

# Reverse Engineering Lab

---

## Brief Introduction

Reverse engineering is the process of working backwards starting from a finished product to understand how it works, how it was designed, and how it functions. This lab specifically focuses on Software Reverse Engineering, but we may end up referring to this as just Reverse Engineering or in an abbreviated form, i.e., RE.

Normally software is compiled. This means that we take the source code, feed it into a compiler and then through the compilation process we get an executable binary.

Software RE does exactly the opposite. It takes an executable binary of a piece of compiled code and decompiles the code so that it becomes readable. Typically, this step can also be done with tools such as **objdump**, but it lacks a crucial part that these other tools like **IDA Pro** or **Ghidra** have, which is turning assembly code back into C or C-like code.

This lab focuses on this last part, and to be able to achieve it, we will be using **Ghidra**, which is an Open-Source tool made by the **NSA - National Security Agency**.

## Objectives

- Understand how to install Ghidra
- Understand some of the basics
  - Creating a project
  - Importing and loading a target binary
- Learning
  - How to read the processed code
  - Variable renaming
  - How to patch instructions

## Pre-Setup

Unfortunately, this Lab cannot be easily run under a Docker container. We're going to explain how to install Ghidra and the necessary dependencies under a Debian-based Linux machine, but in case something goes wrong, the official guide is over at

**<https://github.com/nationalsecurityagency/ghidra>**.

Ghidra is compatible with Linux, Windows, or macOS. Note that Java is also required, so you must first install JDK 21 and do not forget to set the `JAVA_HOME` environment flag.

## Gathering files

For x86-64 machines:

- Just follow the guide and ignore any section that contains **ARM instructions**.

For ARM machines:

- You may follow the guide as normal, but whenever there are specific instructions that you need to follow, we will let you know.

First, run these commands. These will update the repositories and do any necessary upgrades.

```
$ sudo apt update
$ sudo apt upgrade
```

We're now ready to start downloading and installing the necessary packages.

## Installing JDK 21

This next command just installs JDK 21.

```
sudo apt install openjdk-21-jdk
```

## Obtaining a copy of Ghidra

As of writing this guide, the latest version is **Ghidra 12.0.4**, which was released on the 4th of March 2026. As with normal software, it will eventually get updated. In theory, later releases should not drastically change the UI and can probably be used, but this may not be the case.

To obtain Ghidra you can either manually download using this link

**<https://github.com/NationalSecurityAgency/ghidra/releases>** or alternatively run the following commands below (They will download and install wget if it is not already installed (which allows us to download ghidra via the terminal), go to the user's home folder, create a new folder named ghidra and enter it. Finally it downloads Ghidra, saves it as ghidra.zip, unzips the file and then deletes the zip):

```
$ sudo apt install wget
$ cd ~/ && mkdir -p ghidra && cd ghidra
$ wget -O ghidra.zip
https://github.com/NationalSecurityAgency/ghidra/releases/download/Ghidra_12.0.4_build/ghidra_12.0.4_PUBLIC_20260303.zip && unzip ghidra.zip &&
rm ghidra.zip && cd ghidra_12.0.4_PUBLIC
```

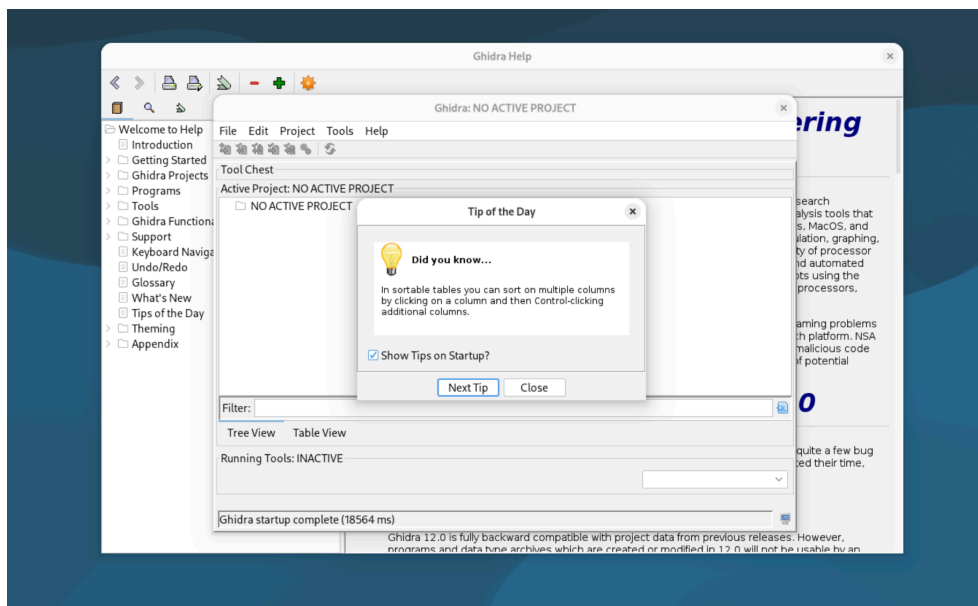
## Launching Ghidra

If you followed every step you should now be able to execute ghidra. Assuming you're still in the **~/ghidra/ghidra\_12.0.4\_PUBLIC** folder you may now execute the following command:

```
$ ./ghidraRun
```

Note that you may need to agree to an Apache 2.0 license to use the software.

After selecting "I Agree" you should be presented with the following screen:



After clicking "Close" you are now ready to proceed to Task 1.

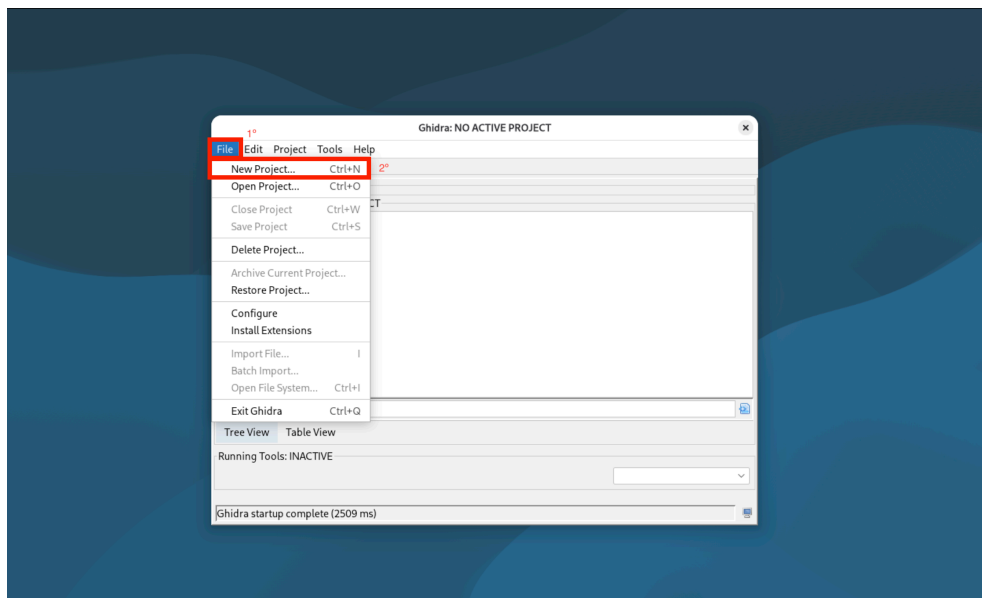
## Task 1 - Loading and understanding a simple program

By the end of this task you should be able to create, import and see the C-like code that ghidra produces. You also need to answer the questions at the end of the task and you also need to include screenshots whenever we ask for them. This is important because otherwise the teacher may not be able to understand your point of view.

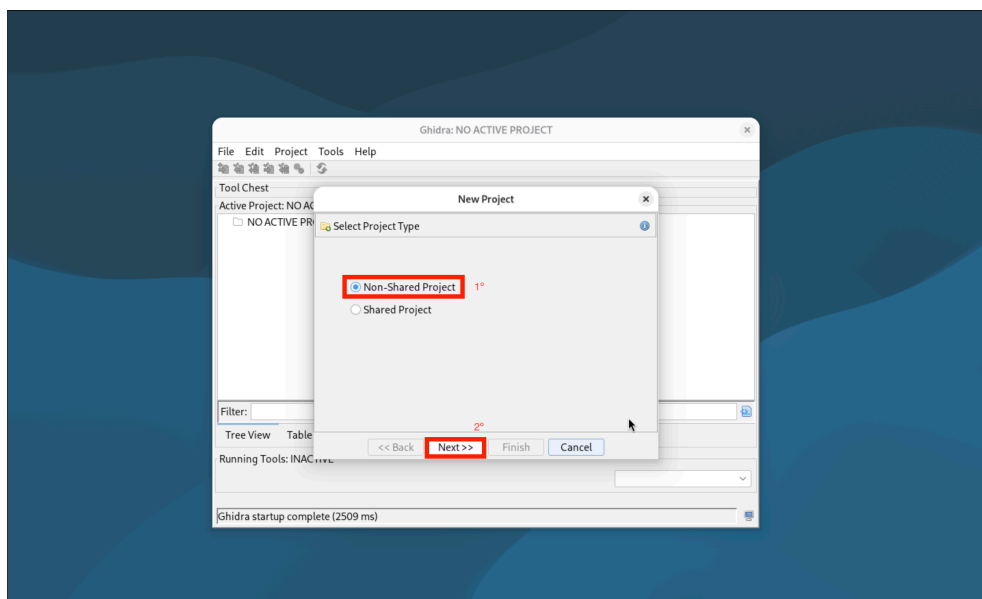
### Task 1.1 - Creating a project and importing the Task1 binary

Before loading any binary you'll first need to create a project.

To do this firstly select the "File" button and afterwards select the "New Project" button.

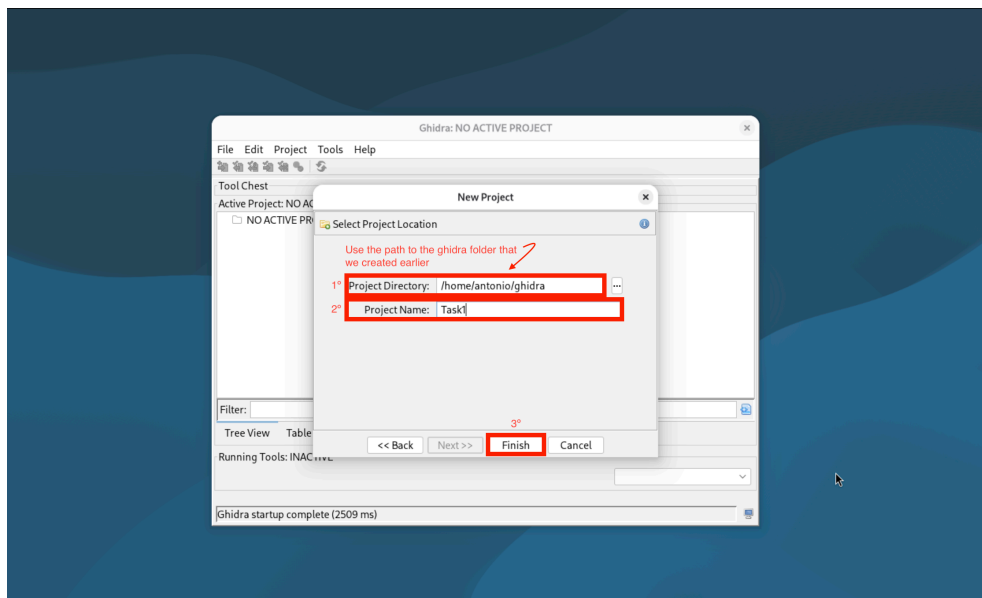


Then select (if not already selected) "Non-Shared Project" and then "Next".

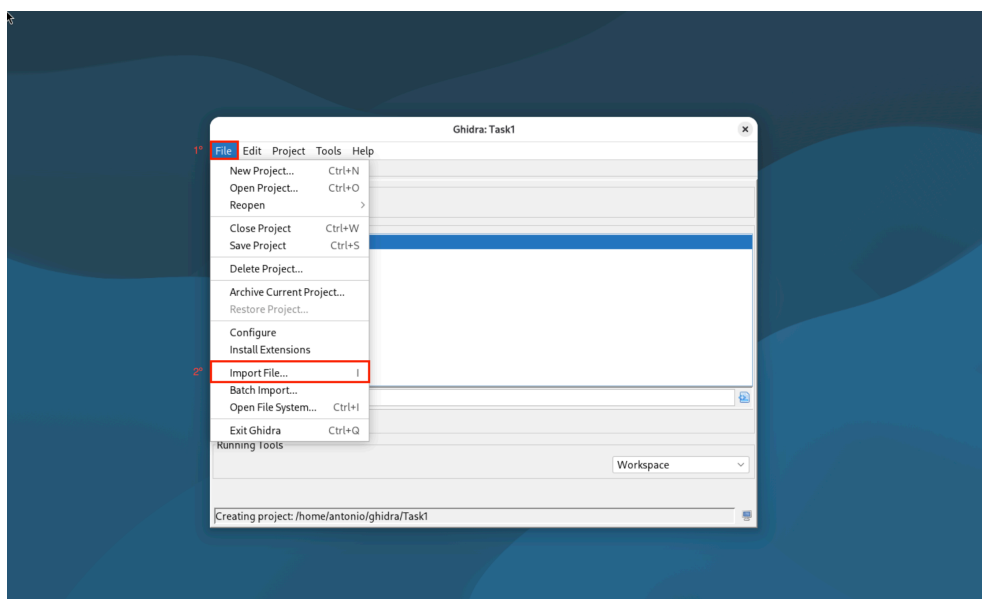


Now be sure to change the "Project Directory" to the ghidra folder we just created. This prevents clutter and keeps everything in one place. Be sure to also name the project, for this one we will be choosing "Task1". Then when everything looks good just press the "Finish" button.





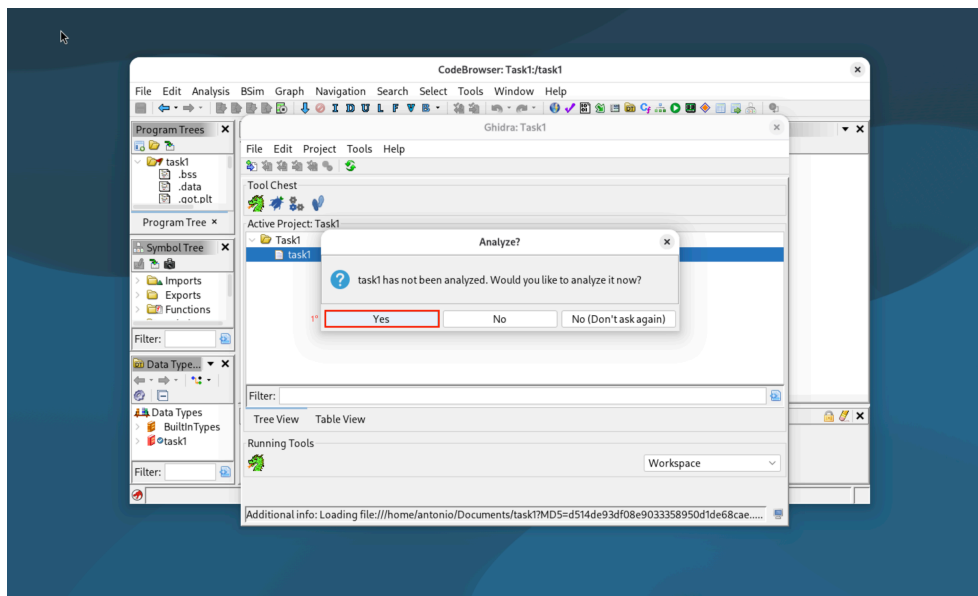
Finally you can now use the "File" menu again to "Import" a file.



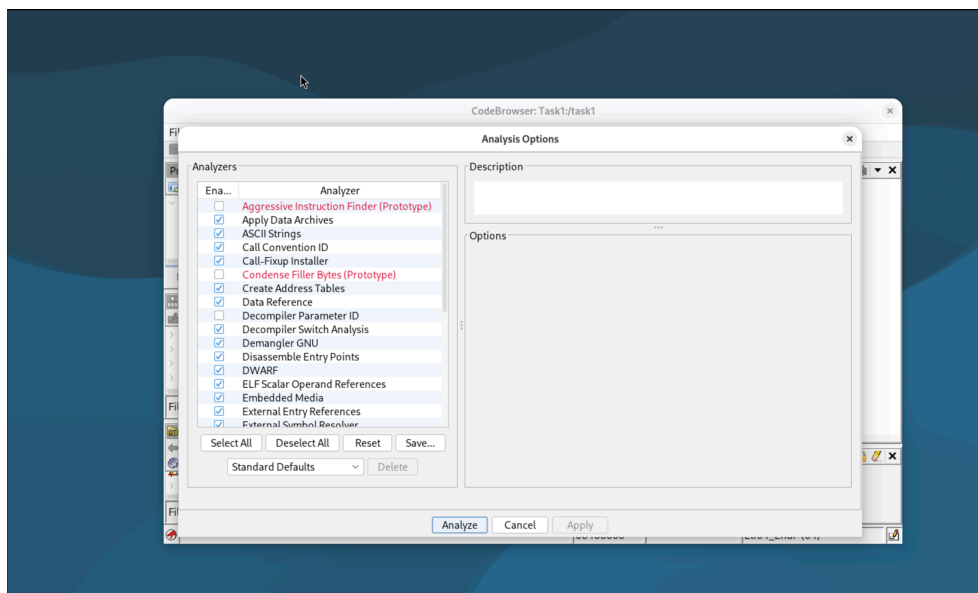
In the popup that appears you may now select the provided Task1 binary file. This binary file has both a x86-64 and a arm64 version. Please choose the one that corresponds to your architecture.

You may execute this binary. It will simply output a string to the standard output. We won't provide the source code for this binary because this task requires the binary to be a blackbox to you.

After importing you should double click the binary that we just imported. The next screens should be asking you whether you'd like to analyze the binary file. We will select "Yes":



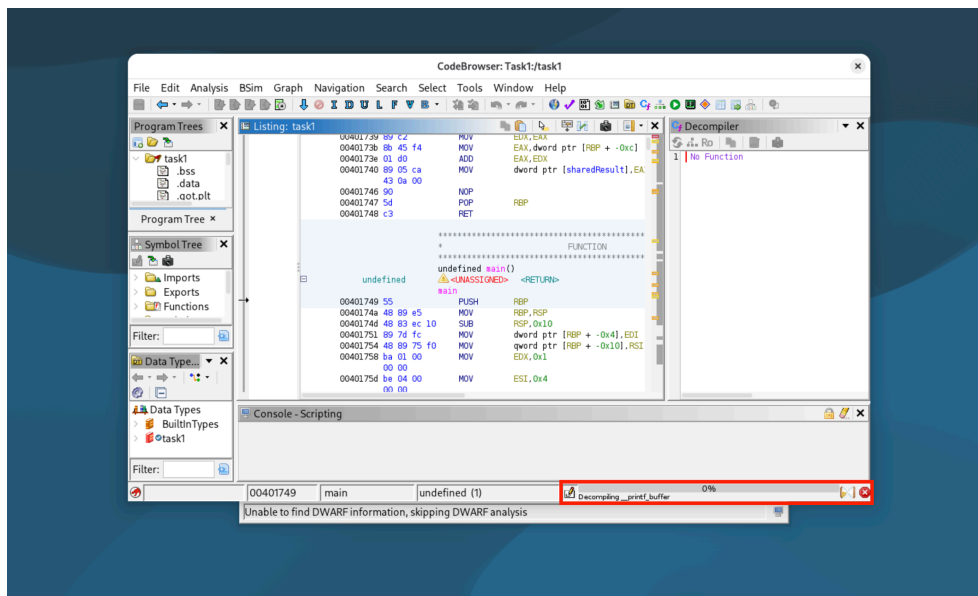
Then you should see another pop-up window where you'd normally select analysis options. These can be left alone for the most part but they can be useful. Here are a few examples:



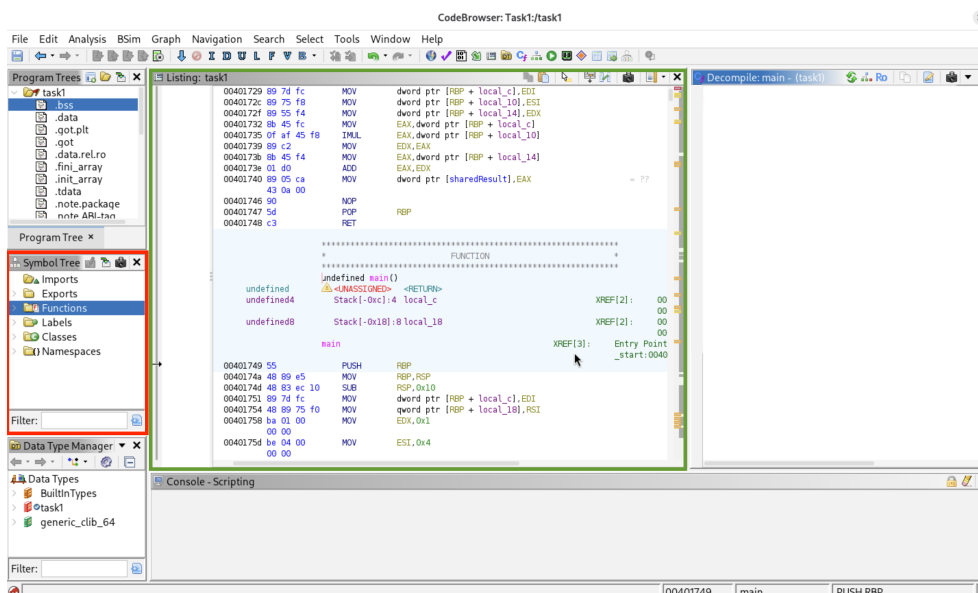
- "ASCII Strings" - Among other options it allows us to define a minimum length for string search. This can be useful if we know the size of a string that we're looking for.
- "Demangler GNU" - This is the tool that tries to translate the "mangled" programming identifiers for example `_Z4sumii` would be turned back into a recognizable function prototype such as `sum(int, int)`

When you're ready you can press the "Analyze" button.

Now give it some time to analyze the binary. You can keep track of the current progress in the right bottom corner.



After the analysis is done you can now start by trying to find the main function. To find functions you can either use listing view, highlighted in green or the "Symbol Tree" highlighted in red.



To use the "Symbol Tree" to find functions start by opening the "Functions" folder. After you should see a bunch of new folders. Since for this first task the main objective is to understand how this works you want to look for the main function.

If you opt to use the listing view you need to scroll until you find the function that you'd like to see the C-like code and then click it.

To jump to other functions that are referenced in main, you may also double-click the function you want to investigate.

## Task 1 Questions

After reaching this part, please take a screenshot that shows both the disassembled code as well as the C or C-like code. On your report answer the following questions:

- Can you describe how the program functions after execution is handed over to the main

function ?

- Draw a simple flow chart of the code execution starting in the first line of the main function and ending in its return (You may use a tool like Draw.io).
- Can you make the program show the "This is not difficult" string in the standard output ?

## Task 2 - Patching instructions and changing code flow in a compiled binary

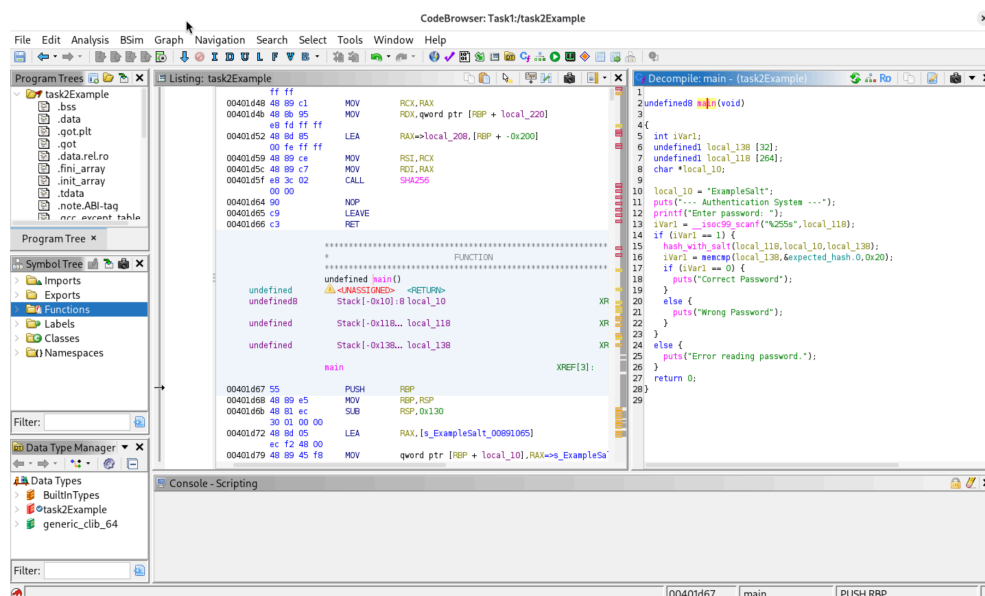
By now you know the basics of how to do reverse engineering of a compiled program / binary. In this next task we will show you how to patch an existing binary to skip over some checks that would prevent the program from executing.

### Task 2.1 - Importing the example task 2 program.

A program may do these kinds of checks in a couple of different ways. Our example binary relies on a password. If the password is right the program simulates a normal execution by printing "Correct Password" but if the password that is given is the wrong one then it prints "Wrong Password".

We want to make it so it always prints "Correct Password" independently of the input being the correct password or not.

If you open the program in the same way that we did the other ones, you should see something like this:



From this we know that the main function contains:

- Variables:
  - Two arrays:
    - 32 byte one (currently named local\_138)
    - 264 byte one (currently named local\_118)
  - Integer (currently named iVar1)
  - Pointer (currently named local\_10)
- 5 different function calls:

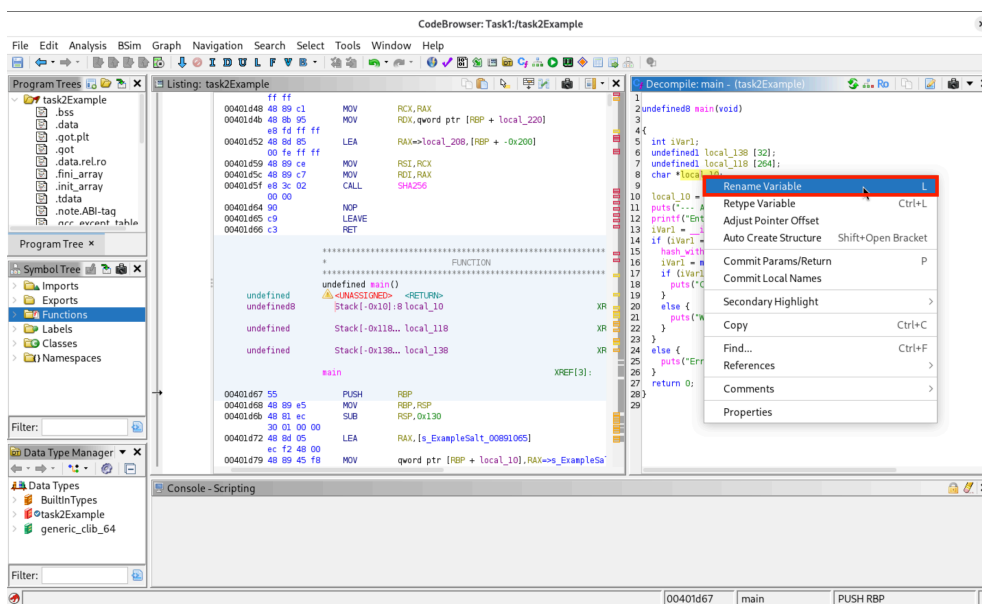
- puts
- printf
- \_\_isoc99\_scanf
- hash\_with\_salt
- memcmp

## Renaming variables and retyping variables

Since the program contains function names that were not obfuscated we can theoretically assume that the "hash\_with\_salt" function will hash something with a given salt (We'll check its behavior later).

We can also explore a little bit further and see that right away the "local\_10" variable is assigned to a value of "ExampleSalt" which indicates that this variable may be used to store the salt used in that function.

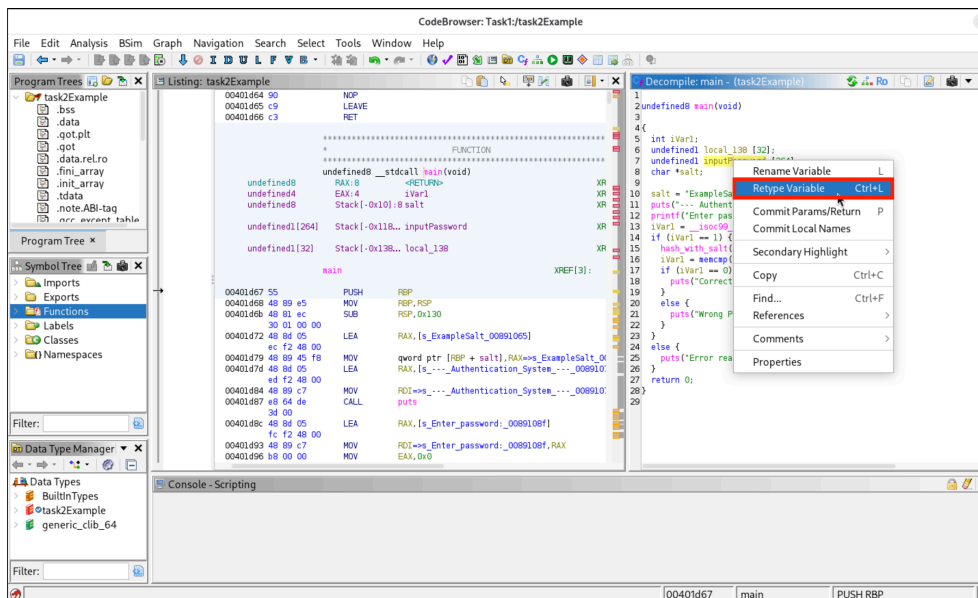
Since we made these discoveries we can now rename variables to help us understand the code better. To do this you can select the variable you want to rename which is in this case "local\_10" and right click it. Once you do you'll find a submenu. From it you should now click "Rename Variable":



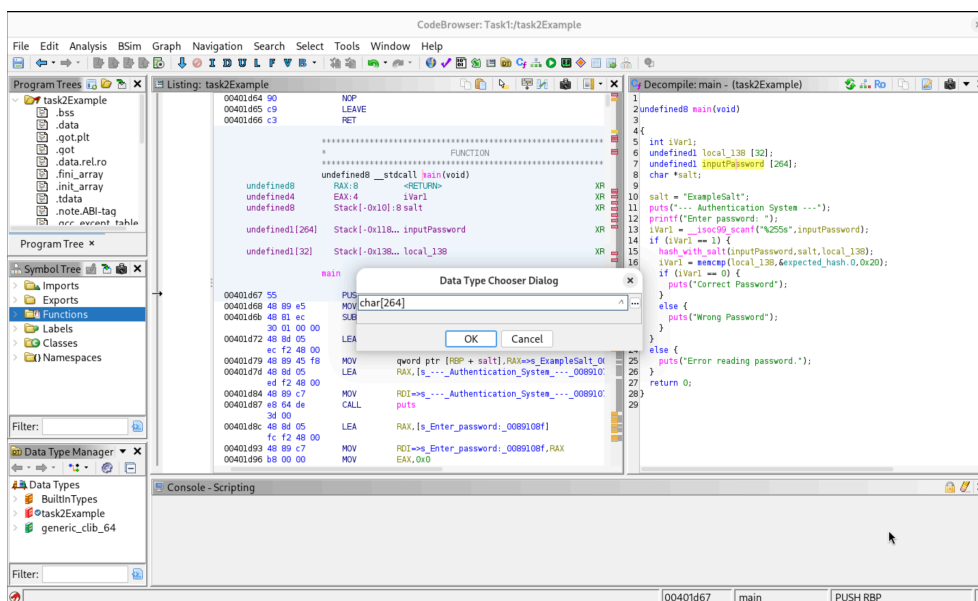
You should rename this one to something like "salt" to be able to quickly identify its function.

Now we can also rename other variables. For example, "local\_118" is used by the "\_\_isoc99\_scanf" function. Since we know that the `scanf` function is used to obtain user input and this function is called right after printing "Enter Password: " to the standard output, therefore we can name it "inputPassword".

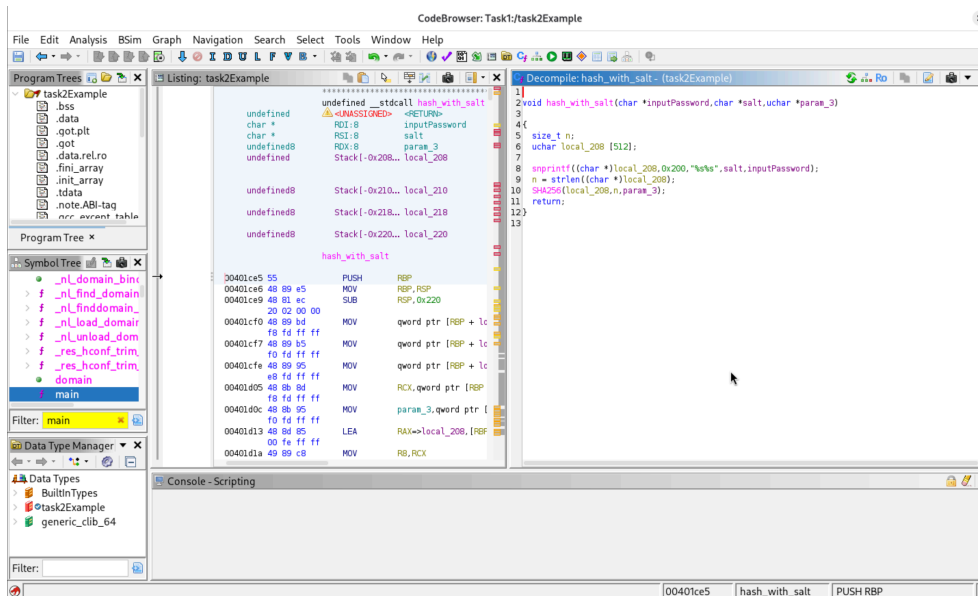
You may also notice that this variable has an unknown type. Since we know the `scanf` function and also know that the data input comes from the user, we can help Ghidra identify this type by using the "Retype Variable" tool in the right-click menu:



Rename it as follows:



In line 14 we can see that we have a call to "memcmp" and the second parameter is the expected hash (once again we only know this because the names were not stripped). If we were unsure we could still go a step further and analyze the "hash\_with\_salt" function. When we do we will start by renaming the parameters. Since we know the parameters coming from main we can use these to our advantage. By the end the inside of this function should look something like this:



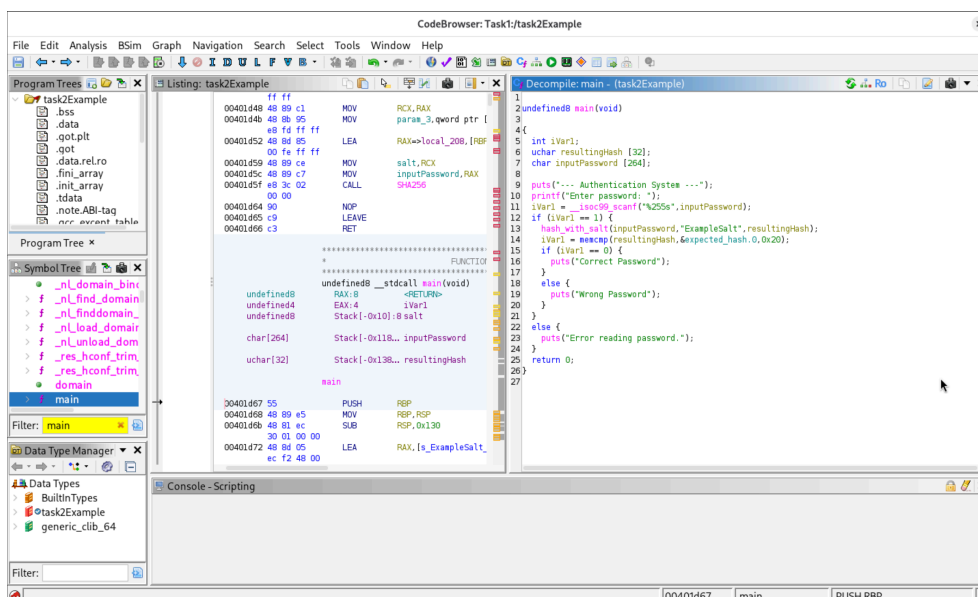
From here we can also start to see some other functions. The one that stands out is the SHA256 function. If we do a little bit of research online we find that this is the SHA256 function that the openssl library provides.

```
unsigned char *SHA256(const unsigned char *data, size_t count, unsigned char *md_buf);
```

Taking this into consideration we can now rename the variables accordingly. Note that md\_buf is the output buffer so we can rename the "local\_138" variable to something like "resultingHash" and the type to "uchar".

## Patching instructions

After renaming the last variable we now have a better idea of what the main function does:



It first creates two arrays where one is used to temporarily store the password that the user inputs, and the other one is used to hold the resulting hash which will be calculated in the "hash\_with\_salt"

function. Finally, a comparison of the 32 bytes is done, and if they match (iVar is 0), the program will output "Correct Password", otherwise the program will fail and output "Wrong Password".

This analysis is important because otherwise we would not be able to ensure a proper license patch. Some programs usually hide multiple calls in a way that patching just a single instruction does not guarantee that the program won't recheck for a license and then fail.

This example program does not do such a thing, but if it did, we might have to patch the binary in some other way (i.e., substituting the hash that it uses to compare with one that we would craft).

Now we are ready to start patching instructions. To do this, we first need to know where to patch. In this case, this is simple: if we look at the assembly that constitutes line 15, we can see:

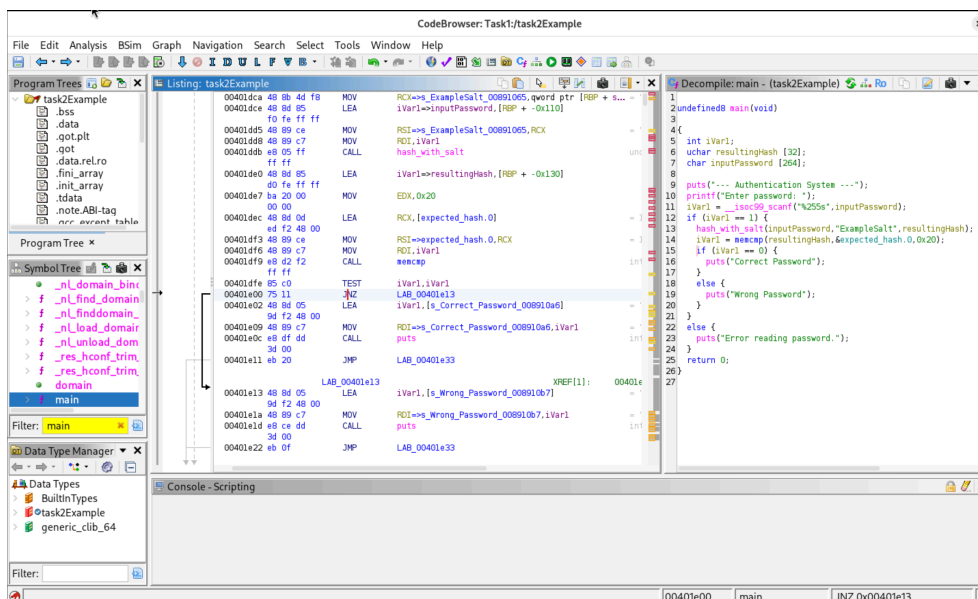
```
TEST iVar1, iVar1
JNZ LAB_00401e13
```

or if you're using the ARM version:

```
cmp w0, #0x0
b.ne LAB_00400b4c
```

Note: If you're not sure about what a specific instruction does you can always look it up online.

In both versions if you look at the left side you will see an arrow if you select the assembly code. This arrow will tell you where the program jumps if the result is not zero.



And we see that if the result is not zero, we jump to the branch of the code that ends up outputting "Wrong Password", so what we want to do is patch:



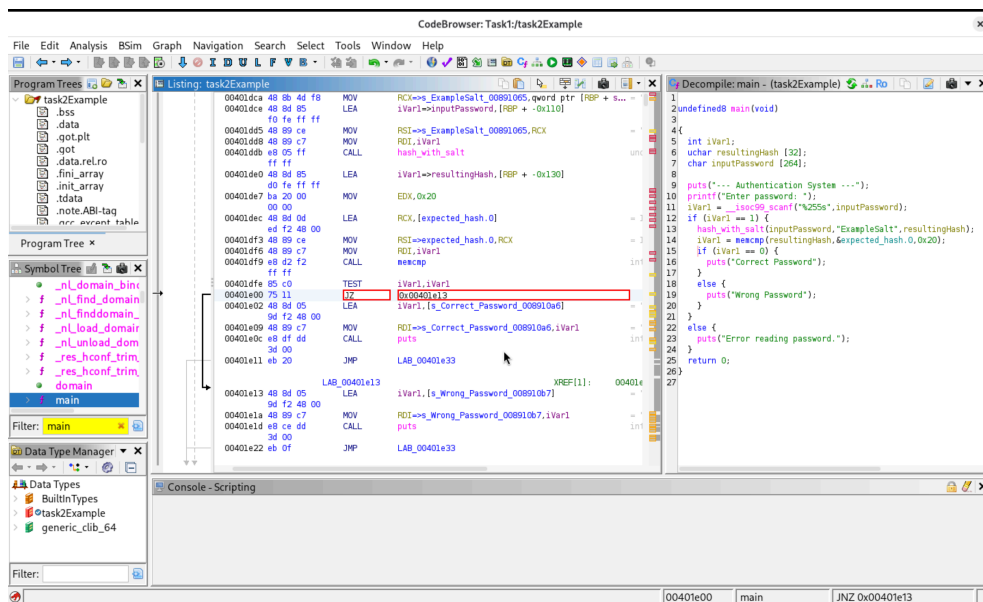
JNZ LAB\_00401e13 -> JZ LAB\_00401e13; Changes "Jump if not zero" to "Jump if zero".

or if you're using the ARM version:

b.ne LAB\_00400b4c -> b.eq LAB\_00400b4c; Changes "Branch if not equal" to "Branch if equal".

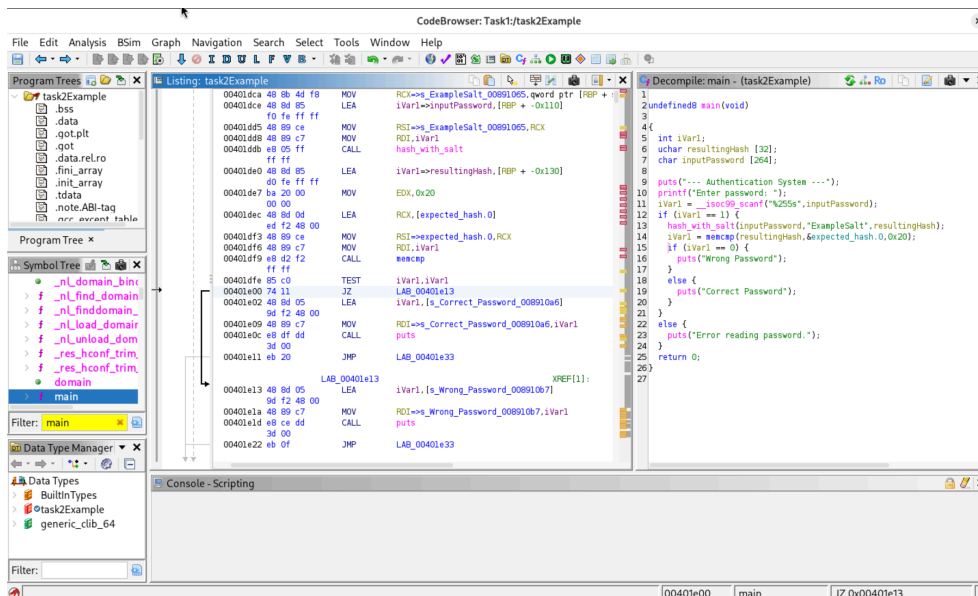
This way, we will instead enter the branch that outputs "Correct Password".

To finally do this, we can select the instruction that we want to patch, right-click it, and press the "Patch Instruction" button. After that, you should see a box prompt where you can type:



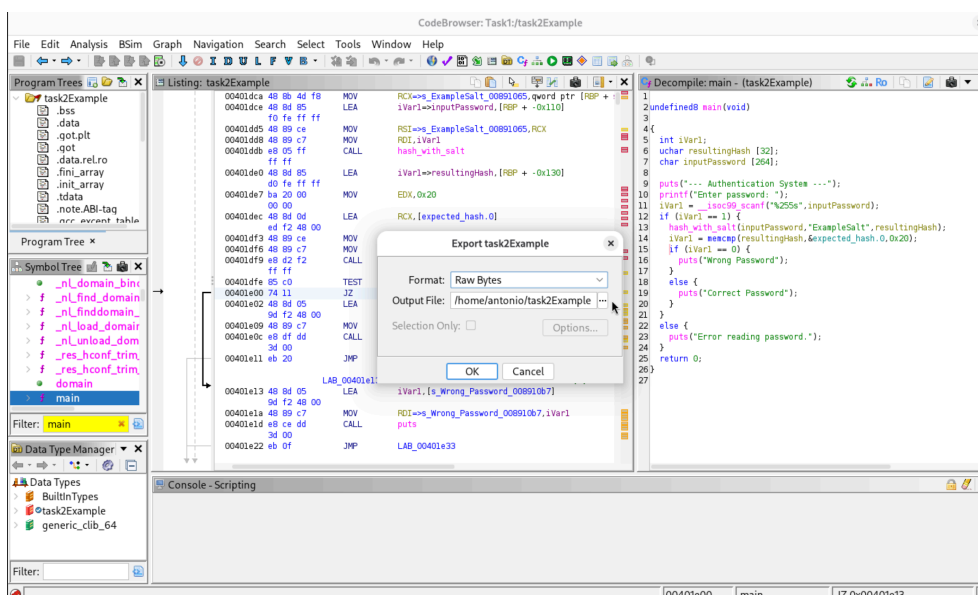
You will then need to alter the left side to the instruction that we want to execute instead. Do not alter the right side because this contains the address where we will be jumping.

You'll also notice that the right side where the C or C-like code is present has changed and now shows us that if the password is wrong, we will instead print "Correct Password":



Now all that is left is exporting the program and trying to run it on our machine. To do this, you can use the "Export Program..." option present in the "File" submenu.

Afterward, you should see a window like this one:



Make sure to select "Original File" for Format and select a directory to place the new binary file.

After you can launch a terminal and try to execute it. You might need to first do a **chmod +x** to make it executable.

```
antonio@debian:~/Documents$ chmod +x ./task2Example
antonio@debian:~/Documents$ ./task2Example
--- Authentication System ---
Enter password: a
Correct Password
```

## Task 2 Questions

Provided with this lab is another binary file named task2. If you run it you can see that it requests a license.sig file. This is bad because we don't have a way to obtain this license. Taking this into consideration you may now import this file into Ghidra and you are tasked with trying to remove this check.

The task 2 binary consists of a simple calculator app. To succeed in this task we require you to provide:

- The changes that you have done to the code for the license check to pass (Note that there may be more than one way to bypass this check).
- A screenshot of the task 2 calculator running.

## Task 3 - Finding Big Vulnerabilities

For this next task we will apply what we have learned in order to show how an attacker can leverage reverse engineering to acquire root privileges.

In this task you should use the appropriate binary file for your architecture (x86-64 or arm).

### Task 3.1 - Analysing telnetd

Telnetd (Telnet daemon) is a background server program that enables remote access to a computer, in 2026 a vulnerability was found in its code that allowed users to get root access without the need of a password.

This lab provides a compiled binary of the telnetd code that had this vulnerability present, analyse it with Ghidra and try to find the code for the function main.

Note: While telnetd is an open-source project, we are approaching this through a reverse engineering lens to demonstrate how these vulnerabilities can be identified in closed-source or proprietary environments where source code is unavailable.

### Task 3.2 - Finding a Vulnerability

As we are trying to gain unauthorised access to the root we are interested in finding the code for the login process, from the main function or through the Symbol Tree find and submit a print of the section responsible for this. You should find a code segment similar to this.

```

1  void start_login(char *host,int autologin,char *name)
2
3
4 {
5     byte *pbVar1;
6     uint uVar2;
7     char *command;
8     long lVar3;
9     undefined8 *puVar4;
10    byte *pbVar5;
11    byte *pbVar6;
12    undefined8 *puVar7;
13    long in_FS_OFFSET;
14    bool bVar8;
15    bool bVar9;
16    bool bVar10;
17    int argc;
18    char **argv;
19    long local_10;
20
21    /* Unresolved local var: char * * cpp[???]
22     Unresolved local var: char * * cpp2[???] */
23    local_10 = *(long *) (in_FS_OFFSET + 0x28);
24    pbVar1 = (byte *) environ;
25    puVar7 = environ;
26    puVar4 = environ;
27    uVar2 = lmodetype;
28    do {
29        while( true ) {
30            lmodetype = uVar2;
31            if (pbVar1 == (byte *) 0x0) {
32                *puVar4 = 0;
33                if (uVar2 == 4) {
34                    setenv("LINEMODE","real",1);
35                }
36                else if ((uVar2 & 0xffffffff) == 1) {
37                    setenv("LINEMODE","kludge",1);
38                }
39                command = expand_line(login_invocation);
40                if (command != (char *) 0x0) {
41                    argcv_get(command,"",&argc,&argv);
42                    execv(*argv,argv);
43                    __syslog_chk(3,1,"%s: %m\n",command);
44                    fatalperror(net,command);
45                    if (local_10 == *(long *) (in_FS_OFFSET + 0x28)) {
46                        return;
47                    }
48                }
49                /* WARNING: Subroutine does not return */

```

When you connect via Telnet, the protocol allows the client to send environment variables (like TERM, DISPLAY, or USER) to the server.

As we can see in the previous code snippet, to login telnetd uses the "argcv\_get" function to build a command-line string to execute the system's login program and "execv" to execute.

We can also observe that the server performs no sanitization on this value. It doesn't check for spaces or special characters, it just drops the string into the command, as we know this is a bad practice because it creates a massive security hole known as Command Injection.

### Task 3.3 - The Root of the problem

For this final task we want to exploit the previously found vulnerability to get root access without the need for a password.

First use the following command to install inetd to run telnetd.

```
$ sudo apt update && sudo apt install openbsd-inetd
```

After that, run the following commands to create a inetd configuration file and run the telnetd program.

```
# Create a temporary config file
```

```
$ echo "2323 stream tcp nowait root $(pwd)/telnetd telnetd" >
my_telnetd.conf
$ sudo inetd -d my_telnetd.conf
```

We provide an attack.py file that makes a request to the telnetd server, however the value for the USER variable is missing, using the telnetd documentation/manual try to find a value for USER that will allow you to login as root without a password, show a screenshot of the resulting terminal showing you have root access.

Note: If you're running a version of Python that is higher than 3.13 you may need to do some additional steps:

First, ensure that you have pip and venv installed.

```
$ sudo apt install python3.13-venv python3-pip
```

Then create a virtual environment and activate it.

```
$ python -m venv venv # or python3
$ source venv/bin/activate
```

After you'll need to install a new package because python no longer includes telnetlib:

```
$ pip install telnetlib3
```

Finally you'll need to do the following change in the attack.py file:

```
import socket
# import telnetlib <---- Comment this line
import telnetlib3 # <----- Uncomment this line
```

Tip: Remember that telnet runs with root privileges.

## Task 3 Questions

- Explain why the USER value you used bypassed the login process and what could be done to fix this problem.