Australian National University

Basic Binary Analysis in Linux

COMP3703 Software Security

Slides prepared by H. Gunadi & A. Tiu. Based on Chapter 5 of Andriesse's "Practical Binary Analysis", No Starch Press, 2019.

Outline

- Introduction to tools for basic binary analysis.
- Walk through a CTF problem using basic binary analysis techniques and tools.

Overview

- Finding the id[illegible] a file.
- Analysing executables: resolving dependencies
- Function identification.
- Basic binary analysis techniques, e.g.,
  - Extracting executable header
  - Extracting strings in executables
  - System calls and library calls
  - Debugger for dynamic information (e.g., GDB).

Tools

GNU Binutils tools:

- c++filt: Filter to demangle encoded C++ symbols.
- objdump: Display information from object files.
- readelf: Display information from any ELF format object file.
- strings: Lists printable strings from files.
- nm: Lists symbols from object files.

Tools

- base64: Encoding to base64 format.
- file: Get some information on the file classification.
- ldd: See the shared libraries used by a binary.
- strace: See the system calls made by a process.
- ltrace: See the library calls made by a process.
- xxd: dump the content of a file in hexadecimal format.
- hexedit: open file in hexadecimal format and edit in-place.
- dd: convert and copy a file.

CTF Walkthrough

The CTF problem

- The CTF problem can be downloaded from Wattle (lab2).
- Payload file: 'lab2/ctf/payload'.
- Objective: discover the flags planted in payload.
- The CTF problem contains several levels; solving the challenge in one level unlocks the challenge for the next level.
- We'll solve Level 1 in this lecture.

Identifying the payload

```
$ file payload
payload: ASCII text

$ head payload
```


```
H4sIABzY61gAA+xaD3RTVZq/Sf+lFJIof1r+2aenKKh0klJKi4MmJaUvWrTSFlgR0jRN20iadpKX
UljXgROKjbUOKuOfWWfFnTlzZs/ZXTln9nTRCTHYERhmZ5c/R2RGV1lUFTAFH/DNYoZD9vvvubd57
bcBl1ln3bL6e9Hvf9+733e/+v+/en0dqId80WYAWLVqI3LpooUXJgUpKFy6yEOsCy6KSRQtLLQsW
EExdWkIEyzceGVA4JLmDgkCaA92XTXel9/9H6ftVNcv0Ot2orCe3E5RiJHuVbUw/fH3SxkbKSS78
v47MJtkgZynS2YhNxYeZa84NLF0G/DLhV66X5XK9TcVnsXSc6xO8S1UCm4q/M5moOCHCqB3Geny2
rD0+u1HFD7I4junVdnpmN8zshll6zglPr1eXL5P96pm+npWLcwdL51CkR6r9UGrGZ801zN+1NhUv
ZelKNXb3gl02+fpkZnwFyy9VvQgsfs5503zH72sqK/2Ov3m+3xcId8/vLi+bX1ZaHOooLqExmVna
6rsbaHpejwKLeQqR+wC+n/ePA3n/duKu2kNvL175+MxD7z75W8GC76aSZLv1xgSdkGnLRV0+/KbD
7+UPnnhwadWbZ459b/Wsl/o/NZ468olxo3P9wOXK3Qe/a8fRmwhvcTVdl0J/UDe+nzMp9M4U+n9J
oX8jhT5HP77+ZIr0JWT8+NvI+OnvTpG+NoV/Qwr9Vyn0b6bQkxTl+ixF+p+m0N+qx743k+wWGlX6
```

Identifying the payload: base64

- Base64 is an encoding technique commonly used to represent binary data in textual form.
- Each character represents 6-bits of information, so to encode a byte you'd need at least two characters.
  - Encoded as A-Z, a-z, 0-9, and two extra characters (+, /).
  - Using the character '=' as padding.
  - Usually organized in neat rows (but not always).
- Use the `base64` command to encode and decode.

```
$ base64 –d payload > decoded_payload
```

Identifying the payload: file type

- A basic (and use[illegible] before starting any analysis is to identify the type [illegible], using the 'file' command.
- It can also 'peek' inside compressed files.


```
$ file decoded_payload

lvl1_decrypt: gzip compressed data, last modified: Mon Apr 10 19:08:12
2017, from Unix

$ file -z decoded_payload

lvl1_decrypt: POSIX tar archive (GNU) (gzip compressed data, last modified:
Mon Apr 10 19:08:12 2017, from Unix)

$ tar xvzf decoded_payload
ctf
67b8601
```

Identifying the payload: file type

- The first file (ctf) is an executable, but failed to execute (missing library).
- The second is an image (bitmap) file.


```
$ file ctf
ctf: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=29aeb60bcee44b50d1db3a56911bd1de93cd2030, stripped

$ file 67b8601
67b8601: PC bitmap, Windows 3.x format, 512 x 512 x 24

$ ./ctf
./ctf: error while loading shared libraries: lib5ae9b7f.so:
       cannot open shared object file: No such file or directory
```

Identifying the payload: ldd

Use ldd to identify dependencies:


```
$ ldd ctf
linux-vdso.so.1 =>  (0x00007ffd4adfb000)
lib5ae9b7f.so => not found
libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6
(0x00007f81bd7c7000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f81bd5b1000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f81bd1e7000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f81bcede000)
/lib64/ld-linux-x86-64.so.2 (0x00007f81bdb49000)
```

Finding the missing shared libray

- Recall that shared library is an ELF file.
- lib5ae9b7f.so doesn't look like a standard library, so likely a custom library.
- Maybe it's hidden somewhere – grep for the 'magic byte' for ELF as a first attempt:

```
$ grep ELF *
Binary file 67b8601 matches
Binary file ctf matches
```

Finding the missing shared library

ELF header hidden in file 67b8601, starting at offset 52.

```
$ xxd  67b8601 | head

00000000: 424d 3800 0c00 0000 0000 3600 0000 2800  BM8.......6...(.
00000010: 0000 0002 0000 0002 0000 0100 1800 0000  ................
00000020: 0000 0200 0c00 c01e 0000 c01e 0000 0000  ................
00000030: 0000 0000 7f45 4c46 0201 0100 0000 0000  .....ELF........
00000040: 0000 0000 0300 3e00 0100 0000 7009 0000  ......>.....p...
00000050: 0000 0000 4000 0000 0000 0000 7801 0000  ....@.......x!..
00000060: 0000 0000 0000 0000 4000 3800 0700 4000  ........@.8...@.
00000070: 1b00 1a00 0100 0000 0500 0000 0000 0000  ................
00000080: 0000 0000 0000 0000 0000 0040 0804 0000  ...............
00000090: 0000 0000 f40e 0000 0000 0000 f40e 0000  ................
```

Finding the missing shared library

- An alternative (a[illegible]r) way is to use an advanced digital forensic t[illegible] binwalk.


```
$ sudo apt install binwalk
$ binwalk 67b8601

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PC bitmap, Windows 3.x format,, 512 x 512 x 24
52            0x34            ELF, 64-bit LSB shared object, AMD x86-64, ...
```

Finding the missing shared library

Extract the executable h

```
$ dd skip=52 count=64 if=67b8601 of=elf_header bs=1
64+0 records in
64+0 records out
```



```
$ xxd elf_header
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0300 3e00 0100 0000 7009 0000 0000 0000  ..>.....p.......
00000020: 4000 0000 0000 0000 7821 0000 0000 0000  @.......x!......
00000030: 0000 0000 4000 3800 0700 4000 1b00 1a00  ....@.8...@.....
```

Extracting the shared library

```
$ readelf -h elf_header
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x970
  Start of program headers:          64 (bytes into file)
  Start of section headers:          8568 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         7
  Size of section headers:           64 (bytes)
  Number of section headers:         27
  Section header string table index: 26
```

e_shoffset

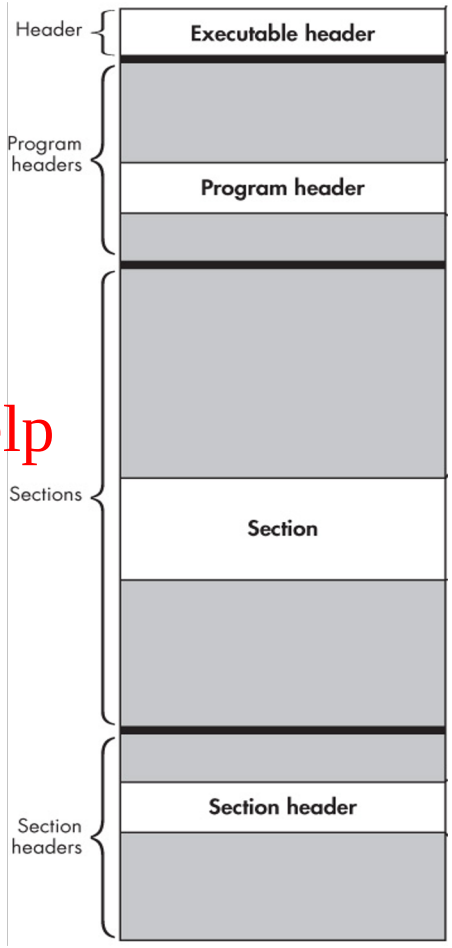
e_shentsize

e_shnum

Extracting the shared library

- Find out the size of the shared library from the executable he
- Recall that section headers are located at the end of the file.
- We have the offset to the section headers and we can also derive the size of the section headers.
- The size of the file of the shared library:

  size = e_shoffset + (e_shnum * e_shentsize)
  = 8,568 + (27 * 64)
  = 10,296



Extracting the shared library

ELF header starts at offs[illegible]ze 10,296 bytes.

```
$ dd skip=52 count=10296 if=67b8601 of=lib5ae9b7f.so bs=1
10296+0 records in
10296+0 records out
10296 bytes (10 kB, 10 KiB) copied, 0.0287996 s, 358 kB/s
```

Analysing the shared library

What does it do? L xported functions.

```
$ readelf –hs lib5ae9b7f.so

... <<some output omitted>> ...

Symbol table '.dynsym' contains 22 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
                  <<some entries omitted>>
    11: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND memcpy@GLIBC_2.14 (6)
    12: 0000000000000bc0   149 FUNC    GLOBAL DEFAULT   12 _Z11rc4_encryptP11rc4_sta
    13: 0000000000000cb0   112 FUNC    GLOBAL DEFAULT   12 _Z8rc4_initP11rc4_state_t
    14: 0000000000202060     0 NOTYPE  GLOBAL DEFAULT   24 _end
    15: 0000000000202058     0 NOTYPE  GLOBAL DEFAULT   23 _edata
    16: 0000000000000b40   119 FUNC    GLOBAL DEFAULT   12 _Z11rc4_encryptP11rc4_sta
    17: 0000000000000c60     5 FUNC    GLOBAL DEFAULT   12 _Z11rc4_decryptP11rc4_sta
    18: 0000000000202058     0 NOTYPE  GLOBAL DEFAULT   24 __bss_start
    19: 00000000000008c0     0 FUNC    GLOBAL DEFAULT    9 _init
    20: 0000000000000c70    59 FUNC    GLOBAL DEFAULT   12 _Z11rc4_decryptP11rc4_sta
    21: 0000000000000d20     0 FUNC    GLOBAL DEFAULT   13 _fini
```

Analysing the shared library

- C++ allows overloaded functions
  - Multiple functions with the same name but different arguments are allowed.
  - Compilers emit *mangled* function names (i.e., original function name + encoding of parameters).
- Function names can be *demangled* using the nm command.

Analysing the shared library

Demangling function nam

```
$ nm -D --demangle lib5ae9b7f.s
0000000000202058 B __bss_start
                 w __cxa_finalize
0000000000202058 D _edata
0000000000202060 B _end
0000000000000d20 T _fini
                 w __gmon_start__
00000000000008c0 T _init
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
                 w _Jv_RegisterClasses
                 U malloc
                 U memcpy
                 U __stack_chk_fail
0000000000000c60 T rc4_decrypt(rc4_state_t*, unsigned char*, int)
0000000000000c70 T rc4_decrypt(rc4_state_t*, std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >&)
0000000000000b40 T rc4_encrypt(rc4_state_t*, unsigned char*, int)
0000000000000bc0 T rc4_encrypt(rc4_state_t*, std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >&)
0000000000000cb0 T rc4_init(rc4_state_t*, unsigned char*, int)
                 U std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
>::_M_create(unsigned long&, unsigned long)
                 U std::__throw_logic_error(char const*)
```

Looking for hints with `strings`

- The `argv[1]` is likely the first argument of the program.
- This provides a clue that the program is expecting a string 'show_me_the_flag' as its first argument.

```
$ strings ctf
/lib64/ld-linux-x86-64.so.2
lib5ae9b7f.so
❷ __gmon_start__
_Jv_RegisterClasses
_ITM_deregisterTMCloneTable
_ITM_registerTMCloneTable
_Z8rc4_initP11rc4_state_tPhi
❸ DEBUG: argv[1] = %s
❹ checking '%s'
❺ show_me_the_flag
>CMb
flag = %s
guess again!
❻ It's kinda like Louisiana. Or Dagobah. Dagobah - Where Yoda lives!
;*3$"
zPLR
GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609
❼ .shstrtab
```

Displaying the return value of a program

- The environment variable '?' in bash stores the return value of the program that has just been run.
- By convention, the return value indicates an error number (0 for no error).
- This is useful to check if a program terminates normally or with errors.


```
$ ./ctf show_me_the_flag
checking 'show_me_the_flag'
ok

$ echo $?
1
```

Tracing system and library calls

- strace: run the prog[illegible]utput all system calls made by the program (e.g., open[illegible]te files).
- ltrace: similar to strace but shows library calls.
- The environment variable LD_LIBRARY_PATH tells the program where to find the libraries.

Tracing system and library calls

Library call tracing shows th uses an environment variable GUESSME.

```
$ LD_LIBRARY_PATH=. ltrace -i -C ./ctf show_me_the_flag
[0x400fe9] __libc_start_main(0x400bc0, 2, 0x7fffaf3e00a8, 0x4010c0 <unfinished ...>
[0x400c44] __printf_chk(1, 0x401158, 0x7fffaf3e236f, 160checking 'show_me_the_flag'
) = 28
[0x400c51] strcmp("show_me_the_flag", "show_me_the_flag") = 0
[0x400cf0] puts("ok"ok
)                                 = 3
[0x400d07] rc4_init(rc4_state_t*, unsigned char*, int)(0x7fffaf3dfe70, 0x4011c0, 66, -1) = 0
[0x400d14] std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::assign(char
const*)(0x7fffaf3dfdb0, 0x40117b, 58, 3) = 0x7fffaf3dfdb0
[0x400d29] rc4_decrypt(rc4_state_t*, std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >&)(0x7fffaf3dfe10, 0x7fffaf3dfe70, 0x7fffaf3dfdb0, 0x7e889f91) = 0x7fffaf3dfe10
[0x400d36] std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
>::_M_assign(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >
const&)(0x7fffaf3dfdb0, 0x7fffaf3dfe10, 0x7fffaf3dfe20, 0) = 0
[0x400d53] getenv("GUESSME")                                  = nil
[0xffffffffffffffff] +++ exited (status 1) +++
```

Tracing system and library calls

- So it seems that GUESSME should contain a password that is passed on to the rc4_decrypt function.
- But still no clues on the expected value of GUESSME.

```
E='foobar' ./ctf show_me_the_flag
 'show_me_the_flag'

guess again!
```



```
$ GUESSME='foobar' ltrace -i -C ./ctf show_me_the_flag
...
[0x400d53] getenv ("GUESSME") = "foobar"
❶ [0x400d6e] std::__cxx11::basic_string<char, std::char_traits<char>,
                std::allocator<char> >:: assign (char const*)
                (0x7fffc7af2b00, 0x401183, 5, 3) = 0x7fffc7af2b00
❷ [0x400d88] rc4_decrypt (rc4_state_t*, std::__cxx11::basic_string<char,
                std::char_traits<char>, std::allocator<char> >&)
                (0x7fffc7af2b60, 0x7fffc7af2ba0, 0x7fffc7af2b00, 0x401183) = 0x7fffc7a
[0x400d9a] std::__cxx11::basic_string<char, std::char_traits<char>,
                std::allocator<char> >:: _M_assign (std::__cxx11::basic_string<char,
                std::char_traits<char>, std::allocator<char> > const&)
                (0x7fffc7af2b00, 0x7fffc7af2b60, 0x7700a0, 0) = 0
[0x400db4] operator delete (void*)(0x7700a0, 0x7700a0, 21, 0) = 0
❸ [0x400dd7] puts ("guess again!"guess again!) = 13
```

Examining instruction-level behaviour

- We know that a stri... "guess again" is pri... when GUESSME contains a wrong value.
- Find the location of "guess again" in .rodata, and identify where it is referenced in .text.
- The string "guess again" is located at address 0x4011af.

```
...bjdump -s --section .rodata ctf

...:     file format elf64-x86-64

Contents of section .rodata:
 401140  01000200 44454255 473a2061 7267765b  ....DEBUG: argv[
 401150  315d203d 20257300 63686563 6b696e67  1] = %s.checking
 401160  20272573 270a0073 686f775f 6d655f74   '%s'..show_me_t
 401170  68655f66 6c616700 6f6b0044 89df919f  he_flag.ok.D....
 401180  887e009a 5b38babe 27ac0e3e 434d6285  .~..[8..'..>CMb.
 401190  55868954 3848a34d 00192d76 40505e3a  U..T8H.M..-v@P^:
 4011a0  00726200 666c6167 203d2025 730a0067  .rb.flag = %s..g
 4011b0  75657373 20616761 696e2100 00000000  uess again!.....
 4011c0  49742773 206b696e 6461206c 696b6520  It's kinda like
 4011d0  4c6f7569 7369616e 612e204f 72204461  Louisiana. Or Da
```

Examining instruction-level behaviour

The "guess again" str[illegible]ed by the instruction at 0x400dcd.
This is the jump targe[illegible]ilure case.


```
$ objdump -M intel -dj .text ctf
..
  400dc0: 0f b6 14 03           movzx  edx,BYTE PTR [rbx+rax*1]
  400dc4: 84 d2                 test   dl,dl
  400dc6: 74 05                 je     400dcd <_Unwind_Resume@plt+0x22d>
  400dc8: 3a 14 01              cmp    dl,BYTE PTR [rcx+rax*1]
  400dcb: 74 13                 je     400de0 <_Unwind_Resume@plt+0x240>
  400dcd: bf af 11 40 00        mov    edi,0x4011af
  400dd2: e8 d9 fc ff ff        call   400ab0 <puts@plt>
  400dd7: e9 84 fe ff ff        jmp    400c60 <_Unwind_Resume@plt+0xc0>
  400ddc: 0f 1f 40 00           nop    DWORD PTR [rax+0x0]
  400de0: 48 83 c0 01           add    rax,0x1
  400de4: 48 83 f8 15           cmp    rax,0x15
  400de8: 75 d6                 jne    400dc0 <_Unwind_Resume@plt+0x220>
...
```

Examining instruction-level behaviour

The failure case is re[illegible] a loop that starts at address 0x400dc0. The loop c[illegible] two strings: one at base address rbx, and the other at base address rcx.


```
$ objdump -M intel -dj .text ctf
..
  400dc0: 0f b6 14 03          movzx  edx,BYTE PTR [rbx+rax*1]
  400dc4: 84 d2                test   dl,dl
  400dc6: 74 05                je     400dcd <_Unwind_Resume@plt+0x22d>
  400dc8: 3a 14 01             cmp    dl,BYTE PTR [rcx+rax*1]
  400dcb: 74 13                je     400de0 <_Unwind_Resume@plt+0x240>
  400dcd: bf af 11 40 00       mov    edi,0x4011af
  400dd2: e8 d9 fc ff ff       call   400ab0 <puts@plt>
  400dd7: e9 84 fe ff ff       jmp    400c60 <_Unwind_Resume@plt+0xc0>
  400ddc: 0f 1f 40 00          nop    DWORD PTR [rax+0x0]
  400de0: 48 83 c0 01          add    rax,0x1
  400de4: 48 83 f8 15          cmp    rax,0x15
  400de8: 75 d6                jne    400dc0 <_Unwind_Resume@plt+0x220>
...
```

Examining instruction-level behaviour

The register rcx shou the address to the string that GUESSME is suppos ch.
So we just need to find this address at runtime.


```
$ objdump -M intel -dj .text ctf
..
  400dc0: 0f b6 14 03           movzx  edx,BYTE PTR [rbx+rax*1]
  400dc4: 84 d2                 test   dl,dl
  400dc6: 74 05                 je     400dcd <_Unwind_Resume@plt+0x22d>
  400dc8: 3a 14 01              cmp    dl,BYTE PTR [rcx+rax*1]
  400dcb: 74 13                 je     400de0 <_Unwind_Resume@plt+0x240>
  400dcd: bf af 11 40 00        mov    edi,0x4011af
  400dd2: e8 d9 fc ff ff        call   400ab0 <puts@plt>
  400dd7: e9 84 fe ff ff        jmp    400c60 <_Unwind_Resume@plt+0xc0>
  400ddc: 0f 1f 40 00           nop    DWORD PTR [rax+0x0]
  400de0: 48 83 c0 01           add    rax,0x1
  400de4: 48 83 f8 15           cmp    rax,0x15
  400de8: 75 d6                 jne    400dc0 <_Unwind_Resume@plt+0x220>
...
```

Dumping a dynamic string using gdb


```
$ gdb ./ctf
(gdb) b *0x400dc8
Breakpoint 1 at 0x400dc8
(gdb) set env GUESSME=foobar
(gdb) run show_me_the_flag
Starting program: /home/binary/code/chapter3/ctf show_me_the_flag
checking 'show_me_the_flag'
ok
Breakpoint 1, 0x0000000000400dc8 in ??()
(gdb) display/i $pc
1: x/i $pc
=> 0x400dc8:    cmp    (%rcx,%rax,1),%dl
(gdb) info registers rcx
rcx            0x615050 6377552
(gdb) info registers rax
rax            0x0      0
(gdb) x/s 0x615050
0x615050:       "Crackers Don't Matter"
(gdb) quit
```

Capture the Flag!


```
$ GUESSME="Crackers D[illegible]er" ./ctf show_me_the_flag
checking 'show_me_the_flag'
ok
flag = 84b34c124b2ba5[illegible]
```

Summary

- Although the program is somewhat artificial and simple, it illustrates many [redacted] of binary analysis: file identification, de-obfuscation, static analysis and dynamic analysis.
- Most of the tools are simple but you can combine them to perform powerful binary analyses.
- We will explore more advanced disassembly tools and techniques next.