System Security
Graded Assignment 2 - Reverse Engineering

Graded Assignment

Distribution: 20.10.2022
Submission deadline: 03.11.2022 h14:00

1 Introduction

Reverse engineering—the ability to reconstruct a program's source code from its compiled binary—is an important skill in the toolbox of a security researcher. Usually, you do not have access to the source code of your target, but have to work with what you have: machine code. For this, you need to be able to make sense of *disassembled* binaries (binaries translated back to assembly) and to understand the output of *decompilers* (who try to get back to pseudocode from assembly - a task that is not easy and thus requires human intervention).

The goal of this assignment is to reverse engineer and understand the internal functioning of a binary we distribute to you: our very own AI-enhanced Pwning Assistant®. Unfortunately, your credentials only allow you to use the less powerful Student Version. To access the real power of the AI (that actually replies to Moodle posts instead of us TAs), you need to understand how to unlock it and bypass the license checks. To support you in this process, we will use tools commonly used for reverse engineering: disassemblers, decompilers, constraint solvers, debuggers, and profiler tools.

1.1 Goals and Setup

This exercise requires you to understand the internal workings of a binary previously unknown to you. Ultimately, the goal is to unlock all of its functionalities, despite it asking for a license key that you do not have.

Setup First, download your executable from
`https://polybox.ethz.ch/index.php/s/0SJvxe4Li5AijBg`.

You will find your executables in the folder named `sha256(<your legi>)`, for example `sha256(12-345-678)`: both a Linux x86-64 and a Linux ARM binary of Pwning Assistant®. They are the same, just compiled for different architectures. We recommend using the Syssec VM to run and reverse the binaries:

- If you have an x86-64 machine, use the regular Linux VM.
- If you have an ARM machine (e.g., Apple Silicon), use the provided ARM Linux VM.

The required dependencies of the executable are `jq` and `curl`: ensure they are installed in the OS you will run the executable in. For example, in Ubuntu, use

```
sudo apt-get install jq curl
```

You can also disassemble and decompile the binary on your native Windows or MacOS system (e.g., with IDA or Ghidra), but you will only be able to run the executable and attach a debugger to it on a Linux environment of the appropriate architecture.

Please note! IDA Free only provides a decompiler for `x86-64` executables, and it requires Internet access. If you are working on an ARM machine you can reverse the x86-64 version in Ida (but you won't be able to attach a debugger later—an optional but useful step in one of the questions), or reverse the ARM version in Ghidra.

2 Exploring the binary

Let's begin by gathering some useful information about the binary.

- What type of file is the provided executable? Run the `file` command on it. What is the meaning of *stripped*? How can you see whether this executable is stripped or not?
- Often the easiest way to get information about the behavior of a binary is to run it! Execute the binary and play around with it, for example by entering your Legi number.

  Try to `strace` the program while you run it. What does `strace` report? Can you spot something interesting?
- The first step to reverse engineering an executable is interpreting it as assembly code – its (more or less) human-readable form. Use `objdump -D` for this. Understand the output, which shows you the binary itself on the left and its interpretation as operation *mnemonics* and their operands on the right. What do all the function names you can see have in common? (Remember your answer to the first question.) Find the main the `main` function. What is its address?

3 Understanding the Student Edition

Reverse engineering tools as IDA or Ghidra are convenient because they offer a *decompiler*: a tool that tries to construct the (pseudo-) code representation from the disassembled code. Unfortunately, this process is not perfect: decompilers use patterns and heuristics to try to reconstruct high-level code. However, compiler optimizations and the general difficulty of the problem[1] make their output imperfect. Further, most of the time, you won't have *symbols*, such as variable and function names. It is up to you, the reverse engineer, to reconstruct them by reading and understanding the generated pseudocode.

Open the binary in the reverse engineering tool (i.e. IDA or Ghidra) of your choice.

- To find the function that prints the very first line, find the string in the strings panel and find the function it is referenced from. We refer to this function as `function0` in the following. Which function is this? What is its address as reported by Ida/Ghidra?

  Study carefully the initial screen that Ghidra or Ida give you: there is assembly code divided in *basic blocks* and forming a *control flow graph.* What is a basic block in this context? How are basic blocks usually separated? What is a control flow graph?

[1]Decompilation is undecidable in general; unless you assume the number of programs is finite, in which case it might only be very hard – not even including subproblems in decompilation like pointer aliasing which are undecidable on their own.

- `function0` does not seem to be the one that asks for your Legi – but you can spot a function call early on in the function body. We'll refer to the called function as `function1`. What is the name of `function1` according to the decompiler? Does the name represent the purpose of the function – why / why not?

- Now, find out what `function1` does. Remember to rename variables and function parameters when you have a hypothesis on what their purpose is to keep track of your progress. Similarly, when you think you figured out what the function does, rename it with a tentative name. What could be a good tentative name for `function1`? What does `function1` do?

- Go back to `function0`. There, you can see the value returned by `function1` gets passed to another function. We'll call that one `function2`. Before reverse engineering `function2`, first understand how it is embedded in `function0`. This will help you later when you do a deep dive in it. Which possible return values can we spot from the checks that `function0` performs with the return value? What does each value correspond to in terms of access level? To answer this, think about what you already know: the string that the program prints when you enter your own Legi, the strings you can see in the binary, and what the program does with the return value.

- Now you should have a good understanding of which value `function2` should return when passed (i) your Legi; (ii) some other specific number that you still do not know; and (iii) any other value. Let's dive deep into `function2`. Read the code and rename variables according to your understanding. Use your knowledge of the function return values as well. What is the mathematical operation that is done on the input Legi value? How does the program hide the expected Legi values, to avoid being detected by, e.g., simple string search on the binary?

  A means to answer these questions, besides statically reading the decompiled code and using one's intuition, is to attach a *debugger*: a tool that allows you to set breakpoints in specific points of the program. When execution reaches a breakpoint, it pauses, and allows you to inspect the program state, variables, what is in the stack, and single-step the execution through the instructions. To know where to set breakpoints, and where to single-step, remember you can switch from the decompiled to the disassembled views easily in Ida and Ghidra: this allows you to understand the mapping between assembly instruction(s) and higher-level code (because gdb will only show you the disassembly, but does not carry a decompiler). A more complex but very powerful approach is to use the capability of Ghidra or Ida to attach to a debugger, to follow the execution from the decompiled code. If you want, you can also try doing it this way.

- Go back to `function0` and reverse the function (referred to as `function3`) that gets called when execution takes the `if` branch corresponding to inputting your Legi number. What does `function3` do? What are its most frequent types of operations? How does the prompt work – which commands does it accept and what do they do?

  Note: At the end, it seems to call a function pointer, the value of which is not known until runtime – ignore it for the moment.

- With the above function, we now reversed all functions related to the Student Edition. We need to go back to the only finding we did not use yet: there seems to be a Legi input for which the program unlocks the Lecturer Edition. What is this value? How did you compute it?

4 Unlocking the Lecturer Edition

You now know how to bypass the Legi check with the special value – however, there seems to be a license check that still prevents us to access the pro functionality. You will now reverse this second part and ultimately write a *keygen* for the software.

- First, get some fast-track satisfaction and immediately access the Lecturer functionality by patching away this check: you can use Ida, Ghidra, or a *hexeditor* to edit the executable and create a copy that does not have the check. To do this, you can adopt different approaches, e.g., `nop` away the function call, or change the `jmp` or `cmp` of some parts of the code. What approach did you choose? Describe your solution in detail. Check that it works by running your modified executable and leveraging the raw power of our AI.

- In software piracy, what you did above is called a `crack`: a modified executable that bypasses license checks (here, the license number) for unauthorized software use. Another approach is instead to understand how the license check works and writing a generator of valid inputs that bypass the checks—a *keygen*[2]. Let's start to understand how the license code check works. We can observe that the first thing it does, to read one's license input, is to convert it into a different format. Which input format does the license code have? How does the program read and interpret this value? Where is this conversion happening?

  Let's dive deep into the function that seems to perform the license check. What does it do and how does it operate on the user input? How many "helper" functions does it use for its checks? Describe each in detail.

- The logical conditions on the license check are too many and too convoluted to be solved by eye. We can use a *solver* like `z3`: we can program it in Python, and find a solution to the check. How does your model look like? Which features did you use from `z3`? What is the final product license you find for your executable?

  Note: This last question is challenging.

[2]This type of illicit software is being phased out with, e.g., online activation checks.