

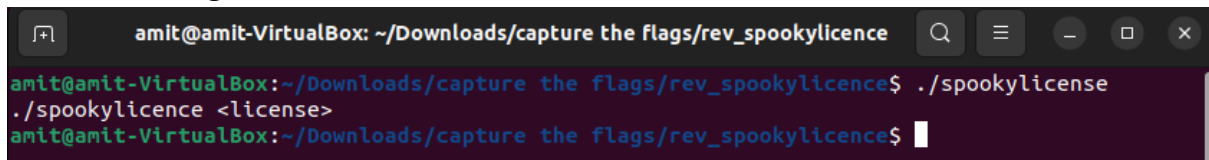
Spooky License

Capture the Flag Challenge.

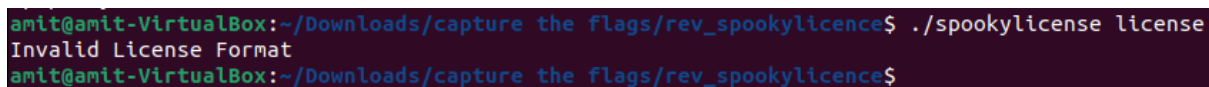
Link: Challenge can be found [here](#).

Overview: The task of this CTF is to correctly identify the required to enter in order to receive valid license, in an ELF file.

When running the file:

A terminal window titled 'amit@amit-VirtualBox: ~/Downloads/capture the flags/rev_spookylicense'. The prompt is 'amit@amit-VirtualBox:~/Downloads/capture the flags/rev_spookylicense\$'. The user enters './spookylicense', and the prompt changes to './spookylicense <license>'. The user then enters 'license', and the prompt returns to 'amit@amit-VirtualBox:~/Downloads/capture the flags/rev_spookylicense\$'.

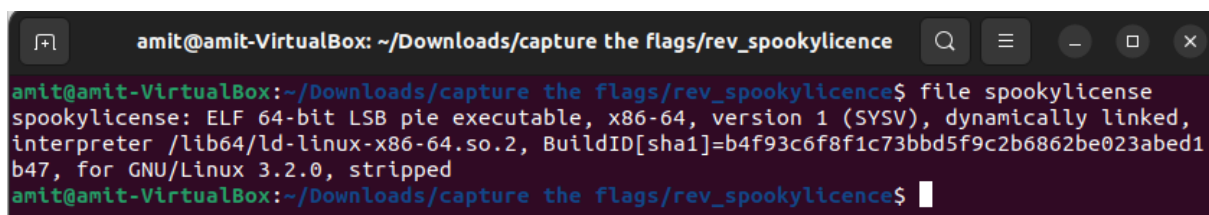
I see that a user input is required:

A terminal window titled 'amit@amit-VirtualBox: ~/Downloads/capture the flags/rev_spookylicense'. The prompt is 'amit@amit-VirtualBox:~/Downloads/capture the flags/rev_spookylicense\$'. The user enters './spookylicense license', and the output is 'Invalid License Format'. The prompt then returns to 'amit@amit-VirtualBox:~/Downloads/capture the flags/rev_spookylicense\$'.

When input is entered – a prompt of wrong format is received.

So in the inspection it will be up for us to discover the correct format, and the correct license, or the correct method to extract the flag.

Method: the first action that was taken in order to inspect the file, is to examine the details of which with ‘file’ command:

A terminal window titled 'amit@amit-VirtualBox: ~/Downloads/capture the flags/rev_spookylicense'. The prompt is 'amit@amit-VirtualBox:~/Downloads/capture the flags/rev_spookylicense\$'. The user enters 'file spookylicense', and the output is 'spookylicense: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=b4f93c6f8f1c73bbd5f9c2b6862be023abed1b47, for GNU/Linux 3.2.0, stripped'. The prompt then returns to 'amit@amit-VirtualBox:~/Downloads/capture the flags/rev_spookylicense\$'.

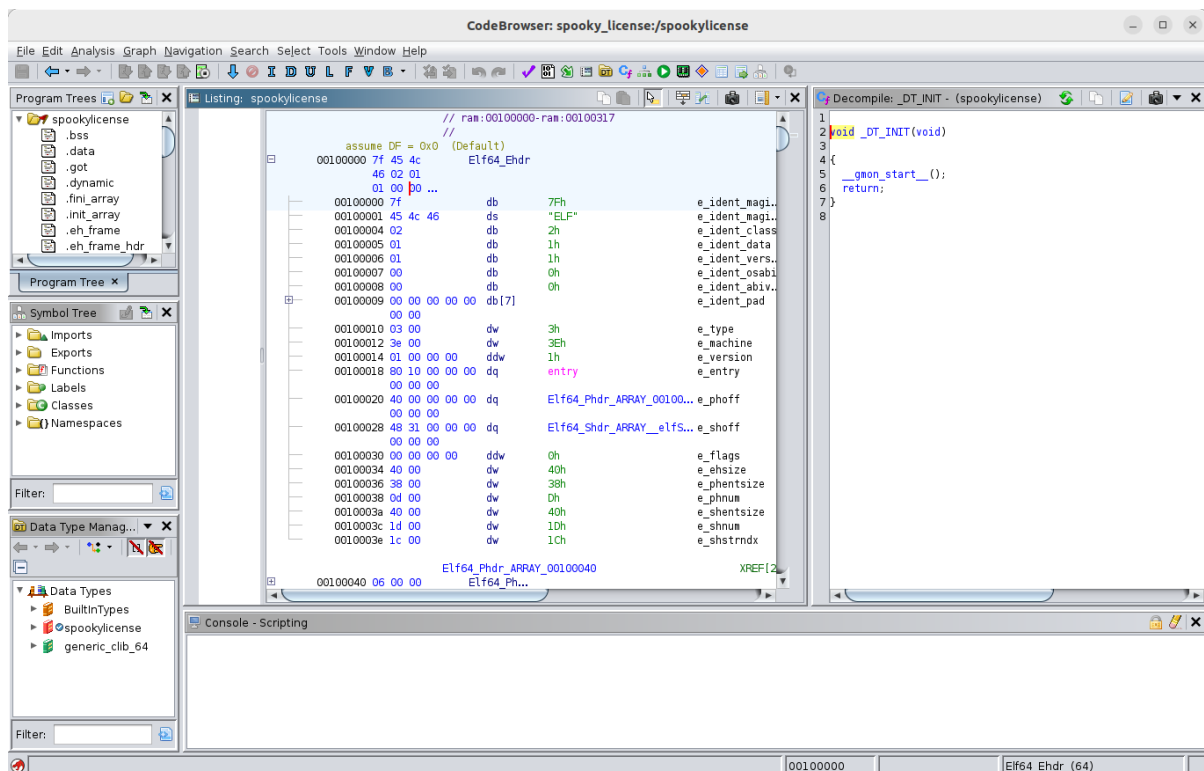
It can be observed that the file is stripped, meaning it is not compiled for debugging.

And indeed, running ‘objdump -d’ on the file reveals that it lacks the internal functions necessary to properly analyze the assembly. Nor it contains the debugging information for proper debugging with gdb.

So in order to inspect the program – I will have to use others tools.

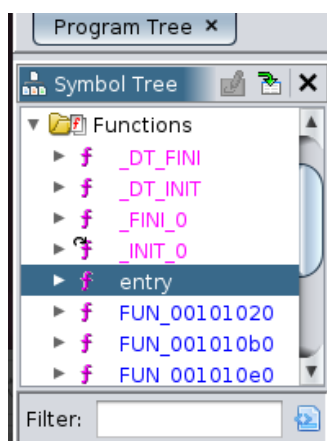
The tool I will use in order to proceed the inspection is disassemble and decompile tool called 'Ghidra'.

I started a new project on Ghidra and inserted the program in it, and I let Ghidra do the hard work of analyzing the program:

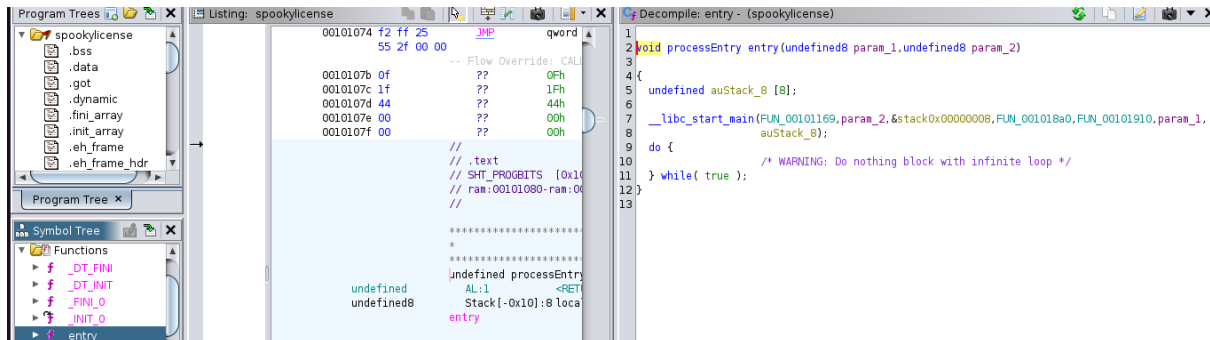


In the attached picture we can see the analyzed program on Ghidra, where on the middle window is the original assembly commands, in the right window is the decompiled code in C pseudo code. And in the left window (vertically middle one) there is Symbol Tree where we can take a look at Ghidra interpreted functions, analyzed from the assembly.

I took a look in it, and I saw among the different functions 'init', 'fini', nameless functioned assigned by their address, and entry:



The entry function is where the code itself starts executing, so I took a look in it:



The Ghidra's decompiled entry function is `libc_start_main` function, which performs all the necessary initializations for the code execution and calls main with its arguments.

We can observe that it takes 2 parameters -

It takes several parameters:

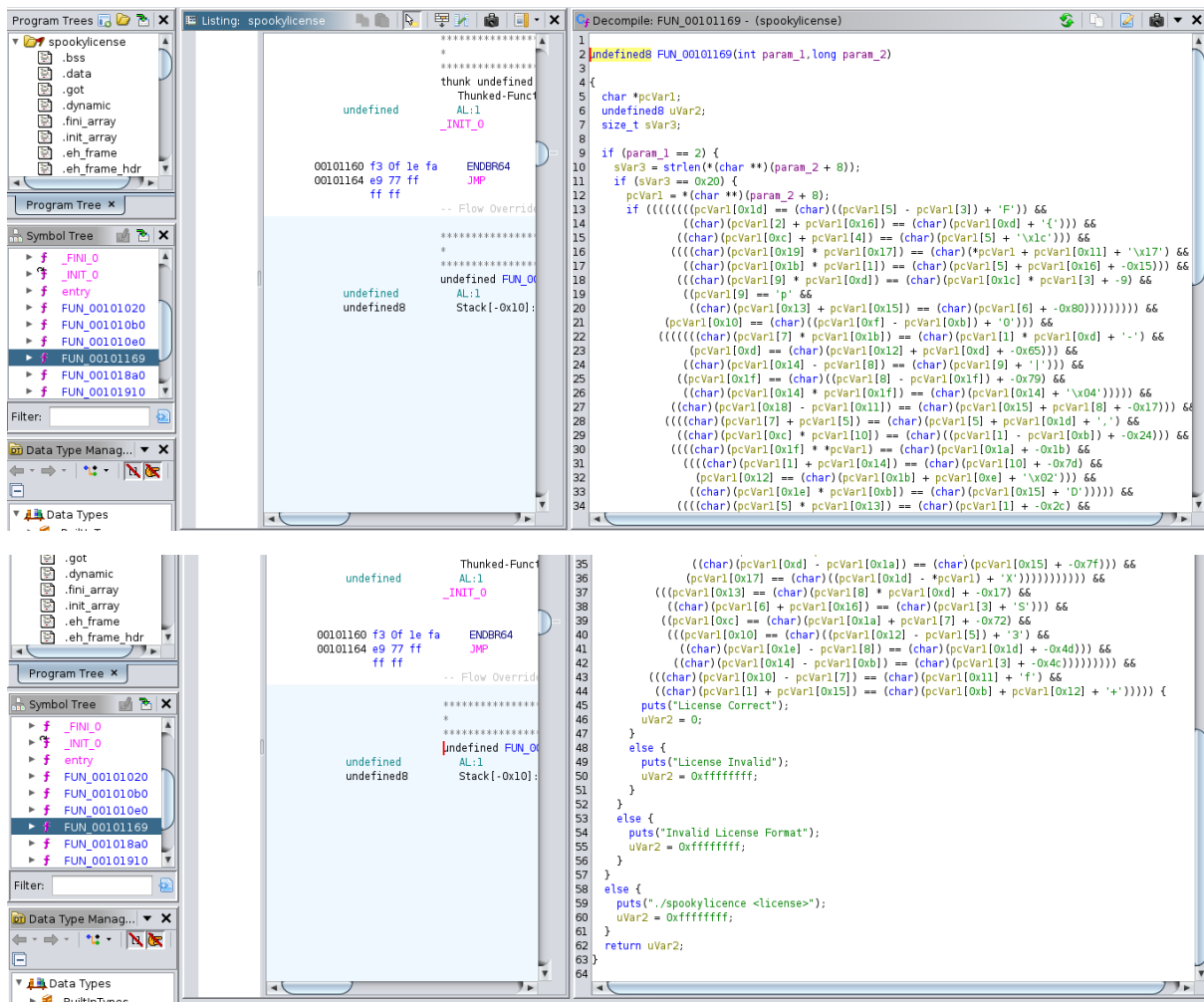
`FUN_001011169` – that is the pointer to the main function .

`param1` – `argc`

`param2` – `argv`

the other parameters are initialization parameters that are not directly required for the main function, or our needs.

Let's take a look at '`FUN_001011169`', or as we call it from this moment – `main`:



Lets analyze the decompiled main:

The first if block – it compares the param1, the argc value to 2.

Basically it checks if the user entered a single parameter (the other one is the default program path).

That's why upon the initial run attempted without any parameters – we received the '<license>' prompt.

Then we have this line:

`sVar3 = strlen(*(char**) (param2+8));`

let's break it down – param2, which is argv is char** type – pointer to pointer.

Or put it in other way – array of pointers.

So, when we take param2 and add 8 to it – we advance it to the next pointer value, as pointer in 64bit program is 8 bytes long.

Basically, it is equal to param2[1].

The addition of 8 is required so the selected parameter will be the user input string, and not the default program path (param2[0]).

As param2+8 is pointer to pointer, the leftmost asterisk is required to 'get in' to the pointer value, such that the 'strlen' function parameter will have a pointer to the user's string input itself.

The strlen is a function that takes in a string (array of chars) and returns its length. We store the result in sVar3 variable and compare it to 0x20, or 32.

In conclusion – the next condition checks the user's input is 32 characters long.

Let's check that:

```
amit@amit-VirtualBox:~/Downloads/capture_the_flag/rev_spookylicense$ ./spookylicense 12345678123456781234567812345678
License Invalid
```

We received different prompt.

This time the format itself is correct, however we still do not have the correct valid license.

Now that we know that the required license is 32 characters long, let's proceed:

In pcVar1 variable we store the pointer to the string.

In the next part – we have a long list of conditions that are separated by '&&', meaning if ALL of them are true – we receive the correct license prompt.

It can be inferred that the correct flag is the correct string that needs to be entered to receive the correct license prompt. So assembly modification or bypassing won't work here, we need to figure out the correct string, and for that we need to analyze the conditions.

Let's examine the first condition:

```
pcVar1[0x1d] == (char)((pcVar1[5] - pcVar1[3] + 'F')
```

it means that the ascii value of character in the index 0x1d, or 29 of the string (counting from 0 of course) should be equal to the ascii value of the character in pcVar1[5], minus the ascii value of the character in pcVar1[3], and we add to that the ascii value of 'F', which is 70 (in decimal).

To illustrate it more clearly. For example, let's assume pcVar1[5] = 'B' and pcVar1[3] = 'A'.

The ascii of 'B' is 66, and the ascii of 'A' is 65. So 66-65+70 = 71.

So, in this example pcVar1[0x1d] value is the ascii of 71 – ‘G’.

Of course, this is just an example, and the values of pcVar1[5] and pcVar1[3] are not known at this moment.

Now that we know the essence of a single condition – lets observe the whole conditions – it can be inferred that we have a system of equations.

There are 32 variables, where each variable is a character of the input string. so, there are 32 equations in the conditions array, that all are needed to be solved in order to discovered the correct string.

The system of equations maybe could be solved manually. But that would take too much time and effort, and there is much margin of error.

An automation and optimization are required here. After some research I opted to use a tool called ‘angr’.

[angr](#) is python library that takes some program and runs symbolic execution in it, meaning it takes some values (symbol) and test the array of conditions with it. And from the symbolic execution (which is not full scale program execution) it evaluates code branches, to check what works and what doesn’t works.

So I created python virtual environment and on that installed angr:

```
amit@amit-VirtualBox:~/Downloads/capture the flags/rev_spookylicence$ python3 -m venv angr
amit@amit-VirtualBox:~/Downloads/capture the flags/rev_spookylicence$ source angr/bin/activate
(angr) amit@amit-VirtualBox:~/Downloads/capture the flags/rev_spookylicence$ python3 -m pip install angr
```

And created python script that uses angr and aid library called ‘claripy’:

```

1 import angr
2 import claripy
3
4 def main():
5     success_address = 0x401876
6     failed_address = 0x401889
7     #the 0x400000+0x1876 address is required by pie executable
8     flag_length = 32
9     project = angr.Project("./spookylicense", load_options={'auto_load_libs':False})
10    argv = [project.filename]
11    sym_arg=claripy.BVS('input', 8*flag_length) # symbolic guessing of 32 bytes
12    argv.append(sym_arg)
13    entry_state = project.factory.entry_state(args=argv)
14    sim = project.factory.simulation_manager(entry_state)
15    sim.explore(find=success_address, avoid=failed_address)
16    if sim.found:
17        print("success")
18        solution = sim.found[0].solver.eval(argv[1], cast_to=bytes)
19        print(solution)
20    else:
21        print("failed")
22
23
24 if __name__ == "__main__":
25     main()

```

What it requires is success and fail address – the address of the section where the program succeded (“License Correct”) and the address of the section where the program fails (“License Invalid”)

The addresses can be found on Ghidra:

0010186d 01 c8	ADD	EAX, ECX	42	((char)(pcVar1[0x14] - pcVar1[0xb]) == (char)(pcVar1[0x10] - pcVar1[7])) == (char)(pcVar1[0x11] + pcVar1[0x15]) == (char)(pcVar1[0x14] - pcVar1[0xb])
0010186f 83 c0 2b	ADD	EAX, 0x2b	43	
00101872 38 c2	CMP	DL, AL	44	
00101874 75 13	JNZ	LAB_00101889	45	puts("License Correct");
00101876 48 8d 3d	LEA	RDI, [s_License_Correct_00102035]	46	uVar2 = 0;
b8 07 00 00			47	
0010187d e8 de f7	CALL	<EXTERNAL>::puts	48	
00101889 48 8d 3d	LEA	RDI, [s_License_Invalid_00102045]	48	else {
b5 07 00 00			49	puts("License Invalid");
00101890 e8 cb f7	CALL	<EXTERNAL>::puts	50	uVar2 = 0xffffffff;
ff ff			51	
			52	
			53	else {

We take the last 4 digits (in hexadecimal) and add 0x400000 to it.

That would be the required success and fail addresses.

The next thing that the python symbolic execution script requires is claripy BVS - BitVector Symbolic that would be the symbolic input.

The BitVector works with bits, and the input is 32 characters string which is 32 bytes. So, the BitVector length should be 32*8 bits long.

We create argv array which contains the program path and the BitVector as our symbolic input, set the necessary parameters for the simulation, including the success and fail address. And start the exploration, if a solution is found, satisfying all 32 equations – it returns success with the correct string, the correct string would be the flag.

Now we run it:

```
(angr) amit@amit-VirtualBox:~/Downloads/capture the flags/rev_spookylicense$ python3 symbolic_exec.py
WARNING | 2023-09-30 13:37:45,887 | angr.storage.memory_mixins.default_filler_mixin | The program is accessing memory with an unspecified value. This could indicate unwanted behavior.
WARNING | 2023-09-30 13:37:45,920 | angr.storage.memory_mixins.default_filler_mixin | angr will cope with this by generating an unconstrained symbolic variable and continuing. You can resolve this by:
WARNING | 2023-09-30 13:37:45,920 | angr.storage.memory_mixins.default_filler_mixin | 1) setting a value to the initial state
WARNING | 2023-09-30 13:37:45,923 | angr.storage.memory_mixins.default_filler_mixin | 2) adding the state option ZERO_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, to make unknown regions hold null
WARNING | 2023-09-30 13:37:45,923 | angr.storage.memory_mixins.default_filler_mixin | 3) adding the state option SYMBOL_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, to suppress these messages.
WARNING | 2023-09-30 13:37:45,924 | angr.storage.memory_mixins.default_filler_mixin | Filling memory at 0x7fffffff0000 with 49 unconstrained bytes referenced from 0x500010 (strlen+0x0 in extern-address space (0x10))
success
b'HTB{The_sp0000000key_liC3nC3K3Y}'
(angr) amit@amit-VirtualBox:~/Downloads/capture the flags/rev_spookylicense$
```

Success! The angr symbolic execution script indeed found the correct string.

Let's confirm it:

```
amit@amit-VirtualBox:~/Downloads/capture the flags/rev_spookylicense$ ./spookylicense HTB{The_sp0000000key_liC3nC3K3Y}
License Correct
amit@amit-VirtualBox:~/Downloads/capture the flags/rev_spookylicense$
```

The program accepted the string, and I got the flag.

Conclusions:

The Capture the flag was indeed challenge, I had to deal with a new situation – when the program is stripped and traditional debugging method could not be used properly in order to investigate the program.

So other methods and tool had be used in order to investigate the program.

The first one is 'Ghidra' – a tool that I had no prior experience and it took me some time to learn to use it properly and understand its features.

It's decompilation and analyzing capabilities were proven effective – seeing the decompiled code helped me a lot and saved me a lot of time, and it showed me that even without gdb and dynamic analysis – ELF programs can still be effectively reversed engineered. If I would use it on previous ELF challenges, maybe I would have solved them faster.

The second situation that I had no prior experience of is when I had to deal with the system of equations, surely – I would have solved those manually, but that would have taken a lot of effort.

'angr' does seems to do good job in handling these situations.

I had no prior knowledge of 'angr' or 'claripy' before its challenge, and I had to research a lot before I decided to use them as my method to deal with the

equations. And it took me time to understand them properly. How to get the success and fail and success addresses, how to understand the mechanism of symbolic execution and get the script to work properly.

I indeed learned a lot from these tools, and certain I can utilize them in the future for future challenges.

The python angr Script used for the Symbolic Execution can be downloaded [here](#).