CS451 – Software Analysis

Lecture 1

Introduction to Binary Code

Elias Athanasopoulos
athanasopoulos.elias@ucy.ac.cy

Software

- Software is developed using high-level programming systems and languages
  - C/C++, Rust, Go, Java, Ruby, Python, etc.
- Some of those systems compile their programs to machine code
  - C/C++, Rust, Go, etc.
- Machine code is assembled to binaries that can be executed on a specific CPU

Bytecode vs machine code

- Not all programming systems produce **machine code**
  - Java produces bytecode that executes on the Java Virtual Machine (JVM)
  - Ruby interprets code at run-time
- Bytecode and interpreted code do not directly execute on the physical CPU
- Bytecode forms a different type of binary code
  - Not really touched in this course

Creating binaries

- Binaries are created by a compiler
- Compiling the source code (e.g., C/C++) and assembling all code in an executable binary involves several steps
  - Compiling source code to object files (machine code)
  - Linking object files to an executable binary
- Binaries have dependencies
  - They use shared libraries (also, binaries)

Our first binary

```c
#include <stdio.h>

int ga = 42;

void foo(void) {
        fprintf(stderr, "The value of the global variable is: %d.\n", ga);
}


int main(int argc, char *argv[]) {

        foo();

        return 1;
}
```

Save the file using the name `first.c` and compile it:

```
$ gcc -Wall first.c -o first
```

Notice the compilation: it is a single line (but it’s not actually). Try to compile using the option `-v`.

Is it a binary?

$ file ./first

./first: **ELF 64-bit LSB executable, x86-64**, version 1 (SYSV), **dynamically linked**, interpreter /**lib64/ld-linux-x86-64.so.2**, for GNU/Linux 3.2.0, BuildID[sha1]=9467f7dbca2046a4e68629d683640f165d0a301e, **not stripped**

A better look

```
$ readelf -h ./first
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:               0x4004f0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          15712 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

Linking

- Binaries can be dynamically or statically linked
- Dynamic linking
  - Code can be reused by linking to shared libraries
  - Code reuse
  - Slower code
- Static linking
  - All code, even the one that is not used at all, is contained in a single binary
  - Code duplication
  - Faster code

Dynamic vs static linking

```
# Dynamic linking (default)
$ gcc -Wall first.c -o first
$ ls -lh ./first
-rwxrwxr-x. 1 elathan elathan 18K Jan 10 11:55 ./first

# Static linking (with -static)
$ gcc -Wall -static first.c -o first
$ ls -lh first
-rwxrwxr-x. 1 elathan elathan 1.6M Jan 10 11:53 first
```

For CentOS (by default static libc is not in the system):

```
$ sudo dnf config-manager --enable powertools
$ sudo yum install glibc-static
```

Inspect shared dependencies

```
# ldd (Load Dynamic Dependencies)
$ ldd -v ./first
    linux-vdso.so.1 (0x00007ffcc6b31000)
    libc.so.6 => /lib64/libc.so.6 (0x00007ffa2480e000)
    /lib64/ld-linux-x86-64.so.2 (0x00007ffa24bd3000)

Version information:
./first:
    libc.so.6 (GLIBC_2.2.5) => /lib64/libc.so.6
/lib64/libc.so.6:
    ld-linux-x86-64.so.2 (GLIBC_2.3) => /lib64/ld-linux-x86-64.so.2
    ld-linux-x86-64.so.2 (GLIBC_PRIVATE) => /lib64/ld-linux-x86-
64.so.2
```

What are these dependencies?

- `libc` is the standard C library
  - `printf`, `malloc()`, etc.
- VDSO (virtual dynamic shared object) is a system that allows the kernel to speed up system calls
- `ld-linux.so` is the ELF interpreter (or loader)

Debugging symbols

- Binaries may contain a lot of information useful for debugging them
- A binary that contains debugging symbols is **not stripped**
  - Various levels of the amount of debugging information that will be embedded
  - Debugging levels can be specified using –g in gcc
- Binaries can be stripped at any time

Stripped vs not stripped

```
$ file ./first
./first: ELF 64-bit LSB executable, x86-64, version 1
(SYSV), dynamically linked, interpreter /lib64/ld-linux-
x86-64.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=9467f7dbca2046a4e68629d683640f165d0a301e,
not stripped
$ ls -lh first
-rwxrwxr-x. 1 elathan elathan 18K Jan 10 11:55 first
$ strip first
$ file ./first
./first: ELF 64-bit LSB executable, x86-64, version 1
(SYSV), dynamically linked, interpreter /lib64/ld-linux-
x86-64.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=9467f7dbca2046a4e68629d683640f165d0a301e,
stripped
$ ls -lh first
-rwxrwxr-x. 1 elathan elathan 6.8K Jan 10 12:37 first
```

Debugging levels

```
% gcc -Wall -g0 first.c -o first.g0
% gcc -Wall -g1 first.c -o first.g1
% gcc -Wall -g2 first.c -o first.g2
% gcc -Wall -g3 first.c -o first.g3
% gcc -Wall first.c -o first
% strip first -o first.str
% ls -lh
-rwxr-xr-x 1 elathan elathan 17K Jan 25 01:40 first
-rwxr-xr-x 1 elathan elathan 17K Jan 25 01:40 first.g0
-rwxr-xr-x 1 elathan elathan 18K Jan 25 01:40 first.g1
-rwxr-xr-x 1 elathan elathan 19K Jan 25 01:40 first.g2
-rwxr-xr-x 1 elathan elathan 41K Jan 25 01:40 first.g3
-rwxr-xr-x 1 elathan elathan 15K Jan 25 01:41 first.str
```

Levels explained (for gcc)

- Level 0
  - Produces no debug information at all
- Level 1
  - Produces minimal information, enough for making backtraces in parts of the program that you don’t plan to debug
  - This includes descriptions of functions and external variables, and line number tables, but no information about local variables.
- Level 2
  - Includes additional information for local variables
- Level 3
  - Includes extra information, such as all the macro definitions present in the program

Binaries have sections

```
$ readelf -SW ./first
There are 30 section headers, starting at offset 0x3d60:


Section Headers:
  [Nr] Name              Type     Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL     0000000000000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS 0000000000400238 000238 00001c 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE     0000000000400254 000254 000020 00   A  0   0  4
 ...
```

`objdump -h` can be also used to display sections' information.

Let’s have a closer look

- Code is located at the `.text` section

```
$ readelf -SW ./first | grep .text
  [Nr] Name          Type       Address            Off     Size    ES Flg Lk Inf Al
  [13] .text         PROGBITS   00000000004004f0   0004f0  0001a5  00  AX  0   0 16
```

- Notice the starting address of the .text section
  - **`0x04004f0`**

Sections contain various data

- A program has many different *objects*
  - Functions, data, other sections, etc.
- These objects have usually names
  - We refer to their names using *symbols*
- These symbols are not needed by the executing code
  - They are helpful for debugging and analysis, as well as for our initial understanding

Inspecting symbols

```
$ readelf --syms ./first
Symbol table '.symtab' contains 105 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000400238     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000400254     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000400274     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000400298     0 SECTION LOCAL  DEFAULT    4
...
```

- All symbols are stored in a specific section `.symtab` (Symbol Table)

`objdump -t` and `nm` can be also used to display symbols' information.

Let’s have a closer look

```
$ readelf --syms ./first | grep main
   Num:    Value          Size Type    Bind   Vis      Ndx Name
   100: 00000000004005fc    27 FUNC    GLOBAL DEFAULT   13 main

$ readelf --syms ./first | grep foo
   Num:    Value          Size Type    Bind   Vis      Ndx Name
    95: 00000000004005d6    38 FUNC    GLOBAL DEFAULT   13 foo
```

- Recall the location of .text: **`0x04004f0`**

Another example

```
$ readelf --syms ./first | grep ga
   Num:    Value          Size Type    Bind   Vis      Ndx Name
    86: 0000000000601024     4 OBJECT  GLOBAL DEFAULT   23 ga
```

- This object should be located at the `.data` section

```
$ readelf -SW ./first | grep .data
  [Nr] Name       Type      Address          Off    Size   ES Flg Lk Inf Al
  [15] .rodata PROGBITS 00000000004006a8 0006a8 00003a 00   A  0   0  8
  [23] .data   PROGBITS 0000000000601020 001020 000008 00  WA  0   0  4
```

Let’s modify the program a bit

```
#include <stdio.h>

int ga = 42;
int g_uninit_var;

void foo(void) {
        fprintf(stderr, "The value of the global variable is: %d.\n", ga);
}


int main(int argc, char *argv[]) {

        foo();

        return 1;
}
```

Save the file using the name `bss.c` and compile it:

```
$ gcc -Wall bss.c -o bss
```

Let’s find the new symbol

```
$ readelf --syms ./bss | grep g_uninit_var
    88: 000000000060104c 4 OBJECT  GLOBAL DEFAULT 24 g_uninit_var
```

- Let’s now find the section that hosts this symbol

```
$ readelf -SW ./bss | grep bss
[Nr] Name         Type       Address          Off    Size   ES   Flg Lk Inf Al
[24] .bss         NOBITS 0000000000601040 001028 000010 00   WA  0    0 32
```

HW: Can you add an uninitialized array in `bss.c` and see how `.bss` is changing? Can you find the new symbol?

Homework

- Use readelf and objdump to display the sections of toy programs that you have created
- Use readelf, objdump, and nm to display the symbols of toy programs that you have created
- In the example program, add a global uninitialized array, a global initialized array, a global constant (use `const`), i.e., read-only, array and try to locate the section each symbol is stored
  - Try again with stripped binaries