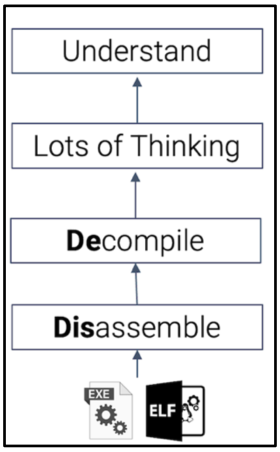
HO# 3.3: Reverse Engineering C Binaries



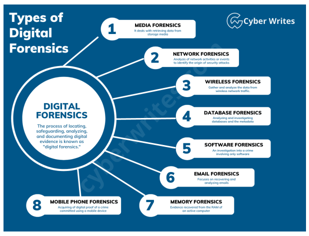
Reverse engineering in cybersecurity refers to the process of analyzing and disassembling software, hardware, or protocols to understand their inner workings, often with the goal of extracting valuable information, identifying vulnerabilities, malicious behavior or weaknesses. In order to perform reverse engineering, you need to have following skills:

- Proficiency in assembly language (e.g., x86/x64, ARM).
- Knowledge of binary file formats (e.g., ELF, PE).
- Familiarity with debugging tools and techniques.
- Understanding of operating systems, memory management, and processor architecture.
- Programming skills (e.g., C, C++, Python).

Key Purposes of Reverse Engineering

- **Malware Analysis:**
  - Understand the behavior of malware (viruses, ransomware, worms).
  - Identify how the malware spreads, its payload, and any vulnerabilities it exploits.
  - Develop antivirus signatures or patches to mitigate the threat.

- **Vulnerability Research:**
  - Analyze software or systems to uncover security flaws.
  - Help in the creation of exploits or, conversely, in fixing vulnerabilities through patches.
- **Software Cracking or Protection:**
  - Understand how software license checks or protections are implemented.
  - Develop mechanisms to strengthen software protection or, in some cases, bypass restrictions (though this can be illegal depending on the context).



- **Digital Forensics:**
  - Analyze compromised systems to determine the root cause of an incident.
  - Extract data or artifacts from damaged or encrypted systems.

How Reverse Engineering Works

- **Static Analysis**: It involves examining the software or system without running it, with the focus on assembly code, binary structure, strings, imports/exports. The tools for this task do not debug or trace execution flow, rather perform static disassembly, Decompilation, or code analysis. The tools used are `readelf, objdump, nm strings, file, hexedit, objcopy, strip, addr2line`.
- **Dynamic Analysis**: It involves analyzing the software or system while it runs, with the focus on its runtime behavior, memory usage and system interactions. The tools used are `GNU gdb (PEDA/GEF), valgrind, ptrace, strace, ltrace, ftrace, perf, frida`.
- **Hybrid Analysis Tools**: The tools used for both static and dynamic analysis are `Radare2, Cutter, IDA Pro, Ghidra, Binary Ninja`.
- **Protocol Analysis**: It involves intercepting and analyzing communication between devices or softwares. The tools used are `Wireshark, tcpdump, Burp Suite, Ettercap, Bettercap, scappy, Fiddler`.

A Hello to Reversing Binaries

Example 1: Hello World with `puts()` Function

```
//module3/3.3/rev1.c
#include <stdio.h>
void main(void){
    puts("Hello World!");
}
```

Create an executable of this simple C program:

`$ gcc rev1.c -o rev1`

Let us view the disassembly of this binary file using say `objdump` command. Right now, we are interested in the disassembly of `main` function shown below:

```
$ objdump -M intel -D ./rev1
$ objdump -M intel -D ./rev1 | grep -A 9 "<main>:"
```

```
0000000000001139 <main>:
    1139:       55                      push   rbp
    113a:       48 89 e5                mov    rbp, rsp
    113d:       48 8d 05 c0 0e 00 00    lea    rax, [rip+0xec0]
    1144:       48 89 c7                mov    rdi, rax
    1147:       e8 e4 fe ff ff          call   1030 <puts@plt>
    114c:       90                      nop
    114d:       5d                      pop    rbp
    114e:       c3                      ret
```

Description:

- The first two lines of assembly represent the function **prologue**. The `push rbp` stores the `rbp` register of caller on stack, so we can use `rbp` for our purpose. The `mov rbp,rsp` copies the value of `rsp`, which is pointing to the stack frame of `main`, to `rbp`. Now both `rsp` and `rbp` are pointing to the same location in memory.
- The third line: `lea rax,[rip+0xec0]` is the load effective address instruction that is calculating an address by adding the contents of `rip` with a constant. This is actually the address of the string passed to the `puts()` function. The `rip` register contains an address, and `0xec0` is the offset from this address to where the string starts. The string is stored in the `.rodata` section, which is part of the binary file and is loaded in memory with the program. It usually contains program constants.
- Let us examine the `.rodata` section.

```
$ objdump -d -s -j .rodata ./rev1
```

```
Disassembly of section .rodata:

0000000000002000 <_IO_stdin_used>:
   2000:       01 00 02 00 48 65 6c 6c 6f 20 57 6f 72 6c 64 21      ....Hello World!
```

- From above output, you can note that the `.rodata` section starts at address `0x2000` while the string starts four bytes farther, i.e., `0x2004`.
- At this moment the contents of rip register are `0x1144` and when you add `0xec0` in it you get `0x2004`, which is the address where the string starts. Thus, the `lea rax, [rip+0xec0]` will place `0x2004` inside the `rax` register, which is the starting address of the string.
- After this, the `mov rdi, rax` instruction will copy the address of the string inside the `rdi` register, which is the only and 1st argument to the `puts` function. ☺
- The fifth instruction calls the `puts()` function. This function is implemented in the GNU C library and its address is stored in the `.plt` section, hence the `puts@plt`. The `.plt` section (Procedure Linkage Table) is used by the dynamic linker/loader for handling function calls to dynamically linked libraries. For example, if the program calls a function in a shared library, the control is transferred to a `.plt` entry, which then calls the dynamic linker to look up the function's real address and jumps to it. On subsequent calls, the `.plt` entry has been updated to directly jump to the function's resolved address. The last two instructions are the function **epilog**, which restores the previously saved `rbp` value into `rbp`. Finally, we have the `ret` instruction ☺

Example 2: Hello World with `printf()` Function

Create an executable of this simple C program:
`$ gcc rev2.c -o rev2`

```c
//module3/3.3/rev2.c
#include <stdio.h>
int main(void){
    int x = 10;
    printf("x=%d\n", x);
    return 0;
}
```

Let us view the disassembly of this binary file using say `objdump` command. Right now, we are interested in the disassembly of `main` function shown below:

`$ objdump -M intel -D ./rev2 | grep -A 14 "<main>:"`

```
0000000000001139 <main>:
    1139:       55                      push   rbp
    113a:       48 89 e5                mov    rbp, rsp
    113d:       48 83 ec 10             sub    rsp, 0x10
    1141:       c7 45 fc 0a 00 00 00    mov    DWORD PTR [rbp-0x4], 0xa
    1148:       8b 45 fc                mov    eax, DWORD PTR [rbp-0x4]
    114b:       89 c6                   mov    esi, eax
    114d:       48 8d 05 b0 0e 00 00    lea    rax, [rip+0xeb0]
    1154:       48 89 c7                mov    rdi, rax
    1157:       b8 00 00 00 00          mov    eax, 0x0
    115c:       e8 cf fe ff ff          call   1030 <printf@plt>
    1161:       b8 00 00 00 00          mov    eax, 0x0
    1166:       c9                      leave
    1167:       c3                      ret
```

Description:

- Most of the code generated by the compiler is similar to the one as in the previous example, so we will discuss the lines which are new to us.
- The `113d: sub rsp,0x10` instruction is part of function prolog, which is actually creating space for the local integer variable named **x**. Although it needs 4 bytes but additional space is allocated for alignment purposes.
- The `1141: mov DWORD PTR [rbp-0x4],0xa` instruction is copying the decimal value 10 (`0xa`) on the stack for variable **x**. Since the variable has type integer, so it is stored in 4 bytes and its value starts 4 bytes below the `rbp`.
- Now before calling the `printf` function, we need to place its two arguments inside the `rdi` and `rsi` registers respectively, starting from right to left.
  - The `1148: mov eax, DWORD PTR [rbp-0x4]` is copying the value of variable x from stack into the `eax` register. Since this will be second argument to the `printf()` function, so in the next instruction, it is copied to `rsi` as shown `114b: mov esi,eax`
  - The `114d: lea rax, [rip+0xeb0]` is loading the address of the format string inside `rax` register. Since this will be first argument to the `printf()` function, in the next instruction it is copied to `rdi` as shown `1154: mov rdi,rax`
  - Next `1157: mov eax, 0x0` the compiler places a zero in the `eax` register and then call to `printf` is made. The System-V AMD64 ABI specifies that before a function from the standard C library is called, the value in the `rax` register must specify the number of floating-point arguments passed to the function in XMM registers. So `rax` register is explicitly set to 0, indicating that no floating-point arguments are passed to `printf`.
- Next `1161: mov eax, 0x0` the compiler places a zero in the `eax` (or `rax`) register as the return value from the `main()` function.
- Finally, we have the function epilog containing the `leave` and the `ret` instructions, that will roll back the FSF.

Example 3: Hello World with Arrays

Create an executable of this simple C program:

$ gcc rev3.c -o rev3

```c
//module3/3.3/rev3.c
#include <stdio.h>
int main(){
    int x[4] = {1,2,3,4};
    printf("x[1] = %d\n",x[1]);
    return 0;
}
```

Let us view the disassembly of this binary file using say `objdump` command. Right now, we are interested in the disassembly of `main` function shown below:

$objdump -M intel -D ./rev3 | grep -A 17 "<main>:"

```
0000000000001139 <main>:
    1139:      55                      push   rbp
    113a:      48 89 e5                mov    rbp, rsp
    113d:      48 83 ec 10             sub    rsp, 0x10
    1141:      c7 45 f0 01 00 00 00    mov    DWORD PTR [rbp-0x10], 0x1
    1148:      c7 45 f4 02 00 00 00    mov    DWORD PTR [rbp-0xc], 0x2
    114f:      c7 45 f8 03 00 00 00    mov    DWORD PTR [rbp-0x8], 0x3
    1156:      c7 45 fc 04 00 00 00    mov    DWORD PTR [rbp-0x4], 0x4
    115d:      8b 45 f4                mov    eax, DWORD PTR [rbp-0xc]
    1160:      89 c6                   mov    esi, eax
    1162:      48 8d 05 9b 0e 00 00    lea    rax, [rip+0xe9b]
    1169:      48 89 c7                mov    rdi, rax
    116c:      b8 00 00 00 00          mov    eax, 0x0
    1171:      e8 ba fe ff ff          call   1030 <printf@plt>
    1176:      b8 00 00 00 00          mov    eax, 0x0
    117b:      c9                      leave
    117c:      c3                      ret
```

Description:

- The `1141: mov DWORD PTR [rbp-0x10],0x1` instruction is copying the decimal value 1 (`0x1`) on the stack as the first integer value inside the array **x**. Since the variable has type integer, so it is stored in 4 bytes and its value starts 0x10 bytes below the `rbp`.
- Similarly, the `1148: mov DWORD PTR [rbp-0xc],0x2` instruction is copying the decimal value 2 (`0x2`) on the stack as the second integer value inside the array **x**. Note that the value is stored starting at address `0xc` bytes below the `rbp`. In the same fashion, all the four values. Of the array are stored on the stack.
- Now before calling the `printf` function, we need to place its two arguments inside the `rdi` and `rsi` registers respectively, starting from right to left.
  - Following two instructions places the 2nd argument inside `esi`, which is `x[1]` i.e., 2:
    ```
    mov eax, DWORD PTR [rbp-0xc]
    mov esi, eax
    ```
  - Next two instruction places the 1st argument inside `esi`, which is address of format string:
    ```
    lea rax,[rip-0xe9b]
    mov rdi,rax
    ```
  - Next `116c: mov eax, 0x0` the compiler places a zero in the `eax` register to specify that XMM registers are not involved in arguments passing.
- Remaining code is already discussed in previous examples.

Example 4: Hello World with if---else

Create an executable of this simple C program:
$ gcc rev4.c -o rev4

```
//module3/3.3/rev4.c
#include <stdio.h>
int main(){
   int x = 10;
   int y = 5;
   if(x<=100){
      y = y - 3;
      printf("Less than\n");
   }
  else{
      y = y + 3;
      printf("Greater than\n");
}
  return 0;
}
```

Let us view the disassembly of this binary file using say `objdump` command. Right now, we are interested in the disassembly of `main` function shown below:

$objdump -M intel -D ./rev4 | grep -A 20 "<main>:"

```
0000000000001139 <main>:
    1139:       55                      push   rbp
    113a:       48 89 e5                mov    rbp, rsp
    113d:       48 83 ec 10             sub    rsp, 0x10
    1141:       c7 45 fc 0a 00 00 00    mov    DWORD PTR [rbp-0x4], 0xa
    1148:       c7 45 f8 05 00 00 00    mov    DWORD PTR [rbp-0x8], 0x5
    114f:       83 7d fc 64             cmp    DWORD PTR [rbp-0x4], 0x64
    1153:       7f 15                   jg     116a <main+0x31>
    1155:       83 6d f8 03             sub    DWORD PTR [rbp-0x8], 0x3
    1159:       48 8d 05 a4 0e 00 00    lea    rax, [rip+0xea4]
    1160:       48 89 c7                mov    rdi, rax
    1163:       e8 c8 fe ff ff          call   1030 <puts@plt>
    1168:       eb 13                   jmp    117d <main+0x44>
    116a:       83 45 f8 03             add    DWORD PTR [rbp-0x8],0x3
    116e:       48 8d 05 99 0e 00 00    lea    rax,[rip+0xe99]
    1175:       48 89 c7                mov    rdi,rax
    1178:       e8 b3 fe ff ff          call   1030 <puts@plt>
    117d:       b8 00 00 00 00          mov    eax,0x0
    1182:       c9                      leave
    1183:       c3                      ret
```

To Do:
Understand the disassembly, and write down description of above compiler generated assembly code.
Happy Learning ☺

Example 5: Hello World with Loops

Create an executable of this simple C program:
$ gcc rev5.c -o rev5

```
//module3/3.3/rev5.c
#include <stdio.h>
int main(){
    int i = 0;
    int limit = 5;
    for(i = 0;i<limit; i++){
      printf("%d ",i);
    }
    printf("\n");
    return 0;
}
```

Let us view the disassembly of this binary file using say `objdump` command. Right now, we are interested in the disassembly of `main` function shown below:

$ objdump -M intel -D ./rev5 | grep -A 22 "<main>:"

```
0000000000001149 <main>:
    1149:       55                      push   rbp
    114a:       48 89 e5                mov    rbp, rsp
    114d:       48 83 ec 10             sub    rsp, 0x10
    1151:       c7 45 fc 00 00 00 00    mov    DWORD PTR [rbp-0x4], 0x0
    1158:       c7 45 f8 05 00 00 00    mov    DWORD PTR [rbp-0x8], 0x5
    115f:       c7 45 fc 00 00 00 00    mov    DWORD PTR [rbp-0x4], 0x0
    1166:       eb 1d                   jmp    1185 <main+0x3c>
    1168:       8b 45 fc                mov    eax, DWORD PTR [rbp-0x4]
    116b:       89 c6                   mov    esi, eax
    116d:       48 8d 05 90 0e 00 00    lea    rax, [rip+0xe90]
    1174:       48 89 c7                mov    rdi, rax
    1177:       b8 00 00 00 00          mov    eax, 0x0
    117c:       e8 bf fe ff ff          call   1040 <printf@plt>
    1181:       83 45 fc 01             add    DWORD PTR [rbp-0x4], 0x1
    1185:       8b 45 fc                mov    eax, DWORD PTR [rbp-0x4]
    1188:       3b 45 f8                cmp    eax, DWORD PTR [rbp-0x8]
    118b:       7c db                   jl     1168 <main+0x1f>
    118d:       bf 0a 00 00 00          mov    edi,0xa
    1192:       e8 99 fe ff ff          call   1030 <putchar@plt>
    1197:       b8 00 00 00 00          mov    eax,0x0
    119c:       c9                      leave
    119d:       c3                      ret
```

To Do:
Understand the disassembly, and write down description of above compiler generated assembly code.
Happy Learning ☺

Example 6: Hello World with Function Call

Create an executable of this simple C program:
`$ gcc rev6.c -o rev6`

```c
//module3/3.3/rev6.c
#include <stdio.h>
int foo(int,int);
int main(){
   int val1 = 10;
   int val2 = 20;
   int sum = foo(val1,val2);
   printf("sum is: %d\n",sum);
   return 0;
}
int foo(int a, int b){
   int out = a + b;
   return out;
}
```

Let us view the disassembly of this binary file using say `objdump` command. Right now, we are interested in the disassembly of `main` function shown below:

```
$ objdump -M intel -D ./rev6 | grep -A 33 "<main>:"

0000000000001139 <main>:
    1139:      55                        push   rbp
    113a:      48 89 e5                  mov    rbp, rsp
    113d:      48 83 ec 10               sub    rsp, 0x10
    1141:      c7 45 fc 0a 00 00 00      mov    DWORD PTR [rbp-0x4], 0xa
    1148:      c7 45 f8 14 00 00 00      mov    DWORD PTR [rbp-0x8], 0x14
    114f:      8b 55 f8                  mov    edx, DWORD PTR [rbp-0x8]
    1152:      8b 45 fc                  mov    eax, DWORD PTR [rbp-0x4]
    1155:      89 d6                     mov    esi, edx
    1157:      89 c7                     mov    edi, eax
    1159:      e8 23 00 00 00            call   1181 <foo>
    115e:      89 45 f4                  mov    DWORD PTR [rbp-0xc], eax
    1161:      8b 45 f4                  mov    eax, DWORD PTR[rbp-0xc]
    1164:      89 c6                     mov    esi, eax
    1166:      48 8d 05 97 0e 00 00      lea    rax, [rip+0xe97]
    116d:      48 89 c7                  mov    rdi, rax
    1170:      b8 00 00 00 00            mov    eax, 0x0
    1175:      e8 b6 fe ff ff            call   1030 <printf@plt>
    117a:      b8 00 00 00 00            mov    eax, 0x0
    117f:      c9                        leave
    1180:      c3                        ret

0000000000001181 <foo>:
    1181:      55                        push   rbp
    1182:      48 89 e5                  mov    rbp,rsp
    1185:      89 7d ec                  mov    DWORD PTR [rbp-0x14], edi
    1188:      89 75 e8                  mov    DWORD PTR [rbp-0x18], esi
    118b:      8b 55 ec                  mov    edx, DWORD PTR [rbp-0x14]
    118e:      8b 45 e8                  mov    eax, DWORD PTR [rbp-0x18]
    1191:      01 d0                     add    eax, edx
    1193:      89 45 fc                  mov    DWORD PTR [rbp-0x4], eax
    1196:      8b 45 fc                  mov    eax, DWORD PTR [rbp-0x4]
    1199:      5d                        pop    rbp
    119a:      c3                        ret
```

Example 7: Cracking a Binary

```
//module3/3.3/rev7.c
#include <string.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
     if(argc==2) {
            printf("Checking License: %s\n", argv[1]);
            if(strcmp(argv[1], "kakamanna")==0)
                  printf("Access Granted!\n");
            else
                  printf("WRONG!\n");
      }else
            printf("Usage: <key>\n");
      return 0;
}
```

```
$ objdump -M intel -d ./rev7 | grep -A 39 "<main>:"
0000000000001159 <main>:
    1159:   55                        push   rbp
    115a:   48 89 e5                  mov    rbp,rsp
    115d:   48 83 ec 10               sub    rsp,0x10
    1161:   89 7d fc                  mov    DWORD PTR [rbp-0x4],edi
    1164:   48 89 75 f0               mov    QWORD PTR [rbp-0x10],rsi
    1168:   83 7d fc 02               cmp    DWORD PTR [rbp-0x4],0x2
    116c:   75 65                     jne    11d3 <main+0x7a>
    116e:   48 8b 45 f0               mov    rax,QWORD PTR [rbp-0x10]
    1172:   48 83 c0 08               add    rax,0x8
    1176:   48 8b 00                  mov    rax,QWORD PTR [rax]
    1179:   48 89 c6                  mov    rsi,rax
    117c:   48 8d 05 81 0e 00 00      lea    rax,[rip+0xe81]
    1183:   48 89 c7                  mov    rdi,rax
    1186:   b8 00 00 00 00            mov    eax,0x0
    118b:   e8 b0 fe ff ff            call   1040 <printf@plt>
    1190:   48 8b 45 f0               mov    rax,QWORD PTR [rbp-0x10]
    1194:   48 83 c0 08               add    rax,0x8
    1198:   48 8b 00                  mov    rax,QWORD PTR [rax]
    119b:   48 8d 15 74 0e 00 00      lea    rdx,[rip+0xe74]
    11a2:   48 89 d6                  mov    rsi,rdx
    11a5:   48 89 c7                  mov    rdi,rax
    11a8:   e8 a3 fe ff ff            call   1050 <strcmp@plt>
    11ad:   85 c0                     test   eax,eax
    11af:   75 11                     jne    11c2 <main+0x69>
    11b1:   48 8d 05 68 0e 00 00      lea    rax,[rip+0xe68]
    11b8:   48 89 c7                  mov    rdi,rax
    11bb:   e8 70 fe ff ff            call   1030 <puts@plt>
    11c0:   eb 20                     jmp    11e2 <main+0x89>
    11c2:   48 8d 05 67 0e 00 00      lea    rax,[rip+0xe67]
    11c9:   48 89 c7                  mov    rdi,rax
    11cc:   e8 5f fe ff ff            call   1030 <puts@plt>
    11d1:   eb 0f                     jmp    11e2 <main+0x89>
    11d3:   48 8d 05 6b 0e 00 00      lea    rax,[rip+0xe6b]
    11da:   48 89 c7                  mov    rdi,rax
    11dd:   e8 4e fe ff ff            call   1030 <puts@plt>
    11e2:   b8 00 00 00 00            mov    eax,0x0
    11e7:   c9                        leave
    11e8:   c3                        ret
```

A Hello to Crack-Me CTFs (`ctf1, ctf2, ctf3, ctf4`)

Download these binaries from the Cyber Security course page from our course link: https://www.arifbutt.me/courses/. For your ease I have compiled all these binaries with `-ggdb` option of `gcc` and the symbols are not stripped from the binary using the `strip` utility. Perform static as well as dynamic analysis of these binaries one by one and see if you can crack the hidden key in each of the binary. Follow the following steps to achieve your goal:

a. Execute the binary, understand its behavior, and try your luck by giving different sample inputs.
b. Use `file` utility and check the details about the file type (whether it is a Linux or Windows binary, whether it is 32-bit or 64-bit, whether data is stored in little endian or big endian and so on).
c. Use `less` utility and try reading the contents of the binary, and see what all information is visible.
d. Use `hexdump -C` utility to
e. Use `ldd` utility to display the shared object files that are linked with this binary.
f. Use `readelf -h` utility to display the header that will contain metadata about the ELF.
g. Use `nm` utility to display the header that will contain metadata about the ELF.
h. Use `strings` utility to extract/display ASCII/Unicode sequences of printable characters embedded inside the binary.
i. Use `objdump -d -M intel` utility to display the disassembly of the main function and any other user defined function.
j. Use `objdump -dsj .rodata` to display the strings used by `printf` and `puts` functions.
k. Use `ltrace` and `strace` to display the strings
l. Finally load the binary inside a debugger and perform dynamic analysis to understand its behavior in depth. For complex CTFs, you cannot capture the hidden key by just using the static analysis tools, and you have to perform dynamic analysis using some disassembler/decompiler of your choice like `gdb (PEDA/GEF), ghidra, radare2, cutter, OllyDbg, binaryninja` and so on.

To Do:

`ctf5`: Download this binary from the Cyber Security course page from your course link: https://www.arifbutt.me/courses/. Perform all the steps that we have performed to do static as well as dynamic analysis of the binaries inside the class on this binary as well. A sample run of the binary is shown in the screenshot, which requires you to give three input values, and on entering all three correctly, it will display the message: "**[+] Good work!**"

```
┌──(kali㉿kali)-[~/IS/module3/3.2/cprogs/reversing]
└─$ ./ctf

[*] Enter the Magic Value: [redacted]
[*] Generating an Obfuscated string.
[+] De-obfuscated string: :
[?] Enter the De-Obfuscated string: [redacted]

[*] Generating an dynamic string
[+] Dynamic string Generated
[?] Enter the string: [redacted]

[+] Good work!
```

- **Task 1:** The first part of the program involves determining a hardcoded string in the main function. This is the simplest task.
- **Task 2:** The second task involves determining a string that is XORed with a key. Your goal is to figure out how the program obfuscates and later de-obfuscates this string. **Hint:** Set a breakpoint at the function responsible for handling the obfuscated string. Step through the function, paying close attention to the XOR operation. Identify the key used and reverse the XOR to get the original string. Inspect the function's arguments and local variables to analyze how the obfuscated string is processed.
- **Task 3:** The third task involves determining a dynamically generated string where the case of each character is flipped. **Hint:** Set a breakpoint inside the function that generates the dynamic string. Step through the code to see how the case of each character is flipped (likely using XOR with 0x20). Inspect the program's memory and registers to view the generated string before and after case-flipping.

Techniques Used to Evade Static and Dynamic Analysis

Dear students, in real life you will seldom get a binary like the ones I have shared. In order to make the binaries, harder to analyze, examine, crack, and debug, developers implement various defensive techniques like:

- **Code Obfuscation:** Involves transforming the program's structure to impede static analysis. Techniques include *string encryption* (hiding sensitive strings by decrypting them at runtime), *function in-lining* (replacing function calls with their actual code to increase complexity), *dead code insertion* (adding irrelevant instructions to mislead analysts), and *control flow obfuscation* (manipulating execution paths to obscure program logic).

- **Anti-Debugging Mechanisms:** Detects and disrupts debugging attempts using methods such as blocking tracing with `ptrace`, process status checks (examining /proc/self/status for a nonzero TracerPid value), and software breakpoints traps (inserting `int 3` or `ud2` instructions to crash debuggers).

- **Anti-Static Analysis Protections:** Prevents easy disassembly and inspection by *stripping symbols* (removing function names and debugging information), *binary packing* (compressing and encrypting executables using tools like Ultimate Packer for Executables UPX), and *code section encryption* (requiring runtime decryption to access executable code).

- **Anti-Dynamic Analysis Techniques:** Detects execution in controlled environments such as sandboxes or virtual machines through system artifact checks (identifying VM-specific hardware or registry keys), timing-based detection (measuring instruction execution time to reveal slowdowns caused by emulation), and system call fingerprinting (observing deviations in syscall behavior under analysis tools).

- **Self-Modifying and Polymorphic Code:** Implements runtime code modification (altering instructions dynamically to disrupt pattern-based detection), polymorphism (changing instruction sequences while preserving functionality), and metamorphism (rewriting entire code segments across executions) to evade signature-based and heuristic analysis.

Further Learning:

- For further Learning visit https://crackmes.one/, and download crackmes, which are small programs designed to test a hacker's reverse engineering skill.
- You can also visit https://pwn.college/, which is an education platform for students to learn about and practice core cybersecurity concepts in a hands-on fashion.
- Visit https://tryhackme.com/, which is an online platform designed for learning cybersecurity through hands-on practice in an interactive gamified environment. It provides CTF-style challenges where users solve puzzles, exploit vulnerabilities, and submit "flags" as proof of success.