Ethical Hacking and Pentesting (COM3031) - SEMR 2024/5 Coursework: Part B Report

Pascal Duncan Lek Hou U
[redacted]

April 10, 2025

Contents

1 Introduction

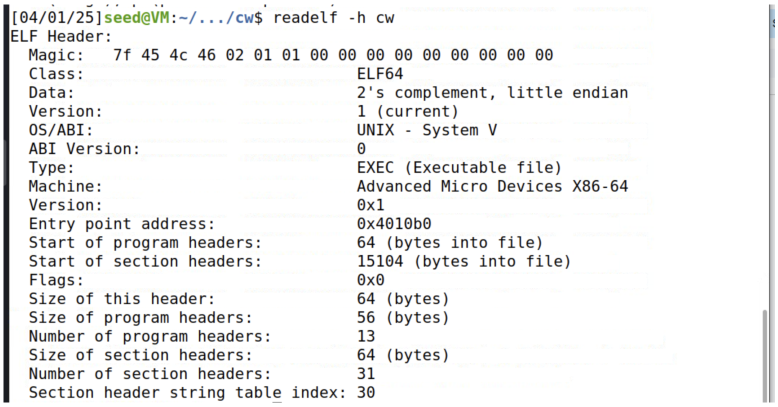
The C Program chosen for analysis can be found in the appendix of this document. The file is named 'cw.c' and was compiled into a binary as 'cw' using `gcc -o cw cw.c -no-pie`.

The structure of the rest of this report is as follows, Section 2 analyses the ELF File's Structure, dissecting contents of the ELF header, particular ELF sections and how data is populated within them. Then, the segments that make up the ELF file and how they differ from sections. Section 3 investigates the content of the symbol table, the functions of the GOT and PLT, and briefly dissect the disassembly of the PLT. Finally, Section 4 discusses shared library dependencies as shown in Figure 9, and some insights from hexdumps to be gained of the raw ELF file.

2 The ELF File Structure

2.1 ELF Header

Running `readelf -h cw`, we can inspect the header and see that 'cw' is an executable file indicated by `EXEC` and has target architecture `Advanced Micro Devices X86-64`. The entry point address is `0x4010b0` (Figure 1).

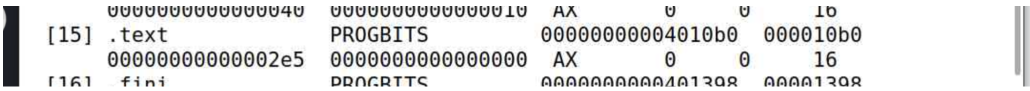


```
[04/01/25]seed@VM:~/.../cw$ readelf -h cw
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4010b0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          15104 (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
```

Figure 1: ELF Header

2.2 ELF Sections

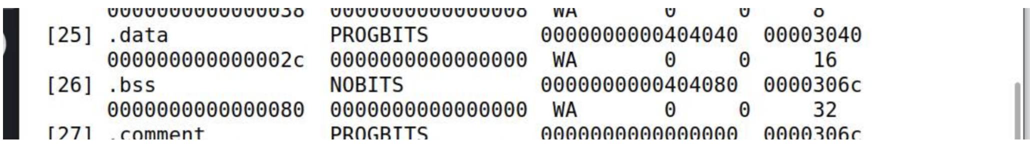
Using `readelf -S cw` shows the section headers contained in the binary. The `.text` section contains the executable instructions for functions in the program, and is marked as executable at runtime indicated with the `X` flag. This would include the functions `main()` (which is the program's entry point after runtime setup), `greet()`, `vulnerableFunction()`, and `printSecretKey()` (which are declared at lines 31, 13, 18, 26 in A). It is addressed at `0x4010b0`, has been allocated `0x2e5` bytes in the binary (Figure 2).



```
       0000000000000040  0000000000000010  AX       0     0     16
  [15] .text             PROGBITS         00000000004010b0  000010b0
       00000000000002e5  0000000000000000  AX       0     0     16
  [16] .fini             PROGBITS         0000000000401398  00001398
```

Figure 2: .text section

`.data` and `.bss` contain the initialised and uninitialised global and static variables respectively (Figure 3). From the section header we can see that `.data` has type `PROGBITS` meaning it holds actual bytes in the file. Whereas `.bss` has type `NOBITS`, so the loader just reserves the space in memory, and does not store any bytes in the file.



```
       0000000000000038  0000000000000008  WA       0     0     8
  [25] .data             PROGBITS         0000000000404040  00003040
       000000000000002c  0000000000000000  WA       0     0     16
  [26] .bss              NOBITS           0000000000404080  0000306c
       0000000000000080  0000000000000000  WA       0     0     32
  [27] .comment          PROGBITS         0000000000000000  0000306c
```

Figure 3: .data and .bss sections

We can see in these two sections with `objdump -s -j <section> -d cw` how data is populated differently. For `.data` (Figure 4), we can extract the contents of the globalMessage and secretKey from the output. Where the secretKey `ad de ef be` is the little-endian form of `0xDEADBEEF` (declared in line 6 of A). However `.bss` (Figure 5) will have allocated space for `uninitialisedArray` and `uninitializedInt` but is unpopulated.

```
[04/01/25]seed@VM:~/.../cw$ objdump -s -j .data -d cw

cw:     file format elf64-x86-64

Contents of section .data:
 404040 00000000 00000000 00000000 00000000  ................
 404050 476c6f62 616c206d 65737361 67652068  Global message h
 404060 6572652e 00000000 efbeadde           ere.........

Disassembly of section .data:

0000000000404040 <__data_start>:
        ...

0000000000404048 <__dso_handle>:
        ...

0000000000404050 <globalMessage>:
  404050:       47 6c 6f 62 61 6c 20 6d 65 73 73 61 67 65 20 68     Global messa
ge h
  404060:       65 72 65 2e 00 00 00 00                             ere.....

0000000000404068 <secretKey>:
  404068:       ef be ad de                                         ....
```

Figure 4: Populated .data

```
[04/01/25]seed@VM:~/.../cw$ objdump -s -j .bss -d cw

cw:     file format elf64-x86-64


Disassembly of section .bss:

0000000000404080 <completed.8061>:
        ...

00000000004040a0 <uninitializedInt>:
        ...

00000000004040c0 <uninitializedArray>:
```

Figure 5: Unpopulated .bss

2.3 ELF Segments

Sections divide the ELF file into logically distinct parts such that each section has a specific purpose and different attributes. Segments are comprised of one or more sections that share similar memory protections or runtime requirements, these are also commonly known as Program Headers as referred in Figure 6.

```
[04/01/25]seed@VM:~/.../cw$ readelf -l cw

Elf file type is EXEC (Executable file)
Entry point 0x4010b0
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000400318 0x0000000000400318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000005a0 0x00000000000005a0  R      0x1000
  LOAD           0x0000000000001000 0x0000000000401000 0x0000000000401000
                 0x00000000000003a5 0x00000000000003a5  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000402000 0x0000000000402000
                 0x0000000000000258 0x0000000000000258  R      0x1000
  LOAD           0x0000000000002e10 0x0000000000403e10 0x0000000000403e10
                 0x000000000000025c 0x00000000000002f0  RW     0x1000
  DYNAMIC        0x0000000000002e20 0x0000000000403e20 0x0000000000403e20
                 0x00000000000001d0 0x00000000000001d0  RW     0x8
  NOTE           0x0000000000000338 0x0000000000400338 0x0000000000400338
                 0x0000000000000020 0x0000000000000020  R      0x8
  NOTE           0x0000000000000358 0x0000000000400358 0x0000000000400358
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_PROPERTY   0x0000000000000338 0x0000000000400338 0x0000000000400338
                 0x0000000000000020 0x0000000000000020  R      0x8
  GNU_EH_FRAME   0x0000000000002098 0x0000000000402098 0x0000000000402098
                 0x000000000000005c 0x000000000000005c  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000002e10 0x0000000000403e10 0x0000000000403e10
                 0x00000000000001f0 0x00000000000001f0  R      0x1
```

Figure 6: ELF segments

As seen in Figure 6 there are 4 `LOAD` headers, these indicate `PT_LOAD` segments that tell the operating system how and where to load portions of the file into memory. We can see the mapping of sections to segments in Figure 7. Starting from segment `02`, this segment contains read-only data structures needed for dynamic linking and runtime metadata. Segment `03` is mapped as read/executable (`R E`) because `.text` and `.plt` contain executable instructions. Segment `05` contains sections such as `.got`, `.data`, `.bss` and are mapped as read/write (`RW`) because `.data` and `.bss` require mutability.

```
Section to Segment mapping:
 Segment Sections...
  00
  01     .interp
  02     .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt
  03     .init .plt .plt.sec .text .fini
  04     .rodata .eh_frame_hdr .eh_frame
  05     .init_array .fini_array .dynamic .got .got.plt .data .bss
  06     .dynamic
  07     .note.gnu.property
  08     .note.gnu.build-id .note.ABI-tag
  09     .note.gnu.property
  10     .eh_frame_hdr
  11
  12     .init_array .fini_array .dynamic .got
```

Figure 7: ELF sections mapped to segments

3 Symbol Table, GOT, PLT

3.1 Symbol Table

The command `nm cw` produces the output in Figure 8 showing the symbol table (containing functions, global variables, etc.) for the given ELF file `cw`.

User defined functions and variables appear in the symbol table along with runtime and system level symbols. e.g.; `main` is located at `0x401255` and listed with `T` to indicate it is defined in the `.text` section. `secretKey` is initialized in `.data` as indicated by `D` and is located at `0x404068`. The uppercasing of the symbol indicates their `global` status and lowercased symbols generally indicated `local` or `non-global` symbols of the same sections as their uppercased counterparts. e.g.; `__GLOBAL_OFFSET_TABLE__` is a local symbol in `.got` which is supported by the indication `d`.

3.2 Global Offset Table (GOT)

The GOT is a lookup table in the ELF binary that holds addresses of variables and functions. For library functions to be stored in the ELF, they must be placed in `.got.plt` which get called via the PLT stubs. When running the program, the code accesses external symbols through entries in the GOT, instead of relying on fixed addresses in the instructions because of unpredictable base addresses.

The functions that are dynamically linked will not be addressed during the linking phase and only resolved when the binary is loaded into memory to be executed. They are identified in the symbol table in Figure 8 with `U` for undefined, and listed here in order of appearance;

- `gets` - used in `vulnerableFunction()` invoked on line 21 in A.
- `__libc_start_main` - sets up the runtime environment and passes control over to the main function (line 31 in A).
- `printf` - is invoked on lines 14, 20, 22, 27, 28, 43 and 44 in A.
- `puts` - included by default.
- `__stack_chk_fail` - included by default.

All of which are part of the GNU C Library loaded via `libc.so.6`.

```
[04/01/25]seed@VM:~/.../cw$ nm cw
000000000040406c B __bss_start
0000000000404080 b completed.8061
0000000000404040 D __data_start
0000000000404040 W data_start
00000000004010f0 t deregister_tm_clones
00000000004010e0 T _dl_relocate_static_pie
0000000000401160 t __do_global_dtors_aux
0000000000403e18 d __do_global_dtors_aux_fini_array_entry
0000000000404048 D __dso_handle
0000000000403e20 d _DYNAMIC
000000000040406c D _edata
0000000000404100 B _end
0000000000401398 T _fini
0000000000401190 t frame_dummy
0000000000403e10 d __frame_dummy_init_array_entry
0000000000402254 r __FRAME_END__
                 U gets@@GLIBC_2.2.5
0000000000404050 D globalMessage
0000000000404000 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
0000000000402098 r __GNU_EH_FRAME_HDR
0000000000401196 T greet
0000000000401000 T _init
0000000000403e18 d __init_array_end
0000000000403e10 d __init_array_start
0000000000402000 R _IO_stdin_used
0000000000401390 T __libc_csu_fini
0000000000401320 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
0000000000401255 T main
                 U printf@@GLIBC_2.2.5
0000000000401219 T printSecretKey
                 U puts@@GLIBC_2.2.5
0000000000401120 t register_tm_clones
0000000000404068 D secretKey
                 U __stack_chk_fail@@GLIBC_2.4
00000000004010b0 T _start
0000000000404070 D __TMC_END__
00000000004040c0 B uninitializedArray
00000000004040a0 B uninitializedInt
00000000004011ad T vulnerableFunction
```

Figure 8: Symbol table

3.3 Procedure Linkage Table (PLT)

These 'undefined' symbols will have an entry in `.plt` and corresponding entries in `.got.plt` so when the binary makes a call to a library function, it goes through the PLT stub which pushes an identifier on the stack, jumps to the "resolver" logic and eventually calls the dynamic linker. The dynamic linker will consult the relocation table to see which function index was pushed and writes the function address (which may be found in shared libraries) (Figure 9) into the GOT entry associated with that function. This is only done once upon first call, every subsequent call goes directly to that address via the PLT stubs, skipping the resolver. This is referred to as lazy binding.

```
[04/03/25]seed@VM:~/.../cw$ ldd cw
        linux-vdso.so.1 (0x00007fffae772000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f7d0fa9a000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f7d0fca1000)
```

Figure 9: Shared Library Dependencies

In Figure 10, the disassembly of the `.plt` allows us to view each stub in assembly code. Starting from `0x401020`, `pushq` places an index on the stack identifying which function is being called. Next, `bnd jmpq` goes to the resolver if the function address is not yet filled in, else it directly jumps to the function. Below, each subsequent 16-byte chunk is another stub for each different function.

```
[04/03/25]seed@VM:~/.../cw$ objdump -d -j .plt cw

cw:     file format elf64-x86-64


Disassembly of section .plt:

0000000000401020 <.plt>:
  401020:       ff 35 e2 2f 00 00       pushq  0x2fe2(%rip)        # 404008 <_GLOBAL_OFFSET_TABLE_+0x8>
  401026:       f2 ff 25 e3 2f 00 00    bnd jmpq *0x2fe3(%rip)        # 404010 <_GLOBAL_OFFSET_TABLE_+0x10>
  40102d:       0f 1f 00                nopl   (%rax)
  401030:       f3 0f 1e fa             endbr64
  401034:       68 00 00 00 00          pushq  $0x0
  401039:       f2 e9 e1 ff ff ff       bnd jmpq 401020 <.plt>
  40103f:       90                      nop
  401040:       f3 0f 1e fa             endbr64
  401044:       68 01 00 00 00          pushq  $0x1
  401049:       f2 e9 d1 ff ff ff       bnd jmpq 401020 <.plt>
  40104f:       90                      nop
  401050:       f3 0f 1e fa             endbr64
  401054:       68 02 00 00 00          pushq  $0x2
  401059:       f2 e9 c1 ff ff ff       bnd jmpq 401020 <.plt>
  40105f:       90                      nop
  401060:       f3 0f 1e fa             endbr64
  401064:       68 03 00 00 00          pushq  $0x3
  401069:       f2 e9 b1 ff ff ff       bnd jmpq 401020 <.plt>
  40106f:       90                      nop
```

Figure 10: Disassembly of PLT

4 Additional Analysis

The binary depends on 3 shared libraries as shown above in Figure 9. We know that all the symbols (in the symbol table in Figure 8) indicated by `U` belong to `libc.so.6`. The other shared libraries still need to be included for other reasons. `ld-linux-x86-64.so.2` must be included as it is the dynamic linker used by the PLT. `linux-vdso.so.1` is a "virtual dynamic shared object", its main purpose is to speed up certain system calls, this is usually included by default and does not correspond to an actual file on disk.

Running `xxd cw` returns the raw hex dump of the ELF binary, though it is more convenient to use tools such as `readelf` and `objdump`, `xxd` may reveal more about the underlying bytes of the file. The most obvious being the ability to see ASCII strings in plaintext, e.g.; "Global message here." declared in line 5 of A is seen at offset `0x3050` (as shown in Figure 11) which we can confirm is located within `.data` from the `readelf` in Figure 3.

```
00003020: 4010 4000 0000 0000 5010 4000 0000 0000  @.@.....P.@.....
00003030: 6010 4000 0000 0000 0000 0000 0000 0000  `.@.............
00003040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00003050: 476c 6f62 616c 206d 6573 7361 6765 2068  Global message h
00003060: 6572 652e 0000 0000 efbe adde 4743 433a  ere.........GCC:
00003070: 2028 5562 756e 7475 2039 2e34 2e30 2d31   (Ubuntu 9.4.0-1
00003080: 7562 756e 7475 317e 3230 2e30 342e 3229  ubuntu1~20.04.2)
00003090: 2039 2e34 2e30 0000 0000 0000 0000 0000   9.4.0..........
```

Figure 11: snapshot of hexdump output of .data

As a lower-level check, we can also confirm that the file is indeed in ELF format by the appearance of the 'magic numbers' `7f 45 4c 46` being the first bytes that appear (as shown in Figure 12). If the file were in another format, there may not be a guarantee that there will be other tools that can analyse the file with as much ease that `readelf` and `objdump` can. A hexdump may be the only method of analysis and can be crucial in identifying hidden ASCII text.

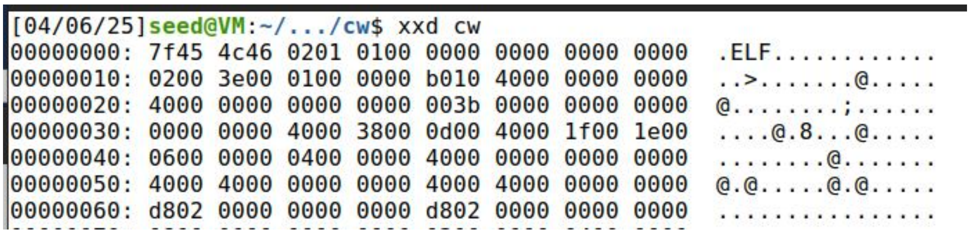


Figure 12: snapshot of hexdump ELF header

Appendices

A Chosen C Program

```c
#include <stdio.h>
#include <string.h>

/* Global variables (in .data since they are initialized) */
char globalMessage[] = "Global message here.";
int secretKey = 0xDEADBEEF;

/* Uninitialized global variables (will be placed in .bss) */
char uninitializedArray[64];
int uninitializedInt;

/* Simple function to demonstrate function pointers */
void greet() {
    printf("Hello from greet()!\n");
}

/* Vulnerable function: potential buffer overflow with gets */
void vulnerableFunction() {
    char buffer[16];
    printf("Enter a string: ");
    gets(buffer); /* Unsafe, used only for demonstration */
    printf("You entered: %s\n", buffer);
}

/* Another function that references the global variables */
void printSecretKey() {
    printf("The secret key is: 0x%X\n", secretKey);
    printf("Global message: %s\n", globalMessage);
}

int main() {
    /* Function pointer demonstration */
    void (*funcPtr)() = greet;
    funcPtr();

    /* Invoke vulnerable function */
    vulnerableFunction();

    /* Demonstrate usage of the .bss variables */
```

```
    strcpy(uninitializedArray, "Populated at runtime (in .bss)");
    uninitializedInt = 42;

    printf("uninitializedArray: %s\n", uninitializedArray);
    printf("uninitializedInt: %d\n", uninitializedInt);

    /* Print secret key and global message */
    printSecretKey();

    return 0;
}
```