**RE lab 03 - Static Analysis**

**Lab files and setup**

Download the lab files from [here](https://pwnthybytes.ro/unibuc_re/03-lab-files.zip) Password is infected

**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.

$ readelf --program-headers test

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.version\_r .rela.dyn .rela.plt .init .plt .plt.got .text .fini .rodata .eh\_frame\_hdr .eh\_frame

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 test

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)

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

**IDA cheat sheet:**

**Navigation**

* To go into another function, double-click the function name either in the function sidebar (left pane) or in the IDA-View or Pseudocode view
* Switching between IDA-View and Pseudocode: Press **Tab** to go to the exact assembly-instruction for the current position. Alternatively, press **F5** to see pseudocode without pinpointing the above.
* Switching between Linear-View and Graph-View: Press **Space**
* Press **Esc** to go to the previous view
* To find usages of the current function/variable/item, right-click and choose **Jump to xref..** or press **x**

**Renaming/Redeclaring**

* Changing the signature of a function: Right-Click the signature in the IDA-View or Pseudocode View and click **Set item type**. Keyboard shortcut: **y**
* Changing the type of a variable: Right-Click the variable in the IDA-View or Pseudocode View and click **Set lvar type**.Keyboard shortcut: **y**
* Changing the name of a function/variable: Right-Click the function/variable in the IDA-View or Pseudocode View and click **Rename global/lvar item**. Keyboard shortcut: **n**

**Reorganizing the stack variables**

* To change a stack variable into something else (smaller, bigger, structure, turn into array) first double click on the variable to go into the Stack frame of that specific function. Observe how much space is you have for your desired actions. Right click on the variable and click **Set type**.
* 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 char, first turn v into an int) and then right-click and choose **Array**. It will be possible to now see some suggestions from IDA regarding the ideal/maximum array size.

**When in doubt**

* Right Click! (or hover)

**Task 1: Reverse engineering with spoilers**

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.

* Rename and retype the 4 functions in the source code (aside from main() ) **(3p)**
* Rename and retype the stack variables in **setup()** and **main()** **(1p)**
* 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)**

In this task, you will learn to navigate through functions in a statically linked and stripped crackme.

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:

* Run the program once, note any strings. Go to the **.rodata** segment (**Ctrl-s**) and find any/all of the strings. Using the xref functionality, determine where the **main()** function is. **(1p)**
* Rename all the functions in **main()** and determine the password checking function. **(1p)**
* 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 **word\_…..** variable in IDA-view and find the location of the start of the alphabet and redeclare that address as a wide C string (**Edit->Strings->Unicode**).
* Again, in the password checking function, observe how the right-hand-sides look now. Redeclare the alphabet with the “const” modifier at the beginning. This should collapse the function and reveal the correct password. Finally check that the password is accepted **(2p)**

**Task 3: Data Structures**

In this task, you will learn to use the Structures functionality of IDA.

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.

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.

* Use the **Structures** tab and create the following list structure (also declare field\_8\_next as a **struc\_1\***pointer) **(2p)**

00000000 struc\_1 struc ; (sizeof=0x10, mappedto\_8)

00000000 field\_0\_idx dd ?

00000004 field\_4 db ?

00000005 db ? ; undefined

00000006 db ? ; undefined

00000007 db ? ; undefined

00000008 field\_8\_next dq ? ; offset

00000010 struc\_1 ends

* 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)**
* Describe what the code does and figure out the correct password **(2p)**