Anti-Reversing Techniques

By:-
Sukriti Singh(116406581)

Agenda

- What is meant by Anti-Reversing Techniques?
- Techniques Discussed
- Premise of the binary considered
- Explanation and Demo of the techniques
- Analysis of a Key Generator and Impact of Code Obfuscation
- Q & A

What is meant by Anti-Reversing Techniques?

- Anti-reversing techniques are techniques deployed which are meant to make the reverse engineering process difficult for a hacker or any malicious user.
- The main goal of various anti-reverse engineering techniques is simply to complicate the process of reversing as much as possible.
- An attacker can use the disassembly of a binary in order to get an insight of the logic of the code as well as reverse engineer their way into getting crucial information.
- In this project, if we try to understand some anti-reversing techniques in order to make our binaries difficult to crack and make as less exploitable as possible.

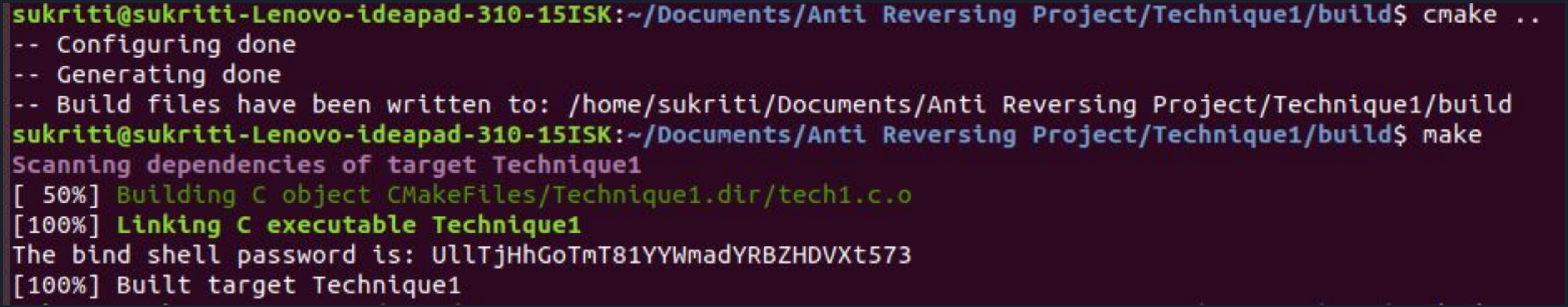
Techniques Discussed

- Avoiding the “-g” flag while compiling
- Use of -S flag to strip away the symbol table
- Use of “fvisibility = hidden”
- Use of LD_PRELOAD
- Changes to elf header
- Use of stack strings
- XOR stack strings

Premise Of The Binary Considered

- The project centers around the obfuscation of a bind shell.
- A Bind shell is a type of shell in which the target machine opens up a communication port or a listener on the victim machine and waits for an incoming connection.
- The attacker then connects to the victim machine's listener which then leads to code or command execution on the server.
- The bind shell requires a password to access the shell.
- In this project, we explore ways to hide as well as recover the shell's password in an iterative manner.
- We are using CMake to take care of the compilation and the build process. After every build, the process generates a new 32-bit password required to access the shell.
- If an attacker attempts to connect to the bind shell through netcat for example, and is able to enter the correct password, the attacker can gain the shell access to the victim's system.

Password Generated after Running the Build Process



```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Technique1/build$ cmake ..
-- Configuring done
-- Generating done
-- Build files have been written to: /home/sukriti/Documents/Anti Reversing Project/Technique1/build
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Technique1/build$ make
Scanning dependencies of target Technique1
[ 50%] Building C object CMakeFiles/Technique1.dir/tech1.c.o
[100%] Linking C executable Technique1
The bind shell password is: UllTjHhGoTmT81YYWmadYRBZHDVXt573
[100%] Built target Technique1
```

The -g Flag

- When we use the “-g” flag while compiling a C code with gcc, the -g option instructs the compiler to include debugging information in the binary.
- This generated several section headers with debugging information such as .debug_info.
- The contents of .debug_info can be viewed by using the “-dwarf=info” flag with objdump.
- Useful information from the point of view of a debugger is contained in the .debug_info section such as full path of the source file, the full path of the compilation directory and even exact line numbers where certain variables are declared.

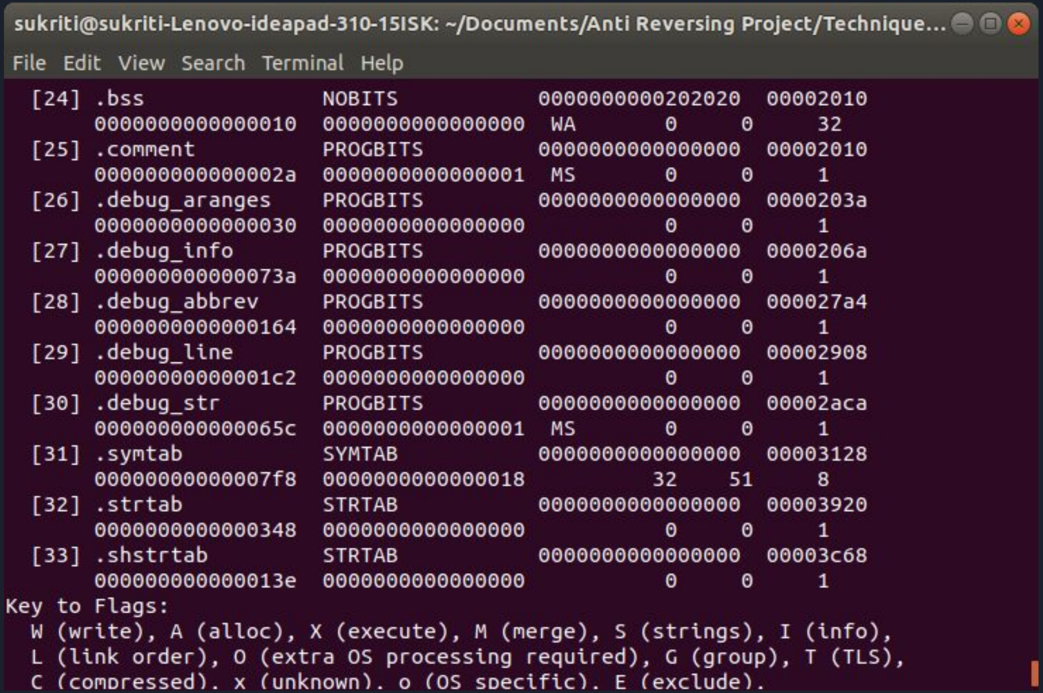
Steps to exploit the presence of the -g flag

https://drive.google.com/open?id=1t1pdsCC-gF1HRbFmMwIpFLeSgsKGDALD

Steps to exploit the presence of the -g flag

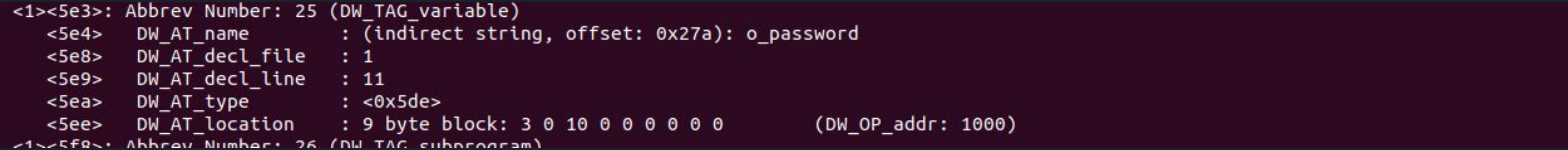
- Compile the C file with the “-g” flag which will now include the debugging information and we can see the .debug_info section in the binary’s section headers.
- By using objdump to view .debug_info, we are able to find out that there is a certain variable called o_password(might be the actual password) which is stored at the virtual address of 0x1000.
- If convert the virtual address into a file offset, we can extract the contents of o_password using hexdump. In order to achieve this, we find the program header the address falls in.
- The virtual address for o_password falls in the range for the first LOAD segment which covers 0x000000 to 0x01190. The first LOAD segment starts at the file offset of 0.
- Hence we are able to use hexdump to find the bind shell’s password at 0x1000.
- Alternatively, now that we know about the o_password variable from examining the objdump, we can even use gdb to print the values of the variable.

Step 1. Running readelf -S ./Techinique1 to display section headers showing debug sections


sukriti@sukriti-Lenovo-ideapad-310-15ISK: ~/Documents/Anti Reversing Project/Technique...
File Edit View Search Terminal Help

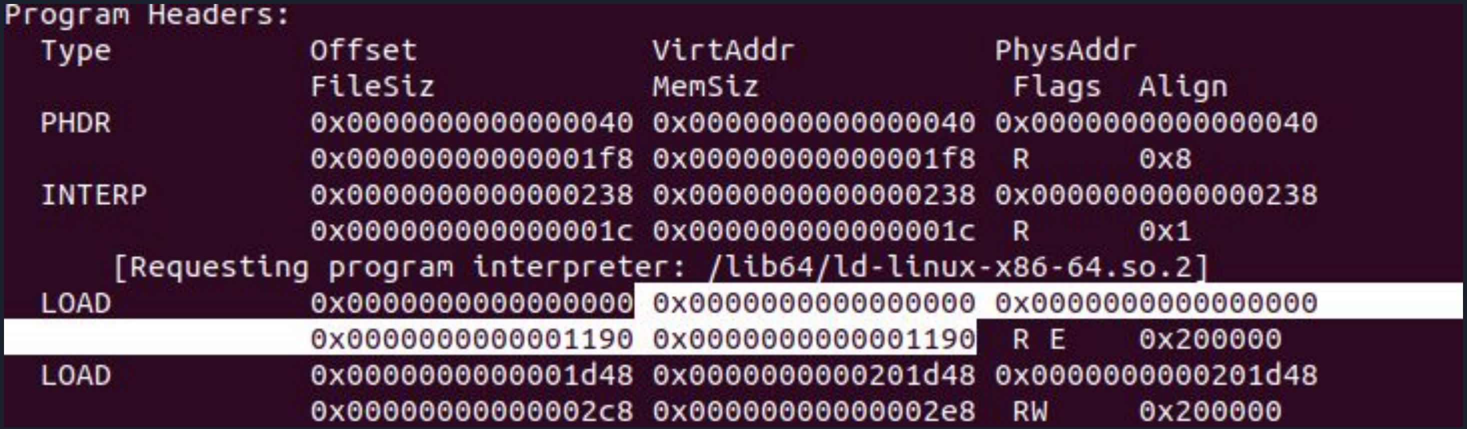
```
  [24] .bss              NOBITS           0000000000202020  00002010
       0000000000000010  0000000000000000  WA       0     0     32
  [25] .comment          PROGBITS         0000000000000000  00002010
       000000000000002a  0000000000000001  MS       0     0     1
  [26] .debug_aranges    PROGBITS         0000000000000000  0000203a
       0000000000000030  0000000000000000           0     0     1
  [27] .debug_info       PROGBITS         0000000000000000  0000206a
       000000000000073a  0000000000000000           0     0     1
  [28] .debug_abbrev     PROGBITS         0000000000000000  000027a4
       0000000000000164  0000000000000000           0     0     1
  [29] .debug_line       PROGBITS         0000000000000000  00002908
       00000000000001c2  0000000000000000           0     0     1
  [30] .debug_str        PROGBITS         0000000000000000  00002aca
       000000000000065c  0000000000000001  MS       0     0     1
  [31] .symtab           SYMTAB           0000000000000000  00003128
       00000000000007f8  0000000000000018          32    51     8
  [32] .strtab           STRTAB           0000000000000000  00003920
       0000000000000348  0000000000000000           0     0     1
  [33] .shstrtab         STRTAB           0000000000000000  00003c68
       000000000000013e  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
```

Step 2. Using objdump --dwarf=info on the binary to find the virtual address of o_password



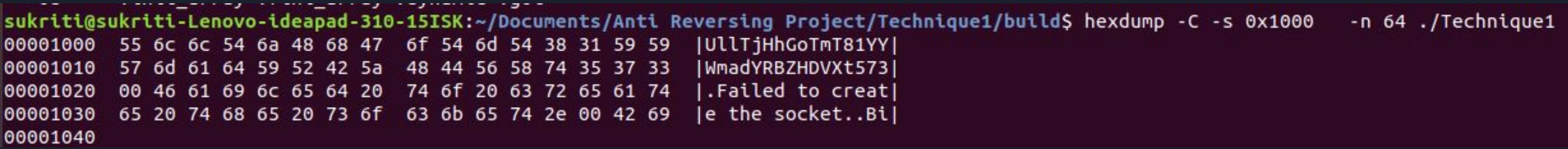
```
<1><5e3>: Abbrev Number: 25 (DW_TAG_variable)
   <5e4>   DW_AT_name        : (indirect string, offset: 0x27a): o_password
   <5e8>   DW_AT_decl_file   : 1
   <5e9>   DW_AT_decl_line   : 11
   <5ea>   DW_AT_type        : <0x5de>
   <5ee>   DW_AT_location    : 9 byte block: 3 0 10 0 0 0 0 0 0        (DW_OP_addr: 1000)
<1><5f8>: Abbrev Number: 26 (DW_TAG_subprogram)
```

Step 3 Find the segment in which the virtual address resides to determine the physical address of o_password



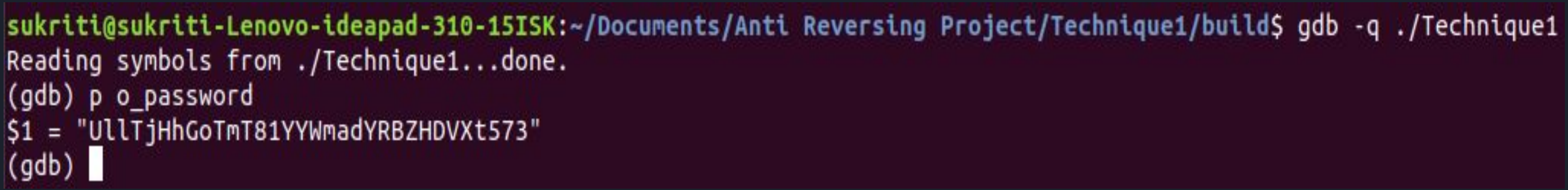
```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000001f8 0x00000000000001f8  R      0x8
  INTERP         0x0000000000000238 0x0000000000000238 0x0000000000000238
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000001190 0x0000000000001190  R E    0x200000
  LOAD           0x0000000000001d48 0x0000000000201d48 0x0000000000201d48
                 0x00000000000002c8 0x00000000000002e8  RW     0x200000
```

Step 4 Use hexdump to print the contents at the address(prints the value of the saved password)



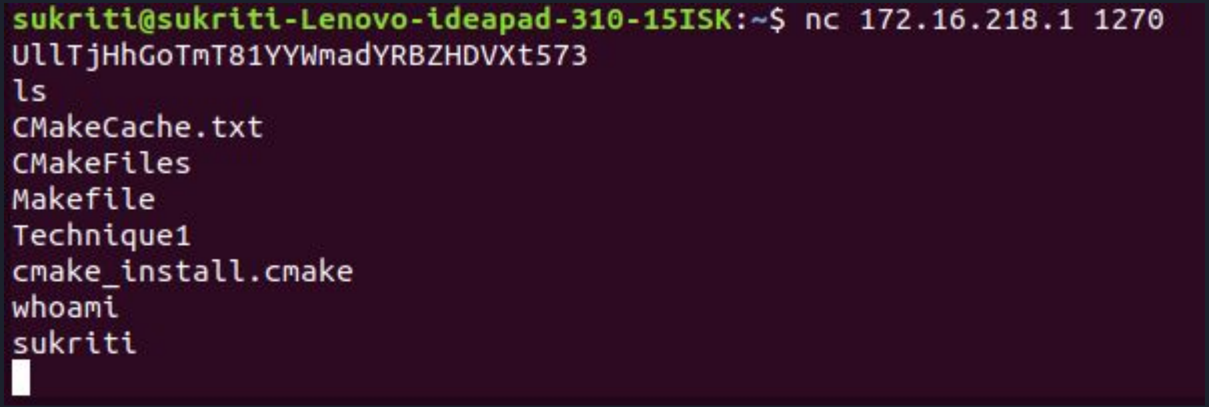
```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Technique1/build$ hexdump -C -s 0x1000   -n 64 ./Technique1
00001000  55 6c 6c 54 6a 48 68 47  6f 54 6d 54 38 31 59 59  |UllTjHhGoTmT81YY|
00001010  57 6d 61 64 59 52 42 5a  48 44 56 58 74 35 37 33  |WmadYRBZHDVXt573|
00001020  00 46 61 69 6c 65 64 20  74 6f 20 63 72 65 61 74  |.Failed to creat|
00001030  65 20 74 68 65 20 73 6f  63 6b 65 74 2e 00 42 69  |e the socket..Bi|
00001040
```

Step 5. We can also print the value of o_password in gdb



```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Technique1/build$ gdb -q ./Technique1
Reading symbols from ./Technique1...done.
(gdb) p o_password
$1 = "UllTjHhGoTmT81YYWmadYRBZHDVXt573"
(gdb)
```

Step 6. Use the password to connect to the victim's shell



```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~$ nc 172.16.218.1 1270
UllTjHhGoTmT81YYWmadYRBZHDVXt573
ls
CMakeCache.txt
CMakeFiles
Makefile
Technique1
cmake_install.cmake
whoami
sukriti
```

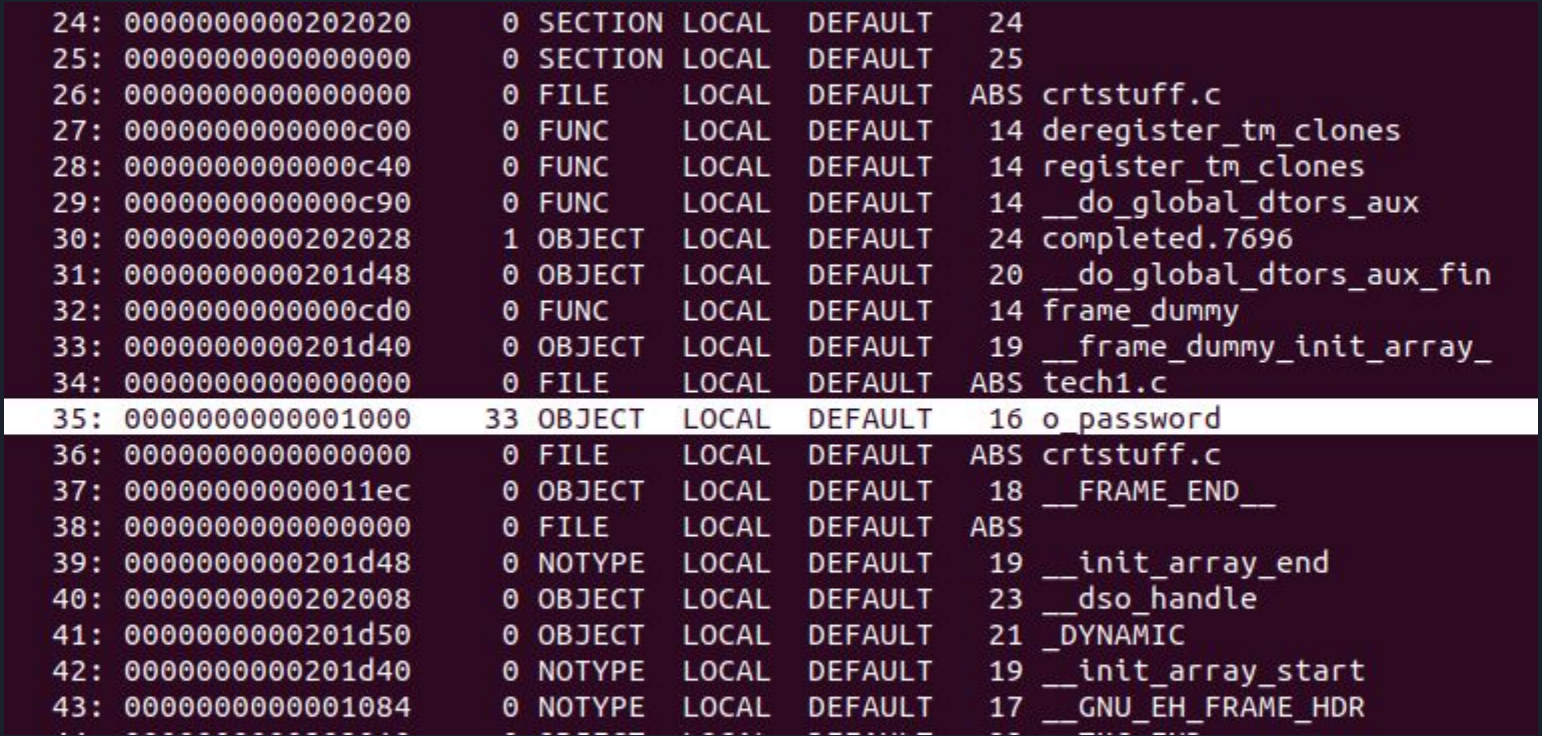
Removing the debugging information

- If we don’t include the -g option in compile flags it will prevent the various .debug_ sections from being generated which also means that GDB and IDA won’t receive the extra variable information to enrich their analysis.
- Since the debug sections are not generated, important debugging information like the path of the source file,the full path of the compilation directory and even exact line numbers where certain variables are declared.
- We won’t be able to recover the virtual address and eventually the actual address of the original password.
- Without -g a GDB user won’t be able to print o_password using “print” as was done earlier in this section.

Removing the symbol table using the -s file while compiling

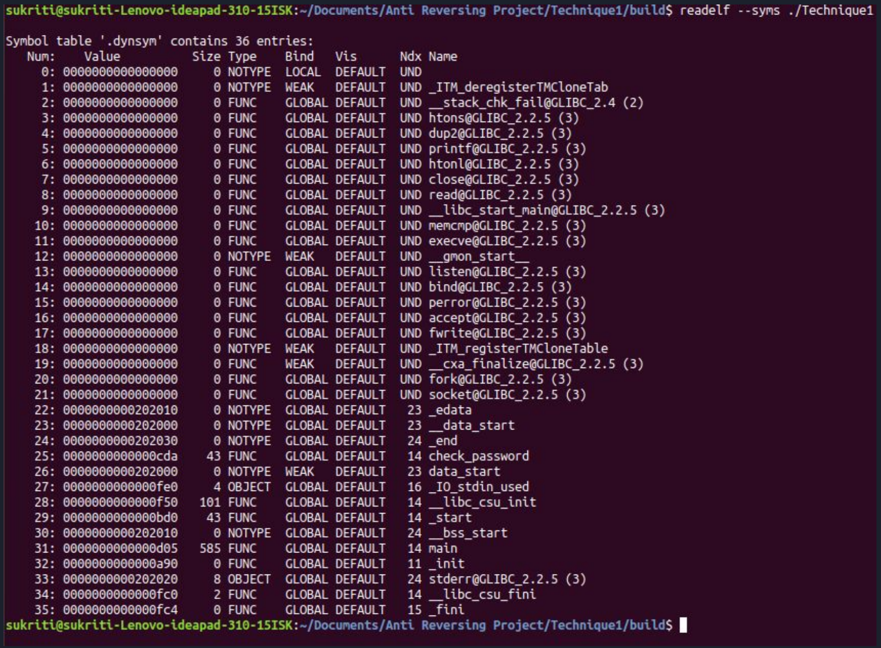
- Even though the debugging information from the binary has been excluded, the symbol table makes finding and extracting the password easy. Hence the value of the o_password can be recovered using hexdump.
- There are two types of symbol tables: .dynsym(dynamic symbol table) and .symtab (symbol table).
- If we compare the two tables, we see that the .dynsym section has the “A” whereas the .symtab doesn’t.
- The presence of the “A” flag, indicates that .dynsym will be loaded into memory when the program is started which means that .symtab is not loaded into memory and is therefore not necessary to execute the program.
- Hence the entire .symtab can be safely removed from the binary.

Getting the address of o_password using the .symtab



```
    24: 0000000000202020     0 SECTION LOCAL  DEFAULT   24
    25: 0000000000000000     0 SECTION LOCAL  DEFAULT   25
    26: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    27: 0000000000000c00     0 FUNC    LOCAL  DEFAULT   14 deregister_tm_clones
    28: 0000000000000c40     0 FUNC    LOCAL  DEFAULT   14 register_tm_clones
    29: 0000000000000c90     0 FUNC    LOCAL  DEFAULT   14 __do_global_dtors_aux
    30: 0000000000202028     1 OBJECT  LOCAL  DEFAULT   24 completed.7696
    31: 0000000000201d48     0 OBJECT  LOCAL  DEFAULT   20 __do_global_dtors_aux_fin
    32: 0000000000000cd0     0 FUNC    LOCAL  DEFAULT   14 frame_dummy
    33: 0000000000201d40     0 OBJECT  LOCAL  DEFAULT   19 __frame_dummy_init_array_
    34: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS tech1.c
    35: 0000000000001000    33 OBJECT  LOCAL  DEFAULT   16 o_password
    36: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    37: 00000000000011ec     0 OBJECT  LOCAL  DEFAULT   18 __FRAME_END__
    38: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS
    39: 0000000000201d48     0 NOTYPE  LOCAL  DEFAULT   19 __init_array_end
    40: 0000000000202008     0 OBJECT  LOCAL  DEFAULT   23 __dso_handle
    41: 0000000000201d50     0 OBJECT  LOCAL  DEFAULT   21 _DYNAMIC
    42: 0000000000201d40     0 NOTYPE  LOCAL  DEFAULT   19 __init_array_start
    43: 0000000000001084     0 NOTYPE  LOCAL  DEFAULT   17 __GNU_EH_FRAME_HDR
```

After Removing the Symbol Table by adding -s to the compiler flags



```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Technique1/build$ readelf --syms ./Technique1

Symbol table '.dynsym' contains 36 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND 
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __stack_chk_fail@GLIBC_2.4 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND htons@GLIBC_2.2.5 (3)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND dup2@GLIBC_2.2.5 (3)
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2.5 (3)
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND htonl@GLIBC_2.2.5 (3)
     7: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND close@GLIBC_2.2.5 (3)
     8: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND read@GLIBC_2.2.5 (3)
     9: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (3)
    10: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND memcmp@GLIBC_2.2.5 (3)
    11: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND execve@GLIBC_2.2.5 (3)
    12: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    13: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND listen@GLIBC_2.2.5 (3)
    14: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND bind@GLIBC_2.2.5 (3)
    15: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND perror@GLIBC_2.2.5 (3)
    16: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND accept@GLIBC_2.2.5 (3)
    17: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fwrite@GLIBC_2.2.5 (3)
    18: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
    19: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (3)
    20: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fork@GLIBC_2.2.5 (3)
    21: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND socket@GLIBC_2.2.5 (3)
    22: 0000000000202010     0 NOTYPE  GLOBAL DEFAULT   23 _edata
    23: 0000000000202000     0 NOTYPE  GLOBAL DEFAULT   23 __data_start
    24: 0000000000202030     0 NOTYPE  GLOBAL DEFAULT   24 _end
    25: 0000000000000cda    43 FUNC    GLOBAL DEFAULT   14 check_password
    26: 0000000000202000     0 NOTYPE  WEAK   DEFAULT   23 data_start
    27: 0000000000000fe0     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
    28: 0000000000000f50   101 FUNC    GLOBAL DEFAULT   14 __libc_csu_init
    29: 0000000000000bd0    43 FUNC    GLOBAL DEFAULT   14 _start
    30: 0000000000202010     0 NOTYPE  GLOBAL DEFAULT   24 __bss_start
    31: 0000000000000d05   585 FUNC    GLOBAL DEFAULT   14 main
    32: 0000000000000a90     0 FUNC    GLOBAL DEFAULT   11 _init
    33: 0000000000202020     8 OBJECT  GLOBAL DEFAULT   24 stderr@GLIBC_2.2.5 (3)
    34: 0000000000000fc0     2 FUNC    GLOBAL DEFAULT   14 __libc_csu_fini
    35: 0000000000000fc4     0 FUNC    GLOBAL DEFAULT   15 _fini
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Technique1/build$ 
```

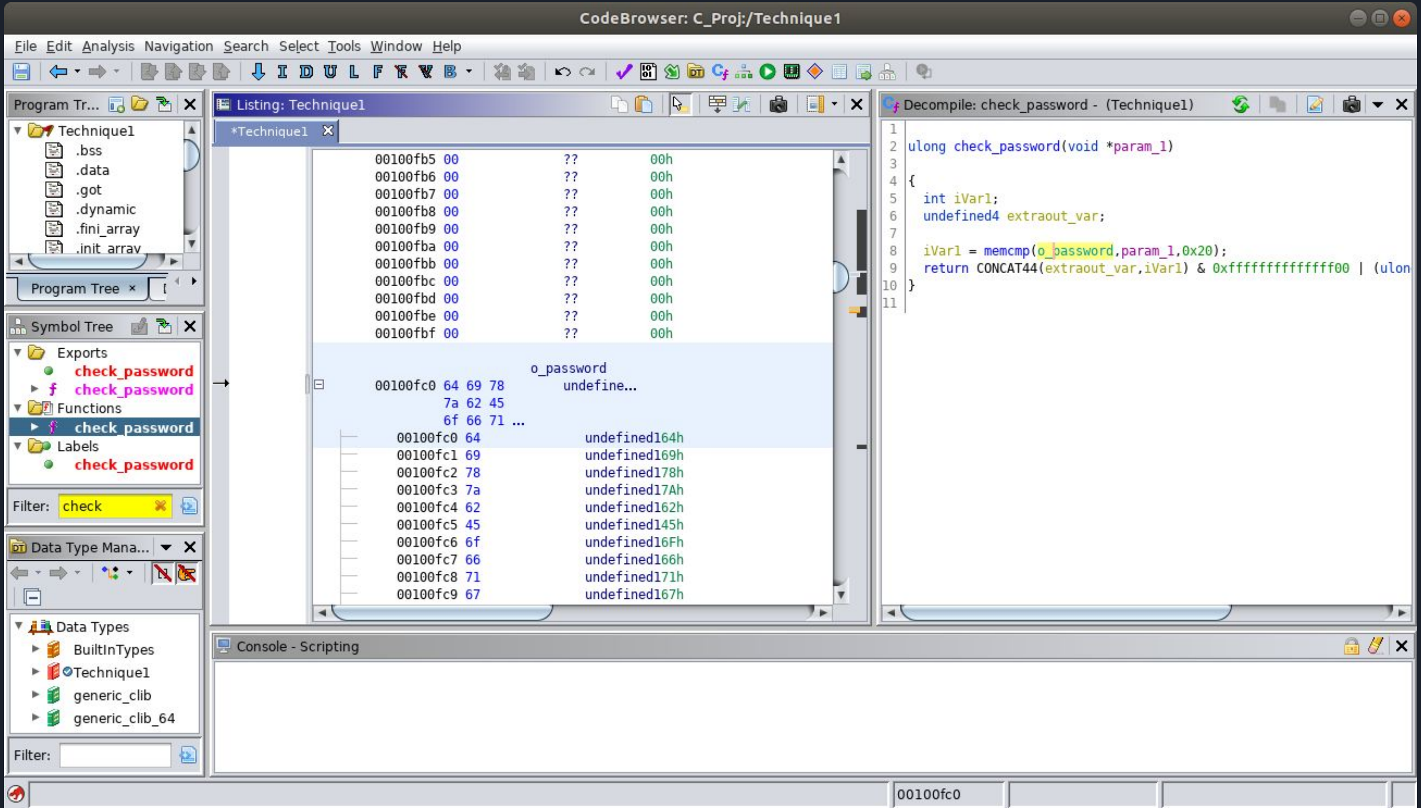
After Removing the Symbol Table

- As we can see in the previous screenshot, the entire .symtab is now gone.
- Not only does this make the binary smaller but it also has disabled easy access to o_password and removed the string “tech1.c”(the C file) from the binary altogether.
- Hence we can remove the .symtab in order to deny a reverse engineer useful data.
- However, this does not hide all the information which can be useful to an attack in order to reverse engineer as we will see in the furthur slides.

Using -fvisibility= hidden

- If we examine the .dynsym, we can still find useful information like the two symbols: check_password and main.
- The .dynsym provides both the starting address of the function as well as its size. These are really useful pieces of information for a dissasembler such as Ghidra.
- On double clicking the o_password variable, we can see all the characters in o_password in hexadecimal form in the disassembler.
- In order to hide the function symbols, we need to add the gcc flag -fvisibility=hidden to the compilation flag which hides all possible symbols.
- Hence this would prevent a reverse engineer from, yet again, easily dumping the password.

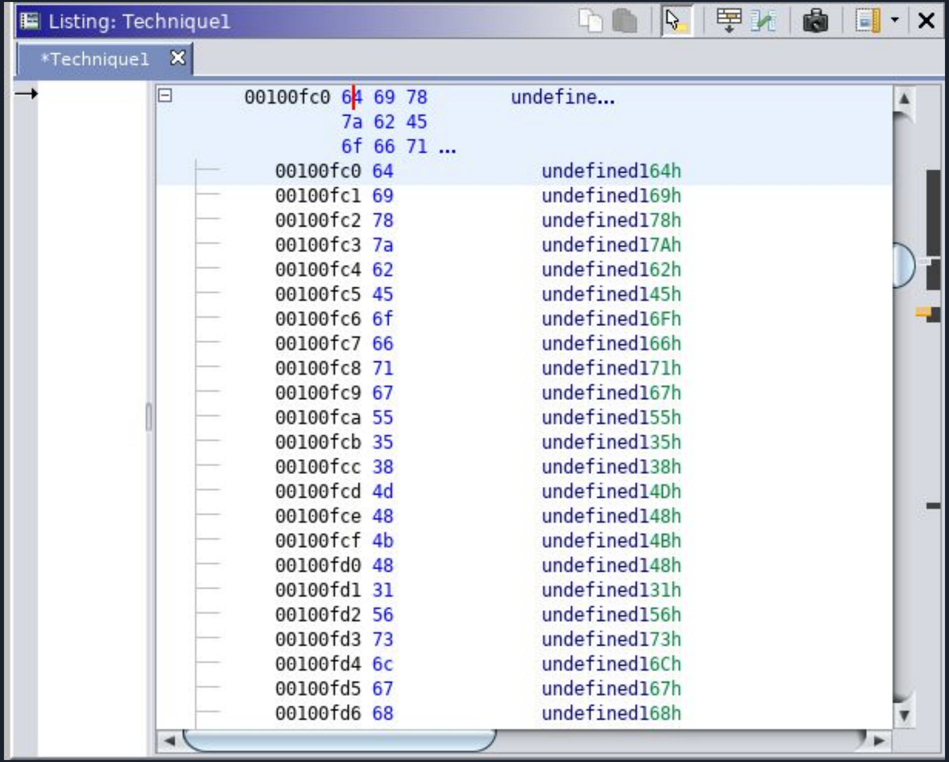
We can see the check_password Function Signature



```
00100fb5 00           ??          00h
00100fb6 00           ??          00h
00100fb7 00           ??          00h
00100fb8 00           ??          00h
00100fb9 00           ??          00h
00100fba 00           ??          00h
00100fbb 00           ??          00h
00100fbc 00           ??          00h
00100fbd 00           ??          00h
00100fbe 00           ??          00h
00100fbf 00           ??          00h

                  o_password
00100fc0 64 69 78     undefine...
         7a 62 45
         6f 66 71 ...
   00100fc0 64           undefined164h
   00100fc1 69           undefined169h
   00100fc2 78           undefined178h
   00100fc3 7a           undefined17Ah
   00100fc4 62           undefined162h
   00100fc5 45           undefined145h
   00100fc6 6f           undefined16Fh
   00100fc7 66           undefined166h
   00100fc8 71           undefined171h
   00100fc9 67           undefined167h
```

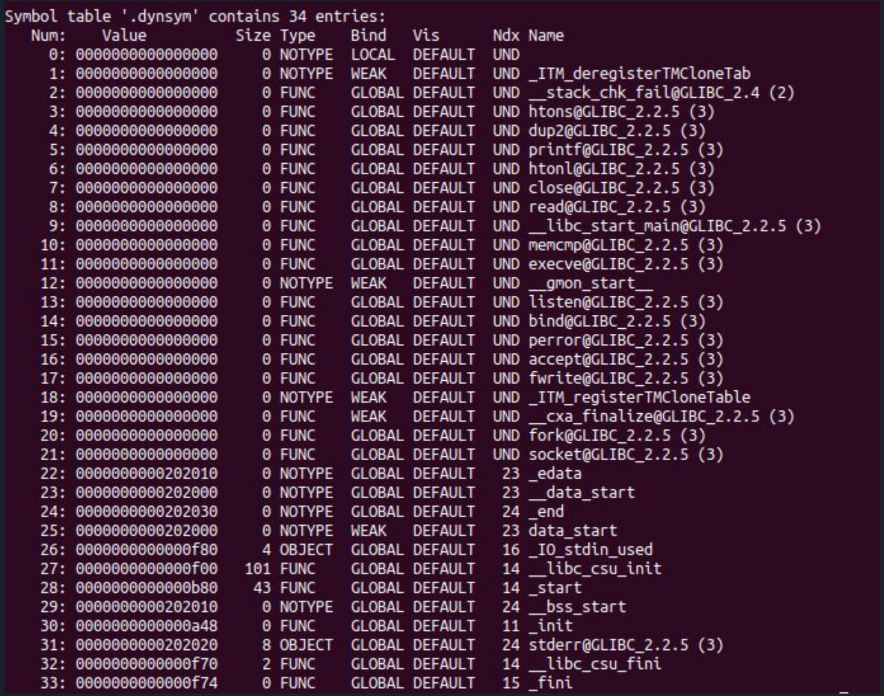
```

ulong check_password(void *param_1)

{
  int iVar1;
  undefined4 extraout_var;

  iVar1 = memcmp(o_password,param_1,0x20);
  return CONCAT44(extraout_var,iVar1) & 0xffffffffffffff00 | (ulon
}

```

Dissassembler showing the password characters in hexadecimal form



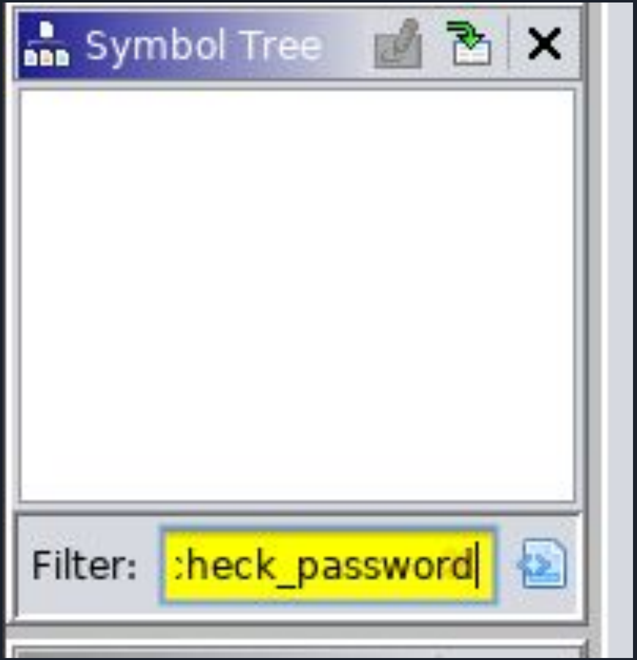
```
00100fc0 64 69 78        undefine...
         7a 62 45
         6f 66 71 ...
   00100fc0 64              undefined164h
   00100fc1 69              undefined169h
   00100fc2 78              undefined178h
   00100fc3 7a              undefined17Ah
   00100fc4 62              undefined162h
   00100fc5 45              undefined145h
   00100fc6 6f              undefined16Fh
   00100fc7 66              undefined166h
   00100fc8 71              undefined171h
   00100fc9 67              undefined167h
   00100fca 55              undefined155h
   00100fcb 35              undefined135h
   00100fcc 38              undefined138h
   00100fcd 4d              undefined14Dh
   00100fce 48              undefined148h
   00100fcf 4b              undefined14Bh
   00100fd0 48              undefined148h
   00100fd1 31              undefined131h
   00100fd2 56              undefined156h
   00100fd3 73              undefined173h
   00100fd4 6c              undefined16Ch
   00100fd5 67              undefined167h
   00100fd6 68              undefined168h
```

After adding the -fvisibility=hidden flag
Main and check_password symbols missing



```
Symbol table '.dynsym' contains 34 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND 
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __stack_chk_fail@GLIBC_2.4 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND htons@GLIBC_2.2.5 (3)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND dup2@GLIBC_2.2.5 (3)
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2.5 (3)
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND htonl@GLIBC_2.2.5 (3)
     7: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND close@GLIBC_2.2.5 (3)
     8: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND read@GLIBC_2.2.5 (3)
     9: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (3)
    10: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND memcmp@GLIBC_2.2.5 (3)
    11: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND execve@GLIBC_2.2.5 (3)
    12: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    13: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND listen@GLIBC_2.2.5 (3)
    14: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND bind@GLIBC_2.2.5 (3)
    15: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND perror@GLIBC_2.2.5 (3)
    16: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND accept@GLIBC_2.2.5 (3)
    17: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fwrite@GLIBC_2.2.5 (3)
    18: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
    19: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (3)
    20: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fork@GLIBC_2.2.5 (3)
    21: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND socket@GLIBC_2.2.5 (3)
    22: 0000000000202010     0 NOTYPE  GLOBAL DEFAULT   23 _edata
    23: 0000000000202000     0 NOTYPE  GLOBAL DEFAULT   23 __data_start
    24: 0000000000202030     0 NOTYPE  GLOBAL DEFAULT   24 _end
    25: 0000000000202000     0 NOTYPE  WEAK   DEFAULT   23 data_start
    26: 0000000000000f80     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
    27: 0000000000000f00   101 FUNC    GLOBAL DEFAULT   14 __libc_csu_init
    28: 0000000000000b80    43 FUNC    GLOBAL DEFAULT   14 _start
    29: 0000000000202010     0 NOTYPE  GLOBAL DEFAULT   24 __bss_start
    30: 0000000000000a48     0 FUNC    GLOBAL DEFAULT   11 _init
    31: 0000000000202020     8 OBJECT  GLOBAL DEFAULT   24 stderr@GLIBC_2.2.5 (3)
    32: 0000000000000f70     2 FUNC    GLOBAL DEFAULT   14 __libc_csu_fini
    33: 0000000000000f74     0 FUNC    GLOBAL DEFAULT   15 _fini
```

After adding the -fvisibility=hidden flag



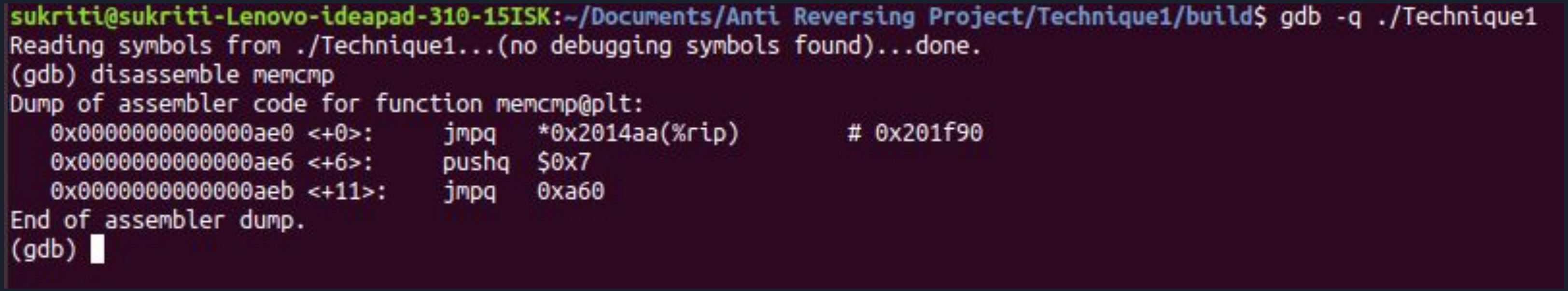
Filter: :heck_password

- After adding the fvisibility=hidden in the gcc flags, we again open the binary in Ghidra.
- Now if we try to search for check_password, we are not able to find the function in the symbol tree.
- Hence there is no way for us to recover the function signature and the value of the original password using the symbol tree.
- Similarly, we can also not find main and hence GDB no longer will be able to break at main()

Using LD_PRELOAD

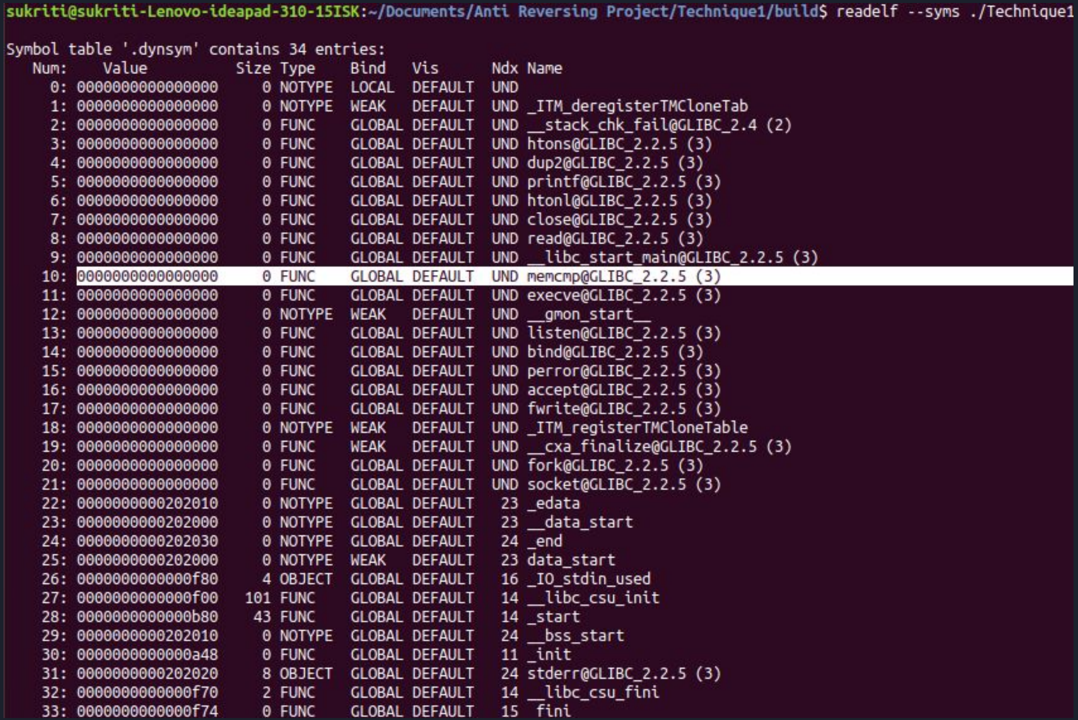
- The resolving of functions at run time can also be useful from an attacker's point of view. In order to exploit this, we make use of the fact that functions whose implementations exist in external libraries that won't be loaded until runtime.
- These functions can be found by viewing the .dynsym and checking for functions with label UND(undefined). We find that memcmp() is one such function.
- Opening the binary in gdb, when we use "disassasmble memcmp", we can see that this command point to an address in the Procedure Linking Table(PLT).
- However if we try the above step once the binary has started executing, we see that memcmp now points to libc.so. By using the dynamic linker's LD_PRELOAD option, we can load our own library before the other shared objects, like libc.so, are loaded.
- This means that we can introduce our own code to handle memcmp() and our function will be executed instead of libc.so's. In our custom code, we print the values of the original and the provided password.

memcmp() is seen to point to address in the PLT(Procedure Linkage Table)



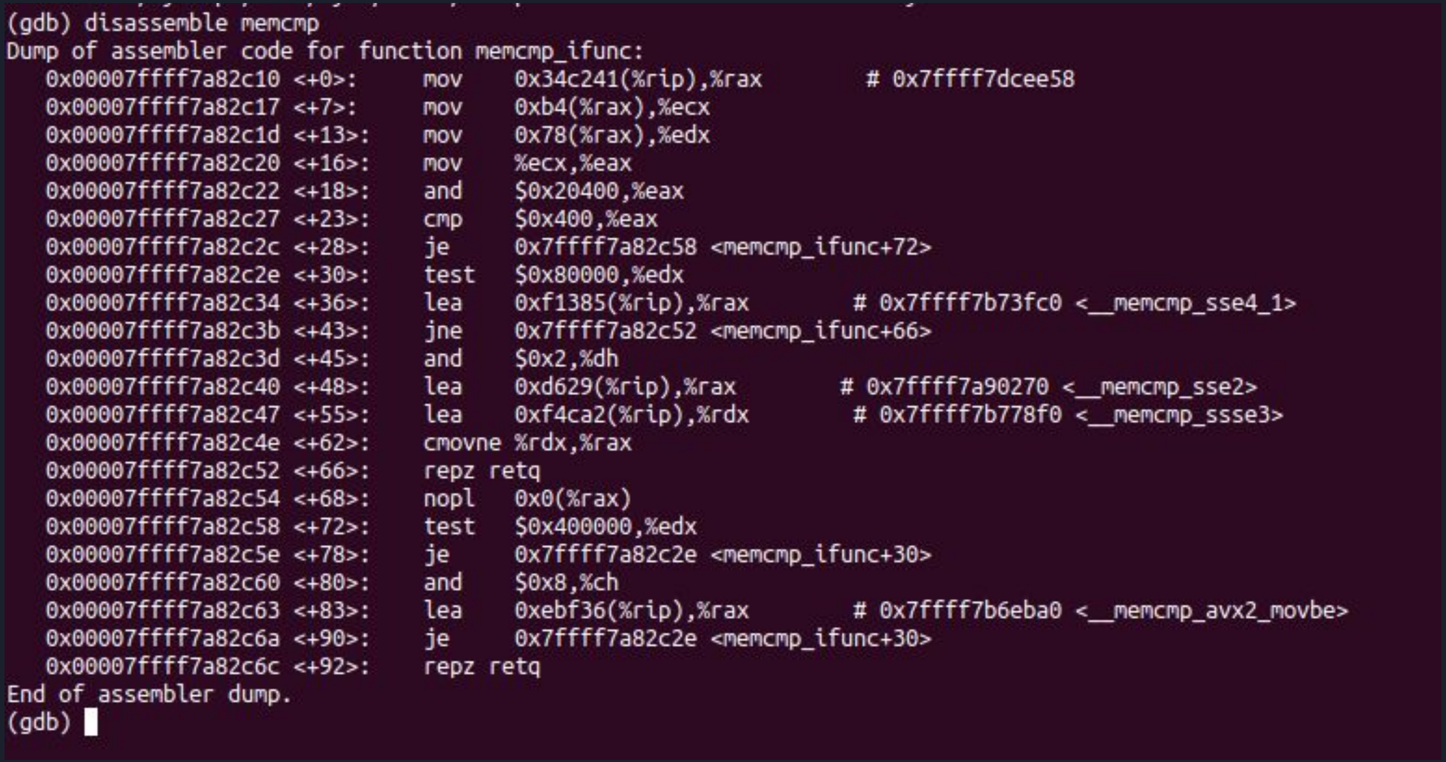
```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Technique1/build$ gdb -q ./Technique1
Reading symbols from ./Technique1...(no debugging symbols found)...done.
(gdb) disassemble memcmp
Dump of assembler code for function memcmp@plt:
   0x0000000000000ae0 <+0>:     jmpq   *0x2014aa(%rip)        # 0x201f90
   0x0000000000000ae6 <+6>:     pushq  $0x7
   0x0000000000000aeb <+11>:    jmpq   0xa60
End of assembler dump.
(gdb) 
```

Memcmp’s implementation exists in an external library



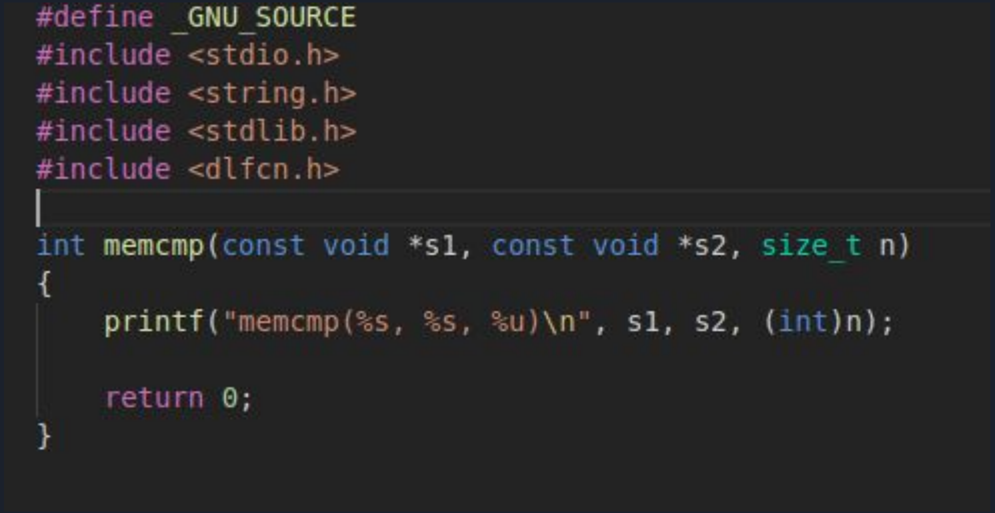
```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Technique1/build$ readelf --syms ./Technique1

Symbol table '.dynsym' contains 34 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __stack_chk_fail@GLIBC_2.4 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND htons@GLIBC_2.2.5 (3)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND dup2@GLIBC_2.2.5 (3)
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2.5 (3)
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND htonl@GLIBC_2.2.5 (3)
     7: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND close@GLIBC_2.2.5 (3)
     8: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND read@GLIBC_2.2.5 (3)
     9: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (3)
    10: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND memcmp@GLIBC_2.2.5 (3)
    11: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND execve@GLIBC_2.2.5 (3)
    12: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    13: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND listen@GLIBC_2.2.5 (3)
    14: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND bind@GLIBC_2.2.5 (3)
    15: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND perror@GLIBC_2.2.5 (3)
    16: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND accept@GLIBC_2.2.5 (3)
    17: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fwrite@GLIBC_2.2.5 (3)
    18: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
    19: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (3)
    20: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fork@GLIBC_2.2.5 (3)
    21: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND socket@GLIBC_2.2.5 (3)
    22: 0000000000202010     0 NOTYPE  GLOBAL DEFAULT   23 _edata
    23: 0000000000202000     0 NOTYPE  GLOBAL DEFAULT   23 __data_start
    24: 0000000000202030     0 NOTYPE  GLOBAL DEFAULT   24 _end
    25: 0000000000202000     0 NOTYPE  WEAK   DEFAULT   23 data_start
    26: 0000000000000f80     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
    27: 0000000000000f00   101 FUNC    GLOBAL DEFAULT   14 __libc_csu_init
    28: 0000000000000b80    43 FUNC    GLOBAL DEFAULT   14 _start
    29: 0000000000202010     0 NOTYPE  GLOBAL DEFAULT   24 __bss_start
    30: 0000000000000a48     0 FUNC    GLOBAL DEFAULT   11 _init
    31: 0000000000202020     8 OBJECT  GLOBAL DEFAULT   24 stderr@GLIBC_2.2.5 (3)
    32: 0000000000000f70     2 FUNC    GLOBAL DEFAULT   14 __libc_csu_fini
    33: 0000000000000f74     0 FUNC    GLOBAL DEFAULT   15 _fini
```

Disassembling memcmp after running the binary



```
(gdb) disassemble memcmp
Dump of assembler code for function memcmp_ifunc:
   0x00007ffff7a82c10 <+0>:     mov    0x34c241(%rip),%rax        # 0x7ffff7dcee58
   0x00007ffff7a82c17 <+7>:     mov    0xb4(%rax),%ecx
   0x00007ffff7a82c1d <+13>:    mov    0x78(%rax),%edx
   0x00007ffff7a82c20 <+16>:    mov    %ecx,%eax
   0x00007ffff7a82c22 <+18>:    and    $0x20400,%eax
   0x00007ffff7a82c27 <+23>:    cmp    $0x400,%eax
   0x00007ffff7a82c2c <+28>:    je     0x7ffff7a82c58 <memcmp_ifunc+72>
   0x00007ffff7a82c2e <+30>:    test   $0x80000,%edx
   0x00007ffff7a82c34 <+36>:    lea    0xf1385(%rip),%rax        # 0x7ffff7b73fc0 <__memcmp_sse4_1>
   0x00007ffff7a82c3b <+43>:    jne    0x7ffff7a82c52 <memcmp_ifunc+66>
   0x00007ffff7a82c3d <+45>:    and    $0x2,%dh
   0x00007ffff7a82c40 <+48>:    lea    0xd629(%rip),%rax        # 0x7ffff7a90270 <__memcmp_sse2>
   0x00007ffff7a82c47 <+55>:    lea    0xf4ca2(%rip),%rdx        # 0x7ffff7b778f0 <__memcmp_ssse3>
   0x00007ffff7a82c4e <+62>:    cmovne %rdx,%rax
   0x00007ffff7a82c52 <+66>:    repz retq
   0x00007ffff7a82c54 <+68>:    nopl   0x0(%rax)
   0x00007ffff7a82c58 <+72>:    test   $0x400000,%edx
   0x00007ffff7a82c5e <+78>:    je     0x7ffff7a82c2e <memcmp_ifunc+30>
   0x00007ffff7a82c60 <+80>:    and    $0x8,%ch
   0x00007ffff7a82c63 <+83>:    lea    0xebf36(%rip),%rax        # 0x7ffff7b6eba0 <__memcmp_avx2_movbe>
   0x00007ffff7a82c6a <+90>:    je     0x7ffff7a82c2e <memcmp_ifunc+30>
   0x00007ffff7a82c6c <+92>:    repz retq
End of assembler dump.
(gdb) 
```

The custom memcpy function definition

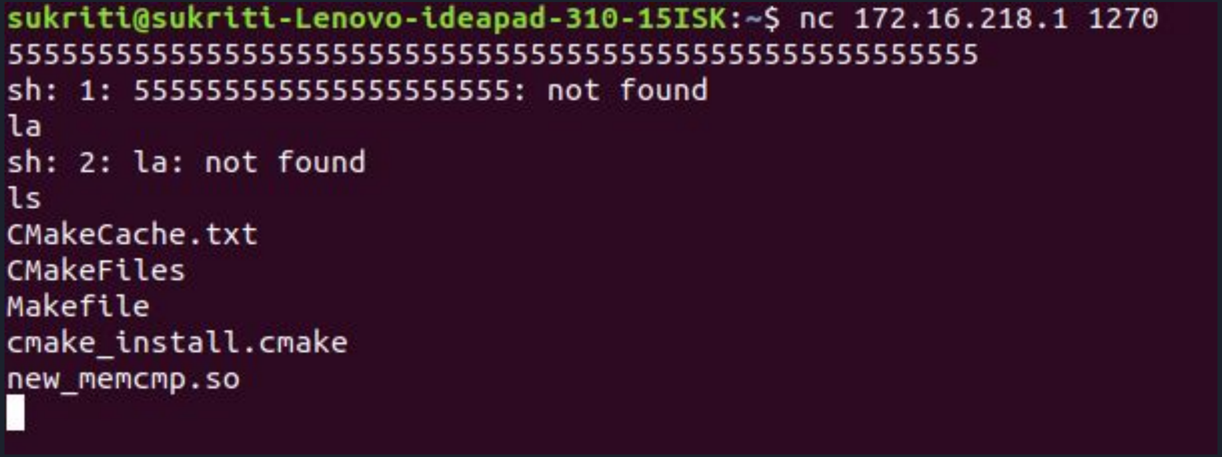


```
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <dlfcn.h>

int memcmp(const void *s1, const void *s2, size_t n)
{
    printf("memcmp(%s, %s, %u)\n", s1, s2, (int)n);

    return 0;
}
```

Demo

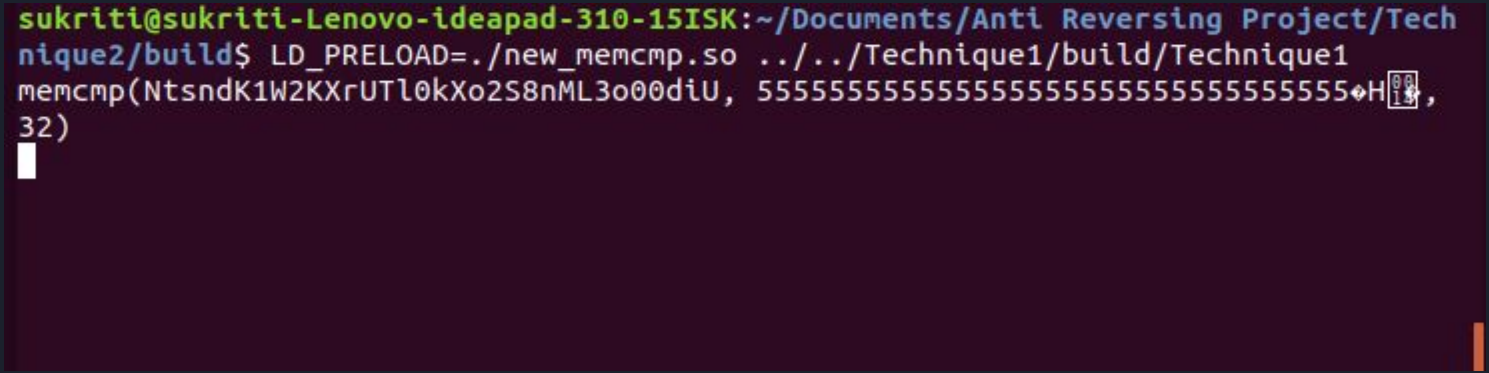
https://drive.google.com/open?id=1EbVRPeU3sJerMVlUv3RVFYohxH-ehvI6

Attempt to connect to the bind shell providing bogus password, using LD_PRELOAD to get get memcmp to return 0 so we are granted access to the bind shell



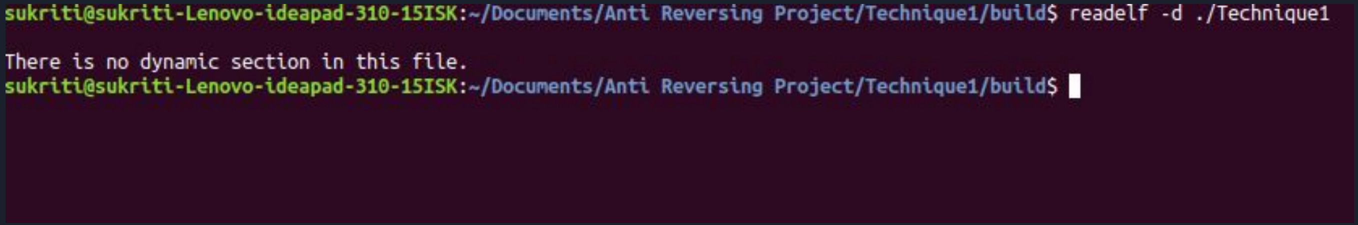
```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~$ nc 172.16.218.1 1270
555555555555555555555555555555555555555555555555555555
sh: 1: 555555555555555555555: not found
la
sh: 2: la: not found
ls
CMakeCache.txt
CMakeFiles
Makefile
cmake_install.cmake
new_memcmp.so
```

The original as well as the provided password gets printed also



```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Tech
nique2/build$ LD_PRELOAD=./new_memcmp.so ../../Technique1/build/Technique1
memcmp(NtsndK1W2KXrUTl0kXo2S8nML3o00diU, 55555555555555555555555555555555�H�,
32)
█
```

Use of -static flag



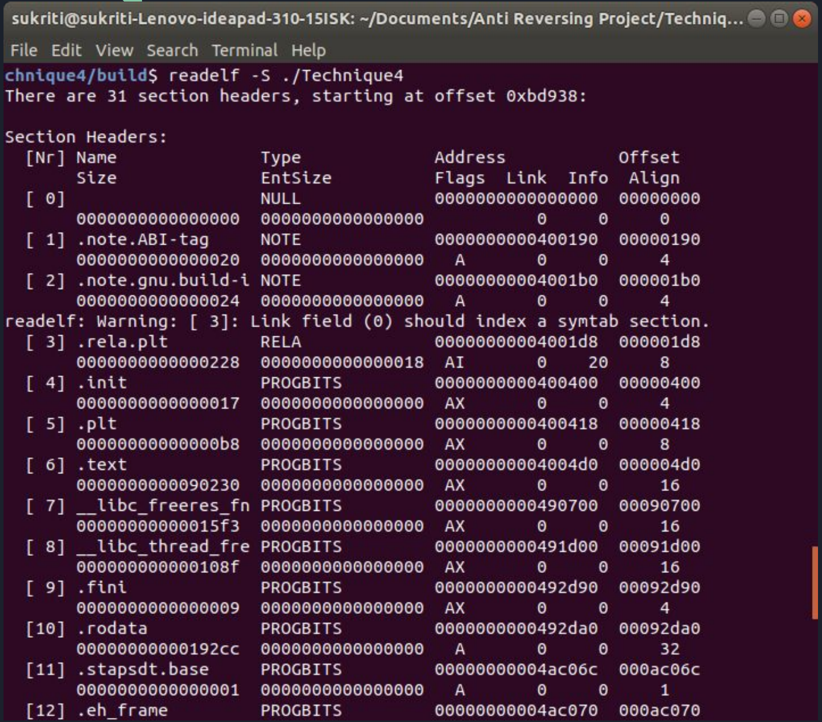
```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Technique1/build$ readelf -d ./Technique1

There is no dynamic section in this file.
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Technique1/build$ 
```

In order to prevent the use of LD_PRELOAD to exploit the binary, we should use the -static flag during compilation which prevents linking with the shared libraries,

Modifying the ELF Header To Prevent Discovery, Parsing and the Display of the Section Header Table

- In order find, parse and display the section headers table, four variables are needed:-
  - Start of sections headers
  - Size of section headers
  - Number of section headers
  - Section header string table index
- On studying the elf header format, we notice the following facts:-
  - Location at the offset of 0x28, points to the start of the section header table(8 bytes).
  - Location at the offset of 0x3A, contains the size of a section header table entry(2 bytes).
  - Location at the offset of 0x3C, contains the number of entries in the section header table(2 bytes).
  - Location at the offset of 0x3E, contains index of the section header table entry that contains the section names(2 bytes).
- If we use a hex editor to zero out the values at these locations, then locating or parsing the section header table would be impossible.

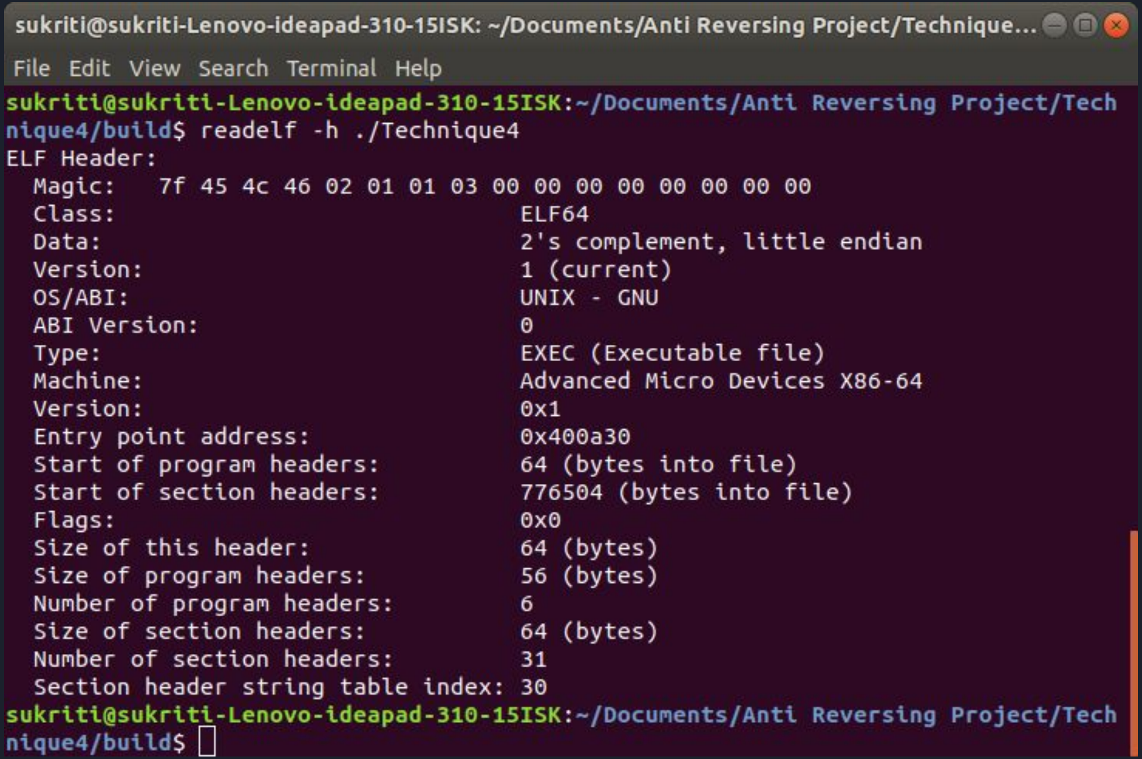
Section Headers


sukriti@sukriti-Lenovo-ideapad-310-15ISK: ~/Documents/Anti Reversing Project/Techniq...
File Edit View Search Terminal Help

```
chnique4/build$ readelf -S ./Technique4
There are 31 section headers, starting at offset 0xbd938:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .note.ABI-tag     NOTE             0000000000400190  00000190
       0000000000000020  0000000000000000   A       0     0     4
  [ 2] .note.gnu.build-i NOTE             00000000004001b0  000001b0
       0000000000000024  0000000000000000   A       0     0     4
readelf: Warning: [ 3]: Link field (0) should index a symtab section.
  [ 3] .rela.plt         RELA             00000000004001d8  000001d8
       0000000000000228  0000000000000018  AI       0    20     8
  [ 4] .init             PROGBITS         0000000000400400  00000400
       0000000000000017  0000000000000000  AX       0     0     4
  [ 5] .plt              PROGBITS         0000000000400418  00000418
       00000000000000b8  0000000000000000  AX       0     0     8
  [ 6] .text             PROGBITS         00000000004004d0  000004d0
       0000000000090230  0000000000000000  AX       0     0     16
  [ 7] __libc_freeres_fn PROGBITS         0000000000490700  00090700
       00000000000015f3  0000000000000000  AX       0     0     16
  [ 8] __libc_thread_fre PROGBITS         0000000000491d00  00091d00
       000000000000108f  0000000000000000  AX       0     0     16
  [ 9] .fini             PROGBITS         0000000000492d90  00092d90
       0000000000000009  0000000000000000  AX       0     0     4
  [10] .rodata           PROGBITS         0000000000492da0  00092da0
       00000000000192cc  0000000000000000   A       0     0     32
  [11] .stapsdt.base     PROGBITS         00000000004ac06c  000ac06c
       0000000000000001  0000000000000000   A       0     0     1
  [12] .eh_frame         PROGBITS         00000000004ac070  000ac070
```

The section headers table is useful for a reverse engineer because it breaks down the binary's address space into very specific chunks. However, the section headers table isn't actually needed for execution.

The Elf Header



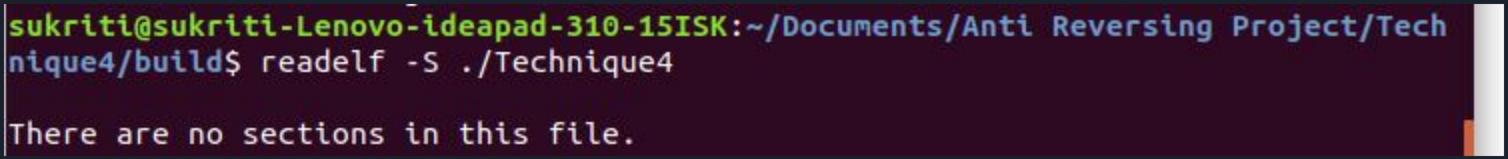
sukriti@sukriti-Lenovo-ideapad-310-15ISK: ~/Documents/Anti Reversing Project/Technique...
File Edit View Search Terminal Help

```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Tech
nique4/build$ readelf -h ./Technique4
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400a30
  Start of program headers:          64 (bytes into file)
  Start of section headers:          776504 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         6
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Tech
nique4/build$ 
```

Demo

https://drive.google.com/open?id=1rko8_xrwpz8dA_Xhm0592TiAE1MjYRgD

After The Required Bytes have been set to Zero Using a Hex Editor



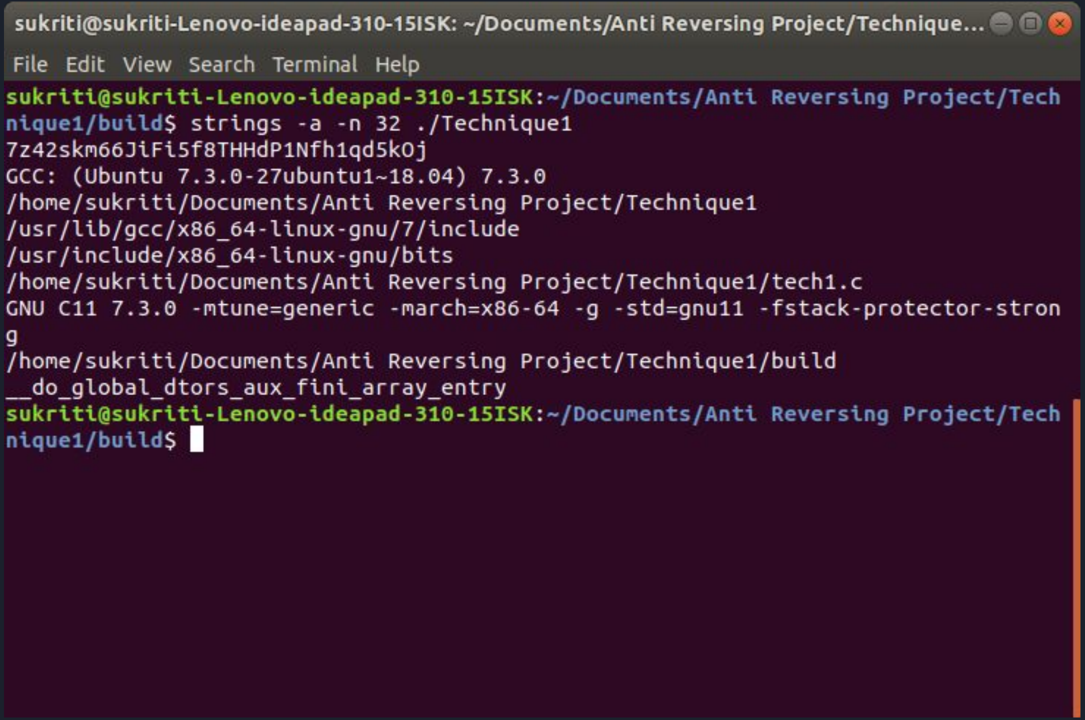
```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Tech
nique4/build$ readelf -S ./Technique4

There are no sections in this file.
```

Use of Stack Strings

- Use of strings command is one of the most common tools used by reverse engineers for string analysis.
- This process can be made difficult by hiding the strings by constructing it with code.
- The goal is to add each byte of the string onto the stack one at a time.
- For example, instead of using: execve("/bin/sh", empty, empty);
  We should use

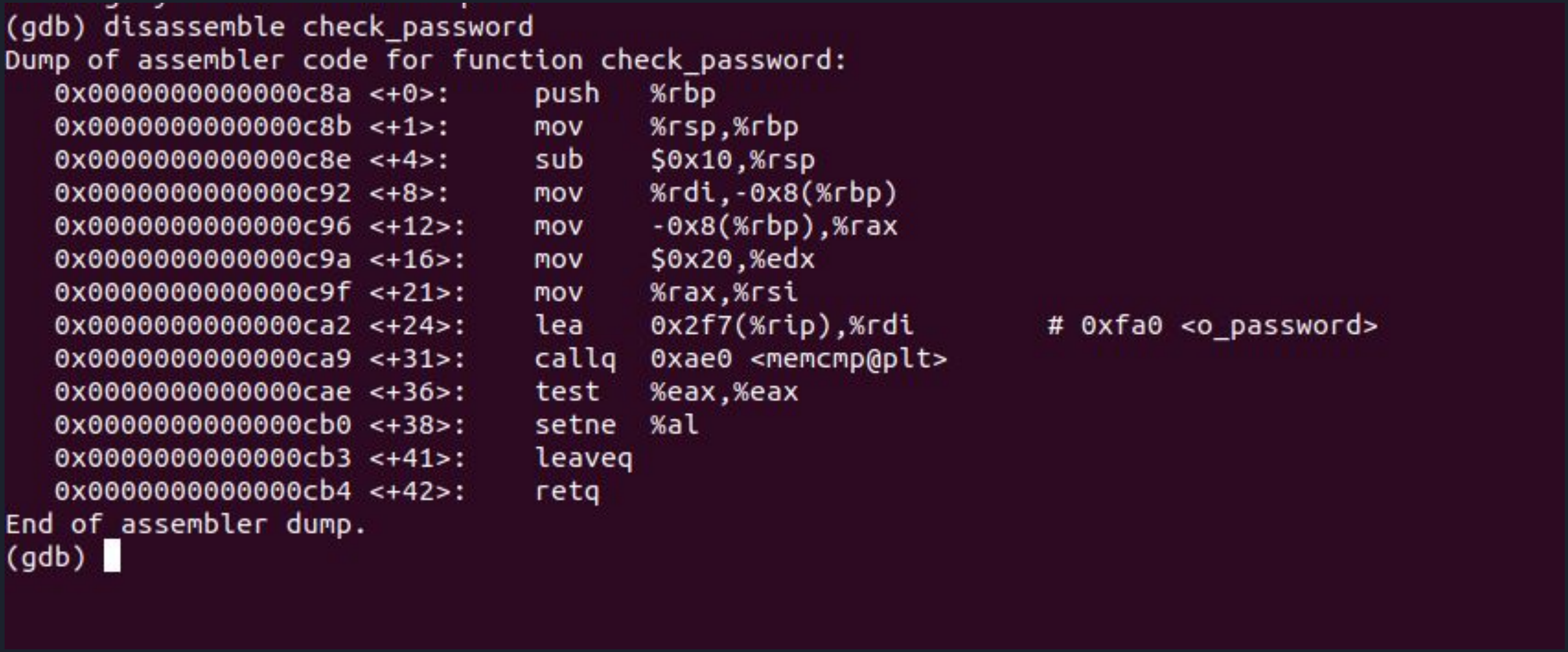
```
char binsh[] = { '/', 'b', 'i', 'n', '/', 's', 'h', 0 }; execve(binsh, empty, empty);
```

Use of strings to Display the Password


sukriti@sukriti-Lenovo-ideapad-310-15ISK: ~/Documents/Anti Reversing Project/Technique...
File Edit View Search Terminal Help

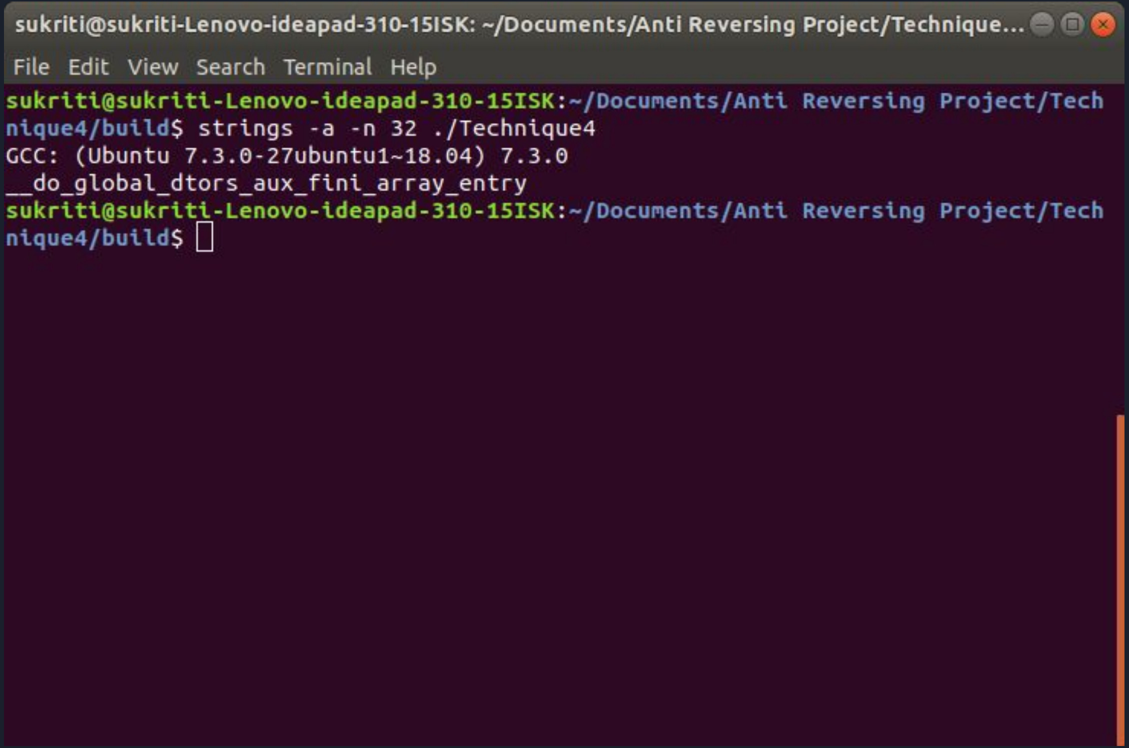
```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Tech
nique1/build$ strings -a -n 32 ./Technique1
7z42skm66JiFi5f8THHdP1Nfh1qd5kOj
GCC: (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0
/home/sukriti/Documents/Anti Reversing Project/Technique1
/usr/lib/gcc/x86_64-linux-gnu/7/include
/usr/include/x86_64-linux-gnu/bits
/home/sukriti/Documents/Anti Reversing Project/Technique1/tech1.c
GNU C11 7.3.0 -mtune=generic -march=x86-64 -g -std=gnu11 -fstack-protector-stron
g
/home/sukriti/Documents/Anti Reversing Project/Technique1/build
__do_global_dtors_aux_fini_array_entry
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Tech
nique1/build$ 
```

Disassembly of check _password showing the memory address where the password string is stored.



```
(gdb) disassemble check_password
Dump of assembler code for function check_password:
   0x0000000000000c8a <+0>:     push   %rbp
   0x0000000000000c8b <+1>:     mov    %rsp,%rbp
   0x0000000000000c8e <+4>:     sub    $0x10,%rsp
   0x0000000000000c92 <+8>:     mov    %rdi,-0x8(%rbp)
   0x0000000000000c96 <+12>:    mov    -0x8(%rbp),%rax
   0x0000000000000c9a <+16>:    mov    $0x20,%edx
   0x0000000000000c9f <+21>:    mov    %rax,%rsi
   0x0000000000000ca2 <+24>:    lea    0x2f7(%rip),%rdi        # 0xfa0 <o_password>
   0x0000000000000ca9 <+31>:    callq  0xae0 <memcmp@plt>
   0x0000000000000cae <+36>:    test   %eax,%eax
   0x0000000000000cb0 <+38>:    setne  %al
   0x0000000000000cb3 <+41>:    leaveq
   0x0000000000000cb4 <+42>:    retq
End of assembler dump.
(gdb) 
```

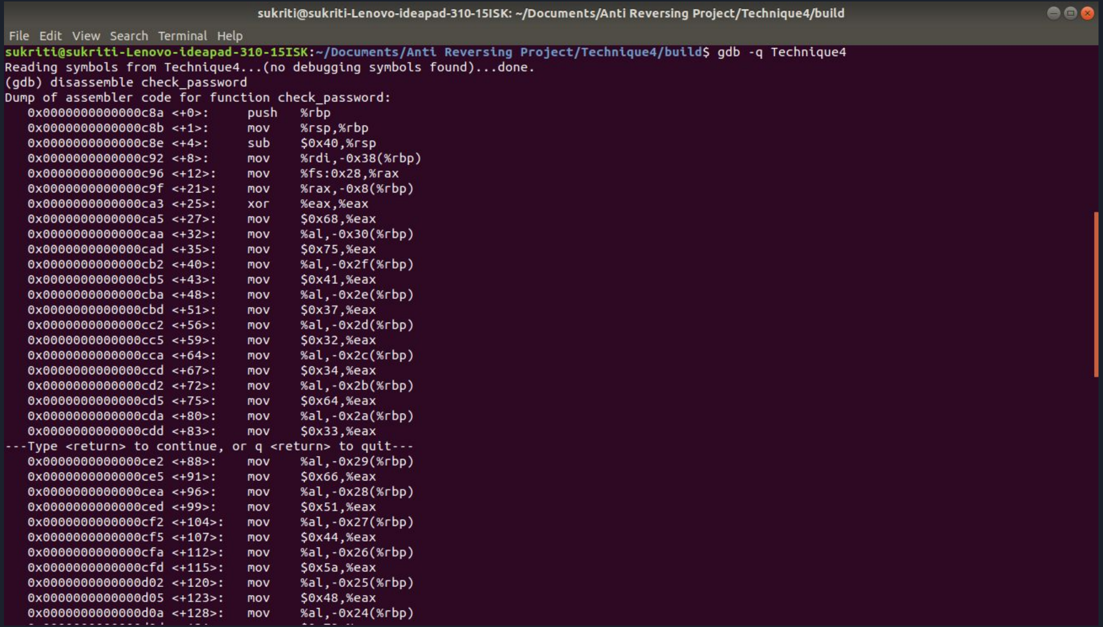
After using Stack Strings, strings command cannot be used to view the password.



sukriti@sukriti-Lenovo-ideapad-310-15ISK: ~/Documents/Anti Reversing Project/Technique...
File Edit View Search Terminal Help

```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Tech
nique4/build$ strings -a -n 32 ./Technique4
GCC: (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0
__do_global_dtors_aux_fini_array_entry
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Tech
nique4/build$ 
```

Even if other obfuscation flags are absent, the disassembly of check_function doesn't directly show the password string

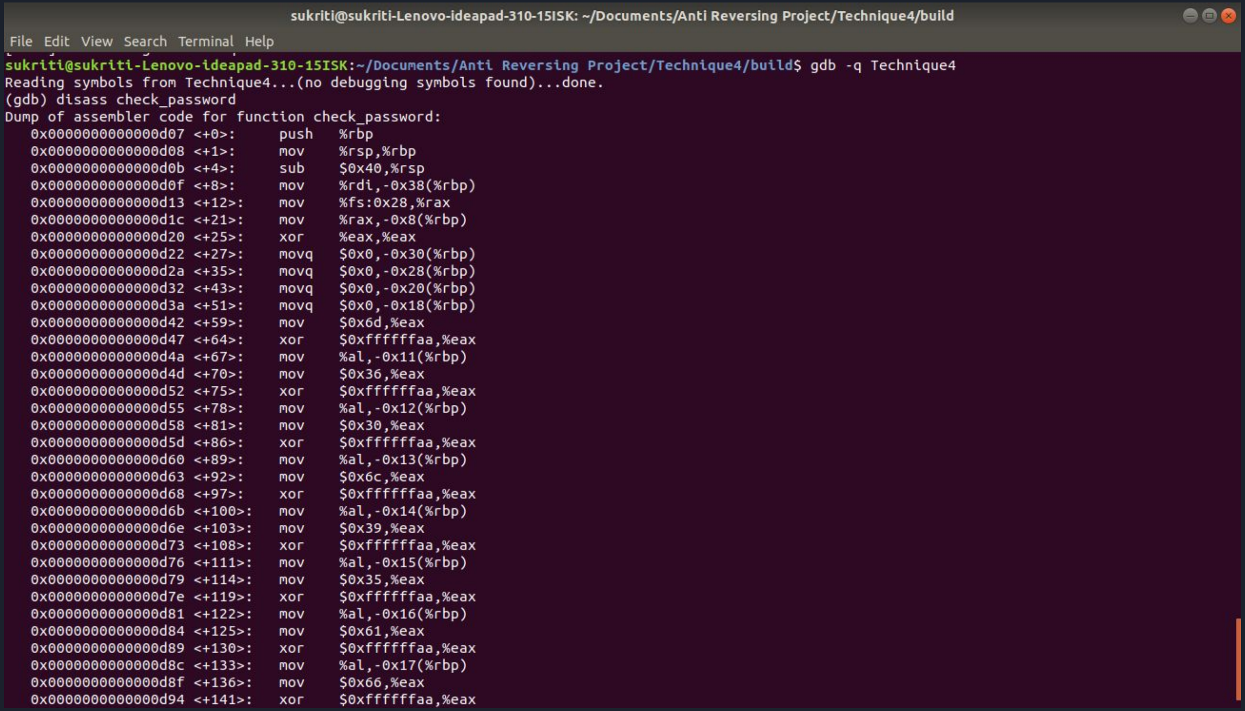

sukriti@sukriti-Lenovo-ideapad-310-15ISK: ~/Documents/Anti Reversing Project/Technique4/build
File Edit View Search Terminal Help

```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Technique4/build$ gdb -q Technique4
Reading symbols from Technique4...(no debugging symbols found)...done.
(gdb) disassemble check_password
Dump of assembler code for function check_password:
   0x0000000000000c8a <+0>:     push   %rbp
   0x0000000000000c8b <+1>:     mov    %rsp,%rbp
   0x0000000000000c8e <+4>:     sub    $0x40,%rsp
   0x0000000000000c92 <+8>:     mov    %rdi,-0x38(%rbp)
   0x0000000000000c96 <+12>:    mov    %fs:0x28,%rax
   0x0000000000000c9f <+21>:    mov    %rax,-0x8(%rbp)
   0x0000000000000ca3 <+25>:    xor    %eax,%eax
   0x0000000000000ca5 <+27>:    mov    $0x68,%eax
   0x0000000000000caa <+32>:    mov    %al,-0x30(%rbp)
   0x0000000000000cad <+35>:    mov    $0x75,%eax
   0x0000000000000cb2 <+40>:    mov    %al,-0x2f(%rbp)
   0x0000000000000cb5 <+43>:    mov    $0x41,%eax
   0x0000000000000cba <+48>:    mov    %al,-0x2e(%rbp)
   0x0000000000000cbd <+51>:    mov    $0x37,%eax
   0x0000000000000cc2 <+56>:    mov    %al,-0x2d(%rbp)
   0x0000000000000cc5 <+59>:    mov    $0x32,%eax
   0x0000000000000cca <+64>:    mov    %al,-0x2c(%rbp)
   0x0000000000000ccd <+67>:    mov    $0x34,%eax
   0x0000000000000cd2 <+72>:    mov    %al,-0x2b(%rbp)
   0x0000000000000cd5 <+75>:    mov    $0x64,%eax
   0x0000000000000cda <+80>:    mov    %al,-0x2a(%rbp)
   0x0000000000000cdd <+83>:    mov    $0x33,%eax
---Type <return> to continue, or q <return> to quit---
   0x0000000000000ce2 <+88>:    mov    %al,-0x29(%rbp)
   0x0000000000000ce5 <+91>:    mov    $0x66,%eax
   0x0000000000000cea <+96>:    mov    %al,-0x28(%rbp)
   0x0000000000000ced <+99>:    mov    $0x51,%eax
   0x0000000000000cf2 <+104>:   mov    %al,-0x27(%rbp)
   0x0000000000000cf5 <+107>:   mov    $0x44,%eax
   0x0000000000000cfa <+112>:   mov    %al,-0x26(%rbp)
   0x0000000000000cfd <+115>:   mov    $0x5a,%eax
   0x0000000000000d02 <+120>:   mov    %al,-0x25(%rbp)
   0x0000000000000d05 <+123>:   mov    $0x48,%eax
   0x0000000000000d0a <+128>:   mov    %al,-0x24(%rbp)
```

XOR Strings

- Using strings stacks complicates the recovery process of the password string but the password can still be recovered since the each character’s hex representation is visible in the disassembly.
- Since can be complicated further by XORing each byte so that the reverse engineer can’t just read the values straight from the disassembly

After Using XOR strings, the password isn’t quite as clear in the disassembly.



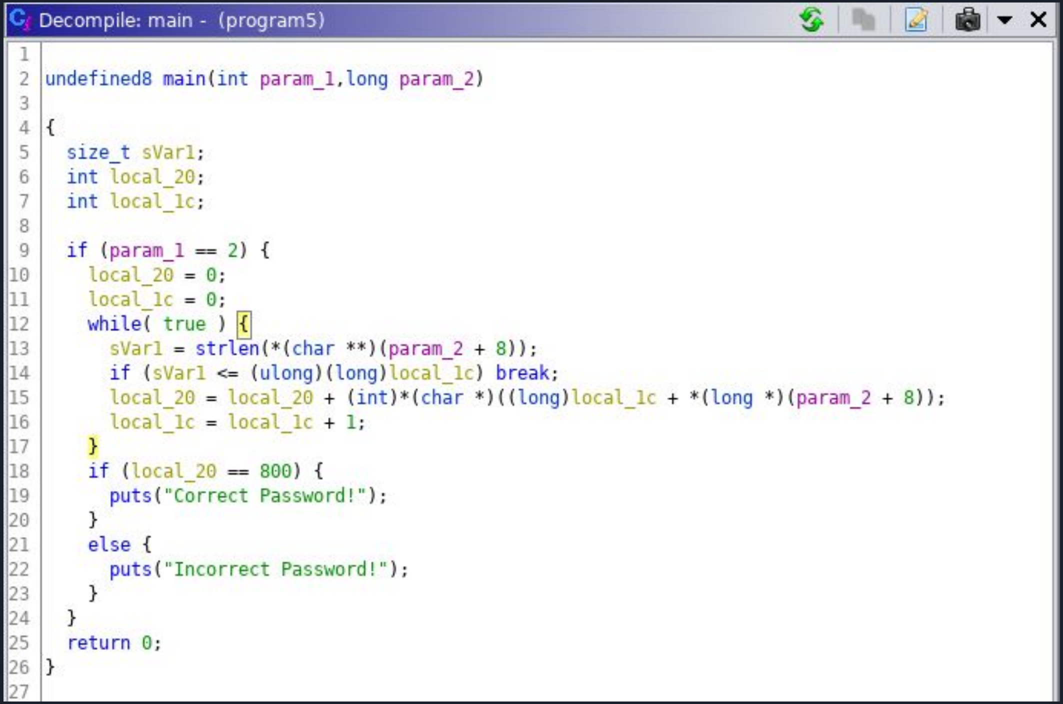
sukriti@sukriti-Lenovo-ideapad-310-15ISK: ~/Documents/Anti Reversing Project/Technique4/build
File Edit View Search Terminal Help

```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Technique4/build$ gdb -q Technique4
Reading symbols from Technique4...(no debugging symbols found)...done.
(gdb) disass check_password
Dump of assembler code for function check_password:
   0x0000000000000d07 <+0>:     push   %rbp
   0x0000000000000d08 <+1>:     mov    %rsp,%rbp
   0x0000000000000d0b <+4>:     sub    $0x40,%rsp
   0x0000000000000d0f <+8>:     mov    %rdi,-0x38(%rbp)
   0x0000000000000d13 <+12>:    mov    %fs:0x28,%rax
   0x0000000000000d1c <+21>:    mov    %rax,-0x8(%rbp)
   0x0000000000000d20 <+25>:    xor    %eax,%eax
   0x0000000000000d22 <+27>:    movq   $0x0,-0x30(%rbp)
   0x0000000000000d2a <+35>:    movq   $0x0,-0x28(%rbp)
   0x0000000000000d32 <+43>:    movq   $0x0,-0x20(%rbp)
   0x0000000000000d3a <+51>:    movq   $0x0,-0x18(%rbp)
   0x0000000000000d42 <+59>:    mov    $0x6d,%eax
   0x0000000000000d47 <+64>:    xor    $0xffffffaa,%eax
   0x0000000000000d4a <+67>:    mov    %al,-0x11(%rbp)
   0x0000000000000d4d <+70>:    mov    $0x36,%eax
   0x0000000000000d52 <+75>:    xor    $0xffffffaa,%eax
   0x0000000000000d55 <+78>:    mov    %al,-0x12(%rbp)
   0x0000000000000d58 <+81>:    mov    $0x30,%eax
   0x0000000000000d5d <+86>:    xor    $0xffffffaa,%eax
   0x0000000000000d60 <+89>:    mov    %al,-0x13(%rbp)
   0x0000000000000d63 <+92>:    mov    $0x6c,%eax
   0x0000000000000d68 <+97>:    xor    $0xffffffaa,%eax
   0x0000000000000d6b <+100>:   mov    %al,-0x14(%rbp)
   0x0000000000000d6e <+103>:   mov    $0x39,%eax
   0x0000000000000d73 <+108>:   xor    $0xffffffaa,%eax
   0x0000000000000d76 <+111>:   mov    %al,-0x15(%rbp)
   0x0000000000000d79 <+114>:   mov    $0x35,%eax
   0x0000000000000d7e <+119>:   xor    $0xffffffaa,%eax
   0x0000000000000d81 <+122>:   mov    %al,-0x16(%rbp)
   0x0000000000000d84 <+125>:   mov    $0x61,%eax
   0x0000000000000d89 <+130>:   xor    $0xffffffaa,%eax
   0x0000000000000d8c <+133>:   mov    %al,-0x17(%rbp)
   0x0000000000000d8f <+136>:   mov    $0x66,%eax
   0x0000000000000d94 <+141>:   xor    $0xffffffaa,%eax
```

Analysis of a Key-Generator

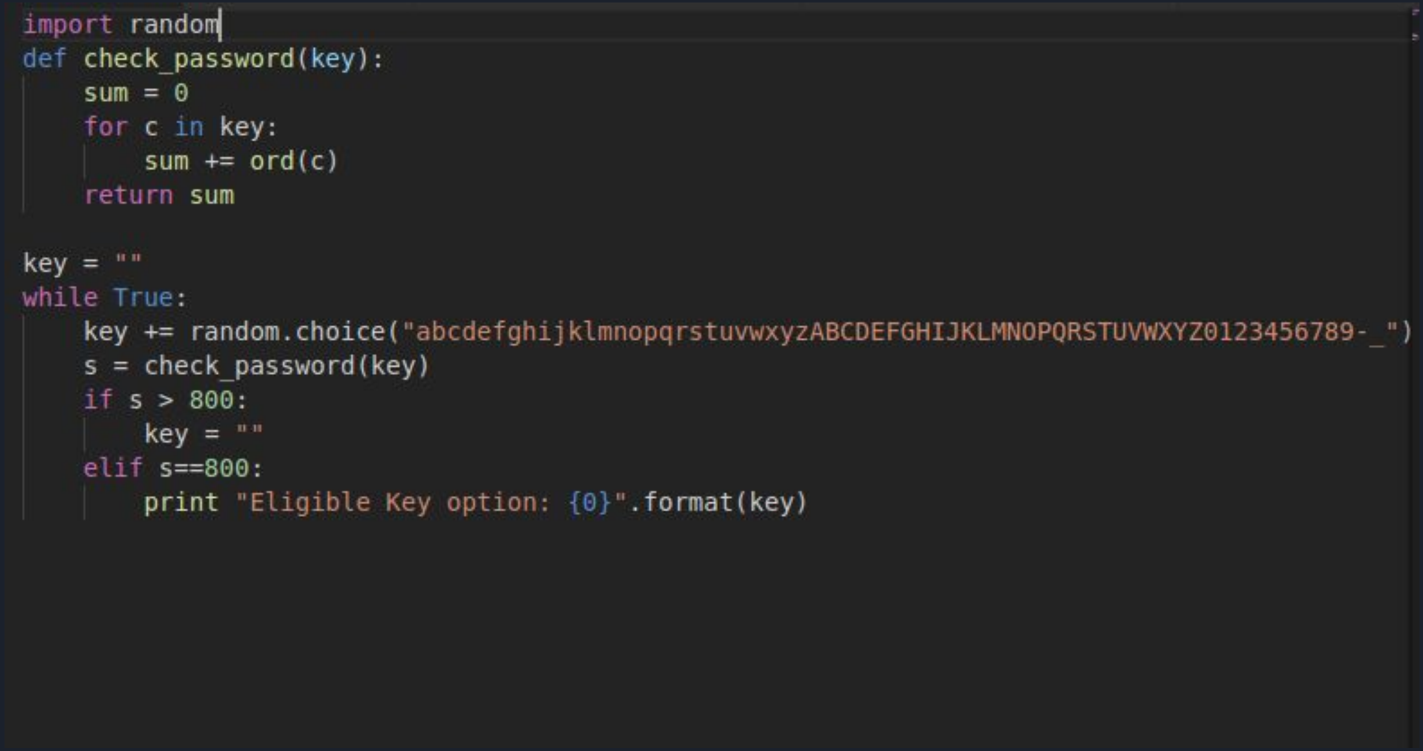
- We are given a binary, for which we try to decode the password. The binary is initially compiled without any of the obfuscation techniques.
- Viewing the code decompilation in ghidra, we see that the values of all the characters are summed up and compared to the value “800”.
- If the sum equals 800, it prints that it is the correct password.
- In order to decode this, we right a python script to generate eligible passwords which equal this value,
- We then test one of the eligible keys with the binary to check if it is a valid password.
- We see the impact of using the obfuscation techniques in cracking this logic.

Decompilation of The Main Password Checking Function Before Obfuscation



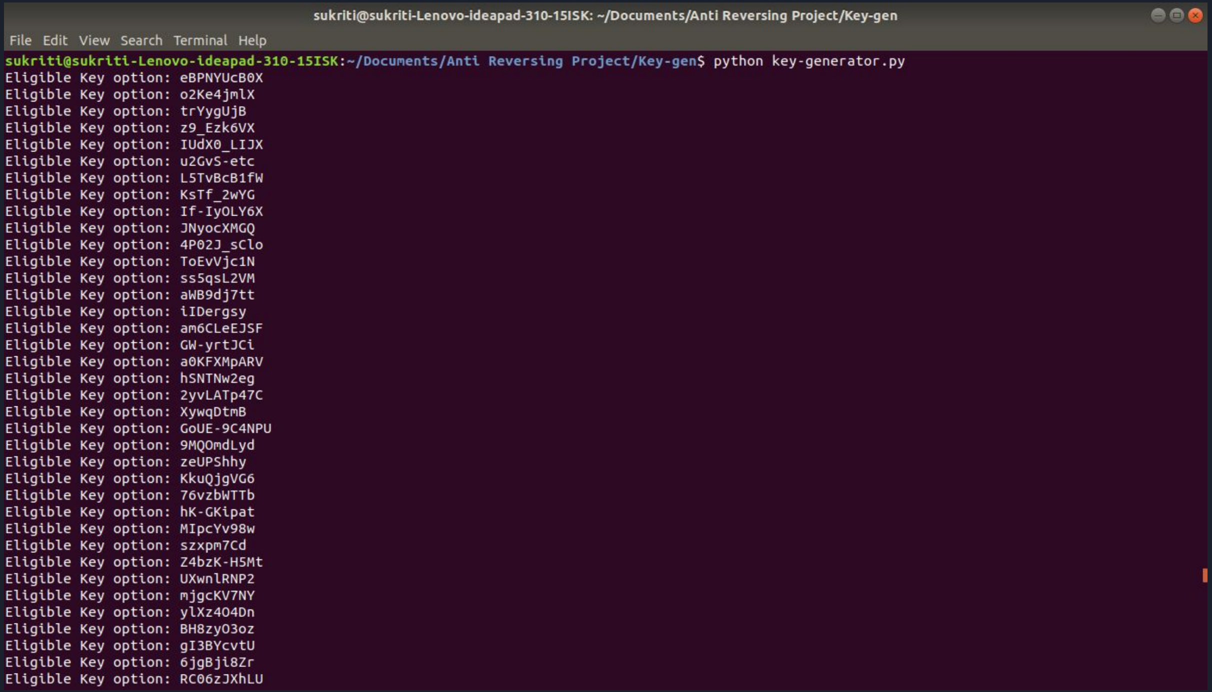
```

undefined8 main(int param_1,long param_2)

{
  size_t sVar1;
  int local_20;
  int local_1c;

  if (param_1 == 2) {
    local_20 = 0;
    local_1c = 0;
    while( true ) {
      sVar1 = strlen(*(char **)(param_2 + 8));
      if (sVar1 <= (ulong)(long)local_1c) break;
      local_20 = local_20 + (int)*(char *)((long)local_1c + *(long *)(param_2 + 8));
      local_1c = local_1c + 1;
    }
    if (local_20 == 800) {
      puts("Correct Password!");
    }
    else {
      puts("Incorrect Password!");
    }
  }
  return 0;
}

```

Key Generator Script



```
import random
def check_password(key):
    sum = 0
    for c in key:
        sum += ord(c)
    return sum

key = ""
while True:
    key += random.choice("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-_")
    s = check_password(key)
    if s > 800:
        key = ""
    elif s==800:
        print "Eligible Key option: {0}".format(key)
```

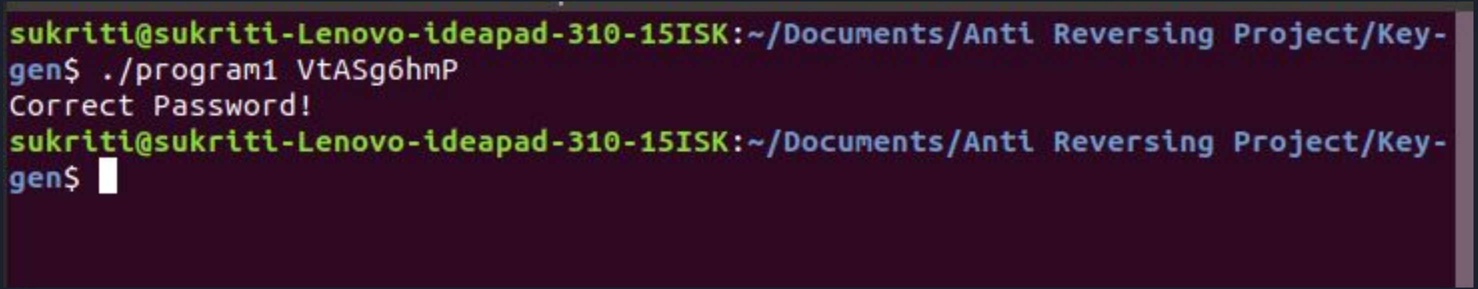
Eligible Passwords Received by the Key Generator



sukriti@sukriti-Lenovo-ideapad-310-15ISK: ~/Documents/Anti Reversing Project/Key-gen
File Edit View Search Terminal Help

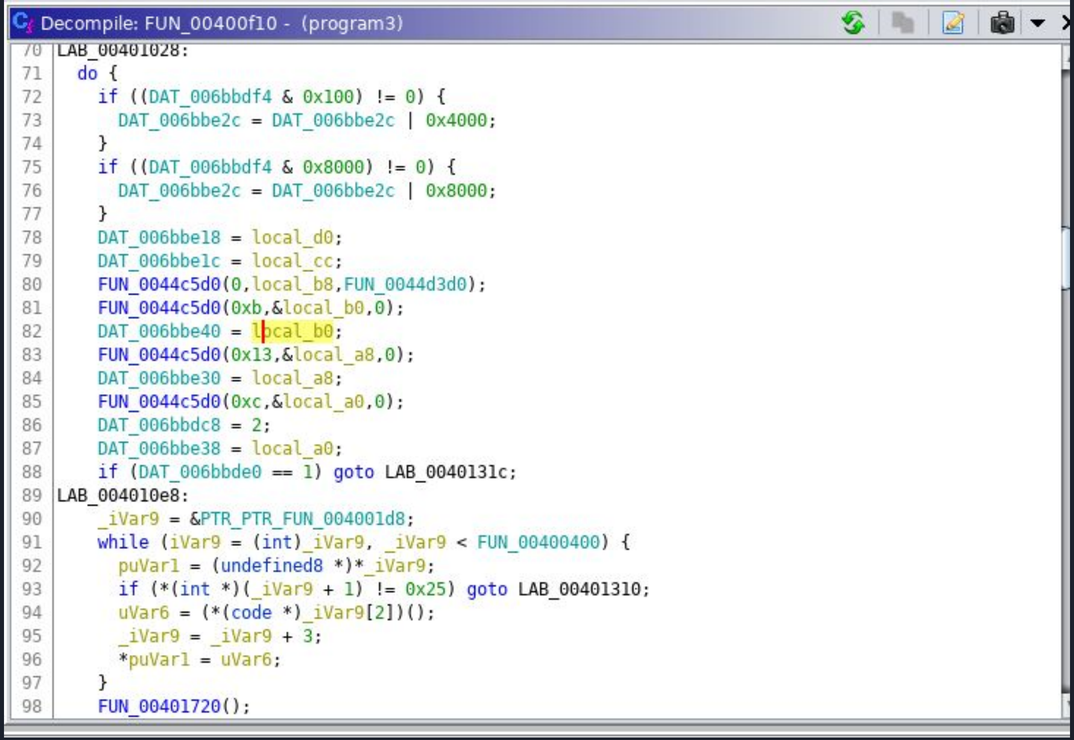
```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Key-gen$ python key-generator.py
Eligible Key option: eBPNYUcB0X
Eligible Key option: o2Ke4jmlX
Eligible Key option: trYygUjB
Eligible Key option: z9_Ezk6VX
Eligible Key option: IUdX0_LIJX
Eligible Key option: u2GvS-etc
Eligible Key option: L5TvBcB1fW
Eligible Key option: KsTf_2wYG
Eligible Key option: If-IyOLY6X
Eligible Key option: JNyocXMGQ
Eligible Key option: 4P02J_sClo
Eligible Key option: ToEvVjc1N
Eligible Key option: ss5qsL2VM
Eligible Key option: aWB9dj7tt
Eligible Key option: iIDergsy
Eligible Key option: am6CLeEJSF
Eligible Key option: GW-yrtJCi
Eligible Key option: a0KFXMpARV
Eligible Key option: hSNTNw2eg
Eligible Key option: 2yvLATp47C
Eligible Key option: XywqDtmB
Eligible Key option: GoUE-9C4NPU
Eligible Key option: 9MQOmdLyd
Eligible Key option: zeUPShhy
Eligible Key option: KkuQjgVG6
Eligible Key option: 76vzbWTTb
Eligible Key option: hK-GKipat
Eligible Key option: MIpcYv98w
Eligible Key option: szxpm7Cd
Eligible Key option: Z4bzK-H5Mt
Eligible Key option: UXwnlRNP2
Eligible Key option: mjgcKV7NY
Eligible Key option: ylXz4O4Dn
Eligible Key option: BH8zyO3oz
Eligible Key option: gI3BYcvtU
Eligible Key option: 6jgBji8Zr
Eligible Key option: RC06zJXhLU
```

Password Cracked



```
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Key-gen$ ./program1 VtASg6hmP
Correct Password!
sukriti@sukriti-Lenovo-ideapad-310-15ISK:~/Documents/Anti Reversing Project/Key-gen$
```

Code Snippet after Obfuscating the Code Making it Difficult to Decipher the Logic



```
LAB_00401028:
  do {
    if ((DAT_006bbdf4 & 0x100) != 0) {
      DAT_006bbe2c = DAT_006bbe2c | 0x4000;
    }
    if ((DAT_006bbdf4 & 0x8000) != 0) {
      DAT_006bbe2c = DAT_006bbe2c | 0x8000;
    }
    DAT_006bbe18 = local_d0;
    DAT_006bbe1c = local_cc;
    FUN_0044c5d0(0,local_b8,FUN_0044d3d0);
    FUN_0044c5d0(0xb,&local_b0,0);
    DAT_006bbe40 = local_b0;
    FUN_0044c5d0(0x13,&local_a8,0);
    DAT_006bbe30 = local_a8;
    FUN_0044c5d0(0xc,&local_a0,0);
    DAT_006bbdc8 = 2;
    DAT_006bbe38 = local_a0;
    if (DAT_006bbde0 == 1) goto LAB_0040131c;
LAB_004010e8:
    _iVar9 = &PTR_PTR_FUN_004001d8;
    while (iVar9 = (int)_iVar9, _iVar9 < FUN_00400400) {
      puVar1 = (undefined8 *)*_iVar9;
      if (*(int *)(_iVar9 + 1) != 0x25) goto LAB_00401310;
      uVar6 = (*(code *)_iVar9[2])();
      _iVar9 = _iVar9 + 3;
      *puVar1 = uVar6;
    }
    FUN_00401720();
```

References

- https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software
- https://www.codeproject.com/Articles/30815/An-Anti-Reverse-Engineering-Guide#VirtualMachines
- http://www.diva-portal.org/smash/get/diva2:1127760/FULLTEXT02
- https://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf
- https://www.youtube.com/playlist?list=PLhixgUqwRTjxglIswKp9mpkfPNfHkzyeN

Thankyou! Feel Free to Ask Questions!