

Getting Started with Linux Reverse Engineering [UPDATED]

by offen5ive | 16-06-2022

What is Reverse Engineering?

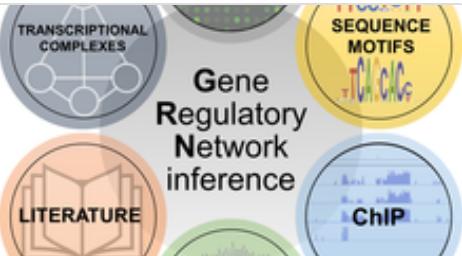
In simple words Reverse Engineering is the Process of taking compiled machine code and retrieving it's code into readable form. The main goal of reverse engineering is to understand how the program is working.

For further reference -

Reverse engineering - Wikipedia

Reverse engineering (also known as backwards engineering or back engineering) is a process or method through the application of which one attempts to understand through deductive reasoning how

W https://en.wikipedia.org/wiki/Reverse_engineering



What is ELF (Executable and Linkable Format)?

The header file `<elf.h>` defines the format of ELF executable binary files. Amongst these files are normal executable files, relocatable object files, core files, and shared objects.

An executable file using the ELF file format consists of an ELF header, followed by a program header table or a section header table, or both. The ELF header is always at offset zero of the file. The program header table and the section header table's offset in the file are defined in the ELF header. The two tables describe the rest of the particularities of the file.

This header file describes the above mentioned headers as C structures and also includes structures for dynamic sections, relocation sections and symbol tables.

Ref -

<p>elf(5) - Linux manual page</p> <p>The header file defines the format of ELF executable binary files. Amongst these files are normal executable files, relocatable object files, core files, and shared objects. An executable file using the</p> <p>https://man7.org/linux/man-pages/man5/elf.5.html</p>	<p>Linux and ELF System Programming Handbook</p> <p>Michael Kerrisk</p> 
---	---

The x86 Assembly Overview

Assembly is the most popular low level language. The program written in high level language, for example C, The code is compiled into machine code that the CPU can understand.

The Stack

A is an abstract data structure which consists of information in a Last In First Out system. You put arbitrary objects onto the stack and then you take them off again, much like an in/out tray, the top item is always the one that is taken off and you always put on to the top.

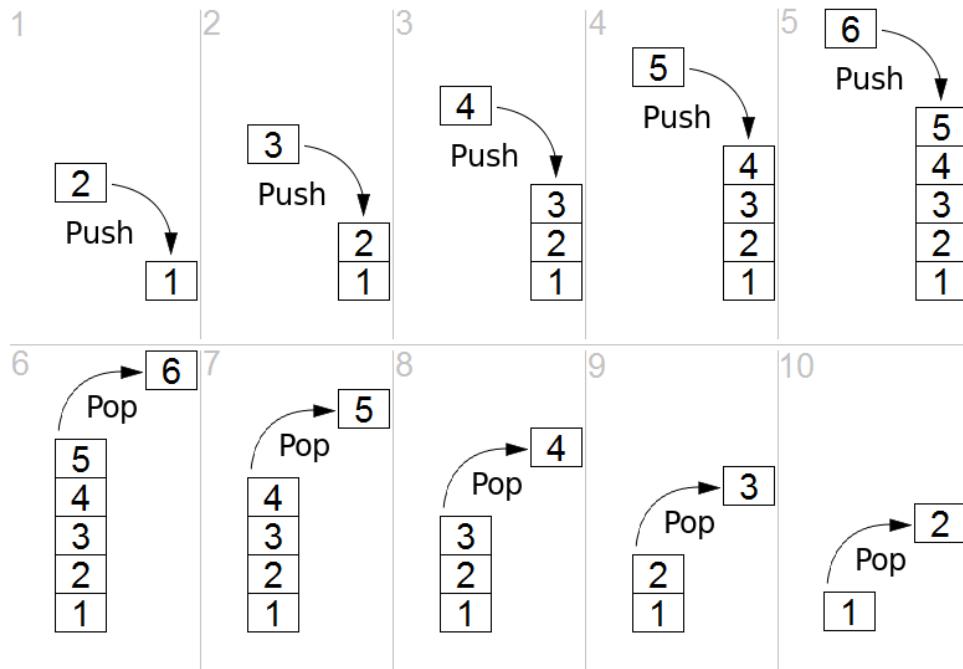
A programs stack isn't generally hardware (though it's kept in memory so it can be argued as such), but the Stack Pointer which points to a current area of the Stack is generally a CPU register. This makes it a bit more flexible than a LIFO (Last in First Out) stack as you can change the point at which the stack is addressing.

For example -

PUSH BL / push BL onto the stack and subtract one from the stack pointer.

POP BL

/ Add one to the stack pointer and POP BL from the stack.



This image makes it easier to understand the Stack last in first out functionality with assembly instructions.

The Registers in x86 Architectures

The register is a small bit of memory inside the CPU used to store and transfer the data and instructions that are being used by the CPU.

- General x86 Registers - EAX, EBX, ECX, EDX
- Index and Pointers - ESI, EDI, EBP, EIP, ESP
- Indicator - EFLAGS

Instructions in Assembly

- “MOV”: move data from one operand into another
- “ADD/SUB/MUL/DIV”: Add, Subtract, Multiply, Divide one operand with another and store the result in a register
- “AND/OR/XOR/NOT/NEG”: Perform logical and/or/xor/not/negate operations on the operand
- “SHL/SHR”: Shift Left/Shift Right the bits in the operand
- “CMP/TEST”: Compare one register with an operand
- “JMP/JZ/JNZ/JB/JS/etc.”: Jump to another instruction (Jump unconditionally, Jump if Zero, Jump if Not Zero, Jump if Below, Jump if Sign, etc.)
- “PUSH/POP”: Push an operand into the stack, or pop a value from the stack into a register
- “CALL”: Call a function. This is the equivalent of doing a “PUSH %EIP+4” + “JMP”. I’ll get into calling conventions later..
- “RET”: Return from a function. This is the equivalent of doing a “POP %EIP”

Debugger - GDB

To learn GDB (GNU Project Debugger) we will run simple hello_world program binary.

The C hello_world Program -

```
#include <stdio.h>
int main() {
printf("Hello World!");
return 0;
}
```

Compiling C program to obtain Executable

```
| gcc <example.c> -o <output filename>
```

→

```
| gcc hello_world.c -o hello_world
```

Running the Binary with GDB

```
(offensive@kali)-[~]
$ gdb hello_world
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef` to start, `gef config` to configure
96 commands loaded for GDB 10.1.90.20210103-git using Python engine 3.9
Reading symbols from hello_world...
(No debugging symbols found in hello_world)
gef> r
Starting program: /home/offensive/hello_world
Hello World![Inferior 1 (process 28714) exited normally]
gef> 
```

Setting Breakpoints

Breakpoints are set to the program for GDB to stop execution and examine the contents of the stack.

For example setting breakpoints to the main function would be

```
| break main
```

or

```
| b main
```

or you can use address of the function to set break point.

```
gef> break main  
Breakpoint 1 at 0x1139  
gef> [REDACTED]
```

Here we have put the breakpoints to the main function now to enter debugging mode we can run the program with 'r'.

```

[ Legend: Modified register | Code | Heap | Stack | String ]

```

registers

```

$rax : 0x000055555555135 → <main+0> push rbp
$rbx : 0x0
$rcx : 0x00007fffffa2718 → 0x00007fffffa4b00 → 0x0000000000000000
$rdx : 0x00007fffffffdf88 → 0x00007fffffff2ec → "TERMINATOR_DBUS_NAME=net.tenshu.Terminator23558193[...]"
$rsp : 0x00007fffffffde80 → 0x000055555555160 → <__libc_csu_init+0> push r15
$rbp : 0x00007fffffffde80 → 0x000055555555160 → <__libc_csu_init+0> push r15
$rsi : 0x00007fffffffdf78 → 0x00007fffffff2d0 → "/home/offensive/hello_world"
$rdi : 0x1
$rip : 0x000055555555139 → <main+4> lea rdi, [rip+0xec4]      # 0x555555556004
$r8 : 0x0
$r9 : 0x00007fffffe21b0 → <_dl_fini+0> push rbp
$r10 : 0x0
$r11 : 0xc2
$r12 : 0x0000555555555050 → <_start+0> xor ebp, ebp
$r13 : 0x0
$r14 : 0x0
$r15 : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000

```

stack

```

0x00007fffffffde80 +0x0000: 0x000055555555160 → <__libc_csu_init+0> push r15 ← $rsp, $rbp
0x00007fffffffde88 +0x0008: 0x00007ffff7e0ad0a → <__libc_start_main+234> mov edi, eax
0x00007fffffffde90 +0x0010: 0x00007fffffffdf78 → 0x00007fffffff2d0 → "/home/offensive/hello_world"
0x00007fffffffde98 +0x0018: 0x0000000010000000
0x00007fffffffdea0 +0x0020: 0x000055555555135 → <main+0> push rbp
0x00007fffffffdea8 +0x0028: 0x00007ffff7e0a7cf → <init_cacheinfo+287> mov rbp, rax
0x00007fffffffdeb0 +0x0030: 0x0000000000000000
0x00007fffffffdeb8 +0x0038: 0xd5838541307da7e8

```

code:x86:64

```

0x5555555555130 <frame_dummy+0> jmp   0x55555555550b0 <register_tm_clones>
0x5555555555135 <main+0>    push   rbp
0x5555555555136 <main+1>    mov    rbp, rsp
→ 0x5555555555139 <main+4>    lea    rdi, [rip+0xec4]      # 0x555555556004
0x5555555555140 <main+11>   mov    eax, 0x0
0x5555555555145 <main+16>   call   0x5555555555030 <printf@plt>
0x555555555514a <main+21>   mov    eax, 0x0
0x555555555514f <main+26>   pop    rbp
0x5555555555150 <main+27>   ret

```

threads

```

[#0] Id 1, Name: "hello_world", stopped 0x55555555139 in main (), reason: BREAKPOINT

```

trace

```

[#0] 0x55555555139 → main()

```

gef> █

after running the program we are in the debugging mode.

To check all the breakpoints we can use use,

info breakpoints

or 'info b' or 'i b'

```
[#0] 0x5555555555139 → main()

gef> info b
Num      Type          Disp Enb Address           What
1        breakpoint    keep y  0x000055555555139 <main+4>
               breakpoint already hit 1 time
gef>
```

we can see the break point we just set to the main functions.

we can delete the breakpoints with

del and the num of the break point that is 1 in the image above so

```
gef> info b
Num      Type          Disp Enb Address           What
1        breakpoint    keep y  0x000055555555139 <main+4>
               breakpoint already hit 1 time
gef> del 1
gef> info b
No breakpoints or watchpoints.
gef>
```

Analyzing the Binary

So we will take simple hello world C program binary.

```
#include<stdio.h>
int main()
{
    char string[] = "Hello World";
    puts(string);
    return 0;
}
```

Running Binary with GDB and setting breakpoints to the main function.

```

__(offensive㉿kali)-[~]
└$ gdb hello_world
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef` to start, `gef config` to configure
96 commands loaded for GDB 10.1.90.20210103-git using Python engine 3.9
Reading symbols from hello_world...
(No debugging symbols found in hello_world)
gef> disass main
Dump of assembler code for function main:
0x000001199 <+0>:    lea    ecx,[esp+0x4]
0x00000119d <+4>:    and    esp,0xffffffff0
0x0000011a0 <+7>:    push   DWORD PTR [ecx-0x4]
0x0000011a3 <+10>:   push   ebp
0x0000011a4 <+11>:   mov    ebp,esp
0x0000011a6 <+13>:   push   ebx
0x0000011a7 <+14>:   push   ecx
0x0000011a8 <+15>:   sub    esp,0x10
0x0000011ab <+18>:   call   0x11ea <__x86.get_pc_thunk.ax>
0x0000011b0 <+23>:   add    eax,0x2e50
0x0000011b5 <+28>:   mov    DWORD PTR [ebp-0x14],0x6c6c6548
0x0000011bc <+35>:   mov    DWORD PTR [ebp-0x10],0x6f57206f
0x0000011c3 <+42>:   mov    DWORD PTR [ebp-0xc],0x646c72
0x0000011ca <+49>:   sub    esp,0xc
0x0000011cd <+52>:   lea    edx,[ebp-0x14]
0x0000011d0 <+55>:   push   edx
0x0000011d1 <+56>:   mov    ebx,eax
0x0000011d3 <+58>:   call   0x1030 <puts@plt>
0x0000011d8 <+63>:   add    esp,0x10
0x0000011db <+66>:   mov    eax,0x0
0x0000011e0 <+71>:   lea    esp,[ebp-0x8]
0x0000011e3 <+74>:   pop    ecx
0x0000011e4 <+75>:   pop    ebx
0x0000011e5 <+76>:   pop    ebp
0x0000011e6 <+77>:   lea    esp,[ecx-0x4]
0x0000011e9 <+80>:   ret

End of assembler dump.
gef> b main
Breakpoint 1 at 0x11a8
gef> 

```

To enter Debugging mode press r to run and press enter

```
[ Legend: Modified Register | Core | Heap | Stack | String ]
Registers
Rax : 0x00000000
Rbx : 0x0
Rcx : 0xffffffffd19c → 0x00000001
Rdx : 0xffffffffd24 → 0x00000000
Rsp : 0xffffffffd0d0 → 0xffffffffd0f0 → 0x00000001
Rbp : 0xffffffffd0d8 → 0x00000000
Rsi : 0xffffffffd0e0 → 0x00000000
Rdi : 0xffffffffd0e8 → 0x00000000
Rbpn : 0x40000000 → sub esp, 0x10
Rflags : [zero carry parity adjust SIGN trap INTERRUPT direction overflow resume virtualx86 identification]
Rcs: 0x0023 Rss: 0x0020 Rds: 0x002b Rbs: 0x0000 Rgs: 0x0063

Stack
0xffffffffd0d0: 0xffffffffd0f0 → 0x00000001 + $esp
0xffffffffd0d4: 0x00004: 0x00000000 + $rbp
0xffffffffd0d8: 0x00008: 0x00000000 ← $rbp
0xffffffffd0dc: 0x0000c: 0x770de0e4 → <_libc_start_main+262> add esp, 0x10
0xffffffffd0e0: 0x00010: 0x77f5a5000 → 0x001e4d0c
0xffffffffd0e4: 0x00014: 0x77f5a5000 → 0x001e4d0c
0xffffffffd0e8: 0x00018: 0x00000000
0xffffffffd0ec: 0x0001c: 0x770de0e40 → <_libc_start_main+262> add esp, 0x10

Code:x86:32
0x565561a4 cmain+11> mov    ebp, esp
0x565561a5 cmain+13> push   ebx
0x565561a7 cmain+14> push   ecx
→ 0x565561a8 cmain+15> call   _Z11__x86_get_pc_thunkax
0x565561a9 cmain+19> add    esp, 0x10
0x565561b0 cmain+23> add    eax, 0x2e50
0x565561b5 cmain+28> mov    DWORD PTR [ebp-0x14], 0x6c6c6548
0x565561b6 cmain+35> mov    DWORD PTR [ebp-0x10], 0x6f5f720f
0x565561c3 cmain+42> mov    DWORD PTR [ebp-0xc], 0x646c72

Threads
[#0] id 1, Name: "hello_world", stopped 0x565561a8 in main (), reason: BREAKPOINT
[#0] 0x565561a8 + main()

Trace
gef> 
```

As we can see ESP register holds the value 0xfffffd0c4

now let's examine pointer.

```
gef> x/s 0xfffffd0c4
0xfffffd0c4:      "Hello World"
gef> 
```

So we can see that it points to the Hello World....

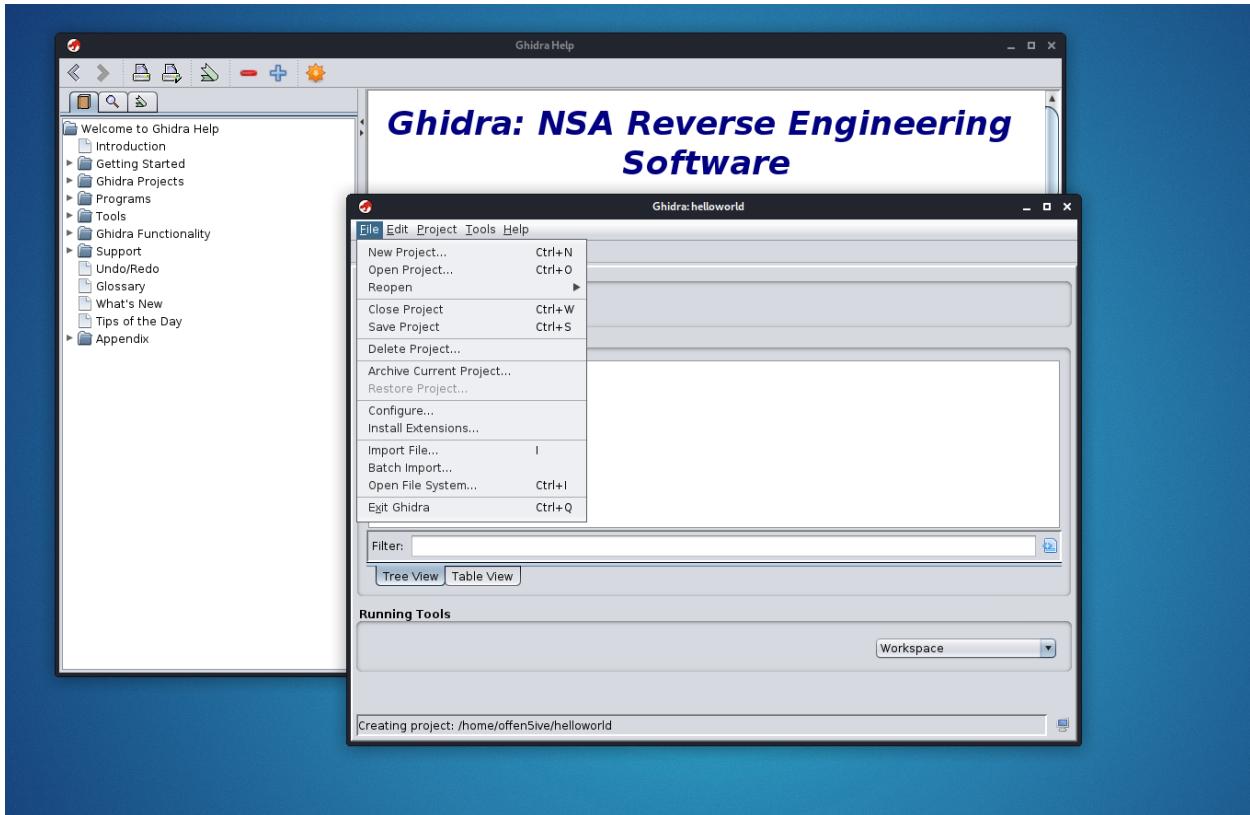
Introduction to Decompiler : Ghidra

Ghidra is a free and open source reverse engineering tool developed by the National Security Agency of the United States. The binaries were released at RSA Conference in March 2019; the sources were published one month later on GitHub. Ghidra is seen by many security researchers as a competitor to IDA Pro.

ref - <https://en.wikipedia.org/wiki/Ghidra>

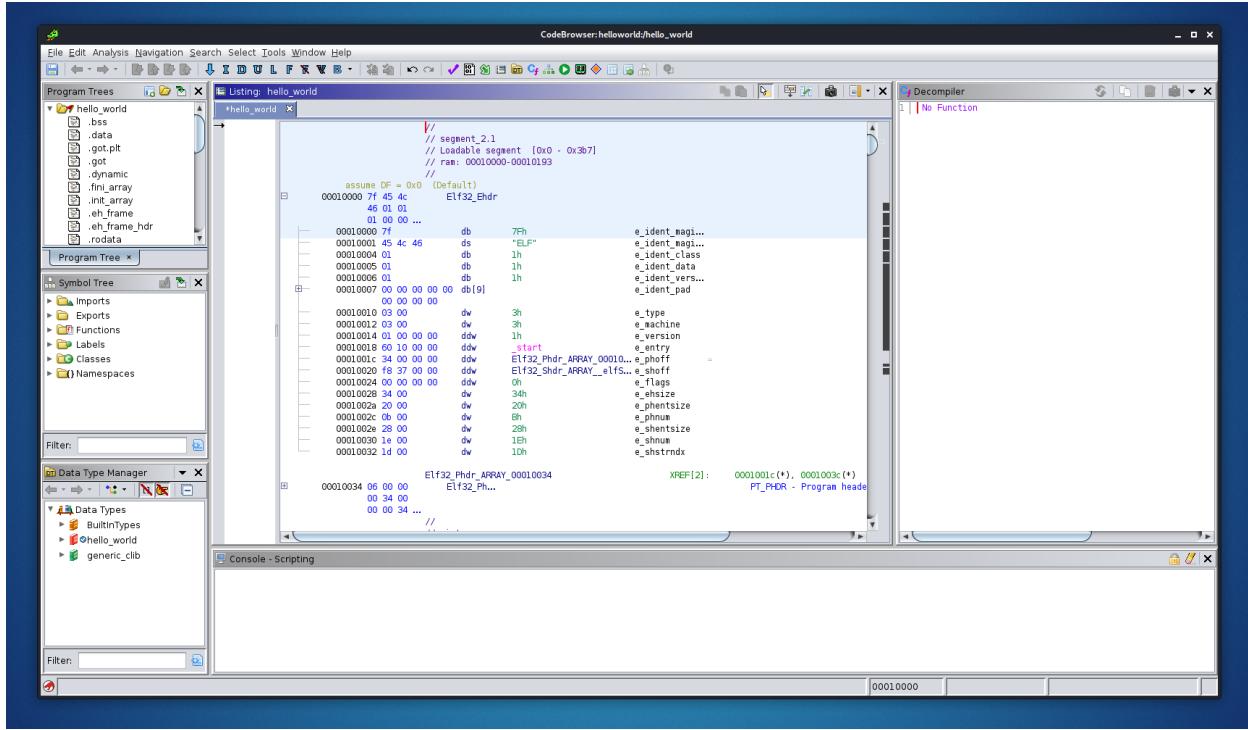
We will take the same hello world program binary and decompile the binary to obtain C program which will be somehow like the binary we compiled.

So open Up ghidra and Create new project and give it a name.



Now under Tool Chest you could see a dragon click on it and your project that you just created will can be seen under Active Projects.

Now click on file tab and open and select the Binary or drag and drop the binary anywhere on ghidra.



Click on Functions to expand it and all the functions used in the program can be seen.

Click on any function suppose main function once you click on it you'll see the decompiled program on the right most window.

Practical Reverse Engineering

So, I'll be solving a challenge from [crackmes.one](#)

So we will start with gathering some information about the binary.

```
[(offensive㉿kali)-~]
$ file crackme
crackme: ELF 64-bit LSB pie executable, x86-64, version 1 (SVSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=d82541db6b77885c07ef5b31b9f53b68088bb1c7, for GNU/Linux 3.2.0, not stripped
[(offensive㉿kali)-~]
```

now with this we know that this is 64bit ELF executable and it's a non-stripped binary.

now let's run this with GDB.

```

└─(offensive㉿kali)-[~]
$ gdb crackme
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...

GEF for linux ready, type `gef` to start, `gef config` to configure
96 commands loaded for GDB 10.1.90.20210103-git using Python engine 3.9
Reading symbols from crackme...
(No debugging symbols found in crackme)
gef>
gef> disass main
Dump of assembler code for function main:
0x000000000000001205 <+0>: push rbp
0x000000000000001206 <+1>: mov rbp,rs
0x000000000000001209 <+4>: sub rsp,0x10
0x00000000000000120d <+8>: mov DWORD PTR [rbp-0x4],edi
0x000000000000001210 <+11>: mov QWORD PTR [rbp-0x10],rsi
0x000000000000001214 <+15>: cmp DWORD PTR [rbp-0x4],0x1
0x000000000000001218 <+19>: je 0x1222 <main+29>
0x00000000000000121a <+21>: cmp DWORD PTR [rbp-0x4],0x2
0x00000000000000121e <+25>: je 0x123f <main+58>
0x000000000000001220 <+27>: jmp 0x125e <main+89>
0x000000000000001222 <+29>: mov rax,QWORD PTR [rbp-0x10]
0x000000000000001226 <+33>: mov rax,QWORD PTR [rax]
0x000000000000001229 <+36>: mov rsi,rax
0x00000000000000122c <+39>: lea rdi,[rip+0xe36]      # 0x2069
0x000000000000001233 <+46>: mov eax,0x0
0x000000000000001238 <+51>: call 0x1050 <printf@plt>
0x00000000000000123d <+56>: jmp 0x125e <main+89>
0x00000000000000123f <+58>: lea rdi,[rip+0xe38]      # 0x207e
0x000000000000001246 <+65>: call 0x1030 <puts@plt>
0x00000000000000124b <+70>: mov rax,QWORD PTR [rbp-0x10]
0x00000000000000124f <+74>: add rax,0x8
0x000000000000001253 <+78>: mov rax,QWORD PTR [rax]
0x000000000000001256 <+81>: mov rdi,rax
0x000000000000001259 <+84>: call 0x1155 <check_password>
0x00000000000000125e <+89>: mov eax,0x0
0x000000000000001263 <+94>: leave
0x000000000000001264 <+95>: ret
End of assembler dump.
gef> █

```

Disassembling the main function we can see the assembly of the binary.

```
gef> info functions
All defined functions:

Non-debugging symbols:
0x0000000000000001000  _init
0x0000000000000001030  puts@plt
0x0000000000000001040  system@plt
0x0000000000000001050  printf@plt
0x0000000000000001060  __cxa_finalize@plt
0x0000000000000001070  _start
0x00000000000000010a0  deregister_tm_clones
0x00000000000000010d0  register_tm_clones
0x0000000000000001110  __do_global_dtors_aux
0x0000000000000001150  frame_dummy
0x0000000000000001155  check_password
0x0000000000000001205  main
0x0000000000000001265  stringCompare
0x0000000000000001320  __libc_csu_init
0x0000000000000001380  __libc_csu_fini
0x0000000000000001384  _fini
gef>
```

with info functions we can see all the functions used in the program and here as we can see the "check_password" function seems interesting so let's set a break point to the function and see what's happening.

```

gef> disass check_password
Dump of assembler code for function check_password:
0x0000555555555155 <+0>:    push   rbp
0x0000555555555156 <+1>:    mov    rbp,rsp
0x0000555555555159 <+4>:    sub    rsp,0x30
0x000055555555515d <+8>:    mov    QWORD PTR [rbp-0x28],rdi
0x0000555555555161 <+12>:   mov    BYTE PTR [rbp-0x20],0x73
0x0000555555555165 <+16>:   mov    BYTE PTR [rbp-0x1f],0x75
0x0000555555555169 <+20>:   mov    BYTE PTR [rbp-0x1e],0x70
0x000055555555516d <+24>:   mov    BYTE PTR [rbp-0x1d],0x65
0x0000555555555171 <+28>:   mov    BYTE PTR [rbp-0x1c],0x72
0x0000555555555175 <+32>:   mov    BYTE PTR [rbp-0x1b],0x5f
0x0000555555555179 <+36>:   mov    BYTE PTR [rbp-0x1a],0x73
0x000055555555517d <+40>:   mov    BYTE PTR [rbp-0x19],0x65
0x0000555555555181 <+44>:   mov    BYTE PTR [rbp-0x18],0x63
0x0000555555555185 <+48>:   mov    BYTE PTR [rbp-0x17],0x72
0x0000555555555189 <+52>:   mov    BYTE PTR [rbp-0x16],0x65
0x000055555555518d <+56>:   mov    BYTE PTR [rbp-0x15],0x74
0x0000555555555191 <+60>:   mov    BYTE PTR [rbp-0x14],0x5f
0x0000555555555195 <+64>:   mov    BYTE PTR [rbp-0x13],0x70
0x0000555555555199 <+68>:   mov    BYTE PTR [rbp-0x12],0x61
0x000055555555519d <+72>:   mov    BYTE PTR [rbp-0x11],0x73
0x00005555555551a1 <+76>:   mov    BYTE PTR [rbp-0x10],0x73
0x00005555555551a5 <+80>:   mov    BYTE PTR [rbp-0xf],0x5f
0x00005555555551a9 <+84>:   mov    BYTE PTR [rbp-0xe],0x35
0x00005555555551ad <+88>:   mov    BYTE PTR [rbp-0xd],0x36
0x00005555555551b1 <+92>:   mov    BYTE PTR [rbp-0xc],0x31
0x00005555555551b5 <+96>:   mov    BYTE PTR [rbp-0xb],0x37
0x00005555555551b9 <+100>:  mov    BYTE PTR [rbp-0xa],0x38
0x00005555555551bd <+104>:  lea    rdx,[rbp-0x20]
0x00005555555551c1 <+108>:  mov    rax,QWORD PTR [rbp-0x28]
0x00005555555551c5 <+112>:  mov    rsi,rdx
0x00005555555551c8 <+115>:  mov    rdi,rax
0x00005555555551cb <+118>:  mov    eax,0x0
0x00005555555551d0 <+123>:  call   0x555555555265 <stringCompare>
0x00005555555551d5 <+128>:  mov    DWORD PTR [rbp-0x4],eax
0x00005555555551d8 <+131>:  cmp    DWORD PTR [rbp-0x4],0x1
0x00005555555551dc <+135>:  jne    0x5555555551f1 <check_password+156>
0x00005555555551de <+137>:  lea    rdi,[rip+0xe23]      # 0x5555555556008
0x00005555555551e5 <+144>:  mov    eax,0x0
0x00005555555551ea <+149>:  call   0x555555555040 <system@plt>
0x00005555555551ef <+154>:  jmp    0x555555555202 <check_password+173>
0x00005555555551f1 <+156>:  lea    rdi,[rip+0xe56]      # 0x555555555604e
0x00005555555551f8 <+163>:  mov    eax,0x0
0x00005555555551fd <+168>:  call   0x555555555050 <printf@plt>
0x0000555555555202 <+173>:  nop
0x0000555555555203 <+174>:  leave 
0x0000555555555204 <+175>:  ret
End of assembler dump.
gef> █

```

we can see that values are moved to rbp register and at address 0x00005555555551bd contents of rbp register is moved to rdx register. So viewing the string inside the rdx register will give us the flag.

So set a break point to the check_password function and run with random argument and analyze the stack.

At address "0x55555555551bd" flag will be in the stack or u can view the content of rdx register at the same address.

```
[ Legend: Modified register | Code | Heap | Stack | String ]
$rax : 0x00007fffffe362 → 0x0072727272727262 ("brrrrrr"?)
$rbx : 0x0
$rcx : 0x00007ffff7ed2f33 → 0x5577ffff0003d48 ("H=?")
$rdx : 0x00007fffffffdec0 → "super_secret_pass_56178"
$rsp : 0x00007fffffffdec0 → 0x00005555555607e → "Checking password.."
$rbp : 0x00007fffffffdef0 → 0x00007fffffffdf10 → 0x00005555555320 → <__libc_csu_init+0> push r15
$rsi : 0x00005555555592a0 → "Checking password..\n"
$rdi : 0x00007fffffff362 → 0x00727272727262 ("brrrrrr"?)
$rip : 0x0000555555551c1 → <check_password+108> mov rax, QWORD PTR [rbp-0x28]
$r8 : 0x14
$r9 : 0x00007ffff7fa2be0 → 0x0000555555596a0 → 0x0000000000000000
$r10 : 0x6e
$r11 : 0x246
$r12 : 0x0000555555555070 → <_start+0> xor ebp, ebp
$r13 : 0x0
$r14 : 0x0
$r15 : 0x0
$eflags: [zero carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
0x00007fffffffdec0 +0x0000: 0x00005555555607e → "Checking password.." ← $rsp
0x00007fffffffdec8 +0x0008: 0x00007fffffe362 → 0x00727272727262 ("brrrrrr"?)
0x00007fffffffded0 +0x0010: \super_secret_pass_56178" ← $rdx
0x00007fffffffded8 +0x0018: "cret_pass_56178"
0x00007fffffffdee0 +0x0020: 0x0038373136355f73 ("s_56178"?)
0x00007fffffffdee8 +0x0028: 0x0000000000000000
0x00007fffffffdef0 +0x0030: 0x00007fffffffdf10 → 0x00005555555320 → <__libc_csu_init+0> push r15 ← $rbp
0x00007fffffffdef8 +0x0038: 0x00005555555525e → <main+89> mov eax, 0x0

0x5555555551b5 <check_password+96> mov    BYTE PTR [rbp-0xb], 0x37
0x5555555551b9 <check_password+100> mov    BYTE PTR [rbp-0xa], 0x38
0x5555555551bd <check_password+104> lea    rdx, [rbp-0x20]
→ 0x5555555551c1 <check_password+108> mov    rax, QWORD PTR [rbp-0x28]
0x5555555551c5 <check_password+112> mov    rsi, rdx
0x5555555551c8 <check_password+115> mov    rdi, rax
0x5555555551cb <check_password+118> mov    eax, 0x0
0x5555555551d0 <check_password+123> call   0x5555555555265 <stringCompare>
0x5555555551d5 <check_password+128> mov    DWORD PTR [rbp-0x4], eax

[#0] Id 1, Name: "crackme", stopped 0x5555555551c1 in check_password (), reason: SINGLE STEP
[#0] 0x5555555551c1 → check_password()
[#1] 0x55555555525e → main()

gef> x/s $rdx
0xffffffffded0: "super_secret_pass_56178"
gef> █
```

So we have got the Flag :D

```
(offensive@kali)-[~]
$ ./crackme super_secret_pass_56178
Checking password..
Good job!
```

