Reverse Engineering For Beginners

by Libero Scarcelli (liberoscarcelli.net)

Table of Contents

Introduction To Reverse Engineering

The steps required to reverse engineer an application can be summarized by the following points, in the given order:

- Seeking approval: is reverse engineering the target application legal?
- Using hex and/or text editors to gather basic information by inspecting the application.
- Using binary analyzers in order to find out the Magic Header and any other additional clue.
- Running the target application within a protected environment, such as a virtual machine, while monitoring the network, log files, the Windows registry and anything else able to disclose important details about the target.
- Disassembling the application, which requires specific assembly and hardware knowledge.

Reverse Engineering On Microsoft Windows Servers

If the target application runs on **Microsoft Windows**, the following are some of the registry keys that should be monitored:

- HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
- HKLM\Software\Microsoft\Windows\CurrentVersion\Run HKLM\Software\Microsoft\ Windows\CurrentVersion\RunOnce
- HKLM\Software\Microsoft\Windows\CurrentVersion\RunOnceEx
- HKLM\Software\Microsoft\Windows\CurrentVersion\RunServices
- HKLM\Software\Microsoft\Windows\N\RunServicesOnce
- HKLM\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run
- HKCU\Software\Microsoft\Windows\CurrentVersion\Run
- HKCU\Software\Microsoft\Windows\CurrentVersion\RunOnce
- HKCU\Software\Microsoft\Windows\CurrentVersion\RunOnceEx
- HKLM\Software\Microsoft\Windows\CurrentVersion\RunServices
- HKLM\Software\Microsoft\Windows\CurrentVersion\RunServicesOnce
- HKCU\Software\Microsoft\Windows NT\CurrentVersion\Windows\Run
- HKCU\Software\Microsoft\Windows NT\CurrentVersion\Windows\Load
- HKLM\SYSTEM\ControlSet***\Control\Session Manager\BootExecuteHKLM\ SOFTWARE\Wow6432Node\Windows\CurrentVersion\Run

Reverse Engineering On Linux Servers

For **Linux** applications instead, the reverse engineer will be monitoring the following points, looking for any suspicious or unusual behaviour:

- How the application interacts with the configuration of the **cron** daemon.
- How the application interacts with the **network** and its configuration.
- How the application interacts with the **USB** ports detection process and with the ports themselves.
- How the application interacts with the **X Server**.

- How the application interacts with the **/boot** directory and any booting related file and directory.
- How the application interacts with the **environment variables** such as $PATH, **aliases** and **shell settings**.

Computers And Binary System

As binary numbers can be easily represented as a list of **ON** and **OFF**, computers like to think in terms of binary numbers. However, humans prefer the hexadecimal system instead, as it produces shorter sequences. The following example shows how, an average Intel computer, internally represents positive integers:

$01010111 = (0 * 2^7) + (1 * 2^6) + (0 * 2^5) + (1 * 2^4) + (0 * 2^3) + (1 * 2^2) + (1 * 2^1) + (1 * 2^0) = +87$

When the most significant bit (MSB) is **zero**, the number is **positive**. Viceversa, in **negative** numbers, the MSB is equal to **one**. The following example shows the steps required to translate positive decimal numbers, into negative ones, in a format that an average Intel CPU can understand:

-13 --> 00001101 (get the positive binary)
00001101 --> 11110010 (reverse '1's and '0's)
11110010 --> 11110011 (add '1')
11110011 --> 11110011 (MSB is already '1')

The methodology that has been just shown is called **two's complement**, while other available approaches are the **one's-complement** and the **sign-magnitude**, which are both not covered in this tutorial. However, the reader might need to learn these systems too.

CPU Registers

CPU registers consist of a small amount of **fast storage**, quickly accessible to CPUs. Some registers have specific hardware functions and might be read-only or write-only. Computers load data from larger memories into registers, where arithmetic calculations take place. Results are then usually stored back to main memory, either by the same instruction, or by the next one. Registers generally can hold from 8 to 64 bits or even more. Modern Intel architectures support the following 16-bit General-Purpose Registers (GPR):

- AX is for storage in arithmetic operations.
- CX is the counter register used in shift, rotate and loop instructions.
- DX is mainly for arithmetic and I/O operations, but it is also used for general storage.
- BX is the base register and it is used to point to data.
- SP points to the top of the stack. When data is placed onto stack, SP decrements (or increments, in platforms such as Intel MCS-51) and as the data is retrieved from the stack, SP increments (or decrements, in platforms such as Intel MCS-51). This register should not be modified.
- BP points at the bottom of the stack.
- SI is used to point at sources in string or stream operations.

- DI is used to point to destinations in string or stream operations.

CPU registers names change slightly according to the size, therefore, the 32-bit version of AX is called **E**AX, while the 64-bit version is named **R**AX (for the platforms that support 32/64-bit registers). However, the most significant bytes of the larger registers can be accessed by the smaller ones:

- EAX is the full 32-bit value.
- AX retrieves the lower 16-bits.
- AL retrieves the lower 8 bits.
- AH retrieves the bits from 8 through 15.

Therefore, the programmer can store values into EAX and then only read from AH or AL. Bear in mind that the opposite is not true, as storing values into AH would not correctly inizialize EAX: this would point at random data.

Segmentation was introduced in 1978 on the Intel 8086 to allow addressing of more than 64 KB. Old architectures divided memories into sections: code segment, stack segment, data segment and more, hence, these registers identify the starting location of these chunks. The following is a list of segment registers:

- SS points to the stack.
- CS points to the code.
- DS points to the data.
- ES points to extra data.
- FS points to more extra data.
- GS points to still more extra data.

As the x86-64 architecture has dropped the support for this technology, CS, SS, DS and ES are always set to zero, while the last two can still contain data. Windows uses these registers to store thread related data, while Linux to save per-CPU information.

The **Instruction Pointer** (IP) is the register that contains the memory address where the next assembly instruction to be executed is stored.

FLAGS Registers are used to store the results of operations and the state of the processor. These are 32-bit registers:

- CF is set if an addition or subtraction borrow a bit beyond the size of the register. Next instruction will check this value.
- PF is set if the number of set bits in the least significant byte is even. Used with jump instructions.
- AF is set if borrow generated out of four least significant bits.
- ZF is set if the result of an operation is zero.
- SF is set if the result of an operation is negative.
- TF is set if step by step execution for debugging is needed.

- IF is set if interrupts are enabled.
- DF is set if string operations read memory backwards.
- OF is set if arithmetic overflow happens.IOPL defines the I/O Privilege Level of the current process.
- NT is set if the current process is linked to the next one.
- RF turns some debug exceptions off to allow debugging.
- VM is set to allow real mode applications to run in protected mode (80386 behaving like a faster 8086).
- AC is set if alignment checking of memory references is done.
- VIF is set with VIP to have the operating system servicing deferred interrupts.
- VIP is set if an interrupt is pending.
- ID is set if the platform supports the CPUID instruction.

Endianness

The x86 architecture supports the little-endian encoding, meaning that multi-byte values are written least significant byte first. Therefore, the value **A3**A2A1A0, is represented as A0A1A2**A3**.

Getting Started With NASM And DDD

Assembly language is usually stored into .asm files, which are translated into machine code by an assembler.

The Asm File Format And Its Sections

Although each assembler supports its own .asm format and syntax, all lines of assembly code consist of the following four sections (in the given order):

label **instruction** **operand** **comment**

- **label** or address field is optional and might be followed by a colon. In jump and loop instructions, this field defines the location to jump to. In the .data sections of the .asm file, this field is used to apply labels to memory regions, in order to mimic the behavior of variables of high level languages.
- **instruction** is a mandatory field that defines the instruction to be executed.
- **operand** is an instruction specific field and might contain from zero up to three values, separated by commas.
- **comment** is a self-explanatory and optional field.

Assembler Directives

Assembler directives are part of the assembler syntax, although they are not related to any CPU instruction set as they are specific to the chosen assembler and disassembler instead. Directives always begin with a period. Usually, .asm files contains at least:

- A data section (**.data**) that contains initialised variables and constants
- A BSS section (**.bss**) that defines uninitialised data
- A text section (**.text**) which contains the program body

The reader is now ready to analyze the first assembly program written using NASM:

```
SECTION .text
   GLOBAL _start
_start:
   MOV ECX, myStr          ; Place a comment here
   MOV EDX, MYSTRLEN
   MOV EBX, 1
   MOV EAX, 4
   INT 0x80
   ; The program is about to terminate
   MOV EAX, 60
   MOV EDI, 0
   SYSCALL
SECTION .data
```

```
myStr DB 'Works!', 0xA
MYSTRLEN EQU $ - myStr
```

Defining Constants And Variables

Constants are defined using the following syntax:

```
;<NAME> EQU <VALUE>
MYCONST EQU 10 ; Just a constant
MYSTRLEN EQU $ - myStr ; As seen before
```

When constants are declared, their type and size are not specified. No memory location is assigned to them either, as this is decided during the assembly step.

Variables have to be declared within the .data section, according to the syntax below:

```
;<NAME> <TYPE> <MAGNITUDE>
sqrTwo DD 1.4142
age DB 10
```

Uninitialized variables have to be declared within the .bss section instead:

```
;<NAME> <TYPE> <VALUE>
array1 RESB 5        ; 5 items byte array
array2 RESD 20       ; 20 items double array
```

Moving Data

To **move data** between registers and memory the MOV instruction is used which copies the second operand into the first one:

```
;MOV <DEST>, <SOURCE>
MOV RAX, QWORD [var1]  ; Move value
MOV RAX, var1  ; Move address
```

Data Conversion Instructions

Conversions are needed when switching from a size to another is necessary: therefore, conversions can be either widening or narrowing. Signed and unsigned conversions follow different rules when a widening one is being attempted: the upper-order bits must preserve the sign. While **narrowing conversions** do not require any special instruction, the programmer must make sure that the final result is the expected one:

```
MOV RBX, 10
; BL is used to access the lower 8-bits of RBX
MOV AL, BL
```

While performing **unsigned widening conversions**, the upper part of the memory or register will be equal to zero, as the number has only magnitude:

```
MOV AL, 10        ; Move 10 into AL
MOV RBX, 0        ; Initialise the whole RBX to 0
; Copy AL to BL leaving all upper zeroes in RBX
MOV BL, AL
```

Alternatively, the more specialized MOVZX instruction can be used.

Signed widening conversions set all upper-order bits equal to 0s or 1s, according to the original value. For example, as explained already in the previous chapter, to convert the number -9 from 16-bit to 32-bit:

```
+9 --> 0000000000001001 (get positive)
0000 0000 0000 1001 --> 1111 1111 1111 0110 (invert)
1111 1111 1111 0110 --> 1111 1111 1111 0111 (add 1)
1111 1111 1111 1111 1111 1111 1111 0111    (pad with 1)
```

Specialized instructions that can be used to perform signed widening conversions are: MOVSX (copies from source to destination and sign extends the value to 16 or 32-bit), MOVSXD (copies from source to destination and sign extends the value to 64-bit), CBW (converts byte in AL into word in AX, preserving the sign) and many more!

Debugging Assembly Programs With DDD

To **debug** the assembly programs presented in this course, it is recommended to install the **Data Display Debugger** (DDD) package, which is a **graphical front-end** for GDB. The user will get the full power of GDB command line and also will be given the ability to:

- Set and remove breakpoints using the mouse.
- Quickly visualize registers content.
- Quickly changing the source code invoking a text editor.
- Much more!

Assembly Integer Arithmetic

To **add** two integers together, the instruction ADD can be used:

```
;ADD  <DEST>, <SOURCE>
ADD ECX, EAX   ; ECX = EAX + ECX
```

The following rules apply:

- Destination and source must be of the same size.
- Final result is stored into destination.
- Destination must be able to store values.
- Both operands cannot be memory.

Addition With Carry

Addition with carry is used when adding numbers **larger than the size of the registers**, as this causes additions subsequent to the first one to include a possible carry from a previous operation. Considering that the carry bit is stored into the RFLAG register, the ADC instruction adds source, destination and carry bit together, storing the result into the destination:

```
;ADC <DEST>, <SRC>
ADD R1, R2      ; might set carry bit
ADC R3, R4      ; use carry bit
```

Increment Operator

The increment instruction is used for incrementing an operand by one. When a memory operand is used, the type has to be specified:

```
;INC <OPERAND>
INC BYTE [var1]      ; var1 = var1 + 1
INC WORD [var2]      ; var2 = var2 + 1
```

Subtraction

To perform **subtractions**, the instruction SUB is used:

```
;SUB <DEST>, <SOURCE>
SUB EAX, 200         ; EAX = EAX - 200
```

The following rules apply:

- Source and destination have to be the same size.
- Final result is stored into destination.

- Both operands cannot be memory.
- Destination cannot be a number.

Decrement Operator

The **decrement** instruction is used for decrementing the operator by one. When a memory operand is used, the type has to be specified:

```
;DEC <OPERAND>
DEC BYTE [var1]       ; var1 = var1 - 1
DEC WORD [var2]       ; var2 = var2 - 1
```

Signed And Unsigned Multiplication

To perform **unsigned multiplications**, the instruction MUL is used. While multiplicand has to be placed into a specific register, chosen according to the size of operands (AL, AX, EAX and RAX), the multiplier can be a memory location or a register. Results are also stored according to specific rules:

- AL * 8-bit = AH (upper), AL (lower)
- AX * 16-bit = DX (upper), AX (lower)
- EAX * 32-bit = EDX (upper), EAX (lower)
- RAX * 64-bit = RDX (upper), RAX (lower)

```
MOV AL, 2
MOV DL, 4
;MUL <SOURCE>
MUL DL
```

For **signed multiplications** instead, the instruction IMUL should be used. It comes with three different syntaxes, some of which do accept immediate values (numeric values such as 1, 2, 3…):

```
;IMUL <SRC>
IMUL AL     ; Same than MUL
;IMUL <DEST>, <SOURCE|IMMEDIATE>
IMUL AX, 10     ; DEST = DEST * SRC|IMM
;IMUL <DEST>, <SOURCE>, <IMMEDIATE>
IMUL RBX, RCX, 1024     ; DEST = SRC * IMM
```

Signed And Unsigned Division

For **signed divisions**, the instruction IDIV is used, while for **unsigned divisions** DIV is used: they both follow the same syntax than MUL. Moreover, the dividend have to be placed into a specific register, according to its size, while the divisor can be either a memory location or a register. Quotient and remainder are also stored into specific registers:

- [AH][AL] / 8-bit = AL (quotient), AH (remainder)
- [DX][AX] / 16-bit = AX (quotient), DX (remainder)
- [EDX][EAX] / 32-bit = EAX (quotient), EDX (remainder)
- [RDX][RAX] / 64-bit = RAX (quotient), RDX (remainder)

Working With Values And Addresses

The next example shows the difference in performance between executing arithmetic using the previous instructions and using **LEA** (Load Effective Address) instead. LEA is similar to MOV, but it works with addresses rather than values, giving the programmer the ability to perform some unsigned integer arithmetic with the content of the memory address pointed at by the source, before storing the result into the destination:

```
SECTION .text
  GLOBAL _start
_start:
  ; Standard methodology
  MOV ECX, 1313H
  MOV EBX, 2121H
  MOV EAX, 2
  MUL EBX
  ADD ECX, EAX     ; ECX = 0x5555
  ; Arithmetic with LEA
  MOV ECX, 1313H
  MOV EBX, 2121H
  LEA ECX, [EBX * 2 + ECX]     ; ECX = 0x5555
  ; Terminate program in Linux
  MOV RAX, 60
  MOV RDI, 0
  SYSCALL
```

The reader can see that while both methodologies produced the same result on the ECX register, the first approach required more lines of code.

Working With The Carry Bit

The following example adds two 8 bytes long variables together working with 4 bytes registers. This is possible because the addition is split into two parts: the second addition will also include the carry bit:

```
SECTION .text
  GLOBAL _start
_start:
  MOV EAX, DWORD [v1]
  MOV EDX, DWORD [v1 + 4]
  ; Add first part
```

```
    ADD EAX, DWORD [v2]
    ; Add second part and carry bit
    ADC EDX, DWORD [v2 + 4]
    MOV DWORD [vSum], EAX
    MOV DWORD [vSum + 4], EDX

    ; Terminate program in Linux
    MOV RAX, 60
    MOV RDI, 0
    SYSCALL

SECTION .data
    ; Labels are 8 bytes while registers only 4
    v1 DQ 0x1ABC000000000000
    v2 DQ 0x2DEF000000000000
    vSum DQ 0
```

The reader can use a calculator to double check the result.

Assembly Floating-Point Arithmetic

Floating-Point operations must run using dedicated registers: the **XMM registers**, created according the SSE extension. Therefore, sixteen new 128-bit registers, named XMM0 through XMM15, were added. Subsequently, these registers have been renamed to **YMM** and expanded to contain up to 256 bits according to the **AVX extension**. The **MXCSR** register contains control and status information for the XMMs ones.

Moving Floating-Point Values Into Floating-Point Registers

As floating-points have to be moved into floating-points registers before they can be operated upon, the instructions **MOVSS** and **MOVSD** can be used to achieve this goal. They both behave exactly like MOV, as they accept two parameters, the first one being the destination. However, while MOVSS is to be used when the source is 32-bit long, MOVSD is for 64-bit long sources. Also, the following rules apply:

- Both operands must be of the correct size.
- Both operands cannot be memory.
- Neither operands can be immediate.

From Integer To Floating-Point

As **integers** have to be converted before they can be used with floating-points, the following instructions can be used according to the size of source and destination:

```
;CVTSS2SD <XMM>, <SRC>
CVTSS2SD XMM3, XMM3     ; 32b FP > 64b FP
;CVTSD2SS <XMM>, <SRC>
CVTSS2SD XMM3, XMM3     ; 64b FP > 32b FP
;CVTSS2SI <XMM>, <SRC>
CVTSS2SD XMM3, XMM3     ; 32b FP > 32b I
;CVTSD2SI <XMM>, <SRC>
CVTSS2SD XMM3, XMM3     ; 64b FP > 32b I
```

Adding Floating-Point Numbers

For **floating-point additions**, the instructions ADDSS (single precision) and ADDSD (double precision) can be used. They work both like ADD: the destination XMM register, which is the first parameter, will contain the result. The following rules apply:

- Destination and source must be of the same size.
- No memory to memory operation is allowed.
- Source operand may not be immediate.

Subtracting Floating-Point Numbers

For **floating-point subtractions**, the instructions SUBSS (single precision) and SUBSD (double precision) can be used. They both work like SUB: the destination XMM register, which is the first parameter, will contain the result. The following rule apply:

- Destination and source must be of the same size.
- No memory to memory operation is allowed.
- Source operand may not be immediate.

Multiply Floating-Point Numbers

For **floating-point multiplications**, the instructions MULSS (single precision) and MULSD (double precision) can be used: the destination XMM register, which is the first parameter, will contain the result obtained by multiplying the first parameter by the second one:

```
;MULSS <XMM>, <SRC>
MULSS XMM2, DWORD [var1]
;MULSD <XMM>, <SRC>
MULSD XMM0, QWORD [var2]
```

The following rules apply:

- Destination and source must be of the same size.
- No memory to memory operation is allowed.
- Source operand may not be immediate.

Dividing Floating-Point Numbers

For **floating-point divisions**, the instructions DIVSS (single precisions) and DIVSD (double precision) can be used: the destination XMM register, which is the first parameter, will contain the result obtained by dividing the first parameter by the second one. The following rules apply:

- Destination and source must be of the same size.
- No memory to memory operation is allowed.
- Source operand may not be immediate.

A Final Example

The following program shows some of the above instructions:

```
SECTION .text
  GLOBAL _start
; In DDD console: info all-registers
; In DDD console: register MXCSR
_start:
```

```
    MOVSS XMM0, DWORD [sPVar1]
    MOVSS DWORD [sPVar2], XMM0
    MOVSS XMM1, DWORD [sPVar2]
    MOVSD XMM2, QWORD [dPVar1]
    MOVSD QWORD [dPVar2], XMM2
    MOVSD XMM3, QWORD [dPVar2]
    MOV EAX, DWORD [sPVarInt]
    CVTSI2SS XMM0, EAX          ; Int -> F-P
    ; Two memory locations, two operations!
    MOVSS XMM0, DWORD [sPVar3]
    ADDSS XMM0, DWORD [sPVar4]
    MOVSS DWORD [sPTot], XMM0
    ; This will not work!!!
    ; ADDSS DWORD [sPVar3], DWORD [sPVar4]
    ; MXCSR register will change now!
    MOVSS XMM0, DWORD [sPVar1]
    DIVSS XMM0, DWORD [sPDiv]

    ; Terminate program in Linux
    MOV RAX, 60
    MOV RDI, 0
    SYSCALL

SECTION .data
    sPVar1 DD 1.41
    sPVar2 DD 0.0
    dPVar1 DQ 2.82
    dPVar2 DQ 0.0
    sPVarInt DD 10
    sPVar3 DD 55.21
    sPVar4 DD 11.15
    sPTot DD 0.0
    sPDiv DD 0.0
```

Assembly Control Instructions

Assembly does not have anything like **if**, **for** or **while**. However, it is possible to modify the flow of a program via conditional and unconditional jump instructions. Assembly labels show the locations to jump to.

Jump Instructions

The instruction **JMP** is really simple to use, as it only accepts the name of the label or an address to jump to. For example, the following snippet could be used to force the program flow to jump to the section of the code that terminates the execution:

```
JMP end_program
; More code goes here
end_program:
  MOV RAX, 60
  MOV RDI, 0
  SYSCALL
```

Bear in mind that each label can only be defined once: the program would not know where to jump to otherwise!

Introduction To Conditional Execution

Conditions within the program can be tested using the instruction CMP, which stores the test result into RFLAG. The CMP instruction is followed by a jump instruction that defines where and if to jump according to the test result. The compare instruction accepts two parameters, the first one of which cannot be immediate. For example, the following code jumps to label ‘do_some’ if the value stored into AX is equal to 10:

```
CMP DX, 10
JE do_some
; More code here
do_some:
; More code here
```

More About Jump Instructions

Other jump instructions that can be used to match possible results returned by CMP (the programmer will need to specify the location to jump to for all of them)

- JL is for signed data and jumps if p1 < p2
- JLE is for signed data and jumps if p1 <= p2
- JG is for signed data, jump if p1 > p2
- JGE is for signed data, jumps if p1 >= p2

- JE jumps if p1 = p2
- JNE jumps if p1 ≠ p2
- JB is for unsigned data, jumps if p1 < p2
- JBE is for unsigned data, jumps if p1 <= p2
- JA is for unsigned data, jumps if p1 > p2
- JAE is for unsigned data, jumps if p1 >= p2

Jump Out Of Range

Bear in mind that, if the target label is not within 128 bytes away from the jump instruction, some assemblers might throw the **jump out of range error**, which has to be fixed by reorganising the code differently.

From Simple To Complex Iterations

Complex **iterations** such as 'for', not natively supported by assembly, can be obtained by using the concepts explained so far:

- Create and manipulate iterations counters using simple instructions such as 'add' and 'sub'
- Monitor these counters using 'cmp'
- Change program flow using jump instructions

Iterating Using Loop

Simple iterations can also be obtained using the LOOP instruction, which only accepts a single parameter: the location to jump to. This instruction works as follows:

- RCX is used as loop index
- RCX is decremented at every cycle
- Loop keeps jumping while RCX ≠ 0

Bear in mind that, as this instruction can only have a single index stored into RCX, managing nested cycles is very difficult as you might have to save the RCX value of the outer loop when the inner one starts!

A Final Example

The reader can now try compiling and executing the following program:

```
SECTION .text
  GLOBAL _start
_start:
  MOV RCX, 10
  MOV RAX, 0
addLoop:
  ADD QWORD [tot], RAX
```

```
    ADD RAX, 1
    DEC RCX
    CMP RCX, 0
    JNE addLoop    ; jump if RCX is not 0
    ; Add with loop
    MOV EAX, 0    ; EAX is tot
    MOV ECX, 10    ; ECX is counter
myloop:
    ADD EAX, ECX    ; EAX = EAX + ECX
    LOOP myloop    ; if ECX > 0 goto myloop

    ; Terminate Linux program
    MOV RAX, 60
    MOV RDI, 0
    SYSCALL

SECTION .data
    tot DQ 0
```

Assembly And Process Stack

A stack is a data structure where items are added and then removed in reverse order (**L**ast **I**n **F**irst **O**ut). To add an item you will need to push it in and to remove an item you will need to pop it out. Therefore, in an imaginary stack, using pseudo-code:

PUSH stack 10
PUSH stack 20
PUSH stack 30
POP stack --> This one prints 30
POP stack --> This one prints 20

The Program Memory Layout

The following table represents the program memory layout:

↓↓ ↓↓ ↓↓ STACK ↓↓ ↓↓ ↓↓	(high memory)
…	(high memory)
empty	(high memory)
…	(high memory)
↑↑ ↑↑ ↑↑ HEAP ↑↑ ↑↑ ↑↑	(high memory)
BSS	(low memory)
DATA	(low memory)
TEXT	(low memory)
RESERVED	(low memory)

Two big sections are present: low memory and high memory. In the low memory section, the following can be found:

- BSS section for uninitialized data.
- Data section for initialized data.
- Text section is for the body of the program.

In the high memory section the **stack** (which grows downwards) and the **heap** (which grows upwards) are located. Stack and heap are not supposed to collide, as the program will crash otherwise!

The **heap** area:

- Is for dynamic memory allocation (reference types, objects and JRE classes).
- Is not automatically deallocated.
- Is where mechanisms such as GC run to remove objects that lost their references.
- Grows upwards.
- Is slow.
- Is not thread-safe.

The **stack** area:

- Is for automatic allocation and deallocation of memory (primitive types, call frames for function calls, references to heap objects, short-lived and method-specific values).
- Grows downwards.
- Is fast.
- Is thread-safe.

The RSP Register And The Stack

Considering that RSP is the register stack pointer and it grows downwards:

- Popping a value out of a stack returns the location pointed to by RSP and increases RSP.
- Pushing a value into a stack decreases RSP and places the new item at the new memory location pointed to by RSP.

Popping And Pushing Data

In the assembly pop and push functions work as follows:

```
; [RSP] copied into "op", RSP = RSP + DQ
POP <OPERAND>
; RSP = RSP - DQ, "op" copied into [RSP]
PUSH <OPERAND>
```

The operand cannot be an immediate value. Only registers and memory are allowed. For 64-bit platforms these instructions operate on DQ (quadwords = 8 bytes).

Different Platforms Might Manage The Stack Differently

Stack and heap management strategies change across different systems: please always double check the official documentation!

A Final Example

The following example will allow the reader to practice the skills learnt in this class:

```
SECTION .text
  GLOBAL _start
_start:
  ; On DDD prompt: x/6d $rbx
  MOV RBX, vals
  MOV R15, 0       ; A flag
  MOV RCX, 3       ; How many items in the stack?
doPush:
  ; On DDD prompt: x/d $rsp
  PUSH QWORD [RBX + R15 * 8]
```

```
    INC R15
    LOOP doPush
    MOV RBX, vals
    MOV R15, 0
    MOV RCX, 3
doPop:
    ; On DDD prompt: p $rax
    POP RAX
    ; On DDD prompt: x/6d $rbx
    MOV QWORD [RBX + R15 * 8], RAX
    INC R15
    LOOP doPop

    ; Terminate Linux program
    MOV RAX, 60
    MOV RDI, 0
    SYSCALL

SECTION .data
    vals DQ 1000, 1001, 1002
```

Assembly Functions

Functions help programmers write code that is easier to maintain, debug and expand. Functions need to be able to:

- Return to the place where they were called.
- Handle arguments, by reference or value.

Dynamically And Statically Allocated Variables

In high-level languages, variables declared inside a functions are:

- **Stack dynamic**, because their address is determined dynamically when the program runs.
- Local, which means that when functions return, the memory space that was occupied by these variables is reused.

On the other hand, **statically** declared variables occupy memory until the whole program ends.

Calling Assembly Functions And Returning Values

Assembly functions are called by using the instruction CALL, while they return values using the instruction RET. Assembly functions are declared using the following format:

```
GLOBAL <funcName>
<funcName>:
  ; Function body goes here
RET
```

More in details, the instruction RET transfers control to the return address located on the stack, generally stored there by CALL.

Standard Calling Conventions For Functions

All functions must behave the same while performing common tasks such as returning values, managing parameters, allocating registers, otherwise the code would break. Therefore, standards calling conventions were created and they are also used by C and C++, which simplifies writing mixed code. The standard that is tought in this course is the one adopted by Linux, while Windows uses a slightly different one.

How CALL And RET Rely On The Stack

Linkage deals with calling functions and returning from them. As these steps both rely on the stack, wrong values stored into this area will make the program crash!

- CALL stores the address to return to by pushing the content of RIP register, which points to the next instruction, into the stack.
- RET transfers control to the return address located on the stack. Programmers need to make sure to pop and push correct values before running this command, otherwise the program will jump to random locations!

How To Pass Arguments To Functions

Arguments can be sent to a function by value or by reference in the following ways:

- Arguments can be stored into registers, although this approach is limited to number of registers. In fact, it is used for the first six integer arguments, while those from the seventh onward may be stored onto stack in reverse order.
- Arguments can be stored into global variables although this is a discouraged practice that is necessary in limited circumstances.
- Arguments can be stored onto stack. This methodology can manage unlimited number of arguments although it is a slower process than the registers option.

The following table clarifies how to manage any number of **integer** arguments according to their size using registers and stack:

	64-bit	**32-bit**	**16-bit**	**8-bit**
1	RDI	EDI	DI	DIL
2	RSI	ESI	SI	SIL
3	RDX	EDX	DX	DL
4	RCX	ECX	CX	CL
5	R8	R8D	R8W	R8B
6	R9	R9D	R9W	R9B
7	Stack	Stack	Stack	Stack
N	Stack	Stack	Stack	Stack

Floating-point arguments can be managed by simply replacing the previous registers with the floating-point ones (from YMM0/XMM0 to YMM7/XMM7).

Leaving A Function: Resetting The Stack

When the function ends, the **stack** is reset by modifying the RSP register, as the calling routine is responsible for resetting the stack:

RSP = RSP + [(number of args) * 8]

Leaving A Function: Returning Values

Calculations were made considering that each argument is 8 byte long. **Returning value** is then stored into a specific register, according to its size and type:

Returning Value	**Register Name**

Byte	AL
Word	AX
Double-Word	EAX
Quad-Word	RAX
Floating-Point	XMM0

How Functions Rely On Registers

The following table shows details about how each registry is used by assembly functions:

Register Name	Usage
RAX	Stores the return value
RBX	Callee saves this value onto stack, does required work and then restores it
RCX	4th Argument
RDX	3rd Argument
RSI	2nd Argument
RDI	1st Argument
RBP	Callee saves this value onto stack, does required work and then restores it
RSP	Stack pointer
R8	5th Argument
R9	6th Argument
R10 and R11	Temporary
From R12 To R15	Callee saves this value onto stack, does required work and then restores it

The registers R8, R9, R10, R11, RDI, RSI, RDX, RCX and all floating-point ones are not preserved after a function call. Therefore, they all can be used without having to write code to preserve their values across a function call.

Call Frame

Items on the stack that are part of a function call are referred to as call frame. A **call frame** may consist of the following:

- Stack dynamic local values.
- Passed arguments.
- Preserved registers.
- Return address, always required.
- Much more…

Simple functions might not need a full call frame if they:

- Do not call another function.
- Do not modify any of saved registers.
- Do not require local variable which are stored onto stack.
- Pass arguments only in registers.

If stack-based arguments or local variables are present, RBP, which points at the base of the stack, has to be pushed and then made pointing at itself. If a pop or a push is performed, this would affect RSP which points at the top of the stack, while RBP will not change. Therefore RBP can be used to access arguments stored into the stack and dynamic local variables. In **Linux**, the first 128 bytes after RSP are reserved and should not be used by the programmer.

A Final Example

In the following example, “numbers” is an array of double words (4 bytes) and the function “numExtractor” finds the first and the last element of the array, the sum of all elements, the average and the median although not all functionalities are fully implemented. As the function requires eight arguments, two of them, avg and sum, will be stored onto stack as only six registers are available. As the function starts:

- It pushes RBP value on top of the stack.
- It copies the value of RSP, which stores the location of the top of the stack, onto RBP. Therefore, RBP points at its own value stored onto the stack.

The function then cycles through the array in order to calculate the sum of all items. The total is then returned as reference into R10 and this step requires two lines of code. The address of the sum located into the stack is calculated by adding 16 to RBP:

Pop Order	Stack Content
4	AVG
3	SUM
2	RIP
1	RBP

Next, the last value of the array is retrieved and then placed into R9. Finally, just before returning, the stack is cleared by using two pop instructions that remove R10 and RBP. When the function returns, the stack is reset by removing the two eight bytes arguments, sum and avg, that were passed to the function using the stack:

```
SECTION .text
  GLOBAL _start
_start:
  ; Parameters section starts - 8th
  ; These parameters are passed using the stack
  PUSH avg
  PUSH sum
  ; These parameters are passed using registers
  MOV R9, last
  MOV R8, md2
  MOV RCX, md1
  MOV RDX, first
```

```
    MOV ESI, DWORD [numElems]
    MOV RDI, numbers
    ; Parameters section ends - 1st
    CALL numExtractor
    ; Clear 2 results onto stack 8 bytes each
    ADD RSP, 16

    ; Terminate Linux program
    MOV RAX, 60
    MOV RDI, 0
    SYSCALL

    GLOBAL numExtractor
numExtractor:
    MOV RAX, 0              ; Total
    MOV R10, 0              ; Flag

    ; RBP is top of the stack
    PUSH RBP
    ; RBP points at itself
    MOV RBP, RSP
    ; In DDD console: print *(int *) ($rsp)
    PUSH R10
sumLoop:
    ; sum = sum + num[i]
    ADD EAX, DWORD [RDI + R10 * 4]
    INC R10
    CMP R10, RSI
    JL sumLoop
    ; Call by reference requires two steps!!!
    ; 1. Retrieve address of sum
    MOV R10, QWORD [RBP + 16]
    ; 2. Return address of sum
    MOV DWORD [R10], EAX

    ; Get numElems
    MOV R10, RSI
    ; Set numElems - 1
    DEC R10
    ; Get last
    MOV EAX, DWORD [RDI + R10 * 4]
    ; In DDD console: print *(int *) ($r9)
    MOV DWORD [R9], EAX

    ; Get first
    MOV EAX, DWORD [RDI]
```

```
    ; In DDD console: print *(int *) ($rdx)
    MOV DWORD [RDX], EAX

    ; Add code to get average and median(s)

    POP R10
    ; Pop stack into RBP to prepare next instruction
    POP RBP
    RET

SECTION .data
    avg DD 0
    sum DD 0
    md1 DD 0
    md2 DD 0
    last DD 0
    first DD 0
    numElems DD 5
    numbers DD 2, 4, 6, 8, 10
```

Stack Buffer Overflow

Stack buffer overflow (SBO) happens when stack-based dynamic variables overflow. The following C code clarifies this concept:

```
int myArray[5] = { 0, 1, 2, 3, 4 };
int myVar = 0;
/* The following lines would cause a SBO
if myArray was stored onto stack */
myArray[6] = 5;
myVar = myArray[6];
myVar = 41756418731968713;
```

How Stack Buffer Overflow Can Make A Program Crash

Stack buffer overflow can make a program crash, because the stack, which contains the function return address, stack-based arguments and preserved registers, is overwritten. Those standard calling conventions that were covered in the chapter about **assembly functions**, make programs flows predictable, which enables ill-intentioned ones to inject code in order to take over the system or damage it. The following graph shows what the stack might look like while executing the previous C snippet, should a SBO occur:

RIP
RBP
↑↑ ↑↑ ↑↑ myArray[] ↑↑ ↑↑ ↑↑
↑↑ ↑↑ ↑↑ myVar ↑↑ ↑↑ ↑↑
RBX
R12
R13

As the arrows show, should myArray or myVar overflow, both the values of RBP and RIP might be overwritten. Considering that RIP points at the next instruction to be executed, this SBO would cause the program to crash! In fact, when RIP is overwritten:

- Functions are not able to return to the calling routines.
- Malicious code might be executed instead, if the program interacts with external components.

How Stack Buffer Overflow Exploits Work

If the platform and/or the program allow for SBO, the following are risk scenarios:

- Untrusted users might gain access to data required by the program.
- Untrusted files might be read by the program.
- Untrusted hosts or networks might send data to the program.

Malicious assembly code to be executed after a SBO is generally typed in HEX to avoid issues with special characters. Many tools, including the Linux shell, allow users to type using HEX. To type command in a Linux shell using HEX:

- Open a Linux shell
- Hold down [CTRL][Shift]
- Type ‘u’
- Type the HEX code
- Release all keys
- Press [Space]

Using HEX assembly code, malicious users might try opening a shell on your system. Attackers will still need to guess the location of RIP through trial error, in order to have RIP to point at the address of the malicious code, although the exact address is not required, as multiple NOP (no operation) can be used. In fact, the system will keep on running into these NOPs until the actual malicious code is found and executed. Considering that the NOP hexadecimal code for the Intel x86 CPU family is 0x90, the following schema clarifies what actually happens in the stack when this type of problem occurs:

…	...
0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90	**⇇ RIP is overwritten**
0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90	**⇇ RBP is overwritten**
0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90	
MALICIOUS CODE	
0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90	
RBX	
R12	
R13	

How To Mitigate And Prevent Buffer Overflow Attacks

The following technologies give users some protection against SBO:

- **ASLR** (Address Space Layout Randomization) is a software based technology that randomises the location where executables, stack and heap are loaded into memory. This can be activated on Linux by setting '/proc/sys/kernel/randomize_va_space' to a 1 or 2. On Windows, from Vista onward, this protection works at application and operating system level, therefore, applications have to be built with the support for ASLR.
- **Canaries** is a software based technology that detects SBO by inserting random values into the RIP. Execution is stopped if these values are modified or deleted. This option is activated by default in GCC but it can be disable by using the "-fno-stack-protector" flag.
- **DEP** (Data Execution Prevention) is a hardware and software based technology that prevents SBO by marking memory areas as either executable or non-executable. DEP cannot be disabled on Linux, while on Windows, from XP onward, can be disabled by using bcdedit.exe.

Calling System Services

From time to time programmers might need the system to perform tasks such as:

- Getting input from the keyboard.
- Printing messages on the console.
- Retrieving the system date.
- File operations (open a file to read or write it).
- Much more…

This is achieved by executing a **system service call**, which is like calling a function that is stored within the operating system. However, as system services do not generally use stack-based arguments, they are limited to six functions parameters, as only registers can be used, as explained in the chapter that covers assembly functions.

How System Service Calls Work

Each system service is associated to a fixed numeric value, which is to be stored into the RAX register before performing the specific system service call. Tables that show the numeric value of each system service are available for each operating system. On Linux, the files ‘unistd_32.h’ and ‘unistd_64.h’ define these values, while on Windows, the NTDLL of the specific installation has to be analyzed. The following table explains how to manage registers with system services:

Register Name	Use
RAX	System service to be called
RDI	1st Argument
RSI	2nd Argument
RDX	3rd Argument
R10	4th Argument
R8	5th Argument
R9	6th Argument

Differences Between INT And SYSCALL

Although explaining the differences between INT and SYSCALL goes well beyond the scope of this course, the following can be said:

- The instruction **INT** triggers a software interrupt, it is slow than other options but has wider support as it is always implemented.
- The instructions **SYSENTER** (32-bit) and **SYSCALL** (64-bit) do not trigger a software interrupt, they are faster than INT but they are not supported by all platforms.

Managing Files Using System Service Calls

The following table teaches how to set registers for opening or creating files:

Register Name	Use
RAX	85
RDI	File name
RSI	File access mode flag

The following table teaches how to set registers for reading files:

Register Name	Use
RAX	0
RDI	File descriptor
RSI	Address used to store characters to read
RDX	Number of characters to read

The following table teaches how to set registers for writing to files:

Register Name	Use
RAX	1
RDI	File descriptor
RSI	Address of characters to write
RDX	Number of characters to write

A Final Example

This chapter ends with the following example, which writes back to the screen the user input:

```
SECTION .text
  GLOBAL _start
_start:

  ; Read input
  MOV RAX, GETUSERINPUT
  MOV RDI, STDIN
  MOV RSI, buffer
  MOV RDX, bufferSize
  SYSCALL

  ; Write user input back
  MOV RDX, RAX
  MOV RAX, WRITEUSERINPUT
  MOV RDI, STDOUT
  MOV RSI, buffer
  SYSCALL

  ; Terminate program
  MOV RAX, 60
  MOV RDI, 0
  SYSCALL
```

```
SECTION .data
   GETUSERINPUT EQU 0
   WRITEUSERINPUT EQU 1
   STDIN EQU 0
   STDOUT EQU 1

SECTION .bss
   bufferSize EQU 32
   buffer RESB bufferSize
```

PE And COFF File Formats

The following points quickly describe the old COFF (Common Object File Format) format:

- COFF is a format for executable, shared library and object code.
- COFF original design was limited, as it had no standard representation for long long data type (at least 64 bits) and limited debugging capabilities.
- Many extensions such as XCOFF and ECOFF were created to overcome these limitations.
- Although COFF was replaced by ELF, it still used on Windows and some Unix-like systems.
- COFF files provide space for debugging information.
- Programmers can customise the format at compile time.

The COFF Format Internal Layout

COFF defines elements that contain information about the symbols and the sections of the program such as .data, .text and .bss. These elements are: file header, optional header information, table of section headers, raw data for initialised sections, relocation information for initialised sections, line numbers, symbol table and string table. The following table describe the internal layout of typical COFF files:

FILE HEADER
OPTIONAL FILE HEADER
.TEXT SECTION HEADER
.DATA SECTION HEADER
.BSS SECTION HEADER
SAMPLE_SECTION HEADER
.TEXT SECTION RAW DATA
.DATA SECTION RAW DATA
SAMPLE_SECTION RAW DATA
.TEXT RELOCATION INFO
.DATA RELOCATION INFO
SAMPLE_SECTION RELOCATION INFO
SYMBOL TABLE
STRING TABLE

From COFF To PE

Microsoft re-engineered COFF to create the **PE** (Portable Executable) format, which defines the standard for executable (image) files, dynamic-link libraries, object code and much more. This is the standard that .exe, .dll, .ocx, .drv, .sys, .mui and many more extensions have to follow.

Moreover, in Windows, **executable** files are referred to as PE, while **objects** files as COFF and while the PE .NET extension can invoke the CLR while the PE32+ handles 64-bit code.

PE files do not contain position-independent code. All addresses emitted by the compiler/linker are fixed ahead of time. If preferred addresses cannot be used, the operating system will recalculate all absolute addresses adapting the code to the new values, which is called rebasing.

The PE Format Internal Layout

The following table describe the internal layout of PE files:

DOS HEADER
PE/COFF HEADER
OPTIONAL HEADER
Data directories arrays
SECTION HEADERS ARRAY
.TEXT SECTION
.BSS SECTION
…
…

A PE file would contain all sections the operating system needs in order to properly execute it:

- .text for the program body.
- .bss for uninitialized variables.
- .data for initialized variables.
- .rsrc for resource directories.
- .edata for export tables.
- .idata for import tables.
- .debug for debug information.
- .cormeta for object files that contain managed code.
- .drective for linker options.
- .pdata for exceptions information.
- .reloc for image relocations.
- Many more!

The **DOS header**:

- Is only present in image files.
- Was introduced with DOS 2.0.
- Starts with “MZ”.
- Is 64 bytes large.
- Displays a warning message when a WINNT executable runs on DOS although this behaviour is highly customizable. This feature is used by antivirus packages to locate executable files in memory.
- Contains several fields, but its last 4 bytes indicate where the PE file header is located: this file offset is placed at location 0x3c during linking.

The **PE signature** is only present in image files and it starts with a 4 bytes signature that says “PE”.

The **COFF header** is only for objects and image files and contains several fields that describe:

- The CPU architecture.

- The size of the section table.
- The linker timestamp.
- The offset of the symbol table.
- The number of symbols in the symbol table.
- The size of the optional header.
- The image type attributes such as whether the file is an executable, a system file, a dynamic link library, for uni-processor or multi-processor, whether should be fully loaded and copied to swap file, whether debug information was removed and much more!
- Many more…

The **optional header** is for image files only and describes how the PE file should be managed. It contains several fields:

- The magic field that says whether the image is a normal executable, a PE32+ executable or a ROM image.
- The version of the linker.
- The size of sections such as .text and .data.
- A pointer to the entry point.
- Pointers to sections such as .text and .data.
- The required operating system version.
- The checksum.
- The required subsystem, such as graphical environment, Windows character subsystem, POSIX character subsystem, EFI ROM image, XBOX and many more!
- The size to reserve onto the stack to allow the program to run smoothly.
- Many more…

The **data directories arrays** allow the operating system to know the location and size of each and every data directory. As this is an actual array, the system needs a correct index to access. The following are some of data directories that can be found:

- The import (.idata) and export (.edata) tables which are used to import and export data to and from external libraries.
- The resource table (.rsrc) used to index all resource by using a multiple-level binary-sorted tree structure.
- The exception table (.pdata) is used for exception handling.
- The base relocation table (.reloc) that contains entries for all base relocations in the image.
- The thread local storage data table (.tls) allow threads to maintain different values for a variable.
- Many more…

Bear in mind that each one of these directories has its own structure, therefore, it is not easy to parse them.

In the **section table**, each row is a 40 bytes long section header while their number is defined by the the NumberOfSections field in the file header.

The **section data** contains initialized data. Several restrictions apply to section data, especially in regards to its location and alignment. However, this topic would go well beyond the scope of this course.

The **COFF relocations** are for object files only and they describe how the section data should be modified when loaded into memory. Image files do not need COFF relocations as referenced symbols addresses are located into a single contiguous address space.

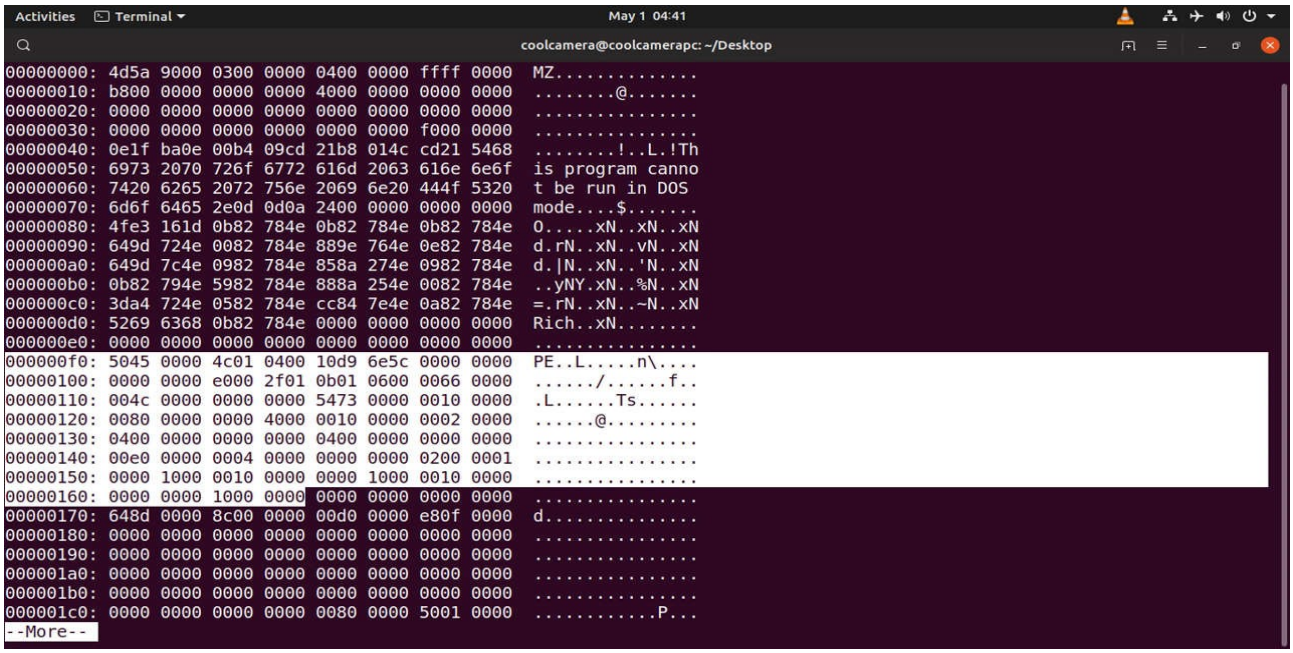
The **COFF symbol table** is the old symbol table inherited from the original COFF standard. A file can contain a COFF symbol table and Visual Studio debug info too. Although some applications still rely on this section, its use is limited.

The **delay-load import table** is for image files only and it allows applications to delay the loading of a dynamic-link library until it is actually needed.

How To Analyze PE Files

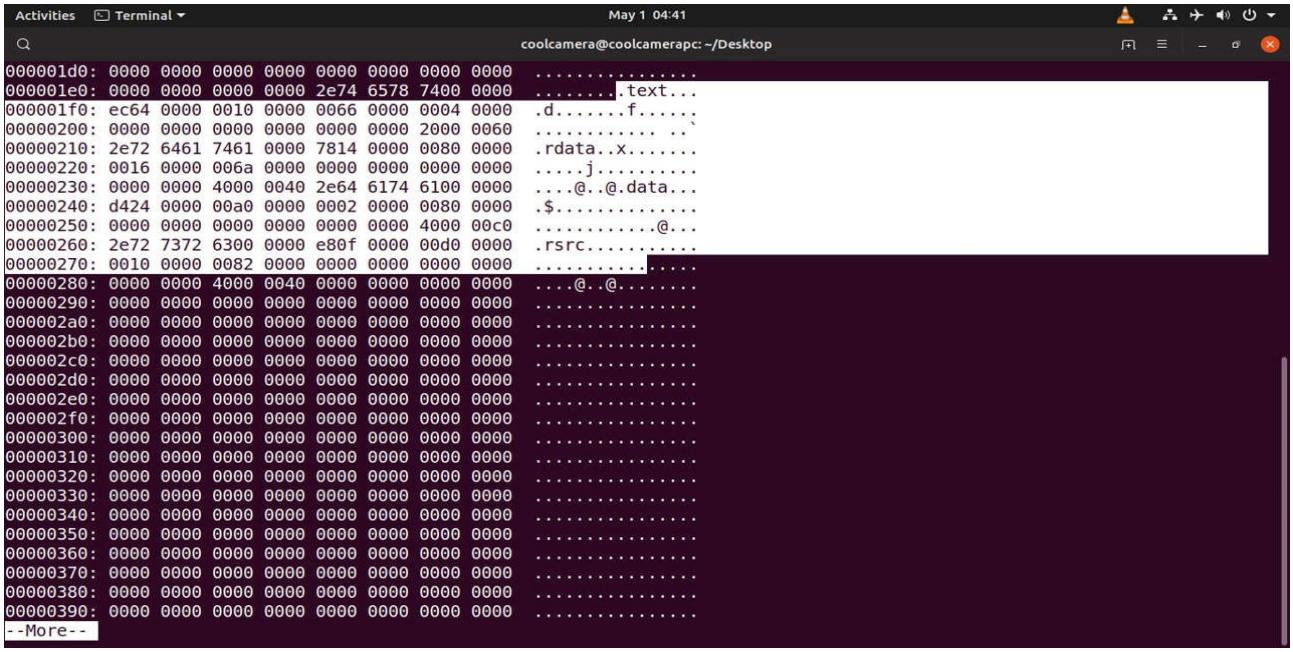
As this chapter is just an introduction to the PE and COFF formats and although they were both only briefly covered so far, the theoretical explanation will end here. However, the following sections will guide the reader throught the steps that are required to analyze the PE file format. Hexadecimal viewers can be used in oder to extract information out of PE files without executing them. On Linux platforms, xxd can be used as follows in order to display 16 octets per row (which should be the default):

```
$> xxd -c 16 [file_name] | more
```

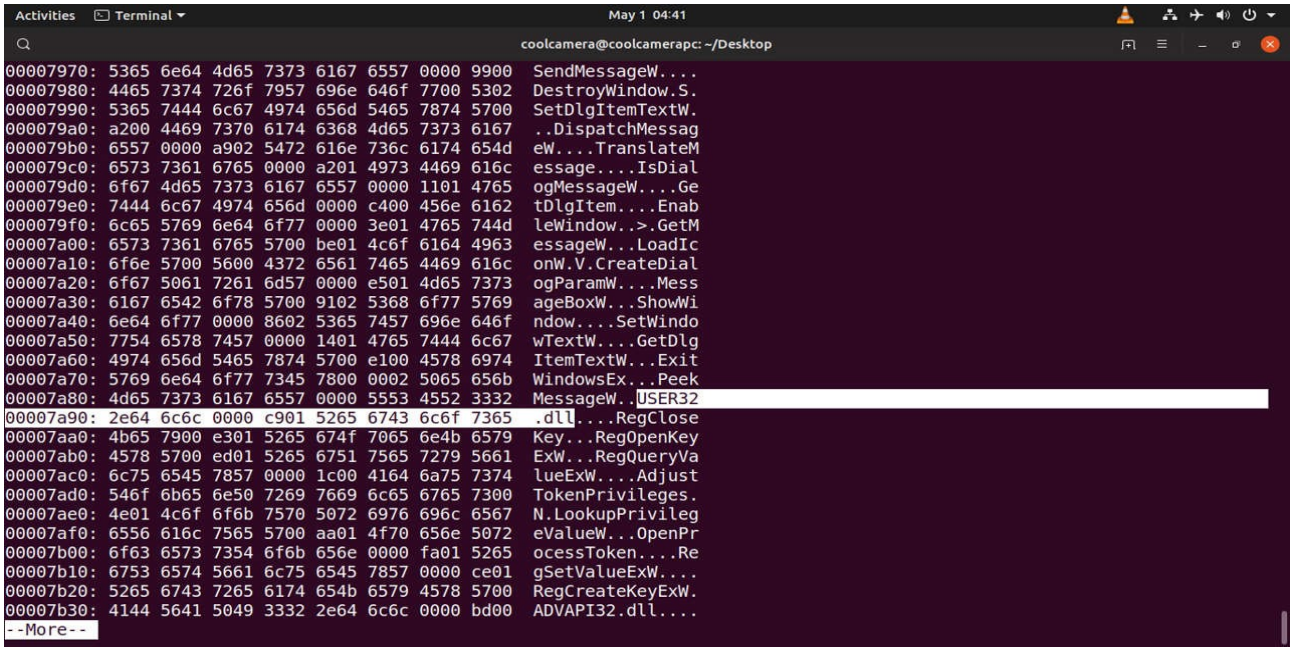


The file begins with “MZ” which is the magic number that indentifies Microsoft executable files. This also confirms that the right section is being analyzed. In fact, within the same area, the typical warning message of Windows applications is clearly visible. The start and the end of the PE header

were highlighted as this is the where the application **entry point** is located. Right after the PE header, the presence of strings such as ".text", ".data" and ".bss" would indicate that the **section table** has been found, as the next picture shows (no precise indication is given of exact boundaries):



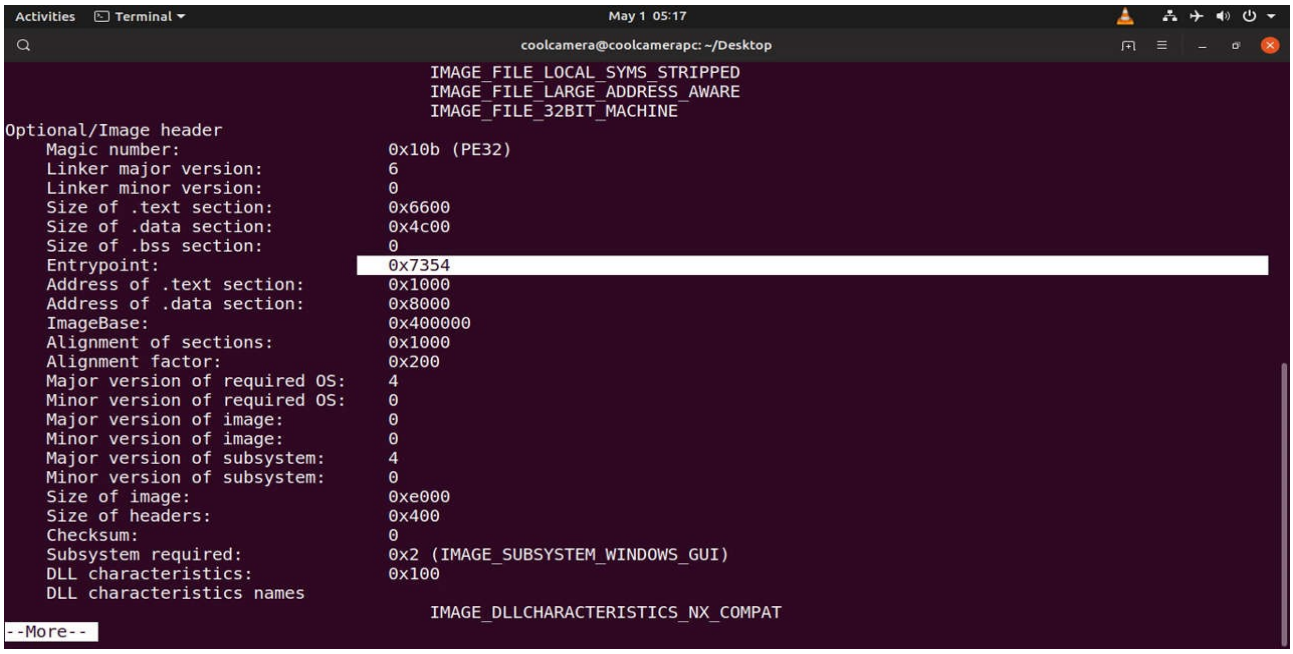
Scrolling down even more, xxd output shows the **libraries** that the application is using:



Analyzing these files using a simple hexadecimal viewer is possible, but time consuming. Specialized applications can make the life of reverse engineers a little bit easier. The package that will be presented in this section is called pev and it is available for both Microsoft and Linux platforms. Once the package is installed, the command line to be launched to start analyzing files is the following:

```
$> readpe [file_name] | more
```

This tool is extremely easy to use as it groups and labels all data efficiently. In fact, as the next picture shows, the application entry point, the address and size of the .text and .data sections, the required subsystem and all required information is readily available to the reverse engineer:



ELF File Format

ELF (Executable and Linkable Format) is the standard binary format for Linux and Unix-like systems on x86 processors. Although in the Linux world filename extensions do not have the same weight than in the Microsoft universe, the ELF format is associated with the following extensions: none, .bin, .o, .ko, .so, .elf, .prx and many more. These days is safe to say that ELF has largely replaced a.out and COFF file formats.

The ELF Format Internal Layout

The following drawing describe the ELF file structure:

ELF HEADER
PROGRAM HEADER TABLE
.TEXT
…
.DATA
SECTION HEADER TABLE

Just by looking at the schema above, the user would immediately guess why system programmers decided to transition from older formats to ELF. In fact, the design of the latter is much easier to manage than any alternative.

The **ELF header**, which can be either 52 bytes long (32-bit architectures) or 64 bytes long (64-bit architectures) contains the following information:

- The magic number
- The application class, which is equal to one for 32-bit applications or equal to two in 64-bit applications
- The endianness, which can be equal to either little-endian or big-endian.
- The ELF version, which for the time being, should always be equal to one.
- The Application Binary Interface version and target platform.
- The file type, which can be ET_NONE for files that do not have any file type, ET_EXEC for executable files, ET_DYN for shared object files, ET_CORE for core files, ET_REL for files before being linked to executable and many more…
- The entry point from where the process starts executing.
- Much more…

The **Program header table**, which is for executable and shared object files only, is an array of structures describing how to map the file into virtual address space for run-time execution, so that different parts of the program can be loaded into different memory locations.

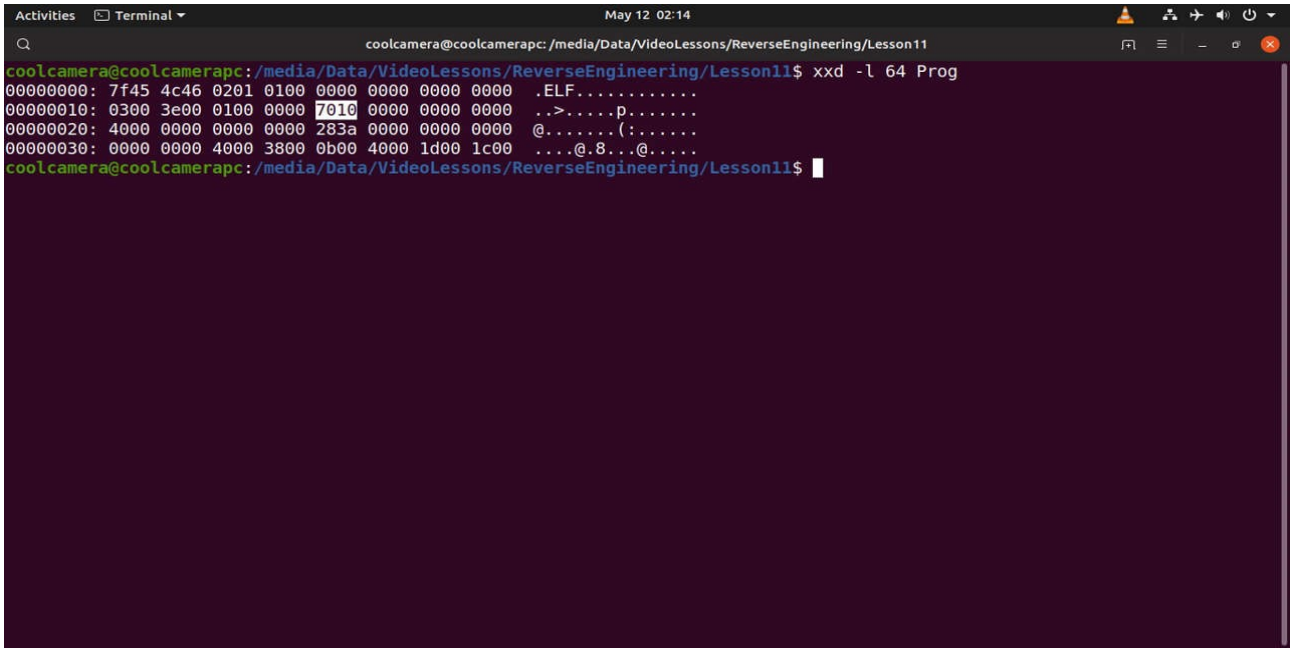
The **section header table** is an array of structures that contain metadata about a given section. Executable files need at least .rodata, .bss, .data, .text. ELF executable files consist of an ELF header, a program header table or a section header table or both of them.

How To Analyze ELF Files

After having created a small C program for Linux (64-bit), xxd was used as follows in order to find and inspect the **elf header**:

```
$> xxd -l 64 [file_name]
```

The option “-l 64” displays the first 64 octects, which is what was needed, as the elf header is 64 octects long on 64-bit machines. The output of the command was the following:



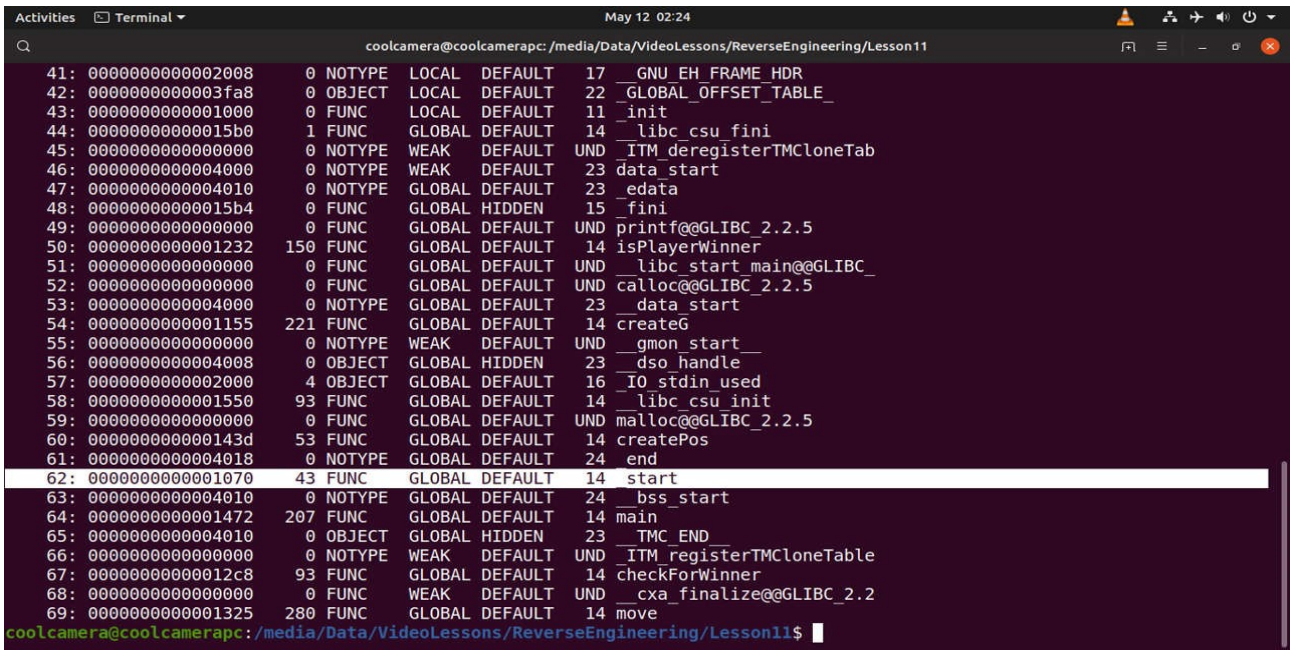
From the top left, in the given order, the meaning of each octect is the following while the entry point is highlighted:

- 7f 45 4c 46 means that this is an ELF file.
- 02 means that this is a 64-bit application.
- 01 means that little endian encoding is used.
- 01 means that ELF version 1 is used.
- This big group of zeros is just a place holder for future developments.
- 70 10 is the entry point. However, as little endian encoding is used, these numbers have to be swapped. Therefore, the actual entry point is 10 70.

In order to confirm these findings, the application **readelf** can be run as follows:

```
$> readelf --symbols [file_name] | more
```

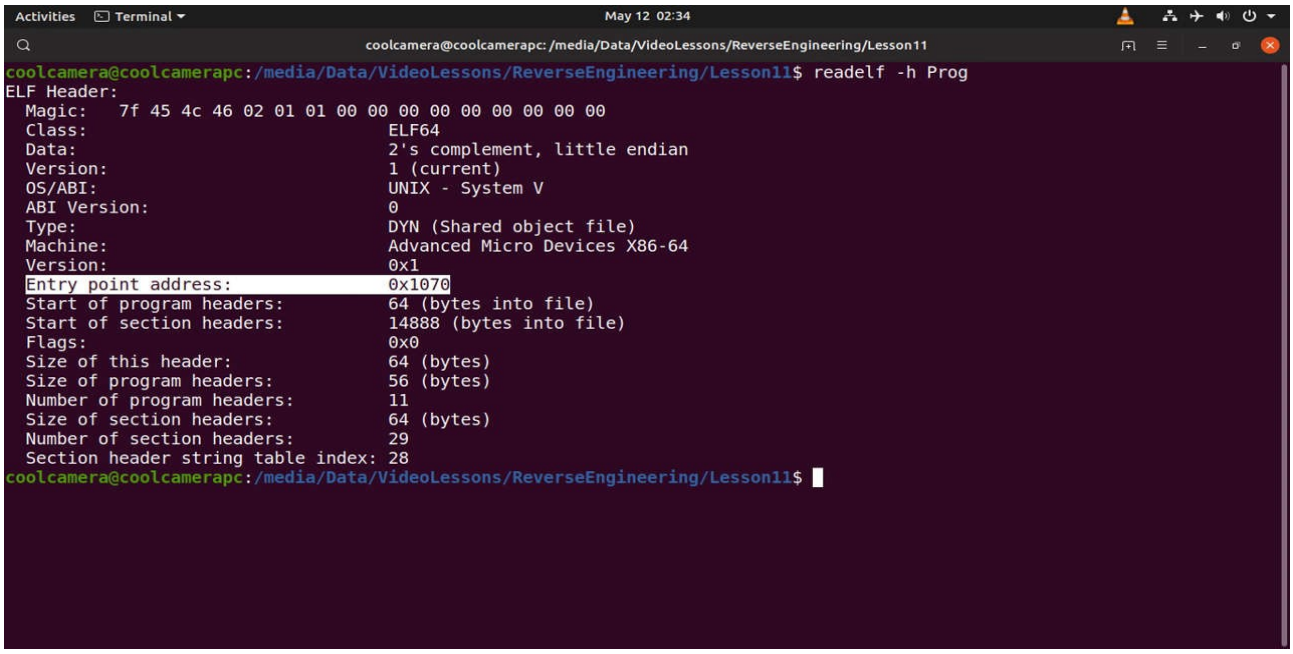
The output of the program was the following:



Perfect! The function **_start**, the entry point, is at 10 70 as expected. However, two lines below, the function **main** shows up, which also sounds like a perfect candidate for being the entry point. However, the _start function calls the main, therefore, the former is the actual entry point. To double check on these findings readelf can be used again to display the ELF **file header**:

```
$> readelf -h [file_name]
```

The output of the program was the following:

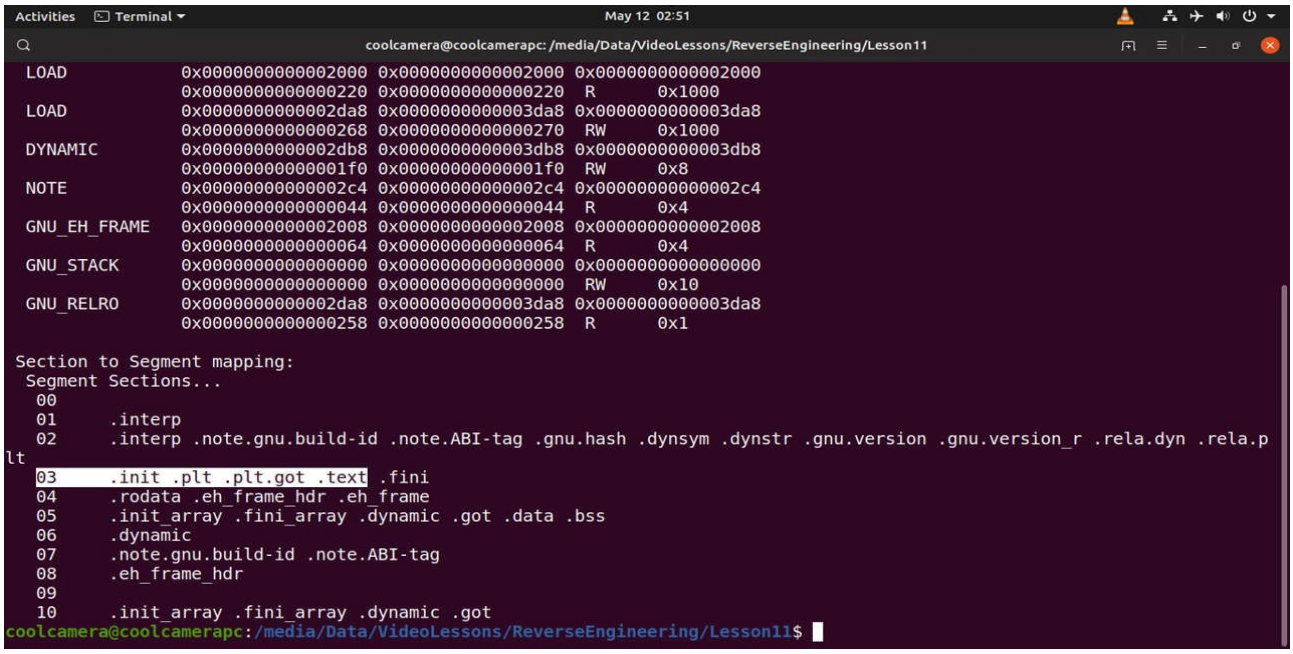


The screenshot above does shed some light that allows the reader to be sure about the actual address of the entry point. The same command is also revealing additional information that reverse engineers find useful.

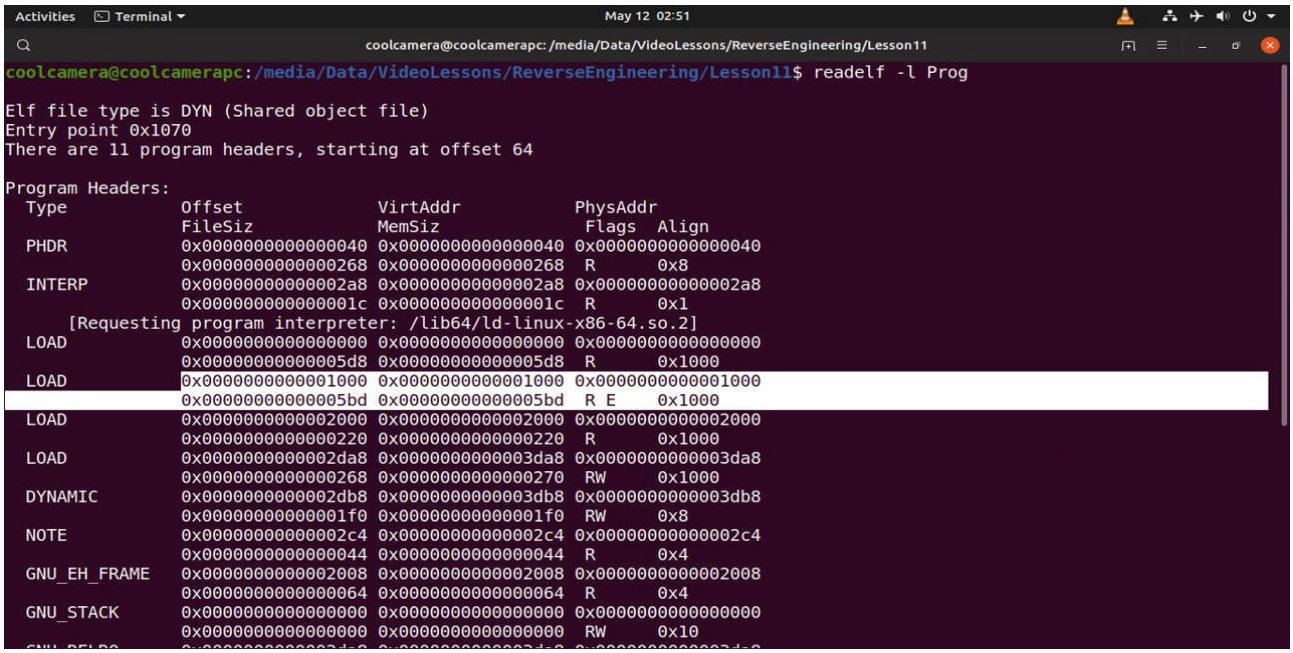
The next readelf option allows the user to inspect the **program header**, which tells the system how to load all the sections of the program into memory:

```
$> readelf -l [file_name]
```

Which produced the following output:



According to the output above, the highlighted area shows that the **.text** section, the program body, is associated to the number three. Two lines below, the **.data** and **.bss** sections, which contain the program variable, are associated to the number five. The initial output of the same command exaplains what this means:



Counting the lines starting from zero, the **.text** area is located on the fourth position (highlighted), which means that the program body will be loaded into memory with R(ead) and E(xecute) permissions as expected. In fact, no one should be allowed to modify the program logic while the software is running. Moreover, **.data** and **.bss** sections would fall into the sixth line, therefore, variables will be loaded into memory with R(ead) and W(rite) permissions, as these are supposed to be modified by the program.

Reverse Engineering With Ghidra

Ghidra is a free reverse engineering tool released by National Security Agency (NSA). This chapter shows how to perform basic reverse engineering of simple packages. As Ghidra needs a working Java JDK, the reader will need to make sure to have the correct version of this development environment installed on the machine. The first application to be analyzed is a simple C program:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  printf("Hello, World!");

  return 0;
}
```

Creating And Managing Projects With Ghidra

After having compiled the application above with a C compiler, the reader will need to download and start Ghidra. Next, the user will need to:

- Create a non-shared Ghidra project as there will not be multiple engineers working on this program
- Choosing the project directory and the project name
- Starting the code browser, which at the moment is triggered by the green dragon icon

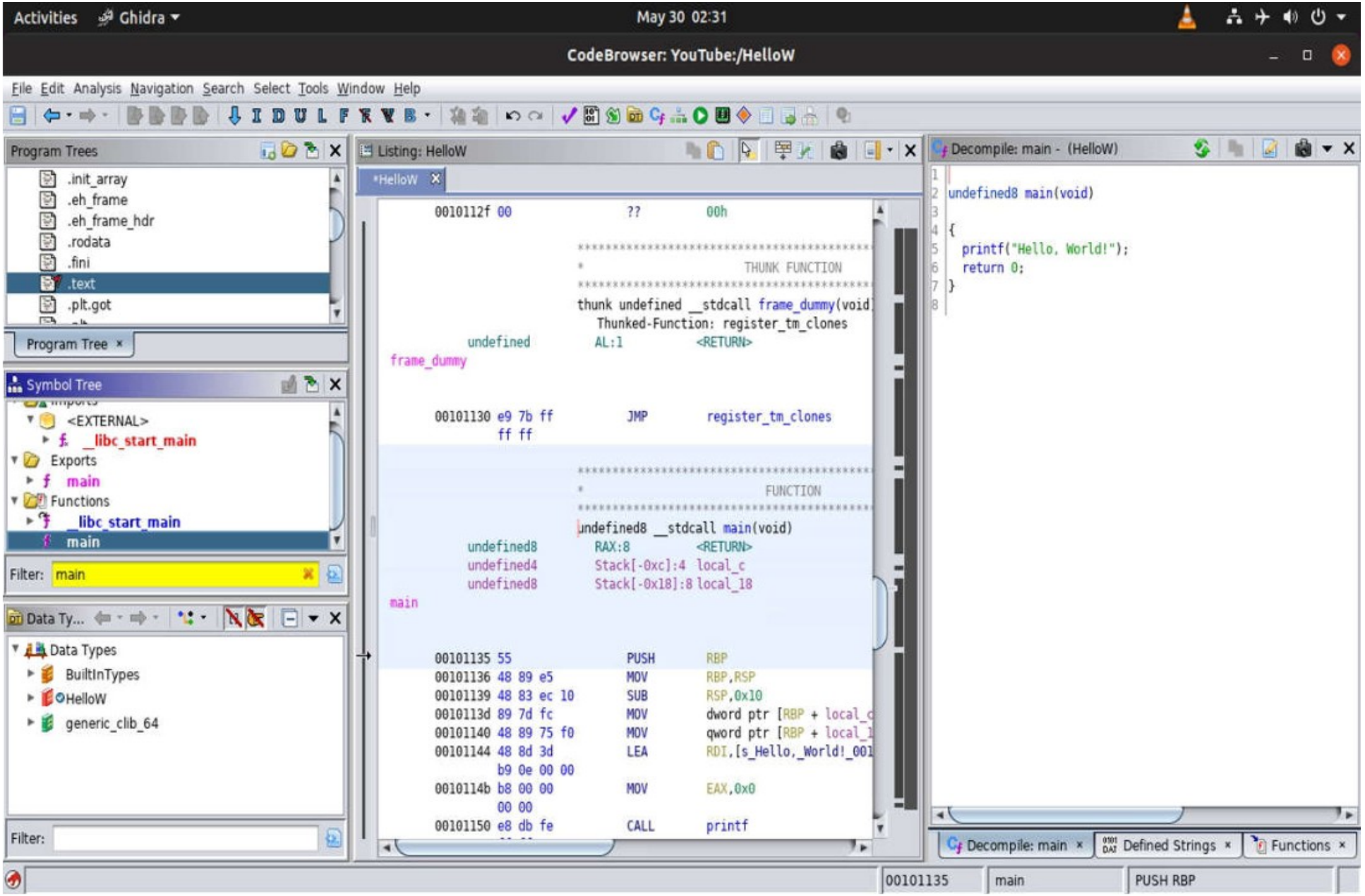
At this stage the newly created project is just an empty folder, the user will need to import the binary version of the C program using the menu “File -> Import File…”. Bear in mind that multiple applications can be imported into the same project.

The user can adjust properties that Ghidra is likely to detect automatically, such as:

- The **file format**, which will be ELF for a Linux binary
- The **language** that was used to create the binary, which is likely to be the GNU Compiler Collection (GCC). The other option should be Visual Studio

Start Analyzing The Executable With Ghidra

Once the binary is successfully imported into the project, Ghidra will analyze the application with all analyzers that the user chooses to include, which is also the last of preparatory steps. The “Symbol Tree” panel on the left can be used to locate any function by name, using the filter text box. Once the main function was located, the right panel, which contains the decompiled code, did show that Ghidra was able to reverse engineer the application, although the resulting source code did not exactly match the original one, as the following screenshot shows:

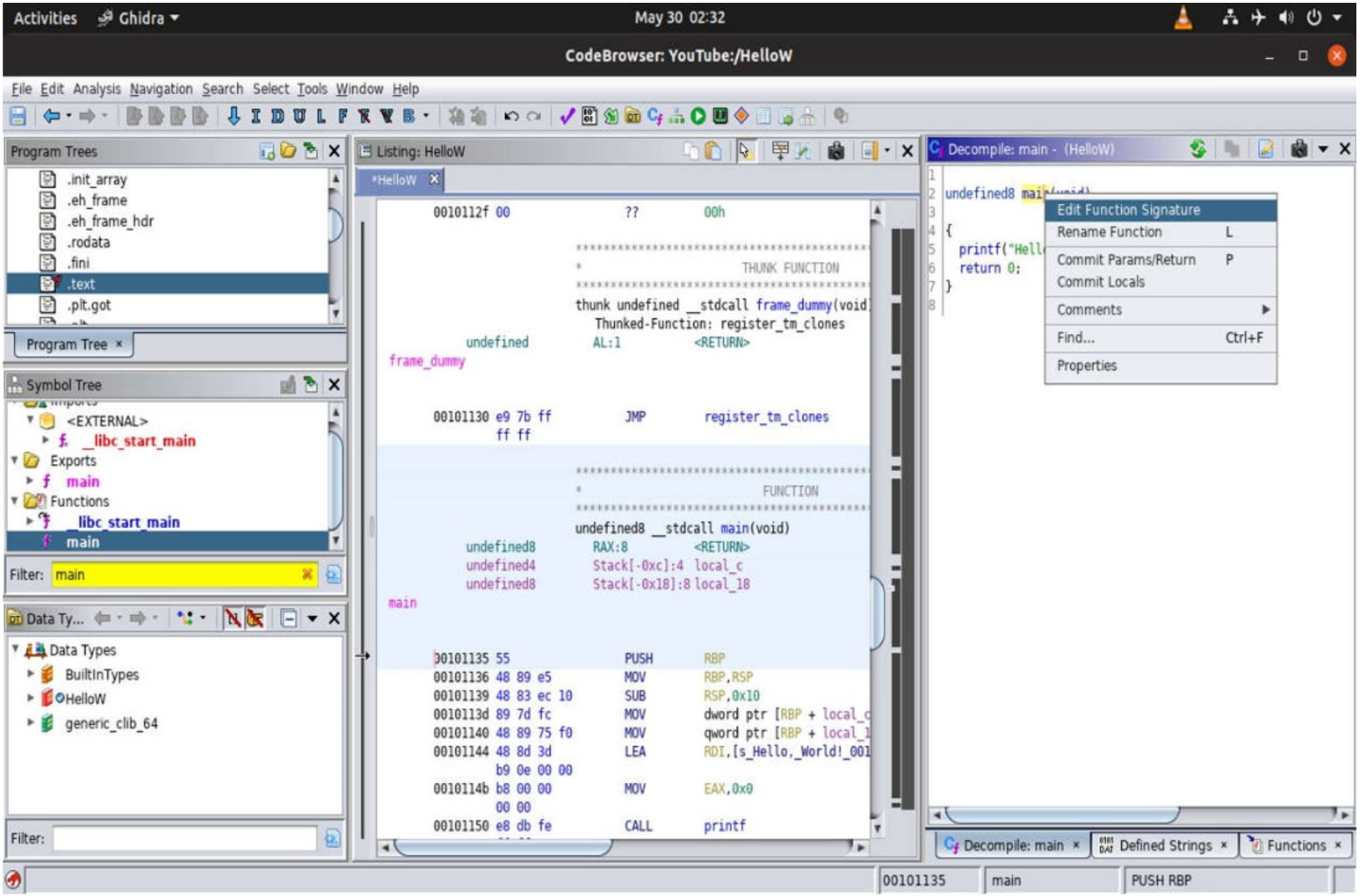

Activities
Ghidra
May 30 02:31
CodeBrowser: YouTube:/HelloW
File Edit Analysis Navigation Search Select Tools Window Help
Program Trees
.init_array
.eh_frame
.eh_frame_hdr
.rodata
.fini
.text
.plt.got
Program Tree
Symbol Tree
<EXTERNAL>
__libc_start_main
Exports
main
Functions
__libc_start_main
main
Filter: main
Data Ty...
Data Types
BuiltInTypes
HelloW
generic_clib_64
Filter:
Listing: HelloW
*HelloW
0010112f 00 ?? 00h
THUNK FUNCTION
thunk undefined __stdcall frame_dummy(void)
Thunked-Function: register_tm_clones
undefined AL:1 <RETURN>
frame_dummy
00101130 e9 7b ff ff ff JMP register_tm_clones
FUNCTION
undefined8 __stdcall main(void)
undefined8 RAX:8 <RETURN>
undefined4 Stack[-0xc]:4 local_c
undefined8 Stack[-0x18]:8 local_18
main
00101135 55 PUSH RBP
00101136 48 89 e5 MOV RBP,RSP
00101139 48 83 ec 10 SUB RSP,0x10
0010113d 89 7d fc MOV dword ptr [RBP + local_c
00101140 48 89 75 f0 MOV qword ptr [RBP + local_1
00101144 48 8d 3d b9 0e 00 00 LEA RDI,[s_Hello,_World!_001
0010114b b8 00 00 00 00 MOV EAX,0x0
00101150 e8 db fe CALL printf
Decompile: main - (HelloW)
undefined8 main(void)
{
printf("Hello, World!");
return 0;
}
Decompile: main
Defined Strings
Functions
00101135
main
PUSH RBP

Managing Functions Signatures With Ghidra

In fact, the **signature** of the function main changed as follows:

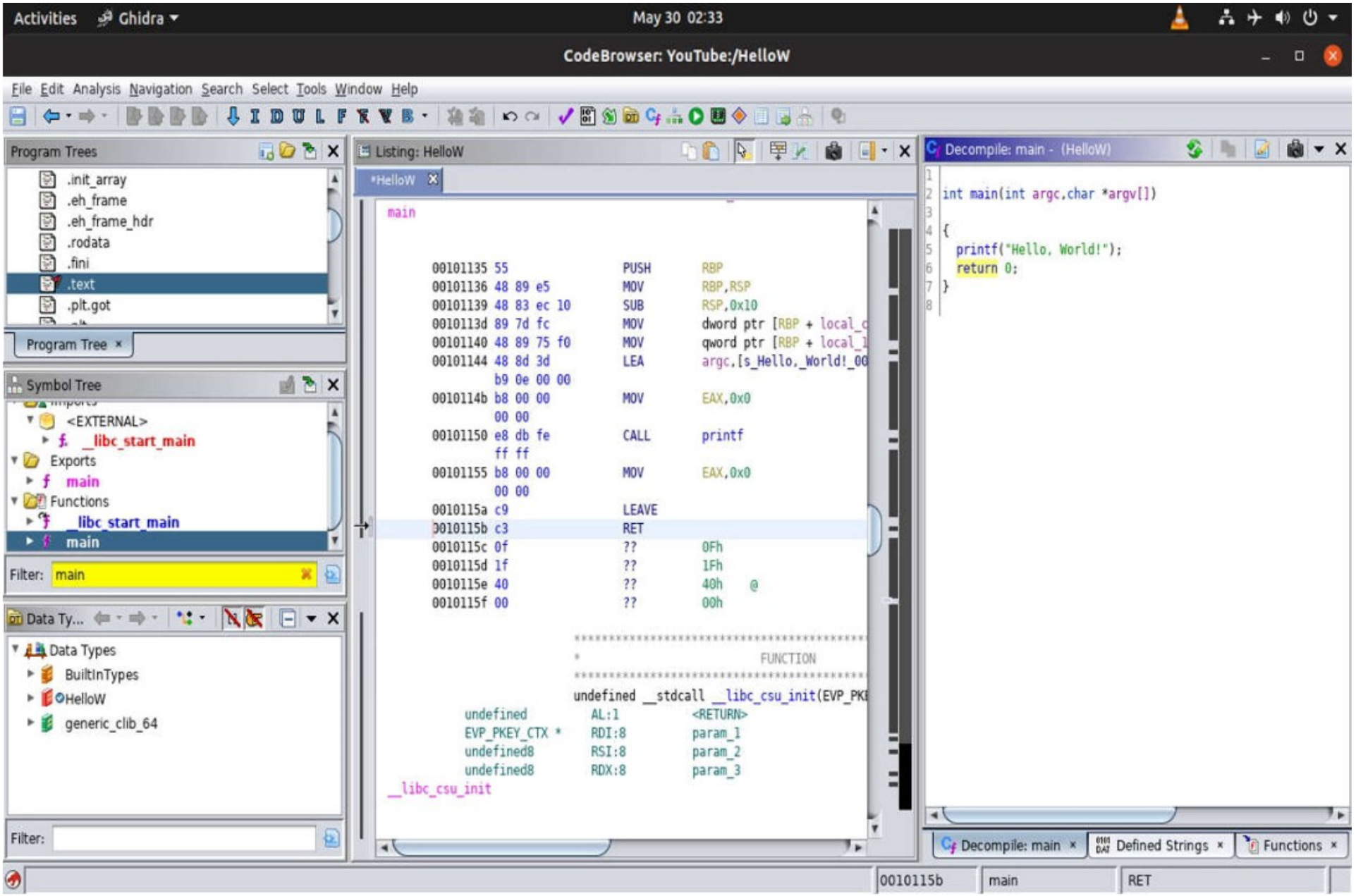
- The return type changed from integer to undefined8
- The two function arguments disappeared

To fix this, the user can right click on the main function, choosing the option that enables the reverse engineer to **change the signature**:


Activities
Ghidra
May 30 02:32
CodeBrowser: YouTube:/HelloW
File Edit Analysis Navigation Search Select Tools Window Help
Program Trees
.init_array
.eh_frame
.eh_frame_hdr
.rodata
.fini
.text
.plt.got
Program Tree
Symbol Tree
<EXTERNAL>
__libc_start_main
Exports
main
Functions
__libc_start_main
main
Filter: main
Data Ty...
Data Types
BuiltInTypes
HelloW
generic_clib_64
Filter:
Listing: HelloW
*HelloW
0010112f 00 ?? 00h
THUNK FUNCTION
thunk undefined __stdcall frame_dummy(void)
Thunked-Function: register_tm_clones
undefined AL:1 <RETURN>
frame_dummy
00101130 e9 7b ff ff JMP register_tm_clones
FUNCTION
undefined8 __stdcall main(void)
undefined8 RAX:8 <RETURN>
undefined4 Stack[-0xc]:4 local_c
undefined8 Stack[-0x18]:8 local_18
main
00101135 55 PUSH RBP
00101136 48 89 e5 MOV RBP,RSP
00101139 48 83 ec 10 SUB RSP,0x10
0010113d 89 7d fc MOV dword ptr [RBP + local_c
00101140 48 89 75 f0 MOV qword ptr [RBP + local_1
00101144 48 8d 3d b9 0e 00 00 LEA RDI,[s_Hello,_World!_001
0010114b b8 00 00 00 00 MOV EAX,0x0
00101150 e8 db fe CALL printf
Decompile: main - (HelloW)
undefined8 main(void)
{
printf("Hell
return 0;
}
Edit Function Signature
Rename Function L
Commit Params/Return P
Commit Locals
Comments
Find... Ctrl+F
Properties
Decompile: main
Defined Strings
Functions
00101135
main
PUSH RBP

Inspecting The Executable With Ghidra

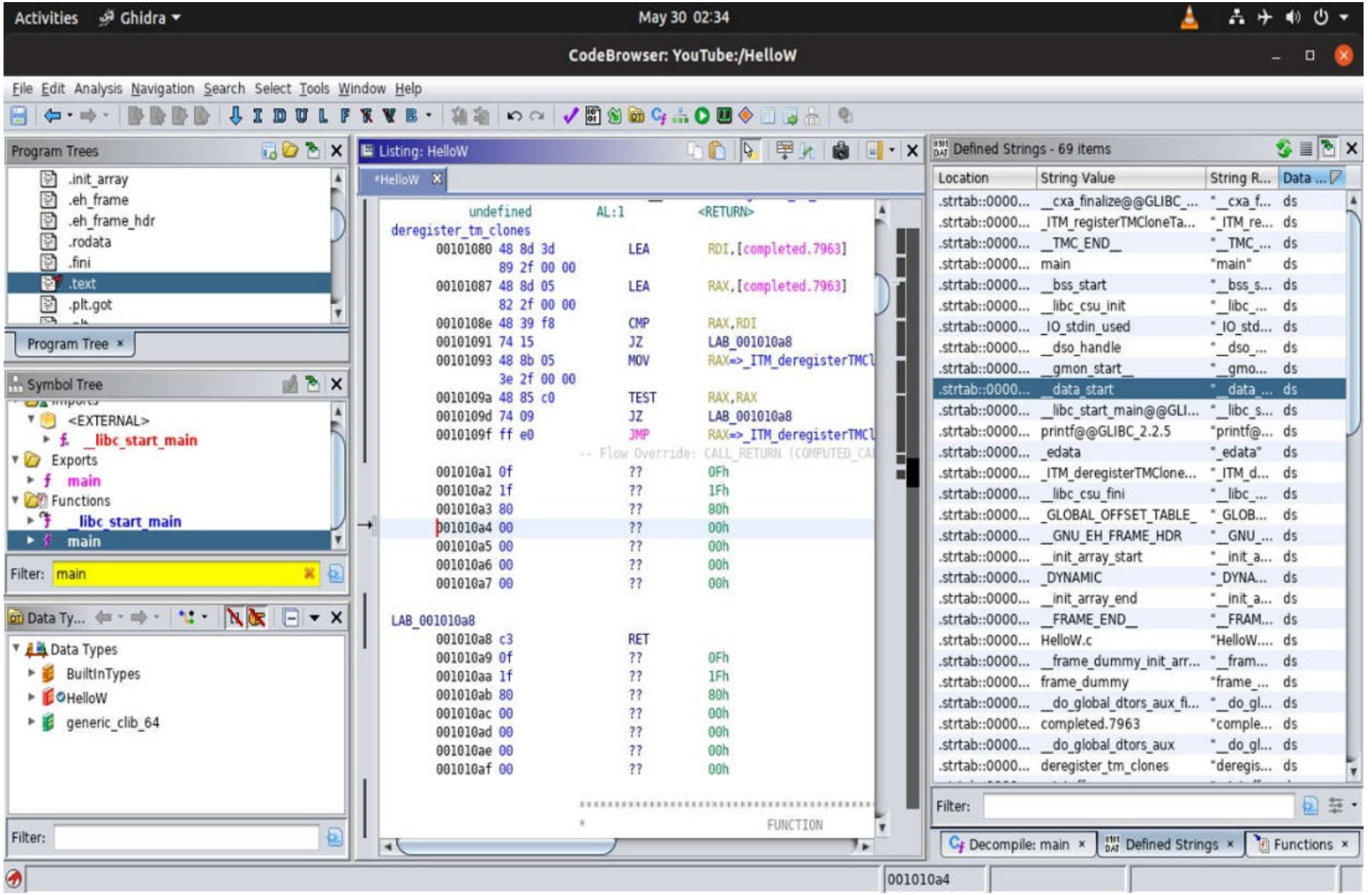
In order to inspect the assembly code that this C program is generating, the user can click on any C instruction: the middle panel will lead the reverse engineer to the corresponding assembly. The screenshot below shows what Ghidra is associating to the return function:


Activities
Ghidra
May 30 02:33
CodeBrowser: YouTube:/HelloW
File Edit Analysis Navigation Search Select Tools Window Help
Program Trees
.init_array
.eh_frame
.eh_frame_hdr
.rodata
.fini
.text
.plt.got
Program Tree
Symbol Tree
<EXTERNAL>
__libc_start_main
Exports
main
Functions
__libc_start_main
main
Filter: main
Data Ty...
Data Types
BuiltInTypes
HelloW
generic_clib_64
Filter:
Listing: HelloW
*HelloW
main
00101135 55 PUSH RBP
00101136 48 89 e5 MOV RBP,RSP
00101139 48 83 ec 10 SUB RSP,0x10
0010113d 89 7d fc MOV dword ptr [RBP + local_c
00101140 48 89 75 f0 MOV qword ptr [RBP + local_1
00101144 48 8d 3d b9 0e 00 00 LEA argc,[s_Hello,_World!_00
0010114b b8 00 00 00 00 MOV EAX,0x0
00101150 e8 db fe ff ff CALL printf
00101155 b8 00 00 00 00 MOV EAX,0x0
0010115a c9 LEAVE
0010115b c3 RET
0010115c 0f ?? 0Fh
0010115d 1f ?? 1Fh
0010115e 40 ?? 40h @
0010115f 00 ?? 00h
FUNCTION
undefined __stdcall __libc_csu_init(EVP_PK
undefined AL:1 <RETURN>
EVP_PKEY_CTX * RDI:8 param_1
undefined8 RSI:8 param_2
undefined8 RDX:8 param_3
__libc_csu_init
Decompile: main - (HelloW)
1 int main(int argc,char *argv[])
2 {
3 printf("Hello, World!");
4 return 0;
5 }
Decompile: main
Defined Strings
Functions
0010115b
main
RET

The content of the middle window, where the assembly is displayed, should be familiar to all readers of this course by now.

Analyzing Strings With Ghidra

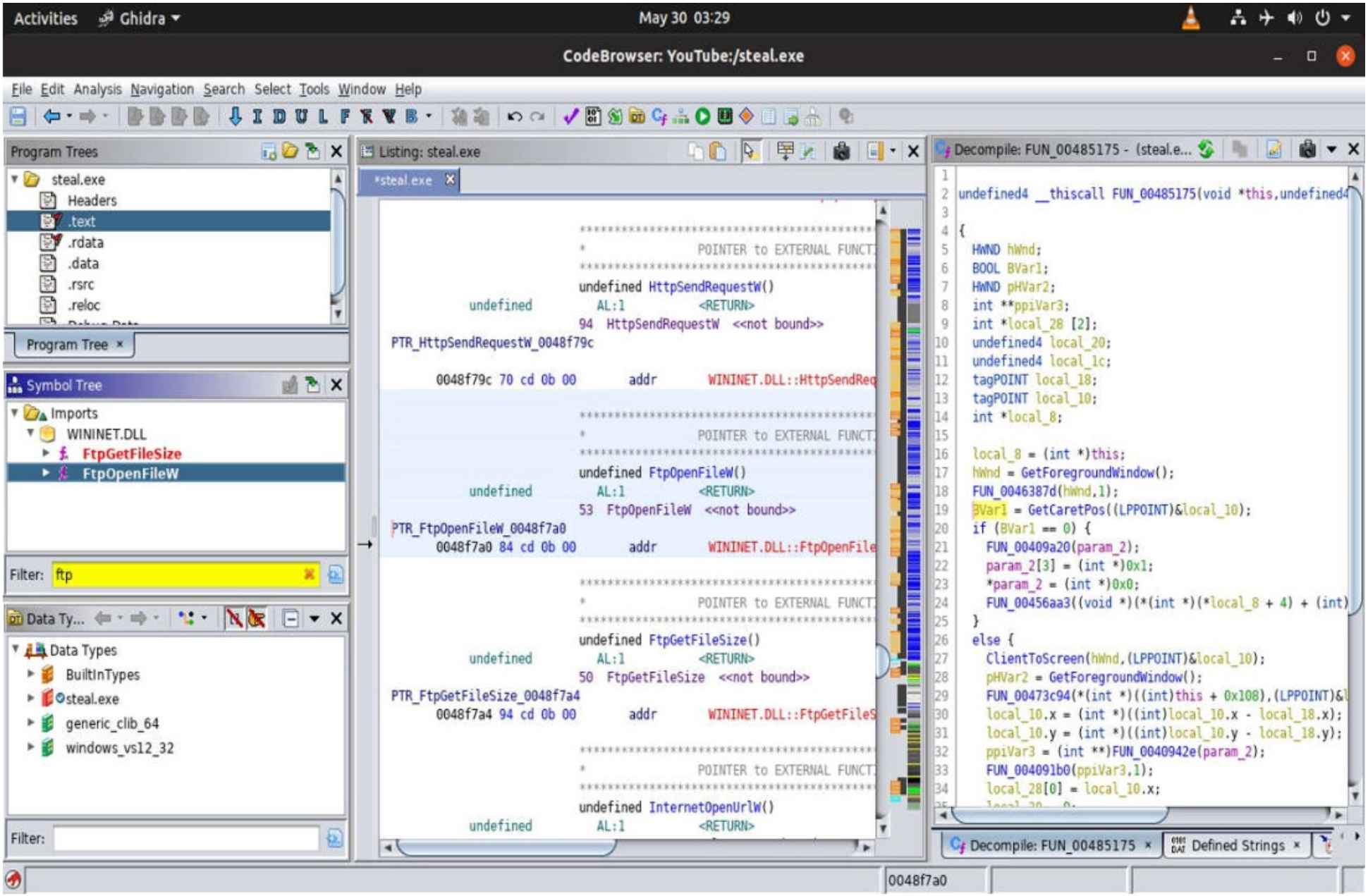
One more important part of the reverse engineering process is the analysis of strings, as they often reveal important information about the application itself, such as passwords, database connection **strings**, external resources details, information about exception handlers and so on. Ghidra can make this step easy, as it can list all strings present in a package: the user just needs to click on the “Defined Strings” tab, located in the right panel. The result would be something similar to the following:


Activities
Ghidra
May 30 02:34
CodeBrowser: YouTube:/HelloW
File Edit Analysis Navigation Search Select Tools Window Help
Program Trees
.init_array
.eh_frame
.eh_frame_hdr
.rodata
.fini
.text
.plt.got
Program Tree
Symbol Tree
<EXTERNAL>
__libc_start_main
Exports
main
Functions
__libc_start_main
main
Filter: main
Data Ty...
Data Types
BuiltInTypes
HelloW
generic_clib_64
Filter:
Listing: HelloW
*HelloW
undefined AL:1 <RETURN>
deregister_tm_clones
00101080 48 8d 3d 89 2f 00 00 LEA RDI,[completed.7963]
00101087 48 8d 05 82 2f 00 00 LEA RAX,[completed.7963]
0010108e 48 39 f8 CMP RAX,RDI
00101091 74 15 JZ LAB_001010a8
00101093 48 8b 05 3e 2f 00 00 MOV RAX=>_ITM_deregisterTMCl
0010109a 48 85 c0 TEST RAX,RAX
0010109d 74 09 JZ LAB_001010a8
0010109f ff e0 JMP RAX=>_ITM_deregisterTMCl
-- Flow Override: CALL_RETURN (COMPUTED_CA
001010a1 0f ?? 0Fh
001010a2 1f ?? 1Fh
001010a3 80 ?? 80h
001010a4 00 ?? 00h
001010a5 00 ?? 00h
001010a6 00 ?? 00h
001010a7 00 ?? 00h
LAB_001010a8
001010a8 c3 RET
001010a9 0f ?? 0Fh
001010aa 1f ?? 1Fh
001010ab 80 ?? 80h
001010ac 00 ?? 00h
001010ad 00 ?? 00h
001010ae 00 ?? 00h
001010af 00 ?? 00h
FUNCTION
Defined Strings - 69 items
Location String Value String R... Data ...
.strtab::0000... __cxa_finalize@@GLIBC_... "__cxa_f... ds
.strtab::0000... _ITM_registerTMCloneTa... "_ITM_re... ds
.strtab::0000... __TMC_END__ "__TMC_... ds
.strtab::0000... main "main" ds
.strtab::0000... __bss_start "__bss_s... ds
.strtab::0000... __libc_csu_init "__libc_... ds
.strtab::0000... _IO_stdin_used "_IO_std... ds
.strtab::0000... __dso_handle "__dso_... ds
.strtab::0000... __gmon_start__ "__gmo... ds
.strtab::0000... __data_start "__data_... ds
.strtab::0000... __libc_start_main@@GLI... "__libc_s... ds
.strtab::0000... printf@@GLIBC_2.2.5 "printf@... ds
.strtab::0000... _edata "_edata" ds
.strtab::0000... _ITM_deregisterTMClone... "_ITM_d... ds
.strtab::0000... __libc_csu_fini "__libc_... ds
.strtab::0000... _GLOBAL_OFFSET_TABLE_ "_GLOB... ds
.strtab::0000... __GNU_EH_FRAME_HDR "__GNU_... ds
.strtab::0000... __init_array_start "__init_a... ds
.strtab::0000... _DYNAMIC "_DYNA... ds
.strtab::0000... __init_array_end "__init_a... ds
.strtab::0000... __FRAME_END__ "__FRAM... ds
.strtab::0000... HelloW.c "HelloW.... ds
.strtab::0000... __frame_dummy_init_arr... "__fram... ds
.strtab::0000... frame_dummy "frame_... ds
.strtab::0000... __do_global_dtors_aux_fi... "__do_gl... ds
.strtab::0000... completed.7963 "comple... ds
.strtab::0000... __do_global_dtors_aux "__do_gl... ds
.strtab::0000... deregister_tm_clones "deregis... ds
Filter:
Decompile: main
Defined Strings
Functions
001010a4

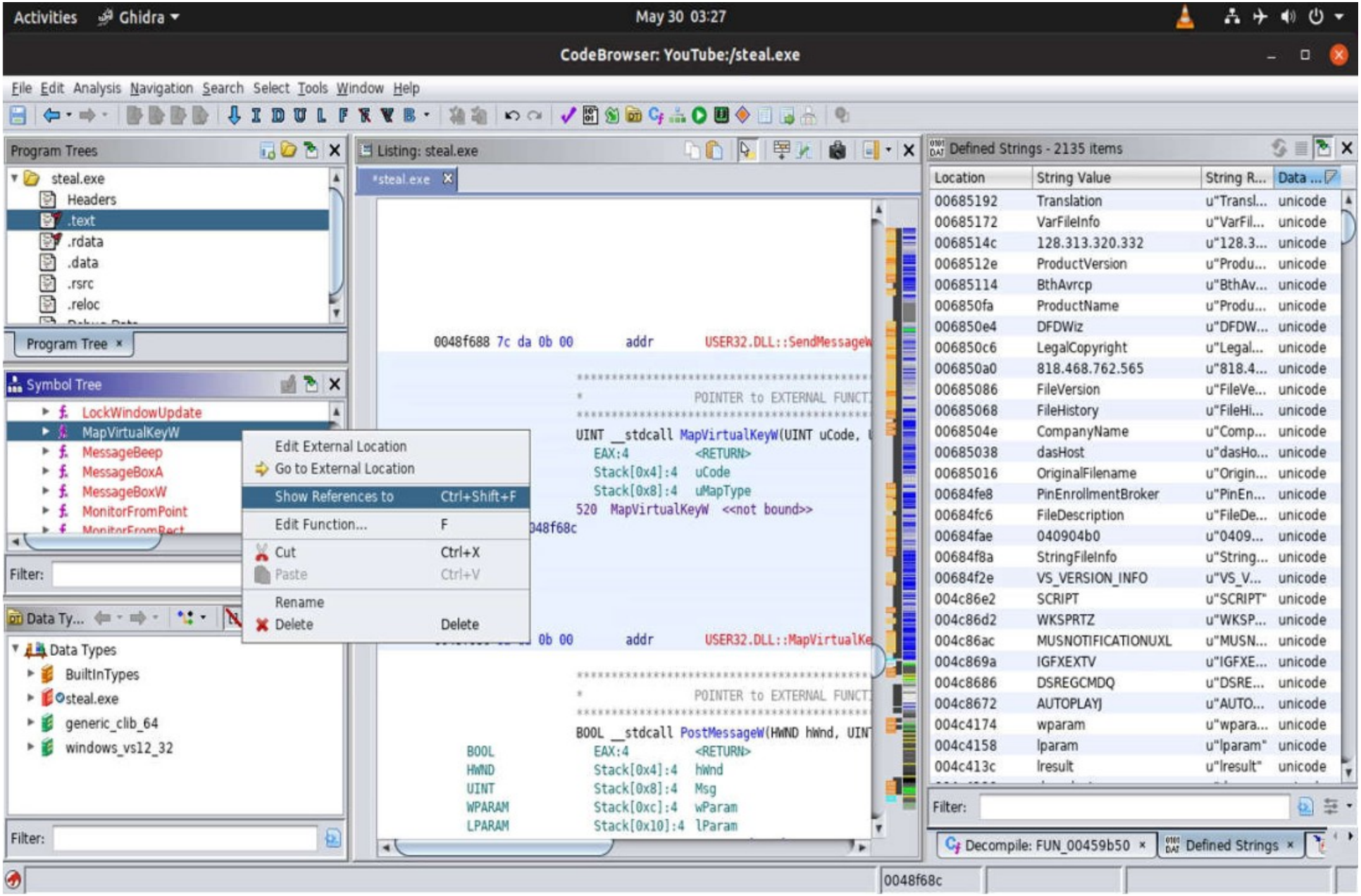
Should the middle panel highlight anything using the red color, that would mean that an encrypted resource was found, which is usually a signal of malicious activity.

Analyzing A Malware With Ghidra

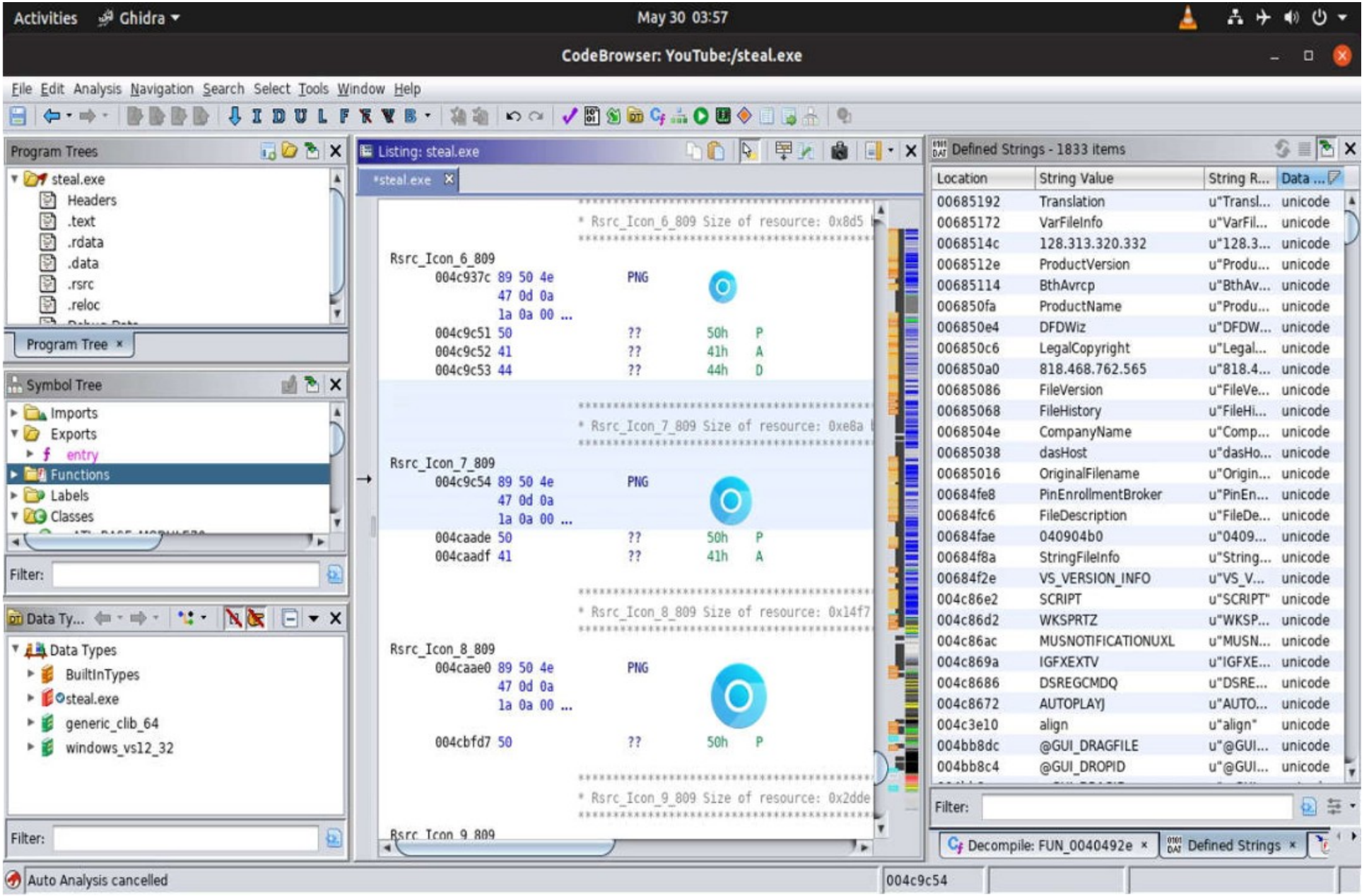
After having finished with the C program, a Windows application that was suspected to be a **malware** was added to the same Ghidra project following the procedure that was shown at the beginning of this page. Using the “Symbol Tree” window again, it was found out that the application was importing suspicious libraries such as FtpOpenFileW and FtpGetFileSize, as the following screenshot shows:


Activities
Ghidra
May 30 03:29
CodeBrowser: YouTube:/steal.exe
File Edit Analysis Navigation Search Select Tools Window Help
Program Trees
steal.exe
Headers
.text
.rdata
.data
.rsrc
.reloc
Program Tree
Symbol Tree
Imports
WININET.DLL
FtpGetFileSize
FtpOpenFileW
Filter: ftp
Data Ty...
Data Types
BuiltInTypes
steal.exe
generic_clib_64
windows_vs12_32
Filter:
Listing: steal.exe
*steal.exe
POINTER to EXTERNAL FUNCT
undefined HttpSendRequestW()
undefined AL:1 <RETURN>
94 HttpSendRequestW <<not bound>>
PTR_HttpSendRequestW_0048f79c
0048f79c 70 cd 0b 00 addr WININET.DLL::HttpSendReq
POINTER to EXTERNAL FUNCT
undefined FtpOpenFileW()
undefined AL:1 <RETURN>
53 FtpOpenFileW <<not bound>>
PTR_FtpOpenFileW_0048f7a0
0048f7a0 84 cd 0b 00 addr WININET.DLL::FtpOpenFile
POINTER to EXTERNAL FUNCT
undefined FtpGetFileSize()
undefined AL:1 <RETURN>
50 FtpGetFileSize <<not bound>>
PTR_FtpGetFileSize_0048f7a4
0048f7a4 94 cd 0b 00 addr WININET.DLL::FtpGetFileS
POINTER to EXTERNAL FUNCT
undefined InternetOpenUrlW()
undefined AL:1 <RETURN>
Decompile: FUN_00485175 - (steal.e...
undefined4 __thiscall FUN_00485175(void *this,undefined4
{
HWND hWnd;
BOOL BVar1;
HWND pHVar2;
int **ppiVar3;
int *local_28 [2];
undefined4 local_20;
undefined4 local_1c;
tagPOINT local_18;
tagPOINT local_10;
int *local_8;
local_8 = (int *)this;
hWnd = GetForegroundWindow();
FUN_0046387d(hWnd,1);
BVar1 = GetCaretPos((LPPOINT)&local_10);
if (BVar1 == 0) {
FUN_00409a20(param_2);
param_2[3] = (int *)0x1;
*param_2 = (int *)0x0;
FUN_00456aa3((void *)(*(int *)(*local_8 + 4) + (int)
}
else {
ClientToScreen(hWnd,(LPPOINT)&local_10);
pHVar2 = GetForegroundWindow();
FUN_00473c94(*(int *)((int)this + 0x108),(LPPOINT)&
local_10.x = (int *)((int)local_10.x - local_18.x);
local_10.y = (int *)((int)local_10.y - local_18.y);
ppiVar3 = (int **)FUN_0040942e(param_2);
FUN_004091b0(ppiVar3,1);
local_28[0] = local_10.x;
Decompile: FUN_00485175
Defined Strings
0048f7a0

Another library, MapVirtualKeyW, which is usually used by keyloggers, was also found. In the “Symbol Tree” window, by right clicking on the suspicious function name and then choosing "Show references to" it is possible to find which sections of code are referencing this function:


Activities
Ghidra
May 30 03:27
CodeBrowser: YouTube:/steal.exe
File Edit Analysis Navigation Search Select Tools Window Help
Program Trees
steal.exe
Headers
.text
.rdata
.data
.rsrc
.reloc
Program Tree
Symbol Tree
LockWindowUpdate
MapVirtualKeyW
MessageBeep
MessageBoxA
MessageBoxW
MonitorFromPoint
Filter:
Edit External Location
Go to External Location
Show References to Ctrl+Shift+F
Edit Function... F
Cut Ctrl+X
Paste Ctrl+V
Rename
Delete Delete
Data Ty...
Data Types
BuiltInTypes
steal.exe
generic_clib_64
windows_vs12_32
Filter:
Listing: steal.exe
*steal.exe
0048f688 7c da 0b 00 addr USER32.DLL::SendMessageW
* POINTER to EXTERNAL FUNCT
UINT __stdcall MapVirtualKeyW(UINT uCode,
EAX:4 <RETURN>
Stack[0x4]:4 uCode
Stack[0x8]:4 uMapType
520 MapVirtualKeyW <<not bound>>
048f68c
0b 00 addr USER32.DLL::MapVirtualKe
* POINTER to EXTERNAL FUNCT
BOOL __stdcall PostMessageW(HWND hWnd, UIN
BOOL EAX:4 <RETURN>
HWND Stack[0x4]:4 hWnd
UINT Stack[0x8]:4 Msg
WPARAM Stack[0xc]:4 wParam
LPARAM Stack[0x10]:4 lParam
Defined Strings - 2135 items
Location String Value String R... Data ...
00685192 Translation u"Transl... unicode
00685172 VarFileInfo u"VarFil... unicode
0068514c 128.313.320.332 u"128.3... unicode
0068512e ProductVersion u"Produ... unicode
00685114 BthAvrcp u"BthAv... unicode
006850fa ProductName u"Produ... unicode
006850e4 DFDWiz u"DFDW... unicode
006850c6 LegalCopyright u"Legal... unicode
006850a0 818.468.762.565 u"818.4... unicode
00685086 FileVersion u"FileVe... unicode
00685068 FileHistory u"FileHi... unicode
0068504e CompanyName u"Comp... unicode
00685038 dasHost u"dasHo... unicode
00685016 OriginalFilename u"Origin... unicode
00684fe8 PinEnrollmentBroker u"PinEn... unicode
00684fc6 FileDescription u"FileDe... unicode
00684fae 040904b0 u"0409... unicode
00684f8a StringFileInfo u"String... unicode
00684f2e VS_VERSION_INFO u"VS_V... unicode
004c86e2 SCRIPT u"SCRIPT" unicode
004c86d2 WKSPRTZ u"WKSP... unicode
004c86ac MUSNOTIFICATIONUXL u"MUSN... unicode
004c869a IGFXEXTV u"IGFXE... unicode
004c8686 DSREGCMDQ u"DSRE... unicode
004c8672 AUTOPLAYJ u"AUTO... unicode
004c4174 wparam u"wpara... unicode
004c4158 lparam u"lparam" unicode
004c413c lresult u"lresult" unicode
Filter:
Decompile: FUN_00459b50
Defined Strings
0048f68c

Finally, Ghidra is able to detect all **embedded resources** and they will be shown in the middle window, as the following screenshot shows:


Activities
Ghidra
May 30 03:57
CodeBrowser: YouTube:/steal.exe
File Edit Analysis Navigation Search Select Tools Window Help
Program Trees
steal.exe
Headers
.text
.rdata
.data
.rsrc
.reloc
Program Tree
Symbol Tree
Imports
Exports
entry
Functions
Labels
Classes
Filter:
Data Ty...
Data Types
BuiltInTypes
steal.exe
generic_clb_64
windows_vs12_32
Filter:
Listing: steal.exe
*steal.exe
* Rsrc_Icon_6_809 Size of resource: 0x8d5
Rsrc_Icon_6_809
004c937c 89 50 4e PNG
47 0d 0a
1a 0a 00 ...
004c9c51 50 ?? 50h P
004c9c52 41 ?? 41h A
004c9c53 44 ?? 44h D
* Rsrc_Icon_7_809 Size of resource: 0xe8a
Rsrc_Icon_7_809
004c9c54 89 50 4e PNG
47 0d 0a
1a 0a 00 ...
004caade 50 ?? 50h P
004caadf 41 ?? 41h A
* Rsrc_Icon_8_809 Size of resource: 0x14f7
Rsrc_Icon_8_809
004caae0 89 50 4e PNG
47 0d 0a
1a 0a 00 ...
004cbfd7 50 ?? 50h P
* Rsrc_Icon_9_809 Size of resource: 0x2dde
Rsrc_Icon_9_809
Defined Strings - 1833 items
Location
String Value
String R...
Data ...
00685192 Translation u"Transl... unicode
00685172 VarFileInfo u"VarFil... unicode
0068514c 128.313.320.332 u"128.3... unicode
0068512e ProductVersion u"Produ... unicode
00685114 BthAvrcp u"BthAv... unicode
006850fa ProductName u"Produ... unicode
006850e4 DFDWiz u"DFDW... unicode
006850c6 LegalCopyright u"Legal... unicode
006850a0 818.468.762.565 u"818.4... unicode
00685086 FileVersion u"FileVe... unicode
00685068 FileHistory u"FileHi... unicode
0068504e CompanyName u"Comp... unicode
00685038 dasHost u"dasHo... unicode
00685016 OriginalFilename u"Origin... unicode
00684fe8 PinEnrollmentBroker u"PinEn... unicode
00684fc6 FileDescription u"FileDe... unicode
00684fae 040904b0 u"0409... unicode
00684f8a StringFileInfo u"String... unicode
00684f2e VS_VERSION_INFO u"VS_V... unicode
004c86e2 SCRIPT u"SCRIPT" unicode
004c86d2 WKSPRTZ u"WKSP... unicode
004c86ac MUSNOTIFICATIONUXL u"MUSN... unicode
004c869a IGFXEXTV u"IGFXE... unicode
004c8686 DSREGCMDQ u"DSRE... unicode
004c8672 AUTOPLAYJ u"AUTO... unicode
004c3e10 align u"align" unicode
004bb8dc @GUI_DRAGFILE u"@GUI... unicode
004bb8c4 @GUI_DROPID u"@GUI... unicode
Filter:
Decompile: FUN_0040492e
Defined Strings
Auto Analysis cancelled
004c9c54

Automated Analysis

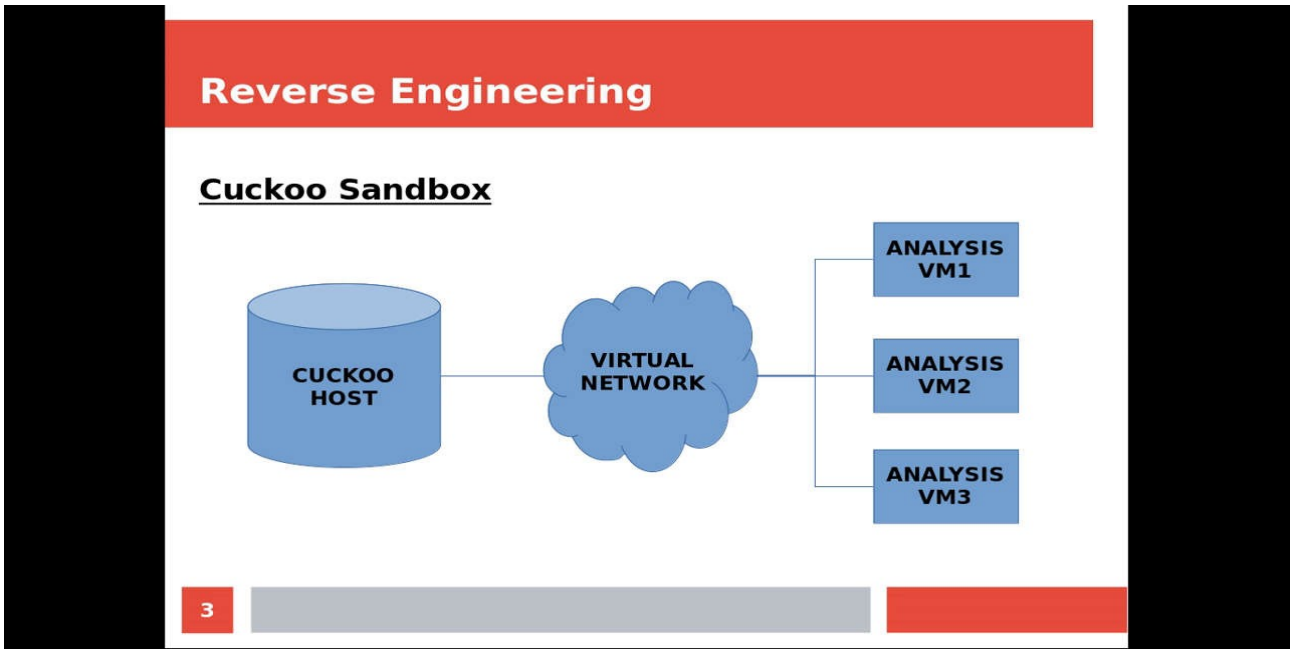
Although automated tools cannot replace experienced reverse engineers, they can still be used in order to retrieve a large amount of useful details. This chapter will explore some of tools available on the market.

Cuckoo Sandbox And Its Architecture

Cuckoo Sanbox is a very good product that all reverse engineers need to know. This package is mostly used as automated malware analysis system. Cuckoo Sandbox can:

- Monitor the behaviour of processes while they run in an **isolated environment**.
- Analyse files as well as websites under Windows, Linux, MacOS and Android virtualized environments.
- Dump and analyse network traffic with or without SSL and TSL encryption.
- Trace API calls.
- Analyse memory.

The Cuckoo Sandbox **architecture** can be quickly explained by the following drawing:



The Cuckoo host runs the main console which is used by the administrator and produces final reports.

Detecting Malware With YARA

Cuckoo Sandbox relies on YARA to automate the malware analysis. YARA is a rules based tool which is extremely helpful to reverse engineers. This tool is mainly used to identify and classify malware families:

- Rules are based on textual or binary information combined with Boolean logical operators.
- Rules can be applied to files or running processes.

YARA can scan a file, a directory or a process via its process id according to rules defined into a .yar file:

```
$> yara [rules_file.yar] [target_file]
$> yara [rules_file.yar] [target_directory]
$> yara [rules_file.yar] [process_id]
```

Engineers can either create their own rules or use the already available ones that are published on GitHub. The following is the basic format of YARA rules:

```
rule [rule_name] : [tag]
{
  condition:
    [condition_test]
}
```

The snippet below is a custom rule that says that if a file is larger than one megabyte, then it should be labelled using the rule name, which is BiggerThan1Mb:

```
import "pe"
rule BiggerThan1Mb : All
{
  condition:
    filesize > 1MB
}
```

Also, please notice the import section that forces YARA into including the pe module that is necessary to work with Windows executable files. The next rule can be appended right after the end of the previous one, in order to label all 32-bit Windows applications as i386:

```
rule i386 : Windows
{
  condition:
    pe.machine == pe.MACHINE_I386
}
```

The following rule is going to label all applications that use a GUI using the label GUI:

```
rule GUI : Windows
{
  condition:
    pe.subsystem == pe.SUBSYSTEM_WINDOWS_GUI
}
```

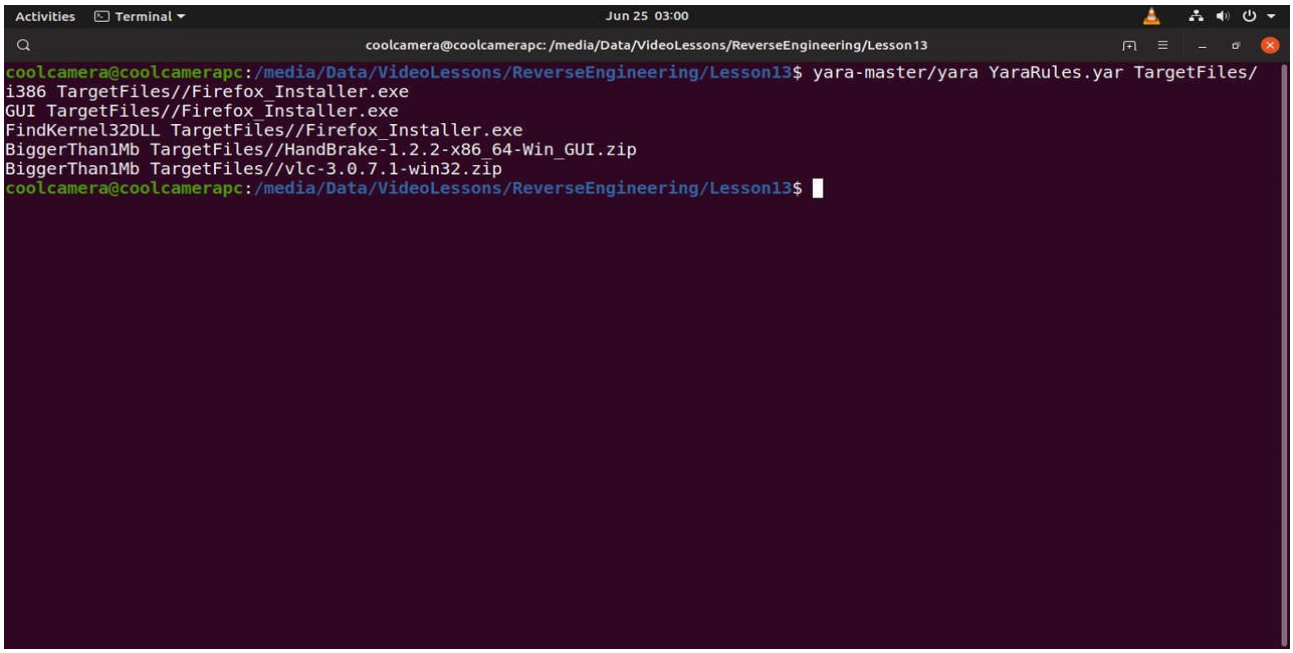
The following rule allows engineers to analyze packages entry point, looking for patterns that might indicate malicious activity:

```
rule ZeroAtEntryPoint : Windows
{
  strings:
    $myString = { E8 00 00 00 00 }
  condition:
    $myString at pe.entry_point
}
```

Finally, the next rule performs a case insensitive search of the string “kernel32.dll”:

```
rule FindKernel32DLL : Windows
{
  strings:
    $myString = "kernel32.dll" nocase
  condition:
    $myString
}
```

These rules were included into a .yar file and then a YARA session was launched targeting a local directory. The result was the following:



The first line says that the Firefox installer is a i386 application, it is a visual application and it is including the “kernel32.dll” string. Moreover, two files were found being bigger than one megabyte.

Password Cracking

Please read carefully the following before attemping any of the techniques described in this page:

- This material was created for educational purposes only
- As already stated in the first lesson, you must always get proper written authorisation to decompile or disassemble any software
- Do not try guessing someone else' password, only work on your own
- Do not use the knowledge gained by watching this video to break any law, including IP law
- If you keep on reading you are agreeing with the mentioned terms and conditions

Creating A Passwords Manager Software To Analyze

During this class the reader will learn how to reverse engineer a program, in order to find out its passwords validation algorithm. The following is a simple C program that validates passwods according to the following constraints:

- Passwords must be 10 characters long.
- The sum of the ASCII code of all characters of the password must be equal to 1085.

The source code of the program is the following:

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

char myPassword[2048];
int passLen, checksum = 0;

void getPassword() {
  while(1) {
    printf("Password: ");
    scanf("%s", myPassword);
    passLen = strlen(myPassword);

    if(passLen == 10 {
      for(int i = 0; i < passLen; i++) checksum += myPassword[i];
      if (checksum == 1085) {
        printf("Ok\n");
        break;
      }
      else printf("Try again\n");
    }
  }
}
int main(int argc, char *argv[]) {
```

```
  getPassword();

  return 0;
}
```

On Linux systems, assuming this program was saved as PasswordProgram.c, it should be quickly compiled without including debugging symbols: those are only seldom shipped with production versions of software packages:

```
$> gcc -o PasswordProgram PasswordProgram.c
```

At this stage the user should familiarize with the application by trying different passwords in order to fully understand how the application works.

Start Analyzing The Passwords Manager

Next, the user should launch the following command, to extract and study the read-only data segment of the application:

```
$> objdump -s -j .rodata PasswordProgram
```

The output reveals the presence of at least three strings:

- Password:
- Ok
- Try again

This would be a very important clue for reverse engineers, as they would now know that the application might return “Ok” for all correct passwords. Therefore, locating the section of the code that prints the successful message might reveal important information about the validation algorithm.

The next step would be to decompile the application. Many diferent methodologies exist, however, this course will show the easiest:

```
$> objdump -f -d -Mintel --disassemble=getPassword PasswordProgram
```

The previous command will disassemble the whole program, producing a very comprehensive output. The following is a simplified version of output of objdump and it also includes plenty of comments that can help the reader:

```
; Save RBP onto stack
PUSH RBP
; RBP points at top of stack
MOV RBP,RSP
; Make space for local variables
```

```
SUB RSP,0x10
; RDI = "Password: "
LEA RDI,[RIP+0xE90]
; Clean EAX, will contain return value
MOV EAX,0x0
; Execute printf
CALL 0x1050 <printf@plt>
; Prepare myPassword for scanf
LEA RSI,[RIP+0x2EDB]
; Prepare %s for scanf
LEA RDI,[RIP+0xE83]
; Clean EAX, will contain return value
MOV EAX,0x0
; Execute scanf
CALL 0x1060 <__isoc99_scanf@plt>
; Prepare user answer for strlen
LEA RDI,[RIP+0x2EC3]
; Execute  strlen
CALL 0x1040 <strlen@plt>
; Save value return value
MOV DWORD PTR [RIP+0x2E98],EAX
; Copy lenght of user string into EAX
MOV EAX,DWORD PTR [RIP+0x2E92]
; Compare answer with 10, if yes..
CMP EAX,0xA
; Move to address getPassword + 8 (116d)
JNE 0x116D <getPassword+8>
; For cycle starts, initialize "i"
MOV DWORD PTR [RBP-0x4],0x0
; Go to 124 and grab strlen of user answer
JMP 0x11E1 <getPassword+124>
; Push value of "i" into EAX
MOV EAX,DWORD PTR [RBP-0x4]
; Convert doubleword to quadword
CDQE
; RDX points at user typed password (array)
LEA RDX,[RIP+0x2E98]
; Grab a char from typed password
MOVZX EAX,BYTE PTR [RAX+RDX*1]
; Copy user char into EDX (lower part of EAX)
MOVSX EDX,AL
; Copy current value of checksum into EAX
MOV EAX,DWORD PTR [RIP+0x2E4F]
; Calculate checksum
ADD EAX,EDX
; Save checksum
```

```
MOV DWORD PTR [RIP+0x2E47],EAX
; Increase index
ADD DWORD PTR [RBP-0x4],0x1
; Save strlen of user answer
MOV EAX,DWORD PTR [RIP+0x2E59]
; Compare strlen to "i", if not end of the string...
CMP DWORD PTR [RBP-0x4],EAX
; Jump to address getPassword + 87
JL 0x11BC <getPassword+87>
; Move checksum into EAX
MOV EAX,DWORD PTR [RIP+0x2E32]
; If checksum is not 1085 (0x43d) then...
CMP EAX,0x43D
; Go to address getPassword + 162
JNE 0x1207 <getPassword+162>
; RDI has to point at the string "Ok"
LEA RDI,[RIP+0xE12]
; Print "Ok"
CALL 0x1030 <puts@plt>
; Go to address getPassword +179
JMP 0x1218 <getPassword+179>
LEA RDI,[RIP+0xE07]
CALL 0x1030 <puts@plt>
JMP 0x116D <getPassword+8>
; Do nothing
NOP
; Destroy stack frame for function getPassword
LEAVE
; Return to caller
RET
```

By having a look at the assembly, the reverse engineer should find out that the instruction that prints “Ok” is located at line 70. Therefore, one will need to scroll up in order to locate the instruction that is triggering this message. Should the test at line 64 fails, the program will jump and ask the user to type another password. Therefore, the test located at line 64 has to succeed in order for the password to be accepted. The reverse engineer will keep on scrolling up until the entire validation algorithm has been understood.

Code Obfuscation

This chapter analyzes techniques that can be used to make the machine code more difficult to read and understand.

Using The Stack To Obfuscate

The **stack** can be used to make the reverse engineering process more difficult:

- Data can be pushed into the stack instead of being initialised in the .data section, in order to hide as much information as possible.
- Assembly opcodes and operands can also be stored into the stack from where they can be retrieved and executed by the program, which makes reverse engineering more difficult.

Bear in mind that modern operating systems might disable execution of code stored into the stack, which then can only be used to store data.

Using The Heap To Obfuscate

The **heap** can also be used to store data and code for later execution. To achieve this goal, heap space will need to be created first: in both Linux and Windows, space on the heap is created by using the malloc() function which will work as follows:

- The dynamic space where to store the code is created on the heap.
- The pointer to the heap space may be created on the stack.

Using The Hexadecimal Format To Obfuscate

Storing data in the **hexadecimal** format helps in making data more difficult to read and be spotted. However, although this technique might create issues to reverse engineers, it is definitely not going to affect the work of automated tools.

Using Encryption To Obfuscate

Encryption is also used to make the process of reverse engineering more difficult. More in particular, the XOR operation is widely used for this purpose as it is very simple to implement: the entire program can be encrypted and decrypted using the same algorithm:

1	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**→**	**Code**
1	**0**	**1**	**1**	**1**	**0**	**1**	**1**	**→**	**Key**
0	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**→**	**Result**

This approach is considered to be weak if used by itself, but stronger if mixed with other technologies. Also, as XOR keys need to be strong, they must be:

- Randomly generated.

- As long as the code to be encrypted.

Code can be encrypted using any of the available algorithms, although none of them will be covered here due to the complexity involved in this topic, however… The reader should never forget that the entry function of any encrypted program MUST be able to understand the code that is to be executed! Therefore, encrypted programs, also need to be able to decrypt themselves.

Moreover, as static analysis can only reveal API functions that are present in the import table, but many more can be loaded from the code by using LoadLibrary and GetProcAddress on Windows platforms or dlopen and dlsym on Linux systems, API functions names can be encrypted and loaded when needed (**dynamic library loading**), in order to prevent tools such as "strings" from revealing their presence.

Using Loop Cycles To Encrypt And Decrypt Programs

Loop cycles controlled by conditional jumps can be used in order to equip a program with the ability to encrypt and decrypt itself. These cycles often point at the memory location or label where the code that has to be encrypted or decrypted is located. The following code is an example of this scenario:

```
; Address of the target code
MOV ESI, 0x40201800
MOV ECX, 0x20
LoopLabel:
MOV AL, [ESI]
; Encrypting with simple math
ADD AL, 0x40
MOV [ESI], AL
INC ESI
DEC ECX
JNZ LoopLabel
```

Control Flow Flattening

Control flow flattening is a technique by which sets of conditional jumps are added where they would not be needed, which makes even simple programs difficult to read.

Garbage Code Insertion

Garbage code insertion is a technique by which lines of code that do absolutely nothing are added to the program, which makes even simple algorithms difficult to read.

Metamorphism

Metamorphism is a technique by which simple sequences of instructions are replaced with similar ones able to accomplish the same task, although more confusing to read. For example, the assembly instruction “ROL R8D, 7” can be rewritten as:

```
PUSH EBX
MOV EBX, R8D
SHL R8D, 7
SHR EBX, 25
OR R8D, EBX
POP EBX
```

Metamorphism makes malwares difficult to detect as the signatures that are stored into the antivirus database will not match any more.

Using Packers And Protectors To Obfuscate

Packers compress executable files, which makes the code difficult to reverse engineer. Self-extracting archives are compressed packages that contain the relevant decompression code. Packers might be able to modify the ELF and PE sections, making compressed files unreadable to programs such as readelf or readpe. **Protectors** instead, can compress and encrypt files often using several layers of encryption and compression.

Using MIME To Obfuscate

The internet SMTP protocol was designed to exchange text only messages, but as one needed to send binary files too, in 1996 the **MIME** standard was created. This protocol allows users to attach binary files to their emails by converting binary into 7-bit ASCII text using Base64 encoding. Modern servers do support 8-bit ASCII text via the 8BITMIME extension. Therefore, using this standard, it is possible to create packages that look like simple text files, although they might encapsulate a malware.

How To Make Software More Difficult To Read

The following section will present a list of of recommendations:

- When possible, programmers should always use relative or indirect addressing, as this might confuse reverse engineers:

```
;segment:[address or offset + index * multiplier]
CS:[BX + SI + 15]
```

- Programmers should use JNC instead of JLE, JGE, etc… as this instruction jumps when the carry flag register (CF) is not set. CF would be set, for example, if the result of an addition was bigger than the capacity of the register, or when a big number was subtracted from a smaller one. JNC jumps make the code more difficult to read.

- Programmers should always try including binary instructions into the assembly program, as these are more difficult to read.
- Programmers should always assume that any working program can be reverse engineered. While it is easy to confuse an engineer, it is much harder to stop an automated reverser.

Some Examples Of Code Obfuscation

The reader will have now the chance to put the theory into practice with the following examples. The next assembly program builds the string “I like pepperoni pizza” onto stack, only by using hexadecimal and without using the .data section:

```
SECTION .text
  GLOBAL _start
_start:
  MOV DWORD [RSP-4], 617A7A69H
  MOV DWORD [RSP-8], 7020696EH
  MOV DWORD [RSP-12], 6F726570H
  MOV DWORD [RSP-16], 70657020H
  MOV DWORD [RSP-20], 656B696CH
  MOV DWORD [RSP-24], 2049H
  MOV EAX, 1
  MOV EDI, EAX
  LEA RSI, [RSP-24]
  MOV EDX, 24
  SYSCALL

  MOV RAX, 60
  MOV RDI, 0
  SYSCALL
```

The reader should build this example and then analyze the executable by running the following command:

```
$> objdump -Mintel -s -j .data -d [file_name]
```

The output is likely to be something like the following:

objdump: section '.data' mentioned in a -j option, but not found in any input file

The next assembly program shows how to use the heap to store and encrypt data and code:

```
SECTION .text
  GLOBAL _start
_start:
  ; Use "brk" to get the beginning of heap
  MOV RAX, 12
```

```
    MOV RDI, 0
    SYSCALL
    ; Save location beginning of heap
    MOV [curr_addr], RAX
    MOV [start_addr], RAX
    ; Use "brk" to increase heap size
    MOV RAX, 12
    MOV RDI, [curr_addr]
    ; Add 10000 to beginning of heap
    ADD RDI, 100000
    SYSCALL
    ; Write someting at the end of heap
    MOV [curr_addr], RAX
    SUB RAX, 8
    ; In DDD console: x/dw $rax
    MOV QWORD [RAX], 12345678
    ; Faster than -->  mov rcx, 0
    XOR RCX, RCX
encryptionLoop:
    XOR BYTE [RCX+RAX], 0x2
    INC RCX
    CMP RCX, 4
    JL short encryptionLoop
    NOP
    MOV RAX, 12
    MOV RDI, [start_addr]
    SYSCALL

    MOV RAX, 60
    MOV RDI, 0
    SYSCALL

SECTION .data
    start_addr DQ 0
    curr_addr DQ 0
```

In order to increase the heap size, the program needs to call brk twice:

- The first call returns the address of the beginning of the heap.
- The second call adds the amount of space requested by the user.

To make sure the program is actualy increasing the heap, the reader can launch the following shell line, making sure to replace [proc_id] with the actual process id of the application:

```
$> fgrep '[heap]' /proc/[proc_id]/maps
```

Anti Reverse Engineering Techniques

Anti debugging techniques aim at preventing the program from running in debug mode, in order to make reverse engineering process more difficult. This also makes code obfuscation more effective, as the engineer would not be able to navigate the code with the debugger.

Anti Reverse Engineering Techniques On Linux

On Linux platforms, as debuggers such as gdb and strace rely upon the function ptrace() that can only be called by one process at a time, programmers can have the code call ptrace() internally, in order to prevent users from opening additional debugging sessions.

On Linux systems, the following file contains a list of properties that describe the process identified by [pid]:

- /proc/[pid]/status

If the process identified by [pid] is being traced, “State” and “TracerPID” will be set accordingly. Monitoring these fields does not produce reliable results as their behaviour can be controlled and modified using several techniques.

Anti Reverse Engineering Techniques On Microsoft Windows

On Windows machines, IsDebuggerPresent is a Kernel32 API function that returns one if the program is being debugged. Programmers can protect their software by having the code checking on this value, killing the execution should the API return one. The following assembly lines explain how a programmer can implement this scenario:

```
XOR EAX, EAX
CALL IsDebuggerPresent
TEST EAX, EAX
JNZ exitNow
```

However, techniques based upon IsDebuggerPresent can be easily circumvented by opportunely setting breakpoints and by manually resetting the EAX register, which prevents the code from jumpimg.

In the WindowsNT operating system family, the Process Environment Block (PEB) structure is strongly linked to the EPROCESS one, which is the process object for the process. PEB contains structures that describe processes such as startup parameters, the program image base address and much more… PEB also contains two fields that can help in identifying whether a process is being debugged:

- BeingDebugged is set to 1 if debug is running and found at offset 0x02.
- GlobalNTFlag is set to 0x70 if debug is running and found at offset 0x68.

These two values should be used as follows:

- Store FS:[18] into EAX to grab the TEB address.
- Store DS:[EAX+0x30] into EAX to grab the PEB address.
- Store DS:[EAX+0x68] into EAX to retrieve GlobalNTFlag.
- Compare EAX with 0x70 to check whether debug is on.
- Copy ZF value into AL which should be 0 or 1.

Exceptions can be used to confuse reverse engineers by:

- Accessing inaccessible memory spaces.
- Dividing by zero.
- Using INT 0x01 and INT 0x03 instructions.

Using The Stack To Confuse The Reverse Engineer

The stack can be manipulated to confuse the engineer and the reverser, by faking a sequence of instructions that mimic function calling conventions where no actual function is being called. For example, this can be achieved by opportunely using the instructions PUSH, POP and RET.

Programs Able To Detect If They Are Run In Debug Mode

Moreover, a program might be able to figure out whether is being run into a debugger by looking at the execution time: any line of code that takes longer than expected to complete might indicate that the program is being debugged. Again, the programmer can protect packages by stopping a program that takes longer than expected to complete its tasks. For example, the reader should test the following program passing different values to CMP at line 21:

```
SECTION .text
  GLOBAL _start
_start:
  ; Get the current timestamp
  RDTSC
  ; Save timestamp into ebx
  MOV EBX, EAX
  ; Next 8 lines of code do nothing
  MOV R8D, 1
  MOV R8D, 2
  MOV R8D, 3
  MOV R8D, 4
  MOV R8D, 5
  MOV R8D, 6
  MOV R8D, 7
  MOV R8D, 8
  ; Get final timestamp
```

```
  RDTSC
  ; Calculate how long it took
  SUB EAX, EBX
  CMP EAX, 0x1000000
  JG debugIsOn
  JMP terminate
debugIsOn:
  MOV RAX, 1
  MOV RDI, 1
  MOV RSI, debugMsg
  MOV RDX, msgLen
  SYSCALL
terminate:
  MOV RAX, 60
  MOV RDI, 0
  SYSCALL

SECTION .data
  debugMsg DB 'Debug might be on'
  msgLen EQU $ - debugMsg
```

Preventing Programs From Running In Virtual Environments

Execution of software packages in virtual environments might be disabled by programmers for several reasons, including preventing users from breaching the license agreement and preventing reverse engineers from running a debugger. As the assembly instruction CPUID returns family, model and brand string for the processor, it can be used to figure out whether the program is running on a real CPU or a hypervisor. However, bear in mind that some hypervisors allow system administrators to hide or change values returned by CPUID by simply editing a configuration file. The following assembly program makes decisions according to specific CPUID values:

```
SECTION .text
  GLOBAL _start
_start:
  MOV EAX, 1
  CPUID
  BT ECX, 31
  JC VMYes
  NOP
  NOP
VMYes:
  NOP

  MOV RAX, 60
  MOV RDI, 0
  SYSCALL
```

Moreover, programmers can prevent the execution in virtual environments by terminating programs when well known hosted hypervisors services are found to be running. For example, the following services should all be monitored:

- vmtoolsd.
- vmacthlp.
- vmwaretray.
- vmwareuser.
- VGAuthService.
- vboxservice.
- vboxtray.
- vboxcontrol.
- qemu-ga.
- vmsrvc.
- vmusrvc.
- Many more…

Programmers might also want to scan the file system in order to look for directories and files that are known to be created by hosted hypervisors:

- system32\drivers\VBoxGuest.sys
- %programfiles%\VMWare
- Many more…

The Windows registry can also be scanned in order to detect any hosted hypervisor installation:

- HKLM\HARDWARE\ACPI\DSDT\VBOX__
- HKLM\HARDWARE\ACPI\FADT\VBOX__
- HKLM\HARDWARE\ACPI\RSDT\VBOX__
- HKLM\SOFTWARE\Oracle\VirtualBox Guest Additions
- Many more…

Moreover, programmers may also want to consider the following information and terminate the execution of the program at any time:

- Small amount of memory might indicate that a program is running into a virtual machine.
- Single processor machines are usually virtual machines.
- Small disk size might indicate that the program is running into a virtual machine.
- Small screen size might indicate that the program is running into a virtual machine.
- Is any debugger or reverser process running?

Modifying Files Structures To Prevent Reverse Engineering

Finally, a programmer should always:

- Implement any strategy able to make memory dumps appear confusing to engineers and automated tools.
- Modify PE and ELF sections so that their properties return wrong values.
- Modify PEB sections so that its structures return wrong values.