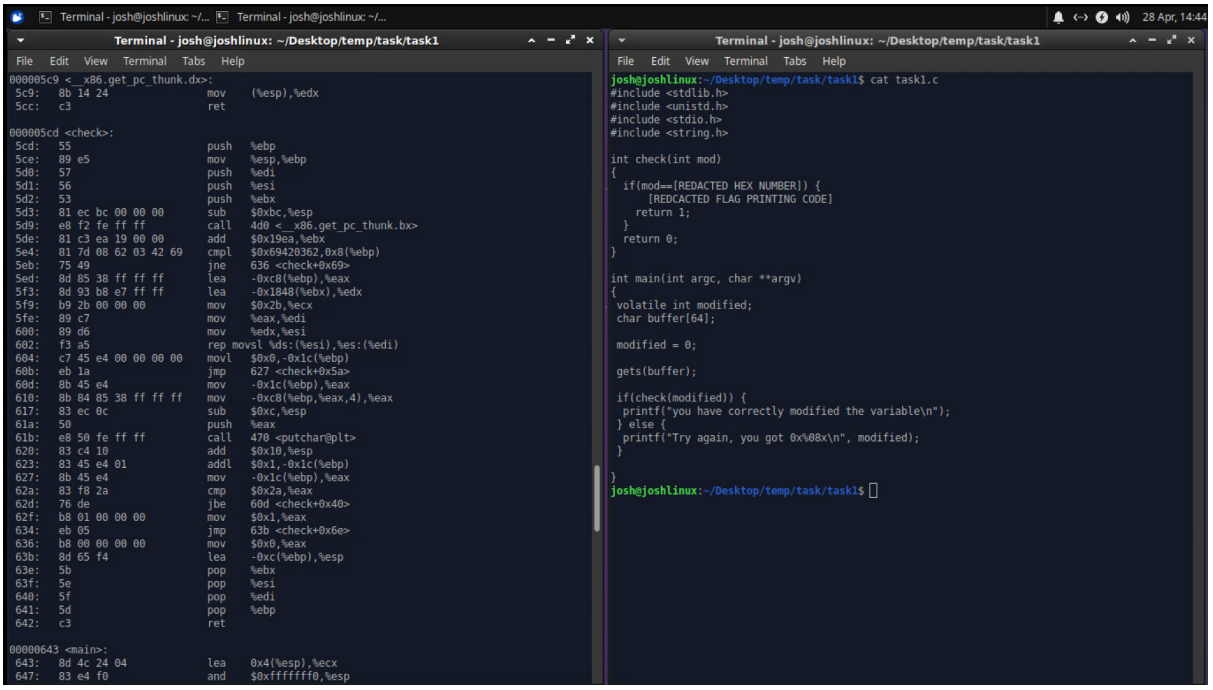
COMP6236 Lab 5 Demonstrator Notes

Task 1 - Check

1. Find the hex which needs modifying from the decompilation listing (objdump -d)



If you look at the disassembly listing of the check function on the left, you can see a couple of items stand out. The main part of this function we are looking to adjust is the cmpl instruction, and following jne instruction, which represent the if statement occurring in theC function of check on the right.

2. Change the number in the cmpl instruction at 5e4 to be something else (zero if you check the redacted source should allow the program to give a flag)

If we look at hex address 5e4 in the disassembly listing, we can see that the hex 81 7d 08 62 03 42 69 disassembles to *cmpl* (compare literal), comparing the value at 0x8(%ebp) with the value 0x69420362.

The following *jne* instruction acts on the output of this comparison, jumping to address 0x636 if the values compared in the numbers from the previous command were not equal, and if not, continues with program flow as normal.

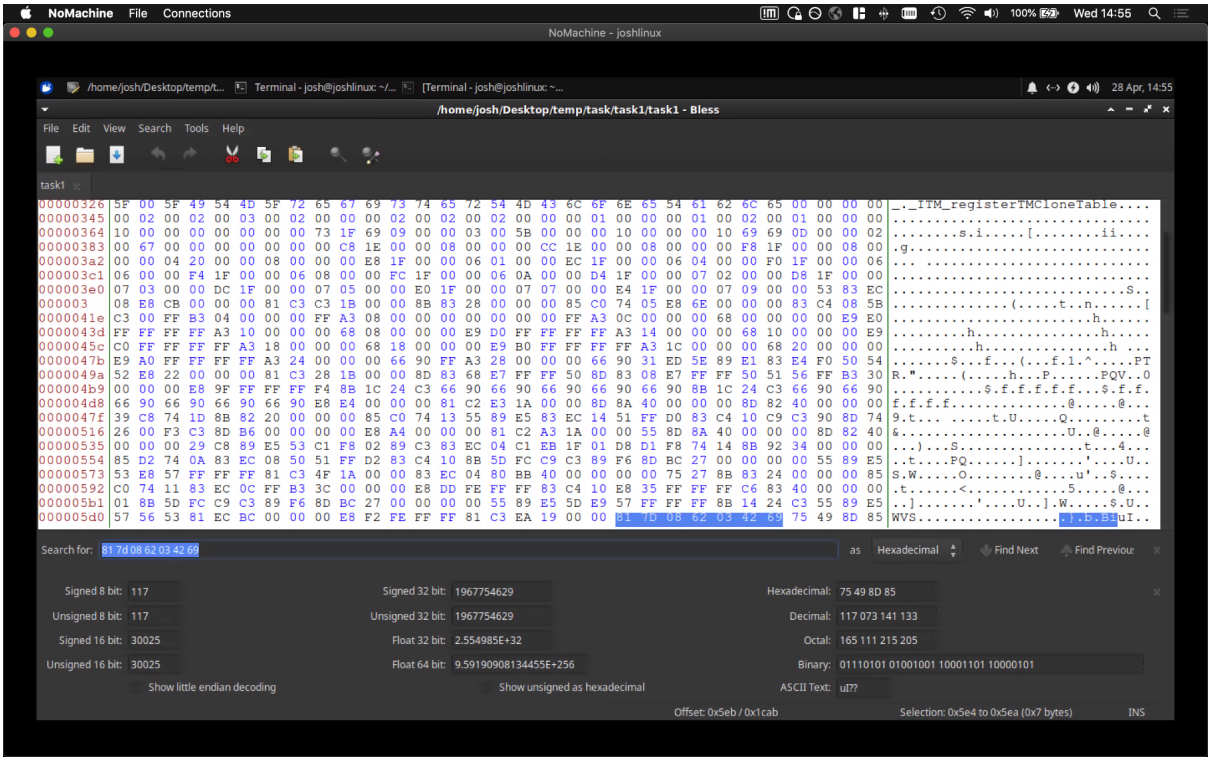
This gives two potential locations which we can modify in the binary to make this check pass and output the flag. The first of these is changing the literal which is being compared against to 0x00000000, as we know that modified=0 from reading the redacted C code. This would make the check pass every time.

The second way we could modify the binary is to change *jne* (jump if not equal) to *je* (jump if equal), inverting the if statement that is taking place. We'll be doing this for Challenge 0 below.

3. Edit the binary with Bless hex editor

In order to change the number that cmpl is comparing against, open your binary in Bless, which is a graphical hex editor. You can also use other hex editors if you wish.

Looking at our disassembly listing, we are looking for the following hex, which describes the cmpl operator and the number it is comparing against.
81 7d 08 62 03 42 69



We can now change this by deleting 62 03 42 69 and replacing it with 00 00 00 00

Save the binary as a new file using File->Save as, and then use chmod+x in the terminal to make it executable. You can see that it now works, bypassing the check.

```
josh@joshlinux:~/Desktop/task$ ./task1-mod2

flag{c4ae6cef-68f9-4d8a-bc38-28292023d78e}
you have correctly modified the variable
josh@joshlinux:~/Desktop/task$
```

Challenge 0 - Simple Binaries

The other way to bypass this simple check is to change the jne instruction occurring on the line below cmpl to a je instruction, inverting the if statement.

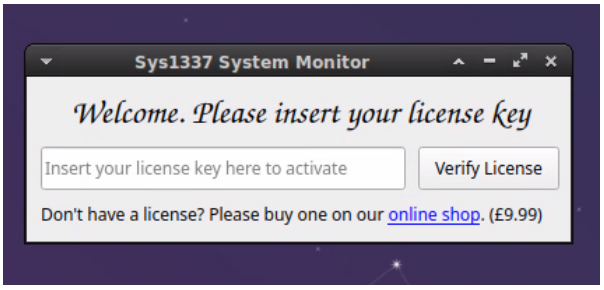
This can be achieved simply by changing the instruction at 5eb from:
75 (jne) to **74** (je).

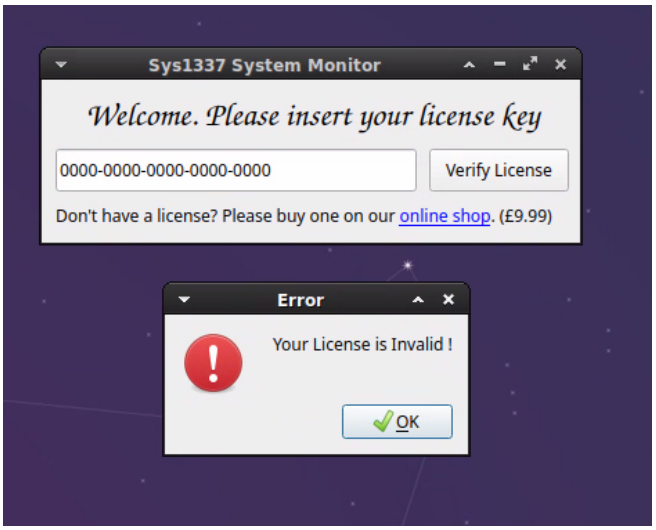
Doing this in your hex editor and then saving the binary again, you will notice that the check is also bypassed.

Task2 - Ghidra

Ghidra simplifies this reverse engineering process, providing many extra features compared to objdump. For Task 2, we will use Ghidra to reverse engineer a more complex binary.

Opening task 2, we can see that it asks for a License key, which needs to be correct for the binary to function.





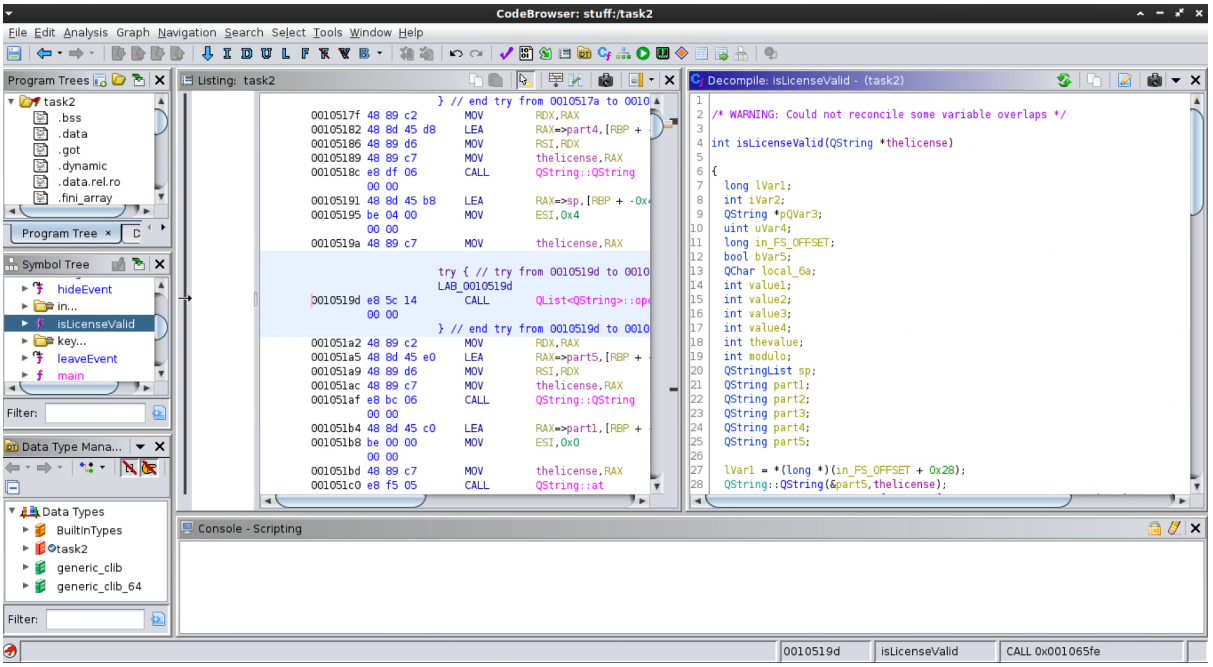
The first step is to open this application with Ghidra and decompile it. You can do this by selecting File->New project and adding the binary. When you open the binary in Ghidra, click “Analyze” with the default options.

The standard Ghidra window is split into two parts. On the left is your decompilation view showing assembler and hex, and on the right is a C-style representation of what the code is doing.

In the left hand bar under “Symbol Tree” expand “functions” and have a look at the function names. Thankfully, this binary has been compiled with debug information present so function names and other useful information is intact.

A look through function names in the binary shows one which is called “IsLicenseValid”. From the name, this might be useful.

Selecting this function you can see the assembly code which makes it up on the left, and a C-style view of the function on the right. This extra disassembly step makes it much easier to look at when compared with Objdump.



Now, let’s see what this function is doing. Looking at the decompiled code on the right, you can see that a string called “thelicense” is taken in and has the following steps applied to it.

1. Is the license numeric?
2. Is the length equal to 0x18
3. Split the license by character 0x2d (-) into 5 variables.
4. Get the first character of the first part as a number (e.g. “1” = 1)
5. Get the second character of the second part as a number (e.g. “1” = 1)
6. Get the third character of the third part as a number (e.g. “1” = 1)
7. Get the fourth character of the fourth part as a number (e.g. “1” = 1)
8. Add all these together and add 1 to the total.
9. Find modulo 15 of this number (the remainder when divided by 15).
10. If it doesn’t equal zero, license is invalid.
11. Else, license is valid.

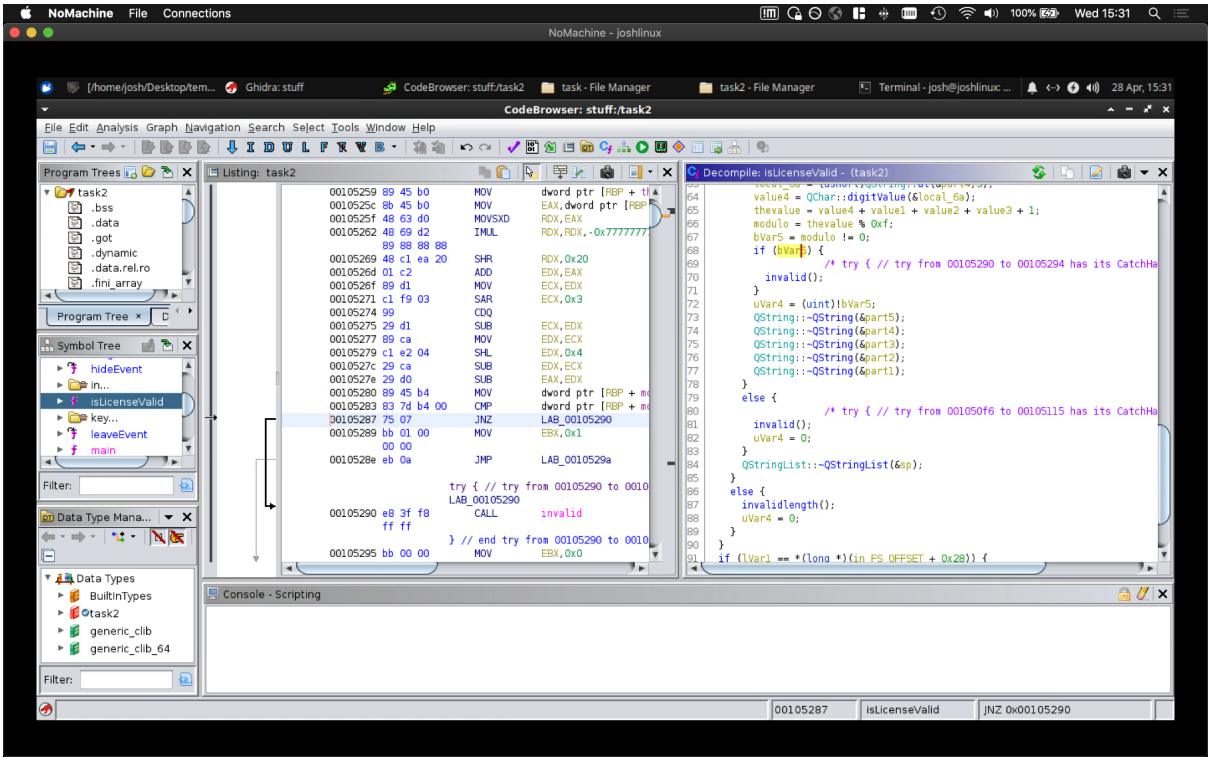
We can patch this binary simply by inverting the check.

Clicking on the final comparison line in the right hand C view shows the line in assembler which needs to be changed. You may see this one decompiles to JNZ not JNE like we had before; that is just due to Ghidra being a more advanced decompiler and inferring the context of the instruction.

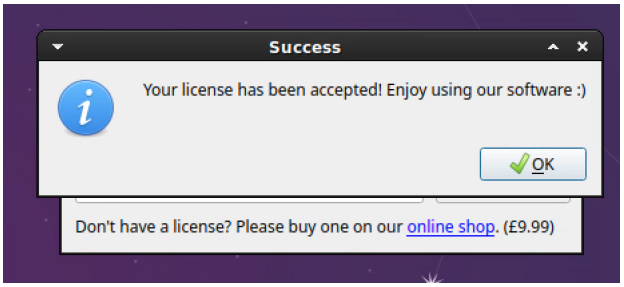
We can again do like we did last time and change this instruction to a 0x74. Ghidra has a built-in hex editor, but binary export is non-functional for ELF binaries. Hence, we will need to again exit it in an external editor like Bless.

Simply search for the hex 83 7d b4 00 75 07 (this is longer than the actual string we want to change as there may be other 74 07 instructions in the binary)

Change the 75 in this line to 74 as done before for task 1



Now, when you run the binary, it should accept any license key that conforms to the other requirements on length and format. So now, **0000-0000-0000-0000-0000** should work.



Challenge 2 - Algorithm

Using the steps we described above, we can work backwards to create a valid license key.

We know our license is split into 5 parts, each of which has 4 characters, like so.
0000-0000-0000-0000-0000

We know that the first character of the first part, plus the second character of the second part, plus the third character of the third part, plus the fourth character of the fourth part, plus 1, needs to be divisible by 15 for the modulo check to pass and the license to be valid.

From here, it should be possible to come up with any valid combination of these, e.g.

9000-0500-0000-0000-0000
5000-0500-0040-0000-0000
8000-0400-0020-0000-0000