README

RE Lab 0x03 - Static Analysis

updated by S. Radu - march 2025
based on previous work by R. Caragea and C. Rusu

Setup

Get your lab files from moodle: 03-lab-files.zip. The password for the zip is `infected`.

As you will recall from the previous session, we learned what the process of going from code to assembly looks like. In this lab we will learn how to go the other way around: from the compiled binary file, back to source code. We'll take a look at the structure of ELF binary files and the tools we can use in order to analyse them.

You can use both Windows and Linux interchangeably for this lab, but Linux will probably be more convenient.

Tools

You won't need all of the tools that I'll mention here, but it's nice to have options right?

Super basic, cli tools

- readelf
- objdump (you have already played with this one in the previous labs)

You can always learn more about them by reading the corresponding man pages.

Advanced

- IDA (Pro?) [1]
  - x86/x86_64 only / cloud decompilation only (free)
  - full RE framework with tons of features (very expensive)
- Binary Ninja [2]
  - modern and really promising project (75$ for students, among others)
- Ghidra [3]
  - NSA-developed, FOSS IDA Pro competitor with huge community support and the ugliest UI on the planet (free)
- Cutter/Rizin [4]
  - minimalist RE tool, built on top of Rizin, a very powerful CLI RE suite of tools. (free)

ELF structure

As described in the course, Linux executables follow the ELF structure. If we want to reverse engineer Linux executables, we must learn some basics about this structure.

Firstly, ELF files can have dual use: as an object file by compilers or as runnable files by the OS interpreter. This is the reason why ELF programs have both segments, also known as program headers and sections. Segments

To investigate program headers on Linux we can use the `readelf` utility in the `binutils` package. Most of the time, the target binary is compiled by gcc or clang. In these cases, the structure (internal organization) of output binaries is basically the same. Let us now look at the structure of a classic “Hello, World” program.

If you are interested in learning more about the ELF file format, or file formats in general, your can check out corkami and [5] [6] [7].

Program Headers

```
$ readelf --program-headers hello-world
```

```
Elf file type is DYN (Shared object file)
Entry point 0x580
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000001f8 0x00000000000001f8  R E    0x8
  INTERP         0x0000000000000238 0x0000000000000238 0x0000000000000238
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x000000000000089c 0x000000000000089c  R E    0x200000
  LOAD           0x0000000000000dd8 0x0000000000200dd8 0x0000000000200dd8
                 0x0000000000000258 0x0000000000000260  RW     0x200000
  DYNAMIC        0x0000000000000df0 0x0000000000200df0 0x0000000000200df0
                 0x00000000000001e0 0x00000000000001e0  RW     0x8
  NOTE           0x0000000000000254 0x0000000000000254 0x0000000000000254
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME   0x0000000000000754 0x0000000000000754 0x0000000000000754
                 0x000000000000003c 0x000000000000003c  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000dd8 0x0000000000200dd8 0x0000000000200dd8
                 0x0000000000000228 0x0000000000000228  R      0x1

 Section to Segment mapping:
  Segment Sections...
```

```
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.ve
   03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .jcr .dynamic .got
```

Sections

```
$ readelf --section-headers  hello-world
```

```
There are 31 section headers, starting at offset 0x1a00:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000238  00000238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             0000000000000254  00000254
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.gnu.build-i NOTE             0000000000000274  00000274
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         0000000000000298  00000298
       000000000000001c  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           00000000000002b8  000002b8
       00000000000000c0  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000000378  00000378
```

```
       0000000000000096  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           000000000000040e  0000040e
       0000000000000010  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          0000000000000420  00000420
       0000000000000020  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA             0000000000000440  00000440
       00000000000000d8  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             0000000000000518  00000518
       0000000000000018  0000000000000018  AI       5    24     8
  [11] .init             PROGBITS         0000000000000530  00000530
       0000000000000017  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         0000000000000550  00000550
       0000000000000020  0000000000000010  AX       0     0     16
  [13] .plt.got          PROGBITS         0000000000000570  00000570
       0000000000000008  0000000000000000  AX       0     0     8
  [14] .text             PROGBITS         0000000000000580  00000580
       00000000000001b1  0000000000000000  AX       0     0     16
  [15] .fini             PROGBITS         0000000000000734  00000734
       0000000000000009  0000000000000000  AX       0     0     4
  [16] .rodata           PROGBITS         0000000000000740  00000740
       0000000000000012  0000000000000000   A       0     0     4
  [17] .eh_frame_hdr     PROGBITS         0000000000000754  00000754
       000000000000003c  0000000000000000   A       0     0     4
  [18] .eh_frame         PROGBITS         0000000000000790  00000790
       000000000000010c  0000000000000000   A       0     0     8
  [19] .init_array       INIT_ARRAY       0000000000200dd8  00000dd8
       0000000000000008  0000000000000008  WA       0     0     8
  [20] .fini_array       FINI_ARRAY       0000000000200de0  00000de0
       0000000000000008  0000000000000008  WA       0     0     8
  [21] .jcr              PROGBITS         0000000000200de8  00000de8
       0000000000000008  0000000000000000  WA       0     0     8
  [22] .dynamic          DYNAMIC          0000000000200df0  00000df0
       00000000000001e0  0000000000000010  WA       6     0     8
  [23] .got              PROGBITS         0000000000200fd0  00000fd0
       0000000000000030  0000000000000008  WA       0     0     8
  [24] .got.plt          PROGBITS         0000000000201000  00001000
```

```
       0000000000000020  0000000000000008  WA       0     0     8
  [25] .data             PROGBITS         0000000000201020  00001020
       0000000000000010  0000000000000000  WA       0     0     8
  [26] .bss              NOBITS           0000000000201030  00001030
       0000000000000008  0000000000000000  WA       0     0     1
  [27] .comment          PROGBITS         0000000000000000  00001030
       000000000000002d  0000000000000001  MS       0     0     1
  [28] .symtab           SYMTAB           0000000000000000  00001060
       0000000000000660  0000000000000018          29    48     8
  [29] .strtab           STRTAB           0000000000000000  000016c0
       000000000000022f  0000000000000000           0     0     1
  [30] .shstrtab         STRTAB           0000000000000000  000018ef
       000000000000010c  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

Interesting sections

For a reverse engineer the following sections are (usually) of interest:

- `.text` - containing the majority of code, both user-written code and boilerplate generated by the compiler
- `.init` - containing code usually generated by the compiler that is supposed to run before main() has been called
- `.fini` - containing code usually generated by the compiler that is supposed to run after main() has been called
- `.plt` - containing code generated by the compiler in order to call functions from libraries
- `.rodata` - containing Read Only Data used by the program (strings, constants, etc)
- `.data` - containing Read/Write Data, used for initialized variables, mutable strings etc

- `.bss` - containing Read/Write Data, used for uninitialized global variables

For an exploit developer the following additional sections are of interest for information leakage/control flow hijacking purposes:

- `.got` / `.got.plt` - (Global Offset Table) containing pointers used in library call resolution
- `.init_array` / `.fini_array` - containing pointers used in the code from the .init/.fini sections

Check the man page of `readelf` and see what other ways you can use it to learn more about your target binary.

Usage of Advanced Tools

Recall the list of advanced tools from the beginning. All of them have the following functionality (*to various degrees*):

- Code Navigation
    - identifying functions and navigating between them
    - various code views (Listing (Assembly), Decompilation, Graph-View, HexView, etc.) and the option of switching between them, while maintaining context.
    - find usages of the current function/variable/item (cross-references/xrefs)
- Renaming/Redeclaring
    - changing the signature of a function
    - changing the type of a variable/item
    - changing the name of a function/variable/item
- Reorganizing the stack variables
    - change a stack variable into something else (smaller, bigger, structure, array); **be mindful of how much space there is available for your desired actions**.

Note that when turning into an array it's ideal to first change the variable into the array unit (e.g. if you want to change a stack space into `int v[30]`, and `v` is currently of type `char`, first turn `v` into an `int`) and then change its type into the desired array. Your tool of choice might give you suggestions regarding the ideal/ maximum array size.

Experiment!

Write a small program in C (or any other compiled language that you like), and open the resulting binary in any of the tools suggested here (or some other that your know of). Learn to navigate your tool, make the output look more pretty, try to work backwards from the binary to the source code.

See how various compiler flags affect the output of your tool (`-g`, `-s`, `0x`, etc).

When in doubt Right Click!, hover, or check the cheatsheet/docs in the resources section

Lab Tasks

For today's tasks you can use any of the *advanced* tools mentioned in the setup section above, **but** in the hints along the way we will assume you are using Ghidra.

Task 1: Reverse engineering with spoilers (5p)

Usually, when reverse engineering, all we have is a binary. Starting from it, we need to reconstruct (mainly through guessing/inferences) what the function names could be, what the variables are used for, what the program does as a whole.

For this task, you have a binary, `task1`, and also its corresponding source code, `task1.c`. Using the stripped binary, you will simulate normal reverse engineering by using the source code (instead of guessing).

Your task is to create a near-original replica of the original source in the IDA interface:

1. Rename and retype the 4 functions in the source code (aside from `main()`) (2p)
2. Rename and retype the stack variables in `setup()` and `main()` (1p)
3. Rename and retype the stack variables (including the arrays) in `chance()` and `gen_rand_string()` (2p)

Task 2: Statically linked crackme - graybox analysis (dynamic + static) (4p)

In this task, you will learn to navigate through functions in a statically linked and stripped `crackme`. (*What is a stripped binary?*)

Since the binary has a whopping 783 functions detected, you do not have the time or motivation to go through all of them. As such, you need to approach the problem in a clever and elegant way:

1. Run the program once, note any strings. Go to the `.rodata` segment and find any/all of the strings. Using the xref (cross-reference) functionality, determine where the `main()` function is. (1p)
2. Rename all the functions in `main()` and determine the password checking function. (1p)
3. In the password checking function, observe how the correct password is generated; we want to make this function more legible. Go to the location of any `DAT_XXX` variable in the Ghidra decompiler view. Notice that each of them is a letter of the alphabet. Find the location of the start of the alphabet and redeclare that address as a wide C string (Right Click->Data->TerminatedUnicode). Again, in the password checking function, observe how the decompiler view looks now. Redeclare the alphabet as constant data (Right Click->Data->Mutability). After changing the data type of the corresponding local variable accordingly, the assignment sequence should collapse and reveal the correct password. Finally check that the password is accepted (2p)

Task 3: Data Structures (6p)

In this task, you will learn to use the *structures* functionality of Ghidra.

Only the simplest programs are written without any sort of data structures in mind. Even basic OOP features are implemented using structures; classes themselves are also compiled as structures. However, after compilation, structure and type information is lost (if we don't have debugging symbols) but we can still observe repeated access patterns and infer what various structures might have looked like (*recall how structure fields are accessed in ASM from the previous lab*).

Look at the code in main and the password checking function, analyze the access patterns and verify that it matches the linked list structure below.

offset	size	mnemonic	data type	field name
0x0	0x4	ddw	dword	field_0_idx
0x4	0x1	db	byte	field_4
0x5	0x1	db	byte	
0x6	0x1	db	byte	
0x7	0x1	db	byte	
0x8	0x8	dq	astruct*	*field_8_next

You can either:

- go to DataTypeManager->task3->Right Click->New->Structure
- go to your variable, Right Click->Auto Create Structure, then Edit Data Type

1. Use the Structure Editor and create the list structure above, and make sure to also declare the last field as a `astruct*` pointer. (2p)

2. In main, cast the buffer returned from `malloc` and the head of the list to this struct type and propagate in the password checking function, renaming and retyping where necessary. (2p)
3. Describe what the code does and figure out the correct password (2p)

Task 4: Bonus (20p)

Check out `./bonus/bonus`. Verify that it work on the remote server **TODO**. No partial points for this one.

You can present solutions for this task until the end of the semester.

Resources

---

1. https://docs.hex-rays.com/9.0/getting-started/basic-usage ↩
2. https://docs.binary.ninja/ ↩
3. https://ghidra-sre.org/CheatSheet.html ↩
4. https://book.rizin.re/ ↩
5. https://raw.githubusercontent.com/corkami/pics/refs/heads/master/binary/elf101/elf101.svg ↩
6. https://media.ccc.de/v/31c3_-5930-en-saal_6-201412291400-funky_file_formats-_ange_albertini ↩
7. https://media.ccc.de/v/38c3-fearsome-file-formats ↩