

Software Reverse Engineering

Table of Contents

Overview.....	2
crackme	3
Challenge 2 - Find the correct password of crackme1	15
Challenge 3 - Find the correct password of crackme2	36
Static And Dynamic Analysis Tools. Stripped vs Non-Stripped Elf.....	55
Most Used Debugger. Crackme – 3	66
Patching in reverse engineering:	84
Anti-Reversing Techniques, How To Deal With Stripped Binaries	106
How To Deal With Anti-Debug and Anti-VM Technique:.....	134
Reversing C++ Binaries. How To Deal With Packed Binaries:	174
Reverse Engineering Python Binaries	200
Automated Reverse Engineering Using Python Angr :.....	212
References:	233

Overview

Reverse engineering in the context of software refers to the process of analyzing a software application to understand its design, architecture, functionality, and behavior, typically without access to its source code. This process involves dissecting the software to recreate a higher-level representation of its components, logic, or operation.

Key Purposes of Reverse Engineering in Software

1. **Understanding Legacy Systems:** Gaining insights into old or undocumented systems to update, maintain, or integrate them with newer technologies.
2. **Debugging and Bug Fixing:** Diagnosing and fixing issues when source code is unavailable or incomplete.
3. **Security Analysis:** Identifying vulnerabilities, understanding malware behavior, or assessing security mechanisms.
4. **Interoperability:** Developing software that interacts with or extends existing systems without relying on proprietary source code.
5. **Educational Purposes:** Learning how software operates or how certain algorithms are implemented.
6. **Competitive Analysis:** Studying competitors' software to glean features or innovations (though this might raise ethical or legal concerns).
7. **License Compliance:** Verifying that a product complies with license agreements or doesn't infringe intellectual property rights.

Techniques Used in Reverse Engineering

- **Static Analysis:** Examining a program's code, binaries, or resources without executing it. Tools like disassemblers (e.g., IDA Pro) and decompilers (e.g., Ghidra) are often used.
- **Dynamic Analysis:** Observing the behavior of a program while it is running, using tools like debuggers (e.g., GDB) or monitoring software.
- **Binary Analysis:** Analyzing the executable file to understand its machine-level instructions and reconstruct higher-level logic.
- **Protocol Analysis:** Inspecting communication protocols to determine how software interacts with other systems or devices.
- **Code Recovery:** Using decompilers or reconstructing pseudo-source code from machine code to study the software.

Legal and Ethical Considerations

Reverse engineering can be legally sensitive, as it may infringe on intellectual property laws or software licensing agreements. However, it is allowed under certain circumstances, such as:

- **Fair Use:** For purposes like interoperability or educational study.
- **Compliance with Law:** To uncover vulnerabilities or assess software integrity. Always consult legal advice or organizational policies before engaging in reverse engineering.

Tools -

- gdb, Ollydbg, xdbg32, xdbg64, Ghidra, Binary Ninja etc.
- Assembly Language Compilers – Masm, Nasm

Some Terms:

Crackmes - The files you practice reverse engineering on are called **crackmes**.

To get crackmes files visit <https://crackmes.one/> it contains something about 4000 **crackmes** files.

#####

Let's crack a software.

crackme

```
→ yt file crackme
crackme: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64
.so.2, BuildID[sha1]=60bbafef09bf85f7b59c933c41ec32d82859bbd4, for GNU/Linux 3.2.0, not stripped
→ yt ./crackme
Enter Password
12345678
Wrong password.
→ yt
```

Here, we use **gdb** to do this.

So, we need to load our executable in gdb debugger.

```
→ yt gdb crackme
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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:
<http://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"...
Reading symbols from crackme...
(No debugging symbols found in crackme)
(gdb)
```

Here, **gdb shell** is opened.

Run –

Run your program.

```
(gdb) run
Starting program: /home/hellsender/yt/crackme
Enter Password
abcdef
Wrong password.      I
[Inferior 1 (process 19741) exited normally]
(gdb)
```

Jaisa ki hum jante hai ki hum jo bhi logic likhte hai use function ke andar rkhte hai.

Info function:

To show all function of executable.

Use it before run the program in gdb. Because it will show lots of functions, **executable function and libraries function** which is used compile and execute the executable.

Joki humare kam ke nhi hai.

Or **exit** from the gdb and **load** and **info function**.

As you can see there are lots of function if we run and show functions

```
(gdb) run
Starting program: /home/ubuntu/Youtube/reverse-engineering/crackme
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Enter Password
wer
Wrong password.
[Inferior 1 (process 91349) exited normally]
(gdb) info function
All defined functions:

File ./argp/argp-fmtstream.h:
266: size_t __argp_fmtstream_point(argp_fmtstream_t);
220: int __argp_fmtstream_putc(argp_fmtstream_t, int);
207: int __argp_fmtstream_puts(argp_fmtstream_t, const char *);
230: size_t __argp_fmtstream_set_lmargin(argp_fmtstream_t, size_t);
242: size_t __argp_fmtstream_set_rmargin(argp_fmtstream_t, size_t);
254: size_t __argp_fmtstream_set_wmargin(argp_fmtstream_t, size_t);
194: size_t __argp_fmtstream_write(argp_fmtstream_t, const char *, size_t);
```

```
File ./argp/argp.h:
526: void __argp_usage(const struct argp_state *);
544: int __option_is_end(const struct argp_option *);
532: int __option_is_short(const struct argp_option *);

File ./bits/stdc-bsearch.h:
20: void *__GI_bsearch(const void *, const void *, size_t, size_t, __compar_fn_t);

File ./csu/libc-start.c:
234: int __libc_start_main_impl(int (*)(int, char **, char **), int, char **, int (*)(int, char **, char **), void (*)());

File ./elf/dl-sysdep.c:
266: void _dl_show_auxv(void);
82: Elf64_Addr _dl_sysdep_start(void **, void (*)(const Elf64_Phdr *, Elf64_Word, Elf64_Addr *, Elf64_auxv_t *));
261: void _dl_sysdep_start_cleanup(void);

File ./elf/dl-tls.c:
630: void *__GI__dl_allocate_tls(void *);
527: void *__GI__dl_allocate_tls_init(void *, _Bool);
640: void __GI__dl_deallocate_tls(void *, _Bool);
```

Here, you can see very less functions are showing without run the executable.

```
(gdb) info function
All defined functions:

Non-debugging symbols:
0x0000000000401000 _init
0x0000000000401060 puts@plt
0x0000000000401070 __stack_chk_fail@plt
0x0000000000401080 __isoc99_scanf@plt
0x0000000000401090 _start
0x00000000004010c0 _dl_relocate_static_pie
0x00000000004010d0 deregister_tm_clones
0x0000000000401100 register_tm_clones
0x0000000000401140 __do_global_dtors_aux
0x0000000000401170 frame_dummy
0x0000000000401176 main
0x00000000004011f0 __libc_csu_init
0x0000000000401260 __libc_csu_fini
0x0000000000401268 _fini
(gdb) |
```

Here, main is entry point program executing. And left side is their address. It's very useful. If we need to point it then we can use its address.

Jaise ki man lo kisi variable ki value dekhni hai to hum kh skte hai ki is address pr jo value hai use dikhao. Jo kuchh bhi hai dikhao.

Disassemble:

Ab hum ise **disassemble** krenge. **Disassemble** krne ke liye hum nam likh skte hai ya fir address.

```
(gdb) disassemble main
Dump of assembler code for function main:
0x0000000000401176 <+0>:    endbr64
0x000000000040117a <+4>:    push   %rbp
0x000000000040117b <+5>:    mov    %rsp,%rbp
0x000000000040117e <+8>:    sub    $0x10,%rsp
0x0000000000401182 <+12>:   mov    %fs:0x28,%rax
0x000000000040118b <+21>:   mov    %rax,-0x8(%rbp)
0x000000000040118f <+25>:   xor    %eax,%eax
0x0000000000401191 <+27>:   lea    0xe6c(%rip),%rdi      # 0x402004
0x0000000000401198 <+34>:   call   0x401060 <puts@plt>
0x000000000040119d <+39>:   lea    -0xc(%rbp),%rax
0x00000000004011a1 <+43>:   mov    %rax,%rsi
0x00000000004011a4 <+46>:   lea    0xe68(%rip),%rdi      # 0x402013
0x00000000004011ab <+53>:   mov    $0x0,%eax
0x00000000004011b0 <+58>:   call   0x401080 <__isoc99_scanf@plt>
```

```
0x00000000004011b5 <+63>:   mov    -0xc(%rbp),%eax
0x00000000004011b8 <+66>:   cmp    $0x539,%eax
0x00000000004011bd <+71>:   jne    0x4011cd <main+87>
0x00000000004011bf <+73>:   lea    0xe50(%rip),%rdi      # 0x402016
0x00000000004011c6 <+80>:   call   0x401060 <puts@plt>
0x00000000004011cb <+85>:   jmp    0x4011d9 <main+99>
0x00000000004011cd <+87>:   lea    0xe54(%rip),%rdi      # 0x402028
0x00000000004011d4 <+94>:   call   0x401060 <puts@plt>
0x00000000004011d9 <+99>:   nop
0x00000000004011da <+100>:  mov    -0x8(%rbp),%rax
0x00000000004011de <+104>:  xor    %fs:0x28,%rax
0x00000000004011e7 <+113>:  je    0x4011ee <main+120>
0x00000000004011e9 <+115>:  call   0x401070 <__stack_chk_fail@plt>
0x00000000004011ee <+120>:  leave 
0x00000000004011ef <+121>:  ret
```

End of assembler dump.

```
(gdb) |
```

It is in **AT&T** flavor of assembly.

We need to set assembly flavor **Intel**. Because we are not more familiar with **AT&T** flavor.

```
End of assembler dump.
(gdb) set disassembly-flavor
att intel
(gdb) set disassembly-flavor intel
```

Disassemble again

```
(gdb) disassemble main
Dump of assembler code for function main:
0x0000000000401176 <+0>:    endbr64
0x000000000040117a <+4>:    push   rbp
0x000000000040117b <+5>:    mov    rbp,rsp
0x000000000040117e <+8>:    sub    rsp,0x10
0x0000000000401182 <+12>:   mov    rax,QWORD PTR fs:0x28
0x000000000040118b <+21>:   mov    QWORD PTR [rbp-0x8],rax
0x000000000040118f <+25>:   xor    eax,eax
0x0000000000401191 <+27>:   lea    rdi,[rip+0xe6c]      # 0x402004
0x0000000000401198 <+34>:   call   0x401060 <puts@plt>
0x000000000040119d <+39>:   lea    rax,[rbp-0xc]
0x00000000004011a1 <+43>:   mov    rsi,rax
0x00000000004011a4 <+46>:   lea    rdi,[rip+0xe68]      # 0x402013
0x00000000004011ab <+53>:   mov    eax,0x0
0x00000000004011b0 <+58>:   call   0x401080 <__isoc99_scanf@plt>

0x00000000004011b5 <+63>:   mov    eax,DWORD PTR [rbp-0xc]
0x00000000004011b8 <+66>:   cmp    eax,0x539
0x00000000004011bd <+71>:   jne    0x4011cd <main+87>
0x00000000004011bf <+73>:   lea    rdi,[rip+0xe50]      # 0x402016
0x00000000004011c6 <+80>:   call   0x401060 <puts@plt>
0x00000000004011cb <+85>:   jmp    0x4011d9 <main+99>
0x00000000004011cd <+87>:   lea    rdi,[rip+0xe54]      # 0x402028
0x00000000004011d4 <+94>:   call   0x401060 <puts@plt>
0x00000000004011d9 <+99>:   nop
0x00000000004011da <+100>:  mov    rax,QWORD PTR [rbp-0x8]
0x00000000004011de <+104>:  xor    rax,QWORD PTR fs:0x28
0x00000000004011e7 <+113>:  je    0x4011ee <main+120>
0x00000000004011e9 <+115>:  call   0x401070 <__stack_chk_fail@plt>
0x00000000004011ee <+120>:  leave
0x00000000004011ef <+121>:  ret

End of assembler dump.
(gdb)
```

Here, we can see this is converted in Intel assembly.

Let's begin reverse engineering.

There are lots of instructions.

First, we need to focus on **call instruction**. Kahan se kahan call ho rha hai program ka flow kaise hai.

Upar hum dekh skte hai ki sabse phla call puts function ko hua hai. agar hum puts function ke bare me janna chahte hai to.

```
man puts
```

```
ubuntu@ip-172-31-8-118: ~/Y  X  ubuntu@ip-172-31-8-118: ~  +  Linux Programmer's Manual

PUTS(3)

NAME
    fputc, fputs, putc, putchar, puts - output of characters and strings

SYNOPSIS
    #include <stdio.h>

    int fputc(int c, FILE *stream);
    int fputs(const char *s, FILE *stream);
    int putc(int c, FILE *stream);
    int putchar(int c);
    int puts(const char *s);

DESCRIPTION
    fputc() writes the character c, cast to an unsigned char, to stream.

    fputs() writes the string s to stream, without its terminating null byte ('\0').

    putc() is equivalent to fputc() except that it may be implemented as a macro which evaluates its argument as an lvalue.

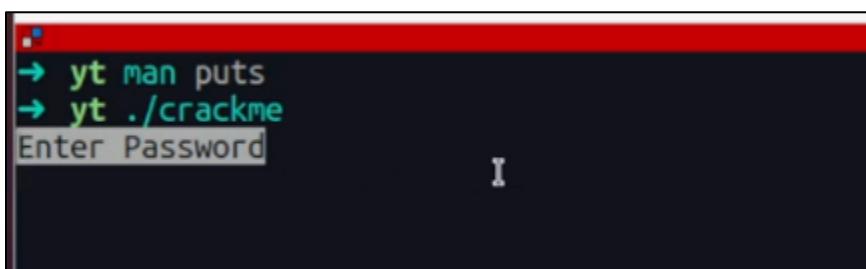
    putchar(c) is equivalent to putc(c, stdout).

    puts() writes the string s and a trailing newline to stdout.

Calls to the functions described here can be mixed with each other and with calls to other output streams.
```

Yh bs print krne ka kam krta hai.

To hume yha samjh aaya sbse phla call print krne ko kr rha hai.



Ab chlte hai next call ki taraf. Next call lg rha hai scnaf ko as we know scanf is used to read data from user(take input).

Agar hum program run krke dekhe to. Hum dekh skte hai ki user se input lene ka kam ho rha hai.

```
ubuntu@ip-172-31-8-118:~$ ./crackme
Enter Password
123456
```

Aage bdte hai humne dekha tha ki input ke bad do bar puts tha lekin yha pr ek hi bar print hua.

```
ubuntu@ip-172-31-8-118:~$ ./crackme
Enter Password
123456
Wrong password.
ubuntu@ip-172-31-8-118:~$ |
```

Iska mtlb program beech ke section ko skip kr hai hai jump kr rha hai.

```
0x00000000004011ab <+53>:    mov    eax,0x0
0x00000000004011b0 <+58>:    call   0x401080 <__isoc99_scanf@plt>
0x00000000004011b5 <+63>:    mov    eax,DWORD PTR [rbp-0xc]
0x00000000004011b8 <+66>:    cmp    eax,0x53
0x00000000004011bd <+71>:    jne    0x4011cd <main+87>
0x00000000004011bf <+73>:    lea    rdi,[rip+0xe50]      # 0x402016
0x00000000004011c6 <+80>:    call   0x401060 <puts@plt>
0x00000000004011cb <+85>:    jmp    0x4011d9 <main+99>
0x00000000004011cd <+87>:    lea    rdi,[rip+0xe54]      # 0x402028
0x00000000004011d4 <+94>:    call   0x401060 <puts@plt>
0x00000000004011d9 <+99>:    nop
0x00000000004011da <+100>:   mov    rax,QWORD PTR [rbp-0x8]
```

Yha pr password compare ho rha hai. agar equal nhi hui to jne intrunction chal rha hai to program to exit kr de rha hai.

And **lea** works as **mov** instruction.

Code flow

```

gdb crackme 116x28
0x0000000000401182 <+12>:    mov    rax,QWORD PTR fs:0x28
0x000000000040118b <+21>:    mov    QWORD PTR [rbp-0x8],rax
0x000000000040118f <+25>:    xor    eax,eax
0x0000000000401191 <+27>:    lea    rdi,[rip+0xe6c]      # 0x402004
0x0000000000401198 <+34>:    call   0x401060 <puts@plt>
0x000000000040119d <+39>:    lea    rax,[rbp-0xc]
0x00000000004011a1 <+43>:    mov    rsi,rax
0x00000000004011a4 <+46>:    lea    rdi,[rip+0xe68]      # 0x402013
0x00000000004011ab <+53>:    mov    eax,0x0
0x00000000004011b0 <+58>:    call   0x401080 <_isoc99_scanf@plt>
0x00000000004011b5 <+63>:    mov    eax,DWORD PTR [rbp-0xc]
0x00000000004011b8 <+66>:    cmp    eax,0x539
0x00000000004011bd <+71>:    jne    0x4011cd <main+87>
0x00000000004011bf <+73>:    lea    rdi,[rip+0xe50]      # 0x402016
0x00000000004011c6 <+80>:    call   0x401060 <puts@plt>
0x00000000004011cb <+85>:    jmp    0x4011d9 <main+99>
0x00000000004011cd <+87>:    lea    rdi,[rip+0xe54]      # 0x402028
0x00000000004011d4 <+94>:    call   0x401060 <puts@plt>
0x00000000004011d9 <+99>:    nop
0x00000000004011da <+100>:   mov    rax,QWORD PTR [rbp-0x8]

```

Here, we use debugger to show execution flow.

```

0x00000000004011ab <+53>:    mov    eax,0x0
0x00000000004011b0 <+58>:    call   0x401080 <_isoc99_scanf@plt>
0x00000000004011b5 <+63>:    mov    eax,DWORD PTR [rbp-0xc]
0x00000000004011b8 <+66>:    cmp    eax,0x539
0x00000000004011bd <+71>:    jne    0x4011cd <main+87>
0x00000000004011bf <+73>:    lea    rdi,[rip+0xe50]      # 0x402016
0x00000000004011c6 <+80>:    call   0x401060 <puts@plt>
0x00000000004011cb <+85>:    jmp    0x4011d9 <main+99>
0x00000000004011cd <+87>:    lea    rdi,[rip+0xe54]      # 0x402028
0x00000000004011d4 <+94>:    call   0x401060 <puts@plt>

```

We put a breakpoint on here, to pause and execute step by step.

```

(gdb) break *0x00000000004011b8
Breakpoint 1 at 0x4011b8
(gdb) run
Starting program: /home/hellsender/yt/crackme
Enter Password
123456

Breakpoint 1, 0x00000000004011b8 in main ()
(gdb) 

```

Yha humne address **cmp** ke address pr break point lagaya. Aur run kiya aur jaise hi humne password diya isne breakpoint ko hit kiya.

```

=> 0x00000000004011b5 <+63>:    mov    eax,DWORD PTR [rbp-0xc]
=> 0x00000000004011b8 <+66>:    cmp    eax,0x539
0x00000000004011bd <+71>:    jne    0x4011cd <main+87>
0x00000000004011bf <+73>:    lea    rdi,[rip+0xe50]      # 0x402016
0x00000000004011c6 <+80>:    call   0x401060 <puts@plt>
0x00000000004011cb <+85>:    jmp   0x4011d9 <main+99>

```

Aur hum dekh skte hai ki break point lg chuka hai.

Iska bad hum register ka information dekhte hai. ki kaun se register me kitna value hai.

```

End of assembler dump.
(gdb) info registers
rax            0x1e240 123456
rbx            0x4011f0 4198896
rcx            0x0 0
rdx            0x0 0
rsi            0x0 0
rdi            0x7fffffff950 140737488345424
rbp            0x7fffffffdea0 0x7fffffffdea0
rsp            0x7fffffffde90 0x7fffffffde90
r8             0xa 10
r9              0x0 0
r10            0x7ffff7f5bac0 140737353464512
r11            0x0 0
r12            0x401090 4198544
r13            0x7fffffffdf90 140737488347024
r14            0x0 0
r15            0x0 0
rip            0x4011b8 0x4011b8 <main+66>
eflags          0x206 [ PF IF ]
cs             0x33 51
ss             0x2b 43

```

```

0x00000000004011b5 <+63>:    mov    eax,DWORD PTR [rbp-0xc]
=> 0x00000000004011b8 <+66>:    cmp    eax,0x539
0x00000000004011bd <+71>:    jne    0x4011cd <main+87>
0x00000000004011bf <+73>:    lea    rdi,[rip+0xe50]      # 0x402016

```

As we can see **rax** and **eax** value are not same then it will jump **0x4011cd**.

And we can both are hex value then let's see in decimal value.

```

(gdb) print $1
$1 = 123456
(gdb)

```

Ab hum next instruction pr chalte hai.

```
(gdb) nexti  
0x00000000004011bd in main ()  
(gdb)
```

Yha hum next instruction pr aa gye.

```
0x00000000004011b8 <+66>:    cmp    eax,0x539  
=> 0x00000000004011bd <+71>:    jne    0x4011cd <main+87>  
0x00000000004011bf <+73>:    lea    rdi,[rip+0xe50]      # 0x402016  
0x00000000004011c6 <+80>:    call   0x401060 <puts@plt>
```

One thing to noted, yh line abhi execute nhi ki gyi hai.

Hum fir next instruction pr chalte hai. using ni command **ni** and **nexti** are same.

```
End of assembler dump.  
(gdb) ni  
0x00000000004011cd in main ()
```

Dissambler me dekhte hai

```
0x00000000004011d4 <+94>:    call   0x401060 <puts@plt>  
0x00000000004011d9 <+99>:    nop  
0x00000000004011da <+100>:   mov    rax,QWORD PTR [rbp-0x8]  
0x00000000004011de <+104>:   xor    rax,QWORD PTR fs:0x28  
0x00000000004011e7 <+113>:   je     0x4011ee <main+120>
```

Again **ni**

```
0x00000000004011d4 <+94>:    call   0x401060 <puts@plt>  
0x00000000004011d9 <+99>:    nop  
0x00000000004011da <+100>:   mov    rax,QWORD PTR [rbp-0x8]  
0x00000000004011de <+104>:   xor    rax,QWORD PTR fs:0x28  
0x00000000004011e7 <+113>:   je     0x4011ee <main+120>
```

Jaisa ki humne galat password dala tha

To humne dekha ki hum wrong password ke sath bahar aa chuke hai.

```
0x00000000004011b8 <+66>:    cmp    eax,0x539  
0x00000000004011bd <+71>:    jne    0x4011cd <main+87>  
0x00000000004011bf <+73>:    lea    rdi,[rip+0xe50]      # 0x402016  
0x00000000004011c6 <+80>:    call   0x401060 <puts@plt>  
0x00000000004011cb <+85>:    jmp    0x4011d9 <main+99>  
0x00000000004011cd <+87>:    lea    rdi,[rip+0xe54]      # 0x402028  
0x00000000004011d4 <+94>:    call   0x401060 <puts@plt>
```

Agar hume beech ke **call** ko access krna hai to hume **cmp** ki value equal krni hogi.

To hum check krte hai **eax** me to user ka value store ho rha hai but **0x539** kaun si value hai.

```
(gdb) print 0x539  
$2 = 1337  
(gdb)
```

Yha dekh skte hai ki **0x539**, **1337** ki hex value hai.

```
→ yt ./crackme  
Enter Password  
1337  
Correct password.  
→ yt
```

So, this is the correct value.

How to remove breakpoint.

1. **List all breakpoints:** Use the info breakpoints command to list all breakpoints along with their numbers.

```
(gdb) info breakpoints
```

2. **Remove a specific breakpoint:** Use the delete command followed by the breakpoint number.

```
(gdb) delete 1
```

This removes breakpoint number 1.

3. **Remove all breakpoints:** To remove all breakpoints at once, use:

```
(gdb) delete
```

4. **Optional confirmation (if prompted):** If GDB prompts for confirmation to delete multiple breakpoints, type y or yes to confirm.

Notes:

- If you want to disable a breakpoint temporarily without deleting it, use the disable command instead of delete.

```
(gdb) disable 1
```

- To re-enable a disabled breakpoint, use:

```
(gdb) enable 1
```

Summary Table:

Command	Description

#####

Challenge 2 - Find the correct password of crackme1

```
→ rev ./crackme1
Argument 1 Missing.
→ rev ./crackme1 abcdef
Wrong Password.
→ rev █
```

Here, we will use **radare2** to do this. Radare2 can convert our assembly language code in high level language.

Let's start **radare2**. It is known as **r2**.

```
→ rev r2 -d ./crackme1
-- Well this is embarrassing ...
[0x7fa438198100]>
```

Radare2 apne aap se kuchh nhi krta hume isse khte pdta hai ki binary ko read kr lo apne andar le aao.

Here, we run command **aaa** to analyze the binary.

```
[0x7fa438198100]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Finding and parsing C++ vtables (avrr)
[x] Skipping type matching analysis in debugger mode (aaft)
[x] Propagate noreturn information (aanr)
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x7fa438198100]> I
```

Show all functions. Use **afl** command.

```
[0x77ea02461290]> afl
0x00401060    1      11  sym.imp.puts
0x00401070    1      11  sym.imp.__stack_chk_fail
0x00401080    1      11  sym.imp.__isoc99_scanf
0x00401090    1      47  entry0
0x004010d0    4      31  sym.deregister_tm_clones
0x00401100    4      49  sym.register_tm_clones
0x00401140    3      32  entry.fini0
0x00401170    1      6   entry.init0
0x00401260    1      5   sym.__libc_csu_fini
0x00401268    1      13  sym._fini
0x004011f0    4      101  sym.__libc_csu_init
0x004010c0    1      5   sym._dl_relocate_static_pie
0x00401176    6      122  main
0x00401000    3      27  sym._init
```

Put a breakpoint on **main**. Use **db main** command. Means **debug breakpoint**.

```
[0x77ea02461290]>
[0x77ea02461290]> db main
[0x77ea02461290]>
```

Now, we use **dc** command to run the program and it will hit breakpoint. And pause. **dc** means debug continue. In **radare2** everything happens with debug.

```
[0x77ea02461290]> dc
INFO: hit breakpoint at: 0x401176
[0x00401176]>
```

This is address of
main function

And here we can see our shell address is also changed.

Ab hume iski disassembly code dekhna hai. to hum run karenge **pdf** command.

```
[0x00401176]> pdf
    ;-- rax:
    ;-- r13:
    ;-- rip:
; DATA XREF from entry0 @ 0x4010b1(r)
122: int main (int argc, char **argv, char **envp);
afv: vars(2:sp[0x10..0x14])
    0x00401176 b    f30f1efa    endbr64
    0x0040117a      55        push rbp
    0x0040117b      4889e5    mov rbp, rsp
    0x0040117e      4883ec10  sub rsp, 0x10
    0x00401182      64488b0425.. mov rax, qword fs:[0x28]
    0x0040118b      488945f8  mov qword [var_8h], rax
    0x0040118f      31c0      xor eax, eax
    0x00401191      488d3d6c0e.. lea rdi, str.Enter_Password ; 0x402004 ; "Enter Password"
    0x00401198      e8c3feffff call sym.imp.puts ; int puts(const char *s)
    0x0040119d      488d45f4  lea rax, [var_ch]
    0x004011a1      4889c6    mov rsi, rax
    0x004011a4      488d3d680e.. lea rdi, [0x00402013] ; "%d"
    0x004011ab      b800000000  mov eax, 0
    0x004011b0      e8cbfeffff call sym.imp.__isoc99_scanf ; int scanf(const char *format)
    0x004011b5      8b45f4    mov eax, dword [var_ch]
    0x004011b8      3d39050000  cmp eax, 0x539 ; 1337
    0x004011bd      750e      jne 0x4011cd
    0x004011bf      488d3d500e.. lea rdi, str.Correct_password. ; 0x402016 ; "Correct password."
    0x004011c6      e895feffff call sym.imp.puts ; int puts(const char *s)
    0x004011cb      eb0c      jmp 0x4011d9
    L-> 0x004011cd      488d3d540e.. lea rdi, str.Wrong_password. ; 0x402028 ; "Wrong password."
    0x004011d4      e887feffff call sym.imp.puts ; int puts(const char *s)
; CODE XREF from main @ 0x4011cb(x)
    -> 0x004011d9      90        nop
    0x004011da      488b45f8  mov rax, qword [var_8h]
    0x004011de      6448330425.. xor rax, qword fs:[0x28]
    <- 0x004011e7      7405      je 0x4011ee
    0x004011e9      e882feffff call sym.imp.__stack_chk_fail ; void __stack_chk_fail(void)
    -> 0x004011ee      c9        leave
    0x004011ef      c3        ret
[0x00401176]> |
```

It provides assembly code in **intel flavor**. We can change it in AT&T assembly also.

Ab hum darse visual mode me dekhenge ise. Iske liye hum **V** use karenge.

```
[0x00401176]>
[0x00401176]> V|
```

```
[0x00401176 [xAdvc]0 672 /home/ubuntu/crackme]> xc @ main
- offset - 7677 7879 7A7B 7C7D 7E7F 8081 8283 8485 6789ABCDEF012345 comment
0x00401176 f30f 1efa 5548 89e5 4883 ec10 6448 8b04 .....UH..H...dH.. ; rip
0x00401186 2528 0000 0048 8945 f831 c048 8d3d 6c0e %(...H.E.1.H.=l.
0x00401196 0000 e8c3 feff ff48 8d45 f448 89c6 488d .....H.E.H..H.
0x004011a6 3d68 0e00 00b8 0000 0000 e8cb feff ff8b =h..... .
0x004011b6 45f4 3d39 0500 0075 0e48 8d3d 500e 0000 E.=9..u.H.=P...
0x004011c6 e895 feff ffeb 0c48 8d3d 540e 0000 e887 .....H.=T.....
0x004011d6 feff ff90 488b 45f8 6448 3304 2528 0000 .....H.E.dH3.%(.. .
0x004011e6 0074 05e8 82fe ffff c9c3 f30f 1efa 4157 .t.....AW ; rcx
0x004011f6 4c8d 3d13 2c00 0041 5649 89d6 4155 4989 L.=.,,AVI.AUI. ; arg3 ; arg2
0x00401206 f541 5441 89fc 5548 8d2d 042c 0000 534c .ATA.UH.-.,,SL ; arg1
0x00401216 29fd 4883 ec08 e8df fdff ff48 c1fd 0374 ).H.....H..t
0x00401226 1f31 db0f 1f80 0000 0000 4c89 f24c 89ee .1.....L..L..
0x00401236 4489 e741 ff14 df48 83c3 0148 39dd 75ea D..A..H..H9.u.
0x00401246 4883 c408 5b5d 415c 415d 415e 415f c366 H...[.]A\A]A^A_.f
0x00401256 662e 0f1f 8400 0000 0000 f30f 1efa c300 f..... ; sym._libc_csu_fini
0x00401266 0000 f30f 1efa 4883 ec08 4883 c408 c300 .....H..H... ; sym._fini ; [16] -r-x sect
0x00401276 0000 0000 0000 0000 0000 0000 0000 0000 ..... .
0x00401286 0000 0000 0000 0000 0000 0000 0000 0000 ..... .
0x00401296 0000 0000 0000 0000 0000 0000 0000 0000 ..... .
0x004012a6 0000 0000 0000 0000 0000 0000 0000 0000 ..... .


```

Here, we can see it is showing hex dump. To see next window press **p**

```
[0x0040104f [xAdvc]0 210 /home/ubuntu/crackme]> afsQ;pd $r.. @ sym._init+79 # 0x40104f
 90          nop
 0x00401050 f30f1efa    endbr64
 0x00401054 6802000000 push 2           ; 2
 0x00401059 f2e9c1fffff bnd jmp section..plt
 0x0040105f 90          nop
;-- section..plt.sec:
; CALL XREFS from main @ 0x401198(x), 0x4011c6(x), 0x4011d4(x)
11: int sym.imp.puts (const char *s);
 0x00401060 f30f1efa    endbr64           ; [14] -r-x section size 48 named .plt.sec
 0x00401064 f2ff25ad2f.. bnd jmp qword [puts]      ; [0x404018:8]=0x401030
 0x0040106b 0f1f440000  nop dword [rax + rax]
; CALL XREF from main @ 0x4011e9(x)
11: void sym.imp.__stack_chk_fail () // noreturn
 0x00401070 f30f1efa    endbr64
 0x00401074 f2ff25a52f.. bnd jmp qword [__stack_chk_fail] ; [0x404020:8]=0x401040 ; "@\x10@"
 0x0040107b 0f1f440000  nop dword [rax + rax]
; CALL XREF from main @ 0x4011b0(x)
11: int sym.imp.__isoc99_scanf (const char *format);
 0x00401080 f30f1efa    endbr64
 0x00401084 f2ff259d2f.. bnd jmp qword [__isoc99_scanf] ; [0x404028:8]=0x401050 ; "P\x10@"
 0x0040108b 0f1f440000  nop dword [rax + rax]
;-- section..text:
;-- _start:
47: entry0 (int64_t arg3);
`- args(rdx)
 0x00401090 f30f1efa    endbr64           ; [15] -r-x section size 469 named .text
 0x00401094 31ed        xor ebp, ebp
 0x00401096 4989d1      mov r9, rdx       ; arg3
 0x00401099 5e          pop rsi
 0x0040109a 4889e2      mov rdx, rsp
 0x0040109d 4883e4f0    and rsp, 0xfffffffffffffff0
 0x004010a1 50          push rax
 0x004010a2 54          push rsp
 0x004010a3 49c7c06012.. mov r8, sym._libc_csu_fini ; 0x401260
 0x004010aa 48c7c1f011.. mov rcx, sym._libc_csu_init ; 0x4011f0
 0x004010b1 48c7c77611.. mov rdi, main       ; rip

```

It will showing disassembly.

Press again **p**

```
[0x0040104f [xaDvc]0 210 /home/ubuntu/crackme]> diq;?t0;f .. @ sym._init+79 # 0x40104f
breakpoint at 0x00000000
- offset -
  1819 1A1B 1C1D 1E1F 2021 2223 2425 2627 89ABCDEF01234567
0x7ffdc5b77318 909d 2202 ea77 0000 0000 0000 0000 ...".w.....
0x7ffdc5b77328 7611 4000 0000 0000 0000 0100 0000 v.@.....
0x7ffdc5b77338 2874 b7c5 fd7f 0000 0000 0000 0000 (t.....
0x7ffdc5b77348 dec1 f433 1a7f 31a4 2874 b7c5 fd7f 0000 ...3..1.(t.....
s:0 z:1 c:0 o:0 p:1
  rax 0x00401176          rbx 0x00000000          rcx 0x004011f0
  rdx 0x7ffdc5b77438      r8 0x77ea0241bf10      r9 0x77ea02447040
  r10 0x77ea02441908     r11 0x77ea0245c660     r12 0x7ffdc5b77428
  r13 0x00401176          r14 0x00000000          r15 0x77ea0247b040
  rsi 0x7ffdc5b77428      rdi 0x00000001          rsp 0x7ffdc5b77318
  rbp 0x00000001          rip 0x00401176          rflags 0x00000246
  orax 0xfffffffffffffff
  0x0040104f    90        nop
  0x00401050    f30f1efa  endbr64
  0x00401054    6802000000 push 2
  0x00401059    f2e9c1fffff bnd jmp section..plt ; 2
  0x0040105f    90        nop
  ;-- section..plt.sec:
  ; CALL XREFS from main @ 0x401198(x), 0x4011c6(x), 0x4011d4(x)
11: int sym.imp.puts (const char *s);
  0x00401060    f30f1efa  endbr64 ; [14] -r-x section size 48 named .plt.sec
  0x00401064    f2ff25ad2f.. bnd jmp qword [puts] ; [0x404018:8]=0x401030
  0x0040106b    0f1f440000  nop dword [rax + rax]
  ; CALL XREF from main @ 0x4011e9(x)
11: void sym.imp.__stack_chk_fail () // noreturn
  0x00401070    f30f1efa  endbr64
  0x00401074    f2ff25a52f.. bnd jmp qword [__stack_chk_fail] ; [0x404020:8]=0x401040 ; "@\x10@"
  0x0040107b    0f1f440000  nop dword [rax + rax]
  ; CALL XREF from main @ 0x4011b0(x)
11: int sym.imp._isoc99_scanf (const char *format);
  0x00401080    f30f1ef0  endbr64
  ; CALL XREF from main @ 0x4011b0(x)
```

it also shows
registers values.

then it will show disassembly as well as registers values.

There are some more modes available press **p** again and again.

If you want to **exit** it press **q**.

Now, we will show **Visual Graph** mode.

Press **VV** and enter.

```
[0x0040104f]>
[0x0040104f]> VV|
```

```
[ubuntu@ip-172-31-8-118: ~] x + v  
[0x00401176]> 0x4011d9 # int main (int argc, char **argv, char **envp);  
    rax 0x00401176          rbx 0x00000000          rcx 0x004011f0          rdx 0x7ffdcc5b7743  
0x77ea02447040  
    r10 0x77ea02441908      r11 0x77ea0245c660      r12 0x7ffdcc5b77428      r13 0x00401176  
0x77ea0247b040  
    rsi 0x7ffdcc5b77428      rdi 0x00000001          rsp 0x7ffdcc5b77318      rbp 0x000000001  
0x00000246  
orax 0xfffffffffffffff  
  
0x401176 [oc]  
; [x] DATA XREF from entry0 @ 0x4010b1(r)  
122: int main (int argc, char **argv, char **envp);  
afv: vars(2:sp[0x10..0x14])  
main,main,..: endbr64  
    push rbp  
    mov rbp, rsp  
    sub rsp, 0x10  
    mov rax, qword fs:[0x28]  
    mov qword [var_8h], rax  
    xor eax, eax  
; 0x402004  
; "Enter Password"  
    lea rdi, str.Enter_Password  
; int puts(const char *s)  
    call sym.imp.puts;[0a]  
    lea rax, [var_ch]  
    mov rsi, rax  
; "%d"  
    lea rdi, [0x00402013]  
    mov eax, 0  
; int scanf(const char *format)  
    call sym.imp._isoc99_scanf;[0b]  
    mov eax, dword [var_ch]  
; 1337  
    cmp eax, 0x539  
    jne 0x4011cd
```



In visual graph mode we can see a jump statement **if false** to kahan jayega **if true** to kahan jayega.

```
[0x00401176]> 0x4011d9 # int main (int argc, char **argv, char **envp);
    rax 0x00401176          rcx 0x004011f0           rdx 0x7ffdc5b77438          r8 0x77ea
    0x77ea02447040          r11 0x77ea0245c660      r12 0x7ffdc5b77428          r13 0x00401176          r14 0x0000
    r10 0x77ea02441908      rsi 0x7ffdc5b77428      cmp eax, 0x539          rbp 0x00000001
    0x77ea0247b040          rdi 0x00000001       rsp 0x7ffdc5b77318          rip 0x004011d9
    0x00000246
    orax 0xfffffffffffffff

0x4011bf [od]
; 0x402016
; "Correct password."
    lea rdi, str.Correct_password.
; int puts(const char *s)
    call sym.imp.puts;[oa]
    jmp 0x4011d9

0x4011cd [oe]
; 0x402028
; "Wrong password."
    lea rdi, str.Wrong_password.
; int puts(const char *s)
    call sym.imp.puts;[oa]

[0x4011d9]
;[x] CODE XREF from main @ 0x4011cb(x)
nop
mov rax, qword [var_8h]
xor rax, qword fs:[0x28]
je 0x4011ee

0x4011e9 [oh]
0x4011ee [oi]
```

We can move it using **arrow key**. And press **shift + r** to change font color (theme).

Press **p** for next style.

```

[0x00401176]> 0x401176 # int main (int argc, char **argv, char **envp);
    rax 0x00401176          rbx 0x00000000          rcx 0x004011f0          rdx 0x7ffdb89eae28          r8 0x7d1b1e81bf10
0x7d1b1e936040          r10 0x7d1b1e930908          r11 0x7d1b1e94b660          r12 0x7ffdb89eae18          r13 0x00401176          r14 0x00000000
0x7d1b1e96a040          rsi 0x7ffdb89eae18          rdi 0x00000001          rbp 0x00000001          rip 0x00401176
0x00000246          orax 0xfffffffffffffff
0x00401176              | 122: int main (int argc, char **argv, char **envp);
0x00401176              | 0x00401176 main,main,..: endbr64
0x00401176              |           rsp 0x7ffdb89ead08
0x00401176              |           rbp 0x00000001
0x00401176              | 0x0040117b      mov rbp, rsp
0x00401176              | 0x0040117e      sub rsp, 0x10
0x00401182              | 0x00401182      mov rax, qword fs:[0x28]
0x0040118b              | 0x0040118b      mov qword [var_8h], rax
0x0040118f              | 0x0040118f      xor eax, eax
; 0x402004
; "Enter Password"
0x00401191              | 0x00401191      lea rdi, str.Enter_Password
; int puts(const char *s)
0x00401198              | 0x00401198      call sym.imp.puts;[oa]
0x0040119d              | 0x0040119d      lea rax, [var_ch]
0x004011a1              | 0x004011a1      mov rsi, rax
; "%d"
0x004011a4              | 0x004011a4      lea rdi, [0x00402013]
0x004011ab              | 0x004011ab      mov eax, 0
; int scanf(const char *format)
0x004011b0              | 0x004011b0      call sym.imp._isoc99_scanf;[ob]
0x004011b5              | 0x004011b5      mov eax, dword [var_ch]
; 1337
0x004011b8              | 0x004011b8      cmp eax, 0x539
0x004011bd              | 0x004011bd      jne 0x4011cd

```

```

0x4011bf [od]
; 0x402016
; "Correct password."
0x004011bf      lea rdi, str.Correct_password.
; int puts(const char *s)
0x004011c6      call sym.imp.puts;[oa]
0x004011cb      jmp 0x4011d9

```

```

0x4011cd [oe]
; 0x402028
; "Wrong password."
0x004011cd      lea rdi, str.Wrong_password.
; int puts(const char *s)
0x004011d4      call sym.imp.puts;[oa]

```

Here, we can see address also with disassembly.

Now, Here, we can see pointer on main function.

```

[0x401176]
; [x] DATA XREF from entry0 @ 0x4010b1(r)
122: int main (int argc, char **argv, char **envp);
afv: vars(2:sp[0x10..0x14])
0x00401176 main,main,..: endbr64
0x0040117a      push rbp
0x0040117b      mov rbp, rsp
0x0040117e      sub rsp, 0x10
0x00401182      mov rax, qword fs:[0x28]
0x0040118b      mov qword [var_8h], rax
0x0040118f      xor eax, eax

```

If we want to move (go to next instruction) press **shift+s**.

```
[0x401176]
; [x] DATA XREF from entry0 @ 0x4010b1(r)
122: int main (int argc, char **argv, char **envp);
afv: vars(2:sp[0x10..0x14])
0x00401176 main,main,..: endbr64
0x0040117a rip: push rbp
0x0040117b mov rbp, rsp
0x0040117e sub rsp, 0x10
```

Ok, ab hum cmp instruction pr chalte hai. because compare import thing in reverse engineering.

```
;-- rip:
0x00401206 837dcc01      cmp dword [var_34h], 1
0x0040120a 7f16          jg 0x401222
```

Here, we can see **var_34h** variable **1** se compare ho rha hai.

To hum dekhte hai ki var_34h me kaun si value hai.

```
x004011b6> 0x4011b6 # int main (int argc, char **argv);-0
; var int64_t var_28h @ rbp-0x28
; var int64_t var_20h @ rbp-0x20
; var int64_t var_1ch @ rbp-0x1c
; var int64_t var_1ah @ rbp-0x1a
; var int64_t var_18h @ rbp-0x18
; arg int argc @ rdi
; arg char **argv @ rsi
0x004011b6 f30f1efa      endbr64
0x004011ba 55             push rbp
0x004011bb 4889e5         mov rbp, rsp
0x004011be 53             push rbx
0x004011bf 4883ec38       sub rsp, 0x38
; argc
0x004011c3 897dcc         mov dword [var_34h], edi
; argv
0x004011c6 488975c0         mov qword [var_40h], rsi
0x004011ca 64488b042528.   mov rax, qword fs:[0x28]
0x004011d3 488945e8         mov qword [var_18h], rax
0x004011d7 31c0             xor eax, eax
```

Upar hum dekh skte hai ki edi ki jo bhi value thi wo **var_34h** me copy hui hai.

Aur **edi** ki jo bhi value thi **argc** means **argument counter**. Then total number of argument ko hum **argc** khte hai.

To yha pr total humber ko 1 se compare kr rha hai. agar total number of argument 1 se jyada hai to go to **true(t)** otherwise **false(f)**.

```

| 0x00401202 c645e600      mov byte [var_1ah], 0
| ;-- rip:
| 0x00401206 837dcc01      cmp dword [var_34h], 1
| 0x0040120a 7f16          jg 0x401222

f t

Missing."
8d3df10d00. lea rdi, str.Argument_1_Missing.
ist char *s)

0x401222 [of]
0x00401222 488d45d0      lea rax, [var_30h]
0x00401226 4889c7          mov rdi, rax
; size_t strlen(const char *s)
0x00401229 e862feffff    call sym.imp.strlen

```

It means we have to give atleast two argument. Ek argument jata hi jata hai. filename ko ek argument consider kiya jata hai. uske aage jo bhi argument dete hai wo second, third, forth .. argument consider hota hai.

```

[0x4011b6]
; DATA XREF from entry0 @ 0x4010f1
257: int main (int argc, char **argv, char **envp);
; var int64_t var_40h @ rbp-0x40
; var int64_t var_34h @ rbp-0x34
; var int64_t var_30h @ rbp-0x30
; var int64_t var_28h @ rbp-0x28
; var int64_t var_20h @ rbp-0x20
; var int64_t var_1ch @ rbp-0x1c
; var int64_t var_1ah @ rbp-0x1a

```

Yha pr **var_34h** ka memory address **rbp-0x34** hai to iska name **var_34h** rkha gya hai.

Ab hume radare ek command run krna hai. To hum ":" press karenge. Hume dikh jayega. Hum yha ek memory location pr kya rkha use dekhna hai to **px** command run karenge @ and **address ka nam**.

```
:> px@rbp-0x34
```

```

| 0x004011d9 48D853337570. |
:> px@rpbp-0x34
- offset -   0 1 2 3 4 5 6 7 8 9 A B C D E 0123456789ABCDE
0x7fff17f7cb2c 0100 0000 5333 7570 6572 5f53 3363 72 ...S3uper_S3cr
0x7fff17f7cb3b 3374 5f50 6133 3377 3072 6400 0000 ef 3t_Pa33w0rd...
0x7fff17f7cb4a dd57 518b a2b2 50cc f717 ff7f 0000 c0 .WQ...P...
0x7fff17f7cb59 1240 0000 0000 0000 0000 0000 0000 00 .@...
0x7fff17f7cb68 b360 3239 ef7f 0000 20b6 5339 ef7f 00 .`29...S9...
0x7fff17f7cb77 0058 ccf7 17ff 7f00 0000 0000 0001 00 .X...
0x7fff17f7cb86 0000 b611 4000 0000 0000 c012 4000 00 ...@...@...
0x7fff17f7cb95 0000 0059 3dc5 2836 e68d 4bd0 1040 00 ...Y=.(6..K..@...
0x7fff17f7cba4 0000 0000 50cc f717 ff7f 0000 0000 00 ...P...
0x7fff17f7ccb3 0000 0000 0000 0000 0000 0059 3d ...Y=...
0x7fff17f7cbc2 25be d9c9 73b4 593d 0be8 5294 53b4 00 %...s.Y=..R.S...
0x7fff17f7cbd1 0000 0000 0000 0000 0000 0000 0000 00 ...
0x7fff17f7cbe0 0000 0000 0000 0100 0000 0000 0000 00 ...
0x7fff17f7cbef 0058 ccf7 17ff 7f00 0068 ccf7 17ff 7f .X....h...
0x7fff17f7cbfe 0000 90d1 5339 ef7f 0000 0000 0000 00 ...S9...
0x7fff17f7cc0d 0000 0000 0000 0000 00d0 1040 00 ...@...
0x7fff17f7cc1c 0000 0000 50cc f717 ff7f 0000 0000 00 ...P...
0x7fff17f7cc2b 00 ...
:>

```

Yha pr sabse phli value hai wahi **var_34h** variable ki value hai.

Press **Enter** to go to visual mode again.

Go to command mode and run further execution. And finish it.

```

:> dc
Argument 1 Missing.
(27610) Process exited with status=0x100
:>

```

Ab hum program ko restart krte hai ek argument ke sath. We will use **oop** pass any argument

```

> oop 1234567
PTRACE_CONT: No such process
(160035) Process exited with status=0x9
child received signal 9
INFO: File dbg:///home/ubuntu/crackme1 1234567 reopened in read-write mode
> dc
INFO: hit breakpoint at: 0x4011b6
>

```

dc for run .

```
0x004011f1 488955d8      mov qword [var_28h], rdx
; '33w0'
0x004011f5 c745e0333377. mov dword [var_20h], 0x30773333
; 'rd'
0x004011fc 66c745e47264  mov word [var_1ch], 0x6472
0x00401202 c645e600      mov byte [var_1ah], 0
;-- rip:
0x00401206 837dcc01      cmp dword [var_34h], 1
0x0040120a 7f16          jg 0x401222
```

f t
|

Ab hum fir dekhte hai. usi memory address pr kya stored hai.

```
:> px@rbp-0x34
- offset -
0 1 2 3 4 5 6 7 8 9 A B C D E 0123456789ABCDE
0x7ffee13e1fcc 0200 0000 5333 7570 6572 5f53 3363 72 ...S3uper_S3cr
0x7ffee13e1fdb 3374 5f50 6133 3377 3072 6400 0000 50 3t_Pa33w0rd...P
0x7ffee13e1fea 9ffb 2816 cb63 f020 3ee1 fe7f 0000 c0 ..(.c. >.....
0x7ffee13e1ff9 1240 0000 0000 0000 0000 0000 0000 00 .@.....
0x7ffee13e2008 b360 8362 517f 0000 20b6 a462 517f 00 .`bQ... .bQ...
0x7ffee13e2017 00f8 203e e1fe 7f00 0000 0000 0002 00 .. >.....
0x7ffee13e2026 0000 b611 4000 0000 0000 c012 4000 00 ...@.... @..
0x7ffee13e2035 0000 0085 d66f efda dd22 49d0 1040 00 ....o..."I..@.
0x7ffee13e2044 0000 0000 f020 3ee1 fe7f 0000 0000 00 .... >.....
0x7ffee13e2053 0000 0000 0000 0000 0000 0000 0085 d6 .....
0x7ffee13e2062 4faf a61f dfb6 85d6 a12f dc18 80b7 00 0...../....
```

Is bar first value **2** hai.

To ab iski value 2 ho gyi to ye jump lega. Aur t wale section me chla jayega.

```
| 0x004011f5 c745e0333377. mov dword [var_20h], 0x30773333
| ; 'rd'
| 0x004011fc 66c745e47264 mov word [var_1ch], 0x6472
| 0x00401202 c645e600 mov byte [var_1ah], 0
| 0x00401206 837dcc01 cmp dword [var_34h], 1
;-- rip:
| 0x0040120a 7f16      jg 0x401222
```

f t

missing."

[3df10d00. lea rdi, str.Argument_1_Missing.

0x401222 [of]
0x00401222 488d45d0
0x00401226 4889c7
; size_t strlen(const char *

Ab hum dusre section me jate hai.

```
[0x401222]
0x00401222 488d45d0      lea rax, [var_30h]
0x00401226 4889c7      mov rdi, rax
; size_t strlen(const char *s)
0x00401229 e862feffff    call sym.imp.strlen;[oe]
0x0040122e 4889c3      mov rbx, rax
0x00401231 488b45c0      mov rax, qword [var_40h]
0x00401235 4883c008      add rax, 8
0x00401239 488b00      mov rax, qword [rax]
0x0040123c 4889c7      mov rdi, rax
; size_t strlen(const char *s)
0x0040123f e84cfefeff    call sym.imp.strlen;[oe]
;-- rip:
0x00401244 4839c3      cmp rbx, rax
0x00401247 7416      je 0x40125f
```

f t

0x40125f [oi]
0x0040125f 488b45c0 mov r

```
0x0040123f e84cfeffff    call sym.imp.strlen;[oe]
;-- rip:
0x00401244 4839c3          cmp rbx, rax
0x00401247 7416             je 0x40125f

0x401249 [og]
; 0x402018
; "Wrong Password."
0x00401249 488d3dc80d00. lea rdi, str.Wrong_Password.
; int puts(const char *s)
0x00401250 e82bfeffff    call sym.imp.puts;[ob]
0x00401255 bf01000000    mov edi, 1
; void exit(int status)

0x40125f [oi]
0x0040125f 488b45c0        mov rax, qword
0x00401263 4883c008        add rax, 8
0x00401267 488b10        mov rdx, qword
0x0040126a 488d45d0        lea rax, [var_3]
0x0040126e 4889d6        mov rsi, rdx
0x00401271 4889c7        mov rdi, rax
; int strcmp(const char *s1, const char *
```

Yha pr **sym.imp.strlen** bs string ki length batate hai.

Is pure section me check ho rha hai ki jo bhi humne password diya hai uske length ko compare kiya ja rha hai. agar equal hai to next true section me jayega agar nhi to false section me jayega.

Man lijiye password 8 length ka hai aur humne 10 length diya to kaise match karega. Isliye ye phle hi out kr de rha hai.

To yha pr hume kaise pta chalega ki kitni length hai.

```
;-- rip:
0x00401244 4839c3          cmp rbx, rax
0x00401247 7416             je 0x40125f
```

To yha pr humari length aur original length in dono registers me pdi hai.

Registers ki value dekhne ke liye dr command chalate hai.

```
| 0x00401249 488030C80000. lea rdx  
:> dr  
rax = 0x00000008  
rbx = 0x00000016  
rcx = 0x0000000c  
rdx = 0x7ffee13e330c  
r8 = 0x00000000  
r9 = 0x7f5162a2fd50  
r10 = 0x004004a6  
r11 = 0x7f516299a660  
r12 = 0x004010d0  
r13 = 0x7ffee13e20f0  
r14 = 0x00000000  
r15 = 0x00000000  
rsi = 0x7ffee13e20f8  
rdi = 0x7ffee13e330c  
rsp = 0x7ffee13e1fc0  
rbp = 0x7ffee13e2000  
rip = 0x00401244  
rflags = 0x00000212  
orax = 0xfffffffffffffff  
:>
```

To yha pr **rax** ki value **8** hai. aur **rbx** ki value **16** hai.

So, humne 8 length ka input dala tha 12345678 aur **original length 16** hai.

So. **0x00000016** password ki original length hai.

Agar hum apne understanding me dekhna chahe to **drr** command fire karenge.

role	reg	value	refstr
R0	rax	18	8 .comment rax
	rbx	16	22 .comment rbx
A3	rcx	c	12 .comment rcx
A2	rdx	7ffee13e330c	[stack] rdi,rdx stack R W 0x3837363534333231 12345
A4	r8	0	0
A5	r9	7f5162a2fd50	/usr/lib/x86_64-linux-gnu/ld-2.31.so r9 library R
	r10	4004a6	4195494 /home/hellsender/yt/rev/crackme1 .dynstr r
en			
	r11	7f516299a660	/usr/lib/x86_64-linux-gnu/libc-2.31.so r11 library
	r12	4010d0	4198608 /home/hellsender/yt/rev/crackme1 .text en
ntry0	program	R X 'endbr64'	'crackme1'
	r13	7ffee13e20f0	[stack] r13 stack R W 0x2
	r14	0	0
	r15	0	0
A1	rsi	7ffee13e20f8	[stack] rsi stack R W 0x7ffee13e32eb
A0	rdi	7ffee13e330c	[stack] rdi,rdx stack R W 0x3837363534333231 12345
SP	rsp	7ffee13e1fc0	[stack] rsp stack R W 0x7ffee13e20f8
RP	rhn	7ffee13e2000	[stack] rhn stack R W 0x0

Hex **0x16** ki value **22** hai.

To hume **22 length** ka input dalna padega tabhi hum is address pr aa payenge.

```

0x401249 [og]
; 0x402018
; "Wrong Password."
0x00401249 488d3dc80d00. lea rdi, str.Wrong_Password.
; int puts(const char *s)
0x00401250 e82bfeffff call sym.imp.puts;[ob]
0x00401255 bf01000000 mov edi, 1
; void exit(int status)
0x0040125a e861feffff call sym.imp.exit;[oc]

```



```

0x40125f [oi]
0x0040125f 488b45c0 mov rax, qword
0x00401263 4883c008 add rax, 8
0x00401267 488b10 mov rdx, qword
0x0040126a 488d45d0 lea rax, [var_3]
0x0040126e 4889d6 mov rsi, rdx
0x00401271 4889c7 mov rdi, rax
; int strcmp(const char *s1, const char *
0x00401274 e837feffff call sym.imp.st
0x00401279 85c0 test eax, eax
0x0040127b 750e jne 0x40128b

```

Ok, ab hum program ko fir se run krte hai.

```

:> ood 1234567890123456789012
native-singlestep: No such process
Stepping failed!
child received signal 9
File dbg:///home/hellsender/yt/rev/crackme1 123456789012345678901
29238
:> dc
hit breakpoint at: 0x4011b6
:>

```

To hum jahan pr string compare ho rha hai wahi pr breakpoint lagate hai.

```

0x00401239      488D00        mov rax, qword [rax]
0x0040123c      4889c7        mov rdi, rax
0x0040123f      e84cfefefff  call sym.imp.strlen      ; s
0x00401244      4839c3        cmp rbx, rax
=< 0x00401247    7416          je 0x40125f
0x00401249      488d3dc80d00  lea rdi, str.Wrong_Password. ; 
0x00401250      e82bfefefff  call sym.imp.puts      ; i
0x00401255      bf01000000   mov edi, 1
0x0040125a      e861fefefff  call sym.imp.exit      ; v
-> 0x0040125f    488b45c0        mov rax, qword [var_40h]
0x00401263      4883c008       add rax, 8
0x00401267      488b10          mov rdx, qword [rax]
0x0040126a      488d45d0       lea rax, [var_30h]
0x0040126e      4889d6          mov rsi, rdx
0x00401271      4889c7          mov rdi, rax

```

Here, we put **breakpoint** and **continue**.

```

0x004012b4      5b            pop rbx
0x004012b5      5d            pop rbp
0x004012b6      c3            ret
:> db 0x00401244
:> dc
hit breakpoint at: 0x401244
:>

```

To ab hum dekhenge ki yha pr **rax** and **rbx** ki value equal ho gyi hai.

Assembly code:

```
0x00401244 4839c3      cmp rbx, rax
0x00401247 7416        je 0x40125f

0x401249 [og]
0x402018
"Wrong Password."
0x00401249 488d3dc80d00. lea rdi, str.Wrong_Password.
int puts(const char *s)
0x00401250 e82bfeffff    call sym.imp.puts;[ob]
0x00401255 bf01000000    mov edi, 1
void exit(int status)
0x0040125a 0041f...ffffe  call sym.imp.exit;[ob]
```

Memory dump:

```
0x40125f [oi]
0x0040125f 488b45c0      mov
0x00401263 4883c008      add
0x00401267 488b10      mov
0x0040126a 488d45d0      lea
0x0040126e 4889d6      mov
0x00401271 4889c7      mov
; int strcmp(const char *s1, c
0x00401274 0027f...ffffe  call
```

```
:> dr
rax = 0x000000016
rbx = 0x000010016
rcx = 0x00000001e
rdx = 0xffff83d402fe
r8 = 0x00000000
r9 = 0x7f37f8dead50
r10 = 0x004004a6
r11 = 0x7f37f8d55660
r12 = 0x004010d0
r13 = 0xffff83d3ec70
r14 = 0x00000000
r15 = 0x00000000
rsi = 0xffff83d3ec78
rdi = 0xffff83d40300
rsp = 0xffff83d3eb40
rbp = 0xffff83d3eb80
rip = 0x00401244
```

Ab hum yha aa chuke hai.

rd.

```
[0x40125f]
;-- rip:
0x0040125f 488b45c0    mov rax, qword [var_40h]
0x00401263 4883c008    add rax, 8
0x00401267 488b10    mov rdx, qword [rax]
0x0040126a 488d45d0    lea rax, [var_30h]
0x0040126e 4889d6    mov rsi, rdx
0x00401271 4889c7    mov rdi, rax
; int strcmp(const char *s1, const char *s2)
0x00401274 e837feffff  call sym.impstrcmp;[oh]
0x00401279 85c0    test eax, eax
0x0040127b 750e    jne 0x40128b
```

f t

```
0x40128b [ok]
; 0x402018
; "Wrong Password."
0x0040128b 488d3d860d00  lea rdi, str1
```

lea rdi, str_Correct_Password

Yha pr **strcmp** string ko compare ke liye use hota hai.

strcmp take two arguments as pointer of string and compare with each other.

Yha pr ye **humare password** aur **original password** ko check krne ke liye use ho rha hoga. Ki equal hai ki nhi.

Jaisa ki hum upar dekh skte hai ki **original password** and **humare input ka password** dono **rsi and rax** registers me mov ho rhe hai.

```
0x00401267 488b10          mov rdx, qword [rax]
0x0040126a 488d45d0        lea rax, [var_30h]
0x0040126e 4889d6          mov rsi, rdx
0x00401271 4889c7          mov rdi, rax
;-- rip:
; int strcmp(const char *s1, const char *s2)
0x00401274 e837feffff      call sym.impstrcmp;[0h]
0x00401279 85c0             test eax, eax
0x0040127b 750e             jne 0x40128b

f t
0x00401274 0x00401274
0x00401275 0x00401275
0x00401276 0x00401276
0x00401277 0x00401277
0x00401278 0x00401278
0x00401279 0x00401279
0x0040127a 0x0040127a
0x0040127b 0x0040127b
0x0040127c 0x0040127c
0x0040127d 0x0040127d
0x0040127e 0x0040127e
0x0040127f 0x0040127f
0x00401280 0x00401280
```

Let's check registers values.

```
:> dr
rax = 0xffff83d3eb50
rbx = 0x00000016
rcx = 0x0000001e
rdx = 0xffff83d402fe
r8 = 0x00000000
r9 = 0x7f37f8dead50
r10 = 0x004004a6
r11 = 0x7f37f8d55660
r12 = 0x004010d0
r13 = 0xffff83d3ec70
r14 = 0x00000000
r15 = 0x00000000
rsi = 0xffff83d402fe
rdi = 0xffff83d3eb50
rsp = 0xffff83d3eb40
rbp = 0xffff83d3eb80
rip = 0x00401274
rflags = 0x00000212
orax = 0xfffffffffffffff
```

Here, we know pointer store values addresses then we use drr to show registers values in readable format.

role	reg	value	refstr
R0	rax	7fff83d3eb50	[stack] rax,rdi stack R W 0x535f726570753353 S3uper_S3cr3t_Pa33w0rd
	rbx	16	22 .comment rbx
A3	rcx	1e	30 .comment rcx
A2	rdx	7fff83d402fe	[stack] rdx,rsi stack R W 0x3837363534333231 1234567890123456789012
A4	r8	0	0
A5	r9	7f37f8dead50	/usr/lib/x86_64-linux-gnu/ld-2.31.so r9 library R X 'endbr64' 'ld-2.31
	r10	4004a6	4195494 /home/hellsender/yt/rev/crackme1 .dynstr r10 program R 0x73006
en			

1234567890123456789012			
A3	rcx	1e	30 .comment rcx
A2	rdx	7fff83d402fe	[stack] rdx,rsi stack R W 0x3837363534333231 1234567890123456789012
A4	r8	0	0
A5	r9	7f37f8dead50	/usr/lib/x86_64-linux-gnu/ld-2.31.so r9 library R X 'endbr64' 'ld-2.31.so'
	r10	4004a6	4195494 /home/hellsender/yt/rev/crackme1 .dynstr r10 program R 0x73006e656c
en			
	r11	7f37f8d55660	/usr/lib/x86_64-linux-gnu/libc-2.31.so r11 library R X 'endbr64' 'libc-2.31
	r12	4010d0	4198608 /home/hellsender/yt/rev/crackme1 .text entry0,section..text,.text,_
ntry0	program	R X 'endbr64' 'crackme1'	
	r13	7fff83d3ec70	[stack] r13 stack R W 0x2
	r14	0	0
	r15	0	0
A1	rsi	7fff83d402fe	[stack] rdx,rsi stack R W 0x3837363534333231 1234567890123456789012
A0	rdi	7fff83d3eb50	[stack] rax,rdi stack R W 0x535f726570753353 S3uper_S3cr3t_Pa33w0rd
SP	rsp	/ffff83d3eb40	[stack] rsp stack R W 0x/ffff83d3ec78
BP	rbp	7fff83d3eb80	[stack] rbp stack R W 0x0
PC		401274	4195494 /home/hellsender/yt/rev/crackme1 .text entry0,section..text,.text,_

```

0x00401267 488b10          mov rdx, qword [rax]
0x0040126a 488d45d0        lea rax, [var_30h]
0x0040126e 4889d6          mov rsi, rdx
0x00401271 4889c7          mov rdi, rax
;-- rip:
; int strcmp(const char *s1, const char *s2)
0x00401274 e837feffff      call sym.imp.strcmp;[oh]
0x00401279 85c0             test eax, eax
0x0040127b 750e             jne 0x40128b

```

Sym.imp(strcmp) return **zero** if both strings are equal.

And test sirf **eax** ki value check karega zero or not.

Ab hum correct password find kr liya hai.

```
|  
|-----  
0x40127d [oj]  
; 0x402028  
; "Correct Password."  
lea rdi, str.Correct_Password.  
; int puts(const char *s)  
call sym.imp.puts;[ob]  
jmp 0x401297  
|-----  
|
```

```
→ rev ./crackme1 S3uper_S3cr3t_Pa33w0rd  
Correct Password.  
→ rev █  
█
```

Command	Description
r2 -d crackme1	
aaa	
afl	
db main	
dc	
pdf	
V	
VV	

```
#####
```

Challenge 3 - Find the correct password of crackme2

```
→ rev ./crackme2
```

```
Enter Password.
```

```
12345
```

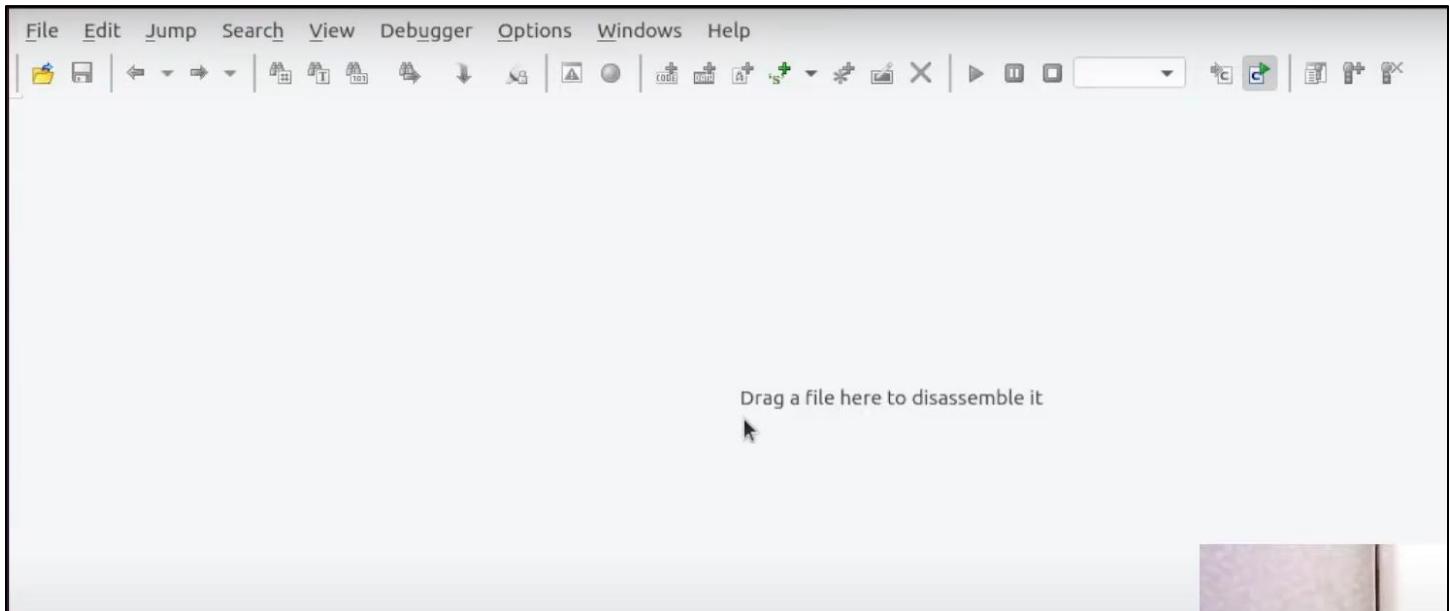
```
Wrong Password.
```

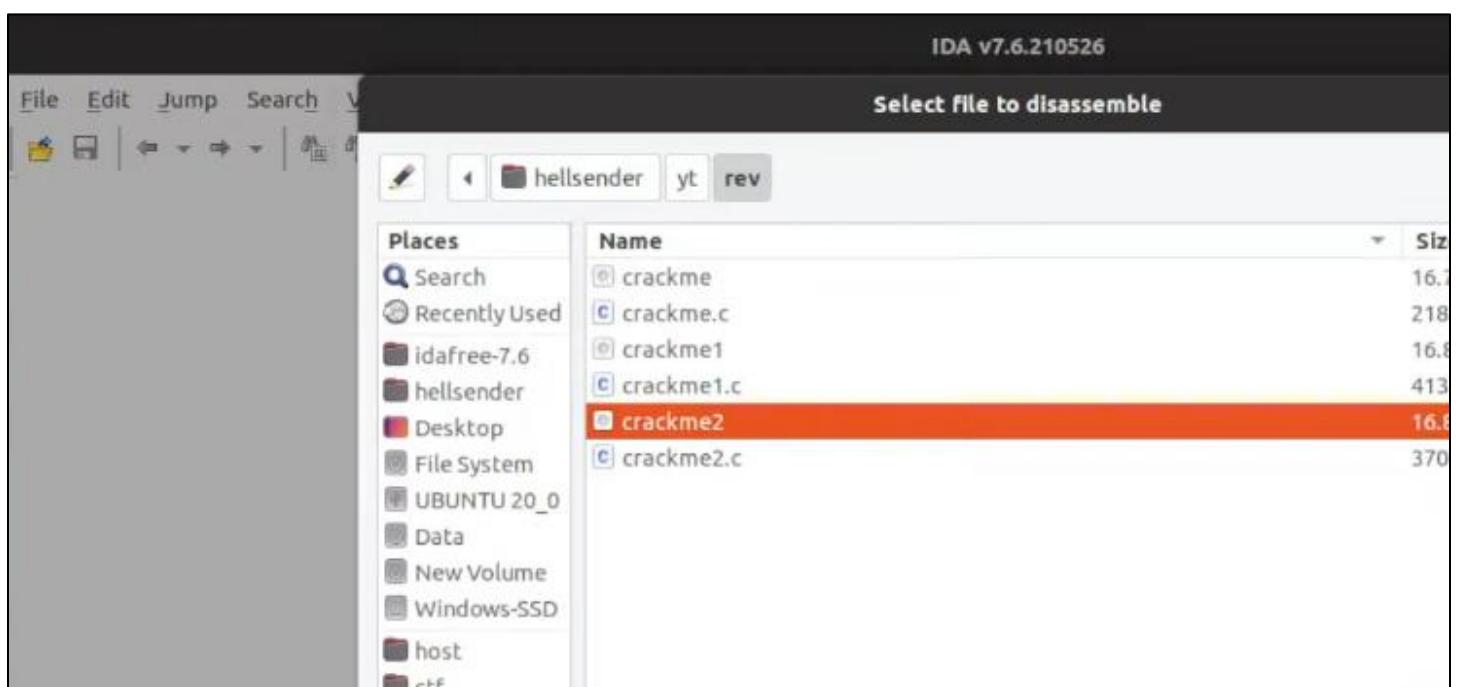
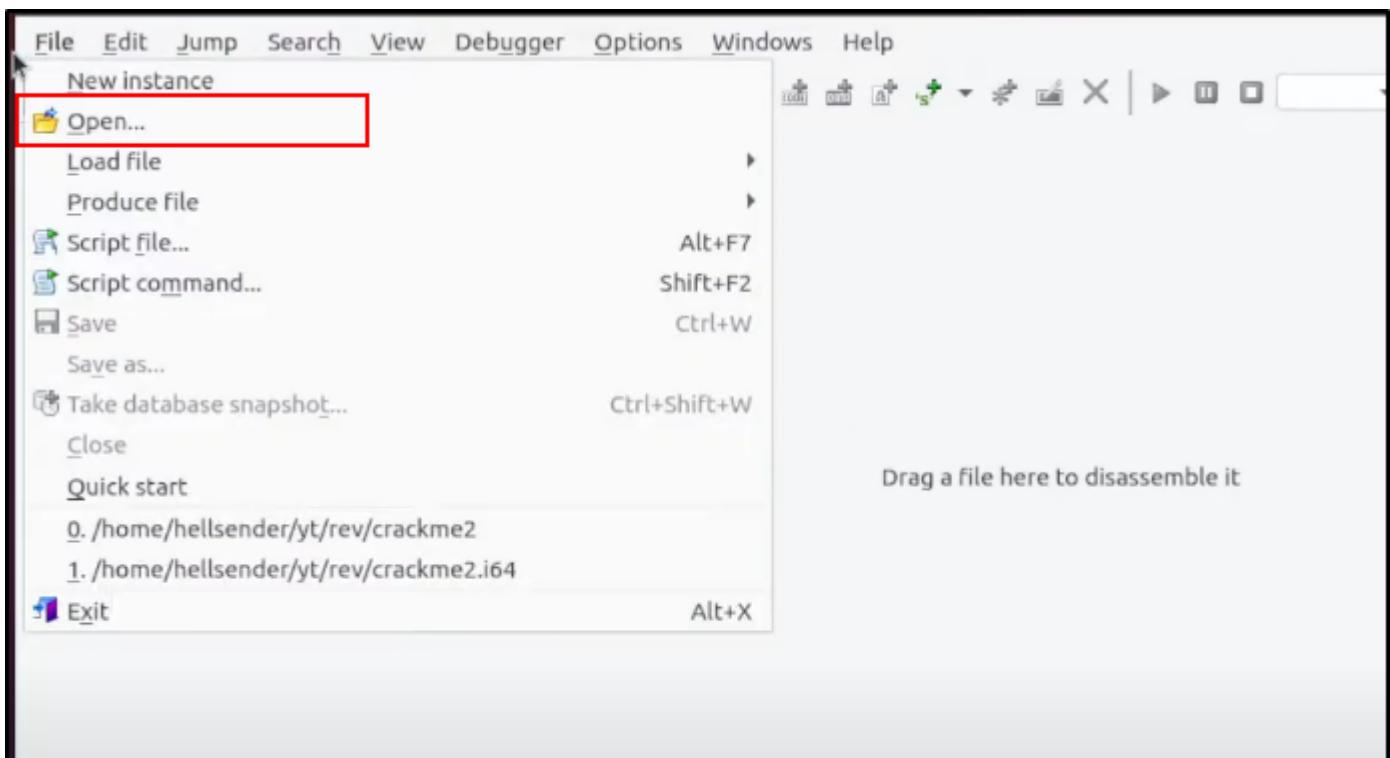
```
→ rev
```

Here, we will use **IDA debugger**.

The **IDA Debugger** is a dynamic analysis tool integrated with **IDA Pro**, widely used for **reverse engineering**. It supports **local and remote debugging** across multiple platforms and architectures. By combining **static and dynamic analysis**, it allows users to execute binaries, set breakpoints, inspect memory, analyze control flow, and debug effectively.

So, download and install IDA.





Load a new file

Load file /home/hellsender/yt/rev/crackme2 as

ELF64 for x86-64 (Executable) [elf64.so]

Binary file



Processor type (double-click to set)

- | | |
|---|---------------|
| Intel Pentium Pro (P6) with MMX | 80686p |
| Intel Pentium protected with MMX | 80586p |
| Intel Pentium real with MMX | 80586r |
| MetaPC (disassemble all opcodes) | metapc |

Loading segment 0x0000000000000000

Analysis

Enabled

Loading offset 0x0000000000000000

Indicator enabled

Kernel options 1

Kernel options 2

Options

- Loading options
- Fill segment gaps
- Load as code segment

- Create segments
- Create FLAT group
- Create imports segment

- Load res
- Rename
- Manual

Help

Cancel

OK

Click on ok.

IDA - crackme2 /home/kali/Desktop/crackme2

File Edit Jump Search View Debugger Options Windows Help

Local Linux debugger

Functions IDA View-A Hex View-1 Local Types Imports Exports

Function name

- `_init_proc`
- `sub_401020`
- `sub_401030`
- `sub_401040`
- `sub_401050`
- `sub_401060`
- `sub_401070`
- `_puts`
- `_strlen`
- `__stack_chk_fail`
- `_gets`
- `_exit`
- `_start`
- `dl_relocate_static_pie`

Line 28 of 29, /exit
Graph overview

Output

Hex-Rays Cloud Decompiler plugin has been loaded (v9.0.0.241217)
The decompilation hotkey is F5.
Please check the Edit/Plugins menu for more information.
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.

IDC

AU: idle Down Disk: 50GB

```
; Attributes: bp-based frame
; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near

var_44= dword ptr -44h
var_40= dword ptr -40h
var_3C= qword ptr -3Ch
var_34= dword ptr -34h
s= byte ptr -30h
var_8= qword ptr -8h

; __unwind {
endbr64
push rbp
mov rbp, rsp
sub rsp, 50h
mov rax, fs:28h
mov [rbp+var_8], rax
xor eax, eax
mov rax, 6472307724244050h
retf
}

main endp

```

100.00% (19,-21) (442,528) 000001B6 00000000004011B6: main (Synchronized with Hex View-1)

File Edit Jump Search View Debugger Options Windows Help

Local Linux debugger

Functions IDA View-A Hex View-1 Local Types Imports Exports

Function name

- `_init_proc`
- `sub_401020`
- `sub_401030`
- `sub_401040`
- `sub_401050`
- `sub_401060`
- `sub_401070`
- `_puts`
- `_strlen`
- `__stack_chk_fail`
- `_gets`
- `_exit`
- `_start`
- `dl_relocate_static_pie`

Line 28 of 29, /exit
Graph overview

Output

Hex-Rays Cloud Decompiler plugin has been loaded (v9.0.0.241217)
The decompilation hotkey is F5.
Please check the Edit/Plugins menu for more information.
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.

IDC

AU: idle Down Disk: 50GB

```
call _strlen
mov [rbp+var_40], eax
cmp [rbp+var_40], 0Bh
jz short loc_40122E

lea rdi, aWrongPassword ; "Wrong Password."
call _puts
mov edi, 0 ; status
call _exit

loc_40122E:
mov [rbp+var_44], 0
jmp short loc_40126B

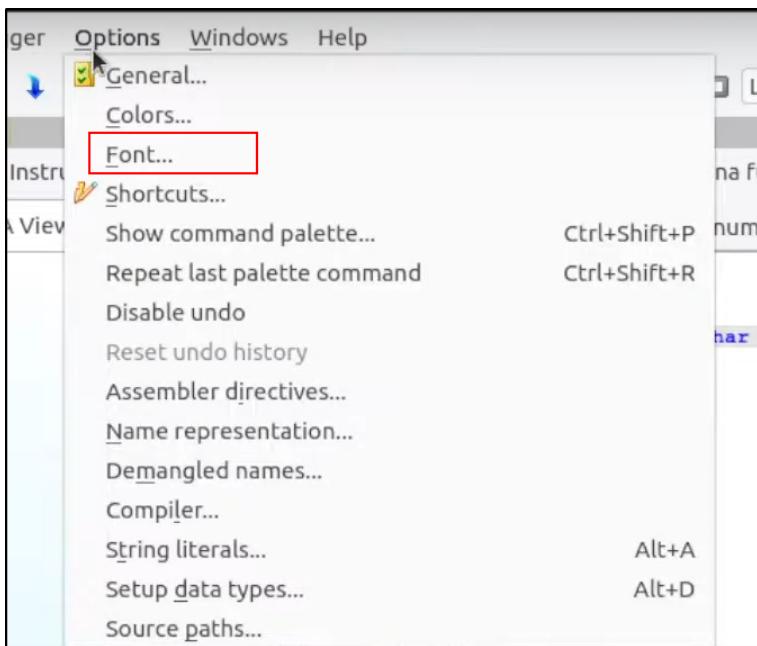
loc_40126B:
mov eax, [rbp+var_44]
cmp eax, [rbp+var_40]
jl short loc_401237

loc_401237:
mov eax, [rbp+var_44]
cdqe
movzx edx, byte ptr [rbp+rax+var_3C]
mov eax, [rbp+var_44]

lea rdi, aCorrectPasswo ; "Correct Passwo
call _puts
mov eax, 0
mov rcx, [rbp+var_8]
xor rcx, fs:28h
jz short locret_401298
```

100.00% (98,711) (1029,527) 000001B6 00000000004011B6: main (Synchronized with Hex View-1)

Make font big.



Yha pr abhi ye disassembler ka kam kr rha hai.

Disassembling means convert **machine code** in **assembly code**.

A screenshot of the IDA View-A assembly window. On the left, the 'Functions' pane shows several functions: register_tm_clones, __do_global_dtors_aux, frame dummy, main, __libc_csu_init, __libc_csu_fini, _term_proc, puts, strlen, __stack_chk_fail, and __libc_start_main. The 'main' function is selected and highlighted with an orange box. The main assembly code window displays the following assembly instructions:

```
s= byte ptr -30h
var_8= qword ptr -8
endbr64
push    rbp
mov     rbp, rsp
sub    rsp, 50h
mov     rax, fs:28h
mov     [rbp+var_8], rax
xor    eax, eax
mov     rax, 6472307724244050h
mov     [rbp+var_3C], rax
mov     [rbp+var_34], 533221h
lea     rdi, s           ; "Enter Passwo
call   _puts
lea     rax, [rbp+s]
mov     rdi, rax
mov     eax, 0
call   _gets
lea     rax, [rbp+1]
```

Here, it is showing all function of your program. As we know in high level programming language program is executed from **main** function.

Agar hum kisi bhi function pr double click karenge to ye us function ke portion ko highlight kr dega.

```
→ rev ./crackme2
Enter Password.
12345
Wrong Password.
→ rev
```

Jaisa ki humne program ko run krke dekh liya tha.

```
, __ unwind {
endbr64
push    rbp
mov     rbp,  rsp
sub    rsp,  50h
mov     rax,  fs:28h
mov     [rbp+var_8],  rax
xor    eax,  eax
mov     rax,  6472307724244050h
mov     [rbp+var_3C],  rax
mov     [rbp+var_34],  533221h
lea     rdi,  s          ; "Enter Password"
call    _puts
lea     rax,  [rbp+s]
mov     rdi,  rax
mov     eax,  0
call    _gets
lea     rax,  [rbp+s]
mov     rdi,  rax          ; s
call    _strlen
mov     [rbp+var_40],  eax
cmp     [rbp+var_40],  0Bh
jz      short loc_40122E
```

Yha pr lea instruction ek string (s) "enter password" ko **rdi** me mov kr rha hai. puts ko call kr rha hai write (print) krne ke liye.

```
[lea    rax,  [rbp+s]]
```

Yha lea instruction use ho rha hai rbp+s ke address ko rax me mov kr rha hai. jaisa ki jante hai mov me [] lagaye to iska mtlb value ko mov kr hai lekin lea me iska bipart hota hai. [] laga ho to address ko mov kr rha hai.

```
lea    rax, [rbp+s]
mov   rdi, rax
mov   eax, 0
call  _gets
```

Yha **_gets** use ho rha hai iska mtlb input liya ja rha hai user se. to **_gets** input leke **[rbp+s]** me store kr dega.

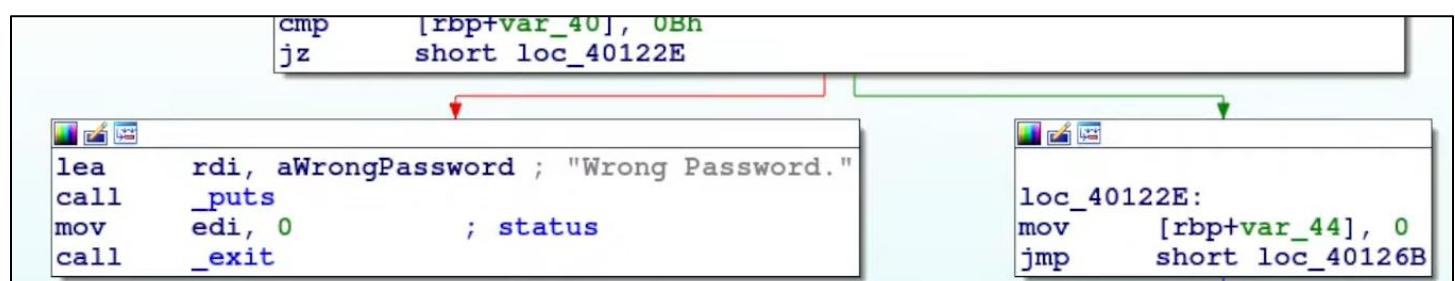
```
call  _gets
lea   rax, [rbp+s]
mov   rdi, rax      ; s
call  _strlen
```

Yha **_strlen** ko call ho rha hai **_strlen** kya krta hai. jo bhi value rdi me hoti hai uska length calculate krke **rax** me return krta hai.

```
mov   [rbp+var_40], eax
cmp   [rbp+var_40], 0Bh
jz    short loc_40122E
```

Yha pr **var_40** me user ke input ka length hai wo ja rha hai. uar hex ke 0B se compare ho rha hai. 0B ka value **11** means **11** se compare ki ja rhi hai.

(Jz means jump if zero / jump if equal) means agar equal hai to aage execute karo otherwise print kr do wrong password.



```

    mov rdi, rax      , s
    call _strlen
    mov [rbp+var_40], eax
    mov [rbp+var_44], 0
    jmp short loc_40124F

loc_40124F:
    mov eax, [rbp+var_44]
    cmp eax, [rbp+var_40]
    jl short loc_40121B

loc_40121B:
    mov eax, [rbp+var_44]

```

Yha variable **var_44** me **0** ko dala tha. Aur **var_44** ko mov krta hai **eax** ke andar. Ab ye **cmp** kr rha humare variable ke length **var_40** aur **eax** ko jisme **var_44** mov kiya gaya aur usme **0** store hai.

Uske bad hai **jl** means **var_40** less than hai **eax** ke value se to correct password print kr do otherwise location **loc_40121B** pr jump kr jao.

```

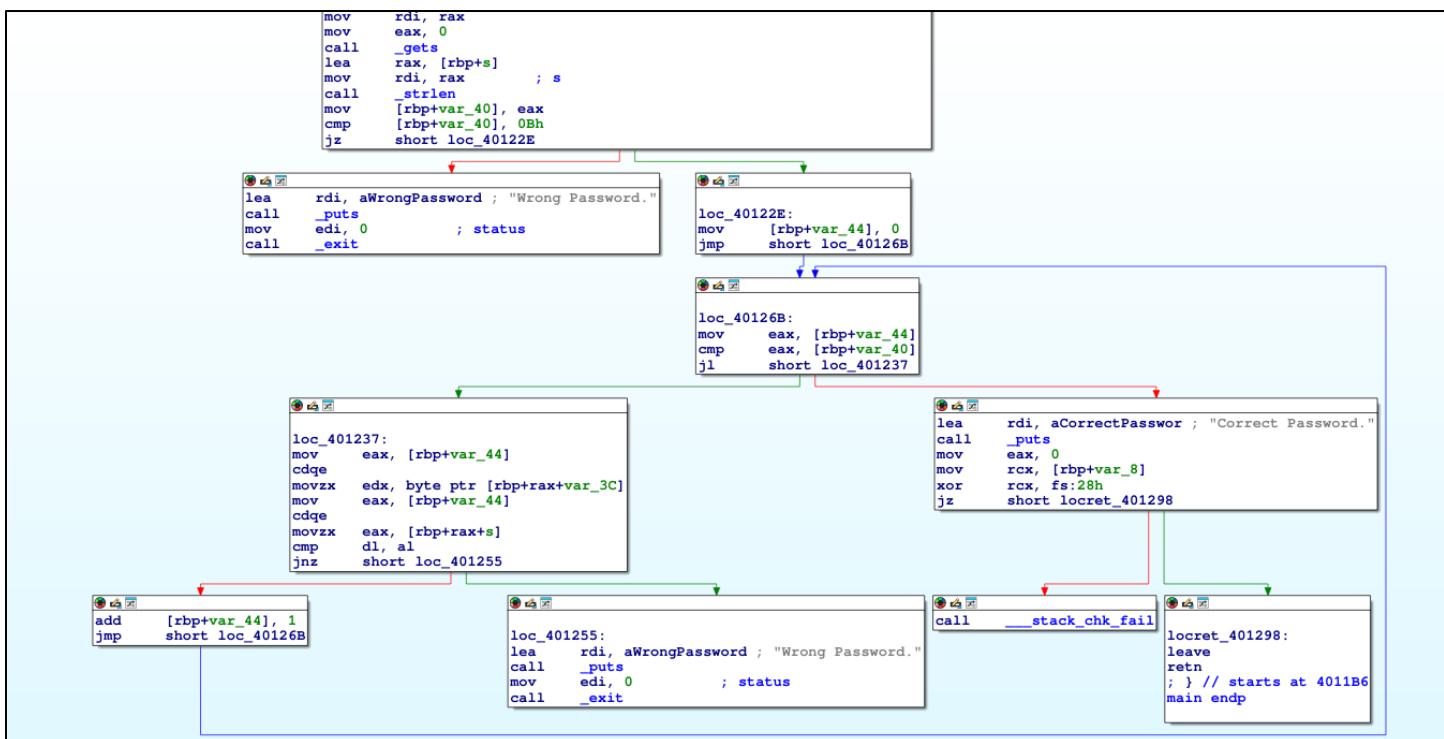
lea    rdi, aCorrectPasswor ; "Correct Password."
call   _puts
mov   eax, 0
mov   rcx, [rbp+var_8]
xor   rcx, fs:28h
jz    short locret_401298

```

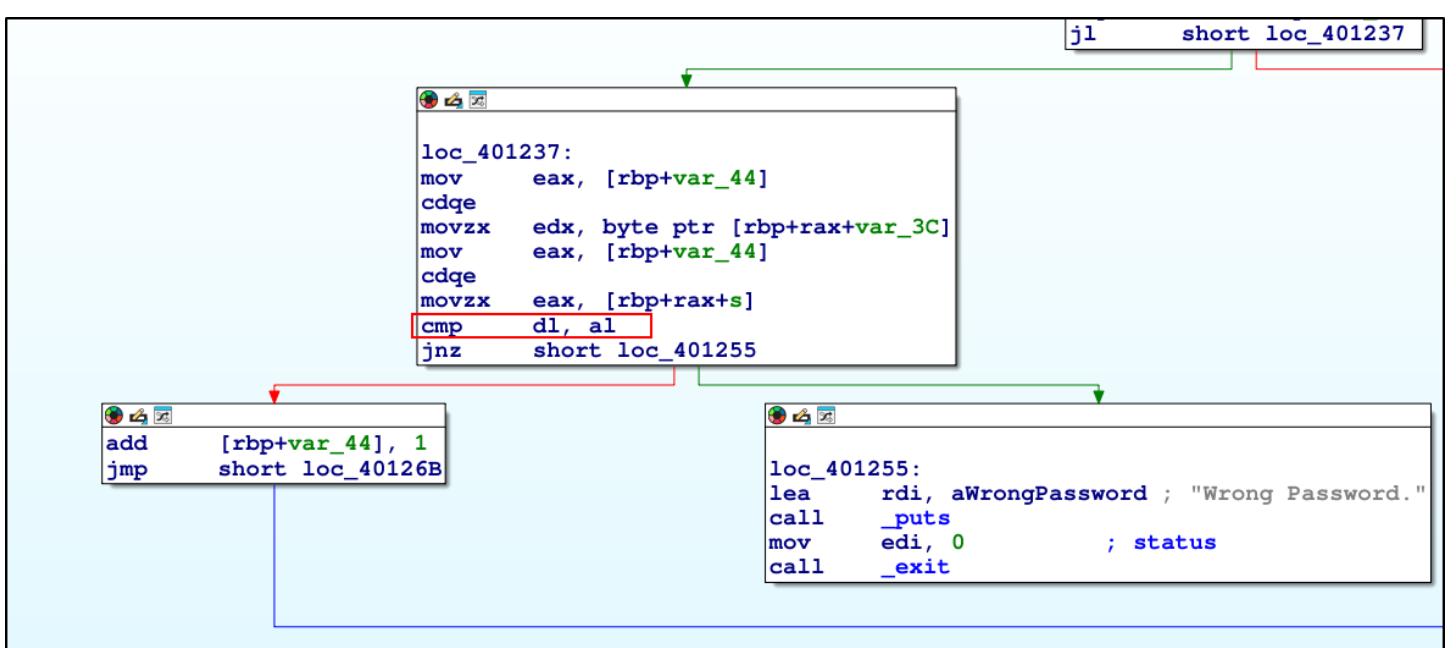
To yha pr humare input ki length kabhi minus me nhi ho sakti. Agar minus me hoti to ye correct password print kr deta.

To ye ab given location pr aa jayega.

```
loc_40121B:  
mov      eax, [rbp+var_44]  
cdqe  
movzx   edx, byte ptr [rbp+rax+var_3C]  
mov      eax, [rbp+var_44]  
cdqe  
movzx   eax, [rbp+rax+s]  
cmp      dl, al  
jnz      short loc_401239
```



Jaisa ki hum upar dekh skte hai ki ek loop chal rha hai aur condition ko check kr rha hai.



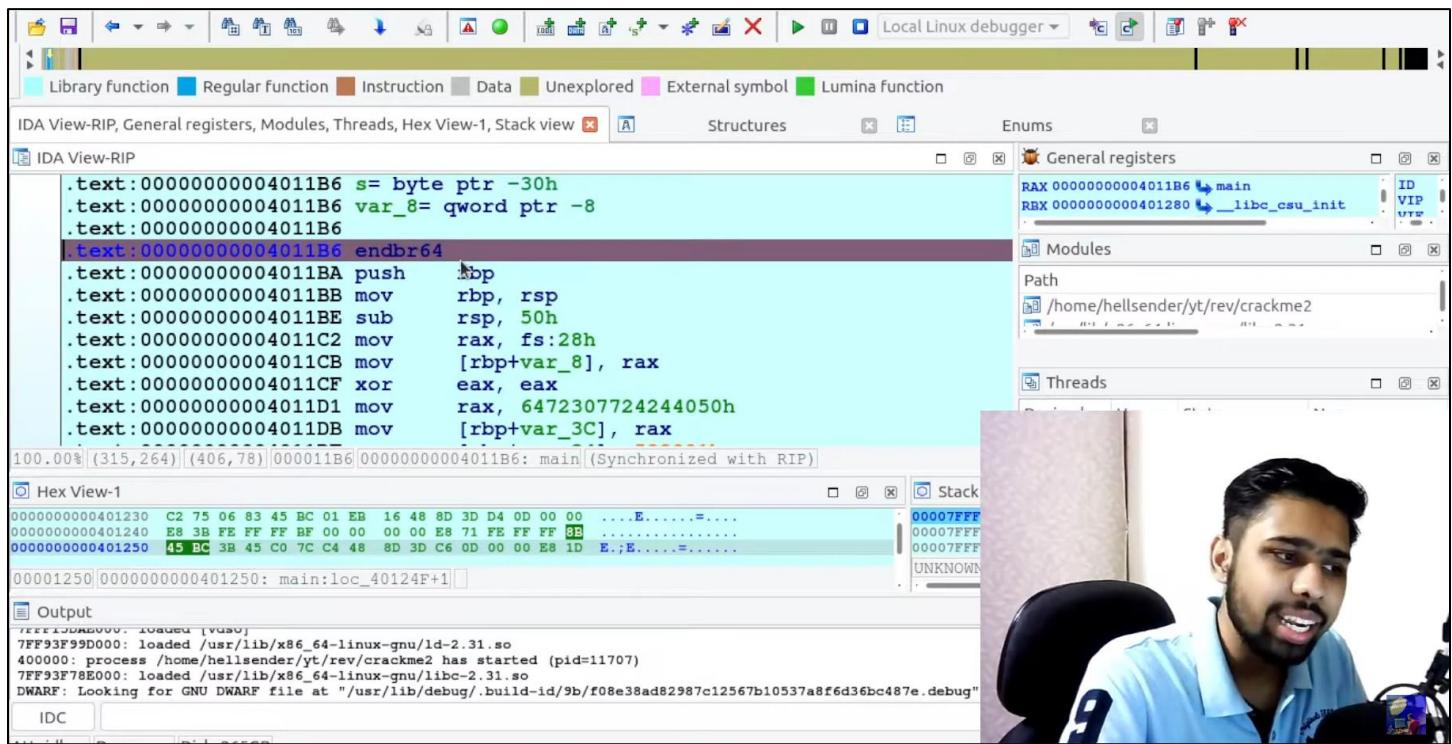
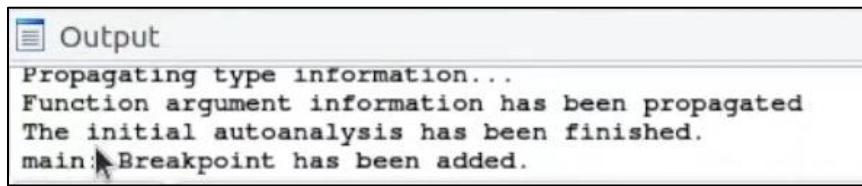
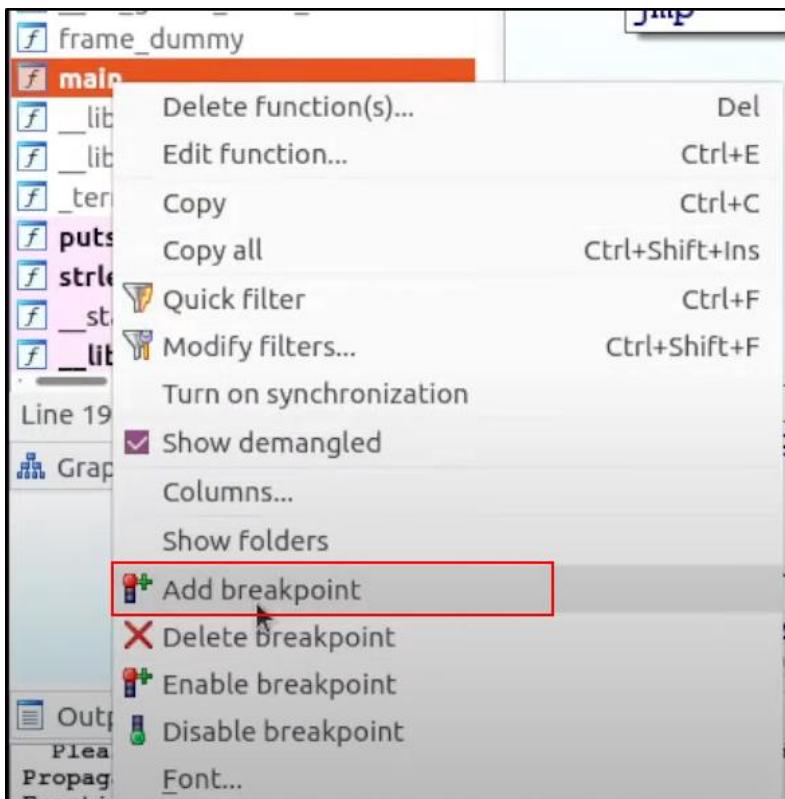
Yha **dl** aur **al** compare ho rha hai. it is very lower part of **rdx** and **rax**.

Agar dono equal nhi hue to ye **loc_401255** pr jayega aur **_exit** ko call kr dega.

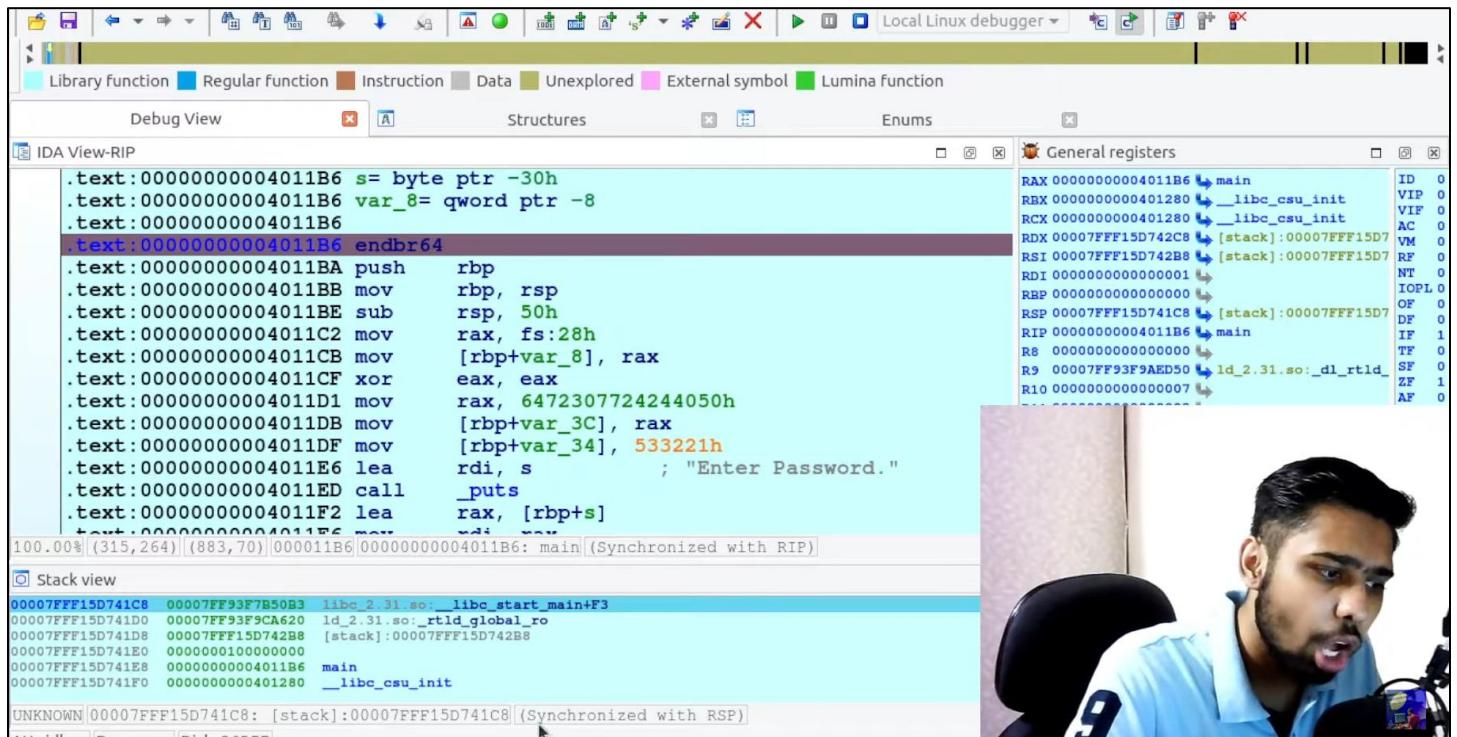
Agar dono equal hue to ye **var_44** me **1** add karega aur loop fir se aayega.

Ab hum debugger ki help se dekhenge ki al aur dl ke andar value kya hai. tabhi hum samjh payenge.

Go to main function and put a breakpoint. Right click on main function. And click on breakpoint.



Yha pr bahut messy si screen dikh rhi hogi. To yha hum kuchh windows ko cut kr dete hai.



IDA View-RIP

```
.text:00000000004011B6 s= byte ptr -30h
.text:00000000004011B6 var_8= qword ptr -8
.text:00000000004011B6
.text:00000000004011B6 endbr64
.text:00000000004011BA push    rbp
.text:00000000004011BB mov     rbp,  rsp
.text:00000000004011BE sub    rsp,  50h
.text:00000000004011C2 mov    rax, fs:28h
.text:00000000004011CB mov    [rbp+var_8],  rax
.text:00000000004011CF xor    eax,  eax
.text:00000000004011D1 mov    rax,  6472307724244050h
.text:00000000004011DB mov    [rbp+var_3C],  rax
.text:00000000004011DF mov    [rbp+var_34],  533221h
.text:00000000004011E6 lea     rdi, s           ; "Enter Password."
.text:00000000004011ED call   _puts
.text:00000000004011F2 lea     rax,  [rbp+s]
.text:00000000004011F6 mov    rbp,  [rbp+s]
```

100.00% (315,264) | (883,70) | 000001B6|00000000004011B6: main|(Synchronized with RIP)

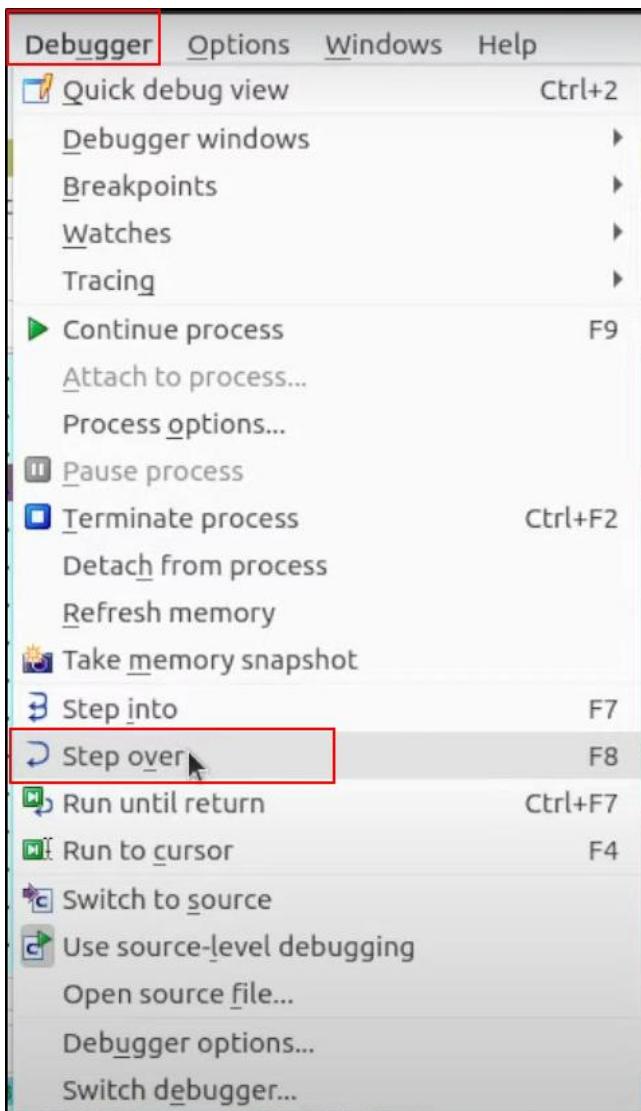
Stack view

00007FFF15D741C8 00007FF93F7B50B3 libc_2_31.so: __libc_start_main+F3
00007FFF15D741D0 00007FF93F9CA620 ld_2_31.so: __rtld_global_ro
00007FFF15D741D8 00007FFF15D742B8 [stack]:00007FFF15D742B8
00007FFF15D741E0 0000000100000000
00007FFF15D741E8 00000000004011B6 main
00007FFF15D741F0 0000000000401280 __libc_csu_init

UNKNOWN|00007FFF15D741C8: [stack]:00007FFF15D741C8|(Synchronized with RSP)

General registers

RAX 00000000004011B6 ↳ main	ID 0
RBX 0000000000401280 ↳ __libc_csu_init	VIP 0
RCX 0000000000401280 ↳ __libc_csu_init	VIF 0
RDX 00007FFF15D742C8 ↳ [stack]:00007FFF15D7	AC 0
RSI 00007FFF15D742B8 ↳ [stack]:00007FFF15D7	VM 0
RDI 0000000000000001 ↳	RF 0
RBP 0000000000000000 ↳	IOPL 0
RSP 00007FFF15D741C8 ↳ [stack]:00007FFF15D7	OF 0
RIP 00000000004011B6 ↳ main	DF 0
R8 0000000000000000 ↳	IF 1
R9 00007FF93F9AED50 ↳ ld_2_31.so: __dl_rtld_	TF 0
R10 0000000000000007 ↳	SF 0
	ZF 1
	AF 0



Here, we will press **f8** to step over. Means execute instruction step by step.

The screenshot shows the assembly view in IDA. The code is as follows:

```
.text:00000000004011B6 endbr64
.text:00000000004011BA push    rbp
.text:00000000004011BB mov     rbp, rsp
.text:00000000004011BE sub    rsp, 50h
.text:00000000004011C2 mov    rax, fs:28h
.text:00000000004011CB mov    [rbp+var_8], rax
.text:00000000004011CF xor    eax, eax
.text:00000000004011D1 mov    rax, 6472307724244050h
.text:00000000004011DB mov    [rbp+var_3C], rax
.text:00000000004011DF mov    [rbp+var_34], 533221h
.text:00000000004011E6 lea     rdi, s           ; "Enter Password."
.text:00000000004011ED call    _puts
.text:00000000004011F2 lea     rax, [rbp+s]
.text:00000000004011F6 mov    rdi, rax
.text:00000000004011F9 mov    eax, 0
```

The instruction `call _puts` is highlighted with a red box.

General registers	
RAX	6472307724244050 ↗
RBX	0000000000401280 ↗ _libc_csu_init
RCX	0000000000401280 ↗ _libc_csu_init
RDX	00007FFF15D742C8 ↗ [stack]:00007FFF15D
RSI	00007FFF15D742B8 ↗ [stack]:00007FFF15D
RDI	0000000000402004 ↗ "Enter Password."
RBP	00007FFF15D741C0 ↗ [stack]:00007FFF15D
RSP	00007FFF15D74170 ↗ [stack]:00007FFF15D
RIP	00000000004011ED ↗ main+37
R8	0000000000000000 ↗

Agar hum terminal pr jakr dekhe to enter password print ho jayega.

```
→ idafree-7.6 ./ida64
Gtk-Message: 23:38:08.013: Failed to load module "canberra-gtk-module"
QXcbClipboard: SelectionRequest too old
Enter Password.
```

Ok, ab hum aage execute krte hai.

```
.text:00000000004011E6 lea    rdi, s          ; "Enter Password."
.text:00000000004011ED call   _puts
.text:00000000004011F2 lea    rax, [rbp+s]
.text:00000000004011F6 mov    rdi, rax
.text:00000000004011F9 mov    eax, 0
.text:00000000004011FE call   _gets
.text:0000000000401203 lea    rax, [rbp+s]
```

Yha ye input mang rha hai. jb tk input nhi denge ye aage nhi badega.

```
→ idafree-7.6 ./ida64
Gtk-Message: 23:38:08.013: Failed to load module "canberra-gtk-module"
QXcbClipboard: SelectionRequest too old
Enter Password.
1234567890
```

Yha pr hum input dal ke enter marte hai.

```

.text:00000000004011FE call    _gets
.text:0000000000401203 lea     rax, [rbp+s] R11 0
.text:0000000000401207 mov     rdi, rax R12 0
.text:000000000040120A call    _strlen . s
.text:000000000040120F mov     [rbp+var_40]
00% (315, 394) (500, 257) 00001203 0000000000401203: main
stack view
7FFF15D74170 00000000000000C2
7FFF15D74178 00007FFF15D741A7 [stack]:00007FFF15D741A7
7FFF15D74180 2424405015D741A6
7FFF15D74188 0053322164723077
7FFF15D74190 3837363534333231
7FFF15D74198 0000000000003039
NOWN 00007FFF15D74170: [stack]:00007FFF15D74170 (Syncr)

```

Yha pr hum phle hi assembly ko pd kr samjha tha ki humari values yha pr store hogi.
Jaisa ki hum dekh skte hai.

Ab ye value **rax** ke andar mov hoga.

```

.text:00000000004011FE call    _gets
.text:0000000000401203 lea     rax, [rbp+s]
.text:0000000000401207 mov     rdi, rax ; s
.text:000000000040120A call    _strlen
.text:000000000040120F mov     [rbp+var_40], eax
.text:0000000000401212 mov     [rbp+var_44], 0

```

Ab **rbp+s** ki value **rax** me jayega fir **rax** ki value **rdi** me jayega.

Fir iska length nikal ke rax ke andar dal dega.

```

.text:0000000000401220 movzx  edx, byte ptr [rbp+rax+var_3C]
.text:0000000000401225 mov    eax, [rbp+var_44]
.text:0000000000401228 cdqe
.text:000000000040122A movzx  eax, [rbp+rax+s]
.text:000000000040122F cmp    dl, al
.text:0000000000401231 jnz    short loc_401239

```

Ab yha pr **dl aur al** compare ho rhe hai.

Inki values dekhte hai. **dl = rdx** and **al = rax**

General registers	
RAX	0000000000000031 ↵
RBX	0000000000401280 ↵ __libc_csu_init
RCX	0000000000000010 ↵
RDX	0000000000000050 ↵
RSI	3938373635343332 ↵
RDI	00007FFF15D74190 ↵ [stack] : 00007FFF15D

man ascii

31	1
50	P

Yha pr ye one by one check kr rha hai ki jo user input ki phli value original password ke phli value se match kr rhi hai.

```
→ idafree-7.6 ./ida64
Gtk-Message: 23:38:08.013: Failed to load module
QXcbClipboard: SelectionRequest too old
Enter Password. I
1234567890
```

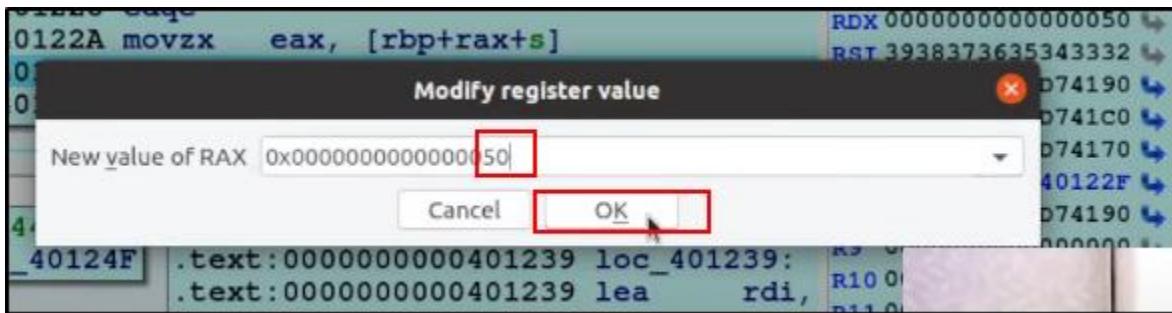
Agar match karegi to agli value ko check karega agar nhi match ki to wrong password print karke exit kr dega.

Yha pr hum inke hex value ko note krenge aur bad me ek password bna denge sbko mila kr.

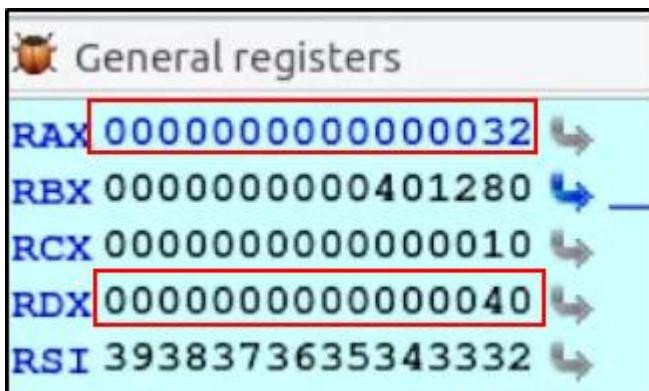
Ab hum al ke value pr double click karenge aur iski value change krke sahi value dal denge.

General registers	
RAX	0000000000000031 ↵
RBX	0000000000401280 ↵ __libc_c
RCX	0000000000000010 ↵
RDX	0000000000000050 ↵
RSI	3938373635343332 ↵
RDI	00007FFF15D74190 ↵ [stack] :
RBP	00007FFF15D741C0 ↵ [stack] :
RSP	00007FFF15D74170 ↵ [stack] :

Aur hum iski value change krke **31** se **50** kr denge. Jo ki sahi password ki value hai.

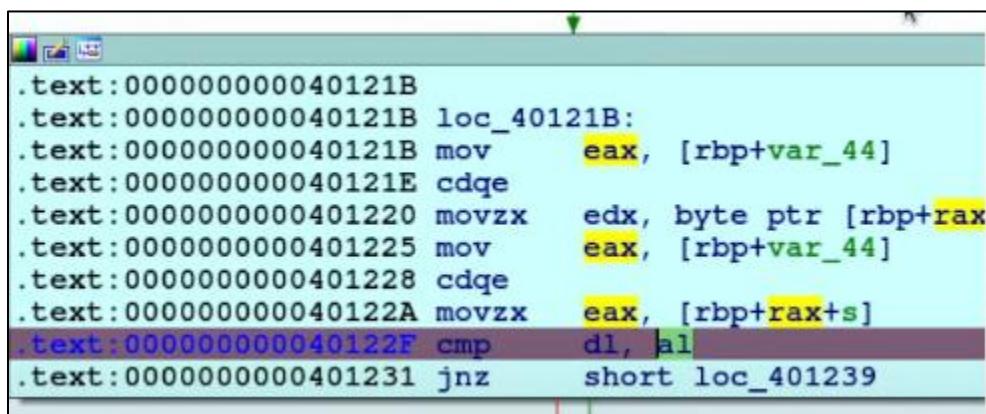


Badte hai new password value ke taraf.

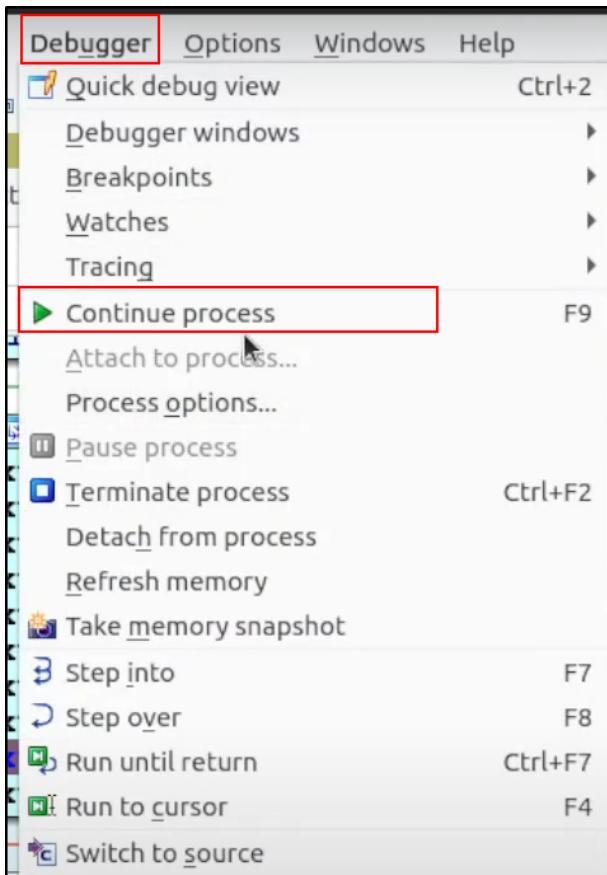


Is bar humari value hai 32 aur original password ki value hai 40. To pichhli bar ki tarah replace kr lenge.

To bar bar krne ke liye hum cmp pr breakpoint lagayenge.



Aur continue process kr denge. Ye cmp ke breakpoint pr ruk jayega aur hum value copy kr lenge.



Press **f9** for continue process.

So, here we noted all values.

```
→ idafree-7.6 man ascii
→ idafree-7.6 0x50402424773072642132
```

We can decode in using cyberchef tool.

Ya fir pwntool ke unhex tool se bhi kr sakte hai.

```
→ idafree-7.6 unhex 50402424773072642132
P@$$w0rd!2%
→ idafree-7.6
```

Finally we got password.

```
→ rev ./crackme2
Enter Password.
P@$$w0rd!2
Correct Password.
→ rev
```

```
#####
#####
```

Static And Dynamic Analysis Tools.

Stripped vs Non-Stripped Elf

There are some tools that can help us to make reverse engineering easy.

1) File command

Agar humko koi file milti hai. to kaise pta karenge ki kaun si file hai. Because linux me extension matter nhi krta hai.

```
→ rev file crackme
crackme: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64
.so.2, BuildID[sha1]=60bbafef09bf85f7b59c933c41ec32d82859bbd4, for GNU/Linux 3.2.0, not stripped
→ rev
```

Yha pr ye 64bit aur 32 bit bhi batayega jo ki reverse engineering perspective se important hai.

Aur hum dekh skte hai **dynamically linked** hai. jaisa ki hum jante hai bs humare program ka hi code enough nhi hota hai kisi binary ko run krne ke liye bahut sare other code bhi dale jate hai. jaise ki **_puts** function hai isko hum apne program me use krte hai lekin hum iska code khud se nhi likhte hai.

Wo code ek file se aata hai jise hum **libc** khte hai.

To **libc** ke code ko humare code ke sath jodne ke do tarike hote hai. ek hota hai **dynamically linked** aur dusra hota hai **statically link**.

Dynamically linked me ye hota hai ki after compilation jaise ki **_puts** function ka code libc me hi rhta hai bs uska reference se hum access krte hai. means humara program jata hai aur whi pdkr wahi execute kr deta hai. dynamically link me file ka size chota hota hai.

Aur **statically linked** me ye hota hai ki **_puts** ka code humare code me uth kr chala aayega. Wahi se pdkr use execute kr deta hi. Statically me file ka size bda hota hai.

Ab humare pass do tarike ki binary hota hai. ek hoti hai **stripped** aur dusri hota hai, **not stripped**.

Stripped binary ka purpose ye hota hai ki aap jitne km se km english ke word use kr sako. Jisse file ka size chota hota hai aur reverse engineering dificult hoti hai. jaise ki reverse engineer ko pta chal jaye ki **main** function yha se start ho rha hai. to uska kam aasan ho jayega.

Jaisa ki hum jante hai ki **main** function ka jo word main hai wo program ko execute krne ke liye jaruri nhi hai. hum uske address se bhi execute kr skte hai. ye reverse engineer ka kam aasan kr deta hai.

Aur **not stripped** theek iske biprit hota hai usme english words bahut jyada use hota hai. aur wo debugging purpose ke liye banaya jata hai.

Jb **dynamically linked** hota hai to ye **libc** ka code use kr rha hota hai. to **libc** file kahan hoti hai.

```
→ rev ldd crackme
    linux-vdso.so.1 (0x00007ffe7db49000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f26919c4000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f2691bd3000)
→ rev
```

.**so** means shared object file. Jaisa ki hum assembly me object file banate the fir link krte the yha pr bhi ye object file hai. shared object hai.

Aur ye bahut important file agar hum ise delete kr de to system chalna band kr de ya next time boot hi na ho paye. Jaisa ki hum jante hai linux pura C se bna hai aur backend me libc ka use krta hai.

Iska use lete hai c program jo hum built-in function likhte ho wo yha stored rhte hai.

To un symbols ko hum kaise pd skte hai uske liye ek tool aata hai **nm**.

```
ubuntu@ip-172-31-8-118:~$ nm crackme
0000000000403e20 d _DYNAMIC
0000000000404000 d _GLOBAL_OFFSET_TABLE_
0000000000402000 R _IO_stdin_used
000000000040217c r __FRAME_END__
0000000000402038 r __GNU_EH_FRAME_HDR
0000000000404040 D __TMC_END__
0000000000404040 B __bss_start
0000000000404030 D __data_start
0000000000401140 t __do_global_dtors_aux
0000000000403e18 d __do_global_dtors_aux_fini_array_entry
0000000000404038 D __dso_handle
0000000000403e10 d __frame_dummy_init_array_entry
    w __gmon_start__
0000000000403e18 d __init_array_end
0000000000403e10 d __init_array_start
    U __isoc99_scanf@@GLIBC_2.7
0000000000401260 T __libc_csu_fini
00000000004011f0 T __libc_csu_init
    U __libc_start_main@@GLIBC_2.2.5
    U __stack_chk_fail@@GLIBC_2.4
00000000004010c0 T __dl_relocate_static_pie
0000000000404040 D _edata
0000000000404048 B _end
0000000000401268 T _fini
0000000000401000 T _init
0000000000401090 T _start
0000000000404040 b completed.8060
0000000000404030 W data_start
00000000004010d0 t deregister_tm_clones
0000000000401170 t frame_dummy
0000000000401176 T main
    U puts@@GLIBC_2.2.5
0000000000401100 t register_tm_clones
ubuntu@ip-172-31-8-118:~$ |
```

To yha pr kuchh symbols ki jarurat hoti hai. unke bina program nhi chal payega. Aur **symbols** jaise main iske bina bhi program chal jayega.

Yha symbols aur machine code me difference hai. symbols upar jo likhe hai wo hai aur english word jaisa ki humne password likha.

Agar hum is binary ko **stripped** bna dunga to bahut sare symbols rhege. Because wo jaruri hai binary ko run krne ke liye. Aur bahut sare ht jayenge. Because wo jaruri nhi hai binary ko run krne ke liye.

Kya ek binary ke andar sirf symbols aate hai. nhi aate balki unke sath english ke words bhi aate hai.

Jaise ki hum password likh diye to machine code me convert nhi hoga. Kyoki use execute nhi krna hota hai. use to bs match krna hota hai.

Yha pr humne jo code likha wo to machine code me convert ho gya. Jaise kuch chije hoti hai jo machine code me convert nhi hoti. Jaise humne password check ka program banaya. Sb kuchh machine code me convert ho gya lekin jo password hai wo machine code me convert nhi hoga.

Usko to execute nhi krna processor ko wo to bs ek string hai english word hai.

To un string ko print krne ke liye hum **strings** command ka use krte hai.

Strings command(ya humara terminal) sirf un hi character(english words) ko print kr pata hai jo **ascii table** me milta hai. uske alawa kisi chij ko nhi print kr pata hai.

```
ubuntu@ip-172-31-8-118:~$ strings crackme1
/lib64/ld-linux-x86-64.so.2
libc.so.6
exit
puts
__stack_chk_fail
strlen
strcmp
__libc_start_main
GLIBC_2.4
GLIBC_2.2.5
__gmon_start__
H=P@@
S3uper_SH
3cr3t_PaH
33w0f
[]A\A]A^A_
Argument 1 Missing.
Wrong Password.
Correct Password.
.*3$"
GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
crtstuff.c
deregister_tm_clones
```

Aur hum niche dekhe to

```
.plt.sec
.text
.fini
.rodata
.eh_frame_hdr
.eh_frame
.init_array
.fini_array
.dynamic
.got
.got.plt
.data
.bss
.comment
ubuntu@ip-172-31-8-118:~$ |
```

Yha pr **.bss** aur **.data** section show ho rha hai.

To ye symbols aur sath hi sath user defined jo english ke word hai use bhi dikhaye.

Ye bahut important tool hai kyoki ye bahut sari information leak kr deta hai.

Agar hum ise cat kare to ye kuchh is tarah se show karega.

```
???H??HoHootHoetP@@@ff.@@@=+.uUH@@Z@@@. ]D@ff.@@@UH@@SH@@8@}{H@@dH% (H@@E@@H@@S3uper_S
@@@a@@H@@E@@H@@H@@E@@H@@7@@@uH@=H9@tH@=+
@@@H@@M@H3
        %(t@@@H@@8[ ]@f@@@AWL@=C+AVI@@AUI@@ATA@@UH@-4+SL)@H@@@H@@t1@@L@@L@@D@@A@@H@@H9@@u@H
@@@H@@Argument 1 Missing.Wrong Password.Correct Password.@@@@D@@@ \@@@pz@@@4zRx
@ @
]
E@@@: *3$"l@@@P$@@@E@C
    D@@@eF@I@E @E( @D@H@G@n8A0A(B BB@@@@@@ @@@ @
8@>@@@@@@ @@ @
k
@@x`@00      @@O@@@@P@`@p@GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0@8@X@|@@@@@@@
@@
```

Ye kuchh english ke words bhi print kr rha hai aur jo ascii table ke bahar ke word hai. to ye print nhi kr pa rha hai.

Ab hum stripped aur not stripped ka example dekhte hai.

```
→ rev strings crackme1.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, char const *argv[])
    char password[] = "S3uper_S3cr3t_Pa33w0rd";
    if (argc < 2) {
        puts("Argument 1 Missing.");
        exit(1);
    if (strlen(password) != strlen(argv[1])) {
        puts("Wrong Password.");
        exit(1);
    if (!(strcmp(password, argv[1]))) {
        puts("Correct Password.");
    else {
        puts("Wrong Password.");
    return 0;
→ rev
```

Ab hum is code ka non stripped binary banayenge **gcc** ke help se.

```
→ rev gcc crackme1.c -o crackme1
```

Ab hum stripped binary banayenge.

```
→ rev gcc -s crackme1.c -o crackme1_stripped
```

Yha pr **-s** lagane se stripped binary banegi.

Dono ka strings krke dekhenge stripped aur not stripped binary ka.

```
→ rev strings crackme1 > file1
→ rev strings crackme1_stripped > file2
```

Ab hum iska difference nikalenge meld tool ki help se.

```
→ rev meld file1 file2
```

Meld File Edit Changes View

Save Undo

file1 — file2

File1

```
/lib64/ld-linux-x86-64.so.2
libc.so.6
exit
puts
__stack_chk_fail
strlen
__cxa_finalize
strcmp
__libc_start_main
GLIBC_2.4
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
u+UH
S3uper_SH
3cr3t_PaH
33w0f
[]A\A]A^A_
Argument 1 Missing.
Wrong Password.
Correct Password.
:*3$"
```

File2

```
/lib64/ld-linux-x86-64.so.2
libc.so.6
exit
puts
__stack_chk_fail
strlen
__cxa_finalize
strcmp
__libc_start_main
GLIBC_2.4
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
u+UH
S3uper_SH
3cr3t_PaH
33w0f
[]A\A]A^A_
Argument 1 Missing.
Wrong Password.
Correct Password.
:*3$"
```

Yha pr abhi tk to same hai.

Meld File Edit Changes View

Save Undo

file1 — file2

File1

```
puts@@GLIBC_2.2.5
edata
strlen@@GLIBC_2.2.5
__stack_chk_fail@@GLIBC_2.4
__libc_start_main@@GLIBC_2.2.5
data_start
strcmp@@GLIBC_2.2.5
__gmon_start__
__dso_handle
_IO_stdin_used
__libc_csu_init
bss_start
main
exit@@GLIBC_2.2.5
__TMC_END__
_ITM_registerTMCloneTable
__cxa_finalize@@GLIBC_2.2.5
.symtab
.strtab
.shstrtab
.interp
.note.gnu.property
.note.gnu.build-id
.note.ABI-tag
```

File2

```
GLIBC_2.4
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
u+UH
S3uper_SH
3cr3t_PaH
33w0f
[]A\A]A^A_
Argument 1 Missing.
Wrong Password.
Correct Password.
:*3$"
GCC: (Ubuntu 9.3.0-1
.shstrtab
.interp
.note.gnu.property
.note.gnu.build-id
.note.ABI-tag
.gnu.hash
.dynsym
.dynstr
.gnu.version
```

Ab hum yha dekh skte hai green box me jo bhi content hai wo extra hai. **stripped** me bahut sare symbols hta diye usi me ek symbol tha **main**. Use bhi hta diya gaya.

Yha pr hum **not stripped** binary ko **gdb** ke andar function ko dekhte hai.

```
Non-debugging symbols:  
0x0000000000001000 _init  
0x0000000000001080 __cxa_finalize@plt  
0x0000000000001090 puts@plt  
0x00000000000010a0 strlen@plt  
0x00000000000010b0 __stack_chk_fail@plt  
0x00000000000010c0 strcmp@plt  
0x00000000000010d0 exit@plt  
0x00000000000010e0 __start  
0x0000000000001110 deregister_tm_clones  
0x0000000000001140 register_tm_clones  
0x0000000000001180 __do_global_dtors_aux  
0x00000000000011c0 frame_dummy  
0x00000000000011c9 main  
0x00000000000012d0 __libc_csu_init  
0x0000000000001340 __libc_csu_fini  
0x0000000000001348 __fini  
(gdb) q
```

In stripped binary:

```
Non-debugging symbols:  
0x0000000000001080 cxa_finalize@plt  
0x0000000000001090 puts@plt  
0x00000000000010a0 strlen@plt  
0x00000000000010b0 __stack_chk_fail@plt  
0x00000000000010c0 strcmp@plt  
0x00000000000010d0 exit@plt  
(gdb) =====  
=====
```

Jb hum binary ko analysis krte hai. to ye do types se hota hai.

1. Static analysis

2. Dynamic analysis

Static analysis me hum binary ko bina run kiye analysis krte hai. ab kisi binary ki hum assembly pd rhe hai wo static analysis hai bina run kiye bhi hum assembly ko pd skte hai.

Aur **dynamic analysis** me hum binary ko run krke analysis krte hai. debugging dynamic analysis me aata hai.

Ab tk humne jitna bhi analysis kiya wo static analysis kiya usme ek bhi bar binary ko run hi kiya.

Ab hum tool use karenge **ltrace**. Ye hume batayega ki hum libc ke kaun-2 se function ko use kr rhe hai. **Ye libc ke function ko with argument dikhata hai.**

```
→ rev ltrace ./crackme2
```

Yha pr hume **full path** dena pdta hai.

```
→ rev ltrace ./crackme2
puts("Enter Password") = 15
gets(0x7ffe177d9180, 0x1d012a0, 0, 0x7f11ba82a1e7)
```

To ye yha wait kr rha hai humare input ka to hum ise input dete hai.

```
→ rev ltrace ./crackme2
puts("Enter Password") = 15
gets(0x7ffe177d9180, 0x1d012a0, 0, 0x7f11ba82a1e7) abcdef
) = 0x7ffe177d9180
strlen("abcdef")
Wrong Password.
exit(0 <no return ...>
+++ exited (status 0) +++
→ rev
```

Ek aur example with crackme1

```
ubuntu@ip-172-31-8-118:~/Documents$ ltrace ./crackme1 abc123
strlen("S3uper_S3cr3t_Pa33w0rd")
strlen("abc123")
puts("Wrong Password."Wrong Password.
)
exit(1 <no return ...>
+++ exited (status 1) +++
ubuntu@ip-172-31-8-118:~/Documents |
```

Yha pr hum dekh skte hai isne password leak kr diya. **Strlen** function ke sath.

strace tool hume ye batata hai ki kaun-2 se system call ko ye call kr rha hai

jaisa ki hum jante hai ki **_puts** function bs fancy sa function hai. backend me ye **write syscall** ko call krta hai.

Niche dekhte hai.

```

fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x1), ...}) = 0
brk(NULL)                                     = 0x1140000
brk(0x1161000)                                = 0x1161000
write(1, "Enter Password\n", 15)Enter Password
)      = 15
fstat(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x1), ...}) = 0
read(0, 1234567890  Our input
"1234567890\n", 1024)                      = 11
write(1, "Wrong Password.\n", 16)Wrong Password.
)      = 16
exit_group(0)                                 = ?
+++ exited with 0 +++

```

Yha pr sabhi syscall ko show kr rha hai. Aadhe se jyada humare kam ke nhi hote hai.

Ye dynamic analysis kr rha hai.

Yha ek tool hota hai jo sb kuchh dikha data hai. ltrace aur strace chhadkr aur ye static analysis me help krtा hai.

```

ubuntu@ip-172-31-8-118:~$ readelf -a crackme1
ELF Header:
  Magic:  7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF64
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: Advanced Micro Devices X86-64
  Version: 0x1
Entry point address: 0x4010d0
Start of program headers: 64 (bytes into file)
Start of section headers: 14864 (bytes into file)
Flags: 0x0
Size of this header: 64 (bytes)
Size of program headers: 56 (bytes)
Number of program headers: 13
Size of section headers: 64 (bytes)
Number of section headers: 31
Section header string table index: 30

Section Headers:
[Nr] Name          Type            Address        Offset
    Size          EntSize        Flags  Link  Info  Align
[ 0]                         NULL           0000000000000000 00000000
    0000000000000000 0000000000000000          0     0     0
[ 1] .interp       PROGBITS        0000000000400318 00000318
    000000000000001c 0000000000000000        A     0     0     1

```

Magic address se linux identify krtा hai ki kaun si type ki file hai.

Section headers me kaun sa section hai uska information hai.

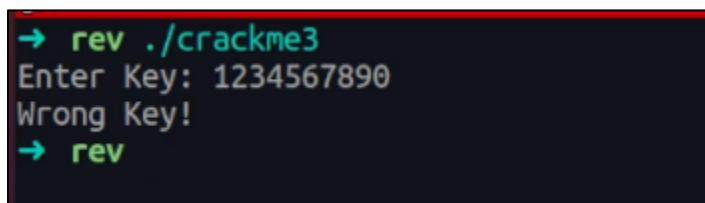
Jaise **.text section**

.bss section etc.

Aur bhi bahut kuchh.

```
#####
```

Most Used Debugger. Crackme – 3



```
→ rev ./crackme3
Enter Key: 1234567890
Wrong Key!
→ rev
```

Ye binary **keygen** category ki hai.

Jaisa ki hum jante hai ki **keygen** program ke bahut sare **key** combination bn sakte hai.

Yha pr hum sare key dal skte hai jo ki sahi hai.

To yha pr hume program ke algorithm ko samjhna pdega to hume pta chaleg ki iski bahut sare key ban skti hai.

Is binary ko crack karne ke liye hum **ghidra** ka use karenge.

Hum **gdb** me apna binary load krte hai. aur function dekhne ki koshish krte hai.

```

Non-debugging symbols:
0x00000000000401000 _init
0x00000000000401070 puts@plt
0x00000000000401080 __stack_chk_fail@plt
0x00000000000401090 printf@plt
0x000000000004010a0 __isoc99_scanf@plt
0x000000000004010b0 __start
0x000000000004010e0 __dl_relocate_static_pie
0x000000000004010f0 deregister_tm_clones
0x00000000000401120 register_tm_clones
0x00000000000401160 __do_global_dtors_aux
0x00000000000401190 frame_dummy
0x00000000000401196 check key
0x000000000004011da main
0x00000000000401270 __libc_csu_init
0x000000000004012e0 __libc_csu_fini
0x000000000004012e8 __fini
(gdb) 

```

Yha pr do functions kam ke dikh rhe hai. **check key** aur **main** function.

Ab hum **main** ko disassemble krte hai.

```

(gdb) disassemble main
Dump of assembler code for function main:
0x000000000004011da <+0>:    endbr64
0x000000000004011de <+4>:    push   rbp
0x000000000004011df <+5>:    mov    rbp,rsp
0x000000000004011e2 <+8>:    sub    rsp,0x10
0x000000000004011e6 <+12>:   mov    rax,QWORD PTR fs:0x28
0x000000000004011ef <+21>:   mov    QWORD PTR [rbp-0x8],rax
0x000000000004011f3 <+25>:   xor    eax, eax
0x000000000004011f5 <+27>:   lea    rdi,[rip+0xe08]      # 0x402004
0x000000000004011fc <+34>:   mov    eax,0x0
0x00000000000401201 <+39>:   call   0x401090 <printf@plt>
0x00000000000401206 <+44>:   lea    rax,[rbp-0x10]
0x0000000000040120a <+48>:   mov    rsi, rax
0x0000000000040120d <+51>:   lea    rdi,[rip+0xd0c]      # 0x402010
0x00000000000401214 <+58>:   mov    eax,0x0
0x00000000000401219 <+63>:   call   0x4010a0 <__isoc99_scanf@plt>
0x0000000000040121e <+68>:   mov    eax,DWORD PTR [rbp-0x10]
0x00000000000401221 <+71>:   mov    edi, eax

```

Yha ye comment krke dikha deta hai ki kaun si chij kisme mov ho rhi hai.

Jaise ki **rdi** me **0x402004** mov ho rha hai.

To hum is value ko dekh lete hai kya hai.

```
End of assembler dump.
```

```
(gdb) x/s 0x402004
```

```
0x402004: _ "Enter Key: "
```

Hex value ko human readable format me covert krne ke liye **x/s** use karenge. **x/s** means examine string.

Ab hum aage chale to ek **_scanf** function hai. jo ki do argument leta hai.

```
0x0000000000401206 <+44>:    lea    rax,[rbp-0x10]
0x000000000040120a <+48>:    mov    rsi,rax
0x000000000040120d <+51>:    lea    rdi,[rip+0xd9c]      # 0x402010
0x0000000000401214 <+58>:    mov    eax,0x0
0x0000000000401219 <+63>:    call   0x4010a0 <_isoc99_scanf@plt>
0x000000000040121e <+68>:    mov    eax,DWORD PTR [rbp-0x10]
0x0000000000401221 <+71>:    mov    edi, eax
0x0000000000401223 <+73>:    call   0x401196 <check_key>
0x0000000000401228 <+78>:    mov    DWORD PTR [rbp-0xc],eax
```

Jaisa ki hum jante hai, **_scanf** do argument leta hai. phla argument **rdi** aur dusra **rsi**. phla argument batayega ki **kis type** ka argument lena hai int, float, char etc. dusra argument ye batayega ki kahan pr store krna hai.

```
(gdb) x/s 0x402010
0x402010: "%d"
(gdb) █
```

Yha pr value **%d** type ka le rha hai.

```
0x00000000004011fc <+34>:    mov    eax,0x0
0x0000000000401201 <+39>:    call   0x401090 <printf@plt>
0x0000000000401206 <+44>:    lea    rax,[rbp-0x10]
0x000000000040120a <+48>:    mov    rsi,rax
0x000000000040120d <+51>:    lea    rdi,[rip+0xd9c]      # 0x402010
0x0000000000401214 <+58>:    mov    eax,0x0
0x0000000000401219 <+63>:    call   0x4010a0 <_isoc99_scanf@plt>
0x000000000040121e <+68>:    mov    eax,DWORD PTR [rbp-0x10]
0x0000000000401221 <+71>:    mov    edi, eax
```

Aur hum **rsi** ki value dekhte hai. to usme **rax** ko mov kiya ja rha hai. **rax** me **[rbp-0x10]** ki value plus, minus krke dala ja rha hai.

Jaisa ki hum jante hai ki koi bhi **syscall** koi bhi value **rax** ke andar return krta hai.

```

End of assembler dump.
(gdb) x/s 0x402004
0x402004:      "Enter Key: "
(gdb) x/s 0x402010
0x402010:      "%d"
(gdb) x/s 0x40202f
0x40202f:      "Wrong Key!"
(gdb) x/s 0x402013
0x402013:      "Correct Key! Now Keygen Me."
(gdb)

```

Ab hum **check_key** function ko disassemble krte hai. aur dekhte hai ki ye kya kr rha hai.

```

(gdb) disassemble check_key
Dump of assembler code for function check_key:
0x0000000000401196 <+0>:    endbr64
0x000000000040119a <+4>:    push   rbp
0x000000000040119b <+5>:    mov    rbp,rs
0x000000000040119e <+8>:    mov    DWORD PTR [rbp-0x4],edi
0x00000000004011a1 <+11>:   mov    eax,DWORD PTR [rbp-0x4]
0x00000000004011a4 <+14>:   sub    eax,0x64
0x00000000004011a7 <+17>:   lea    edx,[rax+rax*1]
0x00000000004011aa <+20>:   movsxd rax,edx
0x00000000004011ad <+23>:   imul  rax,rax,0x51eb851f
0x00000000004011b4 <+30>:   shr    rax,0x20
0x00000000004011b8 <+34>:   mov    ecx,eax
0x00000000004011ba <+36>:   sar    ecx,0x5
0x00000000004011bd <+39>:   mov    eax,edx
0x00000000004011bf <+41>:   sar    eax,0x1f
0x00000000004011c2 <+44>:   sub    ecx,eax
0x00000000004011c4 <+46>:   mov    eax,ecx
0x00000000004011c6 <+48>:   imul  eax,eax,0x64
0x00000000004011c9 <+51>:   sub    edx,eax
0x00000000004011cb <+53>:   mov    eax,edx
0x00000000004011cd <+55>:   test   eax, eax
0x00000000004011cf <+57>:   jne   0x4011d8 <check_key+66>
0x00000000004011d1 <+59>:   mov    eax,0x1
0x00000000004011d6 <+64>:   jmp   0x4011d8 <check_key+66>
0x00000000004011d8 <+66>:   pop    rbp
0x00000000004011d9 <+67>:   ret
End of assembler dump.
(gdb) █

```

Yha pr hum assembly ko samjhne ki koshish krte hai.

```

0x000000000040119b <+5>:    mov    rbp,rs
0x000000000040119e <+8>:    mov    DWORD PTR [rbp-0x4],edi
0x00000000004011a1 <+11>:   mov    eax,DWORD PTR [rbp-0x4]
0x00000000004011a4 <+14>:   sub    eax,0x64
0x00000000004011a7 <+17>:   lea    edx,[rax+rax*1]
0x00000000004011aa <+20>:   movsxd rax,edx
0x00000000004011ad <+23>:   imul  rax,rax,0x51eb851f
0x00000000004011b4 <+30>:   shr    rax,0x20

```

Yha pr **eax** me **0x64** value mov kr rha hai.

To hum yha pr “**p**” or “**print**” command ka use karenge pta krenge ki **0x64** ki value kya hai.

```
End of assembler dump.  
(gdb) p 0x64  
$1 = 100  
(gdb) █
```

But hume yha samjhne me difficulty hoti hai to aise me kam aata hai **decompiler**.

Jaisa ki hum jante hai **compiler** humare high level programming language ke code ko machine level code (low level code) me convert krta hai.

Whi **decompiler** iske biprit kam krta hai. ye low level code ko high level code me convert krta hai.

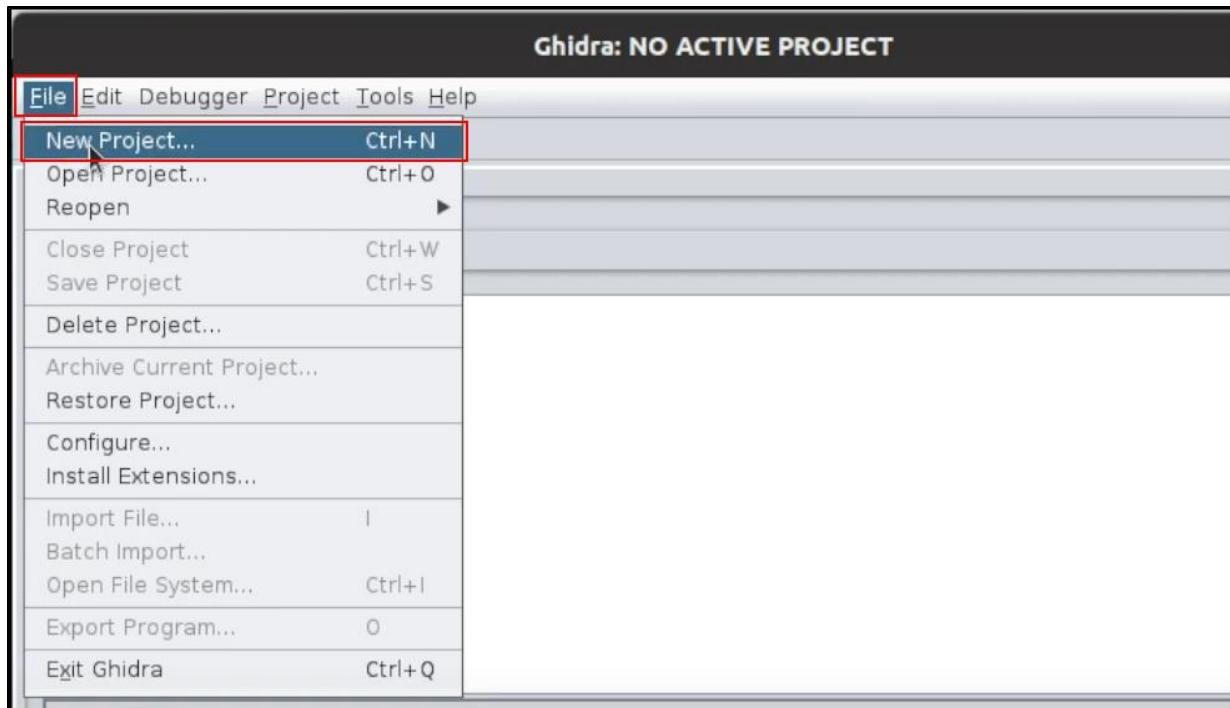
Hum ise jo bhi **elf (binary file)** denge use ye high level me convert krne ka power rkhta hai.

Yha hum **ghidra** decompiler ka use karenge.

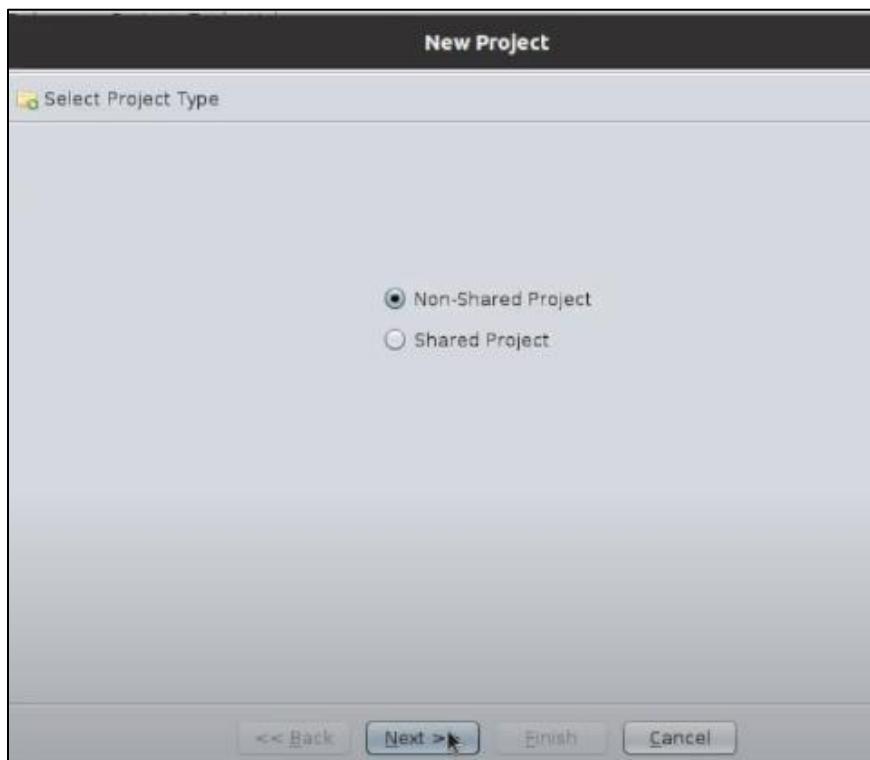


Jaisa ki hum ghidra ko first time open krte hai to ye kuchh is tarah se dikhta hai.

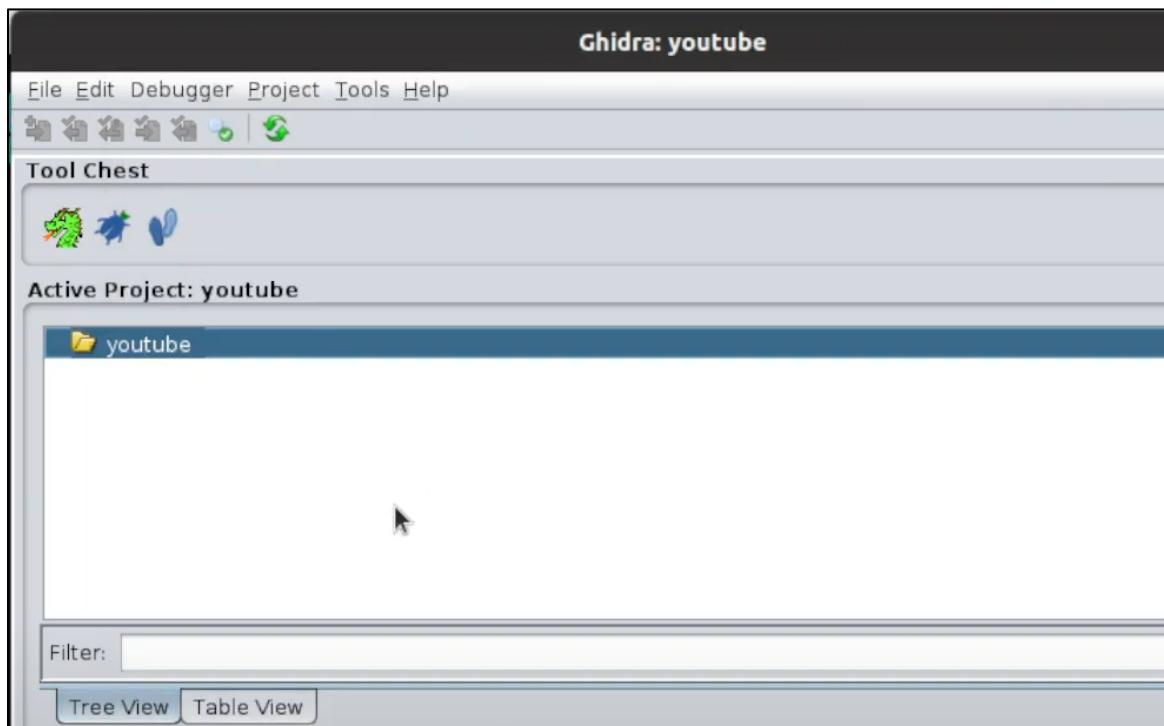
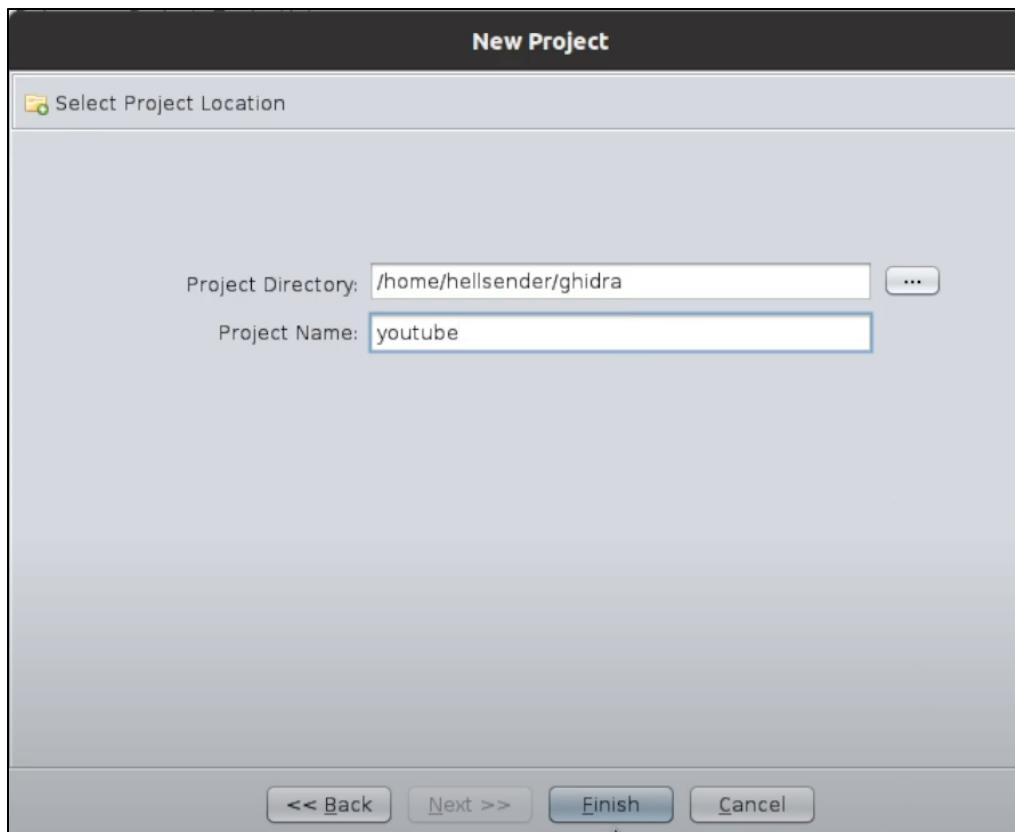
To sbse phle hume **project** create krna pdta hai, uske andar hum apne binary ko open krte hai.



Go to file > New project.

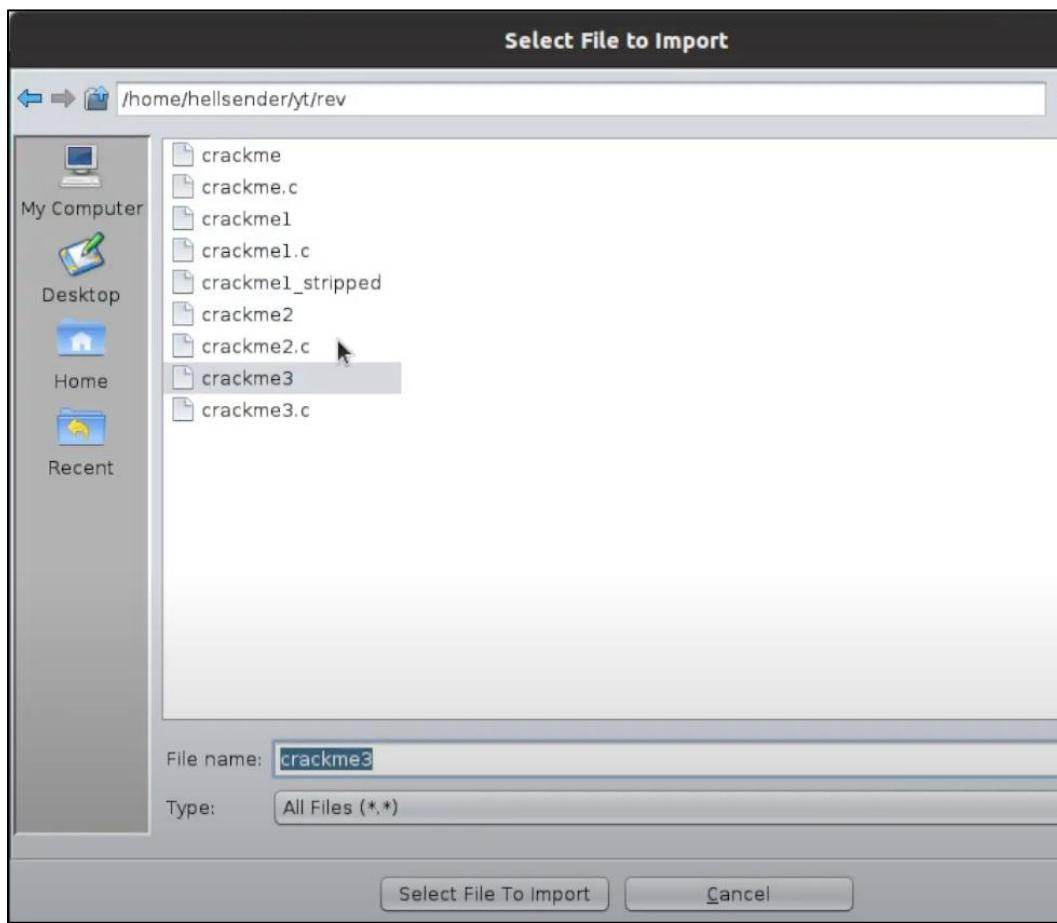
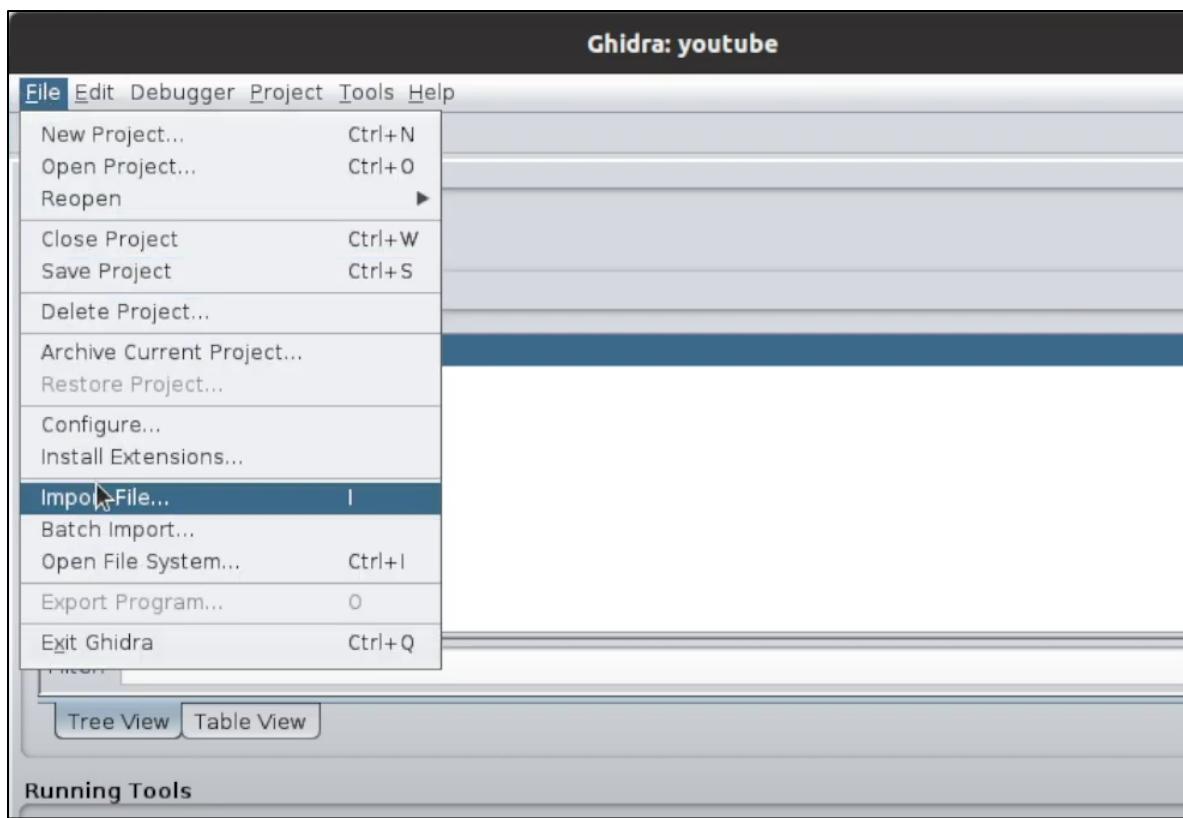


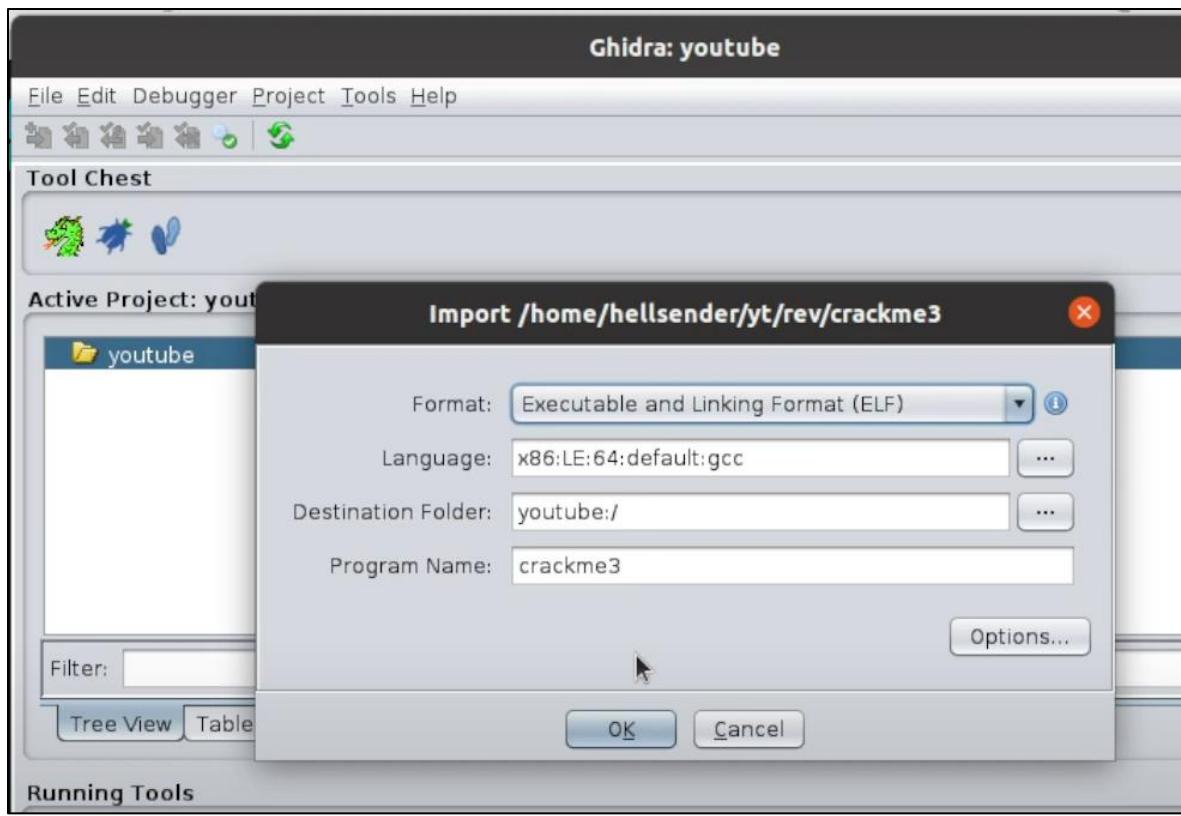
Click on **Non-shared** project click on Next.



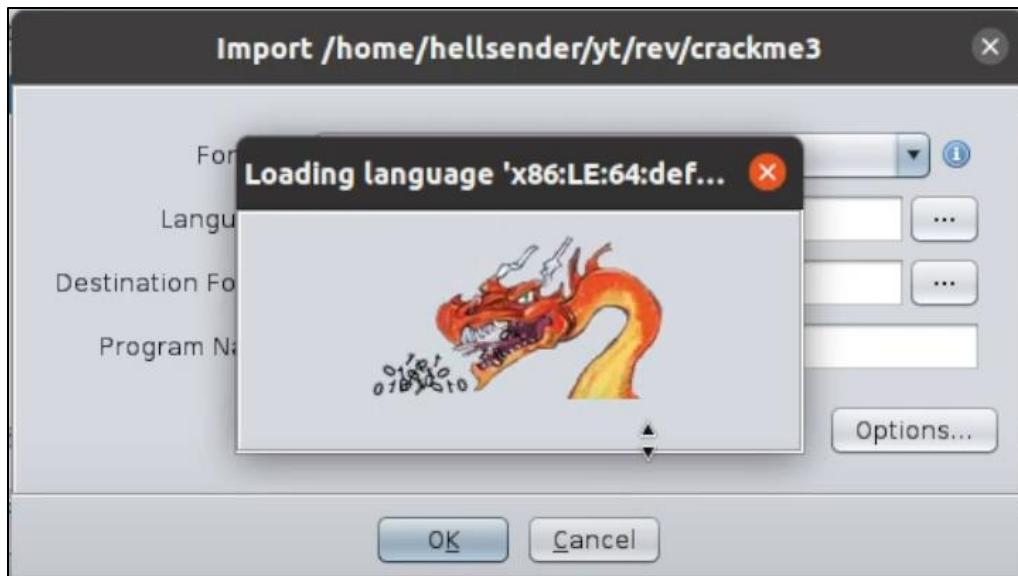
Yha hum dekh skte hai. ki project create ho chuka hai.

Ab hum isme apna binary file dalenge.



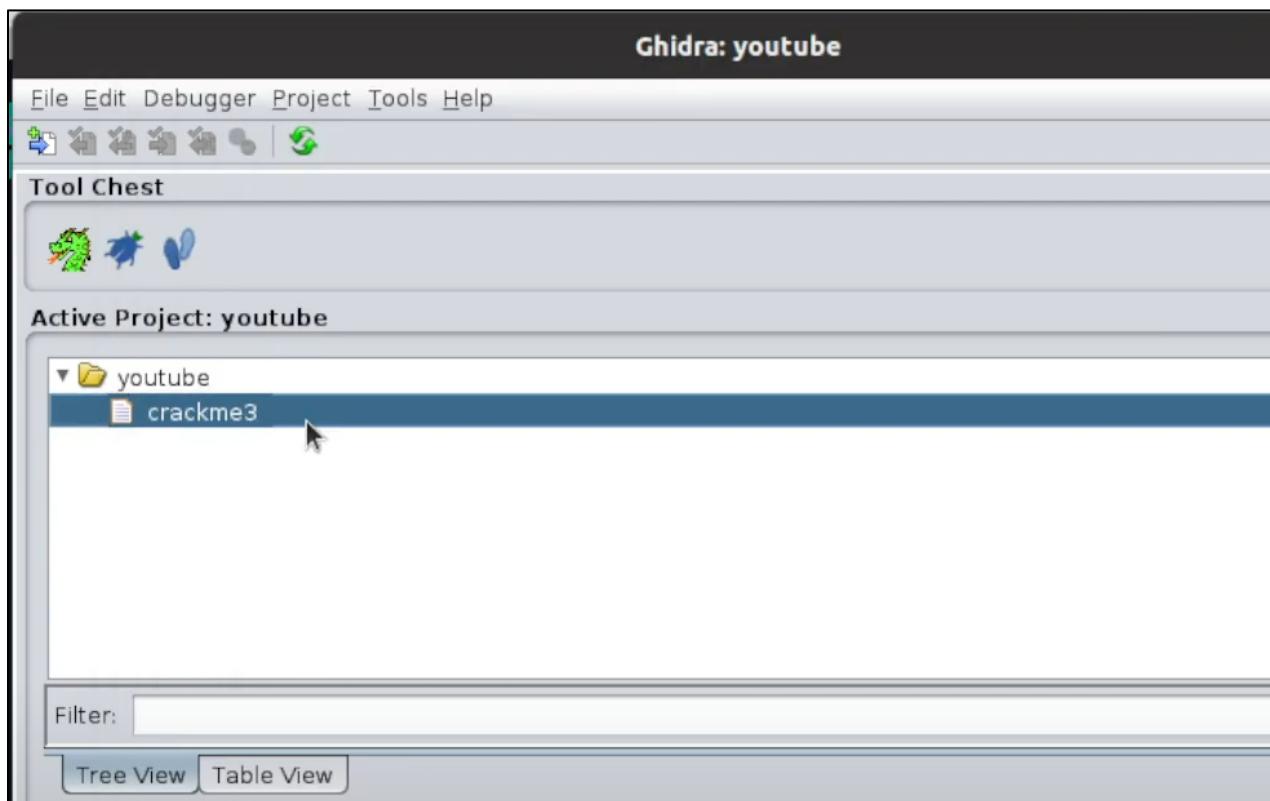


Yha pr ye kuchh information show karega. Jaise hi ok pr click karenge ye analysis karega aur binary ko open kr dega.



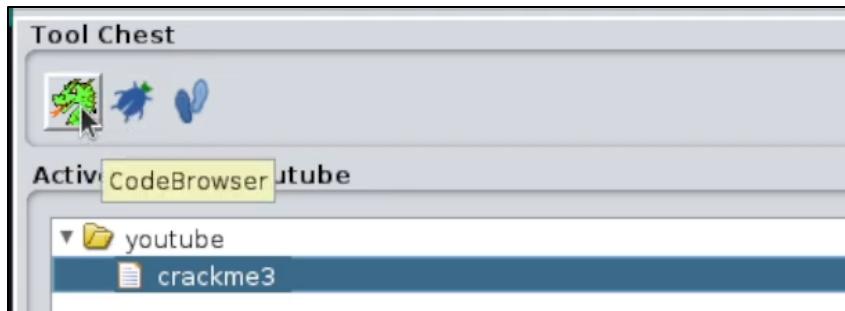
Yha pr ye some info show karega ok pr click krna hai.

 Project File Name:	
Last Modified:	Wed Aug 04 01:00:10 IST 2021
Readonly:	
Program Name:	crackme3
Language ID:	x86:LE:64:default (2.10)
Compiler ID:	gcc
Processor:	x86
Endian:	Little
Address Size:	64
Minimum Address:	00400000
Maximum Address:	_elfSectionHeaders::000007bf
# of Bytes:	7718
# of Memory Blocks:	33
# of Instructions:	16
# of Defined Data:	106
# of Functions:	22
# of Symbols:	54
# of Data Types:	30
# of Data Type Categories:	2
Created With Ghidra Version:	10.0
Date Created:	Wed Aug 04 01:00:08 IST 2021
ELF File Type:	executable
ELF Original Image Base:	0x400000
ELF Prelinked:	false
ELF Required Library [0]:	libc.so.6
ELF Source File [0]:	crtstuff.c



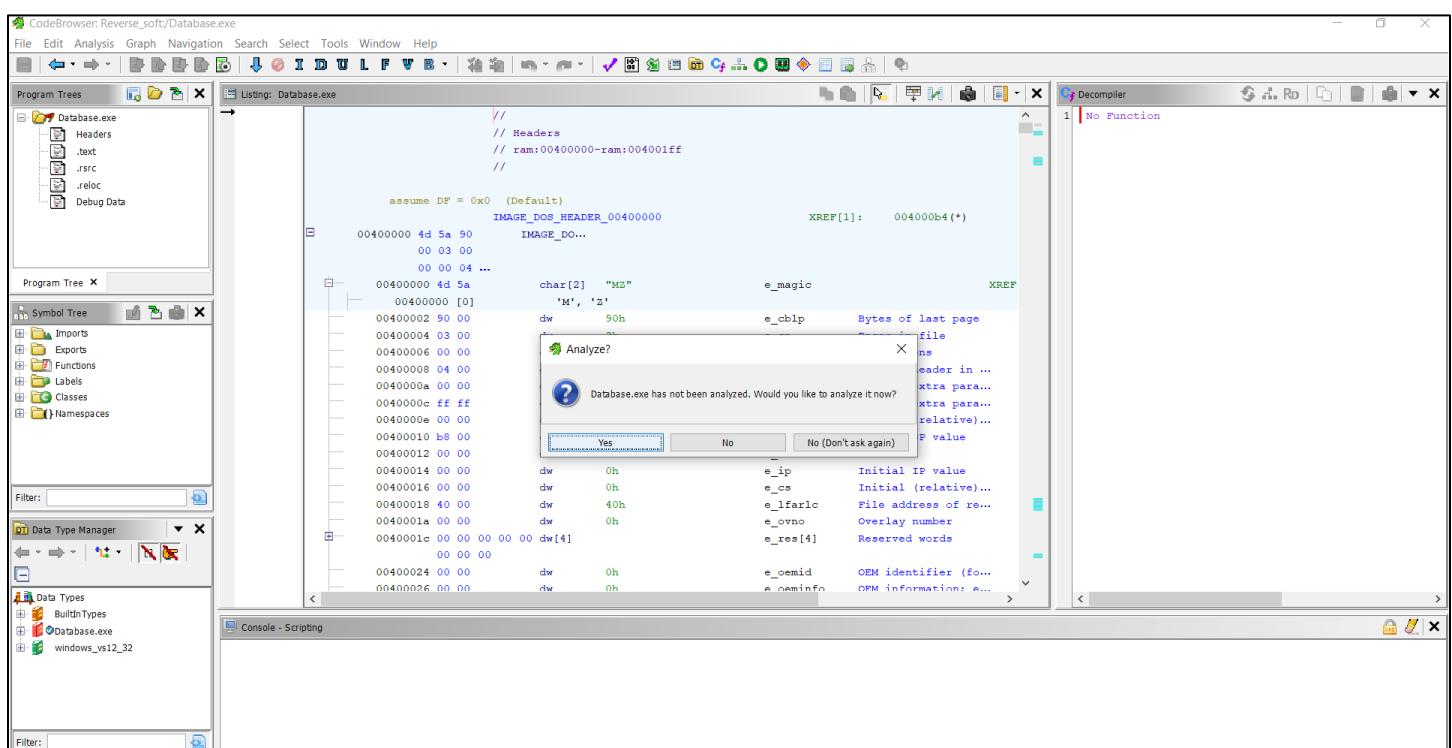
Yha pr humari binary file aa gyi.

Yha pr jitni chahe utni file import kr skte hai.



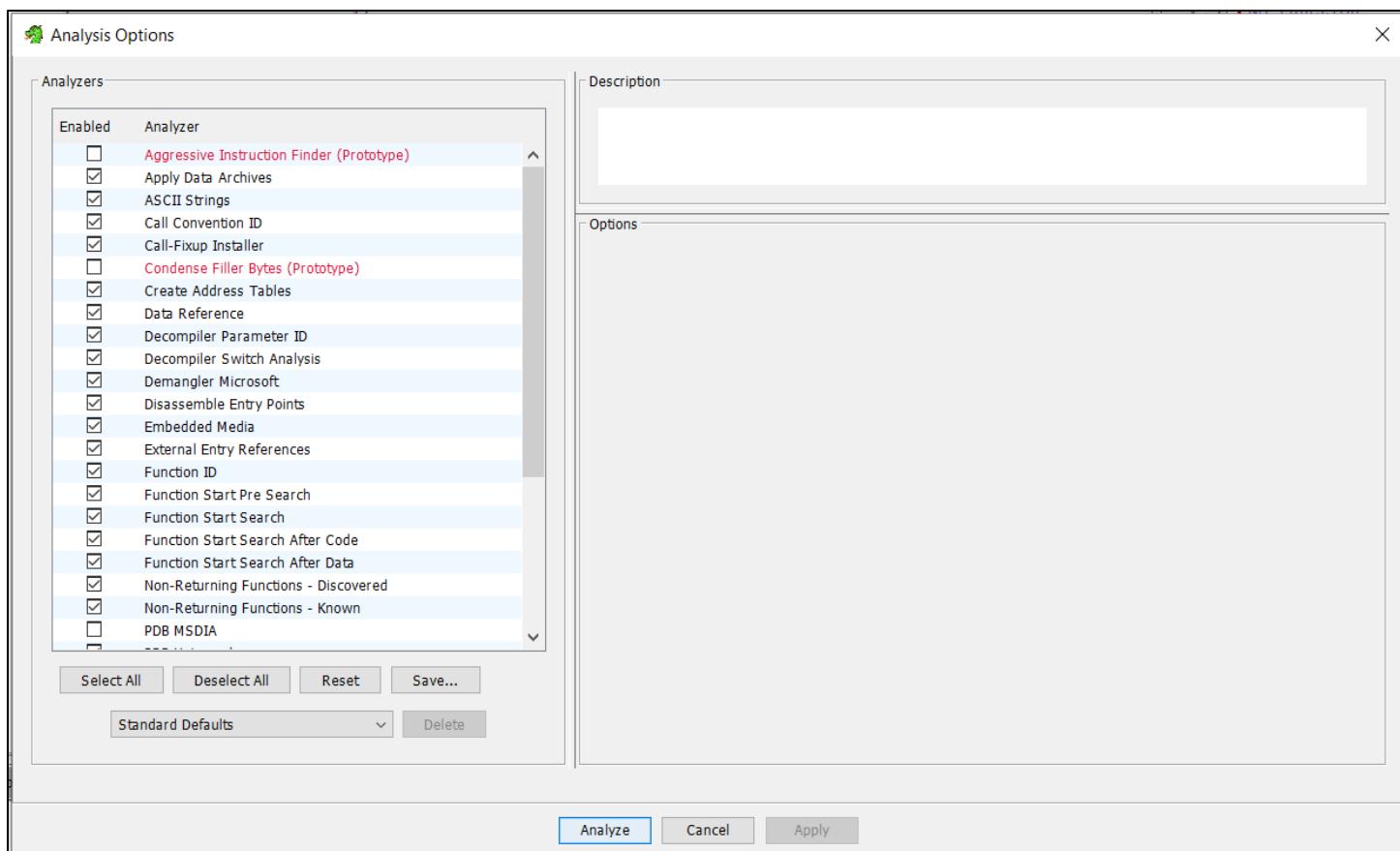
Yha pr do option hai. Ek **codebrowser** jo code ko **decompile** aur **disassemble** karega. Aur dusra **debugger** ka option hai.

Ab hum **crackme3** pr double click karenge to ye codebrowser me open ho jayega.

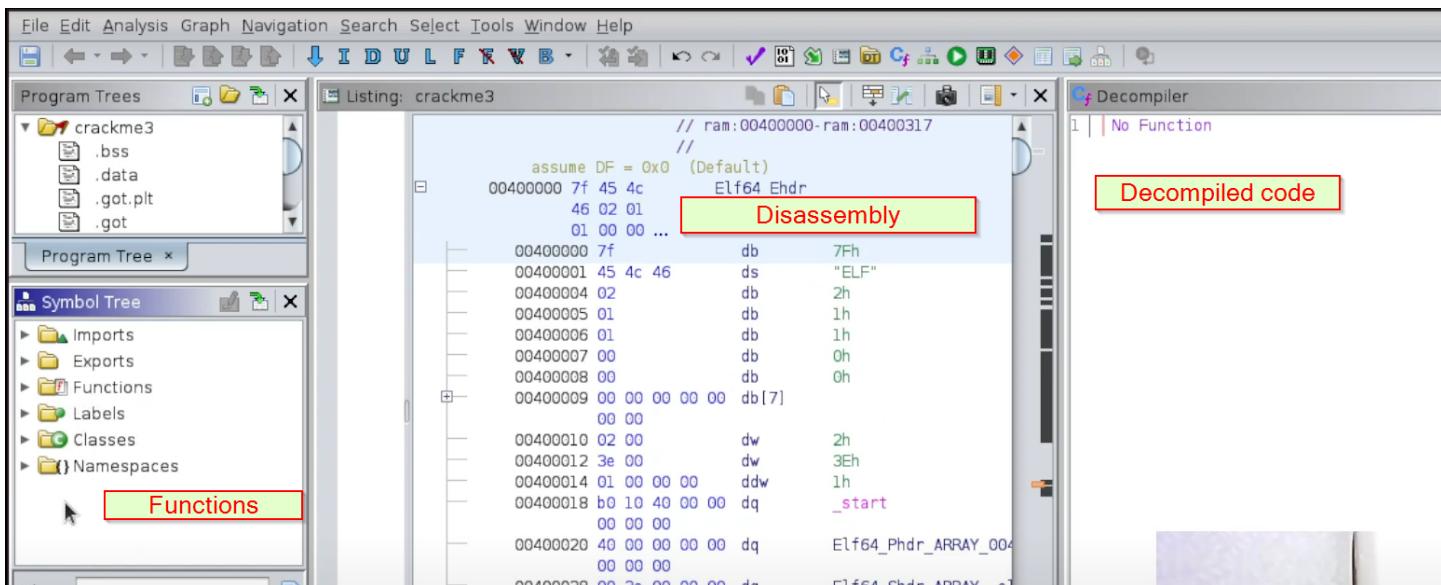


Yha pr ise yes kr dena hai.

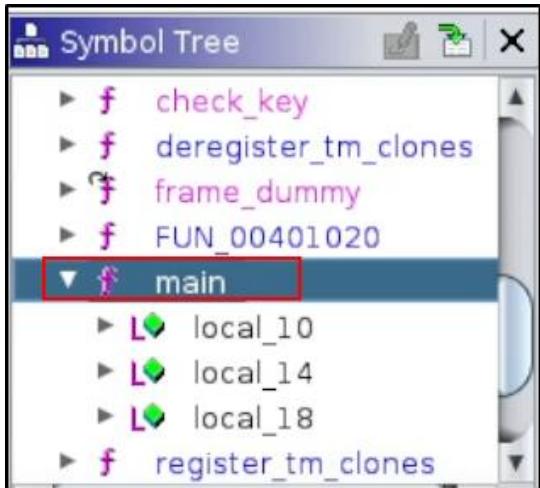
Fir ek aur popup show karega. Analyze pr click kr dena hai.



Fir yha analyze pr click kr dega.



Yha sbse phle hume **main function** find krna hota hai.



Yha pr hum main pr double click karenge.

```
Listing: crackme3
00401223 e8 be 11    CALL    check_key
00401228 89 45 f4    MOV     dword ptr [RBP + local_14], EAX
0040122b 83 7d f4 01  CMP     dword ptr [RBP + local_14], 0x1
0040122f 75 0e        JNZ    LAB_0040123f
00401231 48 8d 3d    LEA     RDI, [s_Correct_Key!_Now_Keygen_Me_.00402013]
db 0d 00 00
00401238 e8 33 fe    CALL    <EXTERNAL>::puts
ff ff
0040123d eb 0c        JMP     LAB_0040124b
LAB_0040123f
0040123f 48 8d 3d    LEA     RDI, [s_Wrong_Key!_0040202f]
e9 0d 00 00
00401246 e8 25 fe    CALL    <EXTERNAL>::puts
ff ff
LAB_0040124b
0040124b b8 00 00    MOV     EAX, 0x0
00 00

Decompile: main ~...
1 undefined8 main(void)
2 {
3     long in_FS_OFFSET;
4     undefined4 local_18;
5     int local_14;
6     long local_10;
7
8     local_10 = *(long *)(in_FS_OFFSET + 0x2);
9     printf("Enter Key: ");
10    _isoc99_scanf(&DAT_00402010,&local_18);
11    local_14 = check_key(local_18);
12    if (local_14 == 1) {
13        puts("Correct Key! Now Keygen Me.");
14    }
15    else {
16        puts("Wrong Key!");
17    }
18    if (local_10 != *(long *)in_FS_OFFSET)
```

Yha pr ye disassembly aur decompile code dono code load ho jayega.

```

1 undefined8 main(void)
2
3 {
4     long in_FS_OFFSET;
5     undefined4 local_18;
6     int local_14;
7     long local_10;
8
9
10    local_10 = *(long *)(in_FS_OFFSET + 0x28);
11    printf("Enter Key: ");
12    __isoc99_scanf(&DAT_00402010,&local_18);
13    local_14 = check_key(local_18);
14    if (local_14 == 1) {
15        puts("Correct Key! Now Keygen Me.");
16    }
17    else {
18        puts("Wrong Key!");
19    }
20    if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
21        /* WARNING: Subroutine does not return */
22        __stack_chk_fail();
23    }
24    return 0;
25 }
26

```

Ye underlined things humare kam ki nhi hai. ye canaries hai.

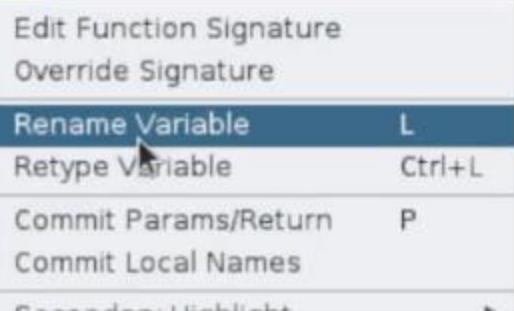
Baki ka code humare liye aasan hai as comparatively **assembly code**.

Yha pr variables ke name humare hisab se nhi milta **decompiler** ko samjh me aata hai usi hisab se rkh data hai.

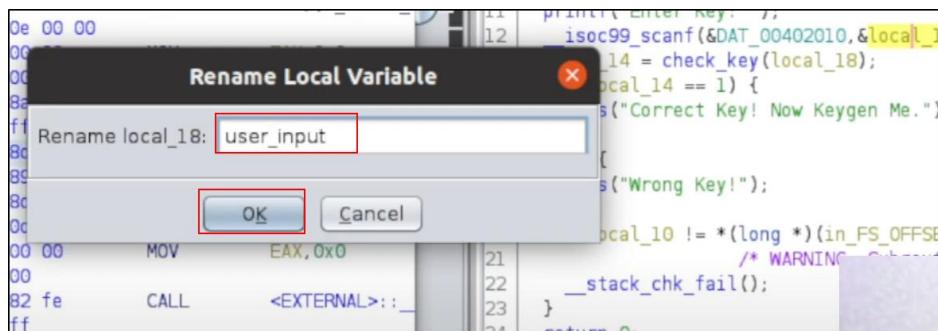
```

local_10 = *(long *)(in_FS_OFFSET + 0x28);
printf("Enter Key: ");
_isoc99_scanf(&DAT_00402010,&local_18);
local_14 = check_key(local_18);
if (local_14 == 1) {
    puts("Correct Key! Now Keygen Me.");
}
else {
    puts("Wrong Key!");
}
if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
    /* WARNING: Subroutine entry
     *          or exit found in stack frame */
    _stack_chk_fail();
}

```



Yha pr hum variables ka nam apne hisab se rename kr lete hai.



```

*(long *)(in_FS_OFFSET + 0x28);
ter Key: ");
canf(&DAT_00402010,&user|input);
check_key(user_input);
l4 == 1) {

```

```

local_10 = *(long *)(in_FS_OFFSET + 0x28);
printf("Enter Key: ");
_isoc99_scanf(&DAT_00402010,&user_input);
local_14 = check_key(user|input);
if (local_14 == 1) {
    puts("Correct Key! Now Keygen Me.");
}
else {
    puts("Wrong Key!");
}

```

Yha pr **check_key** function **user_input** (humare input ke value) ke sath call karega.

Aur ye jo bhi value return karega use **local_14** store kr dega.

`local_14` ko rename kr denge `return_value` se

```

1 undefined8 main(void)
2 {
3
4     long in_FS_OFFSET;
5     undefined4 user_input;
6     int return_value;
7     long local_10;
8
9
10    local_10 = *(long *)(in_FS_OFFSET + 0x28);
11    printf("Enter Key: ");
12    isoc99_scanf(&DAT_00402010,&user_input);
13    return_value = check_key(user_input);
14    if (return_value == 1) {
15        puts("Correct Key! Now Keygen Me.");
16    }
17    else {
18        puts("Wrong Key!");
19    }
20    if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
21        /* WARNING: Subroutine does not return */
22        __stack_chk_fail();
23    }
24    return 0;
25 }
26

```

aur hum **check_key()** function me jate hai aur dekhte hai ki usme kya logic hai.

```

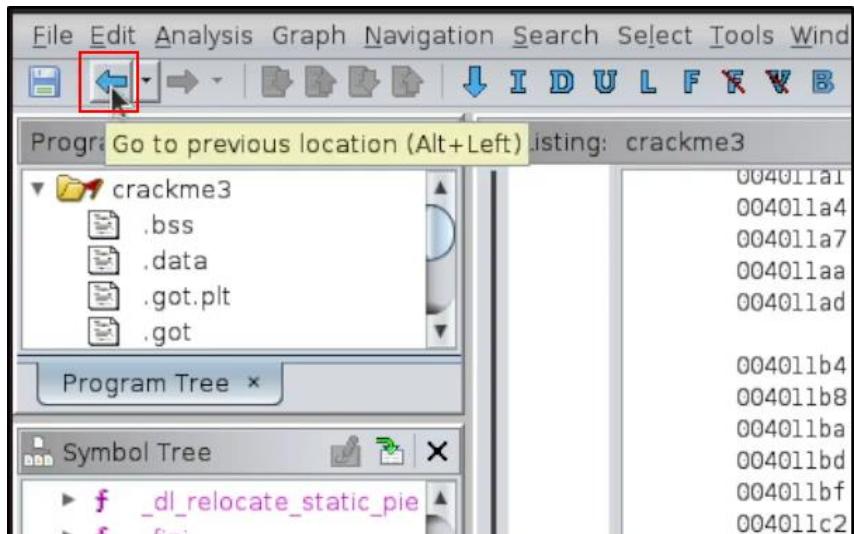
1
2 int check_key(int param_1)
3
4 {
5     int iVar1;
6
7     iVar1 = ((param_1 + -100) * 2) % 100;
8     if (iVar1 == 0) {
9         iVar1 = 1;
10    }
11    return iVar1;
12 }

```

Iske variables ko hum rename kr lete hai.

```

1 int check_key(int user_input)
2 {
3     int result;
4
5     result = ((user_input + -100) * 2) % 100;
6     if (result == 0)
7         result = 1;
8     }
9     return result;
10 }
11 }
```



Ab is arrow pr click karenge to wapas **main** function pr aa jayenge.

```

printf("Enter Key: ");
_isoc99_scanf(&DAT_00402010,&user_input);
return_value = check_key(user_input);
if (return_value == 1)
    puts("Correct Key! Now Keygen Me.");
}
else {
    puts("Wrong Key!");
}
```

Yha hum dekh skte hai ki **return_value** ki value agar **1** hogi tabhi correct key print karega.

Return 1 kb karega jb

```

1 int check_key(int user_input)
2 {
3     int result;
4
5     result = ((user_input + -100) * 2) % 100;
6     if (result == 0)
7         result = 1;
8     }
9     return result;
10 }
11 }
```

Jb is pure calculation ki value **0** hogi to ye return **1** karega.

To hume apni value kuchh aisa dalna padega ki calculation **0** aaye.

To yha **user_input** ki value 200 man lete hai. agar calculate kare to reminder 0 aa jayega.

```

→ rev ./crackme3
Enter Key: 200
Correct Key! Now Keygen Me.
→ rev
```

To 200 correct key hai.

yha humne 1 possible key find kr liya.

Ab aur bhi possible key find krte hai.

Ab hum python ki script ki help se **1000** tk key generate krte hai.

```

→ rev python3
Python 3.8.10 (default, Jun  2 2021, 10:49:15)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license"
>>> for i in range(1000):
...     result = ((i-100)*2)%100
...     if result == 0:
...         print(i)
```

Output

```
>>> for i in range(1000):
...     result = ((i-100)*2)%100
...     if result == 0:
...         print(i)
...
0
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
800
850
900
950
>>> █
```

These are correct key between **0** to **1000**.

```
→ rev ./crackme3
Enter Key: 50
Correct Key! Now Keygen Me.
→ rev ./crackme3
Enter Key: 350
Correct Key! Now Keygen Me.
→ rev █
```

#####

Patching in reverse engineering:

Challenge – crackme4

```
→ rev ./crackme4
Enter Input : 123456
Wrong!!!
```

Yha hum is kam ko krne ke liye **Binary Ninja** ka use karenge.

=====

Agar hum ise strings kare to.

```
→ rev strings crackme4
/lib64/ld-linux-x86-64.so.2
libc.so.6
__isoc99_scanf
puts
__stack_chk_fail
printf
__libc_start_main
GLIBC_2.7
GLIBC_2.4
GLIBC_2.2.5
__gmon_start__
H=H@@
[]A\A]A^A_
Enter Input :
You have cracked me! Now go and patch me!
Wrong!!!
:*3$"
GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
crtstuff.c
deregister_tm_clones
__do_global_dtors_aux
completed.8060
```

To hume yha pr sahi password nikalna hai. agar galat password dalenge to ye wrong!!!

Agar sahi password dalenge to

You have cracked me! Now go and patch me!

Print karega.

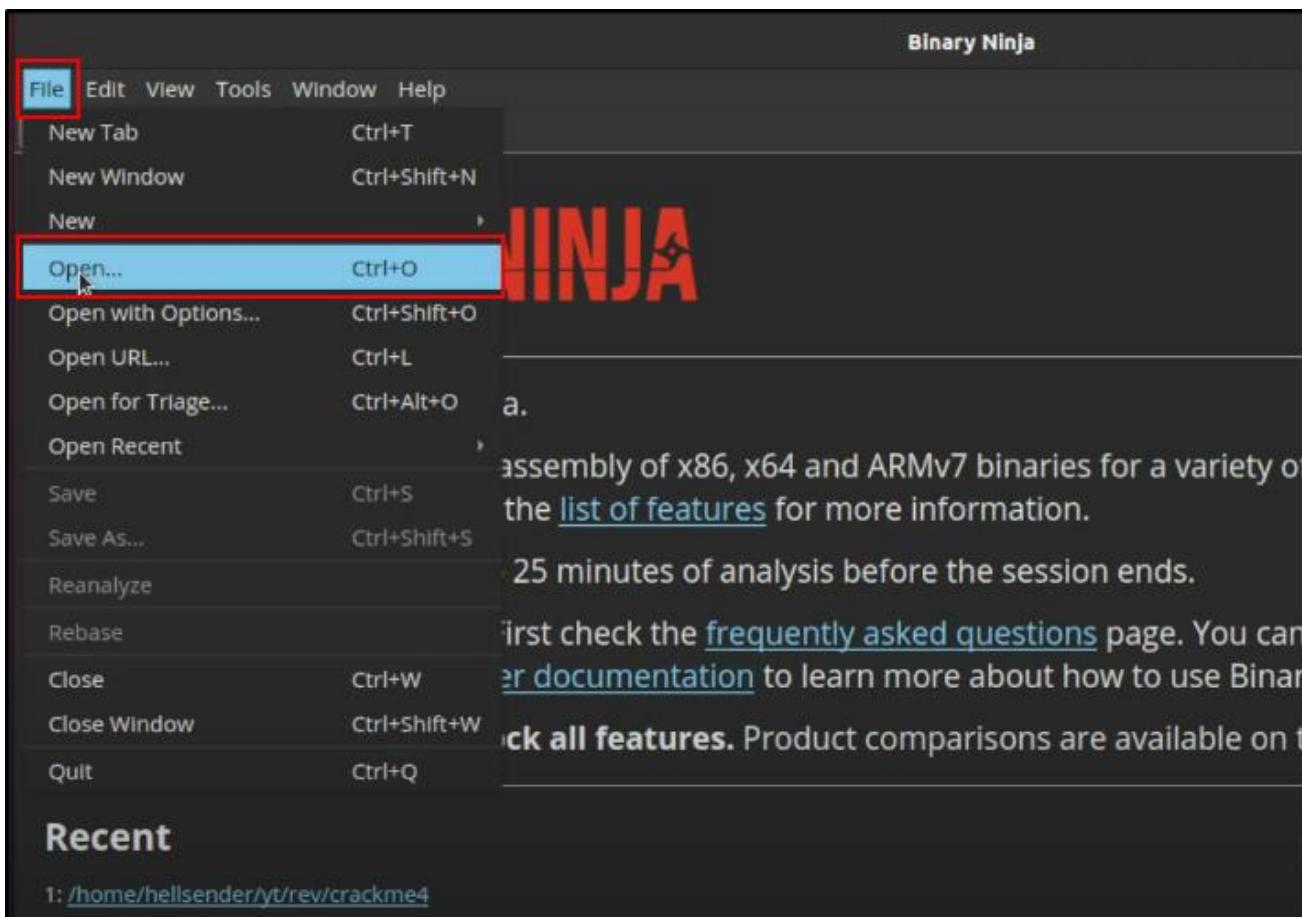
To hum yha binary ninja ka install karenge. Agar hume free use karna hai to cloud pr kr skte hai.

Agar hum offline free version use krna hai to isse bs disassembly aur decompilation pd skte hai aur debugging nhi kr skte hai.

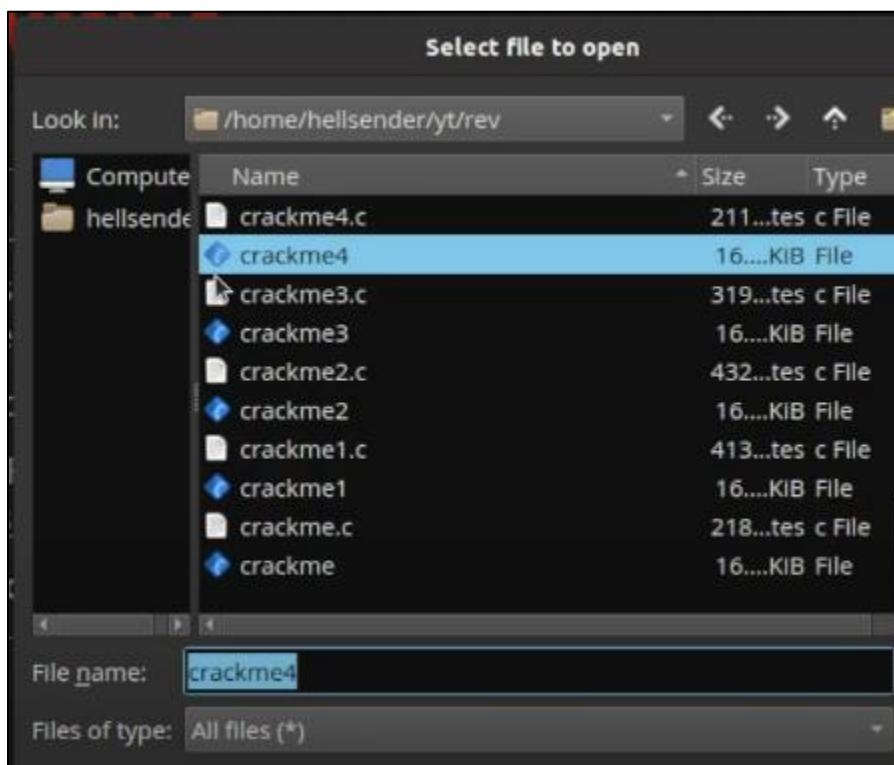
Iska free version sift 30 minute ke liye hi hota hai.

Let's move on binary ninja.

Sbse phle hum file pr jayenge aur open pr click karenge.



Yha hum crackme4 select karenge aur open pr click kr denge.



crackme4 — Binary Ninja

File Edit View Tools Window Help

Symbols

```
_init
sub_401034
puts
__stack_chk_fail
printf
__isoc99_scanf
_start
_dl_relocate_static_pie
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
main
__libc_csu_init
__libc_csu_fini
fini
__libc_start_main@GOT
__gmon_start__@GOT
```

crackme4 (ELF Linear) ×

```
int64_t __start(int64_t arg1, int64_t arg2, void (* arg3)()) __noreturn
    004010c1    int64_t rax
    004010c1    int64_t var_8 = rax
    004010d8    __libc_start_main(main: main, argc: __return_addr.d, argv: &arg_8, init: __libc_csu_init, fini:
    004010d8    noreturn

    004010de                                f4 90      ..
int64_t __dl_relocate_static_pie()
    004010e4    return

    004010e5    66 2e 0f-1f 84 00 00 00 00 00 90      f.....
void deregister_tm_clones()
```

To ye yha pr load ho jayega. Aur hum dekh skte hai main function ko uspr double click karenge. To main function khul jayega.

crackme4 — Binary Ninja

File Edit View Tools Window Help

Symbols

```
_init
sub_401034
puts
__stack_chk_fail
printf
__isoc99_scanf
_start
_dl_relocate_static_pie
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
main
__libc_csu_init
__libc_csu_fini
fini
__libc_start_main@GOT
__gmon_start__@GOT
```

crackme4 (ELF Linear) ×

```
00401181    66 66 2e 0f 1f 84 00-00 00 00 00 0f 1f 40 00      ff.....
int64_t frame_dummy()
00401194    return register_tm_clones() __tailcall

int32_t main(int32_t arg1, char** arg2, char** arg3)
    004011a2    void* fsbase
    004011a2    int64_t rax = *(fsbase + 0x28)
    004011bd    printf(format: "Enter Input : ")
    004011d5    int32_t var_14
    004011d5    __isoc99_scanf(format: data_402017, &var_14)
    004011dd    if (var_14 != 0x499602d2)
    004011f9    |    puts(str: "Wrong!!!")
    004011eb    else
    004011eb    |    puts(str: "You have cracked me! Now go and ...")
    00401203    int64_t rax_5 = rax ^ *(fsbase + 0x28)
    00401214    if (rax_5 == 0)
    00401214    |    return rax_5.d
    0040120e    __stack_chk_fail()
    0040120e    noreturn

00401215    66 2e 0f-1f 84 00 00 00 00 00 90      f...
int32_t __libc_csu_init(int32_t arg1, char** arg2, char** arg3)
```

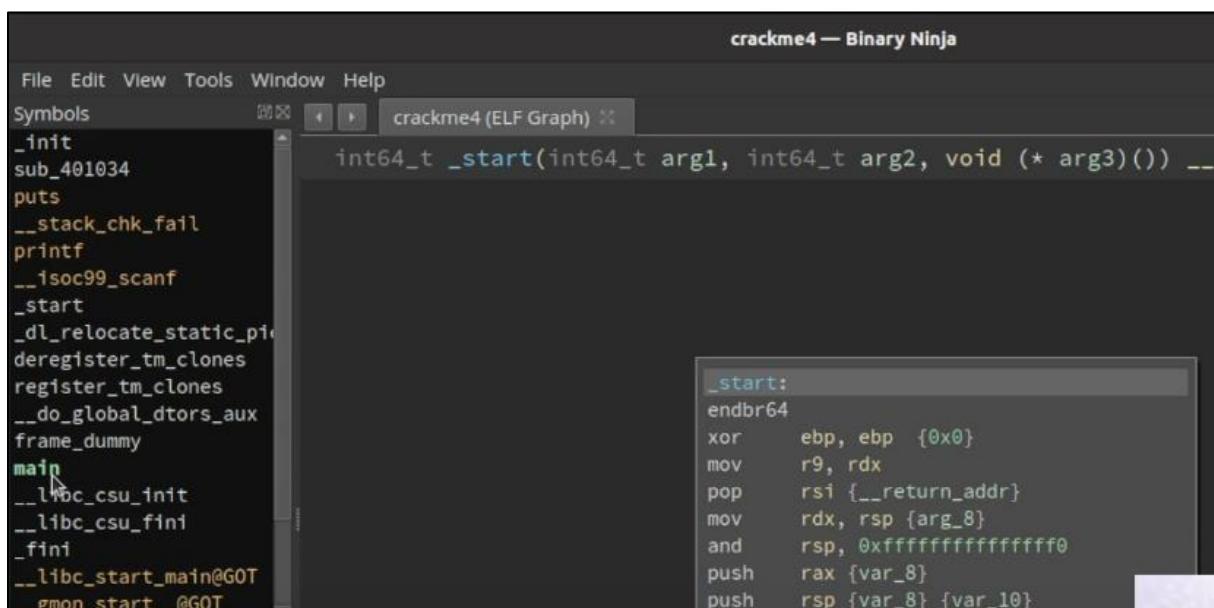
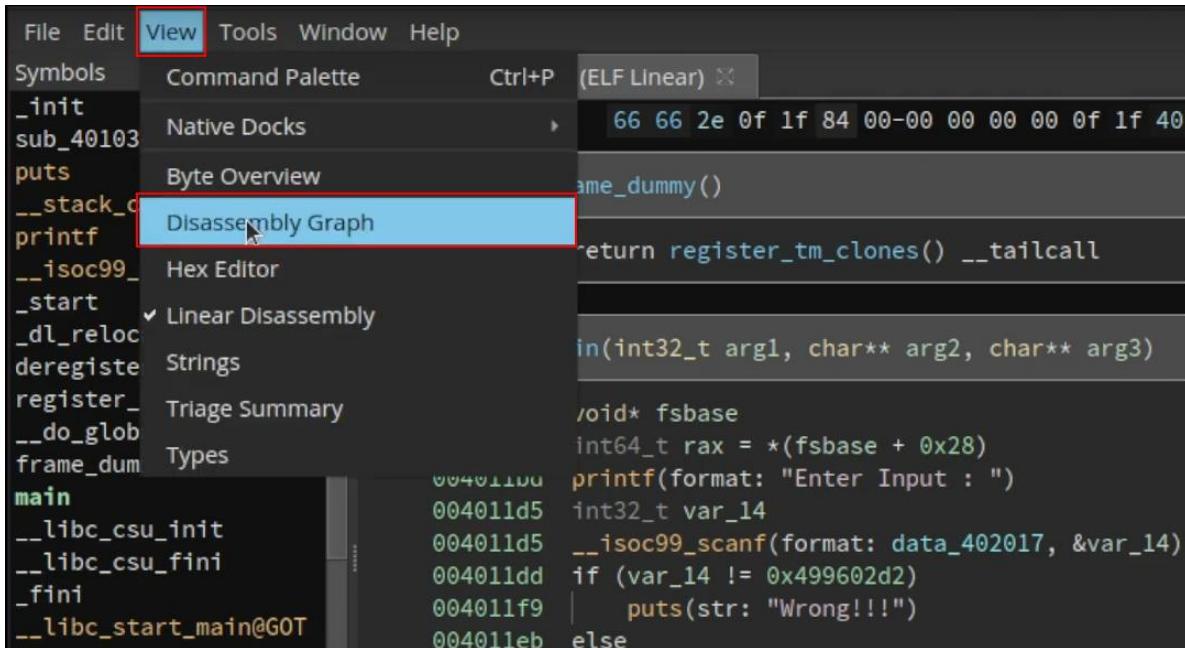
Symbols Tags

Cross References

- + Filter (2)
- Code References {2}
- * _start {2}
 - 004010d1 mov
 - 004010d8 call

Yha main khul chuka hai.

Abhi ye **list view** me show ho rha hai. to view pr jate hai aur **graph view** pr click krte hai.



To ye **graph view** me show krne laga.

Hum fir se **main** pr double click karenge.

crackme4 — Binary Ninja

File Edit View Tools Window Help

Symbols crackme4 (ELF Graph)

```
int32_t main(int32_t arg1, char** arg2, char** arg3)
    lea    rdi, [rel data_402017]
    mov    eax, 0x0
    call   __isoc99_scanf
    mov    eax, dword [rbp-0xc {var_14}]
    cmp    eax, 0x499602d2
    jne    0x4011f2

    rdi, [rel data_40204a] {"Wrong!!!"}
    l     puts

    lea    rdi, [rel data_402020] {"You have cracked me!"}
    call  puts
    jmp    0x4011fe

    nop
    mov    rax, qword [rbp-0x8 {var_10}]
    xor    rax, qword [fs:0x28]
    je    0x401213
```

Yha pr main load ho gaya. Aur kafi achha view show ho rha hai.

Ye yha pr char tarike se code ko pdne ka option deta hai.

crackme4 — Binary Ninja

File Symbols Tags Cross References Filter (2) Code References (2)

char** arg2, char** arg3)

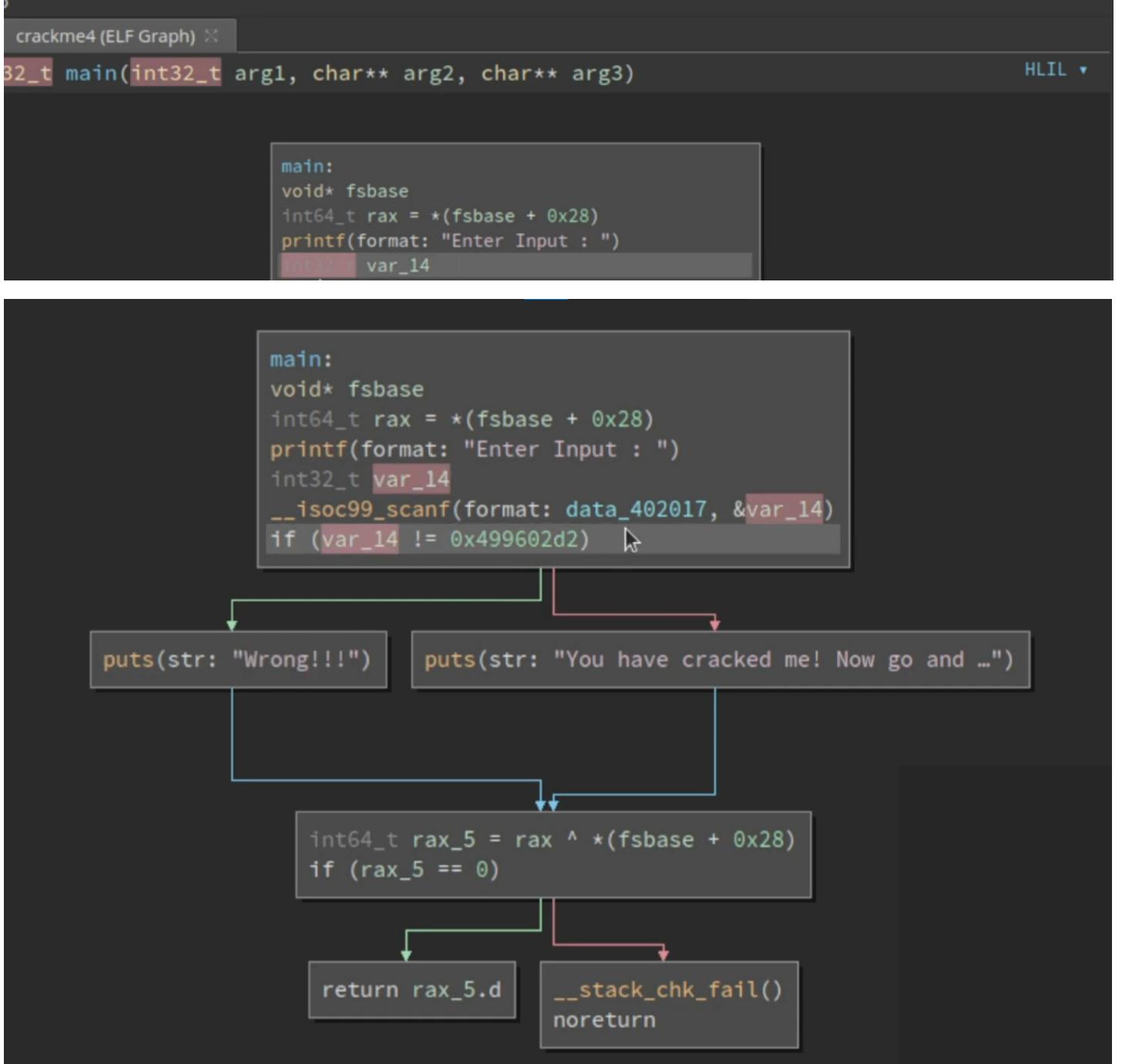
rbp {__saved_rbp}
rbp, rsp {__saved_rbp}
rsp, 0x10
rax, qword [fs:0x28]
qword [rbp-0x8 {var_10}], rax
eax, eax {0x0}
rdi, [rel data_402008] {"Enter Input : "}
eax, 0x0

Disassembly ▾

- ✓ Disassembly
- Low Level IL
- Medium Level IL
- High Level IL

Yha pr **low level** me ye thoda aur human readable bna deta hai. **medium level** thoda aur human readable banata hai. aur **High level** me ye bilkul high level jaisa code banane ki koshish krta hai.

Agar hum high level pr set kare to

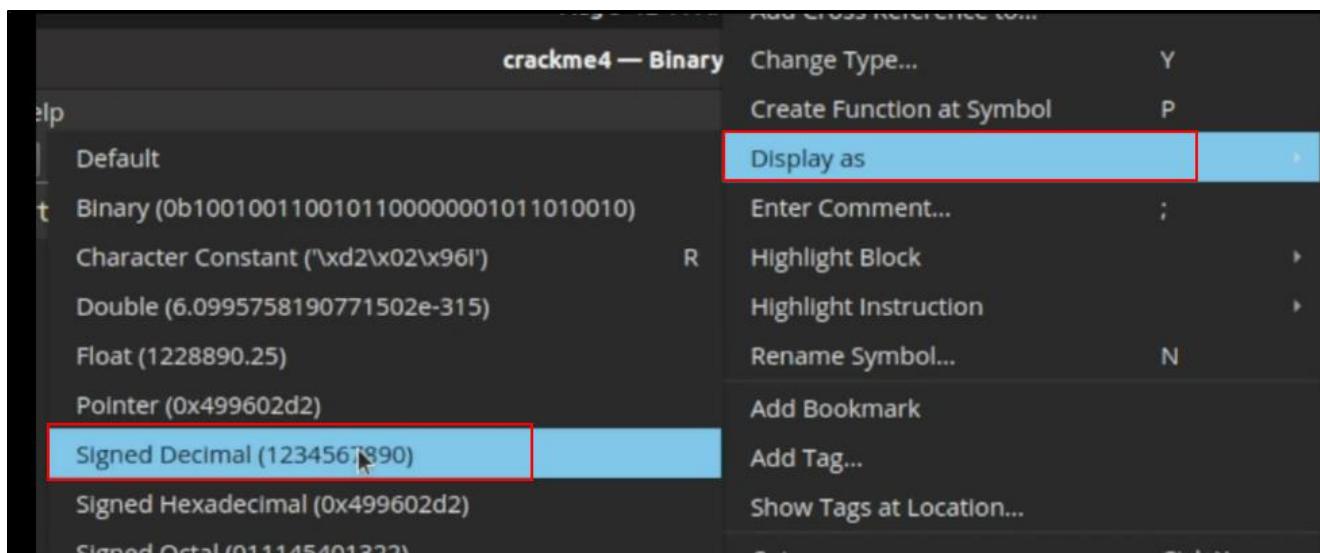


Hum value pr click krke right click karenge.

```
main:  
void* fsbase  
int64_t rax = *(fsbase + 0x28)  
printf(format: "Enter Input : ")  
int32_t var_14  
__isoc99_scanf(format: data_402017, &var_14)  
if (var_14 != 0x499602d2) →  
(str: "Wrong!!!") | puts(str: "You have cracked me! Now go a
```

Aur ise display as **signed Decimal** karenge.

Upar hum dekh skte hai. ki **var_14 int32** type ka hai isliye humne ise integer me convert kiya.



```
main:
void* fsbase
int64_t rax = *(fsbase + 0x28)
printf(format: "Enter Input : ")
int32_t var_14
__isoc99_scanf(format: data_402017, &var_14)
if (var_14 != 1234567890)
    ↓
    tr: "Wrong!!!")    puts(str: "You have cracked me! Now g
```

To yha pr hume patch banana hai to yha **!=** ko hum **==** bna ke patch kr denge ki koi bhi chalaye wo correct password print kare even password galat bhi ho.

Hum wapas se chalenge disassembly code me.

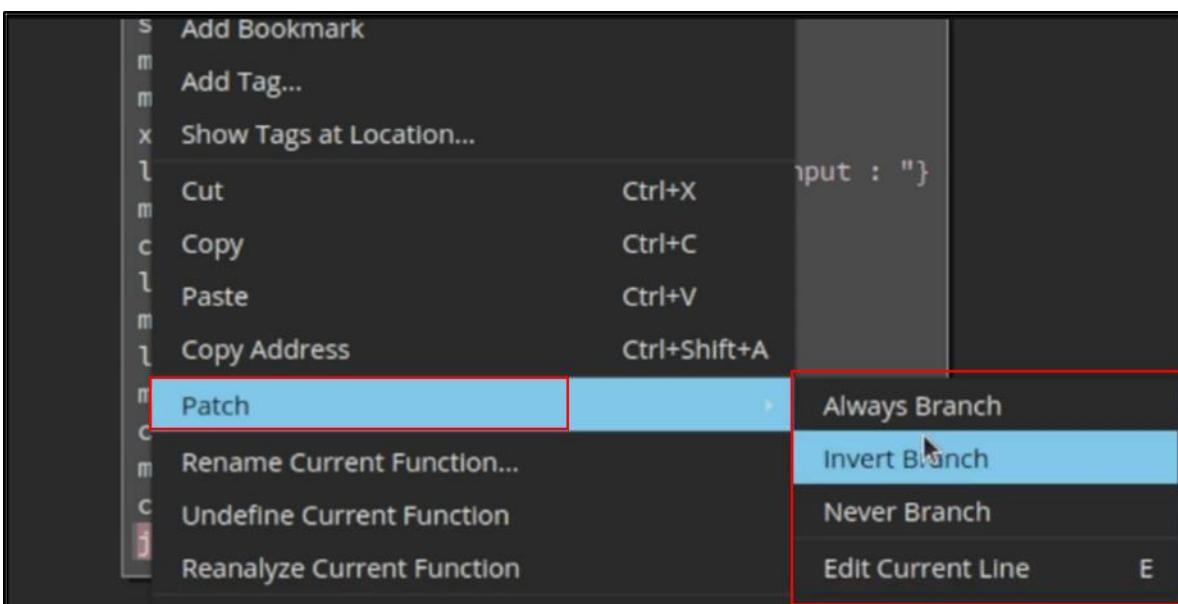
```
32_t main(int32_t arg1, char** arg2, char** arg3)
{
    mov    rbp, rsp {__saved_rbp}
    sub    rsp, 0x10
    mov    rax, qword [fs:0x28]
    mov    qword [rbp-0x8 {var_10}], rax
    xor    eax, eax {0x0}
    lea    rdi, [rel data_402008] {"Enter Input : "}
    mov    eax, 0x0
    call   printf
    lea    rax, [rbp-0xc {var_14}]
```

```
lea    rdi, [rel data_402008] {"Enter Input : "}
mov    eax, 0x0
call   printf
lea    rax, [rbp-0xc {var_14}]
mov    rsi, rax {var_14}
lea    rdi, [rel data_402017]
mov    eax, 0x0
call   __isoc99_scanf
mov    eax, dword [rbp-0xc {var_14}]
cmp    eax, 1234567890
jne    0x4011f2

rdi, [rel data_40204a] {"Wrong!!!"} puts
        lea    rdi, [rel data_402020] {""
call   puts
imp   0x4011fe
```

Jne ko change kr denge **je** me.

Agar hum patch pr right click kare to.



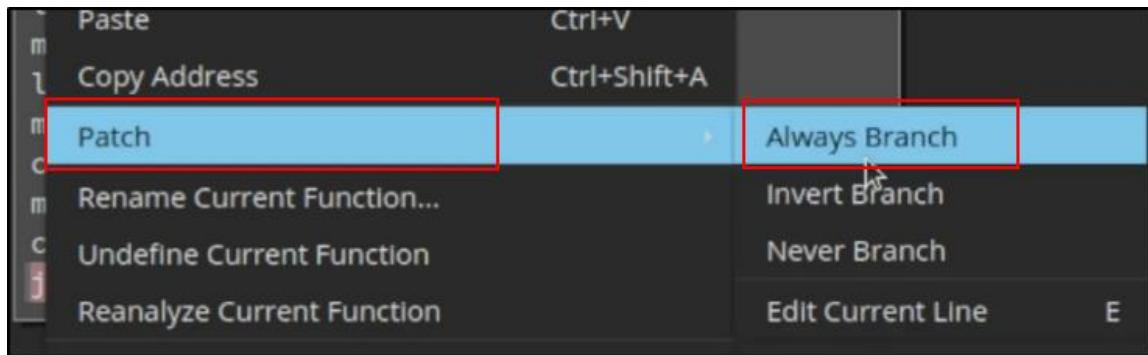
Yha pr humare pass char options hai.

```
mov    eax, dword [rbp-0xc {var_14}]
cmp    eax, 1234567890
jne    0x4011f2

rdi, [rel data_40204a] {"Wrong!!!"} puts
        lea    rdi, [rel data_402020] {""
call   puts
imp   0x4011fe
```

Ye waise “**wrong!!!**” branch jayega kyoki humne galat password diya. hume password nhi pta tha.

Yha hum always branch krke dekhte hai.



Agar hum always branch karenge to ye correct branch remove kr dega aur hamesa “**wrong!!!**” branch pr le jayega. **Jne** ko **jmp** kr dega.

Assembly code:

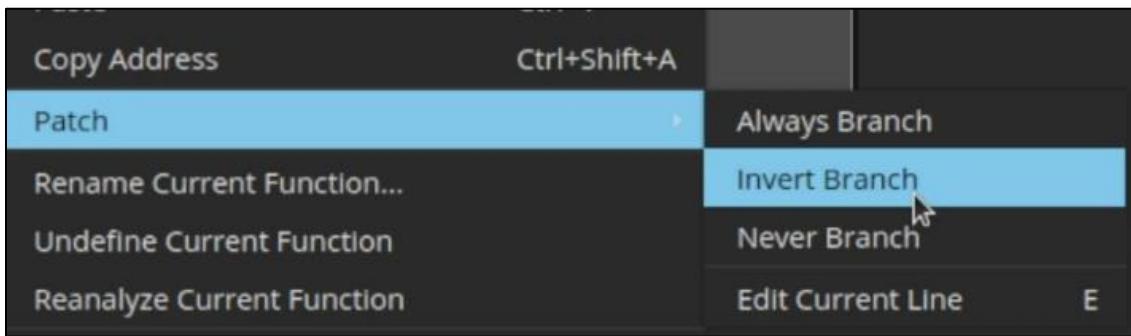
```
mov    eax, 0x0
call   __isoc99_scanf
mov    eax, dword [rbp-0xc {var_14}]
cmp    eax, 1234567890
jmp    0x4011f2
lea    rdi, [rel data_40204a] {"Wrong!!!"}
call   puts
nop
mov    rax, qword [rbp-0x8 {var_10}]
xor    rax, qword [fs:0x28]
je    0x401213
```

The instruction `jmp 0x4011f2` is highlighted with a red box. An arrow points from this jmp instruction to the string `{"Wrong!!!"}`, which is also highlighted with a red box.

To hum **ctrl+z** krke wapas aate hai hume ye nhi krna.

```
mov    eax, 0x0
call   __isoc99_scanf
mov    eax, dword [rbp-0xc {var_14}]
cmp    eax, 1234567890
jne    0x4011f2
data_40204a {"Wrong!!!"}
lea    rdi, [rel data_402020] {"Y
call   puts
jmp    0x4011fe
```

Ab hum invert branch karengे.



Yha humne invert branch kr diya mtlb **1234567890** dale to **wrong** print kr do other **correct password** pr le chalo.

```
mov    eax, 0x0
call   __isoc99_scanf
mov    eax, dword [rbp-0xc {var_14}]
cmp    eax, 1234567890
je    0x4011f2
data_40204a {"Wrong!!!"}
lea    rdi, [rel data_402020] {"Y
call   puts
jmp    0x4011fe
```

Agar hum never kare to..

```

mov    eax, dword [rbp-0xc {var_14}]
cmp    eax, 1234567890
nop
nop
lea    rdi, [rel data_402020] {"You have cracked me! Now go and
call   puts
jmp    0x4011fe

```

↓

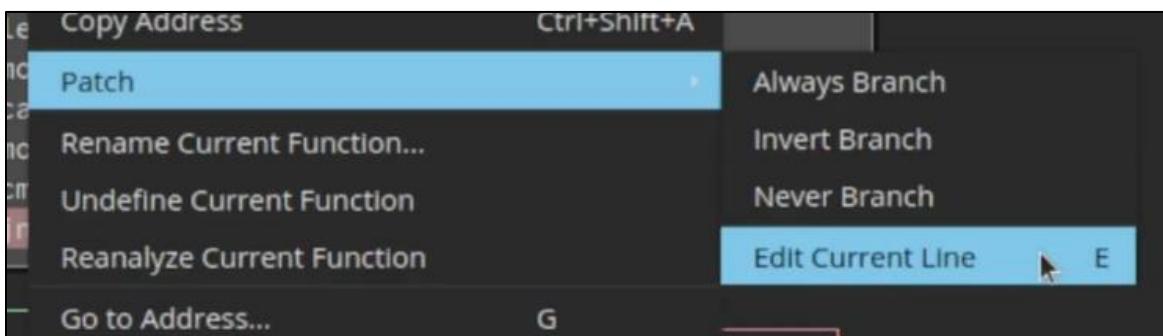
```

nop
mov    rax, qword [rbp-0x8 {var_10}]
xor    rax, qword [fs:0x28]
je     0x401213

```

To ye “**wrong!!**” pr jayega hi nhi use nop kr dega. Aur hamesa correct password pr le jayega.

Aur last option.



Agar aap khud se krna chahte ho to wo bhi kr skte ho edit krke.

```

call   __isoc99_scanf
mov    eax, dword [rbp-0xc {var_14}]
cmp    eax, 1234567890
jne 0x4011f2

```

↓

```

l data_40204a {"Wrong!!!"}

```

```

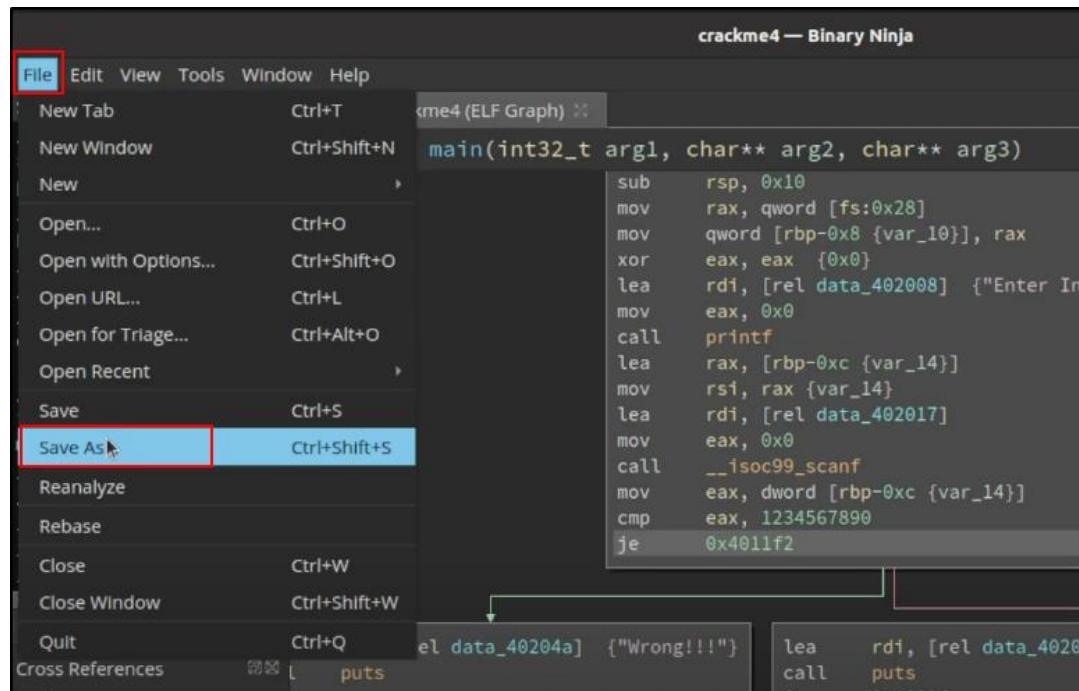
lea    rdi, [rel data_402020] {"You ha
call   puts
jmp    0x4011fe

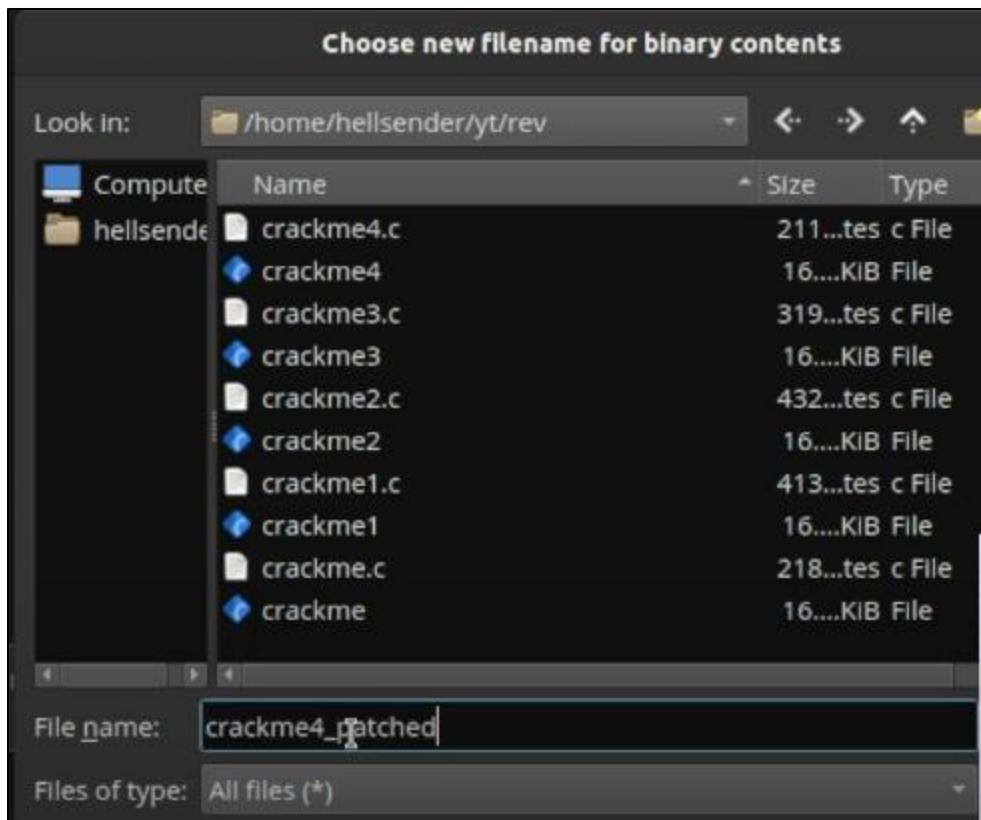
```

To yha pr hum invert branch kr lete hai.

```
call    __isoc99_scanf
mov     eax, dword [rbp-0xc {var_14}]
cmp     eax, 1234567890
je      0x4011f2
40204a] {"Wrong!!!"}                                     ↓
                                                       ↗
lea     rdi, [rel data_402020] {"You ha
call    puts
jmp    0x4011fe
```

Aur hum ise **save as** kr lete hai.





Ise run krke dekhte hai.

```
→ rev chmod +x crackme4_patched
→ rev ./crackme4_patched
Enter Input : 63327636237
You have cracked me! Now go and patch me!
→ rev
```

Yha kuch bhi password dalenge to wrong aayega.

```
→ rev ./crackme4_patched
Enter Input : aasasas
You have cracked me! Now go and patch me!
```

Agar hum sahi password dalenge to wrong aayega.

```
→ rev ./crackme4_patched
Enter Input : 1234567890
Wrong!!!
→ rev
```

Ab hum ise kisi ko bhi share kr skte hai. wo ise use kr skta hai.

=====

Yha pr ek aur **disassembler** ki hum bat kr lete hai. **objdump** jo command line pr disassemble krke dikha deta hai.

```
→ rev objdump -D crackme4 -M intel --disassemble=main
```

-D for disassemble file

-M intel for intel flavor

--disassembler=main for disassemble only main function otherwise it will disassemble all functions.

```
hellsender@hellsender:~/yt/rev 116x28

Disassembly of section .init:
Disassembly of section .plt:
Disassembly of section .plt.sec:
Disassembly of section .text:

0000000000401196 <main>:
401196: 31 0f 1e fa        endbr64
40119a: 55                 push    rbp
40119b: 48 89 e5           mov     rbp,rs
40119e: 48 83 ec 10         sub    rsp,0x10
4011a2: 64 48 8b 04 25 28 00   mov    rax,QWORD PTR fs:0x28
4011a9: 00 00
4011ab: 48 89 45 f8         mov    QWORD PTR [rbp-0x8],rax
4011af: 31 c0               xor    eax,eax
4011b1: 48 8d 3d 50 0e 00 00   lea    rdi,[rip+0xe50]      # 402008
4011b8: b8 00 00 00 00         mov    eax,0x0
4011bd: e8 ce fe ff ff       call   401090 <printf@plt>
4011c2: 48 8d 45 f4           lea    rax,[rbp-0xc]
4011c6: 48 89 c6               mov    rsi,rax
4011c9: 48 8d 3d 47 0e 00 00   lea    rdi,[rip+0xe47]      # 402017
```

Yha pr ye **opcodes (Operation Codes)** hai.

Jaisa ki jante hai humara processor **call, mov, sub, add instructions** ko pdkar nhi samajh pata ki use kya krna hai. wo inhi **opcode** ko pdkr samajhta hai ki use kya krna hai.

```
0000000000401196 <main>:
401196: 31 0f 1e fa        endbr64
40119a: 55                 push    rbp
40119b: 48 89 e5           mov     rbp,rs
40119e: 48 83 ec 10         sub    rsp,0x10
4011a2: 64 48 8b 04 25 28 00   mov    rax,QWORD PTR fs:0x28
4011a9: 00 00
4011ab: 48 89 45 f8         mov    QWORD PTR [rbp-0x8],rax
4011af: 31 c0               xor    eax,eax
4011b1: 48 8d 3d 50 0e 00 00   lea    rdi,[rip+0xe50]      # 402008
```

To yha pr do opcodes hai. **75 0e** to isme ka **First opcode (75)** hume batata hai ki kaun sa instruction use ho rha hai. Aur **Second opcode(oe)** hume **offset** batata hai ki hume kitna dur jump krni hai.

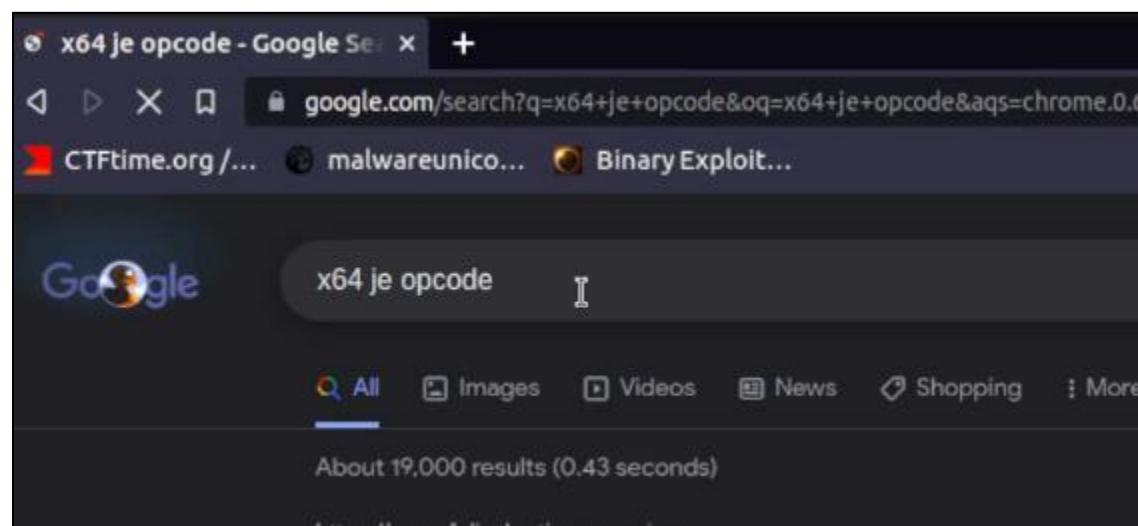
```
4011d5:    e8 c6 fe ff ff        call   4010a0 <_isoc99_scanf@plt>
4011da:    8b 45 f4            mov    eax,DWORD PTR [rbp-0xc]
4011dd:    3d d2 02 96 49      cmp    eax,0x499602d2
4011e2:    75 0e              jne    4011f2 <main+0x5c>
4011e4:    48 8d 3d 35 0e 00 00 lea    rdi,[rip+0xe35]      # 402020 <_IO
4011eb:    e8 80 fe ff ff      call   401070 <puts@plt>
4011f0:    eb 0c              jmp    4011fe <main+0x68>
4011f2:    48 8d 3d 51 0e 00 00 lea    rdi,[rip+0xe51]      # 40204a <_IO
```

Kitni line bad hai. **4011e4** se **4011f2** ki duri kitni hai. use second opcode batata hai.

```
4011dd:    3d d2 02 96 49      cmp    eax,0x499602d2
4011e2:    75 0e              jne    4011f2 <main+0x5c>
4011e4:    48 8d 3d 35 0e 00 00 lea    rdi,[rip+0xe35]
4011eb:    e8 80 fe ff ff      call   401070 <puts@plt>
4011f0:    eb 0c              jmp    4011fe <main+0x68>
4011f2:    48 8d 3d 51 0e 00 00 lea    rdi,[rip+0xe51]
4011f9:    e8 72 fe ff ff      call   401070 <puts@plt>
4011fe:    90                  nop
```

To yha pr hum **jne** ke opcode ko **je** ke opcode se replace kr denge.

Agar hum **je** ka **opcode** pta krna ho to isi disassembly me dekh skte hai ya fir google kr skte hai.



Uske bad hum kisi bhi website se dekh skte hai.

x64 je opcode - Google Search Intel x86 JUMP quick refer... +

Not secure | unixwiz.net/techtips/x86-jumps.html

CTFtime.org... malwareunico... Binary Exploit...

UNIXWIZ.NET

- Home
- Contact
- About
- TechTips
- Tools&Source
- Evo Payroll
- CmdLetters
- Research
- AT&T 3B2
- Advisories
- News/Pubs
- Literacy
- Calif.Voting
- Personal
- Tech Blog
- Evo Blog

Getting the sense for jumps and flags has long been a troublesome area for me, especially since the similar-sounding names. Looking more closely I found that many of the instructions were synonyms for not needed, and in the process found that my copy of Intel's *80386 Programmer's Reference Manual* instructions.

So I have grouped these functionally, with all instruction synonyms in the same row.

Instruction	Description	signed-ness	Flags	short jump opcodes	near jump opcodes
JO	Jump if overflow		OF = 1	70	0F 80
JNO	Jump if not overflow		OF = 0	71	0F 81
JS	Jump if sign		SF = 1	78	0F 88
JNS	Jump if not sign		SF = 0	79	0F 89
JE JZ	Jump if equal Jump if zero		ZF = 1	74	0F 84
JNE JNZ	Jump if not equal Jump if not zero		ZF = 0	75	0F 85

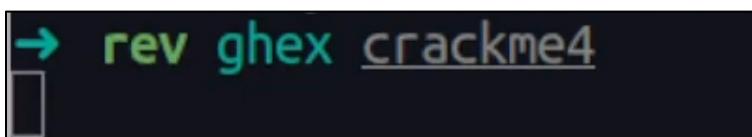
JE JZ	Jump if equal Jump if zero		ZF = 1	74	0F 84
JNE JNZ	Jump if not equal Jump if not zero		ZF = 0	75	0F 85

Ab ise edit krne ke liye hum **hexeditor** ka use karenge. Yha pr koi bhi hex **editor** use kr skte hai.

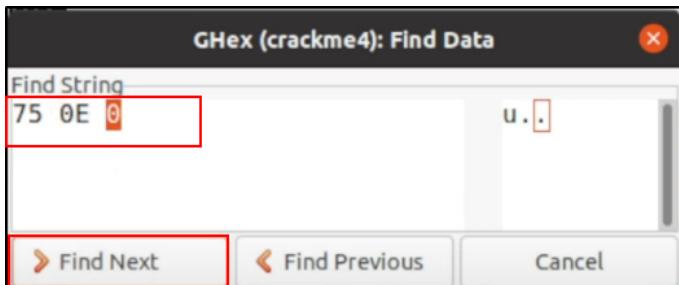
As we know **hexeditor** kisi binary ke **opcodes** ko show krta hai.

Agar aap **cli hexeditor** chahiye to **hexedit** use kr skte hai.

Yha pr hum **ghex** use karenge isko krne ke liye.



The screenshot shows the crackme4 - GHex application window. The menu bar at the top includes File, Edit, View, Windows, and Help. The Edit menu is currently active and open, displaying several options: Undo (Ctrl+Z), Redo (Shift+Ctrl+Z), Copy (Ctrl+C), Cut (Ctrl+X), Paste (Ctrl+V), Find (Ctrl+F) which is highlighted with a red box and has a cursor over it, Advanced Find (Ctrl+F), Replace (Ctrl+H), Goto Byte... (Ctrl+J), Insert Mode (Insert), and Preferences. Below the menu bar, there are status bars for memory sizes: Unsigned 8 bit: 127, Signed 32 bit: 1179403647, Unsigned 32 bit: 1179403647, Signed 16 bit: 17791, and Signed 64 bit: 1179403647. The main window area displays memory dump data starting with address 00000000.



To ise mil gaya ki yha pr ye wala opcode hai.

```
File Edit View Windows Help
000001150C3 66 66 2E 0F 1F 84 00 00 00 00 00 00 0F
000001160F3 0F 1E FA 80 3D DD 2E 00 00 00 00 75 13
000001170E5 E8 7A FF FF C6 05 CB 2E 00 00 01
000001180C3 66 66 2E 0F 1F 84 00 00 00 00 00 00 0F
000001190F3 0F 1E FA EB 8A F3 0F 1E FA 55 48 89
0000011A0EC 10 64 48 8B 04 25 28 00 00 00 48 89
0000011B0C0 48 8D 3D 50 0E 00 00 B8 00 00 00 00
0000011C0FF FF 48 8D 45 F4 48 89 C6 48 8D 3D 47
0000011D0B8 00 00 00 00 E8 C6 FE FF FF 8B 45 F4
0000011E096 49 75 0E 48 8D 3D 35 0E 00 00 E8 80
Signed 8 bit: 117 Signed 32 bit: -1924657547
```

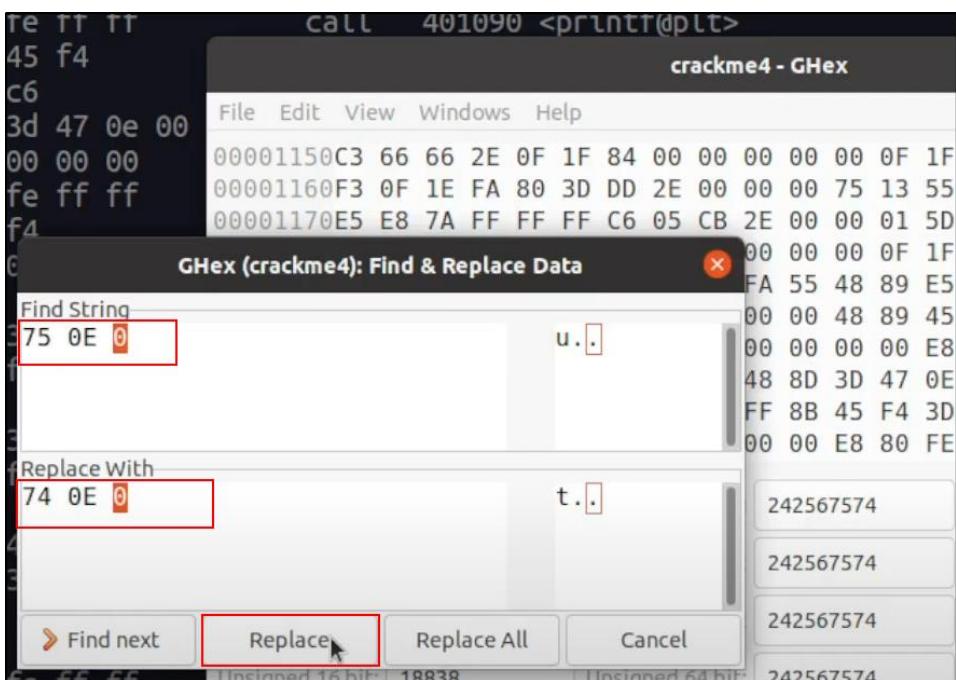
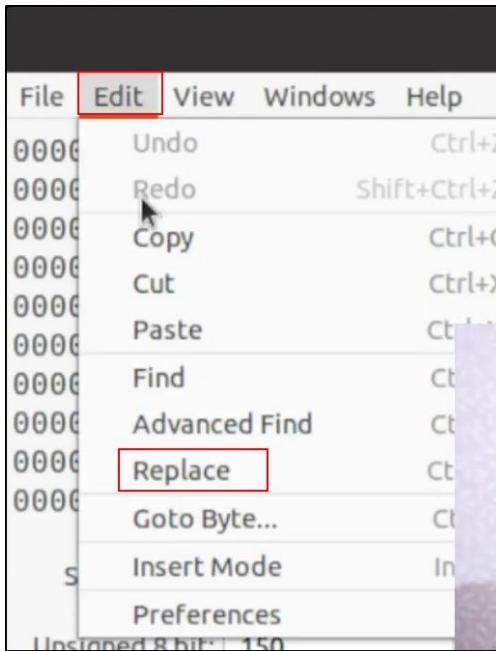
Iske aage aur piche ke opecodes jarur match kr le.

4011dd:	3d d2 02 96 49	cmp	eax,0x49
4011e2:	75 0e	jne	4011f2 <
4011e4:	48 8d 3d 35 0e 00 00	lea	rdi,[rip]

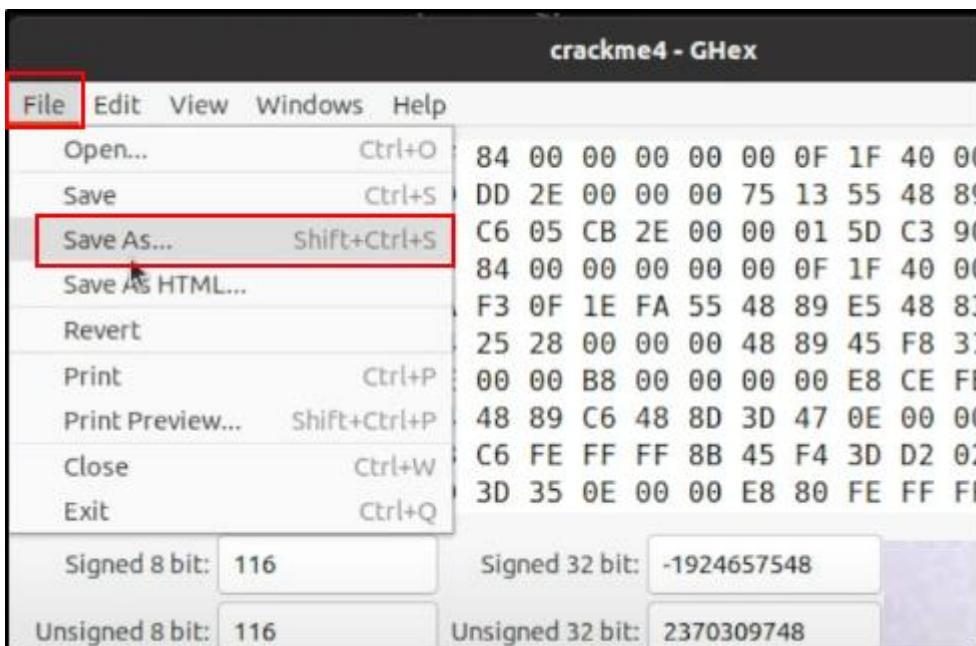
Jaisa ki dono me hai 96 49 **75 0e** 48 8d 3d jo ki sahi hai. yha capital aur small matter nhi krta hai.

Yha achhe se match jarur ke le otherwise ek hi byte pure binary ko corrupt kr skti hai.

Ab hum replace karenge.



Ab hum ise **save as** kr denge.



Run and test.

```
→ rev chmod +x crackme4_patch
→ rev ./crackme4_patch
Enter Input : 1234567
You have cracked me! Now go and patch me!
→ rev
```

Agar hum disassembly me dekhe to.

```
→ rev objdump -D crackme4_patch -M intel --disassemble=main
```

```
4011da: 8b 45 f4          mov    eax,DWORD PTR [rbp-0xc]
4011dd: 3d d2 02 96 49    cmp    eax,0x499602d2
4011e2: I74 0e             je    4011f2 <main+0x5c>
4011e4: 48 8d 3d 35 0e 00 00  lea    rdi,[rip+0xe35]
4011eb: e8 80 fe ff ff    call   401070 <puts@plt>
```

Is tarah ki software ko hum generally **cracked** software but ye **patched** software hote hai.

```
#####
#####
```

Anti-Reversing Techniques, How To Deal With Stripped Binaries

```
→ rev ./crackme5
[BINARY] [KEY]
→ rev ./crackme5 12345678
Bad Serial
→ rev █
```

```
→ rev strings crackme5
/lib64/ld-linux-x86-64.so.2
;vB\
libc.so.6
exit
puts
strlen
__libc_start_main
GLIBC_2.2.5
__gmon_start__
H=@@@
[]A\A]A^A
[BINARY] [KEY]
Bad Serial
Good Serial
:*3$"
GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
.shstrtab
.interp
```

Jaisa ki hum jante hai. binary 2 type ki hoti hai **stripped** and **not stripped**.

Ek aur type hota hai. **binary with debug symbols**

Binary with debug symbols – Isme pura source code chala jata hai jo debugger ko bahut help krta hai. ye debug purpose ke liye hi taiyar kiya jata hai.

Let's create these all.

```
→ symbols gcc crackme5.c -o crackme5 -no-pie Not stripped (Normal)
→ symbols gcc -s crackme5.c -o crackme5_stripped -no-pie Stripped
→ symbols gcc -g gdb crackme5.c -o crackme5_debugged -no-pie
gcc: error: gdb: No such file or directory
→ symbols gcc -ggdb crackme5.c -o crackme5_debugged -no-pie Binary with debug symbols
→ symbols
```

```
→ symbols ls
crackme5 crackme5.c crackme5_debugged crackme5_stripped
→ symbols
```

Yha hum pwndbg ka use karenge. Jiske liye hum iska official github profile visite kr skte hai.

The screenshot shows the GitHub repository page for pwndbg. It features the repository name "pwndbg" in large letters, followed by a brief description: "pwndbg (/poundbaeg/) is a GDB plug-in that makes debugging with GDB suck less, with a focus on features needed by low-level software developers, hardware hackers, reverse-engineers and exploit developers." Below the description are links for "license MIT" and "Discord 35 online".

How to install it.

<https://github.com/pwndbg/pwndbg>

```
git clone https://github.com/pwndbg/pwndbg
cd pwndbg
./setup.sh
```

Let's open these files in **gdb**.

Sabse phle hum **stripped** binary ko open krte hai.

```
→ symbols gdb crackme5_stripped
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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:
<http://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"...
pwndbg: loaded 195 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from crackme5_stripped...
(No debugging symbols found in crackme5_stripped)
pwndbg>
```

Yha isko koi bhi debugging symbols nhi mile.

Ab isme functions dekhte hai.

```
pwndbg> info functions
All defined functions:
I
Non-debugging symbols:
0x0000000000401060  puts@plt
0x0000000000401070  strlen@plt
0x0000000000401080  exit@plt
pwndbg>
```

Yha pr sirf teen functions hi dikhe yh bhi nhi pta chala ki main function hai ki nhi.

Agar hum main ko disassemble kare to.

```
pwndbg> disassemble main
No symbol table is loaded. Use the "file" command.
pwndbg>
```

To iske pas symble table nhi hai. ki yh pta kr paye main ke bare me.

Ab hum **not stripped** ko gdb me load krte hai.

```
→ symbols gdb crackme5
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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:
<http://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"...
pwndbg: loaded 195 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from crackme5...
(No debugging symbols found in crackme5)
pwndbg>
```

Yha hum dekh skte hai ki yha pr bhi debugging symbols nhi mile.

Aur functions dekhe to.

```
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x0000000000401000  _init
0x0000000000401060  puts@plt
0x0000000000401070  strlen@plt
0x0000000000401080  exit@plt
0x0000000000401090  _start
0x00000000004010c0  __dl_relocate_static_pie
0x00000000004010d0  deregister_tm_clones
0x0000000000401100  register_tm_clones
0x0000000000401140  __do_global_dtors_aux
0x0000000000401170  frame_dummy
0x0000000000401176  check_key
0x0000000000401209  main
0x0000000000401270  __libc_csu_init
0x00000000004012e0  __libc_csu_fini
0x00000000004012e8  _fini
pwndbg>
```

Yha **pr main** function hai.

Ab hum debugged binary ko load krte hai.

```
→ symbols gdb crackme5_debugged
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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:
<http://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"...
pwndbg: loaded 195 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from crackme5_debugged...
pwndbg>
```

Yha hum dekh skte hai ki ise debugging symble mile aur ye use read kr rha hai.

Agar hum show functions kare to.

```
pwndbg> info functions
All defined functions:

File crackme5.c:
5:     int check_key(const char *);
18:     int main(int, char **); I

Non-debugging symbols:
0x0000000000401000 _init
0x0000000000401060 puts@plt
0x0000000000401070 strlen@plt
0x0000000000401080 exit@plt
0x0000000000401090 _start
0x00000000004010c0 __dl_relocate_static_pie
0x00000000004010d0 deregister_tm_clones
0x0000000000401100 register_tm_clones
0x0000000000401140 __do_global_dtors_aux
0x0000000000401170 frame_dummy
0x0000000000401270 __libc_csu_init
0x00000000004012e0 __libc_csu_fini
0x00000000004012e8 _fini
pwndbg>
```

Yha ise do functions mile c program ke jise ye sbse upar diya hai. aur uska data type bhi bta rha hai.

Agar hum list command run kare to ye source code bhi dikha data hai.

```
pwndbg> list
5     int check_key(const char *key) {
6         int i;
7
8         if ( strlen(key) != 16 )
9             return 1;
10        for ( i = 0; i < strlen(key); i += 2 )
11        {
12            if ( key[i] - key[i + 1] != -1 )
13                return 1; I
14        }
pwndbg>
```

Agar hume main function ka source code dekhna hai to.

```
pwndbg> list main
14        }
15        return 0;
16    }
17
18    int main(int argc, char *argv[])
19    {
20        if ( argc != 2 ) {
21            puts("[BINARY] [KEY]");
22            exit(0);
23    }
pwndbg> =====
```

Ab problem ye hai ki hum stripped binary ki reverse engineering kaise karenge.

```
→ symbols ls
crackme5  crackme5.c  crackme5_debugged  crackme5_stripped
→ symbols
```

Jisme hume **entry point** ya **main function** kaise pta karenge.

Yha pr humne ek stripped binary ready ki hai. use pwndbg load kr lete hai.

```
→ rev gdb crackme5
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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:
<http://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"...
pwndbg: loaded 195 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from crackme5...
(No debugging symbols found in crackme5)
pwndbg>
```

Yha pr debugging symbles nhi milenge.

Aur agar hum functions dekhne ki koshish kare to.

```
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x0000000000401060  puts@plt
0x0000000000401070  strlen@plt
0x0000000000401080  exit@plt
pwndbg> disassemble main
No symbol table is loaded. Use the "file" command.
pwndbg>
```

Yha koi main function nhi mila.

To kya hota hai jb bhi hum **C program** ko gcc ki help compile krte hai. to wo uske sath
bahut sare functions apne side se add krta hai. jinme se do function hote hai. ek hota
hai **_start** aur dusra hota hai **__libc_start_main**

To humara **main** function directly call nhi hota. Sabse phle ek function hota hai. jo **_start** ko call krta hai.

Binary jaise hi run hoti hai to sbse phle **_start** function call hota hai. to ye function call krta hai **__libc_start_main** function ko. Aur **__libc_start_main** function **main** function ko call krta hai.

To hum unhi do functions ko follow karenge wahi functions aage jake hume main function tk pahuchayenge.

Aap likh diya main break **main** to **main** function nhi nikalega.

Main function run hone se phle bahut kuchh ho rha hota hai aur uske bad bahut kuchh ho rha hota hai binary ke andar abhi ke liye use hum nhi samjhenge.

Sabse phle hume **_start** ka address nikalna hoga to usko nikalne ke do tarike hai.

First way:

Yha **info file** karenge to sare sections open ho jayenge.

```
pwndbg> info file
Symbols from "/home/ubuntu/crackme5".
Local exec file:
  '/home/ubuntu/crackme5', file type elf64-x86-64.
Entry point: 0x401090
0x0000000000400318 - 0x0000000000400334 is .interp
0x0000000000400338 - 0x0000000000400358 is .note.gnu.property
0x0000000000400358 - 0x000000000040037c is .note.gnu.build-id
0x000000000040037c - 0x000000000040039c is .note.ABI-tag
0x00000000004003a0 - 0x00000000004003bc is .gnu.hash
0x00000000004003c0 - 0x0000000000400450 is .dynsym
0x0000000000400450 - 0x0000000000400499 is .dynstr
0x000000000040049a - 0x00000000004004a6 is .gnu.version
0x00000000004004a8 - 0x00000000004004c8 is .gnu.version_r
0x00000000004004c8 - 0x00000000004004f8 is .rela.dyn
0x00000000004004f8 - 0x0000000000400540 is .rela.plt
0x0000000000401000 - 0x000000000040101b is .init
0x0000000000401020 - 0x0000000000401060 is .plt
0x0000000000401060 - 0x0000000000401090 is .plt.sec
0x0000000000401090 - 0x00000000004012e5 is .text
0x00000000004012e8 - 0x00000000004012f5 is .fini
0x0000000000402000 - 0x000000000040202a is .rodata
0x000000000040202c - 0x0000000000402078 is .eh_frame_hdr
0x0000000000402078 - 0x00000000004021a0 is .eh_frame
0x0000000000403e10 - 0x0000000000403e18 is .init_array
0x0000000000403e18 - 0x0000000000403e20 is .fini_array
0x0000000000403e20 - 0x0000000000403ff0 is .dynamic
0x0000000000403ff0 - 0x0000000000404000 is .got
0x0000000000404000 - 0x0000000000404030 is .got.plt
0x0000000000404030 - 0x0000000000404040 is .data
0x0000000000404040 - 0x0000000000404048 is .bss
```

pwndbg>

Jaisa ki hum assembly pdi hai. to **_start** kis section me aayega.

Wo **.text** section me aata hai.

Yha execution start **.text** section ek address se hoga.

```
pwndbg> info file
Symbols from "/home/ubuntu/crackme5".
Local exec file:
  '/home/ubuntu/crackme5', file type elf64-x86-64.
  Entry point: 0x401090
  0x0000000000400318 - 0x0000000000400334 is .interp
  0x0000000000400338 - 0x0000000000400358 is .note.gnu.property
  0x0000000000400358 - 0x000000000040037c is .note.gnu.build-id
```

```
  0x0000000000401020 - 0x0000000000401060 is .plt
  0x0000000000401060 - 0x0000000000401090 is .plt.sec
  0x0000000000401090 - 0x00000000004012e5 is .text
  0x00000000004012e8 - 0x00000000004012f5 is .fini
```

Jaisa ki hum upar dekh pa rha hai **Entry point** pr likha address aur **.text** section ka address same hai.

Second way:

Agar hum **gdb** pr jaye aur **start** command run kare to automatic start function pr le jayega jo humne abhi dekha.

```

pwndbg> start
Program stopped.
0x00007ffff7fe3290 in _start () from /lib64/ld-linux-x86-64.so.2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Disabling the emulation via Unicorn Engine that is used for computing branches as there isn't enough memory (e also:
* https://github.com/pwndbg/pwndbg/issues/1534
* https://github.com/unicorn-engine/unicorn/pull/1743
Either free your memory or explicitly set `set emulate off` in your Pwndbg config
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA [ REGISTERS / show-flags off / show-compact-reg off ]
RAX 0x1c
RBX 0
RCX 0xfffffffffe448 -> 0x7fffffffef6cf ← 'SHELL=/bin/bash'
RDX 0x7ffff7fc9040 (_dl_fini) ← endbr64
RDI 0x7ffff7ffe2e0 ← 0
RSI 0x7ffff7ffe888 ← 0
R8 0x7fff
R9 0xf
R10 0x7ffff7fc3860 ← 0xd0012000000c1
R11 0x206
R12 0x401090 ← endbr64
R13 0x7fffffffef430 ← 1
R14 0
R15 0
RBP 0
RSP 0x7fffffffef430 ← 1
RIP 0x401090 ← endbr64

RBP 0
RSP 0x7fffffffef430 ← 1
RIP 0x401090 ← endbr64 [ DISASM / x86-64 / set emulate off ]
- 0x401090 endbr64
0x401094 xor    ebp, ebp          EBP => 0
0x401096 mov    r9, rdx
0x401099 pop    rsi
0x40109a mov    rdx, rsp
0x40109d and    rsp, 0xfffffffffffffff0
0x4010a1 push   rax
0x4010a2 push   rsp
0x4010a3 mov    r8, 0x4012e0      R8 => 0x4012e0 ← endbr64
0x4010aa mov    rcx, 0x401270      RCX => 0x401270 ← endbr64
0x4010b1 mov    rdi, 0x401209      RDI => 0x401209 ← endbr64
[ STACK ]
00:0000| r13 rsp 0x7fffffffef430 ← 1
01:0008|          0x7fffffffef438 -> 0x7fffffffef6b9 ← '/home/ubuntu/crackme5'
02:0010|          0x7fffffffef440 ← 0
03:0018| rcx   0x7fffffffef448 -> 0x7fffffffef6cf ← 'SHELL=/bin/bash'
04:0020|          0x7fffffffef450 -> 0x7fffffffef6df ← 'PWD=/home/ubuntu'
05:0028|          0x7fffffffef458 -> 0x7fffffffef6f0 ← 'LOGNAME=ubuntu'
06:0030|          0x7fffffffef460 -> 0x7fffffffef6ff ← 'XDG_SESSION_TYPE=tty'
07:0038|          0x7fffffffef468 -> 0x7fffffffef714 ← '_=/usr/bin/gdb'
[ BACKTRACE ]
▶ 0          0x401090 None
  1          0x1  None
  2  0x7fffffffef6b9 None
  3          0x0  None

pwndbg> |

```

Ab agar hum disassemble run kare to.

```

pwndbg> disassemble
No function contains program counter for selected frame.
pwndbg> |

```

Ise koi function nhi mila. gdb symbles hote hai usi se batata hai.

To gdb ye nhi kr payega kyoki iske pass koi symble nhi hai. ise samjh nhi aayega. Pta nhi chalega.

disassemble command ek function ko disassembly krta hai.

=====

pwndbg me hum **registers**, **disassembly**, **stack** aur **backtrace** dekh skte hai.

backtrace hume ye batata hai ki kaun-2 se functions ko humne call kiya hai.

ye note krta hai ki hum kis-2 functions me jate ja rhe hai.

Agar hume **gdb** me kisi chij ko dekhna hai to use krte hai.

To yah pr dete hai. phle hum likhenge **x/** (exmine) fir hume kitni value dekhni hai. jaise 30 fir hume kya dekhna hai hex, string, instruction ya address.

Hex → **x**

String → **s**

Address → **a**

Instruction → **i**

man lo hume 30 instructions dekhne hai ek function ka.

x/30i <address_of_entry_point>

pwndbg> x/30i 0x401090

Yha pr hum **rip** ka bhi use kr skte hai. kyoki execution pointer entry point pr hai aur next execution rip me hota hai. aur kisi bhi **register** ko use krne se phle \$ likhna pdta hai.

```
pwndbg> x/30i $rip
=> 0x401090:    endbr64
  0x401094:    xor    ebp,ebp
  0x401096:    mov    r9,rdx
  0x401099:    pop    rsi
  0x40109a:    mov    rdx,rsp
  0x40109d:    and    rsp,0xfffffffffffff0
  0x4010a1:    push   rax
  0x4010a2:    push   rsp
  0x4010a3:    mov    r8,0x4012e0
  0x4010aa:    mov    rcx,0x401270
  0x4010b1:    mov    rdi,0x401209
  0x4010b8:    call   QWORD PTR [rip+0x2f32]      # 0x403ff0
```

Yha pr assembly pd skte hai lekin humare kam ka nhi hai.

To hum **ni** krke instructions ko pdne ki koshish krte hai.

```
pwndbg> ni
```

Yha next intruction pr pahuch gye.

```
[ DISASM ]-
0x401090  endbr64
► 0x401094  xor    ebp, ebp
  0x401096  mov    r9, rdx
  0x401099  pop    rsi
  0x40109a  mov    rdx, rsp
  0x40109d  and    rsp, 0xfffffffffffff0
  0x4010a1  push   rax
  0x4010a2  push   rsp
  0x4010a3  mov    r8, 0x4012e0
  0x4010aa  mov    rcx, 0x401270
  0x4010b1  mov    rdi, 0x401209
[ STACK ]-
```

Yha pr hum ni krke call syscall pr pahuch jayenge.

```
[ DISASM ]
0x4010a1    push    rax
0x4010a2    push    rsp
0x4010a3    mov     r8, 0x4012e0
0x4010aa    mov     rcx, 0x401270
0x4010b1    mov     rdi, 0x401209
► 0x4010b8    call    qword ptr [rip + 0x2f32] < libc_start_main>
0x4010be    hlt
0x4010bf    nop
0x4010c0    endbr64
0x4010c4    ret
0x4010c5    nop     word ptr cs:[rax + rax]
```

Yhi function bad me fir **main** ko call krta hai.

2nd Trick to find main function ka address:

_start function jb **__libc_start_main** ko **call** krta hai to ise kuchh **arguments** pass krne pdte hai. to ye alag-2 argument pass krta hai. lekin humare liye ek argument jaruri hota hai. **rdi** ke andar ka argument.

jaisa ki hum jante hai **rdi** ke andar sbse phla **argument** jata hai kisi bhi function ka.

Ye phle argument me binary ke **main** address ka address mangta hai **_start** fucntion se.

Agar hum **rdi** ka value dekhe to

```
RCX  0x401270 ← endbr64
RDX  0x7fffffffdf78 → 0x7fffffff fe2c9 ← '/home/hellsender/yt/r
*RDI 0x401209 ← endbr64
RSI  0x1
R8   0x4012e0 ← endbr64
R9   0x7ffff7fe0d50 ← endbr64
R10  0x7
R11  0x2
R12  0x401090 ← endbr64
R13  0x7fffffffdf70 ← 0x1
R14  0x0
R15  0x0
RBP  0x0
RSP  0x7fffffffdf60 → 0x7fffffff df68 ← 0x1c
*RIP  0x4010b8 ← call    qword ptr [rip + 0x2f32]
```

[DISASM]-

Humne phle hi **main** function ka address find kr liya hai.

Yha pr humne main function ka address aise hi nikal liye **__libc_start_main** pr jane ki jarurat hi nhi hui.

```
pwndbg> b *0x401209
```

Ab yha pr breakpoint lagakr continue kr denge to hum main function me pahuch jaenge.

Lekin agar hum **__libc_start_main** me pahuch gye ho to kaise main me pahuchna hai. wo bhi dekh lete hai.

To hum **si (step into)** krte hai.

```
pwndbg> si
```

```
R15 0x0
RBP 0x0
*RSP 0x7fffffffdf58 → 0x4010be ← hlt
*RIP 0x7ffff7ddffc0 (<__libc_start_main>) ← endbr64
[ DISASM ]
▶ 0x7ffff7ddffc0 <__libc_start_main>      endbr64
 0x7ffff7ddfc4 <__libc_start_main+4>    push r15
 0x7ffff7ddfc6 <__libc_start_main+6>    xor eax, eax
 0x7ffff7ddfc8 <__libc_start_main+8>    push r14
 0x7ffff7ddfc a <__libc_start_main+10>   push r13
 0x7ffff7ddffcc <__libc_start_main+12>   push r12
 0x7ffff7ddffce <__libc_start_main+14>   push rbp
 0x7ffff7ddffcf <__libc_start_main+15>   push rbx
 0x7ffff7ddffd0 <__libc_start_main+16>   mov rbx, rcx
 0x7ffff7ddffd3 <__libc_start_main+19>   sub rsp, 0x98
 0x7ffff7ddffda <__libc_start_main+26>   mov qword ptr [rsp + 8], rdx
[ STACK ]
00:0000  rsp 0x7fffffffdf58 → 0x4010be ← hlt
01:0008  0x7fffffffdf60 → 0x7fffffffdf68 ← 0x1c
02:0010  0x7fffffffdf68 ← 0x1c
03:0018  r13 0x7fffffffdf70 ← 0x1
04:0020  rdx 0x7fffffffdf78 → 0x7fffffff fe2c9 ← '/home/hellsender/yt/rev/crackme'
05:0028  0x7fffffffdf80 ← 0x0
06:0030  0x7fffffffdf88 → 0x7fffffff fe2ea ← 'GJS_DEBUG_TOPICS=JS_ERROR;JS_LI'
```

Ab hum **ni(next instruction)** krte jate hai aur ek hi jahan hume **call rax** milega. Wahan pahuchte hai. aur **rax** me humare **main** address ka value hogा.

```

[ DISASM ]
0x7ffff7de0099 <__libc_start_main+217>    mov    rax, qword ptr [rip + 0x1c3e10]
0x7ffff7de00a0 <__libc_start_main+224>    mov    rsi, qword ptr [rsp + 8]
0x7ffff7de00a5 <__libc_start_main+229>    mov    edi, dword ptr [rsp + 0x14]
0x7ffff7de00a9 <__libc_start_main+233>    mov    rdx, qword ptr [rax]
0x7ffff7de00ac <__libc_start_main+236>    mov    rax, qword ptr [rsp + 0x18]
► 0x7ffff7de00b1 <__libc_start_main+241>    call   rax <0x401209>
0x7ffff7de00b3 <__libc_start_main+243>    mov    edi, eax
0x7ffff7de00b5 <__libc_start_main+245>    call   exit <exit>
0x7ffff7de00ba <__libc_start_main+250>    mov    rax, qword ptr [rsp + 8]    I
0x7ffff7de00bf <__libc_start_main+255>    lea    rdi, [rip + 0x18fda2]
0x7ffff7de00c6 <__libc_start_main+262>    mov    rsi, qword ptr [rax]
[ STACK ]
```

Yha hum dekh skte hai **rax** ke sath ek **address** hai. ye **main** function ka address hai.

Aur ye same **address** hai jo humne **rdi** ke andar ka **address** copy kiya tha.

Ab hum step into kr skte hai main function me.

```

pwndbg> si
```

```

[ DISASM ]
► 0x401209  endbr64
0x40120d  push   rbp    I
0x40120e  mov    rbp, rsp
0x401211  sub    rsp, 0x10
0x401215  mov    dword ptr [rbp - 4], edi
0x401218  mov    qword ptr [rbp - 0x10], rsi
0x40121c  cmp    dword ptr [rbp - 4], 2
0x401220  je     0x401238 <0x401238>

0x401222  lea    rdi, [rip + 0xddb]
0x401229  call   puts@plt <puts@plt>

0x40122e  mov    edi, 0
[ STACK ]
```

Ab hum **main** function ke andar padhar chuke hai.

To hum yha pr breakpoint laga dete hai. ki jb bhi hum program run kare to ye main function pr jakr run jaye.

```

pwndbg> b *0x401209
```

Agar aap pwndbg ke tarike se lagana chahte ho to...

```
pwndbg> bp 0x401209
```

To yha pr hum dekhna chahte hai ki main function me kya hai.

Yha hum **x/ (examine)** command use karenge.

```
pwndbg> x/20i $rip
=> 0x401209:    endbr64
    0x40120d:    push   rbp
    0x40120e:    mov    rbp,rsp
    0x401211:    sub    rsp,0x10
    0x401215:    mov    DWORD PTR [rbp-0x4],edi
    0x401218:    mov    QWORD PTR [rbp-0x10],rsi
    0x40121c:    cmp    DWORD PTR [rbp-0x4],0x2
    0x401220:    je     0x401238
    0x401222:    lea    rdi,[rip+0xddb]          # 0x402004
    0x401229:    call   0x401060 <puts@plt>
    0x40122e:    mov    edi,0x0
    0x401233:    call   0x401080 <exit@plt>
    0x401238:    mov    rax,QWORD PTR [rbp-0x10]
    0x40123c:    add    rax,0x8
    0x401240:    mov    rax,QWORD PTR [rax]
    0x401243:    mov    rdi,rax
    0x401246:    call   0x401176
    0x40124b:    test   eax,eax
    0x40124d:    je     0x40125d
    0x40124f:    lea    rdi,[rip+0xdbd]          # 0x402013
pwndbg>
```

Jaisa ki hum dekh skte hai ki yha pr **ret** nhi aaya. To hum instruction ke value bda letे hai.

```
pwndbg> x/23i $rip
=> 0x401209:    endbr64
  0x40120d:    push   rbp
  0x40120e:    mov    rbp,rsp
  0x401211:    sub    rsp,0x10
  0x401215:    mov    DWORD PTR [rbp-0x4],edi
  0x401218:    mov    QWORD PTR [rbp-0x10],rsi
  0x40121c:    cmp    DWORD PTR [rbp-0x4],0x2
  0x401220:    je     0x401238
  0x401222:    lea    rdi,[rip+0xddb]      # 0x402004
  0x401229:    call   0x401060 <puts@plt>
  0x40122e:    mov    edi,0x0
  0x401233:    call   0x401080 <exit@plt>
  0x401238:    mov    rax,QWORD PTR [rbp-0x10]
  0x40123c:    add    rax,0x8
  0x401240:    mov    rax,QWORD PTR [rax]
  0x401243:    mov    rdi,rax
  0x401246:    call   0x401176
  0x40124b:    test   eax,eax
  0x40124d:    je     0x40125d
  0x40124f:    lea    rdi,[rip+0xdbd]      # 0x402013
  0x401256:    call   0x401060 <puts@plt>
  0x40125b:    jmp   0x401269
  0x40125d:    lea    rdi,[rip+0xdba]      # 0x40201e
pwndbg> x/23i $rip
```

Yha **23** instruction pr bhi nhi aaya to hum instruction ki value bda lete hai.

```
pwndbg> x/30i $rip
=> 0x401209:    endbr64
  0x40120d:    push   rbp
  0x40120e:    mov    rbp,rsp
  0x401211:    sub    rsp,0x10
  0x401215:    mov    DWORD PTR [rbp-0x4],edi
  0x401218:    mov    QWORD PTR [rbp-0x10],rsi
  0x40121c:    cmp    DWORD PTR [rbp-0x4],0x2
  0x401220:    je     0x401238
  0x401222:    lea    rdi,[rip+0xddb]          # 0x402004
  0x401229:    call   0x401060 <puts@plt>
  0x40122e:    mov    edi,0x0
  0x401233:    call   0x401080 <exit@plt>
  0x401238:    mov    rax,QWORD PTR [rbp-0x10]
  0x40123c:    add    rax,0x8
  0x401240:    mov    rax,QWORD PTR [rax]
  0x401243:    mov    rdi,rax
  0x401246:    call   0x401176
  0x40124b:    test   eax,eax
  0x40124d:    je     0x40125d
  0x40124f:    lea    rdi,[rip+0xdbd]          # 0x402013
  0x401256:    call   0x401060 <puts@plt>
  0x40125b:    jmp   0x401269
  0x40125d:    lea    rdi,[rip+0xdba]          # 0x40201e
  0x401264:    call   0x401060 <puts@plt>
  0x401269:    mov    eax,0x0
  0x40126e:    leave 
  0x40126f:    ret
  0x401270:    endbr64
```

```
0x401269:    mov    eax,0x0
  0x40126e:    leave 
  0x40126f:    ret
  0x401270:    endbr64
  0x401274:    push   r15
  0x401276:    lea    r15,[rip+0x2b93]          # 0x403e10
pwndbg> nearpc
```

To yha ye **30** pr mil chuka hai. aur agla function **endbr64** se suru ho rha hai.

Agar aap **pwndbg** use kr rhe ho to **nearpc** command use kr skte hai. **rip** ko **pc (program counter)** bhi khte hai.

```

pwndbg> nearpc
► 0x401209 endbr64
0x40120d push rbp
0x40120e mov rbp, rsp
0x401211 sub rsp, 0x10
0x401215 mov dword ptr [rbp - 4], edi
0x401218 mov qword ptr [rbp - 0x10], rsi
0x40121c cmp dword ptr [rbp - 4], 2
0x401220 je 0x401238 <0x401238>

0x401222 lea rdi, [rip + 0xddb]
0x401229 call puts@plt <puts@plt>

0x40122e mov edi, 0
pwndbg>

```

Yha pr rip ke aas paas ke instruction dikh jayenge.

Man lo hume **27** instructions dekhne hai.

```

pwndbg> nearpc 27
► 0x401209 endbr64
0x40120d push rbp
0x40120e mov rbp, rsp
0x401211 sub rsp, 0x10
0x401215 mov dword ptr [rbp - 4], edi
0x401218 mov qword ptr [rbp - 0x10], rsi
0x40121c cmp dword ptr [rbp - 4], 2
0x401220 je 0x401238 <0x401238>

0x401222 lea rdi, [rip + 0xddb]
0x401229 call puts@plt <puts@plt>

0x40122e mov edi, 0
0x401233 call exit@plt <exit@plt>

0x401238 mov rax, qword ptr [rbp - 0x10]
0x40123c add rax, 8
0x401240 mov rax, qword ptr [rax]
0x401243 mov rdi, rax
0x401246 call 0x401176 <0x401176>

0x40124b test eax, eax
0x40124d je 0x40125d <0x40125d>

0x40125f lea rdi, [rip + 0xd11]

```

To yha pr **27** instructions dikh jayenge.

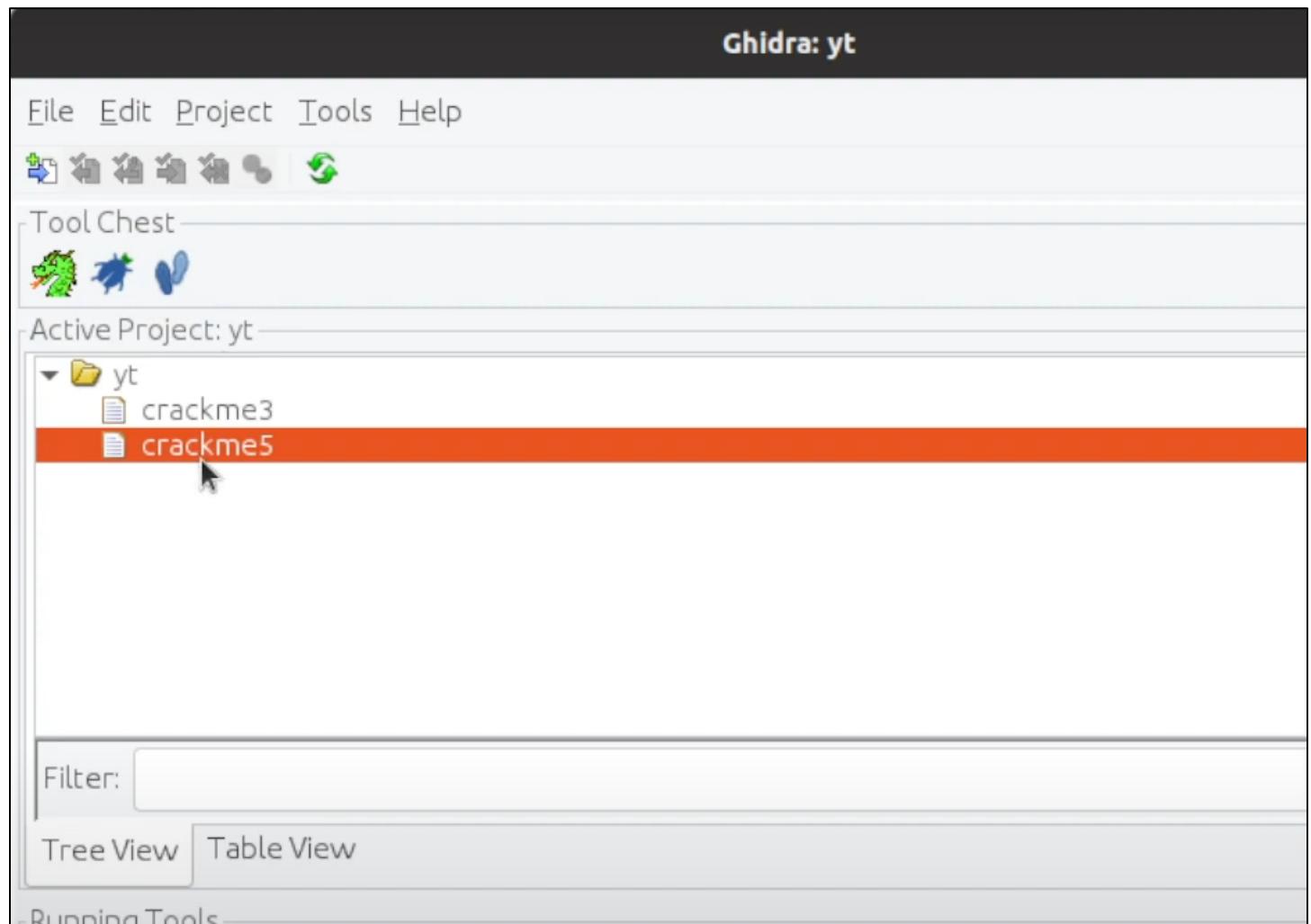
Lekin yha pr dono me difference hai.

```
0x40124d:    je      0x40125d
0x40124f:    lea     rdi,[rip+0xdbd]          # 0x402013
0x401256:    call    0x401060 <puts@plt>
0x40125b:    jmp     0x401269
0x40125d:    lea     rdi,[rip+0xdba]          # 0x40201e
0x401264:    call    0x401060 <puts@plt>
0x401269:    mov     eax,0x0
0x40126e:    leave
```

gdb connect krke registers ki hone wali value bta data hai lekin **pwndbg** nhi batata.

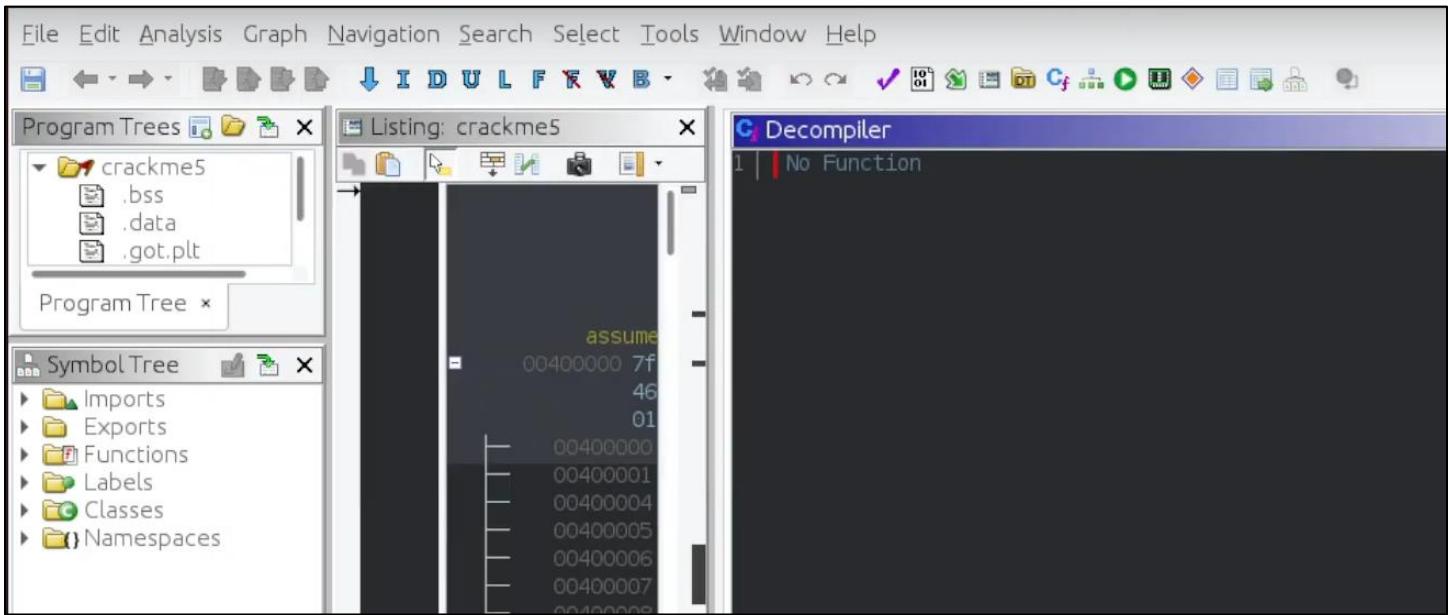
To yha pr hum assembly smajhne me bahut time lagega. To hum decompiler ka help lenge.

Ab hum apna **ghidra** open krte hai.

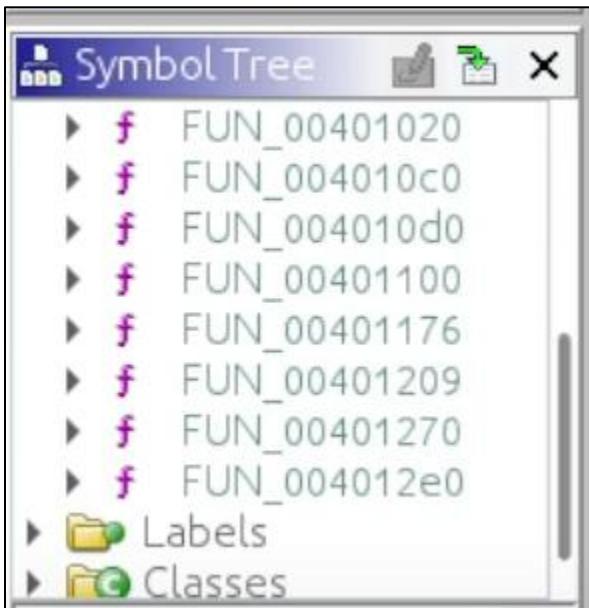


Ise hum **decompiler** pr drag kr dete hai. ya fir double click krke **decompile** kr dete hai.

Iske bad ye analyze ke liye puchega.



Yha pr hum functions ko expend krte hai.



To koi **main** functions nhi dikhega. Kyoki ye **stripped** binary hai. to hum sb pr click krke dekh le **main** function hume alag hi dikh jayega.

Yha pr hume **main** function mil gaya dekhne se hi lg rha hai.

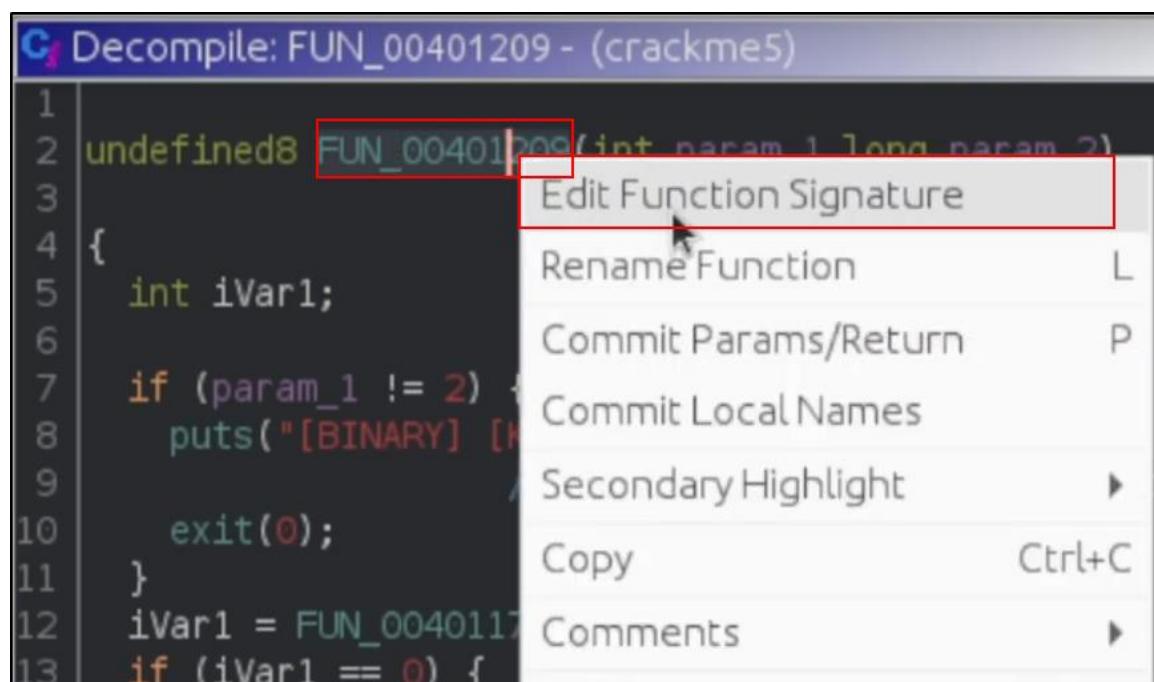
```
undefined8 FUN_00401209(int param_1, long param_2)

{
    int iVar1;

    if (param_1 != 2) {
        puts("[BINARY] [KEY]");
        /* WARNING: Subroutine does not return */
        exit(0);
    }
    iVar1 = FUN_00401176(*(undefined8 *)param_2 + 8));
    if (iVar1 == 0) {
        puts("Good Serial");
    }
    else {
        puts("Bad Serial");
    }
    return 0;
}
```

Yha pr hum functions ke nam change kr lete hai.

Function pr click krte hai aur > **edit function signature** pr click krte hai.



Edit Function at 00401209

```
int main (int argc, char **argv)
```

Function Name:

Calling Convention:

Function Attributes:

- Varargs In Line
- No Return Use Custom Stor

Function Variables

Index	Datatype	Name	Storage
1	undefined8	<RETURN>	RAX:8
2	int	param_1	EDI:4
	long	param_2	RSI:8

Yha first argument **int argc** means **argument counter** (total number of argument) that is abviously int hi hogta.

And second argument character type ka hogta. Jaisa ki hum jante hai jitne bhi argument aate hai wo **char**argv** ke andar aate hai. ye C ++ ke main function me lagana hi pdta hai.

C Decompile: main - (crackme5)

```
1 int main(int argc,char **argv)
2 {
3     int return_value;
4
5     if (argc != 2) {
6         puts("[BINARY] [KEY]");
7                     /* WARNING: Subrou
8         exit(0);
9     }
10    return_value = check_key(argv[1]);
11    if (return_value == 0) {
12        puts("Good Serial");
13    }
14    else {
15        puts("Bad Serial");
16    }
17    return 0;
18 }
```



Ab ye program phle se jyada easy ho gya. Ghidra automatic jo code generate krta hai. wo dekhne me hard lgta hai.

To ye program sabse check krta hai ki do argument pass hua hai ki nhi. Agar nhi to ye **[Binary] [key]** print krke program ko **exit** kr dega.

Agar key pass ho rhi hai to **check_key** function uske **2nd** argument means key ko check krke **return_value** me store kr dega. Yha **[1]** ka mtlb second argument agar zero diya hota to humara **binary file** ho jata.

Uske bad agar **return_value** ki value **0** hoti hai to “**good serial**” otherwise “**bad serial**”

Ab hum **check_key()** function pr double click karenge to ye hume **check_key()** function me enter kr dega.

C# Decompile: check_key - (crackme5)

```
1 undefined8 check_key(char *param_1)
2
3 {
4     size_t sVar1;
5     undefined8 uVar2;
6     int local_1c;
7
8     sVar1 = strlen(param_1);
9     if (sVar1 == 0x10) {
10         for (local_1c = 0; sVar1 = strlen(param_1), (ulong)(long)local_
11             local_1c = local_1c + 2) {
12             if ((int)param_1[local_1c] - (int)param_1[(long)local_1c + 1]
13                 return 1;
14             }
15         }
16     }
17     uVar2 = 0;
18 }
19 else {
20     uVar2 = 1;
21 }
22 return uVar2;
```

Ab hume easy kr lete hai.

0x10 pr right click krke ise decimal me convert ke lenge.

```
size_t keylen;
int local_1c;

keylen = strlen(key);
if (keylen == 0) {
    for (local_1c = 0; local_1c < keylen; local_1c = local_1c + 2)
        if ((int)key[local_1c] - (int)key[(long)local_1c + 1] != -1)
            return 1;
    }
    iVar1 = 0;
}
else {
    iVar1 = 1;
}
return iVar1;
```

Decompile: check_key

Decimal:

Octal:

```
C:\Decompile: check_key - (crackme5)
1 int check_key(char *key)
2
3 {
4     int iVar1;
5     size_t keylen;
6     int local_1c;
7
8     keylen = strlen(key);
9     if (keylen == 16) {
10         for (local_1c = 0; keylen = strlen(key), (ulong)(long)local_1c < keylen; local_1c = local_1c + 2)
11             if ((int)key[local_1c] - (int)key[(long)local_1c + 1] != -1) {
12                 return 1;
13             }
14         }
15         iVar1 = 0;
16     }
17     else {
18         iVar1 = 1;
19     }
20     return iVar1;
21 }
```

Let's try to understand.

C# Decompile: check_key - (crackme5)

```
1 int check_key(char *key)
2 {
3     int iVar1;
4     size_t keylen;
5     int i;
6
7     keylen = strlen(key);
8     if (keylen == 16) {
9         for (i = 0; keylen = strlen(key), (ulong)(long)i < keylen; i = i + 2) {
10            if ((int)key[i] - (int)key[(long)i + 1] != -1) {
11                return 1;
12            }
13        }
14        iVar1 = 0;
15    }
16    else {
17        iVar1 = 1;
18    }
19    return iVar1;
20 }
21 }
```

Let's try to make possible password.

```
→ rev ./crackme5 1212121212121212
Good Serial
→ rev
```

Ek aur try krte hai.

```
→ rev ./crackme5 3434343434343434
Good Serial
→ rev
```

Aise hi bahut sare combination ho skte hai.

```
→ rev ./crackme5 1234567812345678
Good Serial
→ rev
```

```
#####
```

How To Deal With Anti-Debug and Anti-VM Technique:

Challenge – crackme6

```
→ rev ./crackme6
*****
**      rules:      **
*****  
  
* do not bruteforce
* do not patch, find instead the serial.  
  
enter the passphrase: 12345678
try again
→ rev
```

Ise hum gdb me load kr lete hai.

```
→ rev gdb ./crackme6
GNU gdb (Ubuntu 9.2-0ubuntu1-20.04) 9.2
Copyright (C) 2020 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:
<http://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"...
pwndbg: loaded 195 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./crackme6...
(No debugging symbols found in ./crackme6)
pwndbg>
```

info function krte hai.

```
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x000000000000007c8  _init
0x000000000000007f0  putchar@plt
0x00000000000000800  puts@plt
0x00000000000000810  qsort@plt
0x00000000000000820  strlen@plt
0x00000000000000830  __stack_chk_fail@plt
0x00000000000000840  printf@plt
0x00000000000000850  strcmp@plt
0x00000000000000860  tolower@plt
```

```
0x00000000000000830    __stack_chk_fail@plt
0x00000000000000840    printf@plt
0x00000000000000850    strcmp@plt
0x00000000000000860    tolower@plt
0x00000000000000870    ptrace@plt
0x00000000000000880    __isoc99_scanf@plt
0x00000000000000890    exit@plt
0x000000000000008a0    __ctype_b_loc@plt
0x000000000000008b0    __cxa_finalize@plt
0x000000000000008c0    _start
0x000000000000008f0    deregister_tm_clones
0x00000000000000930    register_tm_clones
0x00000000000000980    __do_global_dtors_aux
0x000000000000009c0    frame_dummy
0x000000000000009ca    rot
0x00000000000000b65    compare
0x00000000000000b93    generic_fizz_buzz
0x00000000000000c59    obfuscation
0x00000000000000cf4    swap_row
0x00000000000000e0a    gauss_eliminate
0x000000000000001144   cryptSerial
0x000000000000001263   checkSerial
0x000000000000001343   main
0x0000000000000015a0    __libc_csu_init
0x000000000000001610    __libc_csu_fini
0x000000000000001614    _fini
pwndbg>
```

disassemble main krte hai.

```

pwndbg> disassemble main
Dump of assembler code for function main:
0x00000000000000001343 <+0>:    push   rbp
0x00000000000000001344 <+1>:    mov    rbp,rsp
0x00000000000000001347 <+4>:    sub    rsp,0xb0
0x0000000000000000134e <+11>:   mov    DWORD PTR [rbp-0xa4],edi
0x00000000000000001354 <+17>:   mov    QWORD PTR [rbp-0xb0],rsi
0x0000000000000000135b <+24>:   mov    rax,QWORD PTR fs:0x28
0x00000000000000001364 <+33>:   mov    QWORD PTR [rbp-0x8],rax
0x00000000000000001368 <+37>:   xor    eax,eax
0x0000000000000000136a <+39>:   lea    rdi,[rip+0x46c]      # 0x17dd
0x00000000000000001371 <+46>:   call   0x800 <puts@plt>
0x00000000000000001376 <+51>:   lea    rdi,[rip+0x478]      # 0x17f5
0x0000000000000000137d <+58>:   call   0x800 <puts@plt>
0x00000000000000001382 <+63>:   lea    rdi,[rip+0x454]      # 0x17dd
0x00000000000000001389 <+70>:   call   0x800 <puts@plt>
0x0000000000000000138e <+75>:   mov    edi,0xa
0x00000000000000001393 <+80>:   call   0x7f0 <putchar@plt>
0x00000000000000001398 <+85>:   lea    rdi,[rip+0x46e]      # 0x180d
0x0000000000000000139f <+92>:   call   0x800 <puts@plt>
0x000000000000000013a4 <+97>:   lea    rdi,[rip+0x47d]      # 0x1828
0x000000000000000013ab <+104>:  call   0x800 <puts@plt>
0x000000000000000013b0 <+109>:  mov    edi,0xa
0x000000000000000013b5 <+114>:  call   0x7f0 <putchar@plt>
0x000000000000000013ba <+119>:  movabs rax,0x2073692073696854
0x000000000000000013c4 <+129>:  movabs rdx,0x657320706f742061
0x000000000000000013ce <+139>:  mov    QWORD PTR [rbp-0x30],rax
0x000000000000000013d2 <+143>:  mov    QWORD PTR [rbp-0x28],rdx
0x000000000000000013d6 <+147>:  movabs rax,0x7865742074657263
0x000000000000000013e0 <+157>:  movabs rdx,0x67617373656d2074
0x000000000000000013ea <+167>:  mov    QWORD PTR [rbp-0x20],rax
0x000000000000000013ee <+171>:  mov    QWORD PTR [rbp-0x18],rdx

```

To yha pr main kafi bda hai.

Ab hum **main** pr **breakpoint** lga lete hai.

```

pwndbg> b main
Breakpoint 1 at 0x1347
pwndbg> |

```

Aur ise **run** krte hai.

```

pwndbg> run
Starting program: /home/ubuntu/crackme6
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x0000555555401347 in main ()
Disabling the emulation via Unicorn Engine that is used for computing branches as there isn't enough memory
e also:
* https://github.com/pwndbg/pwndbg/issues/1534
* https://github.com/unicorn-engine/unicorn/pull/1743
Either free your memory or explicitly set `set emulate off` in your Pwndbg config
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA [ REGISTERS / show-flags off / show-compact-reg off ]
RAX 0x555555401343 (main) ← push rbp
RBX 0
RCX 0x5555554015a0 (__libc_csu_init) ← push r15
RDX 0xfffffffffe478 → 0x7fffffff6fd ← 'SHELL=/bin/bash'
RDI 1
RSI 0xfffffffffe468 → 0x7fffffff6e7 ← '/home/ubuntu/crackme6'
R8 0x7ffff7e1bf10 (initial+16) ← 4
R9 0x7ffff7fc9040 (_dl_fini) ← endbr64
R10 0x7ffff7fc3908 ← 0xd00120000000
R11 0x7ffff7fde660 (_dl_audit_preinit) ← endbr64
R12 0x7fffffff6e468 → 0x7fffffff6e7 ← '/home/ubuntu/crackme6'
R13 0x555555401343 (main) ← push rbp
R14 0
R15 0x7ffff7ffd040 (_rtld_global) → 0x7ffff7ffe2e0 → 0x555555400000 ← jg 0x555555400047
RBP 0xfffffffffe350 ← 1
RSP 0xfffffffffe350 ← 1
RIP 0x555555401347 (main+4) ← sub rsp, 0xb0 [ DISASM / x86-64 / set emulate off ]
-----[ DISASM / x86-64 / set emulate off ]-----
▶ 0x555555401347 <main+4>    sub   rsp, 0xb0
                                RSP => 0xfffffffffe350 - 0xb0
0x55555540134e <main+11>    mov    dword ptr [rbp - 0xa4], edi
0x555555401354 <main+17>    mov    qword ptr [rbp - 0xb0], rsi
0x55555540135b <main+24>    mov    rax, qword ptr fs:[0x28]      RAX, [0xfffff7fb0768]
0x555555401364 <main+33>    mov    qword ptr [rbp - 8], rax
0x555555401368 <main+37>    xor    eax, eax          EAX => 0
0x55555540136a <main+39>    lea    rdi, [rip + 0x46c]      RDI => 0x5555554017dd ← sub ch, byte ptr [rdx] , <puts@plt>
0x555555401371 <main+46>    call   puts@plt
                                <puts@plt>

0x555555401376 <main+51>    lea    rdi, [rip + 0x478]      RDI => 0x5555554017f5 ← sub ch, byte ptr [rdx] /* ***/
0x55555540137d <main+58>    call   puts@plt
                                <puts@plt>

0x555555401382 <main+63>    lea    rdi, [rip + 0x454]      RDI => 0x5555554017dd ← sub ch, byte ptr [rdx] /* ***** */
-----[ STACK ]-----
00:0000| rbp rsp 0xfffffffffe350 ← 1
01:0008 +008  0xfffffffffe358 → 0x7ffff7c29d90 (__libc_start_call_main+128) ← mov edi, eax
02:0010 +010  0xfffffffffe360 ← 0
03:0018 +018  0xfffffffffe368 → 0x555555401343 (main) ← push rbp
04:0020 +020  0xfffffffffe370 ← 0x1000000000
05:0028 +028  0xfffffffffe378 → 0x7fffffff6e468 → 0x7fffffff6e7 ← '/home/ubuntu/crackme6'
06:0030 +030  0xfffffffffe380 ← 0
07:0038 +038  0xfffffffffe388 ← 0x48e052766b246d88 [ BACKTRACE ]
-----[ BACKTRACE ]-----
▶ 0 0x555555401347 main+4
  1 0x7ffff7c29d90 __libc_start_call_main+128
  2 0x7ffff7c29e40 __libc_start_main+128
  3 0x5555554008ea _start+42
pwndbg> |

```

Upar hum dekh skte hai ki yha pr hume bahut sare sections milte hai. **registers** ki value mil jati with **telescoping**.

Telescoping means kaun sa chij kis chij ko point kr rha hai batat hai.

[REGISTERS]			
RAX	0x555555401343	(main)	← push rbp
RBX	0x5555554015a0	(__libc_csu_init)	← push r15
RCX	0x5555554015a0	(__libc_csu_init)	← push r15
RDX	0x7fffffffdf88	→ 0x7fffffff2eb	← 'GJS_DEBUG_TOPICS=JS_ERROR;JS_LOG'
RDI	0x1		
RSI	0x7fffffffdf78	→ 0x7fffffff2ca	← '/home/hellsender/yt/rev/crackme6'
R8	0x0		
R9	0x7ffff7fe0d50	← endbr64	I
R10	0x7		
R11	0x2		
R12	0x5555554008c0	(_start)	← xor ebp, ebp
R13	0x7fffffffdf70	← 0x1	
R14	0x0		
R15	0x0		
RBP	0x7fffffffde80	← 0x0	
RSP	0x7fffffffde80	← 0x0	
RIP	0x555555401347	(main+4)	← sub rsp, 0xb0

Jaisa ki hum upar dekh skte hai. **0x7fffffffdf88** → **0x7fffffff2eb** ko point kr rha hai. aur **0x7fffffff2eb** address pr **'GJS_DEBUG_TOPICS=JS_ERROR;JS_LOG'** hai ye bhi bta rha hai.

Ab hum ni krte next instruction pdte jate hai.

```
pwndbg> ni|
```

Agar aapne ek bar **ni** (jo bhi command) likh diya fir bs enter hit kr skte hai. ye previous command chala dega.

```
[ DISASM ]
0x555555401354 <main+17>    mov    qword ptr [rbp - 0xb0], rsi
0x55555540135b <main+24>    mov    rax, qword ptr fs:[0x28]
0x555555401364 <main+33>    mov    qword ptr [rbp - 8], rax
0x555555401368 <main+37>    xor    eax, eax
0x55555540136a <main+39>    lea    rdi, [rip + 0x46c]
► 0x555555401371 <main+46>    call   puts@plt <puts@plt>
s: 0x5555554017dd ← *****
0x555555401376 <main+51>    lea    rdi, [rip + 0x478]
0x55555540137d <main+58>    call   puts@plt <puts@plt>
0x555555401382 <main+63>    lea    rdi, [rip + 0x454]
0x555555401389 <main+70>    call   puts@plt <puts@plt>
0x55555540138e <main+75>    mov    edi, 0xa
```

pwndbg me jo bhi call ke puts function me print hone wala ho wo hume dikh data hai.

```
0x5555554013b5 <main+114>    call   putchar@plt <putchar@plt>
► 0x5555554013ba <main+119>    movabs rax, 0x2073692073696854
0x5555554013c4 <main+129>    movabs rdx, 0x657320706f742061
0x5555554013ce <main+139>    mov    qword ptr [rbp - 0x30], rax
0x5555554013d2 <main+143>    mov    qword ptr [rbp - 0x28], rdx
0x5555554013d6 <main+147>    movabs rax, 0x7865742074657263
0x5555554013e0 <main+157>    movabs rdx, 0x67617373656d2074
```

Ab yha pr ye itne bade hex values mov ho rhi hai. generally aisi chij nhi hoti. Iska mtlb ye koi string mov ho rhi hai kai sare pieces me 3-4 registers ka use krke.

Ise hum **unhex** krke dekhte hai. starting ke **0x** hta denge. Aur unhex command ko pwndbg ke sath aata hai. **vanilla gdb** me nhi aata.

```
pwndbg> unhex 2073692073696854
si sihT
pwndbg> unhex 657320706f742061
es pot a
```

To ye ulta print krta hai. little endian format me ye ulta print krta hai.

Likha hai This is a top se (secret hi hogा)

Ye latest **pwndbg** me string krke bta data hai.

```

▶ 0x5555554013ba <main+119>    movabs  rax, 0x2073692073696854      RAX => 0x2073692073696854 ('This is ')
0x5555554013c4 <main+129>    movabs  rdx, 0x657320706f742061      RDX => 0x657320706f742061 ('a top se')
0x5555554013ce <main+139>    mov     qword ptr [rbp - 0x30], rax
0x5555554013d2 <main+143>    mov     qword ptr [rbp - 0x28], rdx
0x5555554013d6 <main+147>    movabs  rax, 0x7865742074657263      RAX => 0x7865742074657263 ('cret tex')
0x5555554013e0 <main+157>    movabs  rdx, 0x67617373656d2074      RDX => 0x67617373656d2074 ('t messag')
[ STACK ]

```

To yha pr ye phle **rax** ke andar mov kr rha hai fir **rdx** ke andar mov kr rha hai fir ye in dono ko **Stack** ke andar mov kr rha hai.

Ab yha **Stack** kya hai. to **Stack** ek location hai hum apne temparery chijo ko store kr skte hai. **Registers** bhi storage hi hai but ye bahut jyada chote hai. jb humari program run ho rha hota hai to wo currently kahan pr store hota hai. to ye **RAM** ke andar store hote hai.

To **RAM** ke andar bhi alag-2 location hote hai jha pr humari chije store hoti hai. like **Stack** hota hai **Heap** hota hai etc.

To **Stack** ke andar hum un chijo ko store kr skte hai jinka use hume bad me hai. **Registers** ke andar wo chije aati hai jinka use currently hai ya hone wala hai ya ho rha hai.

Stack bhi bahut jayada limited hai. isme bhi bahut badi value nhi rakh skte hai isi ke jaisi ek memory location **Heap** hoti hai wo stack ke mukawale bahut badi hoti hai.

[DISASM]		
0x5555554013ab <main+104>	call	puts@plt <puts@plt>
0x5555554013b0 <main+109>	mov	edi, 0xa
0x5555554013b5 <main+114>	call	putchar@plt <putchar@plt>
0x5555554013ba <main+119>	movabs	rax, 0x2073692073696854
0x5555554013c4 <main+129>	movabs	rdx, 0x657320706f742061
▶ 0x5555554013ce <main+139>	mov	qword ptr [rbp - 0x30], rax
0x5555554013d2 <main+143>	mov	qword ptr [rbp - 0x28], rdx
0x5555554013d6 <main+147>	movabs	rax, 0x7865742074657263
0x5555554013e0 <main+157>	movabs	rdx, 0x67617373656d2074
0x5555554013ea <main+167>	mov	qword ptr [rbp - 0x20], rax
0x5555554013ee <main+171>	mov	qword ptr [rbp - 0x18], rdx
[STACK]		
00:0000 rsp 0xffffffffffffd0 → 0xfffffffffdf78 → 0xfffffffffe2ca ← '/home/hellse		
01:0008 0x7fffffffdd8 ← 0x100000000		I
02:0010 0x7fffffffddde0 ← 0x0		
... ↓	5 skipped	

To yha hum bdi value ko **Stack** ke andar mov kr denge.

```

0x5555554013ee <main+171>    mov    qword ptr [rbp - 0x18], rdx
0x5555554013f2 <main+175>    mov    word ptr [rbp - 0x10], 0x2165
0x5555554013f8 <main+181>    mov    byte ptr [rbp - 0xe], 0
0x5555554013fc <main+185>    sidt   [rbp - 0x96]
0x555555401403 <main+192>    movzx  eax, byte ptr [rbp - 0x91]
► 0x55555540140a <main+199>    cmp    al, 0xff
0x55555540140c <main+201>    jne    main+225 <main+225>
↓
0x555555401424 <main+225>    lea    rax, [rbp - 0x30]
0x555555401428 <main+229>    mov    rsi, rax
0x55555540142b <main+232>    mov    edi, 0xd
0x555555401430 <main+237>    call   rot <rot>

```

Ab yha pr ye compare kr rha hai. but ye equal nhi hai. jaisa ki hum jante hai **rax** ki last **8 bits** al hoti hai.

To hum register ki value dekhe to.

```

*RAX 0x0
RBX 0x5555554015a0 (_libc_csu_init) ← push r15
RCX 0x7ffff7eca1e7 (write+23) ← cmp rax, -0x10
RDX 0x67617373656d2074 ('t messag')
RDI 0x7ffff7fa74c0 (_IO_stdfile_1_lock) ← 0x0
RSI 0x5555556032a0 ← '\n do not patch, find inste

```

Yha rax ki value **0** hai. next line dekhte hai.

```

0x555555401403 <main+192>    movzx  eax, byte ptr [rbp - 0x91]
0x55555540140a <main+199>    cmp    al, 0xff
► 0x55555540140c <main+201> ✓ jne    main+225 <main+225>
↓
0x555555401424 <main+225>    lea    rax, [rbp - 0x30]
0x555555401428 <main+229>    mov    rsi, rax
0x55555540142b <main+232>    mov    edi, 0xd
0x555555401430 <main+237>    call   rot <rot>

```

To ye equal nhi hoga fir jump karege jne means jump if not equal. Aur hum dekh skte hai **jne** pr right ka tick bn kr bhi aa gya.

Man lo agar hume dekhna hai ki ye jump nhi leta to kya hota.

```

pwndbg> nearpc
0x5555554013f2 <main+175>    mov    word ptr [rbp - 0x10], 0x2165
0x5555554013f8 <main+181>    mov    byte ptr [rbp - 0xe], 0
0x5555554013fc <main+185>    sidt   [rbp - 0x96]
0x555555401403 <main+192>    movzx  eax, byte ptr [rbp - 0x91]
0x55555540140a <main+199>    cmp    al, 0xff
► 0x55555540140c <main+201> ✓ jne    main+225 <main+225>
0x55555540140e <main+203>    lea    rdi, [rip + 0x43c]
0x555555401415 <main+210>    call   puts@plt <puts@plt>
0x55555540141a <main+215>    mov    edi, 1
0x55555540141f <main+220>    call   exit@plt <exit@plt>
0x555555401424 <main+225>    lea    rax, [rbp - 0x30]
pwndbg>

```

To kuch rdi me put hota hai. iska mtlb hume jump nhi leni chahiye thi.

Aage **ni** krte hai next instruction pr chalte hai.

```

0x555555401428 <main+229>    mov    rsi, rax
0x55555540142b <main+232>    mov    edi, 0xd
► 0x555555401430 <main+237> call   rot <rot>
    rdi: 0xd
    rsi: 0x7fffffffde50 ← 'This is a top secret text message!' I
    rdx: 0x67617373656d2074 ('t messag')
    rcx: 0x7ffff7eca1e7 (write+23) ← cmp    rax, -0x1000 /* 'H=' */

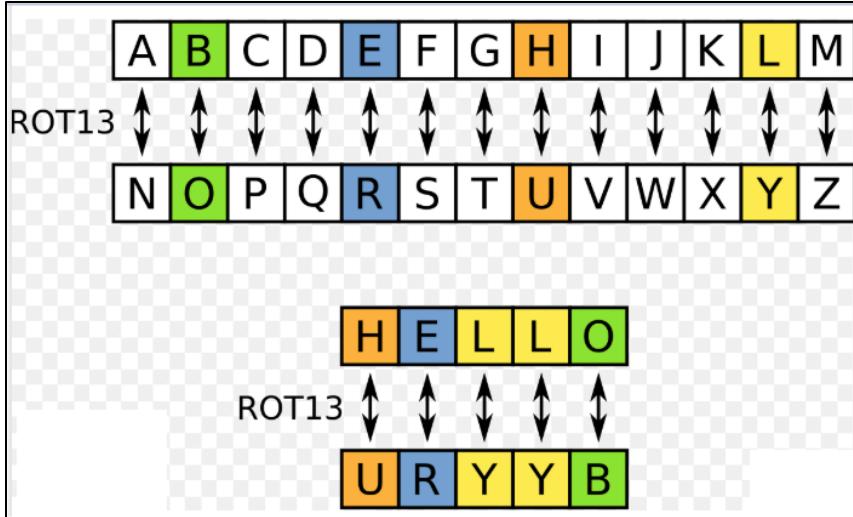
0x555555401435 <main+242>    lea    rax, [rbp - 0x30]
0x555555401439 <main+246>    mov    rsi, rax
0x55555540143c <main+249>    mov    edi, 0xd
0x555555401441 <main+254>    call   rot <rot>

0x555555401446 <main+259>    movabs rax, 0x4145443332694841
[ STACK ]
00:0000| rsp 0x7fffffffddd0 → 0x7fffffffdf78 → 0x7fffffffde2ca ← '/home/hellse
01:0008|          0x7fffffffddd8 ← 0x100000000

```

To yha pr **rot** function ko call ho rhi hai. **rot** function kya krte hai rotate krte hai.

Something like this.



But ye function abhi humare kam ka nhi lg rha hai.

Aage ni krte hai.

```

0x55555401439 <main+246>    mov    rsi, rax
0x5555540143c <main+249>    mov    edi, 0xd
► 0x55555401441 <main+254>    call   rot <rot>
    rdi: 0xd
    rsi: 0x7fffffffde50 ← 'Guvf vf n gbc frperg grkg zrffntr!'
    rdx: 0x42
    rcx: 0x21

0x55555401446 <main+259>    movabs rax, 0x4145443332694841
0x55555401450 <main+269>    movabs rdx, 0x464f434645454244
0x5555540145a <main+279>    mov    qword ptr [rbp - 0x90], rax
0x55555401461 <main+286>    mov    qword ptr [rbp - 0x88], rdx
0x55555401468 <main+293>    mov    word ptr [rbp - 0x80], 0x4546

```

Yha fir ye ek aur **string** ko **rotate** kr rha hai.

Aage ni krte hai.

```

► 0x55555401446 <main+259>    movabs rax, 0x4145443332694841
0x55555401450 <main+269>    movabs rdx, 0x464f434645454244
0x5555540145a <main+279>    mov    qword ptr [rbp - 0x90], rax
0x55555401461 <main+286>    mov    qword ptr [rbp - 0x88], rdx
0x55555401468 <main+293>    mov    word ptr [rbp - 0x80], 0x4546
0x5555540146e <main+299>    mov    byte ptr [rbp - 0x7e], 0x45

```

[STACK]

Yha fir se ye ek aur string ko **rax** ke andar put krta aur dusre string ko **rdx** ke andar fir ye ise **Stack** me mov kr rha hai.

```
0x555555401479 <main+310>    mov    eax, 0
► 0x55555540147e <main+315>    call   printf@plt <printf@plt>
format: 0x555555401861 ← 'enter the passphrase: '
vararg: 0x7fffffffde50 ← 'This is a top secret text message!'

0x555555401483 <main+320>    lea    rax, [rbp - 0x70]
0x555555401487 <main+324>    mov    rsi, rax
0x55555540148a <main+327>    lea    rdi, [rip + 0x1e5]
0x555555401491 <main+334>    mov    eax, 0
0x555555401496 <main+339>    call   __isoc99_scanf@plt <__isoc99_scanf@plt>
```

Yha printf ko call kr rha hai enter theh passphase:

Mtlb ye inpur lete hai kuchh instructions ke bad.

```
0x555555401491 <main+334>    mov    eax, 0
► 0x555555401496 <main+339>    call   __isoc99_scanf@plt <__isoc99_scanf@plt>
format: 0x555555401676 ← 0x7542006425007325 /* '%s' */
vararg: 0x7fffffffde10 ← 0x0

0x55555540149b <main+344>    mov    esi, 0
0x5555554014a0 <main+349>    mov    edi, 0
0x5555554014a5 <main+354>    mov    eax, 0
0x5555554014aa <main+359>    call   ptrace@plt <ptrace@plt>
```

Ab ye string type ka input lega **scanf** functin ke through kyoki yha **%s** diya hai.

```
pwndbg> ni
enter the passphrase: asdf1244
```

Yha pr humne input dal diya.

Aur next instruction pr chalte hai.

```

0x5555554014a5 <main+354>    mov    eax, 0
► 0x5555554014aa <main+359>    call   ptrace@plt <ptrace@plt>
    request: 0x0
    vararg: 0x0

0x5555554014af <main+364>    test   rax, rax
0x5555554014b2 <main+367>    jns    main+391 <main+391>

0x5555554014b4 <main+369>    lea    rdi, [rip + 0x3bd]
0x5555554014bb <main+376>    call   puts@plt <puts@plt>

```

Aur ye yha **ptrace** function ko call kr rha hai jo ki bahut jyada interesting hai.

Next krte hai.

```

0x5555554014a5 <main+354>    mov    eax, 0
0x5555554014aa <main+359>    call   ptrace@plt <ptrace@plt>

► 0x5555554014af <main+364>    test   rax, rax
0x5555554014b2 <main+367>    jns    main+391 <main+391>

0x5555554014b4 <main+369>    lea    rdi, [rip + 0x3bd]
0x5555554014bb <main+376>    call   puts@plt <puts@plt>

0x5555554014c0 <main+381>    mov    edi, 1

```

Yha pr ye check kr rha hai ki rax ki value 0 hai.

[REGISTERS]		
*RAX	0xffffffffffffffffffff	
RBX	0x5555554015a0 (<u>libc_csu_init</u>)	← push r15
RCX	0x0	
*RDX	0xfffffffffffffff80	
RDI	0x0	
RSI	0x0	
*R8	0xffffffff	
R9	0x7c	
*R10	0x0	
*R11	0x286	
R12	0x5555554008c0 (<u>_start</u>)	← xor ebp, ebp
R13	0x7fffffffdf70	← 0x1
R14	0x0	
R15	0x0	
RBP	0x7fffffffde80	← 0x0
RSP	0x7fffffffdd0	→ 0x7fffffffdf78 → 0x7fffffff2ca ← '/home/he...
*RIP	0x5555554014af (main+364)	← test rax, rax

[DISASM]

To hum rax me dekhe to hex ka sbse bda number hai. theek opposite.

Ye wali situation true nhi hai.

```
0x5555554014af <main+364>    test   rax, rax
► 0x5555554014b2 <main+367>  jns    main+391 <main+391>

0x5555554014b4 <main+369>    lea    rdi, [rip + 0x3bd]
0x5555554014bb <main+376>    call   puts@plt <puts@plt>

0x5555554014c0 <main+381>    mov    edi, 1
0x5555554014c5 <main+386>    call   exit@plt <exit@plt>

0x5555554014ca <main+391>    movzx  eax, byte ptr [rbp - 0x90]
                                [ STACK ]
```

Yha pr ye check kr rha hai ki **sign flag** ki value **zero** hai to jump kare otherwise jump mt karo.

jns means jump if sign flag value is **zero**.

Agar hum registers dekhe to.

```
pwndbg> info registers
rax          0xfffffffffffffff -1
rbx          0x5555554015a0  93824990844320
rcx          0x0              0
rdx          0xfffffffffffffff80 -128
rsi          0x0              0
rdi          0x0              0
rbp          0x7fffffffde80  0x7fffffffde80
rsp          0x7fffffffddd0  0x7fffffffddd0
r8           0xfffffff        4294967295
r9           0x7c             124
r10          0x0              0
r11          0x286            646
r12          0x5555554008c0  93824990841024
r13          0x7fffffffdf70  140737488346992
r14          0x0              0
r15          0x0              0
rip          0x5555554014b2  0x5555554014b2 <main+367>
eflags       0x286            [PF SF IF ]
cs           0x33             51
ss           0x2b             43
ds           0x0              0
es           0x0              0
fs           0x0              0
gs           0x0              0
```

To hum yha dekh skte hai jo bhi flag likh kr aate hai unki value **1** hoti hai. aur jo nhi likh kr aate unki value **zero** hoti hai.

Iska mtlb **SF (sign flag)** ki value **1** hai.

To ye **jump** nhi karega. Agar ye jump nhi karega to kya hoga. Ye kisi chij ko **puts** karega aur **exit** kr jayega.

```
0x5555554014af <main+364>    test   rax, rax
► 0x5555554014b2 <main+367>    jns    main+391 <main+391>
0x5555554014b4 <main+369>    lea    rdi, [rip + 0x3bd]
0x5555554014bb <main+376>    call   puts@plt <puts@plt>
0x5555554014c0 <main+381>    mov    edi, 1
0x5555554014c5 <main+386>    call   exit@plt <exit@plt>
```

Next instruction pr jate hai.

```
0x5555554014af <main+364>    test   rax, rax
0x5555554014b2 <main+367>    jns    main+391 <main+391>
0x5555554014b4 <main+369>    lea    rdi, [rip + 0x3bd]
► 0x5555554014bb <main+376>    call   puts@plt <puts@plt>
s: 0x555555401878 ← 'This process is being debugged!!!'
0x5555554014c0 <main+381>    mov    edi, 1
0x5555554014c5 <main+386>    call   exit@plt <exit@plt>
0x5555554014ca <main+391>    movzx  eax, byte ptr [rbp - 0x90]
```

Ab hum yha pr dekh skte hai. ki isne detect kr li ki **debugging** ho rhi hai. but jb humne normal run kiya tha to iska try again ka message print kiya tha.

To ye **anti debug** technique hai. bahut sare malware ye techniques use krte hai taki koi unka malware reverse nhi kr paye.

Bahut sare malware writer ye techniques use krte hai taki unka malware reverse na ho paye.

To yha pr protection lagayi gyi hai lekin usko bypass krne ke techniques bhi honge.

Ab hum dekhte hai ki use pta kaise chala ki hum **debugger** use kr rhe hai.

Yha isne **ptrace (process trace)** call kr rha hai. to ptrace ek achha **syscall** hai jo hume batata hai ki.

```
gdb ./crackme6 116x29
0x5555554014aa <main+359>    call  ptrace@plt <ptrace@plt>
0x5555554014af <main+364>    test  rax, rax
0x5555554014b2 <main+367>    jns   main+391 <main+391>
0x5555554014b4 <main+369>    lea   rdi, [rip + 0x3bd]
► 0x5555554014bb <main+376>    call  puts@plt <puts@plt>
s: 0x555555401878 ← 'This process is being debugged!!!' ┌
0x5555554014c0 <main+381>    mov   edi, 1
0x5555554014c5 <main+386>    call  exit@plt <exit@plt>
0x5555554014ca <main+391>    movzx eax, byte ptr [rbp - 0x90]
0x5555554014d1 <main+398>    xor   eax, 2
0x5555554014d4 <main+401>    mov   byte ptr [rbp - 0x50], al
[ STACK ]
```

Koi ise **trace** to nhi kr rha hai. **trace** means koi ise step by step run to nhi kr rha hai. koi uski memory padne ya registers value dekhne ki koshish to nhi kr rha.

Isi se ise pta chla ki ise koi **debug** kr rha hai.

To **ptrace** ko bypass krne ke two ways hai.

Ab hum ptrace pr breakpoint lagate hai.

```
pwndbg> b *0x5555554014aa
Breakpoint 2 at 0x5555554014aa
pwndbg>
```

Aur run krte hai.

```
pwndbg> run
Starting program: /home/hellsender/yt/rev/crackme6
```

Ab ye main pr ruk gya ise **c (continue)** krte hai.

```

pwndbg> c
Continuing.
*****
**      rules:      **
*****
* do not bruteforce
* do not patch, find instead the serial.

enter the passphrase: 12345

```

Passphrase dal dete hai.

```

0x5555554014a5 <main+354>    mov    eax, 0
0x5555554014aa <main+359>    call   ptrace@plt <ptrace@plt>

► 0x5555554014af <main+364>    test   rax, rax
0x5555554014b2 <main+367>    jns    main+391 <main+391> []

0x5555554014b4 <main+369>    lea    rdi, [rip + 0x3bd]
0x5555554014bb <main+376>    call   puts@plt <puts@plt>

0x5555554014c0 <main+381>    mov    edi, 1
0x5555554014c5 <main+386>    call   exit@plt <exit@plt>

```

Yha pr **rax** ki value agar **0** hai to **debugger** nhi run ho rha hai. lekin **0** ke awala koi aur value hai jaise **1** to **debugger** run ho rha hai.

To hume isko bypass krne ke liye rax ki value **0** krni hogi.

[REGISTERS]	
*RAX	0xffffffffffffffffffff
RBX	0x5555554015a0 (<u>libc_csu_init</u>) ← push r15
RCX	0x0
*RDX	0xffffffffffffffffffff80
RDI	0x0
RSI	0x0
*R8	0xffffffffffff
R9	0x7c
*R10	0x0
*R11	0x286
R12	0x5555554008c0 (_start) ← xor ebp, ebp
R13	0x7fffffffdf70 ← 0x1

To yha pr hum **rax** ki value **0** set kr dete hai.

```
pwndbg> set $rax = 0
```

Next instruction pr chalte hai.

```
[ DISASM ]  
0x5555540149b <main+344>    mov    esi, 0  
0x555554014a0 <main+349>    mov    edi, 0  
0x555554014a5 <main+354>    mov    eax, 0  
0x555554014aa <main+359>    call   ptrace@plt <ptrace@plt>  
  
0x555554014af <main+364>    test   rax, rax  
► 0x555554014b2 <main+367> ✓ Ijns  main+391 <main+391>  
↓  
0x555554014ca <main+391>    movzx  eax, byte ptr [rbp - 0x90]  
0x555554014d1 <main+398>    xor    eax, 2  
0x555554014d4 <main+401>    mov    byte ptr [rbp - 0x50], al  
0x555554014d7 <main+404>    movzx  eax, byte ptr [rbp - 0x8d]  
0x555554014de <main+411>    sub    eax, 0xa  
[ STACK ]
```

To yha pr ye **jump** le rha hai.

```

pwndbg> info registers
rax          0x0          0
rbx          0x5555554015a0    93824990844320
rcx          0x0          0
rdx          0xfffffffffffff80 -128
rsi          0x0          0
rdi          0x0          0
rbp          0x7fffffffde80  0x7fffffffde80
rsp          0x7fffffffddd0  0x7fffffffddd0
r8           0xffffffff     4294967295
r9           0x7c          124
r10          0x0          0
r11          0x286         646
r12          0x5555554008c0  93824990841024
r13          0x7fffffffdf70  140737488346992
r14          0x0          0
r15          0x0          0
rip          0x5555554014b2  <main+367>
eflags        0x246        [ PF ZF IF I ]
cs            0x33         51
ss            0x2b         43
ds            0x0          0
es            0x0          0
fs            0x0          0
gs            0x0          0
pwndbg>

```

To yha hum dekh skte hai ki **SF** nhi set hai mtlb uski value **zero** hai. isliye show nhi kr rha hai.

Ab hum next instruction pr chlte hai.

<pre> 0x5555554014aa <main+359> 0x5555554014af <main+364> 0x5555554014b2 <main+367> ↓ ► 0x5555554014ca <main+391> 0x5555554014d1 <main+398> 0x5555554014d4 <main+401> 0x5555554014d7 <main+404> </pre>	<pre> call ptrace@plt <ptrace@plt> test rax, rax jns main+391 <main+391> movzx eax, byte ptr [rbp - 0x90] xor eax, 2 mov byte ptr [rbp - 0x50], al movzx eax, byte ptr [rbp - 0x8d] </pre>
--	--

Next instructions pr chalte rhete hai.

```
0x555554014e1 <main+414>    mov    byte ptr [rbp - 0x4f], al
0x555554014e4 <main+417>    movzx eax, byte ptr [rbp - 0x8e]
0x555554014eb <main+424>    add    eax, 0xc
0x555554014ee <main+427>    mov    byte ptr [rbp - 0x4e], al
► 0x555554014f1 <main+430>    movzx eax, byte ptr [rbp - 0x8e]
0x555554014f8 <main+437>    mov    byte ptr [rbp - 0x4d], al
0x555554014fb <main+440>    movzx eax, byte ptr [rbp - 0x8f]
0x55555401502 <main+447>    add    eax, 1
0x55555401505 <main+450>    mov    byte ptr [rbp - 0x4c], al
```

To yha pr assembly me samajhne me der hogi to hum decompiler ka use kr lete hai.

Yha hum IDA ka use karenge.

```
→ idafree-7.6 ./ida64 ~/yt/rev/crackme6
```

Yha pr binary load ho chuki hai.

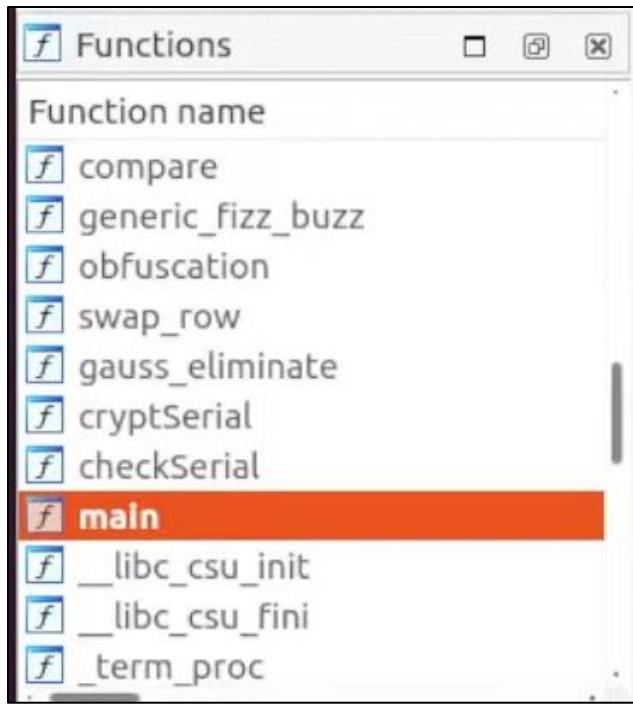
The screenshot shows the IDA Pro debugger interface. The top menu bar includes File, Edit, Jump, Search, View, Debugger, Options, Windows, and Help. Below the menu is a toolbar with various icons. The status bar at the bottom indicates "Local Linux de". A legend at the top right defines symbols: Library function (light blue), Regular function (blue), Instruction (brown), Data (grey), Unexplored (yellow-green), External symbol (pink), and Lumina function (green). The main window has tabs for Functions, IDA View-A (selected), Hex View-1, Structures, Enums, and others. The Functions tab lists several standard C library functions like _tolower, _ptrace, __isoc99_scanf, _exit, and __ctype_b_loc. The IDA View-A tab displays assembly code for the main function:

```
; Attributes: bp-based frame
; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

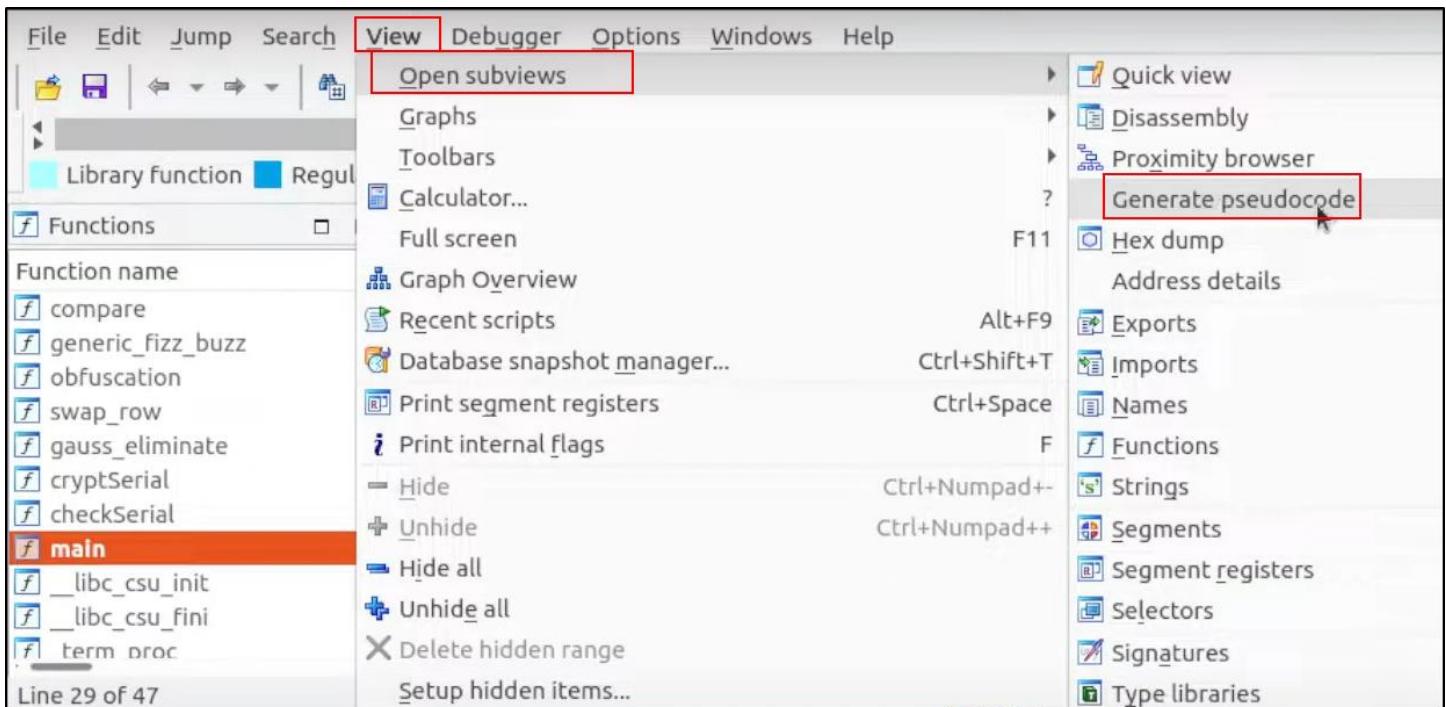
var_B0= qword ptr -0B0h
var_A4= dword ptr -0A4h
var_9C= dword ptr -9Ch
var_96= byte ptr -96h
var_90= qword ptr -90h
var_88= qword ptr -88h
var_80= word ptr -80h
var_7E= byte ptr -7Eh
s2= byte ptr -70h
s1= byte ptr -50h
var_4F= byte ptr -4Fh
var_4E= byte ptr -4Eh
var_4D= byte ptr -4Dh
var_4C= byte ptr -4Ch
var_30= qword ptr -30h
var_28= qword ptr -28h
var_20= qword ptr -20h
var_18= qword ptr -18h
var_10= word ptr -10h
var_E= byte ptr -0Eh
var_8= dword ptr -8
```

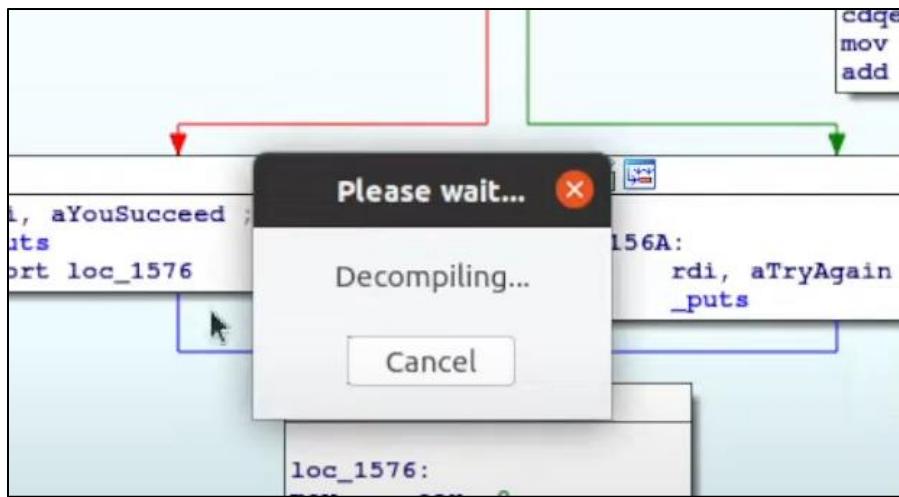
The assembly code is annotated with variable names and types. The bottom status bar shows "100.00% (-28, 32) (22, 211) 000001343 00000000000001343: main (Synchronize)".

Yha sbse phle **main** function pr aa jana hai.



Ab decompiler ke liye **f5** bhi press kr skte hai.





Ye thoda time lega aur psudocode generate kr dega.

```

1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int i; // [rsp+14h] [rbp-9Ch]
4     char v5[6]; // [rsp+1Ah] [rbp-96h] BYREF
5     __int64 v6[4]; // [rsp+20h] [rbp-90h] BYREF
6     char s2[32]; // [rsp+40h] [rbp-70h] BYREF
7     char s1[32]; // [rsp+60h] [rbp-50h] BYREF
8     char v9[40]; // [rsp+80h] [rbp-30h] BYREF
9     unsigned __int64 v10; // [rsp+A8h] [rbp-8h]
10
11    v10 = __readfsqword(0x28u);
12    puts("*****");
13    puts("**      rules:      **");
14    puts("*****");
15    putchar(10);
16    puts("* do not bruteforce");
17    puts("* do not patch, find instead the serial.");
18    putchar(10);
19    strcpy(v9, "This is a top secret text message!");
20    __sidt(v5);
21    if ( v5[5] == -1 )
22    {
23        puts("VMware detected");
24        exit(1);
25    }
26    rot(13LL, v9);
27    rot(13LL, v9);

```

Let's analyze it.

```
if ( v5[5] == -1 )
{
    puts("VMware detected");
    exit(1);
}
```

Yha pr koi array hai. jiska 6th value agar -1 hota hai to ye "**vmware detected**" show krke **exit** ho jayega.

To yha hum **vmware** pr run kare ya host machine pr run kare kya fark padta hai. lekin agar ye **malware** hua to **malware analyst** ka kam kitna hard ho jayega.

Kyoki wo host machine nhi kharab karega aur vmware pr run kar nhi skta. To wo fs jata hai. to ise hume bypass krna pdta hai.

```
putchar(10);
puts("* do not bruteforce");
puts("* do not patch, find instead the serial.");
putchar(10);
strcpy(v9, "This is a top secret text message!");
__sidt(v5);
if ( v5[5] == -1 )
{
    puts("VMware detected");
    exit(1);
}
rot(13LL, v9);
rot(13LL, v9);
qmemcpy(v6, "AH123DEADBEEFCOFFEE", 19);
printf("enter the passphrase: ");
__isoc99_scanf("%s", s2);
if ( ptrace(PTRACE_TRACEME, 0LL) < 0 )
{
    const char[]
    puts("This process is being debugged!!!");
    exit(1);
}
```

Humne ek chij nhi dekha upar ek message diya tha "this is a top secret text message!"

Aur ise **v9** variable me dal kr do bar **rot** function se **rotate** krwya ja rha hai. jiska kuchh mtlb nhi hai. bs distract krne ke liye hai. ise rabbit holes khte hai.

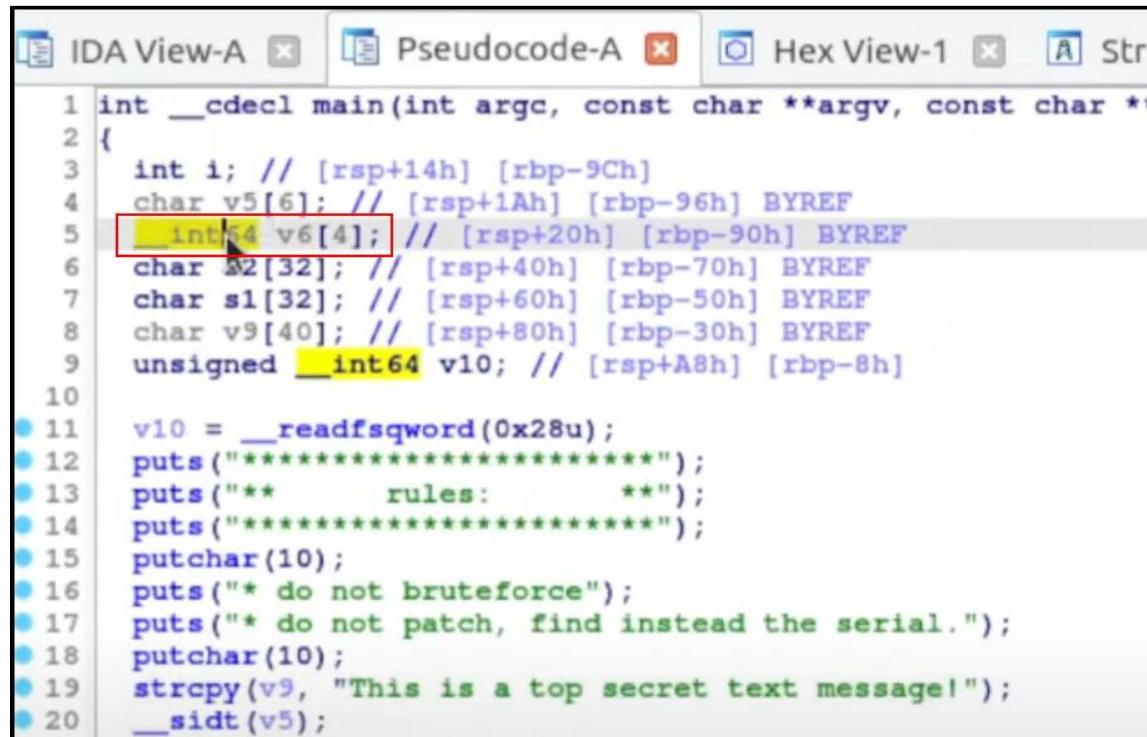
Rabbit hole ka mtlb aapko confuse kare aur gol-2 ghumaye. Ye analyst ko confuse krne ke liye diya jata hai taki wo kahin aur bhatak jaye.

Man lijiye hume knowladge na hoti to hum rot ke andar chale jate aur kafi sari calculation hoti aur usi me fase rh jate.

Yha code ko aage dekhe to

```
    rot(13LL, v9);
    rot(13LL, v9);
    qmemcpy(v6, "AH123DEADBEEFCOFFEE", 19);
    printf("enter the passphrase: ");
    __isoc99_scanf("%s", s2);
    if ( ptrace(PTRACE_TRACE_ME, 0LL) < 0 )
{
```

Yha pr **v6** me ek **string** ja rha hai.



```
1 int __cdecl main(int argc, const char **argv, const char **
2 {
3     int i; // [rsp+14h] [rbp-9Ch]
4     char v5[6]; // [rsp+1Ah] [rbp-96h] BYREF
5     int64 v6[4]; // [rsp+20h] [rbp-90h] BYREF
6     char s2[32]; // [rsp+40h] [rbp-70h] BYREF
7     char s1[32]; // [rsp+60h] [rbp-50h] BYREF
8     char v9[40]; // [rsp+80h] [rbp-30h] BYREF
9     unsigned __int64 v10; // [rsp+A8h] [rbp-8h]
10
11     v10 = __readfsqword(0x28u);
12     puts("*****");
13     puts("**      rules:      **");
14     puts("*****");
15     putchar(10);
16     puts("* do not bruteforce");
17     puts("* do not patch, find instead the serial.");
18     putchar(10);
19     strcpy(v9, "This is a top secret text message!");
20     __sidt(v5);
```

Jaisa ki hum dekh skte hai yha **v6** me ek **string** ja rhi hai aur **IDA** ise **int64** smajh rha hai.

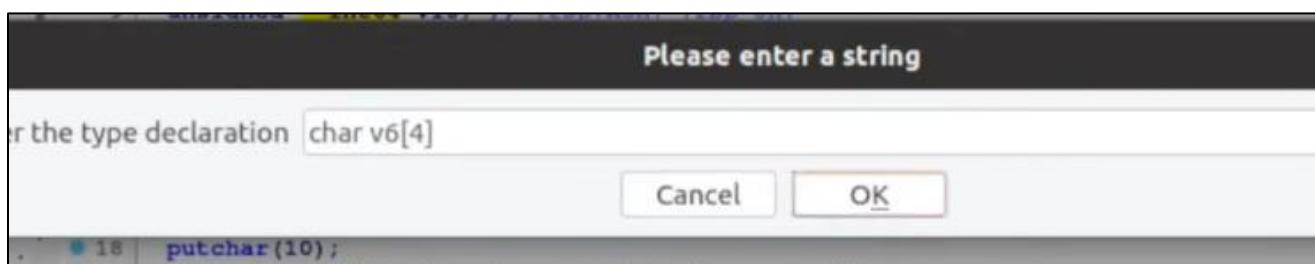
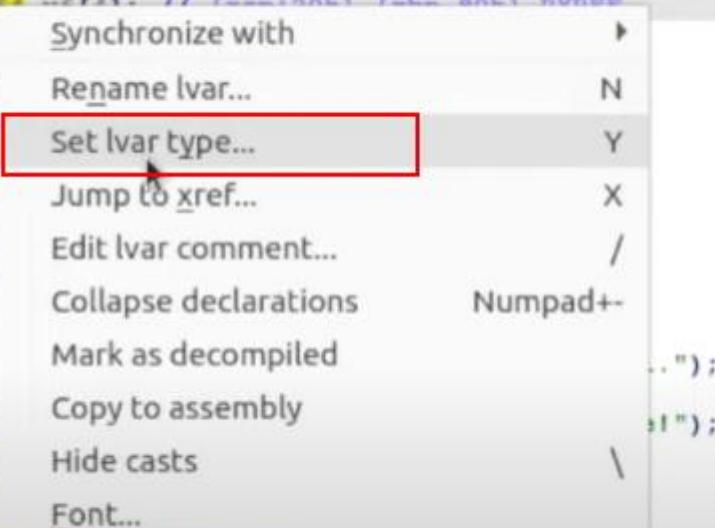
Iska hum type change krke char kr denge.

```

int __cdecl main(int argc, const char **argv, const char **env)
{
    int i; // [rsp+14h] [rbp-9Ch]
    char v5[6]; // [rsp+1Ah] [rbp-96h] BYREF
    int v6; // [rsp+20h] [rbp-90h] BYREF
    char s2[32]; // [rsp+40h] [rbp-70h] BYREF
    char s1[32]; // [rsp+60h] [rbp-50h] BYREF
    char v9[40]; // [rsp+80h] [rbp-30h] BYREF
    unsigned __int64 v10; // [rsp+A8h] [rbp-8h]

    v10 = __readfsqword(0x28u);
    puts("*****");
    puts("**      rules:      **");
    puts("*****");
    putchar(10);
}

```



Ab ye hume warning de skta hai. ki aapka output galat ho skta hai.

```

IDB View-A Pseudocode-A Hex View-1 Struct
1 // local variable allocation has failed, the output may be wr
2 int __cdecl main(int argc, const char **argv, const char **env)
3 {
4     int i; // [rsp+14h] [rbp-9Ch]
5     char v5[6]; // [rsp+1Ah] [rbp-96h] BYREF
6     char v6[4]; // [rsp+20h] [rbp-90h] OVERLAPPED BYREF
7     char s2[32]; // [rsp+40h] [rbp-70h] BYREF
8     char s1[32]; // [rsp+60h] [rbp-50h] BYREF
9     char v9[40]; // [rsp+80h] [rbp-30h] BYREF
10    unsigned __int64 v10; // [rsp+A8h] [rbp-8h]
11
12    v10 = __readfsqword(0x28u);
13    puts("*****");
14    puts("**      rules:      **");
15    puts("*****");
16    putchar(10);
}

```

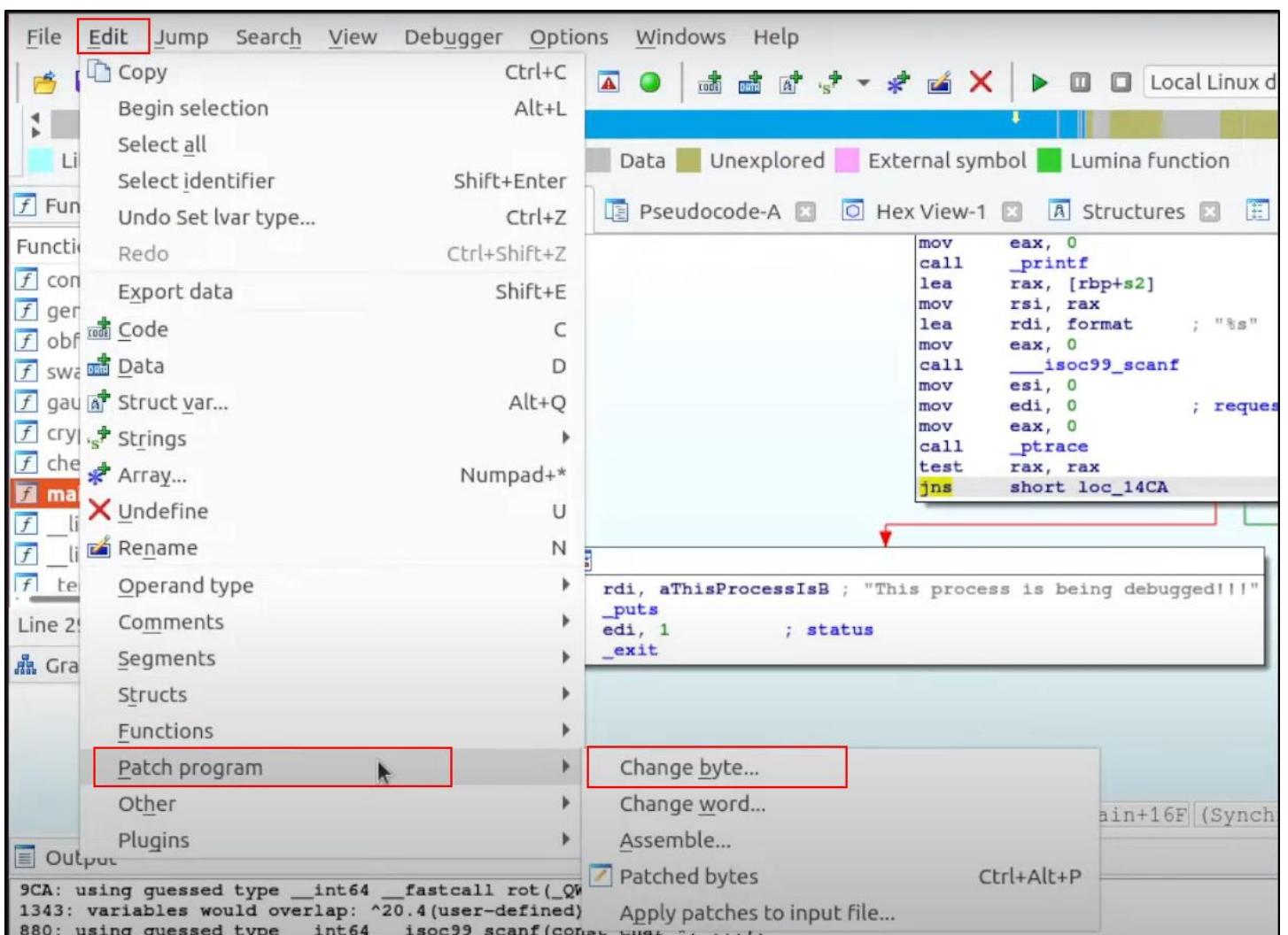
Ab **anti debug** technique ko bypass krne ke liye **binary ninja** se **patch** bna lete hai phle **rax** ke value jo change kiye the wo temporary tha.

Ab hum **disassembly** pr fir se aa jate hai.

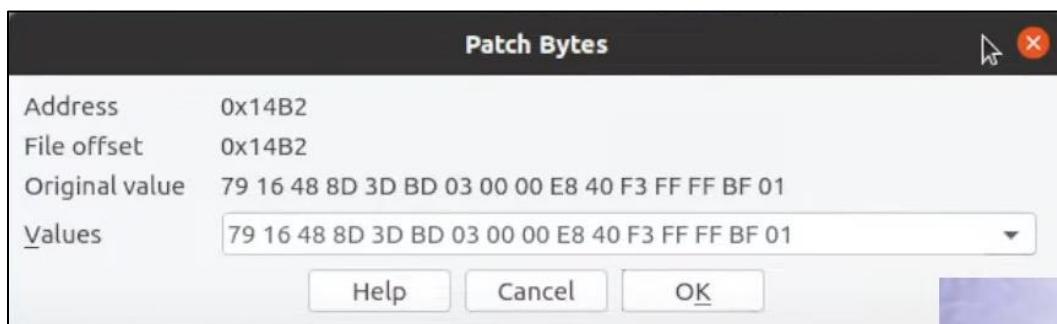
The screenshot shows the IDA Pro interface with the 'IDA View-A' tab selected (highlighted by a red box). The assembly code for the **main** function is displayed. A specific instruction, **jns**, is highlighted with a red box. A callout box below it shows the assembly code for a printf statement:

```
lea    rdi, aThisProcessIsB ; "This process is being debugged!!!"
call  _puts
mov   edi, 1      ; status
call  _exit
```

jns ko change krne ke liye hum **edit** pr jate hai. **patch program** pr jayenge. **Change byte** pr click karenge.



Yha pr aa gye opcode.



Lekin hum yha nhi karenge. Yha jyada calculation krni pdti hai.

```
hellsender@he ~ % binaryninja ./binaryninja ~/yt/rev/crackme6
```

Ab hum apni binary ko load karenge. Aur **main** function pr chlte hai.

The screenshot shows the Immunity Debugger interface with the file "crackme6 (ELF Linear)" loaded. The assembly view on the right shows the following pseudocode:

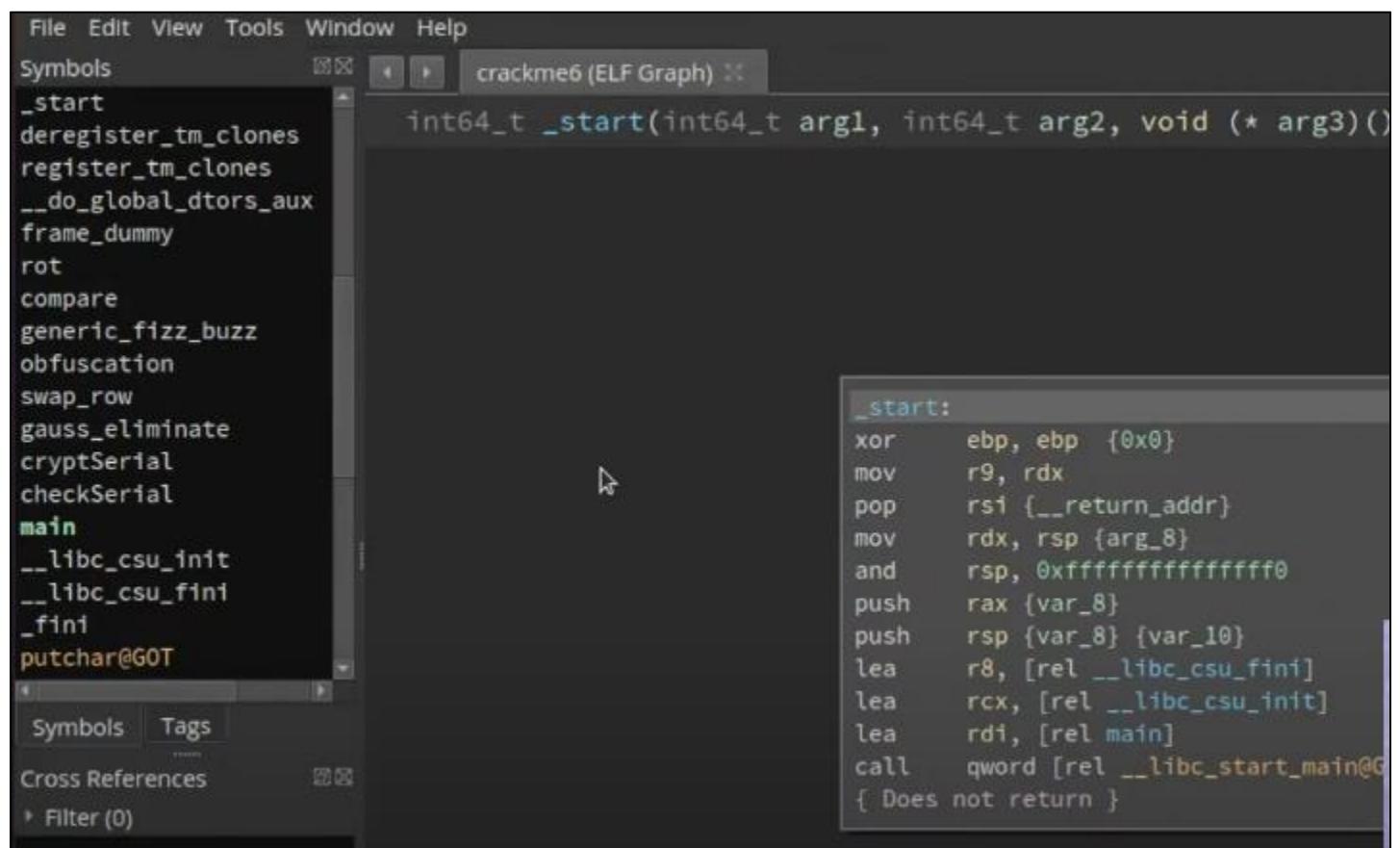
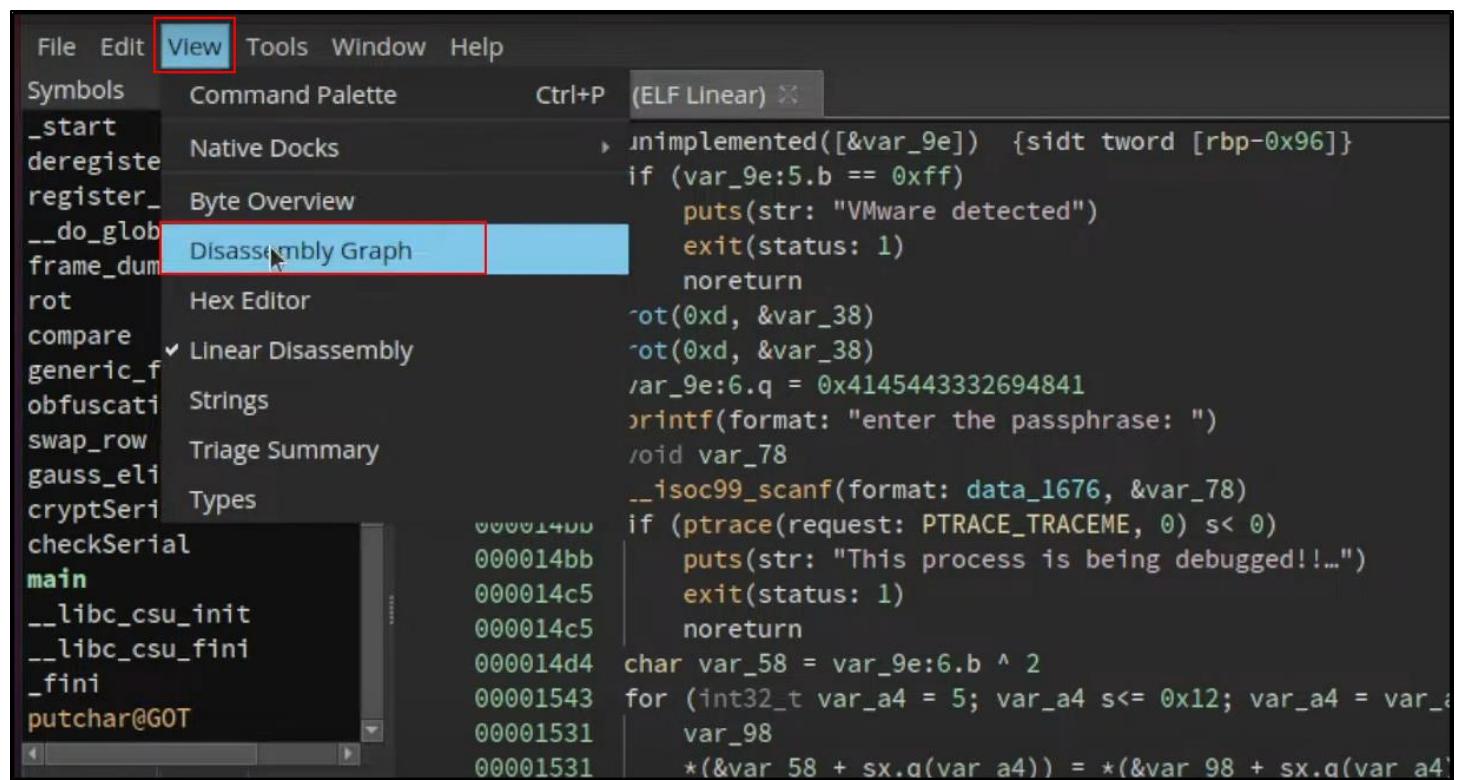
```
000013fc    unimplemented([&var_9e]) {sidt tword [rbp-0x96]}
00001415    if (var_9e:5.b == 0xff)
00001415        puts(str: "VMware detected")
0000141f        exit(status: 1)
0000141f        noreturn
00001430    rot(0xd, &var_38)
00001441    rot(0xd, &var_38)
0000145a    var_9e:6.q = 0x4145443332694841
0000147e    printf(format: "enter the passphrase: ")
00001496    void var_78
00001496    __isoc99_scanf(format: data_1676, &var_78)
000014bb    if (ptrace(request: PTRACE_TRACEME, 0) < 0)
000014bb        puts(str: "This process is being debugged!!...")
000014bb        exit(status: 1)
000014c5        noreturn
000014d4    char var_58 = var_9e:6.b ^ 2
00001543    for (int32_t var_a4 = 5; var_a4 <= 0x12; var_a4 = var_a4 + 1)
00001531        var_98
00001531        *(&var_58 + sx.q(var_a4)) = *(&var_98 + sx.q(var_a4))
00001558    if (strcmp(&var_58, &var_78) != 0)
00001558        puts(str: "try again")
00001571    else
00001571        puts(str: "you succeed!!")
00001563    if ((rax ^ *(fsbase + 0x28)) == 0)
00001563        return 0
00001590    __stack_chk_fail()
00001590    noreturn
```

The left sidebar shows symbols like _start, deregister_tm_clones, register_tm_clones, __do_global_dtors_aux, frame_dummy, rot, compare, generic_fizz_buzz, obfuscation, swap_row, gauss_eliminate, cryptSerial, checkSerial, main, __libc_csu_init, __libc_csu_fini, _fini, putchar@GOT, and various cross-references for _start.

To yha pr hum ise **psudocode** isliye bolte hai kyoki exact **C** ka code hum nhi nikal pate hai.

Aur sabhi **decompiler** assembly code ko alag-2 tarike se **decompile** krte hai.

To hum chalte hai **disassembly** me.



Ab hum **main** pr double click krte hai.

The screenshot shows the Immunity Debugger interface with the file "crackme6 (ELF Graph)" open. The left pane displays a list of symbols, including `_start`, `deregister_tm_clones`, `register_tm_clones`, `__do_global_dtors_aux`, `frame_dummy`, `rot`, `compare`, `generic_fizz_buzz`, `obfuscation`, `swap_row`, `gauss_eliminate`, `cryptSerial`, `checkSerial`, `main` (which is selected and highlighted in red), `__libc_csu_init`, `__libc_csu_fini`, `_fini`, and `putchar@GOT`. The right pane shows the assembly code for the `main` function:

```
int32_t main(int32_t arg1, char** arg2, char** arg3)

main:
    push    rbp {__saved_rbp}
    mov     rbp, rsp {__saved_rbp}
    sub     rsp, 0xb0
    mov     dword [rbp-0xa4 {var_ac}], edi
    mov     qword [rbp-0xb0 {var_b8}], rsi
    mov     rax, qword [fs:0x28]
    mov     qword [rbp-0x8 {var_10}], rax
    xor     eax, eax {0x0}
    lea     rdi, [rel data_17dd] {"*****"}
    call    puts
    lea     rdi, [rel data_17f5] {"**      rules:"}
    call    puts
    lea     rdi, [rel data_17dd] {"*****"}
    call    puts
    mov     edi, 0xa
    call    putchar
    lea     rdi, [rel data_180d] {"* do not bruteforce
    call    puts
```

To yha pr **main** function khul gya. Right side dekh skte hai. ki "**vmware detected**" message show ho gya.

The screenshot shows the assembly code for the `main` function, specifically the instruction at address `0x40101851`:

```
lea     rdi, [rel data_1851] {"VMware detected"}
call   puts
mov     edi, 0x1
call   exit
```

The string `"VMware detected"` is highlighted in red.

```

    mov    qword [rbp-0x30 {var_38}], rax  {0x2073692073696854}
    mov    qword [rbp-0x28 {var_30}], rdx  {0x657320706f742061}
    mov    rax, 0x7865742074657263
    mov    rdx, 0x67617373656d2074
    mov    qword [rbp-0x20 {var_28}], rax  {0x7865742074657263}
    mov    qword [rbp-0x18 {var_20}], rdx  {0x67617373656d2074}
    mov    word [rbp-0x10 {var_18}], al   0x2165
    mov    byte [rbp-0xe {var_16}], 0x0
    sidt   tword [rbp-0x96 {var_9e}] {var_9e} {var_9e}
    movzx  eax, byte [rbp-0x91 {var_9e+0x5}]
    cmp    al, 0xff
    jne    0x1424

```

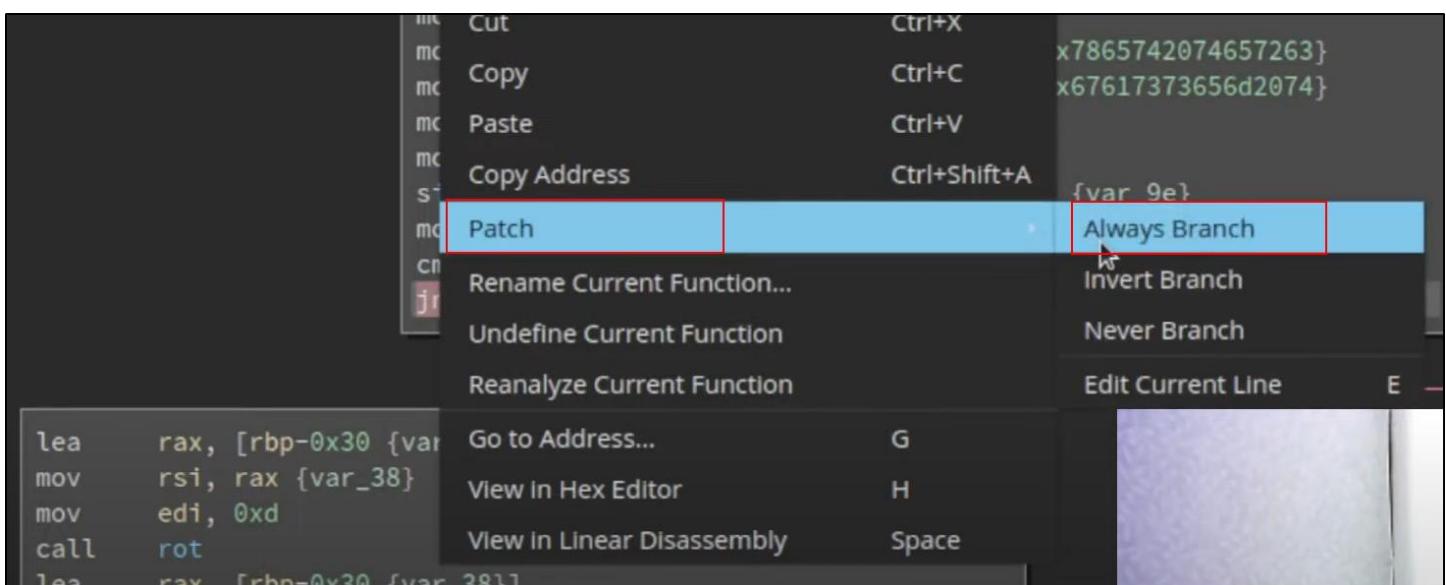
```

    lea    rax, [rbp-0x30 {var_38}]
    mov    rsi, rax {var_38}
    mov    edi, 0xd
    call   rot
    lea    rax, [rbp-0x30 {var_38}]
    mov    rsi, rax {var_38}
    mov    edi, 0xd

```

Agar hum jne ko je karenge to ye **virtual machine** pr to run karegi lekin host machine pr run krna band kr degi.

Hume koi aisa way dudna hai ki ye har bar left side wale branch pr aaye. To hum **jne** ki jagah **jmp** use karenge pr right click karenge **patch** pr jayenge fir **always branch** krenge.



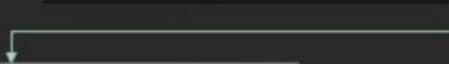
Agar hume **red** wale branch pr jump karana hota to hum **never branch** kr dete. Fir ye humesa **red** wale pr jump krtा.

To hune yha pr always branch kr diya hai.

```
mov    qword [rbp-0x20 {var_20}], rax  0x100
mov    qword [rbp-0x18 {var_20}], rdx  0x676
mov    word [rbp-0x10 {var_18}], 0x2165
mov    byte [rbp-0xe {var_16}], 0x0
sidt   tword [rbp-0x96 {var_9e}] {var_9e} {var_9e}
movzx  eax, byte [rbp-0x91 {var_9e+0x5}]
cmp    al, 0xff
jmp    0x1424
```

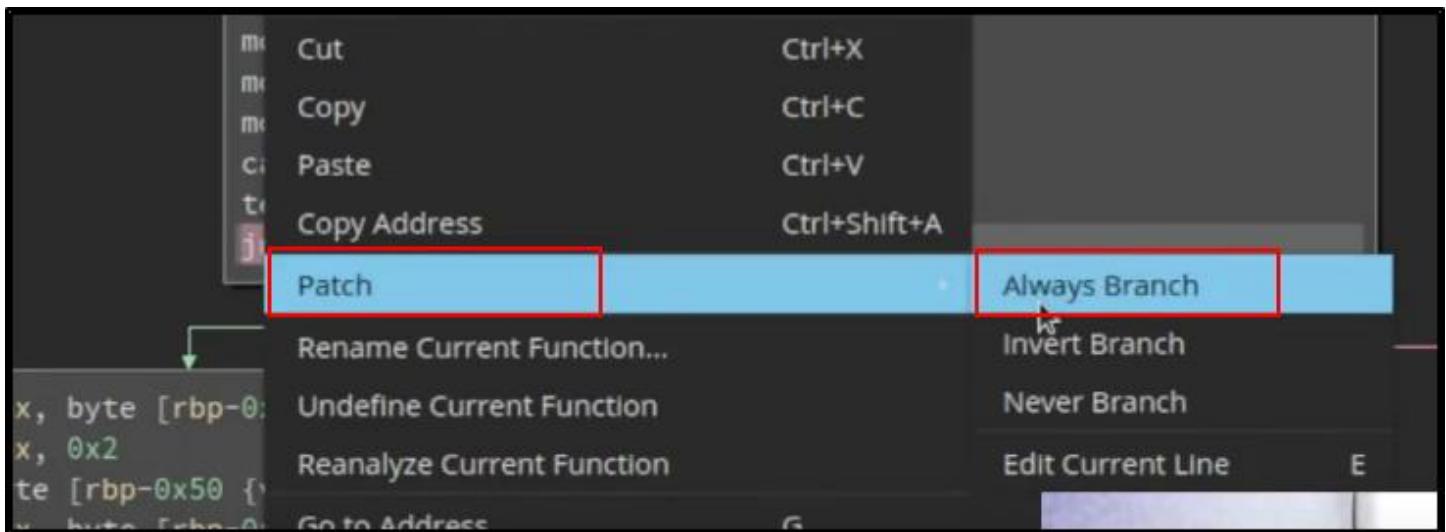


```
mov    esi, 0x0
mov    edi, 0x0
mov    eax, 0x0
call   ptrace
test   rax, rax
jns    0x14ca
```



```
lea    rdi, puts
mov    edi, exit
```

```
lea    rdi, [rel data_1878] {"This process is being debugged!!..."}
call   puts
```



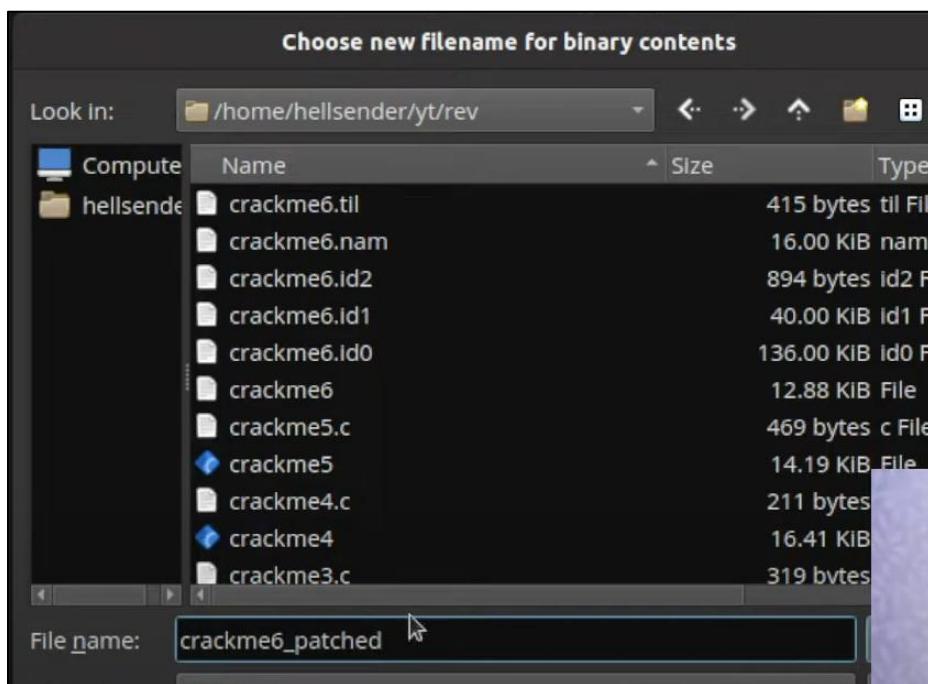
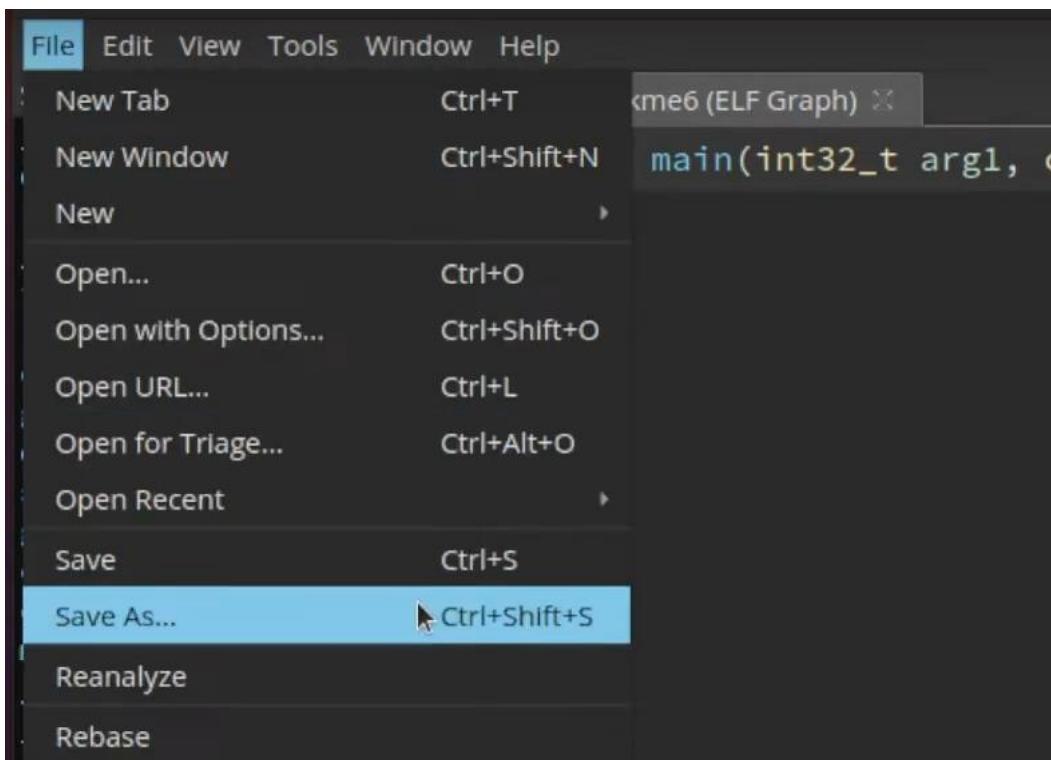
A screenshot of the Immunity Debugger interface showing assembly code. The code includes instructions such as 'mov', 'call', 'test', and 'jmp'. A specific 'jmp' instruction at address 0x14ca is highlighted with a red box. An arrow points from this highlighted instruction down to another section of assembly code below, which contains instructions like 'movzx', 'xor', 'mov', 'sub', and 'mov'. This second section is also highlighted with a red box.

```
mov    eax, 0x0
mov    eax, 0x0
call   ptrace
test   rax, rax
jmp   0x14ca

movzx eax, byte [rbp-0x90 {var_9e+0x6}]
xor   eax, 0x2
mov   byte [rbp-0x50 {var_58}], al
movzx eax, byte [rbp-0x8d {var_9e+0x9}]
sub   eax, 0xa
mov   byte [rbp-0x4f {var_57}], al
movzx eax, byte [rbp-0x90 {var_9e+0x6}]
```

Ab in dono ko kahi bhi run karao koi tension nhi. Ab humne ise bhi **patch** kr di. Anti VM ko bhi aur **anti debugger** ko bhi.

Ab isko **save as** kr lete hai.



Ab hum ise **run** krke dekh lete hai. ki ye sahi se work kr rhi hai ya nhi. Kyoki abhi-2 ek bhi byte idhar udhar hua to **binary** corrupt ho jati hai.

```

pwndbg> q
→ rev chmod +x crackme6_patched
→ rev ./crackme6_patched
*****
**      rules:      **
*****  

* do not bruteforce
* do not patch, find instead the serial.  

enter the passphrase: 12345
try again
→ rev

```

To ab tk humne **anti debugging** aur **anti vm technique** ko bypass kiya. Reversing nhi kiya.

```

● 30 printf("enter the passphrase: ");
● 31 __isoc99_scanf("%s", s2);
● 32 if ( ptrace(PTRACE_TRACEME, 0LL) < 0 )
● 33 {
● 34     puts("This process is being debugged!!!");
● 35     exit(1);
● 36 }
● 37 s1[0] = v6[0] ^ 2;
● 38 s1[1] = v6[3] - 10;
● 39 s1[2] = v6[2] + 12;
● 40 s1[3] = v6[2];
● 41 s1[4] = v6[1] + 1;
● 42 for ( i = 5; i <= 18; ++i )
● 43     s1[i] = v6[i] - 1;
● 44 if ( !strcmp(s1, s2) )
● 45     puts("you succeed!!!");
● 46 else
● 47     puts("try again");
● 48 return 0;
● 49 }

```

Ye portion humare kam ka lgta hai. ise samajhne ki koshish krte hai.

```

8 rot(13LL, v9);
9 qmemcpy(v6, "AH123DEADBEEFCOFFEE", 19);
0 printf("enter the passphrase: ");
1 __isoc99_scanf("%s", input);
2 if ( ptrace(PTRACE_TRACE_ME, 0LL) < 0 )
3 {
4     puts("This process is being debugged!!!!");
5     exit(1);
6 }
7 s1[0] = v6[0] ^ 2;
8 s1[1] = v6[3] - 10;
9 s1[2] = v6[2] + 12;
0 s1[3] = v6[2];
1 s1[4] = v6[1] + 1;
2 for ( i = 5; i <= 18; ++i )
3     s1[i] = v6[i] - 1;
4 if ( !strcmp(s1, input) )
5     puts("you succeed!!!");
6 else
7     puts("try again");
8 return 0;
9 }
```

Yha hum dekh skte hai ki **s1** ek variable hai jiske andar **v6** ke values pr operation perform krke ek-2 krke store kiya ja rha hai.

Iske bad **s2** se compare kiya ja rha hai **s2** humara **input** hai.

Agar dono equal hue to ye print kr dega "**you succeed**" otherwise "**try again**".

Yha **s1** ke andar **original password** hai.

Yha hum algorithm samajhte hai ki password kaise generate kr rha hai.

Isko rename krke hum **original** kr dete hai.

\wedge means xor

Is code mai subline pr le chalte hai aur syntax C select kr letे hai.

```
untitled •  
1 v6 = "AH!23DEADBEEFCOFFEE"  
2 |  
3 original[0] = v6[0] ^ 2;  
4 original[1] = v6[3] - 10;  
5 original[2] = v6[2] + 12;  
6 original[3] = v6[2];  
7 original[4] = v6[1] + 1;  
8 for ( i = 5; i <= 18; ++i )  
9 | original[i] = v6[i] - 1;
```

Isko solve krne ke do tarike hai.

```
solver.c •  
1 #include<stdio.h>  
2  
3 int main() {  
4 char v6[] = "AH!23DEADBEEFCOFFEE";  
5 char original[50];  
6 int i;  
7 original[0] = v6[0] ^ 2;  
8 original[1] = v6[3] - 10;  
9 original[2] = v6[2] + 12;  
10 original[3] = v6[2];  
11 original[4] = v6[1] + 1;  
12 for ( i = 5; i <= 18; ++i )  
13 | original[i] = v6[i] - 1;  
14  
15 puts(original);  
16  
17 return 0;  
18 }
```

Compile and run krte hai.

```
→ rev gcc solver.c -o solver
→ rev ./solver
C(uiICD@CADDEBNEEDD+
→ rev
```

To yha pr ye abnormal output show kr rha hai.

Isko ek bar aur run krke dekh lete hai.

```
→ rev ./solver
C(uiICD@CADDEBNEEDDVe
→ rev ./crackme6
*****
**      rules:      **
*****

* do not bruteforce
* do not patch, find instead the serial. I

enter the passphrase: C(uiICD@CADDEBNEEDDVe
try again
→ rev
```

Yha pr hum is code ko python me write karenge.

```
untitled •
1 main = "AHi23DEADBEEFCOFFEE"
2
3 original = []
4
5 original.append(ord(main[0]) ^ 2)
```

ord() function **character** ko **integer** me convert krta hai. yha pr jo "A" hai wo integer me convert hoga fir uska **xor** hoga.

Fir use character me convert karenge.

```
untitled •  
1 main = "AHi23DEADBEEFCOFFEE"  
2  
3 original = []  
4  
5 original.append(chr(ord(main[0]) ^ 2))  
6 original.append(chr(ord(main[3]) - 10))  
7 original.append(chr(ord(main[2]) + 12))  
8 original.append(main[2])  
9 original.append(chr(ord(main[1]) + 1))  
10 for i in main[5:]:  
11     original.append(chr(ord(i)-1))  
12  
13 print("".join(original))
```

Output

```
→ rev chmod +x solver.py  
→ rev python3 solver.py  
C(uiICD@CADDEBNEEDD
```

Yha pr agar hum **C** ke program se compare kare to

```
→ rev ./solver  
C(uiICD@CADDEBNEEDD♦?♦  
→ rev
```

Ye bhi lgbhg same hai. bs last me two extra character add kr diya.

```
→ rev ./crackme6  
*****  
**      rules:      **  
*****  
  
* do not bruteforce  
* do not patch, find instead the serial.  
  
enter the passphrase: C(uiICD@CADDEBNEEDD  
you succeed!!
```

```
#####
#####
```

Reversing C++ Binaries. How To Deal With Packed Binaries:

Challenge – 7

```
→ rev ./crackme7
Enter Password: aaaaaa
Wrong Password!
→ rev
```

Ise hum gdb me open kr lete hai.

```
→ rev gdb ./crackme7
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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:
<http://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"...

pwndbg: loaded 195 commands. Type pwndbg [filter] for a list.

pwndbg: created \$rebase, \$ida gdb functions (can be used with print/break)

Reading symbols from ./crackme7...

(No debugging symbols found in ./crackme7)

pwndbg> █

Agar hum info functions kare to.

```
pwndbg> info functions  
All defined functions:  
pwndbg> █
```

Isne ek bhi functions show nhi kiye. Aisa **stripped binary** me bhi nhi hota hai. wha pr bhi kuchh functions or symbols show ho jate hai. jo ki binary ko run hone me jaruri hote hai.

```
pwndbg> disassemble main  
No symbol table is loaded. Use the "file" command.  
pwndbg> █
```

Agar hum ise start krke dekhe to.

```
[ DISASM ]  
► 0x49ffd0  push  rax  
0x49ffd1  push  rdx  
0x49ffd2  call   0x4a029f <0x4a029f>  
  
0x49ffd7  push  rbp  
0x49ffd8  push  rbx  
0x49ffd9  push  rcx  
0x49ffda  push  rdx  
0x49ffdb  add   rsi, rdi  
0x49ffde  push  rsi  
0x49ffdf  mov   rsi, rdi  
0x49ffe2  mov   rdi, rdx  
  
[ STACK ]  
00:0000| rsp 0x7fffffff020 ← 0x1  
01:0008| 0x7fffffff028 → 0x7fffffff359 ← '/home/hellsender/yt/rev/crackme7'  
02:0010| 0x7fffffff030 ← 0x0  
03:0018| 0x7fffffff038 → 0x7fffffff37a ← 'GJS_DEBUG_TOPICS=JS ERROR;JS LOG'  
04:0020| 0x7fffffff040 → 0x7fffffff39b ← 'SSH_AUTH_SOCK=/run/user/1000/keyring/ssh'  
05:0028| 0x7fffffff048 → 0x7fffffff3c4 ← 'SESSION_MANAGER=local:hellsender:@/tmp/.ICE-unix  
E-unix/2387'  
06:0030| 0x7fffffff050 → 0x7fffffff41e ← 'GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen  
68a0f7a'  
07:0038| 0x7fffffff058 → 0x7fffffff474 ← 'SSH_AGENT_PID=2329'  
[ BACKTRACE ]  
[...]
```

Yha hum dekh skte hai ki ye baki binaries ke kafi different hai. ye upar kisi function ko call kr rha hai.

Jbki baki binary ki sbse phle **_start** call hota hai aur wo **__libc_start_main** ko call krtा hai.

To ye thoda unusual binary hai.

Ab hum iske bare information gathering krte hai.

```
→ rev file crackme7  
crackme7: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, no section header  
→ rev
```

Yha **statically linked** ka mtlb isme **shared object file (libc)** ka bhi code aa jata hai aur programmer ne jo code likha hai wo bhi. to kafi badi binary ho jayegi kyoki statically linked hai.

Agar hum normal binary dekhe to uska information kuchh is tarah se aata hai.

```
ubuntu@ip-172-31-8-118:~$ file crackme6  
crackme6: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-6  
4.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=6d5e1ea9a99de8dd8b9e18a01ec6925571114484, not stripped  
ubuntu@ip-172-31-8-118:~$ |
```

Ab hum strings kr lete hai.

```
ei"579  
_vd-id%ABI-  
^lT.,"  
n.f<Zo  
B.eh  
gcc`f  
ot      +  
%}gR  
iJent  
l;,W  
Az"!G  
'Q,q  
"G^l  
H5H%  
]l*/  
]l(@b  
UPX!  
UPX!  
→ rev
```

To iska bhu result bahut unusual aaya hai. last me sections ke symbols show hote hai jaise .bss , .text , .data etc ek normal binary me. Lekin yha pr **UPX** show ho rha hai.

To iska ek reason hai jiske wajah se ye ho rha hai.

Hum pwntools install kr lete hai. jiske sath ek tool aata hai **checksec**. Jo binary ka security check krtा hai.

```
→ rev sudo pip3 install pwntools
```

```

→ rev checksec crackme7
[*] Checking for new versions of pwntools
    To disable this functionality, set the contents of /home/hellsender/.cache/.pwntools-cache-3.8/update
    Or add the following lines to ~/.pwn.conf or ~/.config/pwn.conf (or /etc/pwn.conf system-wide):
        [update]
        interval=never
[*] You have the latest version of Pwntools (4.6.0)
[*] '/home/hellsender/yt/rev/crackme7'
Arch:      amd64-64-little
RELRO:    No RELRO
Stack:    No canary found
NX:       NX enabled
PIE:     No PIE (0x400000)
Packer:   Packed with UPX
→ rev

```

To yha hum dekh skte hai ki ek packer ka use kiya gya hai. jiska nam hai **UPX**. Aur anti reversing ke kam aata hai.

Ab hum smjhne ki koshish karenge ki **UPX** anti reversing me kaise kam aati hai aur ye reverse engineer ka kam kaise hard kr deti hai.

So, what is **packing** and how to **unpack** it and reverse it.

Man lijiye humare pas ek binary jo hello world print kr rha hai.



Ab chahta hum ki main is binary ko packed binary bna du. to ise ek packer ke help se pack kr dunga. Packer basically ek software hota hai. jo humre binary ko pack krta hai.

Jb hum kisi binary ko pack krte hai. man lijiye ek example lete hai ise ek box ke andar pack kr dete hai. ab ye jo humne pack kiya ye usually format me pack nhi hoti hai. hum ise is tarike se pack krte hai ki ye compress ho jata hai iska size kafi chhota ho jata hai. aur ye data readable format me bhi nhi rhta.

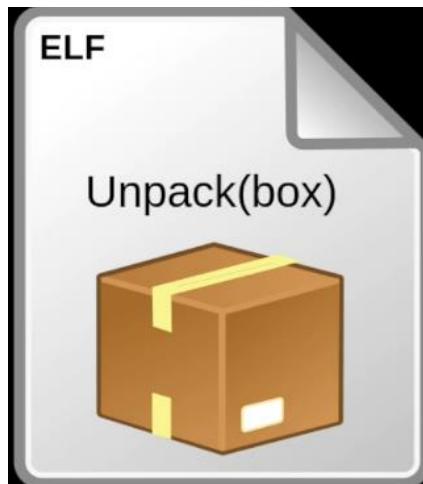
Software engineer packer ka use isliye krte hai ki unke program ka size chhota ho jaye. Yha packer anti reversing me bhi use hota hai. jaisa ki readable format me nhi hota to ise reverse engineer bhi nhi pd payega. Lekin ye linux me run hogा to linux bhi nhi pd payega

isko. Kyoki ye box me hide ho gaya. Linux bhi nhi run kr payega. Iska original code kaise milega linux ko run krne ke liye.



Jb packer kisi program ko pack krta hai. to original code ko compress krke rkhe deta hai box me aur apna chhota sa code rkhe deta hai box ke bahar. Jo isliye hota hai ki kaise ise unpack krna hai aur oringinal code ko bahar lana hai. jb ye binary run kr rhi ho. Run time pr.

Jb binary pack hoti hai to fayde. Iska koi string nhi dekh skta. Code nhi pd skta. Compress ho jati hai.



How can linux read it.

Jaise hi ye binary run karengi sbse phle unpack wala code run hoga. Aur box ko khol dega. Aur jo original code hoga use wapas le aayega aur run krna suru kr dega. Jo ki print hello world. Aur ye sb run time pr hi hoga.

=====

So, humare pas bhi jo **binary** hai. wo bhi **packed binary** hai. aur use **UPX packer** se pack kiya gaya hai.

Agar hume **UPX** install krna hai. to hum **apt install upx** karenge install ho jayega.

```
UPX!
UPX!
→ rev checksec crackme7
[*] Checking for new versions of pwntools
    To disable this functionality, set the contents of /home/hellsender/.cache/.pwntools-cache-3.8/updat
    Or add the following lines to ~/.pwn.conf or ~/.config/pwn.conf (or /etc/pwn.conf system-wide):
        [update]
        interval=never
[*] You have the latest version of Pwntools (4.6.0)
[*] '/home/hellsender/yt/rev/crackme7'
Arch:      amd64-64-little
RELRO:    No RELRO
Stack:    No canary found
NX:      NX enabled
PIE:     No PIE (0x400000)
Packer:   Packed with UPX
→ rev █
```

Yha pr hum dekh skte hai. ki **UPX checksec** me likha dikh jayega. Aur **strings** me bhi jayega jisse hum identify kr skte hai.

To hum **UPX** se **pack** kr skte hai to usi se **unpack** bhi kr skte hai.

```
→ rev upx -d crackme7 -o crackme7_unpacked
          Ultimate Packer for eXecutables
          Copyright (C) 1996 - 2018
UPX 3.95           Markus Oberhumer, Laszlo Molnar & John Reiser Aug 26th 2018

  File size       Ratio       Format       Name
-----
  2370480 <-    771308   32.54%   linux/amd64  crackme7_unpacked

Unpacked 1 file.
→ rev █
```

Yha pr humari binary **unpack** ho chuki hai.

```
→ rev ls
crackme  crackme1.c  crackme2.c  crackme3.c  crackme4.c  crackme5.c  crackme7      crackme7_unpacked
crackme1 crackme2    crackme3    crackme4    crackme5    crackme6    crackme7.cpp  crackme.c
→ rev
```

```
→ rev file crackme7_unpacked
crackme7_unpacked: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, BuildID[sha1]=5068695043f32052ef
28c83ed08a9a13e9f207e0, for GNU/Linux 3.2.0, not stripped
→ rev
```

Ab yha **file** command binary ki information aasani se pd pa rha hai.

String command bhi run krke dekh skte hai.

```
_ZNKSt7__cxx1112basic_stringIwSt11char_traitsIwESaIwEE4copyEPwmm
__realloc_hook
_ZNSolsEl
__gconv_get_builtin_trans
_ZN9_gnu_cxxeqIPcNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEEEbRKNS_17__normal_iteratorIT_T0_EESD_
_ZNKSt7__cxx118num_punctIcE11do_groupingEv
_ZTINSt7__cxx1117moneypunct_bynameIwLb1EEE
_ZNSt6locale5_Impl16_M_install_facetEPKNS_2idEPKNS_5facetE
_ZNKSt13basic_fstreamIcSt11char_traitsIcEE7is_openEv
_ZTVSi
_dl_tlsdesc_resolve_rela
_dl_hwcap
_ZNSbIwSt11char_traitsIwESaIwEE4rendEv
__madvise
_ZNKSt9basic_iosIcSt11char_traitsIcEE4fillEv
_ZNSt7__cxx1117moneypunct_bynameIcLb0EEC1ERKNS_12basic_stringIcSt11char_traitsIcESaIcEEE
_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE6insertEmPKcm
_ZNSsC2EOSSs
_ZNSspLEc
_ZNKSt7num_putIwSt19ostreambuf_iteratorIwSt11char_traitsIwEEE13_M_insert_intIyEES3_S3_RSt8ios_basewT_
_ZNSt10moneypunctIcLb0EED1Ev
_IO_file_jumps
_ZNSolsEj
_ZNSbIwSt11char_traitsIwESaIwEE6assignERKS2_mm
_ZNSbIwSt11char_traitsIwESaIwEEC1EPKumDKS1
```

```
.note.ABI-tag
.rela.plt
.init
.text
__libc_freeres_fn
.fini
.rodata
.stapsdt.base
.eh_frame
.gcc_except_table
.tdata
.tbss
.init_array
.fini_array
.data.rel.ro
.got
.got.plt
.data
__libc_subfreeres
__libc_IO_vtables
__libc_atexit
.bss
__libc_freeres_ptrs
.comment
.note.stapsdt
→ rev
```

To yha pr sare **sections** show ho rhe hai.

Aur ye c++ ka code hai to strings thode complex hote hai. aur ye statically linked hai to isme bahut sare function aayenga.

Ise **gdb** me open kr lete hai.

```
0x000000000005705d0  __unordtf2
0x000000000005707e0  __letf2
0x000000000005707e0  __lttf2
0x00000000000570a70  __sfp_handle_exceptions
0x00000000000570af0  base_of_encoded_value
0x00000000000570b70  read_encoded_value_with_base
0x00000000000570cc0  __gcc_personality_v0
0x00000000000570f40  free_derivation
0x00000000000571000  free_modules_db
0x00000000000571050  free_mem
0x00000000000572010  free_mem
0x00000000000572040  free_mem
0x00000000000572080  do_release_all
0x000000000005720a0  free_mem
0x000000000005720d0  free_category
0x00000000000572180  __nl_locale_subfreeres
0x000000000005723a0  __nl_archive_subfreeres
0x000000000005724a0  free_mem
0x00000000000572560  __nl_finddomain_subfreeres
0x000000000005725c0  __nl_unload_domain
0x000000000005726c0  buffer_free
0x00000000000572710  free_slotinfo
0x00000000000572790  free_mem
0x00000000000572a90  free_mem
0x00000000000572b80  free_mem
0x00000000000572bd0  free_mem
0x00000000000572c30  __fini
pwndbg> b main
Breakpoint 1 at 0x404c85
pwndbg> r
```

Aur **main** pr breakpoint lagate hai. aur **run** krte hai.

RIP 0x404c85 (main) ← endbr64

[DISASM]

```
► 0x404c85 <main>    endbr64
 0x404c89 <main+4>    push   rbp
 0x404c8a <main+5>    mov    rbp, rsp
 0x404c8d <main+8>    sub    rsp, 0x10
 0x404c91 <main+12>   mov    rax, qword ptr fs:[0x28]
 0x404c9a <main+21>   mov    qword ptr [rbp - 8], rax
 0x404c9e <main+25>   xor    eax, eax
 0x404ca0 <main+27>   lea    rax, [rbp - 9]
 0x404ca4 <main+31>   mov    rdi, rax
 0x404ca7 <main+34>   call   Password::Password() <Password::Password()>

 0x404cac <main+39>   mov    eax, 0
```

[STACK]

ni ke sath aage badte jate hai.

```
0x404c91 <main+12>   mov    rax, qword ptr fs:[0x28]
0x404c9a <main+21>   mov    qword ptr [rbp - 8], rax
0x404c9e <main+25>   xor    eax, eax
0x404ca0 <main+27>   lea    rax, [rbp - 9]
0x404ca4 <main+31>   mov    rdi, rax
► 0x404ca7 <main+34>   call   Password::Password() <Password::Password()>
  rdi: 0x7fffffffdec7 ← 0x3847d6d3acf90000
  rsi: 0x7fffffff008 → 0x7fffffff33e ← '/home/hellsender/yt/rev/crackme7_unpacked'
  rdx: 0x7fffffff018 → 0x7fffffff368 ← 'GJS_DEBUG_TOPICS=JS_ERROR;JS_LOG'
  rcx: 0x52cd90 (_dl_debug_state) ← endbr64

 0x404cac <main+39>   mov    eax, 0
 0x404cb1 <main+44>   mov    rdx, qword ptr [rbp - 8]
 0x404cb5 <main+48>   xor    rdx, qword ptr fs:[0x28]
 0x404cbe <main+57>   je    main+64 <main+64>

 0x404cc0 <main+59>   call   __stack_chk_fail_local <__stack_chk_fail_local>
```

[STACK]

Yha pr **Password** class ke **Password()** function ko call ho rha hai. aur **class** aur **function** ka nam same hai to iska mtlb ye **constructor** hai. class aur constructor ke beech me **scope resolution operator(::)**.

Agar object banega to **constructor** to call hoga hi hoga.

To hum si (step into) to Password() function ke andar aa gye.

```

[ DISASM ]
► 0x404d2e <Password::Password()>      endbr64
 0x404d32 <Password::Password()+4>    push   rbp
 0x404d33 <Password::Password()+5>    mov    rbp, rsp
 0x404d36 <Password::Password()+8>    sub    rsp, 0x30
 0x404d3a <Password::Password()+12>   mov    qword ptr [rbp - 0x28], rdi
 0x404d3e <Password::Password()+16>   mov    rax, qword ptr fs:[0x28]
 0x404d47 <Password::Password()+25>   mov    qword ptr [rbp - 8], rax
 0x404d4b <Password::Password()+29>   xor    eax, eax
 0x404d4d <Password::Password()+31>   lea    rdx, [rbp - 0x12]
 0x404d51 <Password::Password()+35>   mov    rax, qword ptr [rbp - 0x28]
 0x404d55 <Password::Password()+39>   mov    rsi, rdx
[ STACK ]

```

Ye constructor hai to ye kuchh special functions ko call krta hai. ya variable ko initialize krta hai.

ni krke next instructions ke or badte hai.

```

0x404d51 <Password::Password()+35>    mov    rax, qword ptr [rbp - 0x28]
0x404d55 <Password::Password()+39>    mov    rsi, rdx
0x404d58 <Password::Password()+42>    mov    rdi, rax
► 0x404d5b <Password::Password()+45>    call   Password::get_password(char*) <Password::get_password(char*)>
  rdi: 0x7fffffffdec7 ← 0x3847d6d3acf90000
  rsi: 0x7fffffffde9e ← 0x7fffffff0080000
  rdx: 0x7fffffffde9e ← 0x7fffffff0080000
  rcx: 0x52cd90 (_dl_debug_state) ← endbr64

0x404d60 <Password::Password()+50>    nop
0x404d61 <Password::Password()+51>    mov    rax, qword ptr [rbp - 8]
0x404d65 <Password::Password()+55>    xor    rax, qword ptr fs:[0x28]
0x404d6e <Password::Password()+64>    je    Password::Password()+71 <Password::Password()+71>

0x404d70 <Password::Password()+66>    call   __stack_chk_fail_local <__stack_chk_fail_local>
[ STACK ]

```

To yha pr ye **get_password(char*)** ko call kr rha hai jo argument me character array lete hai. jaisa ki name se smajh skte hai. ye password lena ka kam kr rha hogा.

Ab **get_password(char*)** ko **si** krte hai step into aur get password ko read krte hai.

ni krke aage badte hai.

```

[ DISASM ]
 0x404e56 <Password::get_password(char*)+8>    sub   rsp, 0x10
 0x404e5a <Password::get_password(char*)+12>   mov   qword ptr [rbp - 8], rdi
 0x404e5e <Password::get_password(char*)+16>   mov   qword ptr [rbp - 0x10], rsi
 0x404e62 <Password::get_password(char*)+20>   lea   rsi, [rip + 0x16e1c0]
 0x404e69 <Password::get_password(char*)+27>   lea   rdi, [rip + 0x1cf650] <0x5d44c0>
► 0x404e70 <Password::get_password(char*)+34>   call  0x46dbf0 <0x46dbf0>
[ STACK ]

 0x404e75 <Password::get_password(char*)+39>   mov   rdx, qword ptr [rip + 0x1cccd7c] <0x5d1bf8>
 0x404e7c <Password::get_password(char*)+46>   mov   rax, qword ptr [rbp - 0x10]
 0x404e80 <Password::get_password(char*)+50>   mov   esi, 0xb
 0x404e85 <Password::get_password(char*)+55>   mov   rdi, rax
 0x404e88 <Password::get_password(char*)+58>   call  fgets <fgets>
[ STACK ]

```

Yha pr ek function call ho rha hai. jise **gdb** nhi samjh pa rha hai nhi to ye hume bta deta function ka nam.

Lekin hum iska **argument** dekhkr smjh skte hai hai. kya function hai. **rdi, rsi** ki value dekhkr.

```
RAX 0xffffffffdec7 ← 0x3847d6d3acf90000
RBX 0x4005f0 ← 0x0
RCX 0x52cd90 (_dl_debug_state) ← endbr64
RDX 0x7fffffffde9e ← 0x7fffffff00800000
*RDI 0x5d44c0 (std::cout) → 0x5cedc0 → 0x46c490 ← endbr64
RSI 0x573029 ← 'Enter Password: '
R8 0x2
R9 0x2
R10 0x7
R11 0x1
R12 0x4a93c0 (__libc_csu_fini) ← endbr64
R13 0x0
R14 0x5d1018 (_GLOBAL_OFFSET_TABLE_+24) → 0x4f4080 (__rawmemchr_avx2) ← endbr64
```

To hum dekh skte hai **std::cout** function hai. to ye print krne ka kam krta hai.

Aur **rsi** me “**enter password**” hai to ye isi ko print kr rha hai.

Aage **next instructions** pr chalte hai.

```
► 0x404e88 <Password::get_password(char*)+58>    call   fgets <fgets>
  s: 0x7fffffffde9e ← 0x7fffffff00800000
  n: 0xb
  stream: 0x5d1a20 (_IO_2_1_stdin_) ← 0xfbad2088

0x404e8d <Password::get_password(char*)+63>    mov    rdx, qword ptr [rbp - 0x10]
```

Yha **fgets** function call ho rha hai. jaisa ki hume pta hai fgets input lene ka kam krta hai.

```
0x404e85 <Password::get_password(char*)+55>    mov    rdi, rax
► 0x404e88 <Password::get_password(char*)+58>    call   fgets <fgets>
  s: 0x7fffff1fde9e ← 0x7fffffff00800000
  n: 0xb
  stream: 0x5d1a20 (_IO_2_1_stdin_) ← 0xfbad2088
```

Yha pr ye **3 arguments** le rha hai.

First, input ko kis address pr store kare.

Second, **input** ki length kitni hai. **0xb** means **11** length. Jha tk hume lgta hai ki fgets **null byte** ko bhi count krta hai. to may bhi input 10 length. Null byte string ko end krne ke kam aata hai.

Third, kha se le rha hai. to **stdin** se.

To jaise hi **ni** karenge to ye input mangega.

```
pwndbg> ni  
Enter Password:aaaaaaaaaaaaaaaaaaaaaa
```

To hum yha thode se jyada 'a' de denge taki hum pta kar paye ki input **10** length ka le rha hai ya **11** length ka.

Ab hum us length ko examine krte hai.

```
pwndbg> x/s 0x7fffffffde9e  
0x7fffffffde9e: "aaaaaaaaaa" I  
pwndbg>
```

To yha pr usne **10** length ka input liya hai. aur ek null pointer.

So, password ye password ho skta hai. jo 10 length ka hoga ya usse km length ka.

Next instruction pr chlte hai.

```
0x404e98 <Password::get_password(char*)+74>    mov    rdi, rax  
► 0x404e9b <Password::get_password(char*)+77>    call   Password::check_password(char*) <Passw  
    rdi: 0x7fffffffdec7 ← 0x3847d6d3acf90000  
    rsi: 0x7fffffffde9e ← 'aaaaaaaaaa'  
    rdx: 0x7fffffffde9e ← 'aaaaaaaaaa'  
    rcx: 0x6161616161616161 ('aaaaaaaa')  
  
0x404ea0 <Password::get_password(char*)+82>    nop
```

Ab aa gaya **check_password** function jo argument me character array le rha hai. jo ki **humara password** hoga aur **original password** se **check** karega.

Aur ye bhi password class ke andar hi hai.

Password::check_password(char*)

```
► 0x404e9b <Password::get_password(char*)+77>    call   Password::check_password(char*) <Passw  
    rdi: 0x7fffffffdec7 ← 0x3847d6d3acf90000  
    rsi: 0x7fffffffde9e ← 'aaaaaaaaaa'  
    rdx: 0x7fffffffde9e ← 'aaaaaaaaaa'  
    rcx: 0x6161616161616161 ('aaaaaaaa')  
  
0x404ea0 <Password::get_password(char*)+82>    nop
```

Yha hum dekh skte hai. ki password **rsi** ke andar pass ho rha hai.

To yha rdi ke andar kyo nhi pass ho rha hai. kyoki **object oriented programming** me **rdi** ke andar first argument **this** pass hota hai.

Ab hum **check_password(char*)** function ke andar chalte hai. **si** command se.

```
*RSP 0x7fffffffde58 → 0x404ea0 (Password::get_password(char*)+82) ← nop
*RIP 0x404d78 (Password::check_password(char*)) ← endbr64
[ DISASM ]
▶ 0x404d78 <Password::check_password(char*)>      endbr64
 0x404d7c <Password::check_password(char*)+4>      push   rbp
 0x404d7d <Password::check_password(char*)+5>      mov    rbp, rsp
 0x404d80 <Password::check_password(char*)+8>      sub    rsp, 0x30
 0x404d84 <Password::check_password(char*)+12>     mov    qword ptr [rbp - 0x28], rdi
 0x404d88 <Password::check_password(char*)+16>     mov    qword ptr [rbp - 0x30], rsi
 0x404d8c <Password::check_password(char*)+20>     mov    rax, qword ptr fs:[0x28]
 0x404d95 <Password::check_password(char*)+29>     mov    qword ptr [rbp - 8], rax
 0x404d99 <Password::check_password(char*)+33>     xor    eax, eax
 0x404d9b <Password::check_password(char*)+35>     movabs rax, 0x706d552267616b4c
 0x404da5 <Password::check_password(char*)+45>     mov    qword ptr [rbp - 0x13], rax
[ STACK ]
```

Ab hum dekh lete hai kitne functions hai. Password class ke andar.

disassemble Password:: likhkr **tab** hit karenge. To ye show kr dega total kitne functions hai.

```
pwndbg> disassemble Password::
Password::Password()           Password::check_password(char*)  Password::get_password(char*)
pwndbg> disassemble Password::
```

To yha hum **check_password(char*)** ko **disassemble** krte hai.

```
pwndbg> disassemble Password::
Password::Password()           Password::check_password(char*)  Password::get_password(char*)
pwndbg> disassemble Password::check_password(char*)
No symbol table is loaded. Use the "file" command.
pwndbg> disassemble Password::check_password()
No symbol table is loaded. Use the "file" command.
pwndbg> disassemble Password::check_password
No symbol table is loaded. Use the "file" command.
pwndbg> c
```

To ye yha disassemble nhi kr pa rha hai.

To hum **context** kr lete hai. jisse ye dubara se print (load) ho jayega.

```
pwndbg> context
```

[DISASM]

```

0x404d78 <Password::check_password(char*)>    endbr64
0x404d7c <Password::check_password(char*)+4>  push   rbp
0x404d7d <Password::check_password(char*)+5>  mov    rbp, rsp
0x404d80 <Password::check_password(char*)+8>  sub    rsp, 0x30
► 0x404d84 <Password::check_password(char*)+12> mov    qword ptr [rbp - 0x28], rdi
0x404d88 <Password::check_password(char*)+16>  mov    qword ptr [rbp - 0x30], rsi
0x404d8c <Password::check_password(char*)+20>  mov    rax, qword ptr fs:[0x28]
0x404d95 <Password::check_password(char*)+29>  mov    qword ptr [rbp - 8], rax
0x404d99 <Password::check_password(char*)+33>  xor    eax, eax
0x404d9b <Password::check_password(char*)+35>  movabs rax, 0x706d552267616b4c
0x404da5 <Password::check_password(char*)+45>  mov    qword ptr [rbp - 0x13], rax

```

[STACK]

```

00:0000 | rbp 0x7fffffffde20 ← 0x0
01:0008 | 0x7fffffffde28 → 0x4d21ea (fgets+154) ← mov     edx, dword ptr [rbx]
02:0010 | 0x7fffffffde30 ← 0x0
03:0018 | 0x7fffffffde38 → 0x4005f0 ← 0x0
04:0020 | 0x7fffffffde40 → 0x7fffffffde70 → 0x7fffffffdeb0 → 0x7fffffffded0 → 0x4a9320 (_
05:0028 | 0x7fffffffde48 → 0x4a93c0 (__libc_csu_fini) ← endbr64
06:0030 | rbp 0x7fffffffde50 → 0x7fffffffde70 → 0x7fffffffdeb0 → 0x7fffffffded0 → 0x4a9320 (_
07:0038 | 0x7fffffffde58 → 0x404ea0 (Password::get_password(char*)+82) ← nop

```

[BACKTRACE]

```

► f 0           0x404d84 Password::check_password(char*)+12

```

Ab hum **disassemble** likhenge to ye isi function ko **disassemble** kr dega.

```

pwndbg> disassemble
Dump of assembler code for function _ZN8Password14check_passwordEPc:
0x0000000000404d78 <+0>:    endbr64
0x0000000000404d7c <+4>:    push   rbp
0x0000000000404d7d <+5>:    mov    rbp,rsp
0x0000000000404d80 <+8>:    sub    rsp,0x30
=> 0x0000000000404d84 <+12>:   mov    QWORD PTR [rbp-0x28],rdi
0x0000000000404d88 <+16>:   mov    QWORD PTR [rbp-0x30],rsi
0x0000000000404d8c <+20>:   mov    rax,QWORD PTR fs:0x28
0x0000000000404d95 <+29>:   mov    QWORD PTR [rbp-0x8],rax
0x0000000000404d99 <+33>:   xor    eax,eax
0x0000000000404d9b <+35>:   movabs rax,0x706d552267616b4c
0x0000000000404da5 <+45>:   mov    QWORD PTR [rbp-0x13],rax
0x0000000000404da9 <+49>:   mov    WORD PTR [rbp-0xb],0x2369
0x0000000000404daf <+55>:   mov    BYTE PTR [rbp-0x9],0x0
0x0000000000404db3 <+59>:   mov    DWORD PTR [rbp-0x18],0x0
0x0000000000404dba <+66>:   cmp    DWORD PTR [rbp-0x18],0x9
0x0000000000404dbe <+70>:   jg    0x404e14 <_ZN8Password14check_passwordEPc+156>
0x0000000000404dc0 <+72>:   mov    eax,DWORD PTR [rbp-0x18]
0x0000000000404dc3 <+75>:   movsxd rdx,eax
0x0000000000404dc6 <+78>:   mov    rax,QWORD PTR [rbp-0x30]
0x0000000000404dca <+82>:   add    rax,rdx
0x0000000000404dcd <+85>:   movzx  edx,BYTE PTR [rax]
0x0000000000404dd0 <+88>:   mov    eax,DWORD PTR [rbp-0x18]

```

To yha pr ye disassemble ho gaya.

To ab hum yha **cmp**, **test** pr focus krte hai. kyoki **cmp**, **test** ye jyada useful hoti hai reversing point of view se.

```
0x00000000000404dd3 <+91>: cdqe
0x00000000000404dd5 <+93>: movzx eax,BYTE PTR [rbp+rax*1-0x13]
0x00000000000404dda <+98>: xor eax,0x2
0x00000000000404ddd <+101>: cmp dl,al
0x00000000000404ddf <+103>: je 0x404e0d <_Z18Password14check_passwordEPc+149>
0x00000000000404de1 <+105>: lea rsi,[rip+0x16e21d] # 0x573005
```

Jb bhi **dl**, **al** ho to yha array variable hota hai. jha ek character **dl** ke andar hoga, ek character **al** ke andar hoga aur dono compare ho rhe honge for loop ke andar ek-2 krke. Jaise humne koi string di. To string ka first char, second char, third char... so on. Aapas me cmp hota hai.

```
0x00000000000404dda <+98>: xor eax,0x2
0x00000000000404ddd <+101>: cmp dl,al
```

Yha **eax** ka 2 se **xor** ho rha hai. jaise ki hum jante hai. **al** eax ki last byte hoti hai. fir compare ho rha hai.

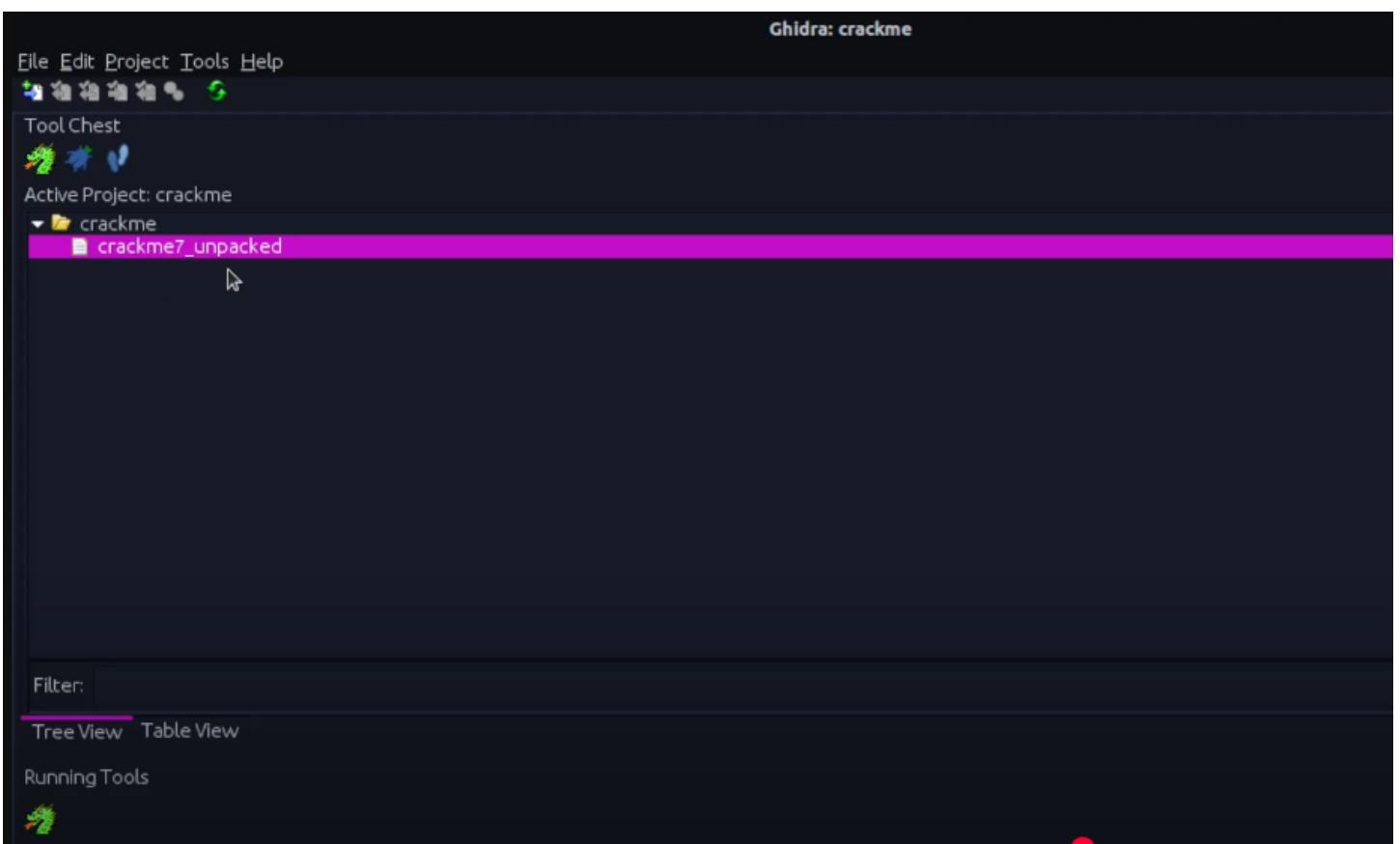
To aisi situation me hum **decompiler** ke pas jate hai. wo ise aasan bna dega. Waise hum yha assembly ko bhi read krke samajh skte hai.

Yha pr hum **al** ki value **registers** ko dekhkr copy krte rhenge aur hume original password pta chal jayega.

To ye tarika hai dynamically analysis ka ye easy hai.

Ab dekhte hai **static analysis decompiler** ki help se.

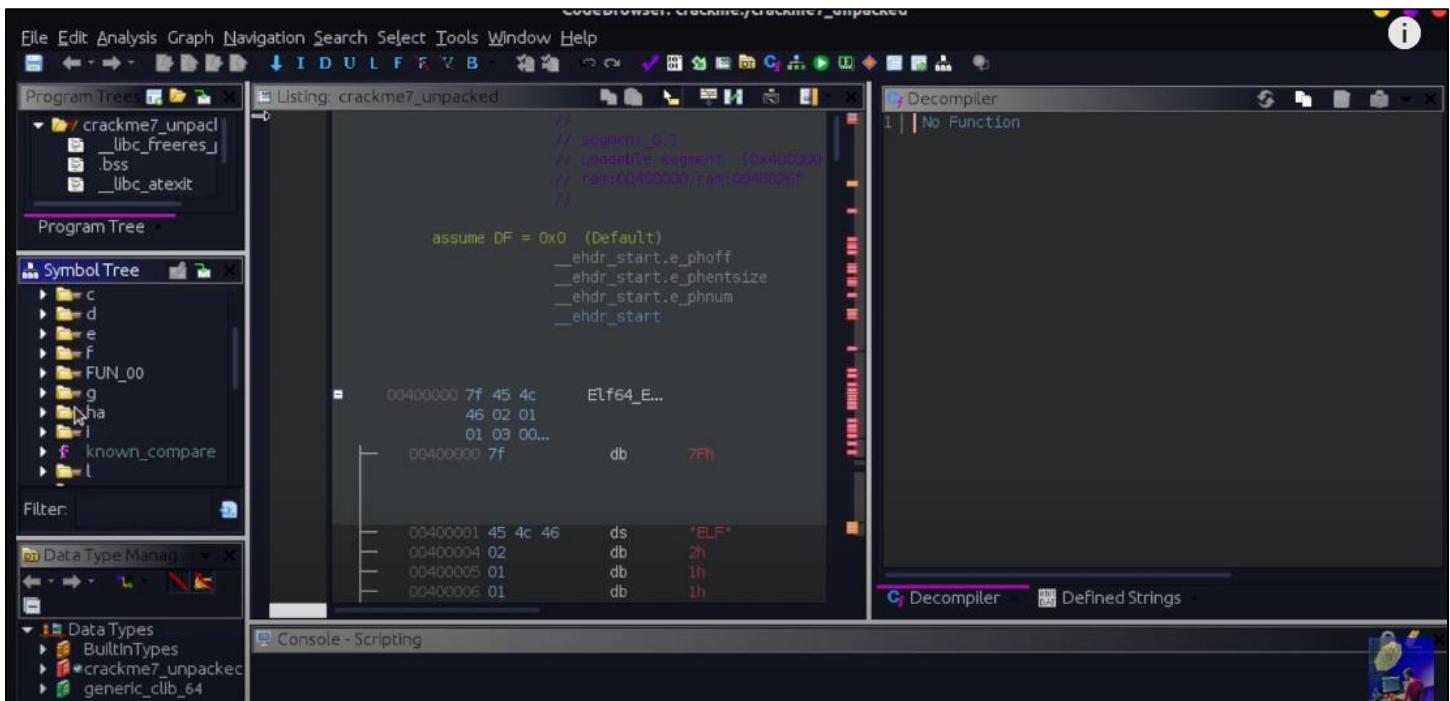
Hum apni binary **ghidra** me load kr lete hai.



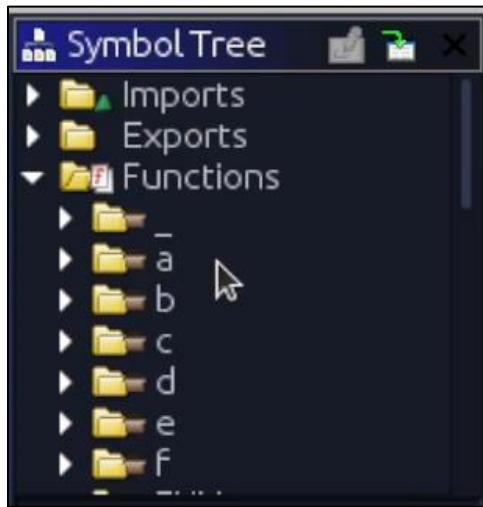
To ye statically linked hai to ye bahut jayada time lega analysis krne me. 5 min ke liye chhad dete hai analysis krne ke liye.

Ye sbka psudocode, decompliel, disassembly banana isliye ye time deta hai.

Yha sb kuchh ready ho gaya hai.



Yha hum dekhe to bahut sare functions hai to isne a,b,c,d nam se sort kr diya.



To hum main ke andar jayege. Wahan hume main function mil jayega.



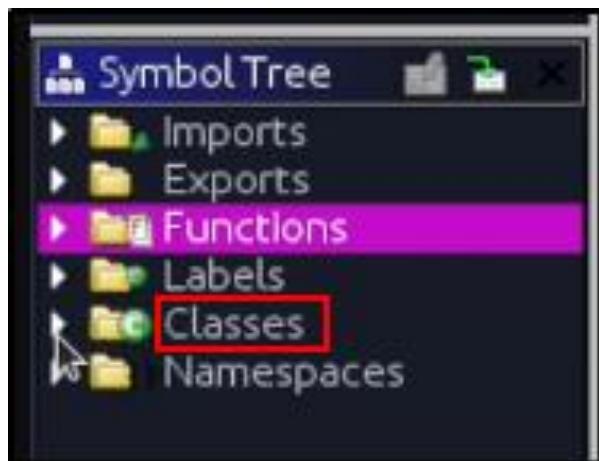
Yha **main** function open ho gya.

```
Decompile: main - (crackme7_unpacked)  ⌂ ⌃ ⌄ ⌅ ⌆ ⌇ ⌈ ⌉ ⌊ ⌋
```

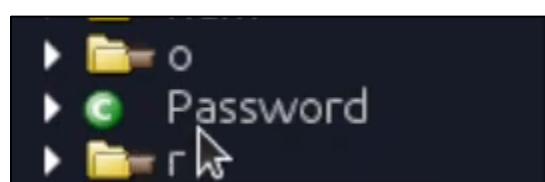
```
1
2 undefined8 main(void)
3
4 {
5     long in_FS_OFFSET;
6     Password local_11;
7     long local_10;
8
9     local_10 = *(long *)(in_FS_OFFSET + 0x28);
10    Password::Password(&local_11);
11    if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
12        /* WARNING: Subroutine does not return */
13        stack_chk_fail();
14    }
15    return 0;
16 }
```

To yha pr kuchh jyada nhi hai bs **Password** class **Password()** function ko call kr rha hai.

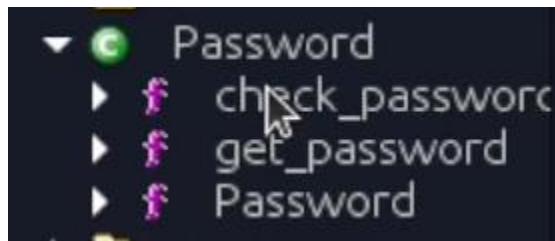
Yha pr hume function ke niche classes ka bhi folder show kr rha hai.



Agar hum class me chale to hume password class bhi mil jayegi.



Agar ise hum expand karenge to sare functions dikh jayenge.



Aap yha se bhi kisi bhi function ko follow kr skte ho.

Hum password pr double click krke open kar lenge.

```
1 undefined8 main(void)
2 {
3     long in_FS_OFFSET;
4     Password local_11;
5     long local_10;
6
7     local_10 = *(long *)(in_FS_OFFSET + 0x28);
8     Password::Password(&local_11);
9     if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
10         /* WARNING: Subroutine does not return */
11         stack_chk_fail();
12     }
13     return 0;
14 }
```

The screenshot shows the decompiled C code for the 'main' function. Line 10, which contains the call to the 'Password' constructor ('Password::Password(&local_11)'), is highlighted with a red rectangular box. The code uses standard C syntax with variable declarations and a conditional branch.

Cg Decompile: Password - (crackme7_unpacked)

```
1 /* Password::Password() */
2
3
4 void __thiscall Password::Password(Password *this)
5
6 {
7     long in_FS_OFFSET;
8     char local_1a [10];
9     long local_10;
10
11    local_10 = *(long *)(in_FS_OFFSET + 0x28);
12    get_password(this,local_1a);
13    if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
14        /* WARNING: Subroutine does not return */
15        __stack_chk_fail();
16    }
17    return;
18 }
```

Yha **Password()** function open ho gya hai.

To yha pr **Password()** function jayada kam nhi krta bs **get_password()** ko call krta hai.

To hum **get_password()** function pr double click krke open krte hai.

Cg Decompile: get_password - (crackme7_unpacked)

```
1 /* Password::get_password(char*) */
2
3
4 void __thiscall Password::get_password(Password *this,char *param_1)
5
6 {
7     std::operator<<((basic_ostream *)std::cout,"Enter Password: ");
8     fgets(param_1,0xb,(FILE *)stdin);
9     check_password(this,param_1);
10    return;
11 }
```

To ye function humare kam ka hai. ye **Enter password** print krta hai aur **fgets** ka use krke user se input leta hai aur **param_1** me store krta hai. uske bad **check_password()** function me pass kr rha hai.

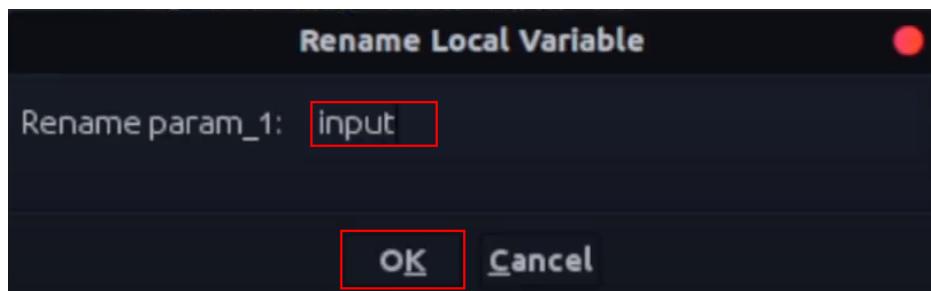
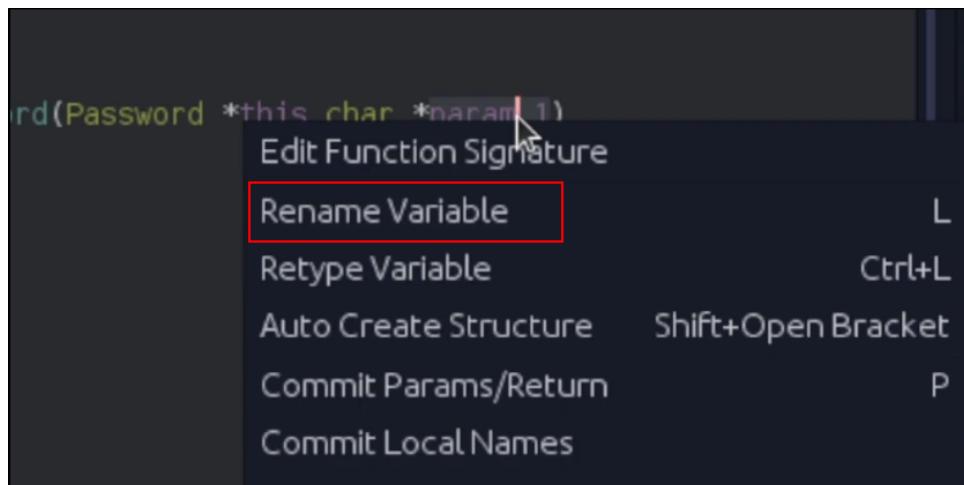
Ab hum chalte hai **check_password()** ke andar.

C:\Decompile: check_password - (crackme7_unpacked)

```
1 /* Password::check_password(char*) */
2
3 void __thiscall Password::check_password(Password *this, char *param_1)
4 {
5     basic_ostream *pbVar1;
6     long in_FS_OFFSET;
7     int local_20;
8     undefined8 local_1b;
9     undefined2 local_13;
10    undefined local_11;
11    long local_10;
12
13    local_10 = *(long *)(in_FS_OFFSET + 0x28);
14    local_1b = 0x706d552267616b4c;
15    local_13 = 0x2369;
16    local_11 = 0;
17    local_20 = 0;
18    while( true ) {
19        if (9 < local_20) {
20            pbVar1 = std::operator<<((basic_ostream *)std::cout, "Correct Password!");
21            std::operator<<(pbVar1, "\n");
22
23        }
```

```
while( true ) {
    if (9 < local_20) {
        pbVar1 = std::operator<<((basic_ostream *)std::cout, "Correct Password!");
        std::operator<<(pbVar1, "\n");
        if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
            /* WARNING: Subroutine does not return */
            __stack_chk_fail();
        }
        return;
    }
    if (input[local_20] != (*(byte *)((long)&local_1b + (long)local_20) ^ 2)) break;
    local_20 = local_20 + 1;
}
pbVar1 = std::operator<<((basic_ostream *)std::cout, "Wrong Password!");
std::operator<<(pbVar1, "\n");
```

Yha hum param1 ko change krke input kr lete hai.



```
Decompile: check_password - (crackme7_unpacked)
1
2 /* Password::check_password(char*) */
3
4 void __thiscall Password::check_password(Password *this,char *input)
5
6 {
7     basic_ostream *pbVar1;
8     long in_FS_OFFSET;
9     int local_20;
10    undefined8 local_1b;
```

Yha hum **local_20** ko rename krke i kr lete hai kyoki ye iterator hai.

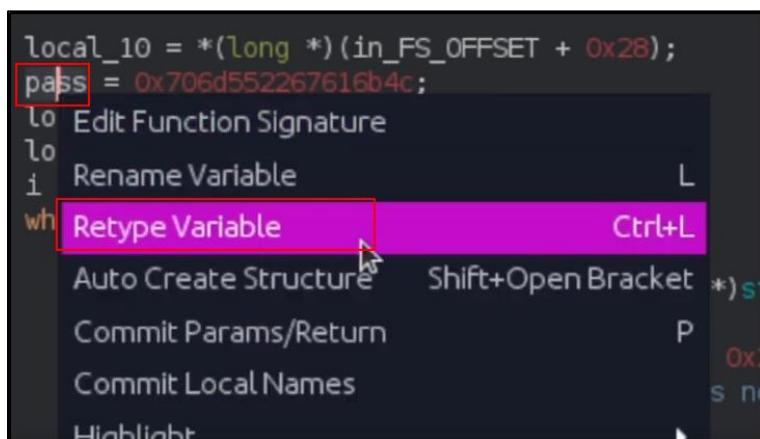
```

13 long local_10;
14
15 local_10 = *(long *)(in_FS_OFFSET + 0x28);
16 local_1b = 0x706d552267616b4c;
17 local_13 = 0x2369;
18 local_11 = 0;
19 i = 0;
20 while( true ) {
21     if (9 < i) {
22         pbVar1 = std::operator<<((basic_ostream *)std::cout, "Correct Password!");
23         std::operator<<(pbVar1, "\n");
24         if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
25             /* WARNING: Subroutine does not return */
26             __stack_chk_fail();
27         }
28         return;
29     }
30     if (input[i] != (*(byte *)((long)&local_1b + (long)i) ^ 2)) break;
31     i = i + 1;
32 }
33 pbVar1 = std::operator<<((basic_ostream *)std::cout, "Wrong Password!");
34 std::operator<<(pbVar1, "\n");

```

Yha pr ye if condition humare kam ka hai.

Yha hum **local_1b** ko rename kr lete hai. ise if condition me long dikha rha to ise retype krke char bna dete hai.



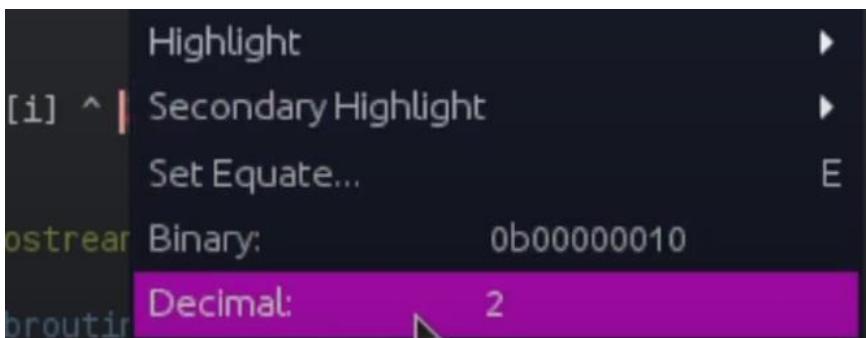
Now,

Decompiled C++ code:

```
13 long local_10;
14
15 local_10 = *(long *)(in_FS_OFFSET + 0x28);
16 _pass = 0x706d552267616b4c;
17 local_13 = 0x2369;
18 local_11 = 0;
19 i = 0;
20 while( true ) {
21     if (9 < i) {
22         pbVar1 = std::operator<<((basic_ostream *)std::cout, "Correct Password!");
23         std::operator<<(pbVar1, "\n");
24         if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
25             /* WARNING: Subroutine does not return */
26             __stack_chk_fail();
27         }
28         return;
29     }
30     if (input[i] != (byte)((&pass)[i] ^ 2U)) break;
31     i = i + 1;
32 }
33 pbVar1 = std::operator<<((basic_ostream *)std::cout, "Wrong Password!");
34 std::operator<<(pbVar1, "\n");
```

Yha pass **i** value ko ek-2 krke **2** se **xor** krke rha hai. fir original pura password bn jayega. Jisko compare karega **input** se.

Yha ye **2U** likha hai. ise decimal me dekhenge to ye **2** hai.



Yha hum python ki help se iske every value ko **2** se **xor** karenge to hume pura password mil jayega.

Isko copy krke unhex kr lete hai.

```
→ rev unhex 706d552267616b4c
```

```
pmU"gakL%
```

```
→ rev
```

Yha pr ye **8 length** ka hai yha pr **%** ka sign count nhi hoga ye **zsh** apne glti se deta hai.

Lekin yha 8 length ka password nhi ye ghidra ke glti se ho rha hai.

```
local_10 = *(long *)(in FS OFFSET + 0x28);  
_pass = 0x706d552267616b4c;  
local_13 = 0x2369;  
local_11 = 0;  
i = 0;
```

Yha do character niche dusre variable me stored hai. ise copy krte hai aur ise bhi **unhex** kr lete hai.

Yha pr **ghidra** jb bhi **8 length** se jyads password hota hai. To use dusre variable me dal deta hai.

Kyoki ek register ke andar **8 bytes** hi aa skte hai. kyoki registers maximum **64 bit** ka hi hota hai. 64 bit / 8 bit = 8 bytes.

Agar aapko **password** ya **flag** half milta hai to use dusre variable me dekh lena chahiye.

```
→ rev unhex 706d552267616b4c
```

```
pmU"gakL%
```

```
→ rev unhex 2369
```

```
#i%
```

```
→ rev
```

Ab hum ise jod lete hai. Note: **unhex** hamesa result ulta deta hai **little endian** format me.

```
→ rev Lkag"Umpi#
```

Ye humari final string hai. jiske every character ko hum **2 se xor** karenge.

Ab hum python ka program likhna suru krte hai.

```
string = 'Lkag"Umpi#'  
for i in string:  
    print(chr(ord(i)^2))
```

Yha single quote humne isliye use kiya kyoki string me double phle se hi hai.

```
print(chr(ord(i)^2))
```

Yha **ord()** function se phle humne i ko **integer** me convert kiya fir **2 se xor** kiya.

Fir humne jo bhi **result** aaya use **char()** function se **character** me convert kiya.

```
→ rev python3  
Python 3.8.10 (default, Jun 2 2021, 10:49:15)  
[GCC 9.4.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> string = 'Lkag"Umpi#'  
>>> for i in string:  
...     print(chr(ord(i)^2))  
...  
N  
i  
c  
e  
  
W  
o  
r  
k  
!  
>>> █ █
```

To yha pr “**Nice Work!**” humara password hai.

Let ‘s test the password.

```
→ rev ./crackme7  
Enter Password: Nice Work!  
Correct Password!  
→ rev █
```

Congratulation! Yha humne packed binary ko reverse engineer kr liya. Lekin hume abhi-2 manually bhi unpacking krni pdti hai. kyoki malware banane wale khud ka packer bnate hai.

```
#####
```

Reverse Engineering Python Binaries

Challenge - 8

```
→ rev ./crackme8
Enter Password - aaaa
W04K H4RD!!!
→ rev
```

Sbse phle hum **file** command run krke dekh lete hai.

```
→ rev file crackme8
crackme8: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/
dID[sha1]=f6af5bc244c001328c174a6abf855d682aa7401b, for GNU/Linux 2.6.32, stripped
→ rev
```

Yha pr kuchh aisa nhi milta ki hum pta kr ske ki ye binary kis language ka use krke banayi gyi hai.

Ab hum strings command run kr lete hai. isse hume pta chal jayega ki kis language pr bni hai kyoki **C**, **C++**, **python** inki binary alag-2 hoti hai.

```
→ rev strings crackme8
```

```
blibz.so.1
xbase_library.zip
zPYZ-00.pyz
&libpython3.8.so.1.0
.shstrtab
.interp
.note.gnu.build-id
.note.ABI-tag
```

```
.rela.plt
.init
.text
.fini
.rodata
.eh_frame_hdr
.eh_frame
.init_array
.fini_array
.dynamic
.got
.got.plt
.data
.bss
.comment
pydata
→ rev
```

Python ke binary ke andar py word bahut dekhne ko milta hai.

Agar hum strings ko grep py kare to hume pta chal jayega.

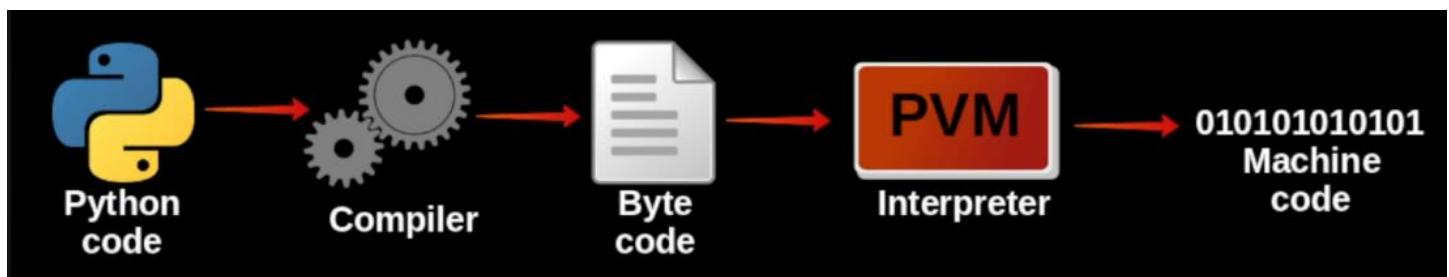
```
→ rev strings crackme8 | grep -i "py"
```

```
Cannot dlsym for PyObject_SetAttrString
Cannot dlsym for PyObject_GetAttrString
Cannot dlsym for PyObject_Str
Cannot dlsym for PyRun_SimpleString
Cannot dlsym for PySys_AddWarnOption
Cannot dlsym for PySys_SetArgvEx
Cannot dlsym for PySys_SetObject
Cannot dlsym for PySys_SetObject
Cannot dlsym for PySys_SetPath
Cannot dlsym for PyEval_EvalCode
PyMarshal_ReadObjectFromString
Cannot dlsym for PyMarshal_ReadObjectFromString
Cannot dlsym for PyUnicode_FromString
Cannot dlsym for Py_DecodeLocale
Cannot dlsym for PyMem_RawFree
Cannot dlsym for PyUnicode_FromFormat
Cannot dlsym for PyUnicode_Decode
Cannot dlsym for PyUnicode_DecodeFSDefault
Cannot dlsym for PyUnicode_AsUTF8
Cannot dlsym for PyUnicode_Join
Cannot dlsym for PyUnicode_Replace
pyi-
Error loading Python lib '%s': dlopen: %s
```

Bahut sare **py** likhe mil jayenge.

Ab hum phle dekhte hai ki jo python ka high level code hota hai. use machine level code me covert kaise kare. Fir hum use reverse engineer karenge.

Python ka code directly binary me nhi convert hota hai. jaisa ki hum jante hai **C** ka code phle **assembly** me covert hota hai. fir **machine code** me convert hota hai. python ke andar dono use hota hai **compiler** bhi aur **interpreter** bhi.



Sabse phle python ka **compiler** high level code ko **Byte code** me convert krta hai. fir **PVM (python Virtual Machine)** use machine code me convert krta hai. jaisa JVM hota hai theek waisa hi **PVM** bhi hota hai.

Byte code assembly ke jaisa code hota hai.

Let's see Byte code in python.

```
→ rev python3
Python 3.8.10 (default, Jun 2 2021, 10:49:15)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> def add(a,b):
...     return a+b
...
>>> add(1,2)
3
>>> import dis
>>> dis.dis(add)
 2      0 LOAD_FAST              0 (a)
      2 LOAD_FAST              1 (b)
      4 BINARY_ADD             I
      6 RETURN_VALUE           |
>>>
```

Yha humne byte code dekhne ke liye. **dis** module ko import kiya **dis** means **disassemble**.

Aur uske bad **add() function** ko disassemble kiya.

```
 2      0 LOAD_FAST              0 (a)
      2 LOAD_FAST              1 (b)
      4 BINARY_ADD             I
      6 RETURN_VALUE           |
```

Yha pr instructions likhe hai. aur instruction kis pr perform krna hai. wo bhi likha hai.

Yha pr **LOAD_FAST** instruction phle a ko stack ke andar load kr rha hai. fir b ko load kr rha hai.

BINARY_ADD instruction do variable ko **add** kr rha hai.

Uske bad **RETURN_VALUE** value return kr rha hai.

2	0 LOAD_FAST 2 LOAD_FAST 4 BINARY_ADD 6 RETURN_VALUE	0 (a) 1 (b)
---	--	----------------

aur **(a)** ke aage jo likha hai wo number hai. variable ka **0,1** agar koi **c** variable hota to uska number **2** ho jata.

2	0 LOAD_FAST 2 LOAD_FAST 4 BINARY_ADD 6 RETURN_VALUE	0 (a) 1 (b)
---	--	----------------

Aur instruction se phle jo number likha hai wo **offsets** hai. har instruction do bytes ka hota hai. ek hota hai instruction ek hoti hai value dono milakar do byte. Isliye number **0,2,4,6** hai. do-2 ke distance pr.

2	0 LOAD_FAST 2 LOAD_FAST 4 BINARY_ADD 6 RETURN_VALUE	0 (a) 1 (b)
---	--	----------------

Ye **2** line number hai code ka.

Actual me **byte code** memory me hota hai. wo human readable format me nhi hota. **disassembly** dikhati hai ki **byte code** dikhta kaisa hai. **dis** hume dikhane ki koshish krti hai humare readable format me. **Byte code** aisa nhi hota hai **byte code** ko terminal **print** nhi kr skta. **byte code** bytes me hota hai. jise hum nhi pd skte hai.

=====

Hume **python** ke **binary** ko **reverse engineering** karne ke liye **byte code** chahiye.

Jaisa ki hum jante hai ki Agar hume **.py** to **.exe** covert krna ya **.py to .elf** convert krna. To hum **pyinstaller** use krte hai.

pyinstaller ka ek khas bat hai. jb **pyinstaller** koi binary banata hai python ki to wh ek kam krtा hai. ki jaise **.text section** hota hai **.bss section** hota hai. waise hi ek section add kr deta hai. jiska nam hota hai **pydata**. Aur uske python ka kafi sara data rkھ deta hai.

Aur us data ke andar **byte code** bhi hota hai.

```
→ rev ls  
crackme crackme1.c crackme2.c crackme3.c crackme4.c crackme5.c crackme7  
crackme1 crackme2 crackme3 crackme4 crackme5 crackme6 crackme7.cpp  
→ rev [redacted]
```

Mtlb **crackme8.py** hai uska **byte code**, **crackme8** binary ke andar hoga. **pydata** section ke andar.

To hum check kr lete hai. ki crackme8 jo hai. usme pydata available hai ki nhi.

```
→ rev readelf -a crackme8 [redacted]
```

Yha **readelf** command use krenge **-a** for all uske bad binary file.

```
[16] .eh_frame_hdr PROGBITS 0000000000409f80 00009f80  
    0000000000000314 0000000000000000 A 0 0 4  
[17] .eh_frame PROGBITS 000000000040a298 0000a298  
    0000000000001250 0000000000000000 A 0 0 8  
[18] .init_array INIT_ARRAY 000000000040cde8 0000bde8  
    0000000000000008 0000000000000008 WA 0 0 8  
[19] .fini_array FINI_ARRAY 000000000040cdf0 0000bdf0  
    0000000000000008 0000000000000008 WA 0 0 8  
[20] .dynamic DYNAMIC 000000000040cdf8 0000bdf8  
    0000000000000200 0000000000000010 WA 6 0 8  
[21] .got PROGBITS 000000000040cff8 0000bff8  
    0000000000000008 0000000000000008 WA 0 0 8  
[22] .got.plt PROGBITS 000000000040d000 0000c000  
    0000000000000280 0000000000000008 WA 0 0 8  
[23] .data PROGBITS 000000000040d280 0000c280  
    0000000000000004 0000000000000000 WA 0 0 1  
  
[23] .data PROGBITS 000000000040d280 0000c280  
    0000000000000004 0000000000000000 WA 0 0 1  
[24] .bss NOBITS 000000000040d2a0 0000c284  
    000000000000173f0 0000000000000000 WA 0 0 32  
[25] .comment PROGBITS 0000000000000000 0000c284  
    0000000000000059 0000000000000001 MS 0 0 1  
[26] pydata PROGBITS 0000000000000000 0000c2dd  
    000000000067d615 0000000000000000 0 0 1
```

To jo **pydata** section hai isme **byte code** hai. usko extract krne ke liye. hum ek tool use karenge. Jiska nam **objcopy** hai.

```
→ rev objcopy --dump-section pydata=pydata.data crackme8
```

--dump-section → means section ko dump karna hai.

pydata=pydata.data → means pydata section uske bad kis nam se save krna hai
pydata.data uske bad file ka nam.

```
→ rev objcopy --dump-section pydata=pydata.data crackme8
→ rev ls
crackme  crackme1.c  crackme2.c  crackme3.c  crackme4.c  crackme5.c  crackme7    crackme8    crackme.c
crackme1 crackme2    crackme3    crackme4    crackme5    crackme6    crackme7.cpp  crackme8.py  pydata.data
→ rev
```

To yha pr **pydata** ke andar only **byte code** hi nhi hota hai uske sath bahut sara code aur bhi hota hai.

```
→ rev file pydata.data
pydata.data: zlib compressed data
→ rev
```

Uske liye hum phle se bane ek script ka use lenge. Jiske liye google pr jana hai.
pyinstxtractor likhna hai.

pyinstxtractor - Google Search

pyinstxtractor

All Videos News Shopping Maps More

About 5,610 results (0.36 seconds)

<https://github.com/extremecoders-re/pyinstxtractor>

extremecoders-re/pyinstxtractor: PyInstaller Extractor - GitHub

PyInstaller Extractor is a Python script to extract the contents of a PyInstaller generated

Sbse phla github ka link visit krte hai. aur **pyinstxtractor.py** file khol lete hai.

pyinstxtractor/pyinstxtractor

extremecoders-re / pyinstxtractor Public

Code Issues Pull requests Discussions Wiki Security Insights

extremecoders-re Search for pyinstaller magic instead of using a fixed position

Latest commit b71941d 11 days ago History

2 contributors

417 lines (314 sloc) 14.5 KB

Raw Blame

```
1 """
2 PyInstaller Extractor v2.0 (Supports pyinstaller 4.5.1, 4.5, 4.4, 4.3, 4.2, 4.1, 4.0, 3.6, 3.5, 3.4, 3.3, 3.2, 3.1, 3.0, 2.1, 2.0)
3 Author : Extreme Coders
4 E-mail : extremecoders(at)hotmail(dot)com
5 Web   : https://0xec.blogspot.com
6 Date  : 26-March-2020
7 Url   : https://github.com/extremecoders-re/pyinstxtractor
```

Raw pr click krte hai. aur ise terminal pr **wget** kr lete hai.

```
https://raw.githubusercontent.com/extremecoders-re/pyinstxtractor/master/pyinstxtractor.py
CTFtime.org /... malwareunico... Binary Exploit... Nightmare path tool outline

"""
PyInstaller Extractor v2.0 (Supports pyinstaller 4.5.1, 4.5, 4.4, 4.3, 4.2, 4.1, 4.0, 3.6, 3.5, 3.4, 3.3, 3.2, 3.1, 3.0,
Author : Extreme Coders
E-mail : extremecoders(at)hotmail(dot)com
Web : https://0xec.blogspot.com
Date : 26-March-2020
Url : https://github.com/extremecoders-re/pyinstxtractor

For any suggestions, leave a comment on
https://forum.tuts4you.com/topic/34455-pyinstaller-extractor/

This script extracts a pyinstaller generated executable file.
```

```
→ rev wget https://raw.githubusercontent.com/extremecoders-re/pyinstxtractor/master/pyinstxtractor.py
Will not apply HSTS. The HSTS database must be a regular and non-world-writable file.
ERROR: could not open HSTS store at '/home/hellsender/.wget-hsts'. HSTS will be disabled.
--2021-09-21 23:31:22-- https://raw.githubusercontent.com/extremecoders-re/pyinstxtractor/master/pyinstxtractor.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.111.133, 185.199.110.133, 185.199.109.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.111.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 14870 (15K) [text/plain]
Saving to: 'pyinstxtractor.py'

pyinstxtractor.py          100%[=====] 14.52K --.-KB/s   in 0s

2021-09-21 23:31:23 (35.9 MB/s) - 'pyinstxtractor.py' saved [14870/14870]

→ rev
```

Yha pr ye download ho jayegi.

```
→ rev ls
crackme crackme1.c crackme2.c crackme3.c crackme4.c crackme5.c crackme7 crackme8 crackme.c pyinstxtractor.py
crackme1 crackme2 crackme3 crackme4 crackme5 crackme6 crackme7.cpp crackme8.py pydata.data
→ rev chmod +x pyinstxtractor.py
→ rev
```

```
→ rev chmod +x pyinstxtractor.py
→ rev python3 pyinstxtractor.py pydata.data
[+] Processing pydata.data
[+] Pyinstaller version: 2.1+
[+] Python version: 38
[+] Length of package: 6805013 bytes
[+] Found 46 files in CArchive
[+] Beginning extraction...please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
[+] Possible entry point: pyi_rth_pkgutil.pyc
[+] Possible entry point: pyi_rth_multiprocessing.pyc
[+] Possible entry point: pyi_rth_inspect.pyc
[+] Possible entry point: crackme8.pyc
[+] Found 223 files in PYZ archive
[+] Successfully extracted pyinstaller archive: pydata.data

You can now use a python decompiler on the pyc files within the extracted directory
→ rev
```

Yha pr isne sara data extract kr diya.

```
→ rev ls
crackme    crackme2   crackme3.c  crackme5  crackme7  crackme8.py  pydata.data_extracted
crackme1   crackme2.c  crackme4   crackme5.c  crackme7.cpp crackme.c  pyinstxtractor.py
crackme1.c crackme3   crackme4.c  crackme6   crackme8
→ rev cd pydata.data_extracted
→ pydata.data_extracted ls
base_library_zip libexpat.so.1      libreadline.so.8      pyimod01_os_path.pyc  pyi_rth_multiprocessing.pyc
crackme8.pyc    libffi.so.7       libssl.so.1.1      pyimod02_archive.pyc  pyi_rth_pkutil.pyc
libbz2.so.1.0   liblzma.so.5     libtinfo.so.6      pyimod03_importers.pyc PYZ-00.pyz
libcrypto.so.1.1 libmpdec.so.2     libz.so.1        pyimod04_ctypes.pyc  PYZ-00.pyz_extracted
lib-dynload    libpython3.8.so.1.0  pyiboot01_bootstrap.pyc pyi_rth_inspect.pyc
→ pydata.data_extracted
```

Yha pr humare kam ka file **crackme8.pyc** hi hai.

To ye **byte code** hai. agar ise cat kare to ye human readable format me nhi hota hai.

Ab isko wapas **Python ke code** me convert krne ke liye ek tool use karenge. Jiska nam **uncompyle6** hai.

```
→ pydata.data_extracted sudo pip3 install uncompyle6
```

To yha pr phle se hi install hai.

```
→ pydata.data_extracted sudo pip3 install uncompyle6
Requirement already satisfied: uncompyle6 in /usr/local/lib/python3.8/site-packages/uncompyle6-6.0.1-py3.8.egg
Requirement already satisfied: xdis<5.1.0,>=5.0.4 in /usr/local/lib/python3.8/site-packages/xdis-5.0.4-py3.8.egg
Requirement already satisfied: spark-parser<1.9.0,>=1.8.9 in /usr/local/lib/python3.8/site-packages/spark-parser-1.8.9-py3.8.egg
Requirement already satisfied: click in /usr/local/lib/python3.8/site-packages(click-8.0.1-py3.8.egg)
Requirement already satisfied: six>=1.10.0 in /usr/local/lib/python3.8/site-packages/six-1.10.0-py3.8.egg
WARNING: Running pip as the 'root' user can result in broken permissions. It is recommended to use a virtual environment instead: https://pip.pypa.io/en/stable/user_guide/virtual-environments/
→ pydata.data_extracted
```

Ab hum tool ko run krte hai.

```
→ pydata.data_extracted uncompyle6 crackme8.pyc > crackme8.py
→ pydata.data_extracted subl crackme8.py
```

Aur ye **byte code** ko bilkul exact python ke original code me convert kr dega. Yha pr **C** ke jaisa nhi hota ki **psudocode** me convert kare.

Yha hume bilkul **original code** milta hai.

```
crackme8.py
1 # uncompyle6 version 3.7.4
2 # Python bytecode 3.8 (3413)
3 # Decompiled from: Python 3.8.10 (default, Jun 2 2021, 10:49:15)
4 # [GCC 9.4.0]
5 # Embedded file name: crackme8.py
6 import sys
7 passwd = input('Enter Password - ').replace('\n', '')
8 if 'cj{bj^ewcjfxj' == lambda passwd: ''.join((chr((ord(i) ^ 9) - 4 + 2) for i in
9     passwd)):
10    sys.exit('G00D J08!!!')
11 else:
12    sys.exit('W04K H4RD!!!')
13 # okay decompiling crackme8.pyc
```

```
in passwd))(passwd):
```

Ye python ke original bs yha minor sa changes hai. lekin code ke logic pr koi impact nhi pda.

```
lambda passwd: ''.join((chr((ord(i) ^ 9) - 4 + 2) for i in passwd))(passwd)
```

Ab hum bs is code ko samjhna hai.

```
def abc(passwd):
    ''.join((chr((ord(i) ^ 9) - 4 + 2) for i in passwd))
abc(passwd)
```

Yha humne thoda simplify kiya hai.

Hum ise thoda aur simplify kr lete hai.

```
def abc(passwd):
    a = ""
    for i in passwd:
        a += (chr((ord(i) ^ 9) - 4 + 2))

abc(passwd)
```

Yha hum is code ko reverse krne ka code likhenge.

```
b = ""  
string = 'cj{bj^ewcjfxj'  
for i in string:  
    b+= chr(((ord(i)-2)+4)^9)  
print(b)
```

Yha is code ko reverse order me likhenge. Jaise ki 2 plus ho rha tha to 2 minus hogा. 4 minus ho rha tha to 4 plus hogा. Aur sbse last me 9 xor ho rha tha to 9 xor hi hogा. Kyoki xor ka ek rule hota hai. aur last me sbko character me convert kr lenge.

xor ka rule.

$$\begin{aligned}a \wedge b &= c \\a \wedge c &= b \\c \wedge b &= a\end{aligned}$$

Ab hum program ko run krte hai. aur password get krte hai.

```
→ pydata.data_extracted python3  
Python 3.8.10 (default, Jun 2 2021, 10:49:15)  
[GCC 9.4.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> b = ""  
>>> string = 'cj{bj^ewcjfxj'  
>>> for i in string:  
...     b+= chr(((ord(i)-2)+4)^9)  
...  
>>> print(b)  
letmeinplease  
>>>
```

Let's check the password.

```
→ rev ./crackme8
Enter Password - letmeinplease
GOOD JOB!!
→ rev
```

```
#####
#####
```

Automated Reverse Engineering Using Python Angr :

Challenge - 9

```
→ rev ./crackme9
Hotel Orlando Door Puzzle v1
-----
This puzzle, provided by Hotel Orlando, is in place to give the bellhops enough time to get your luggage
We have really slow bellhops and so we had to put a serious _time sink_ in front of you.
Have fun with this puzzle while we get your luggage to you!

-Hotel Orlando Bellhop and Stalling Service

Your guess, if you would be so kind:
aaaa
Sadly, that is the incorrect key. If you would like, you could also sit in our lobby and wait.
```



Description:- Yha pr ye ek reverse engineering ka **CTF** hai jisme ek hotel ke room ka door kholna hai. door kholne ke liye hume **flag** ki jarurat padegi. Jisse door khola ja ske. Aur flag pane ke liye hume ise reverse engineer krna pdega.

CodeBrowser: crackme:/crackme9

File Edit Analysis Graph Navigation Search Select Tools Window Help

Program Trees Listing: crackme9 Decompiler

Symbol Tree

Data Type Manager

Console - Scripting

The screenshot shows the CodeBrowser interface with the project 'crackme9' selected. The assembly listing window displays the start of the program with the instruction 'assume DF = 0x0'. The decompiler window shows 'No Function'. The symbol tree window lists various symbols, with 'main' highlighted in pink.



Decompile: main - (crackme9)

```

1
2 undefined8 main(void)
3
4 {
5     int iVar1;
6     long in_FS_OFFSET;
7     undefined local_58 [72];
8     long local_10;
9
10    local_10 = *(long *)(in_FS_OFFSET + 0x28);
11    puts("Hotel Orlando Door Puzzle v1");
12    puts(-----);
13    puts(
14        "This puzzle, provided by Hotel Orlando, is in place to give the bellhops a
15        our luggage to you."
16    );
17    puts("We have really slow bellhops and so we had to put a serious _time sink_ in
18    puts("Have fun with this puzzle while we get your luggage to you!");
19    puts("\n\t-Hotel Orlando Bellhop and Stalling Service\n");
20    puts("Your guess, if you would be so kind: ");
21    __isoc99_fscanf(stdin,&DAT_001031b6,local_58);
22    iVar1 = check_flag(local_58);
23
24    __isoc99_fscanf(stdin,&DAT_001031b6,local_58);
25    iVar1 = check_flag(local_58);
26    if (iVar1 == 1) {
27        puts("I see you found the key, hopefully your bags are in your room by this point.");
28    }
29    else {
30        puts(
31            "Sadly, that is the incorrect key. If you would like, you could also sit
32            it.");
33    }
34    if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
35        /* WARNING: Subroutine does not return */
36        __stack_chk_fail();
37    }

```

Ab hum **main** function ko pdna suru karenge.

```
18 puts("\n\t-Hotel Orlando Bellhop and Stalling Service\n");
19 puts("Your guess, if you would be so kind: ");
20     isoc99_fscanf(stdin,&DAT_001031b6,input);
21 return = check_flag(input);
22 if (return == 1) {
23     puts("I see you found the key, hopefully your bags are in your ro
24 }
25 else {
26     puts(
27         "Sadly, that is the incorrect key. If you would like, you cou
28         it.");
29 }
30 if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
31     /* WARNING: Subroutine does not return */
32     __stack_chk_fail();
33 }
```

Yha humne kuchh variables ka nam rename krke meaningful rkh diya hai.

Ab hum **check_flag()** function ke andar chalte hai.

Cf Decompiler: check_flag - (crackme9)

```
1
2 undefined8 check_flag(char *input)
3
4 {
5     size_t input_len;
6     undefined8 uVar1;
7
8     input_len = strlen(input);
9     if (input_len == 29) {
10         if (input[19] == '6') {
11             input[6] = input[6] & '\x03';
12             if (input[0x10] == 'n') {
13                 input[0x14] = input[0x14] + -8;
14                 input[0x1a] = input[0x1a] + -6;
15                 if (input[0xd] == 'r') {
16                     if (input[0x14] == '%') {
17                         if (input[0xf] == 'n') {
18                             if (input[10] == 'p') {
19                                 input[0x10] = input[0x10] + '\a';
20                                 if (input[0x10] == 'u') {
21                                     if (input[3] == '{') {
22                                         input[9] = input[9] + '\x01';
23                                         if (input[0x13] == '6') {
```

Decompile: check_flag - (crackme9)

```
input[7] = input[7] + '\b';
input[8] = input[8] + '\x04';
input[0xd] = input[0xd] + -10;
input[0x1a] = input[0x1a] + '\a';
if (input[0xe] == 'u') {
    if (input[0x16] == 't') {
        if (input[2] == 'n') {
            input[0xf] = input[0xf] + 3;
        if (input[0x1a] == 'm') {
            input[6] = input[6] + '\x05';
            input[0x13] = input[0x13] + -10;
            input[0x1c] = input[0x1c] + '\x04';
            if (input[6] == 'k') {
                input[3] = input[3] + -9;
                if (input[6] == 'k') {
                    input[0x11] = input[0x11] + -5;
                    input[0x11] = input[0x11] + -6;
                    if (input[1] == 's') {
                        if (input[0x12] == 'e') {
                            input[4] = input[4] + '\x05';
                            if (input[6] == 'k') {
                                if (input[0xc] == '*') {
                                    if (input[2] == 'n') {
```

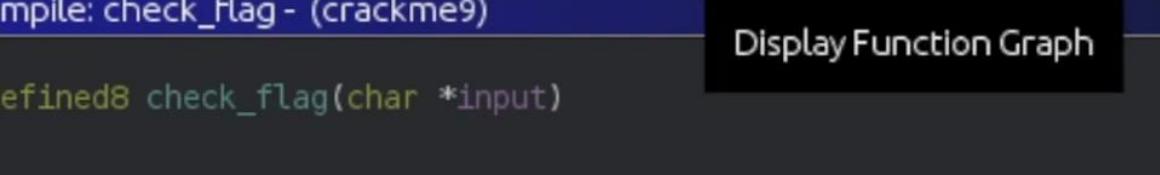
```
if (input[2] == '\n') {
    if (input[0x1b] == 'y') {
        if (input[0x1c] == 'y') {
            if (input[0x13] == '*') {
                if (input[8] == 'f') {
                    if (input[0x1c] == 'y') {
                        if (input[9] == ',') {
                            if (input[2] == '\n') {
                                if (input[0xd] == 'd') {
                                    if (input[0x16] == 't') {
                                        if (input[5] == '8') {
                                            if (input[7] == 'k') {
                                                if (input[9] == ',') {
                                                    if (input[1] == 's') {
                                                        if (*input == 'k') {
                                                            if (input[8] == 'f') {
                                                                if (input[1] == 's') {
                                                                    if (input[0x15] == 'q') {
                                                                        if (input[0x1b] == 'q') {

```

Jaisa ki hum dekh skte hai. yha pr bahut sare **if-else** conditions hai. agar hum paper pen lekar solve karenge to bahut sara time lag jayega.

To hum python ka help lenge. Iske 6-7 line ka humari ye problem solve kr dega.

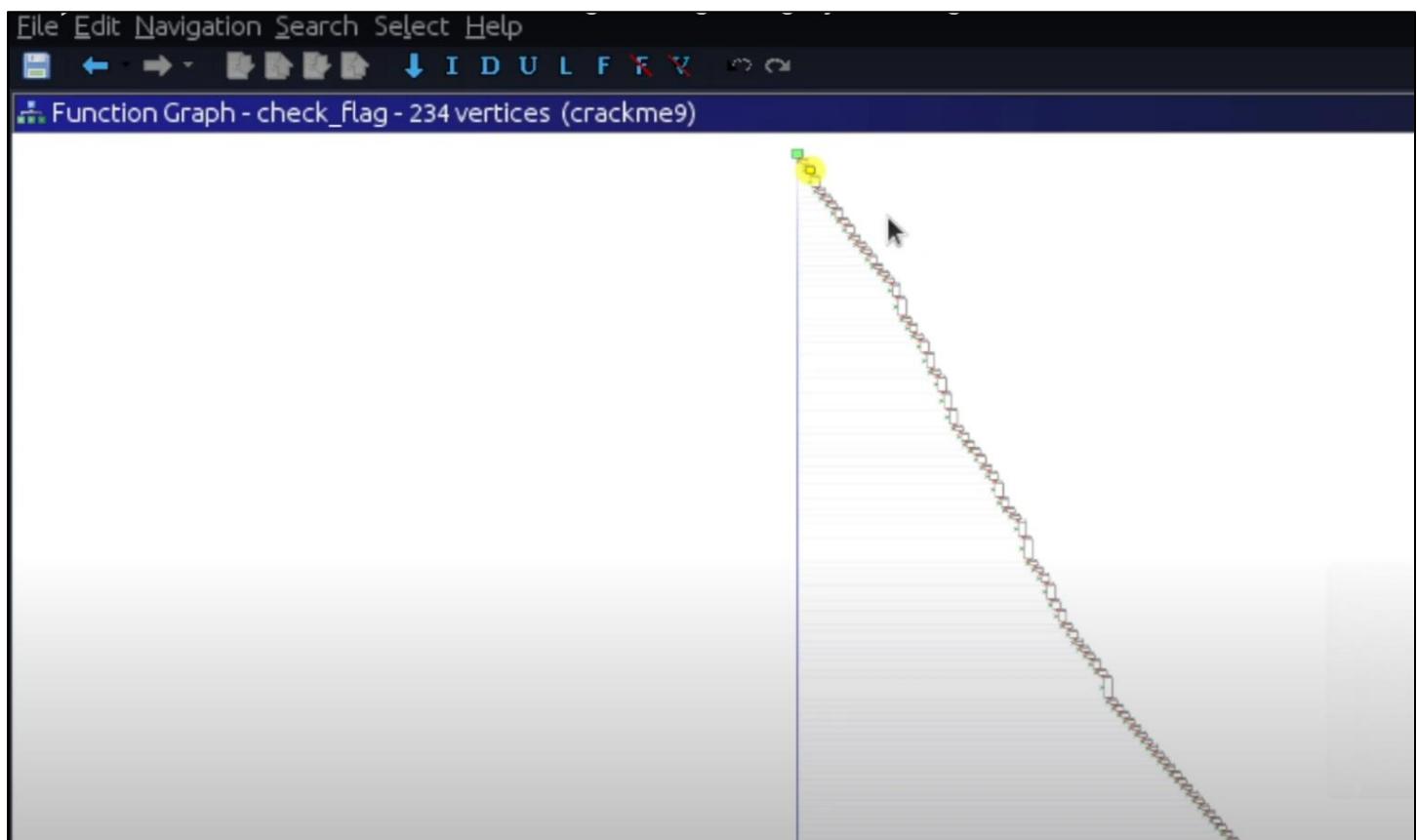
Yha hum **angr** module ka use karenge. Ye automated me help krta hai **symbolic execution** ka help leta hai.



The screenshot shows the Immunity Debugger interface with the title bar "Cj Decompile: check_flag - (crackme9)". The assembly code decompiled by Immunity Debugger is as follows:

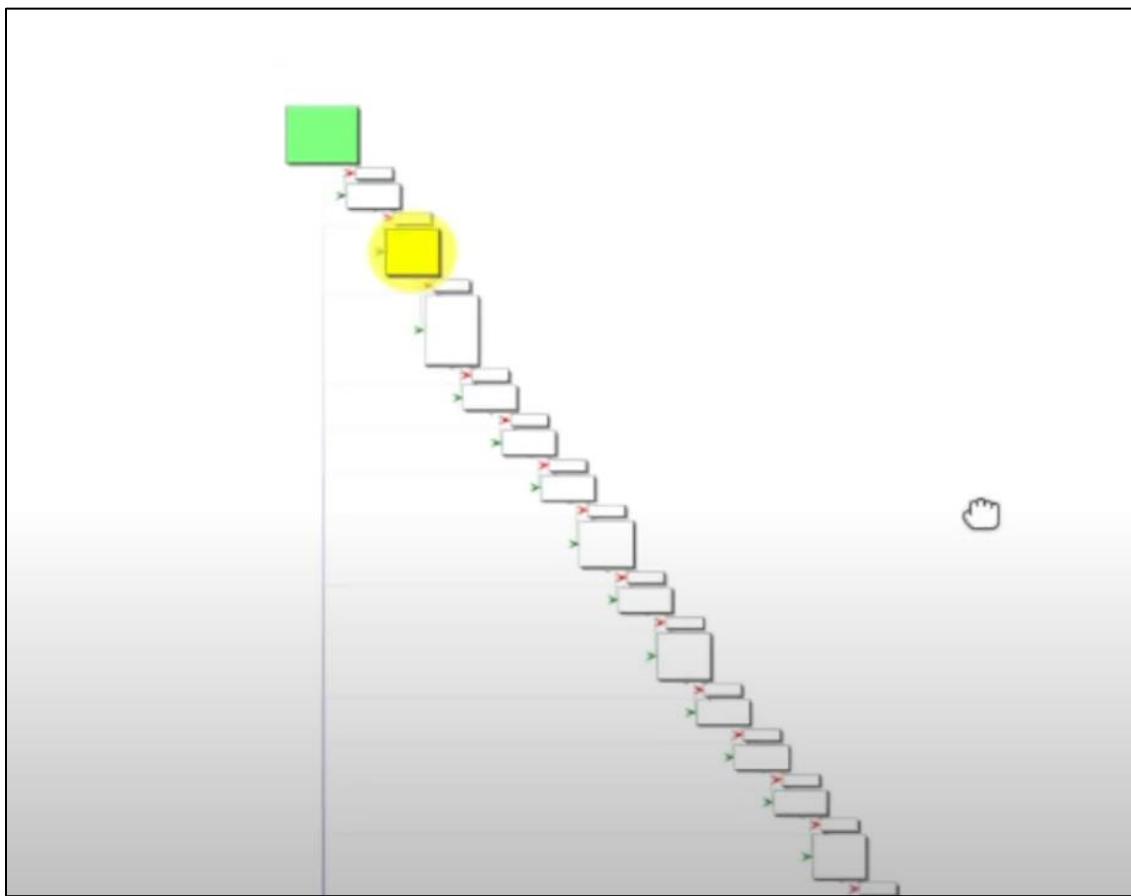
```
1 undefined8 check_flag(char *input)
2
3 {
4     size_t input_len;
5     undefined8 uVar1;
6
7     input_len = strlen(input);
8     if (input_len == 29) {
9         if (input[19] == '6') {
10            input[6] = input[6] + '\x03';
11            if (input[0x10] == '\n') {
12                input[0x11] = input[0x11] - 0x10;
13            }
14        }
15    }
16 }
```

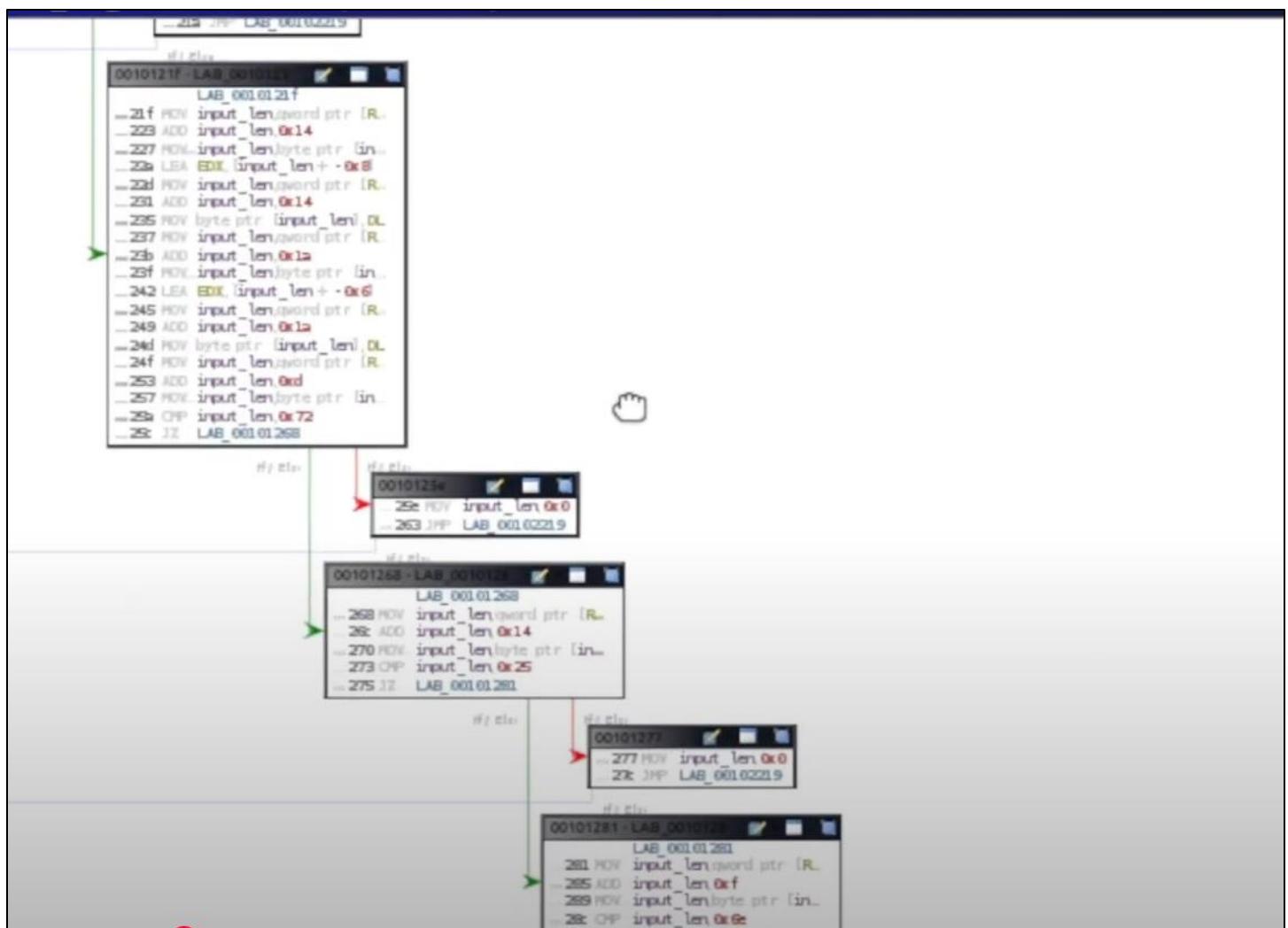
Symbolic execution smjhne ke liye hum ghidra pr chalte hai. aur flow chart ya function graph jo bhi hai. usko open krte hai.



Ab hum yha dekh skte hai. ki ek accha sa graph bn gya hai.

Hum ise zoom krte jate hai.





Yha hum dekh skte hai. ki kitne sare branches hai.



Upar length compare kr rha hai agar itna hai to yha jayega, agar itna nhi hua to kahan jayega.

Yha pr python **input** me kuchh bhi value nhi put kr deta. Wo har branch pr jata hai. aur uska ek **equation** banata hai. ki agar mai isme ye wala **symbolic value** put karunga to kis branch pr jayega.

Equation ko solve ye **z3** ki help se krtा hai. **z3** ek **library** hai. jise microsoft ne banaya hai **theorems** ko proof krne ke liye. **equations** ko solve krne ke liye.

To **z3** ki help se equations ko solve krta hai. aur sbke andar jayega aur check karega ki kya value put karunga to is branch me jaunga. Jitne possible routes hote hai. to sabke andar wo jata hai.

Yha hum **bruteforce** nhi kr rhe hai. yha pr wo **intelligently equation** banakr kr solve kr rha hai. ki kya value rkhunga to mai andar ja skta hun aur kya value rkhunga to mai bahar aa skta hun.

Kuchh logo claim kiya ki **AI(Artificial Intelligent)** hai. lekin **angr** walo aisa kuchh nhi likha hai iske bare me.

To yha hum angr ko batayenge ki jb tk tujhe correct password nhi mil jaye tu sare branches pr jata ja. Jaise hi correct password mil jaye wahi ruk ja aur bta ki tune aisa kaun sa input dala ki correct password mil gya.

```
crackme9.py
1 #!/usr/bin/python3
2
3 import angr
4
5 project = angr.Project('./crackme9')
```

Yha humne angr ko import kr liya.

Uske bad hume project banana hota hai. jaise ghidra ko aise hi nhi start krte hai. phle project banana hota hai.

Project function ka **P** capital hota hai. usme humen **binary ka path** dena hota hai as argument.

```
crackme9.py
1 #!/usr/bin/python3
2
3 import angr
4
5 project = angr.Project('./crackme9')
6 entry = project.factory.entry_state()
```

Yha pr humen batana padega ki kahan se start krna hai. to hume address dene ki jarurat nhi hai. automatic pta laga leta hai entry point kya hai. **main** function se start krna hai ya **_start** se automatic pta lga leta hai.

Yha **entry_state()** function pta laga leta hai. ki kahan se entery hai program ki. Jo bhi entery point milega **entry** variable ke andar store kr dega.

Agar hume khud se address dena hai entry point ka to hum aise de skte hai.

```
|entry = project.factory.entry_state(addr=0x4567)|
```

Hex ke format me.

Ab hum ek manager banayenge to input output ko manage karega. Jo binary run karega binary ka output aa rha hai. binary ka input hai wo bhi wahi manage karega. Kyoki itne sare branches hai iske andar kaun kahan ja rha hai sb manage karega. Kya output aa rha hai kya input ja rha hai.

```
| 7 manager = project.factory.simgr(entry)|
```

To yha pr project ke andar factory hota hai. aur manager factory me hota hai jise hum khte hai **simulation manager**.

simgr() function me hum entry put karenge kahan se run krna hai.

```
| 8 manager.explore(find=0x004022c1, avoid=0x004022cf)|
```

Yha pr explore function pure binary ko explore karega. Har routes pr jayega.

To yha pr **find =** ke andar hum koi string bhi de skte hai. ya address de skte hai. address sahi rhta hai. to is address pr pahuch kr ruk jayega. Analysis krna band kr dega.

Iske bad kis adres ko avoid krna hai. **avoid=** ke andar use de dete hai.

To hume kis jagah pr jana hai. iske liye hum ghidra me chalte hai.

```
C:\Decompile: main - (crackme9)
13 | puts(
14 |     "This puzzle, provided by Hotel Orlando, is in place to give
15 |         );
16 | puts("We have really slow bellhops and so we had to put a serious
17 | puts("Have fun with this puzzle while we get your luggage to you
18 | puts("\n\t-Hotel Orlando Bellhop and Stalling Service\n");
19 | puts("Your guess, if you would be so kind: ");
20 | __isoc99_fscanf(stdin,&DAT_001031b6,input);
21 | return = check_flag(input);
22 | if (return == 1) {
23 |     puts(*I see you found the key, hopefully your bags are in your
24 | }
25 | else {
26 |     puts(
27 |         "Sadly, that is the incorrect key. If you would like, you can
28 |         );
29 | }
```

Yha hume is puts pr jana hai. jahan "**I see you found the key**" likha hai. to hum isi ka address de denge ki tu jaise hi yha pr aaye yhi ruk ja. Aur jo bhi input dala tha print kr de.

To puts pr click karenge. Aur dissassembly me jayenge aur iska address copy kr lenge.

The screenshot shows the Ghidra interface with two panes. The left pane, titled 'Listing: crackme9', displays assembly code for the main function. The right pane, titled 'Decompile: main -', shows the corresponding C decompilation. A red box highlights the instruction at address 001022c1, which is a call to the puts function.

```

Listing: crackme9
00102291 70 00 00    LLR      TDA=Input, LDR=Output
        b0
00102295 48 8d 35    LEA      RSI,[DAT_001031b6]
        1a 0f 00
        00
0010229c 48 89 c7    MOV      RDI,RAX
0010229f b8 00 00    MOV      EAX,0x0
        00 00
001022a4 e8 d7 ed    CALL    <EXTERNAL>::__isoc99_fscanf
        ff ff
001022a9 48 8d 45    LEA      RAX=>input,[RBP + -0x50]
        b0
001022ad 48 89 c7    MOV      RDI,RAX
001022b0 e8 f4 ee    CALL    check_flag
        ff ff
001022b5 83 f8 01    CMP    return,0x1
001022b8 75 0e        JNZ    LAB_001022c8
001022ba 48 8d 3d    LEA      RDI,[s_I_see_you_found_the_k
        ff 0e 00
        00
001022c1 e8 ca ed    CALL    <EXTERNAL>::puts
        ff ff
001022c6 eb 0c        JMP    LAB_001022d4

```

```

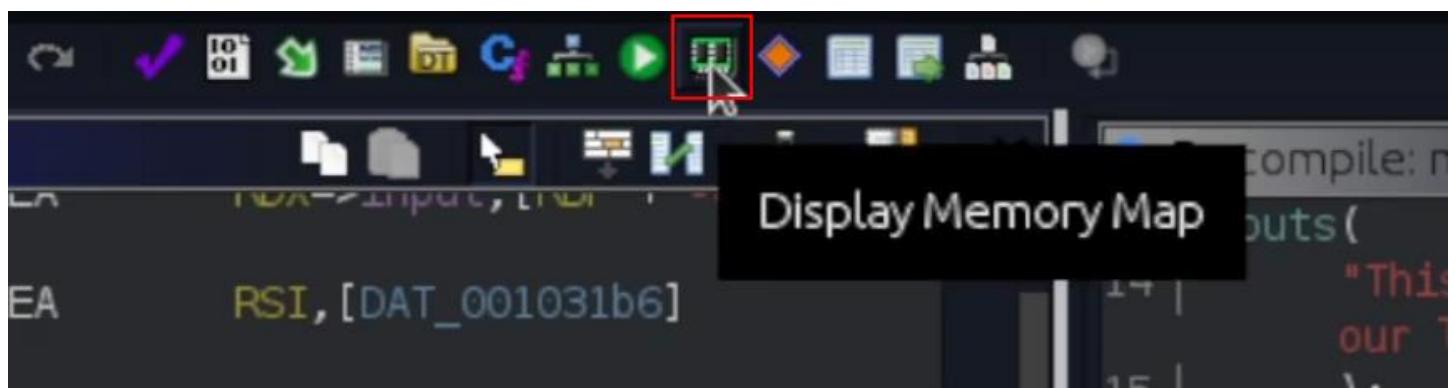
Decompile: main -
13 puts(
14     "This puz;
15     our lugga
16 );
17 puts("We have
18 puts("Have fu
19 puts("\n\tHo
20 puts("Your gu
21 __isoc99_fscn
22 return = chec
23 if (return ==
24     puts("I see
25 } else {
26     puts(
27         "Sadly,
28         it."
29 );
30 if (local_10
31     __stack_chk
32 , _stack_chk

```

Lekin jaisa ki hum jante hai. binary ka addreass randomise hota rhta hai. yha jb **angr** run hogा to uske **puts** ka address dusra hogा aur **ghidra** ke **puts** ka address dusra kyoki memory address to randomise hoti rhti hogा.

To iske liye hum **ghidra** ka **base address** fir se set kr dete hogा jo **angr** ka hota hogा.

Uske liye ye chip wale icon pr click krte hogा.



Uske bad house ka icon aa rha hogा. Set image base.

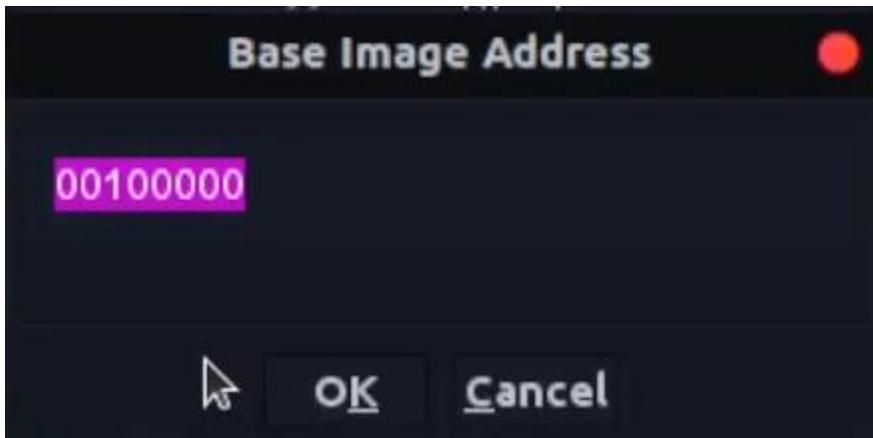
Memory Map [CodeBrowser: crackme:/crackme9]

File Edit Help

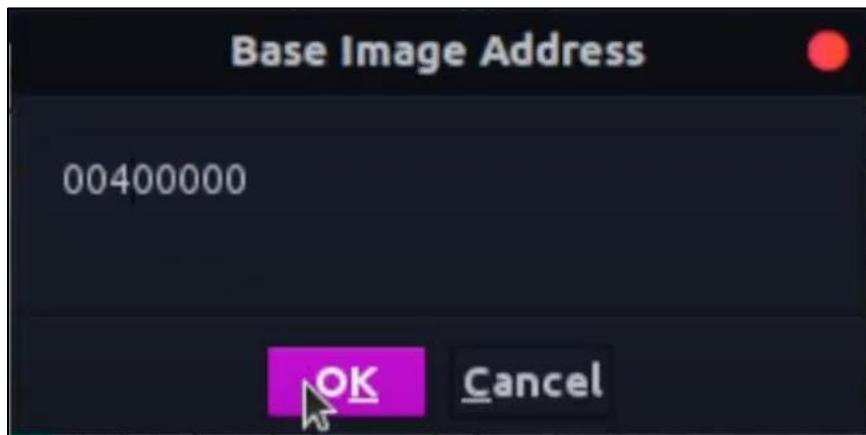
Memory Map - Image Base: 00100000

Memory Blocks

Name	Base	End	Length	R	W	X	Volat...	Ov...	Type	Initializ...	Byte Sow...	Source	Comm...	Set Image Base
segme...	00100000	00100317	0x318	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>	File: cra...	Elf Loa...	Loadab...	
interp	00100318	00100333	0x1c	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>	File: cra...	Elf Loa...	SHT_PR...	
note.g...	00100338	00100357	0x20	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>	File: cra...	Elf Loa...	SHT_N...	
note.g...	00100358	0010037b	0x24	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>	File: cra...	Elf Loa...	SHT_N...	
note.A...	0010037c	0010039b	0x20	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>	File: cra...	Elf Loa...	SHT_N...	
gnu.ha...	001003a0	001003c7	0x28	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>	File: cra...	Elf Loa...	SHT_G...	
dynsym	001003c8	001004cf	0x108	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>	File: cra...	Elf Loa...	SHT_DY...	
dynstr	001004d0	00100593	0xc4	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>	File: cra...	Elf Loa...	SHT_ST...	
gnu.ve...	00100594	001005a9	0x16	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>	File: cra...	Elf Loa...	SHT_G...	
gnu.ve...	001005b0	001005ef	0x40	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>	File: cra...	Elf Loa...	SHT_G...	
rela.dyn	001005f0	001006c7	0x7d8	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>	File: cra...	Elf Loa...	SHT_RF...	



Isko change krke 4 lakh kr dena hai.



Yha hume 00400000 isliye kiya kyoki **angr** jb bhi run hota hai. to **base address** 00400000 leta hai **by default**.

The screenshot shows the Immunity Debugger interface. On the left, the assembly listing shows a sequence of instructions, with the instruction at address 004022c1 highlighted with a red box. On the right, the decompiled C code shows a puts() function call at line 13.

```

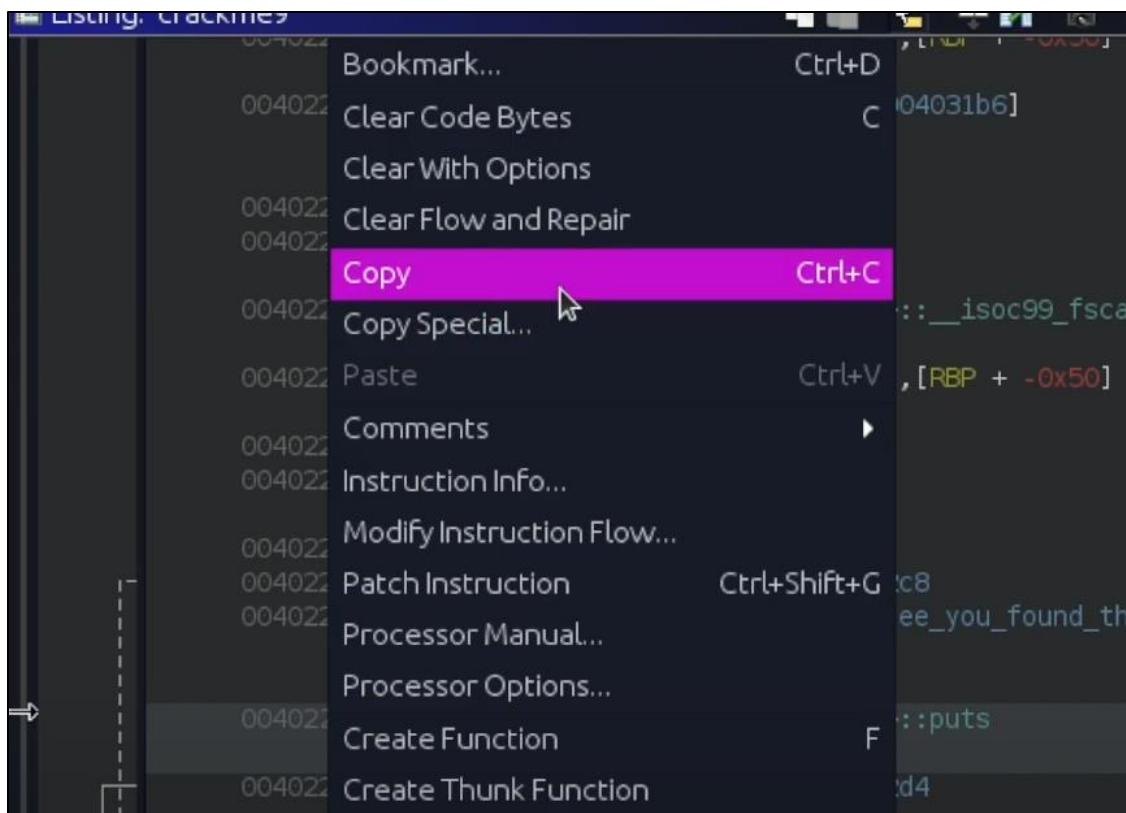
Listing: crackme9
00402201 48 8d 35      LEA    RSI,[DAT_004031b6]
00402205 b0             ; 
00402209 48 8d 35      LEA    RSI,[DAT_004031b6]
0040220d 1a 0f 00      MOV    RDI,RAX
00402211 00 00          ; 
00402215 00 00          ; 
00402219 e8 d7 ed      CALL   <EXTERNAL>::__isoc99_fscanf
0040221d ff ff          ; 
00402221 48 8d 45      LEA    RAX=>input,[RBP + -0x50]
00402225 b0             ; 
00402229 48 89 c7      MOV    RDI,RAX
0040222d 0e f4 ee      CALL   check_flag
00402231 ff ff          ; 
00402235 83 f8 01      CMP    return,0x1
00402239 75 0e          JNZ    LAB_004022c8
0040223d 48 8d 3d      LEA    RDI,[s_I_see_you_found_the_k
00402241 ff 0e 00      ; 
00402245 00 00          ; 
00402249 e8 ca ed      CALL   <EXTERNAL>::puts
0040224d ff ff          ; 
00402251 eb 0c          JMP    LAB_004022d4

Decompile: main - (crackme9)
13 | puts(
14 |     "This puzzle,
15 |     our luggage to
16 | );
17 | puts("We have real
18 | puts("Have fun wit
19 | puts("\n)t-Hotel C
20 | puts("Your guess,
21 | __isoc99_fscanf(st
22 | return = check_flag
23 | if (return == 1) {
24 |     puts("I see you
25 | } else {
26 |     puts(
27 |         "Sadly, that
28 |         it."
29 |     );
30 |     if (local_10 != *(

33 | } _stack_chk_fail

```

Iske bad puts ka address copy kr lena hai.



Aur avoid wala address bhi copy kr lete hai. jo **incorrect key** ka hai.

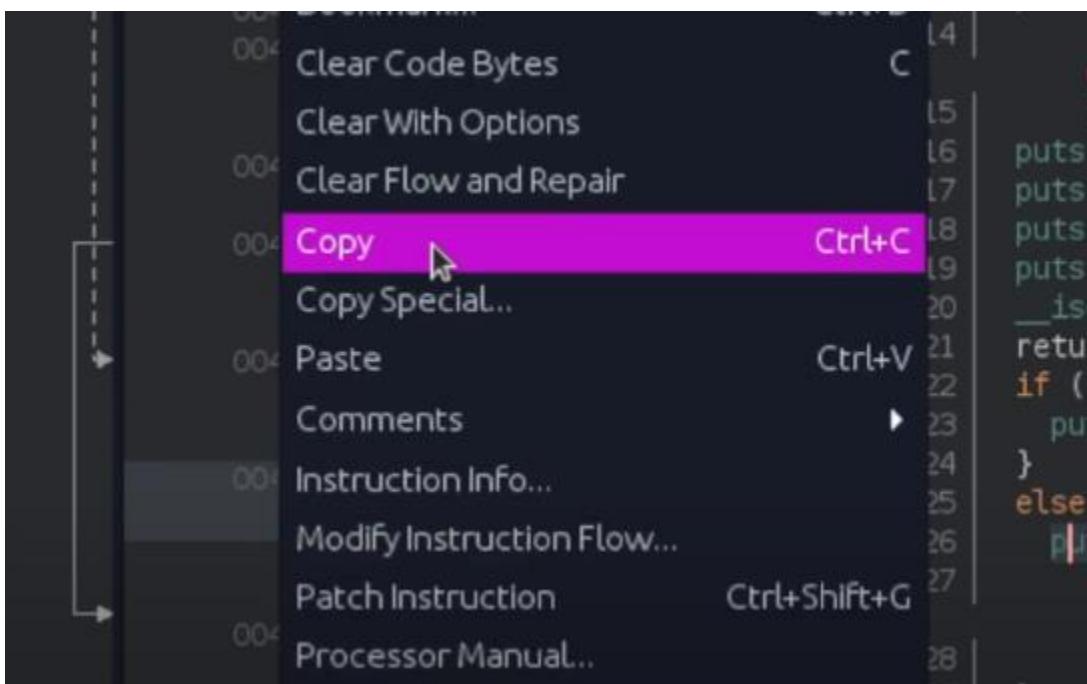
The screenshot shows the Immunity Debugger interface. On the left is the assembly view, and on the right is the decompiled C-like code. The assembly code includes instructions like JNZ, LEA, CALL, and JMP. The decompiled code shows puts() calls and an if-else block. A red box highlights the instruction at address 004022cf, which is a CALL instruction.

```

00402203 83 3d 00          JNZ
004022ba 48 8d 3d          LEA
    ff 0e 00
    00
004022c1 e8 ca ed          CALL
    ff ff
004022c6 eb 0c          JMP
    LAB_004022c8
004022c8 48 8d 3d          LEA
    41 0f 00
    00
004022cf e8 bc ed          CALL
    ff ff
    LAB_004022d4
004022d4 b8 00 00          MOV
    00 00

Decompile: main - (crackme9)
13 | puts(
14 |     "This puzzle, provided by Hotel Orlando Bellhop and Stewardess
15 |     our luggage to you."
16 | );
17 | puts("We have really slow bellhops and
18 | puts("Have fun with this puzzle while waiting for your room
19 | puts("\n\t-Hotel Orlando Bellhop and Stewardess
20 | puts("Your guess, if you would be so kind
21 | __isoc99_fscanf(stdin,&DAT_004031b6,input);
22 | return = check_flag(input);
23 | if (return == 1) {
24 |     puts("I see you found the key, hopefully
25 | else {
26 |     puts(
27 |         "Sadly, that is the incorrect key
28 |     );

```



Aur avoid wale me paste kr denge.

Ab yha pr hum if-else se check kr lete hai. ki key mila ki nhi. Agar mila to print kr do agar nhi mila to “**not found**” show kr do. Otherwise nhi milega to ye error throw kr dega.

```

9  if manager.found:
10     print(manager.found[0].posix.dumps(0))
11 else:
12     print("Not Found!")

```

Yha ek binary ke bahut sare key ho skte hai. isliye found ki humne index value set kr di [0] means first key.

Aur dumps me **(0)** isliye hai. jaisa ki jante hai assmbly me **stdin (standard input)** ki value **0** hoti hai.

```
crackme9.py x
1 #!/usr/bin/python3
2
3 import angr
4
5 project = angr.Project('./crackme9')
6 entry = project.factory.entry_state()
7 manager = project.factory.simgr(entry)
8 manager.explore(find=0x004022c1, avoid=0x004022cf)
9 if manager.found:
10     print(manager.found[0].posix.dumps(0))
11 else:
12     print("Not Found!")
```

Ab hum is program ko run krte hai.

Let's check the flag correct or not.

```
→ rev ./crackme9
Hotel Orlando Door Puzzle v1
-----
This puzzle, provided by Hotel Orlando, is in place to give the bellhops enough time to get your
We have really slow bellhops and so we had to put a serious _time sink_ in front of you.
Have fun with this puzzle while we get your luggage to you!

-Hotel Orlando Bellhop and Stalling Service

Your guess, if you would be so kind:
sun{b3llh0p5-rump1n6-qu1ckly}
I see you found the key, hopefully your bags are in your room by this point.
→ rev
```

References:

https://www.tutorialspoint.com/assembly_programming/index.htm

<https://asmtutor.com/#lesson1>

<https://www.nasm.us/>

<https://github.com/NationalSecurityAgency/ghidra/releases>

<https://zydis.re/>

<https://github.com/mschwartz/assembly-tutorial?tab=readme-ov-file#linux-syscalls>

<https://github.com/mytechnotalent/Reverse-Engineering>