Laboratory 3: Reverse Engineering Software Licence

Week 3

Welcome to Laboratory 3, in this lab you will learn how to bypass software licences. The lab is structured where you have a set of tasks and set of challenges. The tasks are meant to prepare you for the challenges that will increase in difficulty. You will use tools such as, gcc & gdb; Also you will learn how to use Ghidra. Please remember, before we introduce any assignment, we will present you with several challenges, that escalate in difficulty to help you off your feet, and break up what you need to learn to smaller chunks.

Submission Instructions

There is no marks for this Lab. However, please put your answers into the feedback sheet that will be provided in the session.

Experimental Setup

This time we have prepared two version. A vagrant file and an OVA virtual machine image, because we have not extensively tested the vagrant image, and we anticipate it may cause issues for some of you.

Vagrant

Vagrant works by reading configuration files in your current directory and using them to create VMs. Hence, in order to spin up the VM, you may need to use the cd command in your terminal to change-directory into the correct folder. Change directory into a working folder from which you wish to complete the labs. **You will need to make it if it doesn't exist, and then cd into it for example:**

```
#!/bin/bash
cd Documents/comp6236-labs
```

Download the lab0 VM image from the UoS Git server by typing the following:

```
#!/bin/bash
git clone https://git.soton.ac.uk/comp6236/lab5
```

Change into the lab0 folder so we can use vagrant:

```
#!/bin/bash
cd lab1
```

You should now be able to run the following command to

```
#!/bin/bash
vagrant up
```

If you have the virtualbox window open, you will notice that a new VM appears, and vagrant begins to build it. It should take a couple of minutes. When the build process is complete, you can connect to a shell on your newly created VM using the following command:

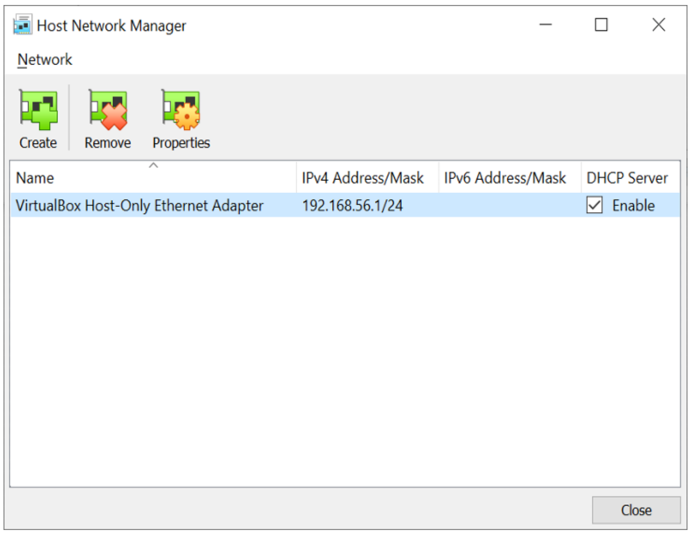
```
#!/bin/bash
vagrant ssh
```

On windows you need to install x11. Please follow these instructions

VM Image

Please download the VM from here, and import it into Virtualbox.

1. You need to go *File* → *Host Network Manager* and make a host network if one doesn't exist already.
2. Make sure DHCP enabled is ticked as illustrated in Figure 1 or vm will hang at boot forever.
3. Then go to VM network settings and check it's set to that host only network, and specify the network you created or the one that exists.
4. Wait for the VM to boot, and on boot login with User: info and Password: info to see the current IP address printed.



Name	IPv4 Address/Mask	IPv6 Address/Mask	DHCP Server
VirtualBox Host-Only Ethernet Adapter	192.168.56.1/24		✓ Enable

Figure 1: DHCP enabled

1 Tools

In this section, we will describe the most basic and essentials tools to do buffer overflows and will not go into advance tools. The reason for this, you as students need to master the essentials tools first before using specialised tools to carry advance attacks. You do not need to install any packages, everything is included in the vagrant image.

1.1 gcc

gcc is short for GNU Compiler Collection. gcc is a collection of programming compilers including C, C++, Objective-C, Fortran, Java, and Ada. You can compile a C language program by running the following command.

Once installed, a basic C file, like the one shown on our C language page, can be compiled by running the command below.

Listing 1: Compile C program

```bash
#!/bin/bash
gcc -o test test.c
```

You can review the manual page for gcc manual pages or the gcc page https://gcc.gnu.org. For example if we want to understand what the command above means, looking at the manual we can decipher it as:

Table 1: gcc command breakdown

Command	Output flag	Output name	Input file	Extension
gcc	-o	test	test	.c

1.2 gdb

gdb is short for GNU Debugger. gdb allows you to see what is going on 'inside' another program while it executes – or what another program was doing at the moment it crashed. gdb support languages like C, C++, Objective-C, Fortran, Java, Ada, and many more. You can debug a C language program by running the following command.

Listing 2: Debug C program

```
#!/bin/bash
gcc -ggdb -o test test.c
```

To review essential flags, and options please read the following cheat sheet.

Table 2: gcc command with gdb option breakdown

Command	Options	Output flag	Output name	Input file	Extension
gcc	-ggdb	-o	test	test	.c

1.3 Ghidra

Ghidra is a software reverse engineering (SRE) framework developed by NSA's Research Directorate for NSA's cybersecurity mission. It helps analyse malicious code and malware like viruses, and can give cybersecurity professionals a better understanding of potential vulnerabilities in their networks and systems.

We didn't want to introduce this tool earlier, because we wanted you to learn to use native tools that you could have on any operating system. This tool is limited in what operating system you can install it on it as we discovered building this lab. Also, since you have been using gdb, this tool should be easy to pickup.

2 Guides

Here are some online resources that you will find useful.

Ghidra tutorial in reverse engineering for window (absolute begineer): This is quick tutorial by Dr Kishou Yusa to help you get started.

Intro to reverse engineering with Ghidra: Here is another quick tutorial by Dr Vickie Li.

Editing an Executable Binary File with Ghidra: This tutorial should help you understand how to patch binaries in Ghidra.

Task 0 - Can You Check?

Your challenge is modify the number the program is comparing against. Here is the redacted code, you need to change the flow of the program to bypass the check

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int check(int mod)
{
  if(mod==[REDACTED HEX NUMBER]) {
      [REDCACTED FLAG PRINTING CODE]
    return 1;
  }
  return 0;
}

int main(int argc, char **argv)
{
 volatile int modified;
 char buffer[64];

 modified = 0;

 gets(buffer);

 if(check(modified)) {
  printf("you have correctly modified the variable\n");
 } else {
  printf("Try again, you got 0x\%08x\n", modified);
 }
}
```

First,

1. Make a copy of the binary.
2. Change permissions.
3. Decompile the binary with objdump -d

```
#!/bin/bash
cp task1 task1-modified
chmod +x task1-copy
objdump -d task1-copy
```

Breakdown

Let's try to break it down using the following questions:

Question: Where does the comparison take place that determines if you get the flag or not?

Answer: Search the binary for the Hex value.

Question: How can you modify this binary to get the flag every time?

Answer: Use Bless editor. Its installed in the VM.

Solution

1. Find the hex which needs modifying from the decompilation listing (objdump -d)
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)
3. Edit the binary with Bless hex editor
4. Run the binary again and see if your patch worked.

Challenge 0 - Simple Binaries

Now you have edited the value which was used as part of a comparison statement in the binary, try instead modifying an instruction.

Can you change the Jump instruction in an unpatched binary (on the line below the comparison), to invert the jump, thus giving the flag for every execution?

You may find the coder32 instruction reference useful: http://ref.x86asm.net/

Task 2 - Glory of Ghidra

Start Ghidra using command ghidra in command line.

```
#!/bin/bash
$ ghidra
```

from a terminal and wait a bit

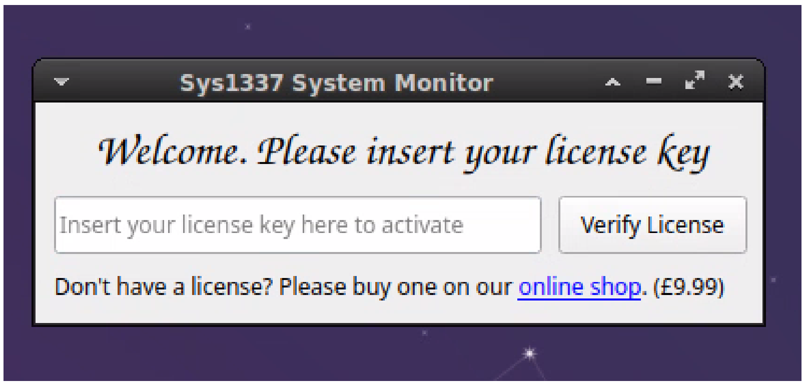
If you are using the virtual image provided use

```
#!/bin/bash
$ ghidraRun
```

from a terminal and wait a bit

Ghidra is a disassembler, just like objdump, but with a graphical interface and some intelligent analysis. Try opening the task1 binary in Ghidra and explore it. Notice how the C view on the right gives a nice reverse-engineered approximation of the function in C. Compare this against the redacted source, it's very similar see?

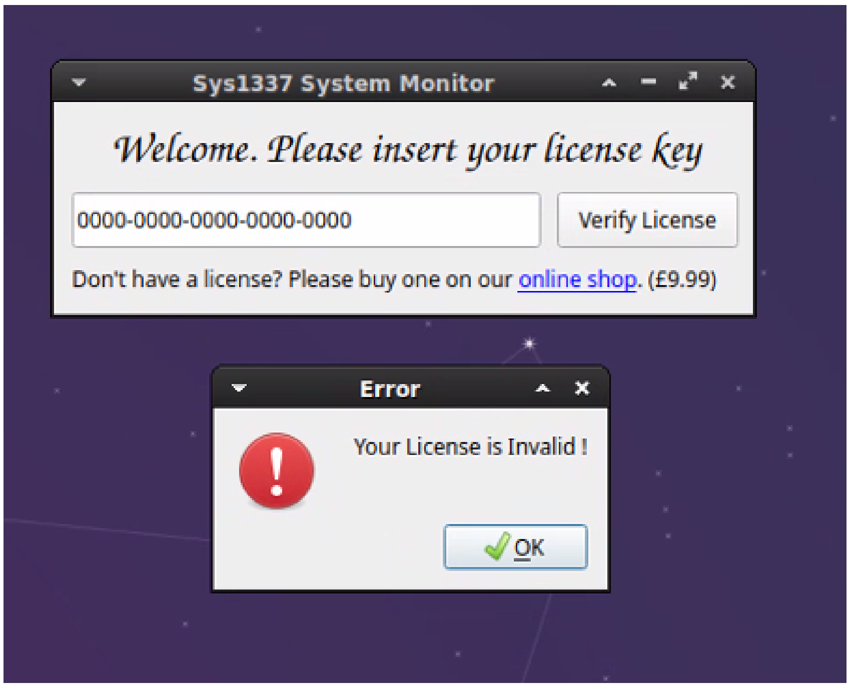
Try running the task2 binary you will see this application shown in figure 2, you notice that it requires a license key as illustrated in figure 3


Welcome. Please insert your license key

Don't have a license? Please buy one on our online shop. (£9.99)

Figure 2: Sys1337 System Monitor Application

Your task is to explore the binary, find what function checks the license, and see what happens when the license is invalid, etc. Then, use Ghidra or Bless to patch the binary to remove the license key check. Please check the resources for help.



Welcome. Please insert your license key

0000-0000-0000-0000-0000

Don't have a license? Please buy one on our online shop. (£9.99)

Your License is Invalid !

Figure 3: Sys1337 System Monitor Application Activation Screen

Golden Challenge 2 - Calculate Me

Can you figure out the algorithm task 2 is using to determine correct license keys and use it to generate your own valid license key that will work in an unpatched version of the app?