CSC 583 Spring 2024 Lab 1

Dr. Si Chen

Stack and Stack Frame

The goals of this lab are to:

- Understand the concepts of **Stack** and **Stack Frame**.
- Navigate and analyze binary executables using **OllyDbg**.
- Understand the role of **function calls**, **parameters**, and **return values** in software functionality.
- Develop strategies for binary modification that achieve desired outcomes without introducing errors or instability.

Introduction

This lab is inspired by a series of reverse engineering tutorials aimed at beginners, showcasing the practical application of code analysis techniques. The tutorials, popular for their approachable format and comprehensive coverage, serve as a valuable resource for those interested in delving into the realm of software reverse engineering. Through this lab, participants will gain hands-on experience with binary manipulation, enhancing their understanding of software internals. The primary focus is to apply learned concepts in a controlled environment, reinforcing theoretical knowledge with practical skills. This approach not only demystifies the underlying mechanics of executable files but also equips learners with the tools necessary for effective problem-solving in the field of cybersecurity.

Background

The lab exercises are structured around a challenge involving a binary file, **lab1.exe**, that presents a "Nag screen" upon execution. The task is to remove this screen by modifying the binary, using **OllyDbg**, a popular debugger for reverse engineering. This scenario mimics real-world reverse engineering tasks, where understanding the software's flow and manipulating its execution are essential skills. By engaging with this exercise, learners will navigate the complexities of binary analysis, gaining insights into the software's structure and behavior, and the impact of assembly-level modifications.

The **lab1.exe** for debugging is crafted in Visual Basic. Before diving into debugging, it's useful to grasp the features of Visual Basic files.

0.0.1 VB-Specific Engine

Visual Basic files leverage a VB-specific engine called **MSVBVM60.DLL** (Microsoft Visual Basic Virtual Machine 6.0), also known as The Thunder Runtime Engine. For instance, to display a message box, VB code necessitates calling the **MsgBox** function. Actually, the VB editor genuinely invokes the **rtcMsgBox** function within **MSVBVM60.DLL**, which consequently functions by calling the **MessageBoxW** function (Win32 API) inside **user32.dll** (this can also be directly invoked in VB code).

0.0.2 Native Code and P-Code

Based on the compilation options employed, VB files may be compiled into native code (N-Code) and P-Code. Native code generally utilizes IA-32 instructions more decipherable by debuggers, whereas P-Code is an interpreter language utilizing self-parsing instructions (bytecode) through a virtual machine implemented by the VB engine. Accurate parsing of VB's P-Code necessitates analysis of the VB engine and emulation implementation.

0.0.3 Event Handlers

Visual Basic is extensively used for developing GUI programs, making its IDE interface ideally suited for GUI programming. VB programs operate on an event-driven model employed by the Windows operating system, meaning that user code does not exist in functions like **main()** or **WinMain()**; instead, user code is found within various event handlers.

0.0.4 Undocumented Structures

VB utilizes various structures to store information (such as Dialog, Control, Form, Module, Function, etc.) within the file. Since Microsoft has not officially disclosed these structures, debugging VB files can become somewhat more challenging.

Objectives and Targets

Download `lab1.exe` to your Windows XP VM and run it to display a Nag screen, as shown in Figure 1. Your task is to completely remove the Nag screen by modifying the binary program using `OllyDbg`.

Experiment Setup

1. Start Windows XP in VirtualBox.
2. Inside Windows XP, download or copy `lab1.exe` to a folder, accessible via:
   `https://www.cs.wcupa.edu/schen/sec24/download/lab1.exe`
3. Open `lab1.exe` with `Ollydbg`.

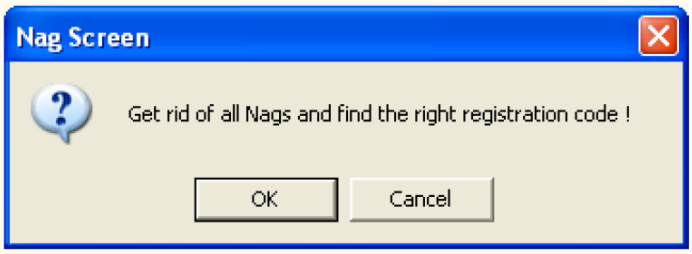

Figure 1: Nag Screen of lab1

4. Review the following questions in "Lab Exercise" and include your answers in your report.

Lab Exercise: Remove the Nag screen

Step 1: Analysis and Patching Process

The goal of this exercise is to eliminate the Nag screen by modifying the program's execution flow. This involves identifying the specific function call that triggers the Nag screen, which, through analysis with `OllyDbg`, is found to be a call to the `rtcMsgBox` function from the Visual Basic runtime library, `MSVBVM50`.

Step 2: Identifying and Modifying the Function Call

Utilizing `OllyDbg`'s "Search for - All intermodular calls" feature helps locate the `rtcMsgBox` function calls. Setting breakpoints on these calls allows us to pause execution right at the critical moment before the Nag screen is displayed, offering a precise location for modification.

Upon analyzing the call at 0x402CFE, Dr. Chen decides to modify the "CALL XXXX" instruction to "ADD ESP,14", effectively skipping over the call to the Nag screen function. To maintain the integrity of the code flow, two "NOP" instructions are added to fill in the gap left by the shortened instruction. This strategic patch removes the Nag screen without altering the program's core functionality.

However, this modification caused errors because `rtcMsgBox()` needs to return a value of 1 to indicate successful display, which his modification did not account for.

Question: Which CPU register is used to store the return value (1) of the function `rtcMsgBox()`? Why?

Dr. Chen found another way to "hack" this program by changing the instruction at 0x402C17 from "PUSH EBP" to "RETN 4", successfully removing the Nag screen.

Question: What is the meaning of "PUSH EBP, MOV EBP, ESP"?

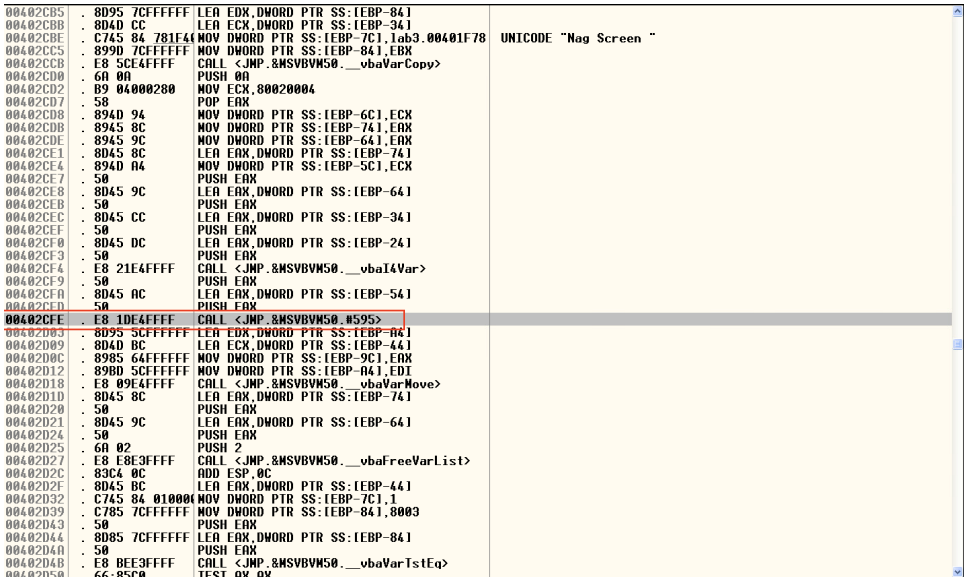

Figure 2: Disassembled code showing the call to rtcMsgBox at 0x402CFE

Question: Please explain why changing the instruction at 0x402C17 from "PUSH EBP" to "RETN 4" removes the Nag screen.

Lab Exercise: Finding the Registration Code

In this exercise, you will explore the process of identifying and validating a registration code for lab1.exe using OllyDbg. The goal is to understand how conditional checks and string comparisons are performed in assembly language and how to manipulate these checks to discover the correct registration code.

Step 1: Analyzing the Initial Message Box

When an incorrect registration code is entered, the program displays a message box indicating the failure. Your first task is to locate the assembly code responsible for this message box.

Question: Based on the search for referenced text strings in OllyDbg, identify the address where the "RegCode is wrong!" message is handled. What instruction is immediately above this referenced string?

Step 2: Finding the String Comparison Function

The program uses a string comparison function to compare the entered registration code against the correct one. This function is typically named `__vbaStrCmp` in Visual Basic compiled programs.

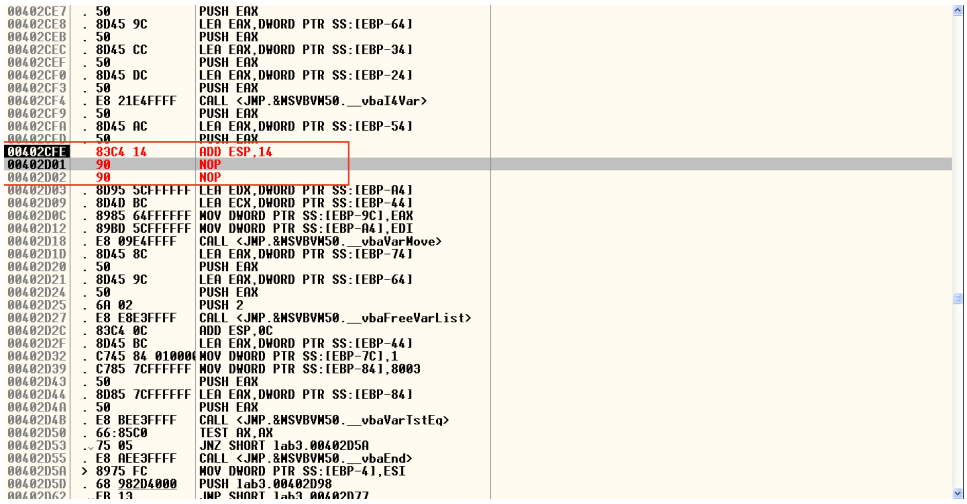

Figure 3: Modified disassembled code of lab1

Question: Locate the call to the `__vbaStrCmp` function related to the registration code check. What are the two strings being compared by this function?

Step 3: Discovering the Correct Registration Code

Upon finding the string comparison, you'll notice the hardcoded correct registration code in the vicinity of the comparison function call.

Question: What is the hardcoded registration code found near the `__vbaStrCmp` function call? Hint: Look for a string that is compared against the user input.

Step 4: Confirming the Registration Code

After identifying the correct registration code, the next step involves confirming its validity within the program's flow.

Question: After entering the discovered registration code, which address contains the assembly instructions for displaying the success message box?

Bonus Question: Considering the assembly instructions that handle the correct registration code, how does the program flow differ from handling an incorrect code?

Note: Remember to document your process, including screenshots and explanations of how you located the correct registration code. This will form the basis of your lab report submission.

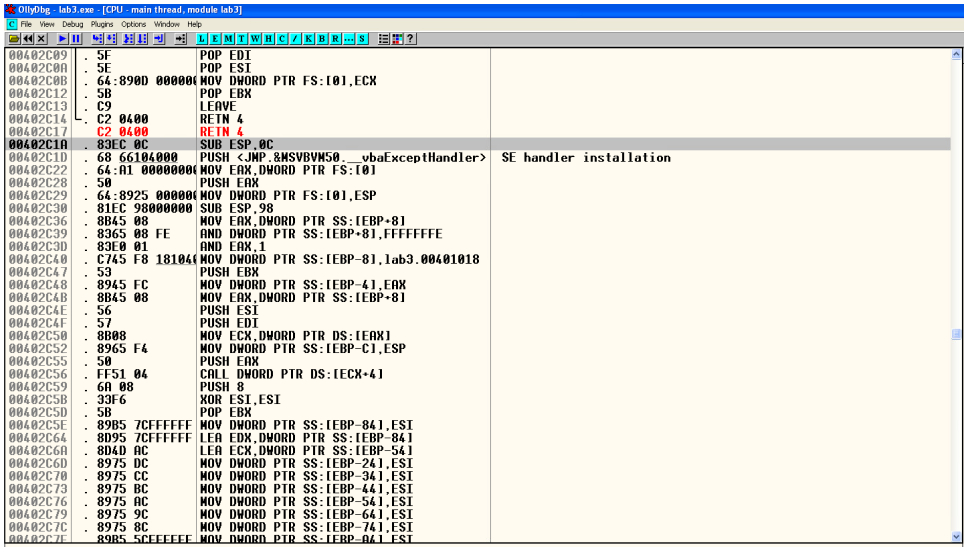

Figure 4: Another method to modify the disassembled code of lab1

Hint

Check the lecture slides and video – Class 4 Stack and Stack Frame, and Class 2 for IA32 CPU register and X86 ASM basics.

Submission

- The lab due date is available on our course website. Late submissions will not be accepted.
- Submit your assignment directly to D2L.
- Include a detailed project report in PDF format describing your process, including screenshots of the final result.
- No copy or cheating is tolerated. If your work is based on others', give clear attribution, or you will fail this course.