the sequence of gadgets
* Resume execution at the syscall to get a shell
  + Execute the following in the bash shell when your shell code call works (i.e., when you get the $ bash prompt in your debugging session)

|  |
| --- |
| $ /usr/games/cowsay -f dragon "Grrrr-rop!"  < Grrr! >  --------  \ / \ //\  \ |\\_\_\_/| / \// \\  /0 0 \\_\_ / // | \ \  / / \/\_/ // | \ \  @\_^\_@'/ \/\_ // | \ \  //\_^\_/ \/\_ // | \ \  ( //) | \/// | \ \  ( / /) \_|\_ / ) // | \ \_\  ( // /) '/,\_ \_ \_/ ( ; -. | \_ \_\.-~ .-~~~^-.  (( / / )) ,-{ \_ `-.|.-~-. .~ `.  (( // / )) '/\ / ~-. \_ .-~ .-~^-. \  (( /// )) `. { } / \ \  (( / )) .----~-.\ \-' .~ \ `. \^-.  ///.----..> \ \_ -~ `. ^-` ^-\_  ///-.\_ \_ \_ \_ \_ \_ \_}^ - - - - ~ ~-- ,.-~  /.-~ |

### Defenses

* Control flow integrity

## References

* Ritvik Sachdev and Purushottam Kulkarni, final project for JHU-ISI EN.650.660, 2016
* <https://gitlab.com/underpantsgnomes/softwaresecurity/ropexploit>
* <https://github.com/0vercl0k/rp>
* 2012, Roemer et al., Return-Oriented Programming: Systems, Languages, and Applications, ACM Transactions on Information and System Security, Volume 15 Issue 1, March 2012
* Sys-V ABI, AMD64