ISLA MUKHEEF

COMPUTER ENGINEERING DERPTMENT **AREL UNIVERSITY**

Content:

1. Introduction

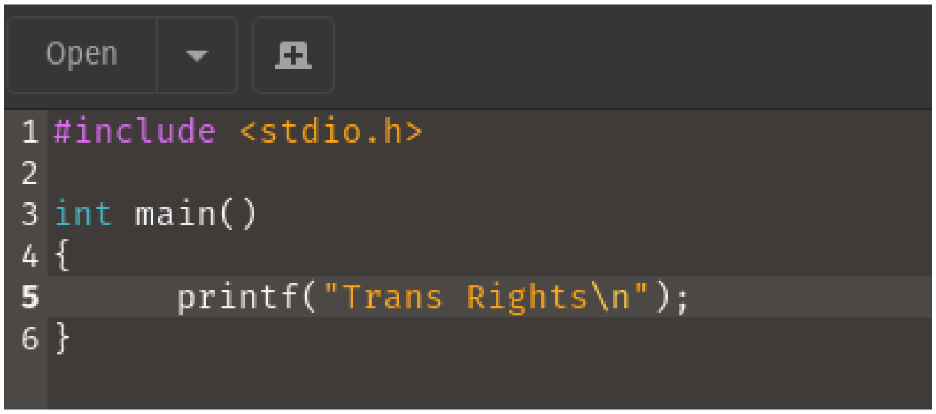
I have always wanted to know how people crack games or programs or how do they discover vulnerabilities in these applications. I thought by learning programming I would understand how it works but it was not the case. I tried to learn assembly and it did not help, there is always a missing thing that I could not find. It was the mindset, people who try to teach you something they will teach you from their perspective and they will not teach you thinks they do not understand when I realized that I understood that I was looking in the wrong place and I need to find people who do break things for fun and when I did that, I found my way into reverse engineering. At the beginning it was not cool because I was looking at an assembly code of a program and trying to understand the idea behind it and I was quite sure the developers themselves would not understand what they wrote, I was right, programmers write code, that is their job I should break it and show them that they did it wrong so I need them to make mistakes or write unnecessary lines that may lead to RCE or logical vulnerabilities. I started doing some CTF and saw how happy I am when I solve a challenge and from that moment, I started to read books about reverse engineering but mostly about malware since their Authers usually do it in a smart way to hide it activity. I do not call myself professional and I am far from that, but I will try to explain and demonstrate the basics of reverse engineering and how it works.

1.1. What is reverse engineering:

‘Reverse engineering is a process or method through the application of which one attempts to understand through deductive reasoning how a device, process, system, or piece of software accomplishes a task with very little (if any) insight into exactly how it does’ [1]. Let’s say you have a friend who sent you a cool program that they made which takes two parameters but makes use of only one or it has a trick etc... And you don't have access to the source code. Reversing the program would be the only solution I would think of to understand why and how it works. So Reverse engineering is when someone tries to understand an idea or something about a file that they don't have access to it source code. Keep in mind that in some cases you may have access to the source code which would be handier, and it helps you a lot, but I don't think someone would bother to reverse an open-source program!

1.2. What is Assembly Language:

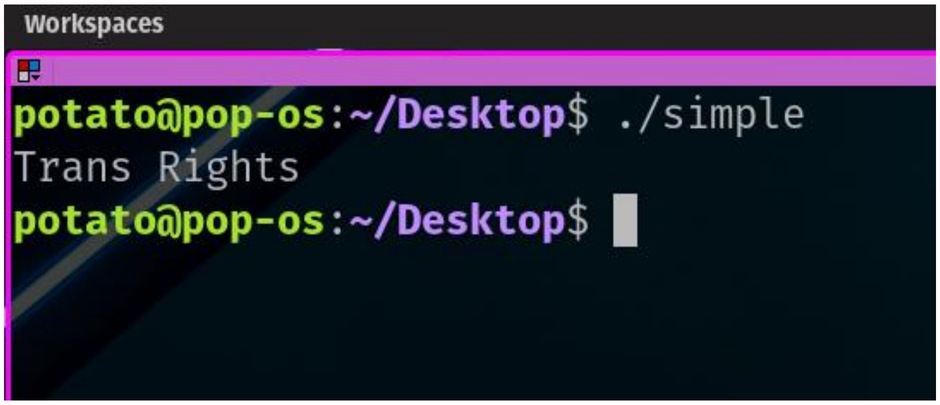
Basically, Assembly is a low-level programming language. What does low-level mean? It’s a way of saying that it’s of close to computer language (machine language) and it’s not human readable so it may be hard for us to understand. We need to have basic knowledge about assembly to move forward but understanding everything is not required. By going into details explaining a simple Hello Word program was written in C++ will be a good way to start, also the Assembly syntax will be Intel syntax[2].



```
#include <stdio.h>

int main()
{
        printf("Trans Rights\n");
}
```

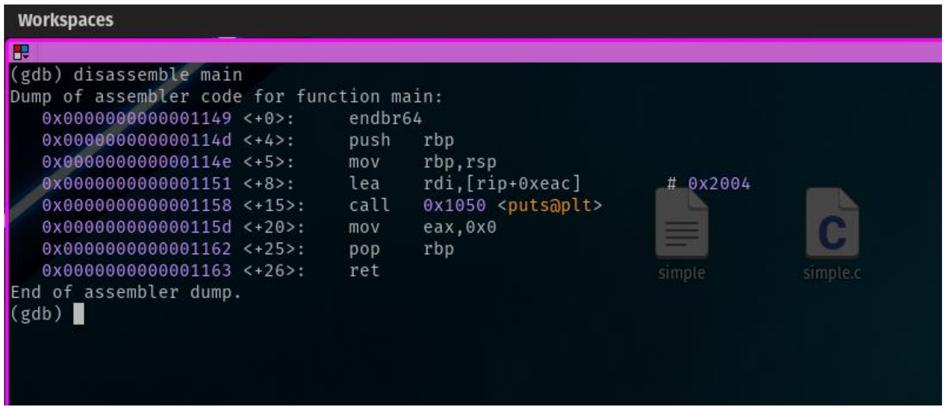
This is our code and when we run it, we get this.

[1] https://en.wikipedia.org/wiki/Reverse_engineering
[2] https://en.wikipedia.org/wiki/X86_assembly_language



```
potato@pop-os:~/Desktop$ ./simple
Trans Rights
potato@pop-os:~/Desktop$ 
```

Now let’s look at the Assembly.



```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000001149 <+0>:     endbr64
   0x000000000000114d <+4>:     push   rbp
   0x000000000000114e <+5>:     mov    rbp,rsp
   0x0000000000001151 <+8>:     lea    rdi,[rip+0xeac]        # 0x2004
   0x0000000000001158 <+15>:    call   0x1050 <puts@plt>
   0x000000000000115d <+20>:    mov    eax,0x0
   0x0000000000001162 <+25>:    pop    rbp
   0x0000000000001163 <+26>:    ret
End of assembler dump.
(gdb) 
```

Note: with GDB you should change the disassemble flavor to intel by typing
‘**set disassemble-flavor intel**’

Then you should write ‘**disassemble main**’. (I wrote main because I already know that there are no other functions you usually need to check before doing that).

Let's try to understand what this assembly code really means.

1- It starts with **endbr64** which is a **NOP** instruction. It’s there to prevent **control-flow attacks**[3]

2- We use **PUSH** instruction to push something onto the stack so here we're PUSHing the **rbp**(base pointer) to preserve current frame pointer.

3- **MOV rbp ,rsp** Here we are creating new stack pointer by moving the **rsp** to the **rbp**.

4- **LEA rdi,[rip+0xeac]** Here we're putting the text (Trans Rights) into the register **rdi**. I'm pretty sure you’re confused now so let me explain it more. We use [] to say the address of something, so saying [rip+0xeac] would be the address of rip + 0xace which is equal to 0x2004 and the address 0x2004 is a pointer to our value that will be printed. We then placed the address in the rdi so rdi has our value now. Just to be clearer sometimes the compiler uses different ways like

[3] https://www.linuxplumbersconf.org/event/2/contributions/147/attachments/72/83/CET-LPC-2018.pdf

pushing the value into the stack before calling a function. Or like now using the rdi register to store the address of the first argument of a function.

5- **CALL 0x1050** Here we're calling the printf function also don't forget that we already have the address of the value that will be printed saved in the rdi register.

6- **MOV eax ,0x0** Here we're storing the return value from the printf function into the eax register.

7- **POP rbp** in here we're popping the rbp to terminate the function.

8- **RET** sets rip to the value stored at rsp. Which means that it pops the return address off the stack and returns control to that location.

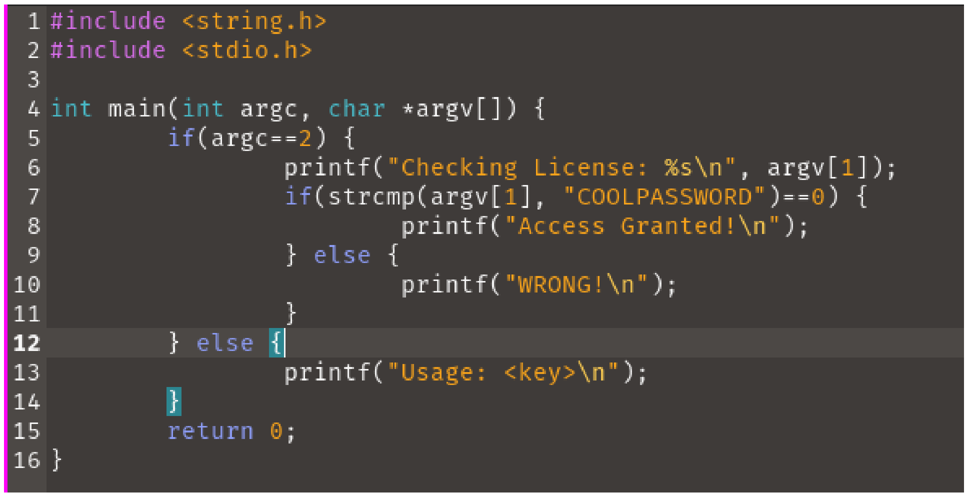
This is how the assembly works.

2. So, you want to crack a program!

2.1. Static analysis

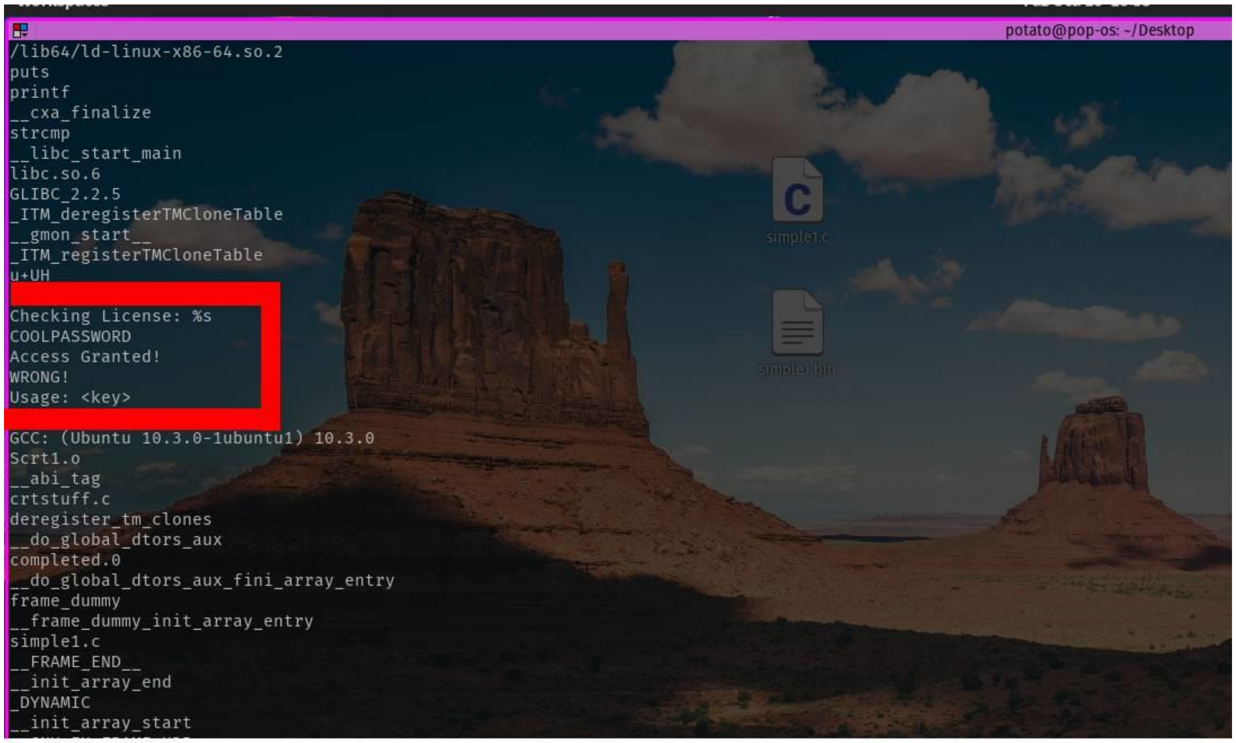
Static analysis is basically when you do not run the program, The hello world program that was discussed before is the best example. Looking at a file assembly code and trying to understand the idea behind it.

In this section we are going to do a static analysis to get a key that is required to use this program.



```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], "COOLPASSWORD")==0) {
			printf("Access Granted!\n");
		} else {
			printf("WRONG!\n");
		}
	} else {
		printf("Usage: <key>\n");
	}
	return 0;
}
```

This is the source code and the password going to be COOLPASSWORD but let us just act as if we do not really know the password and we need to get it. There is a lot of approaches that we can try but the first thing we should look at is the strings. So where do the strings really go after we write a program? Well, it gets stored in the data section and we can just look it. There is different ways to that, but I will do it with Linux strings[4] command. '*Strings filename | less* 'will give us these results

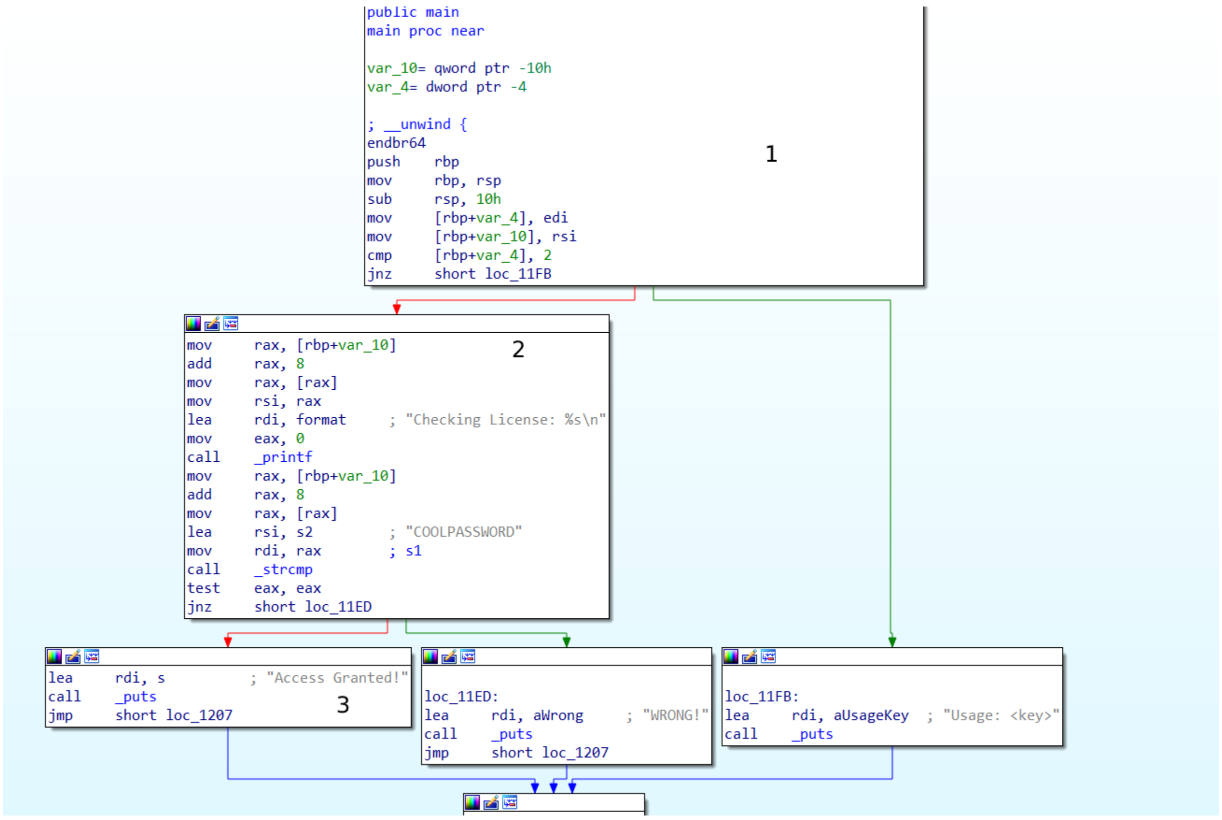


```
/lib64/ld-linux-x86-64.so.2
puts
printf
__cxa_finalize
strcmp
__libc_start_main
libc.so.6
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
u+UH

Checking License: %s
COOLPASSWORD
Access Granted!
WRONG!
Usage: <key>

GCC: (Ubuntu 10.3.0-1ubuntu1) 10.3.0
Scrt1.o
__abi_tag
crtstuff.c
deregister_tm_clones
__do_global_dtors_aux
completed.0
__do_global_dtors_aux_fini_array_entry
frame_dummy
__frame_dummy_init_array_entry
simple1.c
__FRAME_END__
__init_array_end
_DYNAMIC
__init_array_start
```

Most of the results is not useful now but look at the highlight part these all the strings that we saw in our source code and just by guessing I would say COOLPASSWORD is the password. That

[4] https://linux.die.net/man/1/strings

was simple but reading the assembly would be more useful. I will use IDA PRO[5] instead of gdb as disassembler.



```
public main
main proc near

var_10= qword ptr -10h
var_4= dword ptr -4

; __unwind {
endbr64
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_4], edi
mov     [rbp+var_10], rsi
cmp     [rbp+var_4], 2
jnz     short loc_11FB
```

```
mov     rax, [rbp+var_10]
add     rax, 8
mov     rax, [rax]
mov     rsi, rax
lea     rdi, format     ; "Checking License: %s\n"
mov     eax, 0
call    _printf
mov     rax, [rbp+var_10]
add     rax, 8
mov     rax, [rax]
lea     rsi, s2         ; "COOLPASSWORD"
mov     rdi, rax        ; s1
call    _strcmp
test    eax, eax
jnz     short loc_11ED
```

```
lea     rdi, s          ; "Access Granted!"
call    _puts
jmp     short loc_1207
```

```
loc_11ED:
lea     rdi, aWrong     ; "WRONG!"
call    _puts
jmp     short loc_1207
```

```
loc_11FB:
lea     rdi, aUsageKey  ; "Usage: <key>"
call    _puts
```

Here you can see how the program really works. To get the word access granted the program should go from 1 to 3. The first box is about checking if we entered a password or not you can see before it does the ' jnz   short loc_11FB' it going to compare the number 2 against [rbp+var_4] which it is checking if argc has two parameters. If when running the program, we did not give it a parameter(password) it will take us to loc_11FB. Let us say we gave it a password now it going to move to box number 2. We see that there are a lot of familiar strings that we saw in the source code. At the end of the box, we see 'jnz   short loc_11ED' and before that we can see a call to _strcmp so here we can assume that it is going to check on two strings and if they match it will print Access Granted! Otherwise, it is going to take us to loc_11ED. IDA is trying to help us so it gave us the string that is stored in the register s2 'lea   rsi, s2   ; "COOLPASSWORD"' the call to _strcmp should be done with two parameters and it will be using rsi and rdi. The password we wrote gonna be in the register rdi and the password going to be in the rsi. So now without running the program we managed to get a lot of information about it, and we were able to find the required password. In this case we did not need to check on a lot of things but in case that we are dealing with a malware sample it's going to be more than just looking at the assembly, I did author an article about reverse engineering the WannaCry

[5] https://hex-rays.com/ida-pro/

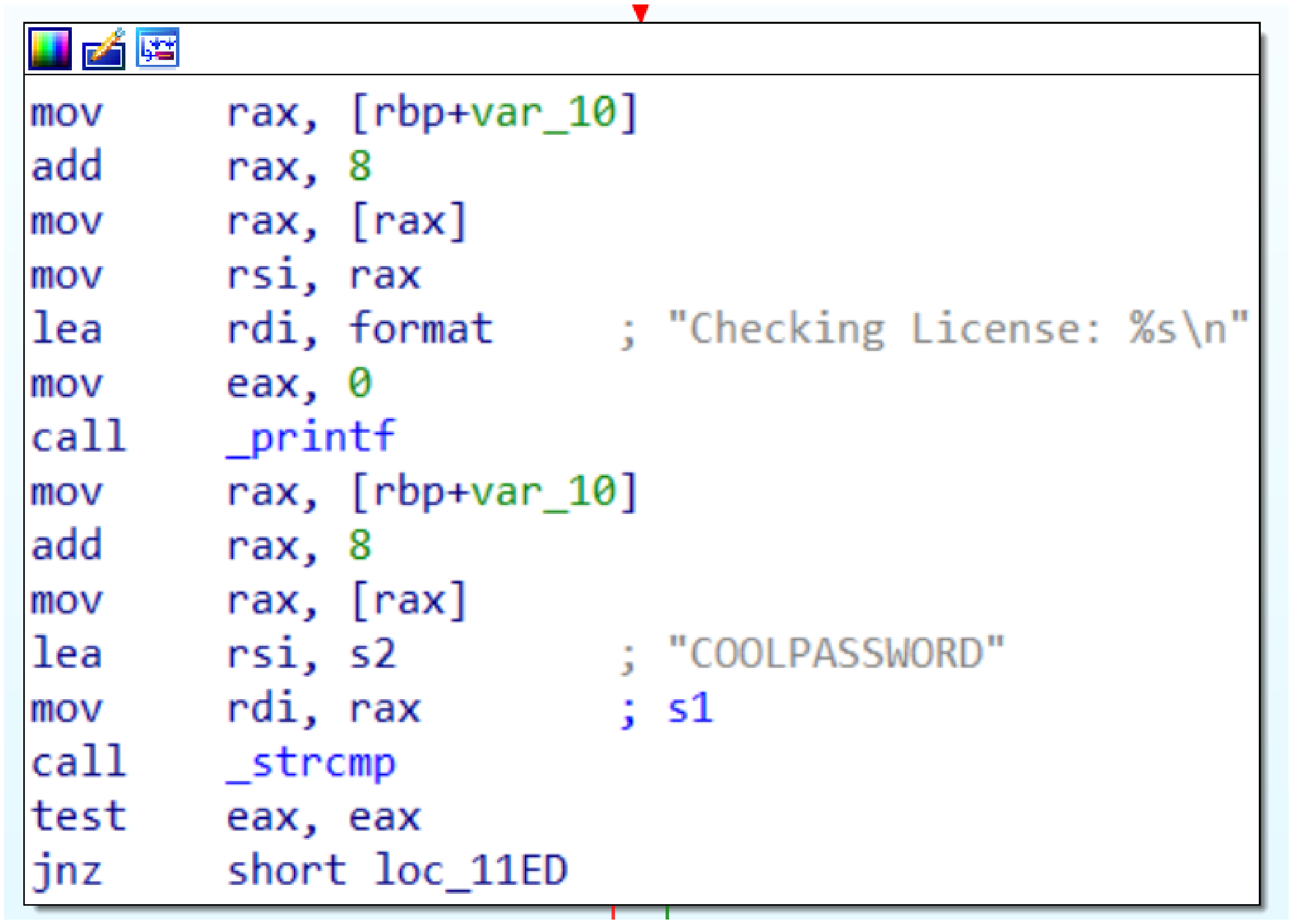
ransomware (WannaCry Static Analysis[6]) I talked about more detailed things there on how to do static reverse engineering.

2.1.Dynamic analysis

Dynamic analysis is running the program and trying to see what it does in action. It is the easiest way to understand how the program really works. I do not really like to run any file or use any that I got online, so I just run it in a virtual machine and see what it does. Usually, it is more than just running. Sometimes you need to keep your eyes on a lot of things like the register keys in if it is windows program or if it is going to spawn a service or if it is going to inject itself into another service or program etc... If someone wants to get into cyber security the first thing, they should learn is not to trust anyone, anything. By that I mean even if you going to download a 'trusted' program it may do a lot more than it say in the documentation. That is why open-source programs are preferred by a lot of people. Doing dynamic analysis on the program that we wrote is going to give us more than one way to bypass the password.

From our static analysis we already know that in the second box to get to the third box and get the word "Access Granted!" We are going to compare two strings. So, what going to happen if we manipulate the results of the _strcmp!

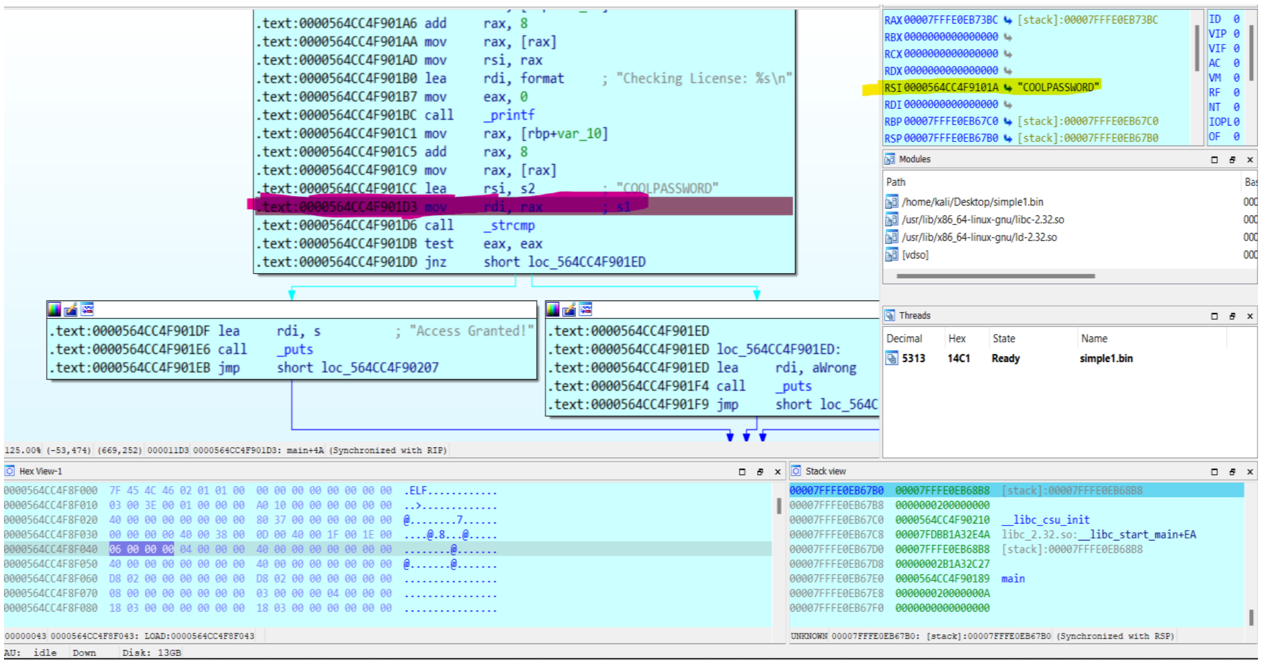
[6] https://islamukheef.medium.com/wannacry-static-analysis-1f390ce6e360



```
mov     rax, [rbp+var_10]
add     rax, 8
mov     rax, [rax]
mov     rsi, rax
lea     rdi, format     ; "Checking License: %s\n"
mov     eax, 0
call    _printf
mov     rax, [rbp+var_10]
add     rax, 8
mov     rax, [rax]
lea     rsi, s2         ; "COOLPASSWORD"
mov     rdi, rax        ; s1
call    _strcmp
test    eax, eax
jnz     short loc_11ED
```

After the _strcmp function finishes it will return the results in eax register and after it will do the result' test eax, eax' the thing that will happen here is going to affect the zero flag. Why does it do that? Well, if we go to the next instruction, we see that we have jnz[7] and this instruction depends on the ZF so if the results is not zero it will jump to this location' loc_11ED' There is two different approaches that came up to my mind when I saw this the first one is just by running the program and change the zero flag to 0 so it will not do the jump. Let us do that. Just before doing that I am running IDA PRO in windows and as i mentioned before it is dangerous to run any program or file on your main operating system, so we need to set up a remote debugger[8] to do that from our Linux virtual machine. Now we can safely play with the program.

[7] https://www.aldeid.com/wiki/X86-assembly/Instructions/jnz
[8] https://eviatargerzi.medium.com/remote-debugging-with-ida-from-windows-to-linux-4a98d7095215



```
.text:0000564CC4F901A6 add     rax, 8
.text:0000564CC4F901AA mov     rax, [rax]
.text:0000564CC4F901AD mov     rsi, rax
.text:0000564CC4F901B0 lea     rdi, format     ; "Checking License: %s\n"
.text:0000564CC4F901B7 mov     eax, 0
.text:0000564CC4F901BC call    _printf
.text:0000564CC4F901C1 mov     rax, [rbp+var_10]
.text:0000564CC4F901C5 add     rax, 8
.text:0000564CC4F901C9 mov     rax, [rax]
.text:0000564CC4F901CC lea     rsi, s2         ; "COOLPASSWORD"
.text:0000564CC4F901D3 mov     rdi, rax        ; s1
.text:0000564CC4F901D6 call    _strcmp
.text:0000564CC4F901DB test    eax, eax
.text:0000564CC4F901DD jnz     short loc_564CC4F901ED

.text:0000564CC4F901DF lea     rdi, s          ; "Access Granted!"
.text:0000564CC4F901E6 call    _puts
.text:0000564CC4F901EB jmp     short loc_564CC4F90207

.text:0000564CC4F901ED
.text:0000564CC4F901ED loc_564CC4F901ED:
.text:0000564CC4F901ED lea     rdi, aWrong
.text:0000564CC4F901F4 call    _puts
.text:0000564CC4F901F9 jmp     short loc_564C

125.00% (-53,474) (669,252) 000011D3 0000564CC4F901D3: main+4A (Synchronized with RIP)
```

```
RAX 00007FFFE0EB73BC ↳ [stack]:00007FFFE0EB73BC
RBX 0000000000000000 ↳
RCX 0000000000000000 ↳
RDX 0000000000000000 ↳
RSI 0000564CC4F9101A ↳ "COOLPASSWORD"
RDI 0000000000000000 ↳
RBP 00007FFFE0EB67C0 ↳ [stack]:00007FFFE0EB67C0
RSP 00007FFFE0EB67B0 ↳ [stack]:00007FFFE0EB67B0

ID   0
VIP  0
VIF  0
AC   0
VM   0
RF   0
NT   0
IOPL 0
OF   0
```

Modules

Path	Bas
/home/kali/Desktop/simple1.bin	000
/usr/lib/x86_64-linux-gnu/libc-2.32.so	000
/usr/lib/x86_64-linux-gnu/ld-2.32.so	000
[vdso]	000

Threads

Decimal	Hex	State	Name
5313	14C1	Ready	simple1.bin

Hex View-1

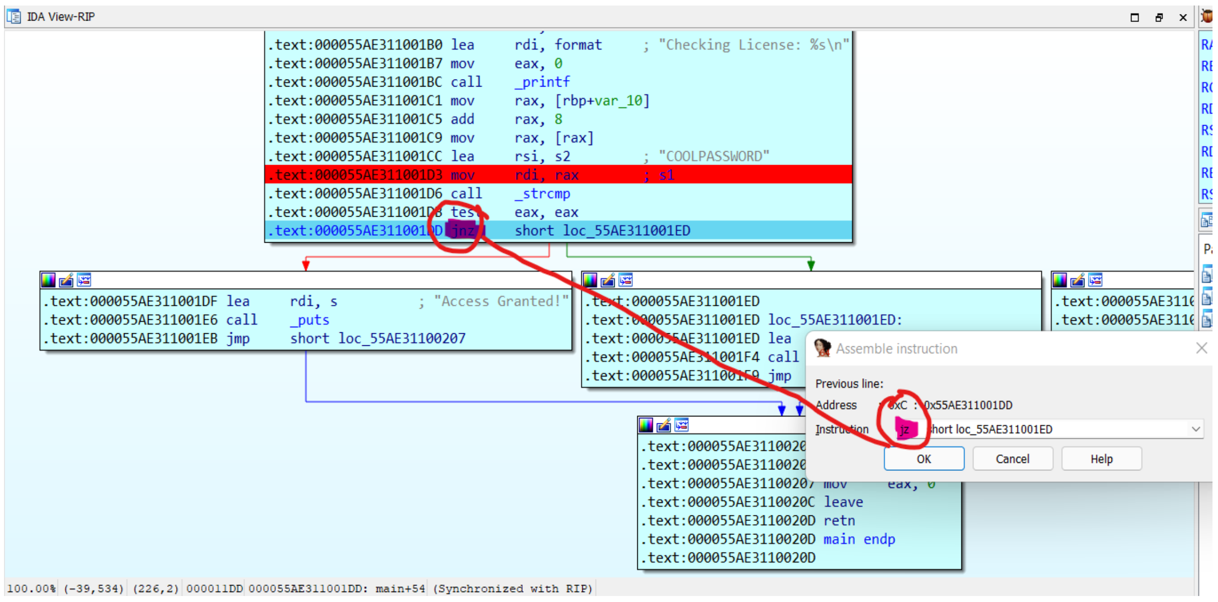
```
0000564CC4F8F000  7F 45 4C 46 02 01 01 00  00 00 00 00 00 00 00 00  .ELF............
0000564CC4F8F010  03 00 3E 00 01 00 00 00  A0 10 00 00 00 00 00 00  ..>.............
0000564CC4F8F020  40 00 00 00 00 00 00 00  80 37 00 00 00 00 00 00  @........7......
0000564CC4F8F030  00 00 00 00 40 00 38 00  0D 00 40 00 1F 00 1E 00  ....@.8...@.....
0000564CC4F8F040  06 00 00 00 04 00 00 00  40 00 00 00 00 00 00 00  ........@.......
0000564CC4F8F050  40 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  @.......@.......
0000564CC4F8F060  D8 02 00 00 00 00 00 00  D8 02 00 00 00 00 00 00  ................
0000564CC4F8F070  08 00 00 00 00 00 00 00  03 00 00 00 04 00 00 00  ................
0000564CC4F8F080  18 03 00 00 00 00 00 00  18 03 00 00 00 00 00 00  ................

00000043 0000564CC4F8F043: LOAD:0000564CC4F8F043
```

Stack view

```
00007FFFE0EB67B0  00007FFFE0EB68B8  [stack]:00007FFFE0EB68B8
00007FFFE0EB67B8  0000000200000000
00007FFFE0EB67C0  0000564CC4F90210  __libc_csu_init
00007FFFE0EB67C8  00007FDBB1A32E4A  libc_2.32.so:__libc_start_main+EA
00007FFFE0EB67D0  00007FFFE0EB68B8  [stack]:00007FFFE0EB68B8
00007FFFE0EB67D8  00000002B1A32C27
00007FFFE0EB67E0  0000564CC4F90189  main
00007FFFE0EB67E8  000000020000000A
00007FFFE0EB67F0  0000000000000000

UNKNOWN 00007FFFE0EB67B0: [stack]:00007FFFE0EB67B0 (Synchronized with RSP)
```

```
AU:  idle   Down    Disk: 13GB
```

I set a breakpoint just before the _strcmp function call so we can see what going to happen. Can you see up on the right we said the string 'COOLPASSWORD' will be loaded into RSI register and that is what happened great! Now what is going to happen if we change the zero-flag value to 1? It will stop the jump and keep going as if we entered the right password but what if we wanted to not go through all of this and waste our time changing the zero-flag value each time? Well, we can patch the program and create a new version (cacked one) that will accept any password except the right one. So let us change the instruction from jnz to jz so it will jump if the zero-flag value is zero and, in our case, would allow us to bypass the password.



IDA View-RIP

```
.text:000055AE311001B0 lea     rdi, format     ; "Checking License: %s\n"
.text:000055AE311001B7 mov     eax, 0
.text:000055AE311001BC call    _printf
.text:000055AE311001C1 mov     rax, [rbp+var_10]
.text:000055AE311001C5 add     rax, 8
.text:000055AE311001C9 mov     rax, [rax]
.text:000055AE311001CC lea     rsi, s2         ; "COOLPASSWORD"
.text:000055AE311001D3 mov     rdi, rax        ; s1
.text:000055AE311001D6 call    _strcmp
.text:000055AE311001DB test    eax, eax
.text:000055AE311001DD jnz     short loc_55AE311001ED

.text:000055AE311001DF lea     rdi, s          ; "Access Granted!"
.text:000055AE311001E6 call    _puts
.text:000055AE311001EB jmp     short loc_55AE31100207

.text:000055AE311001ED
.text:000055AE311001ED loc_55AE311001ED:
.text:000055AE311001ED lea
.text:000055AE311001F4 call
.text:000055AE311001F9 jmp

.text:000055AE3110020
.text:000055AE3110020
.text:000055AE31100207 mov     eax, 0
.text:000055AE3110020C leave
.text:000055AE3110020D retn
.text:000055AE3110020D main endp
.text:000055AE3110020D

100.00% (-39,534) (226,2) 000011DD 000055AE311001DD: main+54 (Synchronized with RIP)
```

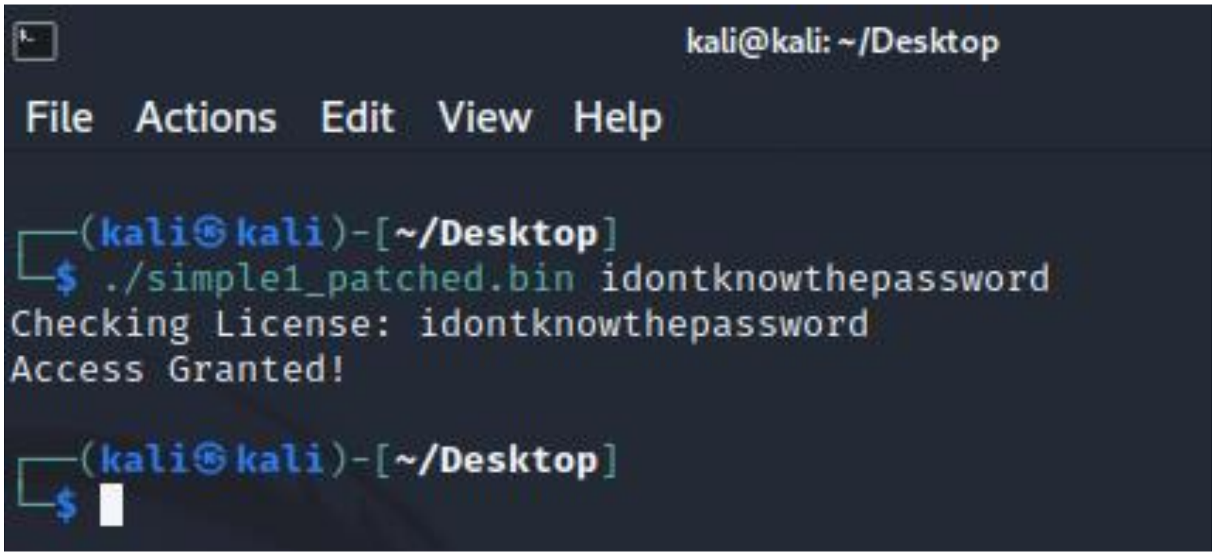
Assemble instruction

Previous line:

Address 0xC : 0x55AE311001DD

Instruction jz short loc_55AE311001ED

After pressing ok, we will have our patched program and we can run and check if it really works.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./simple1_patched.bin idontknowthepassword
Checking License: idontknowthepassword
Access Granted!

┌──(kali㉿kali)-[~/Desktop]
└─$ 
```

And it worked great.

Summary

We started by explaining what assembly and reverse engineering is and how it works and then we talked a bit about static and Dynamic analysis and after we used these methods to crack a simple program that was written in c.