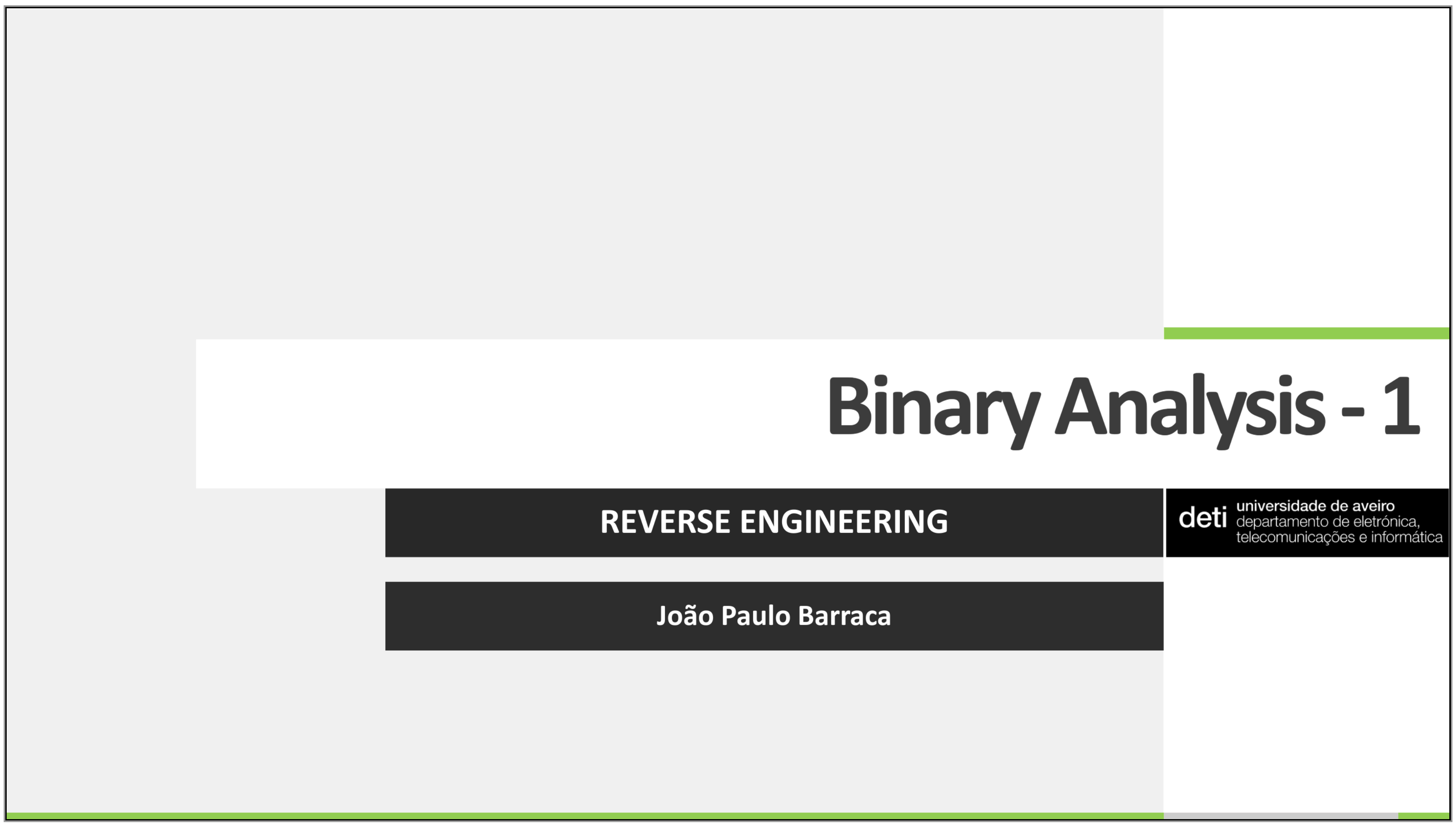
# Binary Executable Files

## Lecture Notes

Analyzing Binary Executable files such as ELF. Focus on structure and Symbol linking.

Previous Next




Download here

## Practical Tasks

### Exercise 1

Compile a small C program with 1-2 function, and using objdump, analyze the output of object created as well as the final binary.

As a possible C program you may consider:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

char* buffer = "Hello World\n";

void bar(void) {
        int fd = open("hello.txt", O_CREAT | O_WRONLY);
        write(fd, buffer, strlen(buffer));
        close(fd);
}

void foo(void){
        printf("%s", buffer);
}

int main(int argc, char **argv){
        foo();
        bar();
        return 0;
}
```

After the program is compiled (`gcc -o prog prog.c`) you can use objdump to inspect it. To obtain the sections you can use `objdump -h prog` and the result should be something like:

```
objdump -h prog

prog:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000000318  0000000000000318  00000318  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.gnu.property 00000020  0000000000000338  0000000000000338  00000338  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .note.gnu.build-id 00000024  0000000000000358  0000000000000358  00000358  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .note.ABI-tag 00000020  000000000000037c  000000000000037c  0000037c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .gnu.hash     00000024  00000000000003a0  00000000000003a0  000003a0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .dynsym       00000108  00000000000003c8  00000000000003c8  000003c8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .dynstr       000000a7  00000000000004d0  00000000000004d0  000004d0  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .gnu.version  00000016  0000000000000578  0000000000000578  00000578  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .gnu.version_r 00000030  0000000000000590  0000000000000590  00000590  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .rela.dyn     000000d8  00000000000005c0  00000000000005c0  000005c0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 .rela.plt     00000078  0000000000000698  0000000000000698  00000698  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 11 .init         00000017  0000000000001000  0000000000001000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .plt          00000060  0000000000001020  0000000000001020  00001020  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .plt.got      00000008  0000000000001080  0000000000001080  00001080  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 14 .text         0000017b  0000000000001090  0000000000001090  00001090  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 15 .fini         00000009  000000000000120c  000000000000120c  0000120c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 16 .rodata       0000001b  0000000000002000  0000000000002000  00002000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 17 .eh_frame_hdr 0000003c  000000000000201c  000000000000201c  0000201c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 18 .eh_frame     000000ec  0000000000002058  0000000000002058  00002058  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 19 .init_array   00000008  0000000000003dd0  0000000000003dd0  00002dd0  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 20 .fini_array   00000008  0000000000003dd8  0000000000003dd8  00002dd8  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 21 .dynamic      000001e0  0000000000003de0  0000000000003de0  00002de0  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 22 .got          00000028  0000000000003fc0  0000000000003fc0  00002fc0  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 23 .got.plt      00000040  0000000000003fe8  0000000000003fe8  00002fe8  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 24 .data         00000018  0000000000004028  0000000000004028  00003028  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 25 .bss          00000008  0000000000004040  0000000000004040  00003040  2**0
                  ALLOC
 26 .comment      0000001e  0000000000000000  0000000000000000  00003040  2**0
                  CONTENTS, READONLY
```

You can also obtain the symbols present in the binary using `objdump -tT prog`, which will show the `symbol table` and the `dynamic symbol table`.

```
$ objdump -tT prog

prog:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*	0000000000000000              Scrt1.o
000000000000037c l     O .note.ABI-tag	0000000000000020              __abi_tag
0000000000000000 l    df *ABS*	0000000000000000              crtstuff.c
00000000000010c0 l     F .text	0000000000000000              deregister_tm_clones
00000000000010f0 l     F .text	0000000000000000              register_tm_clones
0000000000001130 l     F .text	0000000000000000              __do_global_dtors_aux
0000000000004040 l     O .bss	0000000000000001              completed.0
0000000000003dd8 l     O .fini_array	0000000000000000              __do_global_dtors_aux_fini_array_entry
0000000000001170 l     F .text	0000000000000000              frame_dummy
0000000000003dd0 l     O .init_array	0000000000000000              __frame_dummy_init_array_entry
0000000000000000 l    df *ABS*	0000000000000000              prog.c
0000000000000000 l    df *ABS*	0000000000000000              crtstuff.c
0000000000002140 l     O .eh_frame	0000000000000000              __FRAME_END__
0000000000000000 l    df *ABS*	0000000000000000              
0000000000003de0 l     O .dynamic	0000000000000000              _DYNAMIC
000000000000201c l       .eh_frame_hdr	0000000000000000              __GNU_EH_FRAME_HDR
0000000000003fe8 l     O .got.plt	0000000000000000              _GLOBAL_OFFSET_TABLE_
0000000000000000       F *UND*	0000000000000000              __libc_start_main@GLIBC_2.34
0000000000000000  w      *UND*	0000000000000000              _ITM_deregisterTMCloneTable
0000000000004028  w      .data	0000000000000000              data_start
0000000000000000       F *UND*	0000000000000000              write@GLIBC_2.2.5
0000000000004040 g       .data	0000000000000000              _edata
0000000000001179 g     F .text	0000000000000057              bar
000000000000120c g     F .fini	0000000000000000              .hidden _fini
0000000000000000       F *UND*	0000000000000000              strlen@GLIBC_2.2.5
0000000000000000       F *UND*	0000000000000000              printf@GLIBC_2.2.5
0000000000000000       F *UND*	0000000000000000              close@GLIBC_2.2.5
0000000000004038 g     O .data	0000000000000008              buffer
0000000000004028 g       .data	0000000000000000              __data_start
0000000000000000  w      *UND*	0000000000000000              __gmon_start__
0000000000004030 g     O .data	0000000000000000              .hidden __dso_handle
0000000000002000 g     O .rodata	0000000000000004              _IO_stdin_used
00000000000011d0 g     F .text	000000000000001b              foo
0000000000004048 g       .bss	0000000000000000              _end
0000000000001090 g     F .text	0000000000000022              _start
0000000000004040 g       .bss	0000000000000000              __bss_start
00000000000011eb g     F .text	0000000000000020              main
0000000000000000       F *UND*	0000000000000000              open@GLIBC_2.2.5
0000000000004040 g     O .data	0000000000000000              .hidden __TMC_END__
0000000000000000  w      *UND*	0000000000000000              _ITM_registerTMCloneTable
0000000000000000  w    F *UND*	0000000000000000              __cxa_finalize@GLIBC_2.2.5
0000000000001000 g     F .init	0000000000000000              .hidden _init


DYNAMIC SYMBOL TABLE:
0000000000000000      DF *UND*	0000000000000000 (GLIBC_2.34) __libc_start_main
0000000000000000  w   D  *UND*	0000000000000000  Base        _ITM_deregisterTMCloneTable
0000000000000000      DF *UND*	0000000000000000 (GLIBC_2.2.5) write
0000000000000000      DF *UND*	0000000000000000 (GLIBC_2.2.5) strlen
0000000000000000      DF *UND*	0000000000000000 (GLIBC_2.2.5) printf
0000000000000000      DF *UND*	0000000000000000 (GLIBC_2.2.5) close
0000000000000000  w   D  *UND*	0000000000000000  Base        __gmon_start__
0000000000000000      DF *UND*	0000000000000000 (GLIBC_2.2.5) open
0000000000000000  w   D  *UND*	0000000000000000  Base        _ITM_registerTMCloneTable
0000000000000000  w   DF *UND*	0000000000000000 (GLIBC_2.2.5) __cxa_finalize
```

The full contents of this tool are omitted. Run it and determine:

- How many sections are present?
- How many symbols are present?

Strip the binary and repeat the same process with objdump. Then compare both results.

In particular, answer:

- What happened to symbols?
- What happened to function names, and functions?
- What happened to the file size?

## Exercise 2

Compile a small C program with 1-2 functions and external libraries. As an example, you can consider a program that creates a thread (`libpthread`) or compresses a file (`libz`). Any other function is adequate, as long as they are from external libraries.

One example is present at the `zlib` repository: https://raw.githubusercontent.com/madler/zlib/master/test/example.c You can compile this code with `gcc -o example example.c -lz`.

Using a Hex editor, identify the magic values of an `ELF`, and the values of its header. You can use readelf to guide you by presenting the values that you can find in the hex editor.

```
$ readelf -h example

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 (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1270
  Start of program headers:          64 (bytes into file)
  Start of section headers:          20368 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

Using readelf, process the file, and identify the main sections (`readelf -S`) and its content. The following snippet show the content for section 25, which is the `.data` section. It will contain global tables and global variables.

```
$ readelf -x 25 example

Hex dump of section '.data':
  0x00005130 00000000 00000000 38510000 00000000 ........8Q......
  0x00005140 68656c6c 6f2c2068 656c6c6f 21000000 hello, hello!...
  0x00005150 5a310000 00000000                   Z1......
```

Inspect the `.plt` jump table and the `.got` offset table. You can actually disassemble the `.plt` section with `objdump -M intel -d`. The output will show that for each symbol, there is some code to resolve the function. The first entry will be related to the generic code for relocation, while the next entries will contain code specific for each symbol.

```
Disassembly of section .plt:

0000000000001020 <gzclose@plt-0x10>:
    1020:       ff 35 e2 3f 00 00       push   QWORD PTR [rip+0x3fe2]        # 5008 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:       ff 25 e4 3f 00 00       jmp    QWORD PTR [rip+0x3fe4]        # 5010 <_GLOBAL_OFFSET_TABLE_+0x10>
    102c:       0f 1f 40 00             nop    DWORD PTR [rax+0x0]

0000000000001030 <gzclose@plt>:
    1030:       ff 25 e2 3f 00 00       jmp    QWORD PTR [rip+0x3fe2]        # 5018 <gzclose@Base>
    1036:       68 00 00 00 00          push   0x0
    103b:       e9 e0 ff ff ff          jmp    1020 <_init+0x20>

0000000000001040 <free@plt>:
    1040:       ff 25 da 3f 00 00       jmp    QWORD PTR [rip+0x3fda]        # 5020 <free@GLIBC_2.2.5>
    1046:       68 01 00 00 00          push   0x1
    104b:       e9 d0 ff ff ff          jmp    1020 <_init+0x20>
```

In this case, the `.got` will be at `rip+0x3fe2`. If the actual value of the function is found, the instruction pointer will jump to that address. Otherwise, it jumps back to the `.plt`, a value is pushed to the stack (an index), an then the generic resolver is called.

Create a diagram (drawing) of the binary file and represent its structure from the perspective of the ELF structure, a segment view, and a section view. It is important to understand which parts of the ELF file are actually loaded into segments, and where they will be placed in the memory. The structure is important to analyze to see how the bytes map to segments and sections.

## Exercise 3

The LIEF library allows extensive manipulation of binary files, including `ELF` objects. Using LIEF, make a small python script that prints information about an ELF, that may be relevant for future reverse engineering tasks.

In particular, determine:

- The type of file and architecture
- The list of libraries loaded
- The compiler used
- The list of symbols from external libraries
- The address of the program entry point
- Information whether the program is using `RELRO`, `PIE`, and `Canaries`
- Information whether the program is stripped

## Exercise 4

An important feature of dynamic analysis is the interception, redirection, and even modification of symbols. This can be easily achieved using the `LD_PRELOAD` flag for the dynamic linker.

The following snippet allows us to override any function with a custom implementation, and call the original function (or just forbid its execution). In this situation we will use `LD_PRELOAD=libover.so prog`, where `libover` will contain this code, while the `prog` is a standard program under analysis.

```
void (*original_foo)(void) = NULL;

void foo() { // Function to override
    if (original_foo == NULL) { // First time execution: load the real address
      original_foo = dlsym(RTLD_NEXT, "foo");
    }

    printf("foo entry\n");
    original_foo();  // call original function.
    printf("foo exit\n");
}
```

To compile it use: `gcc -o libover.so -shared -fPIC libover.c -dl`.

Taking this as an example, write a library to intercept communications with secure sockets, printing the contents before they are encrypted. Test the library with an application such as `wget`. For `wget`, you can dump the list of dynamic symbols using objdump to look for potential symbols to override.

```
$ objdump -T /usr/bin/wget |grep gnutls
...
0000000000000000      DF *UND*  0000000000000000 (GNUTLS_3_4) gnutls_record_recv
...
0000000000000000      DF *UND*  0000000000000000 (GNUTLS_3_4) gnutls_certificate_verify_peers2
...
0000000000000000      DF *UND*  0000000000000000 (GNUTLS_3_4) gnutls_record_send
```

Three symbols are interesting as they may allow to bypass certificate validation, inspect data sent or data received.

## Tools and links

- objdump: https://man7.org/linux/man-pages/man1/objdump.1.html
- readelf: https://man7.org/linux/man-pages/man1/readelf.1.html
- LIEF: https://lief-project.github.io/
- gnutls:https://gnutls.org/documentation.html
- HxD: https://mh-nexus.de/en/hxd/
- bvi: http://bvi.sourceforge.net/
- ImHex: https://github.com/WerWolv/ImHex
- HexWorkshop: http://www.hexworkshop.com/
- ghex: https://wiki.gnome.org/Apps/Ghex
- HexEdit: https://hexed.it/
- FileInsight: https://github.com/nmantani/FileInsight-plugins

PREVIOUS
Android Dynamic Anaylsis

NEXT
Static Analysis of Applications

Last updated on 15 Mar 2024

(c) 2024 Me. This work is licensed under {license}

Published with Hugo Blox Builder — the free, open source website builder that empowers creators.