A Sophomore-Level Reverse Software Engineering Project in Computer Systems

Cynthia C. Fry
Department of Computer Science
Baylor University

Gennie Mansi
Senior, Statistics, Honors Program
Baylor University

Kevin Kulda
Senior, Baylor Business Fellow, Management Information Systems
Baylor University

Abstract

On your first day on the job with a new company, you are presented with a challenge. A piece of executable code has been found on an older server, and you must determine what the code is designed to do. In CSI 2334, "Introduction to Computer Systems (Computer Systems)", we introduce to the students a group project simulating such an event. Group projects are used frequently to provide similar learning environments that capitalize on the benefits of peer-to-peer instruction and cooperative learning.

The challenge is presented, the students are put on teams, and then the work begins. This paper will document the process taken by the student teams to:

- Determine how to view a binary file.
- Determine what tools are available for use.
- Work with the tools and the executable file to determine whether the file is an old game, a piece of malicious code, or both.
- Once the nature of the binary file is known, students will
  1. Modify game play,
  2. Quarantine the malicious code, or
  3. Both.
- Formulate a final report and presentation to be made to a panel of experts.

This paper will document the process conducted by one of the student teams from the Spring 2019 semester, and the methods of assessment used to evaluate each team's results.

Copyright ©2020, American Society for Engineering Education 1

1 Introduction

So how does one explore an executable without executing it?

Given the plethora of malware, techniques in detecting malicious code have evolved quickly. Novel security threats such as the inclusion of malware that can decompress and decipher itself only increase the need for new detection techniques [1]. Additionally, hackers obfuscate the executable's function, making it difficult to identify and isolate malicious segments of code. There are a variety of ways to obfuscate code, including encoding the data in unusual forms, reordering loops and expressions, inserting irrelevant code, removing comments, and removing or replacing identifier names [2]. Consequently, researchers aim to assist the tedious and time consuming process of manual deobfuscation by developing tools that can quickly detect, analyze, and simplify obfuscated code.

An increasing number of deobfuscation tools use machine learning techniques. For example, a group of researchers from Carnegie Mellon developed a tool that uses machine learning techniques to perform semantic analysis of code in order to remove instructions that write to registers or memory without contributing to the outcome or effect of the function. They also applied related techniques to detect similarities in malware semantics [3]. Other research entails applying Deep Learning based systems and Deep Neural Networks [4], classification [5], and different versions of the perceptron algorithm [6]. Research in malicious software detection even extends into detecting malicious smartphone applications, which is becoming an increasing concern with the advent of open source platforms such as Android [7].

In practice, the most common and well-known reverse engineering tool is IDA, particularly IDA Pro, which is a state-of-the art dissembler [5]. Plug-ins developed for IDA include tools such as a deobfuscator developed by the Riverside Research Institute, which uses pattern recognition and instruction emulation to deobfuscate code [8]. The major disadvantage to IDA Pro is that its license is extremely expensive.

In April 2019, the National Security Agency (NSA) released a free, open-source disassembler called GHIDRA that can analyze compiled code from Windows, Mac OS, and Linux. It can perform decompiliation, assembly, graphing and scripting, and supports plug-ins and scripts that other developers create [9]. This disassembler has been used widely in US government agencies to investigate suspicious software and various malware strains. Some have speculated that GHIDRA may catch up to IDA, since the NSA released it as an open source program [10]. Another tool that is becoming increasingly popular due to its clean interface and its adaptability to a variety of platforms, architectures, and compilers is BinaryNinja [11].

In this paper we will provide an overview of the project from the students' perspective. This perspective will include: Project Initiation, Students' Approach to the Problem, Capabilities of Various Online Tools, Discovery: Functionality of the Project, Design: Modifications to Malicious Segments and Functionality, and Implementation.

2 Project Initiation

Computer Systems is a sophomore-level semester (15-week) course which introduces the concepts of computer system organization, and examines the relationship between software and

Copyright ©2020, American Society for Engineering Education 2

hardware. Computer organization and representation of information in a computer are also discussed. The Intel 80x86 family of processors are used as the basis for this study. Much of the course focuses on the low-level language interpretation structure provided by the x86, with low-level language assignments for the Intel 80x86 on a personal computer.

On the first day of class the students are told that an unknown executable has been downloaded to their machines. They are then presented with a series of questions to help connect the relevance of the current (as well as preparatory) course to the project challenge:

- How do you know if the file is safe to open?
- What might happen when you open the file?
- Is there a way to tell what the file's behavior is before opening it?

After an in-class discussion, the students are asked to answer these questions in a brief white paper.

Mid-way through the semester, once computer organization, the basic instruction set, the debugger and the disassembler have been presented, the group project is introduced. For the Spring 2019 semester, the project was introduced as follows:

> *"This morning our internal servers, while doing some routine history scans, discovered a newly cataloged executable that had no tracking data available. As this raises our Agency's security level to orange, your team has been assigned to determine what this code does."*

Further topics are presented (reversing, methods of reversing, types of malware, etc.), and the students are tasked with doing some independent research on how to determine the behavior of the mysterious binary file. In particular, the teams are asked to investigate currently available software to assist in their project. Incrementally, their findings are presented to the instructor in a series of progress reports. They have the remaining weeks in the semester to complete their investigation, present their findings, and turn in a final report after completing a final presentation.

The groups are determined through a rigorous questionnaire administered through the Comprehensive Assessment of Team Member Effectiveness system (CATME.org). Time is spent in class discussing the importance of working together and learning to value each team member's contributions. The team members must assess each other's effectiveness with both a formative assessment conducted midway through the group project, as well as a summative assessment after the final projects are submitted. Students are given two assignments regarding team building and team dynamics before they begin the formative assessment.

These assessments are graded by evaluating the consideration used by each student in their assessment of their team members. This grade becomes an individual component in the student's overall course grade and an adjustment factor is determined from it. The individual adjustment factor is used as a multiplier in each student's group project grade to dispense credit fairly among team members.

All materials used in this course in the Spring 2019 semester can be found here:

Copyright ©2020, American Society for Engineering Education 3

https://classnotes.ecs.baylor.edu/wiki/CSI 2334 Fry Spring 2019, (username: CSI2334S19, password: FryMansiKuldaASEE2020). This includes the CSI 2334 Course syllabus, the CSI 2334 Course calendar, and the CSI 2334 Spring 2019 Project (among other items).

3 Students' approach to the problem

The students' approach to the problem had three phases. The first entailed a preliminary exploration of the executable and choosing which tools to use. The next phase entailed identifying how the executable functioned and how to isolate malicious segments of code. Finally, students implemented and tested their own additional modifications.

Most students do not have previous experience with reverse engineering binary executables, so the first step is an exploration of the different tools available for reverse engineering software. A preliminary search reveals both IDA (particularly IDA Pro) and BinaryNinja as the most used platforms. Free demo versions are available for both softwares, and these are the ones most used by students. Once a platform(s) is chosen, students must explore what settings and views are most helpful for analysis, and how to switch and alter those views.

The initial exploration entails scrolling through the executable's contents, trying to separate out functions that seemed central to the program's operation versus functions that were intended to obfuscate the purpose of the executable. While students familiarized themselves with the structure of the program, they searched for strings that consistently appeared in the console window, such as 'Enter you guess'. Upon finding the string, students explored the area around that string to understand the execution flow. Anytime they lost their spot in the executable and did not know how to return, they could always return to their starting spot by searching for the string and retracing their steps. They then proceeded to identify large sections of the code: where did input and output occur? Where were the lists of guessed and unguessed letters updated? Where were the comparisons that determined if the user won or lost?

After obtaining a basic understanding of the executable's function, students searched for malicious segments to eliminate before implementing their own modifications. The bulk of the students' time was spent in this phase. They primarily used IDA's debugger to search through the disassembled executable for malicious code. The students would set a breakpoint at different points of interest and step through the program, following along simultaneously in BinaryNinja, to observe how the program behaved. Students inserted comments in the code about new patterns and functions that they discovered. Upon finding suspect segments of code, students modified the contents of the executable and observed the effects to see if the problem was eliminated. They reverted back to the previous version of the executable if the modifications had unexpected or undesired results.

Finally, students implemented and tested their additional modifications. In the previous stage, students had been deliberate in taking notes and discussing various features to alter. Therefore, they simply explored the different ideas they liked most. In BinaryNinja, once the executable was altered, the graphical view would immediately reflect the result of the alteration on the program's flow. Students altered the executable, confirmed that the alteration was the one desired, and then tested the alteration. The implementations were tested both by running the executable and by using the debugger in IDA to analyze the values stored in memory and in

Copyright ©2020, American Society for Engineering Education 4

stack registers.

4 Capabilities of various online tools

When reverse engineering the executable, 4 categories of tools prove particularly useful:

1. *Reverse engineering platforms*

   These are tools that are most distinctly characterized by providing a direct interface between the user and the disassembled code at its lowest level. The range of features offered by various platforms differs widely from platform to platform. Features may include, among others, a built-in hex editor, the ability to immediately see live disassembly, both graphical and text views of the code, and a debugger. These features aim to help elucidate the function of the code, assisting the user to better understand and explore the function and flow of both individual segments of the code and the executable as a whole.

2. *Decompilers*

   Decompilers are computer programs that construct high level source files from an executable. They effectively reverse the work the compiler does when it converts a source file to an executable. Examples of situations in which a decompiler may be helpful include debugging programs or recovering a lost source code archive. Different decompilers support converting source code compiled from different high-level languages. As compilers undergo more frequent changes, picking a recent and up-to-date decompiler is becoming increasingly important.

3. *Hex Editors*

   Upon opening an executable in a reverse engineering platform or IDE, what may appear as a series of assembly language instructions is ultimately binary data. Hex editors are programs used to alter the exact contents of that data. Hex editors typically represent the binary content of a file in hexadecimal format and usually display in parallel the ASCII character representation of each byte of code. Hex editors have a variety of uses, including correcting corrupted data and changing or adding instructions to a file without having to recompile the file.

4. *Virtual Machines*

   Virtual machines are independent instances of operating systems running on top of host operating system. Virtual machines are useful for running executables to test if the executable is malicious because the damage done by a malicious file is isolated to the virtual machine. If a virtual machine gets infected or corrupted by a malicious the user can simply destroy the instance and create a new one again.

Below is a table that summarizes different tools that were options for completing the task and challenges students had in utilizing them.

Copyright ©2020, American Society for Engineering Education 5

Tool	Strengths	Weaknesses	Use within Analysis
IDA Demo Version https://www.hex-rays.com/products/ida/	Debug, Disassemble, Graphically flowchart code	Save changes, Report errors in disassembly, Allow comments in disassembly	Debugging disassembly, Viewing disassembly, Finding object code in hex dump
BinaryNinja Demo Version https://binary.ninja/	Save changes to executable, Graphically flowchart disassembly, Report errors in disassembly	Debug, save comments, allow multiple viewing windows	Viewing disassembly, editing and saving the object code, finding stack usage errors
VirtualBox https://www.virtualbox.org/	Allow instances of different operating systems to be booted up	Install guest additions to allow full screen mode	Creating Ubuntu, Windows Server, and Windows 10 instances to run executable on
Snowman https://derevenets.com/	Easy to install and run	Decompile	Decompiling (briefly)
Boomerang http://boomerang.sourceforge.net/	N/A	Difficult to install	Did not complete installation to use
WINE https://www.winehq.org/	Supposed to allow windows programs to run on top of other operating systems	Difficult to install, could not located DLL files in directory	Did not complete installation to use

Table 1: Tools used and explored for this analysis.

5 Discovery: Functionality of the project

The students concluded that the executable was essentially a Trojan horse. Although it appeared to be a hangman game, it conducted unauthorized, malicious behavior. It generated a word of random length, composed of a random set of letters. The user is then prompted to enter a guess. If the letter guessed was not in the word, the message "Not in word" was printed to the screen, the guesses-left counter was decremented, and the user was re-prompted to enter a guess. If the guess was correct, the console displayed "Good guess !!", the guesses counter was not decremented, and the user was re-prompted to enter a guess. As the user proceeded to guess letters, the console window displayed updated lists with the user-guessed letters, unguessed letters, and letters correctly guessed in the word. If the user entered a letter that they had guessed before or if the character entered is not a lowercase alphabetic letter, the user was re-prompted to enter a guess. If all the letters were guessed correctly then the message "You won !!" was output to the console; otherwise the message "You lost" was output. In addition to this

Copyright ©2020, American Society for Engineering Education 6

basic functionality, the executable had several malicious behaviors, which are detailed later in this paper.

The code contained six challenges for students to discover and mitigate. Below is the description of the challenges from the perspective of the students' analysis:

1. *Exhaustion of Heap*

   Directly prior to outputting the unguessed letters to the screen, the executable would push the value 17D78400h onto the stack. Then, a sub-function call was entered that requested the system to dynamically allocate 17D78400h bytes worth of memory. Thus, within three to four guess entries from the user, the amount of memory on the heap would be exhausted and an error would be thrown. Figure 5.1 displays this section of the executable with students' comments denoting the key lines involved in this error.

```
int32_t __fastcall sub_426b60(int32_t* arg1, int32_t arg2 @ esi)
mov      eax, 0xcccccccc
rep stosd dword [edi]  {0x0}  {0xcccccccc}  {0xcccccccc}
pop      ecx {var_1b8}
mov      eax, dword [data_437014]
xor      eax, ebp {__saved_ebp}
mov      dword [ebp-0x10 {var_14}], eax
push     eax {var_1b8}
lea      eax, [ebp-0xc {var_10}]
mov      dword [fs:0x0], eax {var_10}
mov      dword [ebp-0x18 {var_1c}], ecx
// Here is the large value passed as the size parameter to the "new" function to
// dynamically allocate memory.
push     0x17d78400 {var_1bc}
// This call is to the "new" function that dynamically allocates memory.
call     sub_411aeb
add      esp, 0x4
mov      dword [ebp-0x11c {var_120}], eax
mov      eax, dword [ebp-0x11c {var_120}]
mov      dword [ebp-0x24 {var_28}], eax
mov      esi, esp {var_1b8}
push     sub_411271 {var_1bc}
push     0x4324e4 {var_1c0}  {"Unguessed letters:"}
mov      eax, dword [std::cout@IAT]
push     eax {var_1c4}
call     sub_411bf4
```

Figure 5.1: Screen clipping of code that causes the heap to be exhausted.

2. *Lag time error*

   Upon entering an incorrect guess, there was a noticeable delay in the game play. The lag was caused in a function that was supposed to return -1 if the guess was incorrect so the guesses counter would be decremented. This function was only called if an incorrect guess was entered. The lag was created because in the function there was a loop that iterated 7530h times, thus causing a noticeable delay in the game. Figure 5.2 and Figure 5.3 display the original executable function calls with comments.

Copyright ©2020, American Society for Engineering Education 7

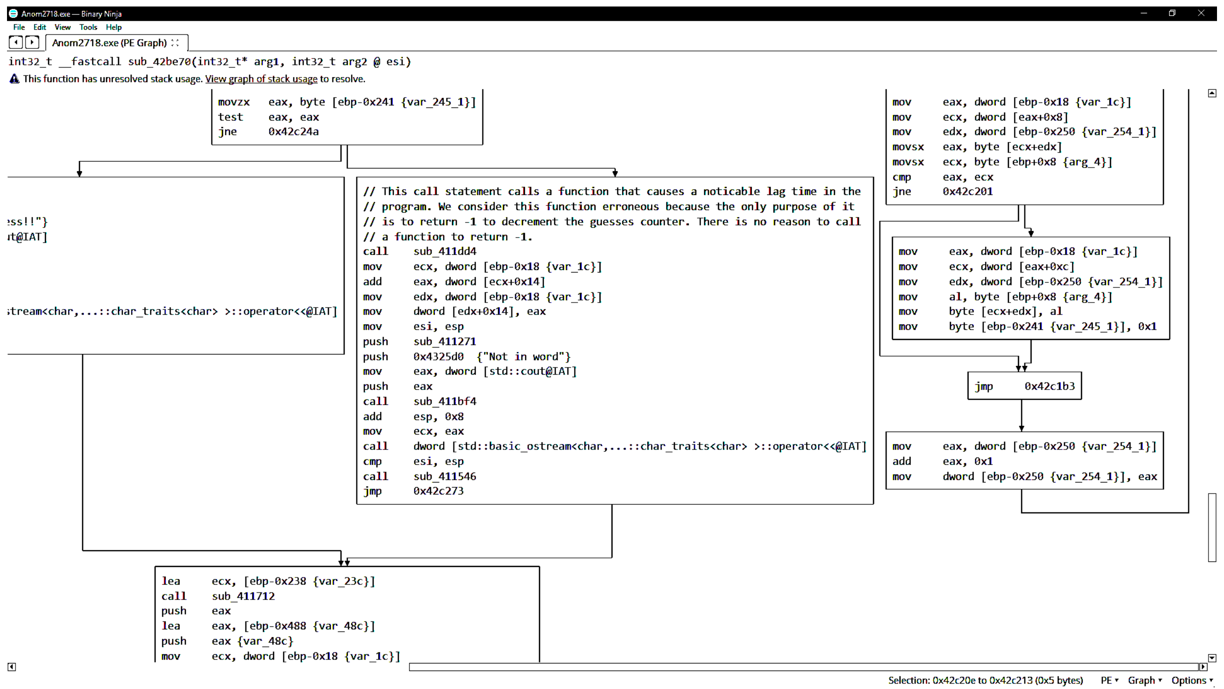


Figure 5.2: The function sub_411dd4 calls another function within itself that contains a loop that executes thousands of times, causing a lag in the game play.

Copyright ©2020, American Society for Engineering Education 8

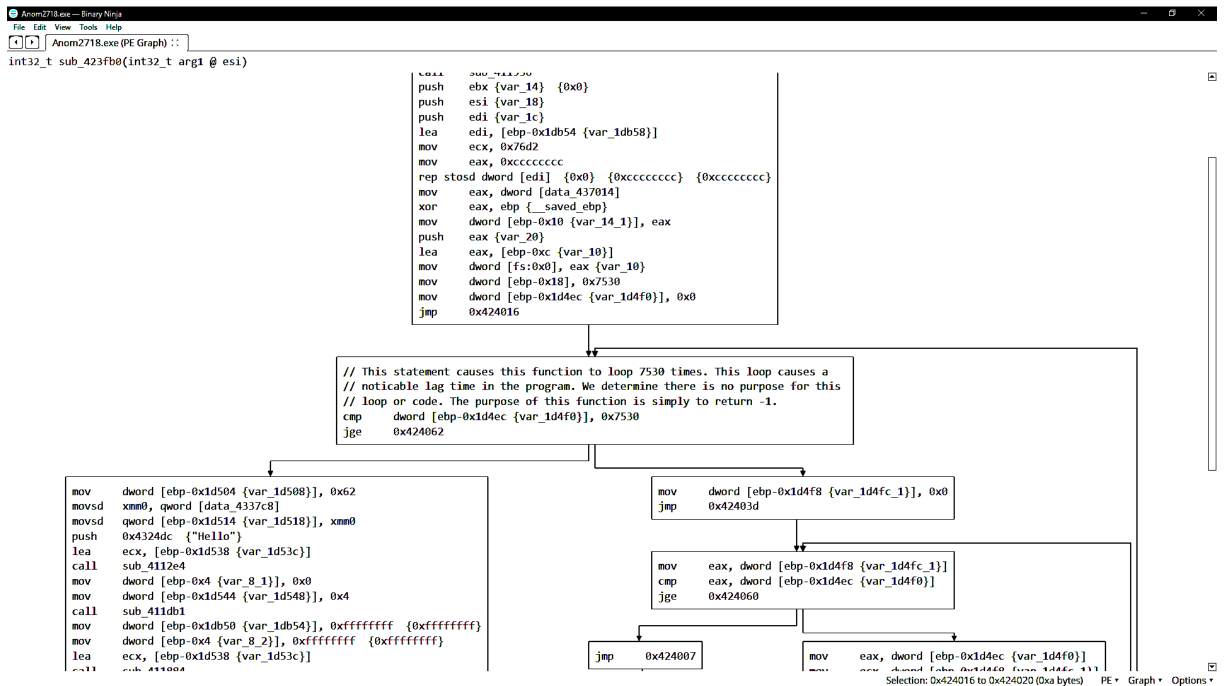


Figure 5.3: Screen clipping of the subfunction call in sub_411dd4 that contains the loop the causes the lag time error.

3. *Stack error upon guessing the letter ‘a’*

   The executable crashed immediately if the letter ‘a’ was guessed. Figure 5.4 displays the section of the executable that checks if the letter ‘a’ was guessed. Just before the comparison with the letter ‘e’, there is the instruction *cmp eax, 0x61*. As mentioned before the byte loaded into eax by the instruction *movsx eax,byte[ebp-0x8]* is the character that was just entered by the user. That value is compared against the value 0x61h, which is the ASCII character value for ‘a’. As can be seen, from the conditional jump following the comparison if the letter was not ‘a’, then the section of code that throws a stack error is skipped.

Copyright ©2020, American Society for Engineering Education 9

```
Anom2718.exe (PE Graph)
int32_t __fastcall sub_42be70(int32_t* arg1, int32_t arg2 @ esi)
This function has unresolved stack usage. View graph of stack usage to resolve.
push    esi {var_4e8}
push    edi {var_4ec}
push    ecx {var_4f0}
lea     edi, [ebp-0x4dc {var_4e0}]
mov     ecx, 0x134
mov     eax, 0xcccccccc
rep stosd dword [edi]  {0x0}  {0xcccccccc}  {0xcccccccc}
pop     ecx {var_4f0}
mov     eax, dword [data_437014]
xor     eax, ebp
mov     dword [ebp-0x10 {var_14}], eax
push    eax {var_4f0}
lea     eax, [ebp-0xc {var_10}]
mov     dword [fs:0x0], eax {var_10}
mov     dword [ebp-0x18 {var_1c}], ecx
movsx   eax, byte [ebp+0x8 {arg_4}]
// Below is the instruction that checks if the letter 'a' was guessed, so that a
// stack error can be thrown.
cmp     eax, 0x61
jne     0x42bec8

movsx   eax, byte [ebp+0x8 {arg_4}]
// Below is the instruction that checks if the letter 'e' was guessed, so that
// 100 text files can be created.
cmp     eax, 0x65
jne     0x42c0dc

call
jmp

mov     dword [ebp-0x130 {var_134}], 0x101
```

Figure 5.4: Screen clipping of code where the executable checks if the letter ‘a’ or the letter ‘e’ was guessed.

4. *Malicious text file creation*

   If the user guessed the letter ‘e’, the executable created 100 text files all with the same content (“Hey there”) and saved these text files to the desktop of the machine on which it was run. Figure 5.4 is an image of the section of code that includes the instruction used to check if the letter ‘e’ was guessed. The comments in the image denote which instruction performs this check.

   The byte loaded into eax by the instruction *movsx eax,byte[ebp-0x8]* is the character that was just entered by the user. That value is compared against the value 0x65h, which is the ASCII character value for ‘e’. As can be seen, from the conditional jump following the comparison if the letter was not ‘e’, then the section of code that creates the text files is skipped. Otherwise, the section of code displayed in Figure 5.5 is entered.

Copyright ©2020, American Society for Engineering Education 10

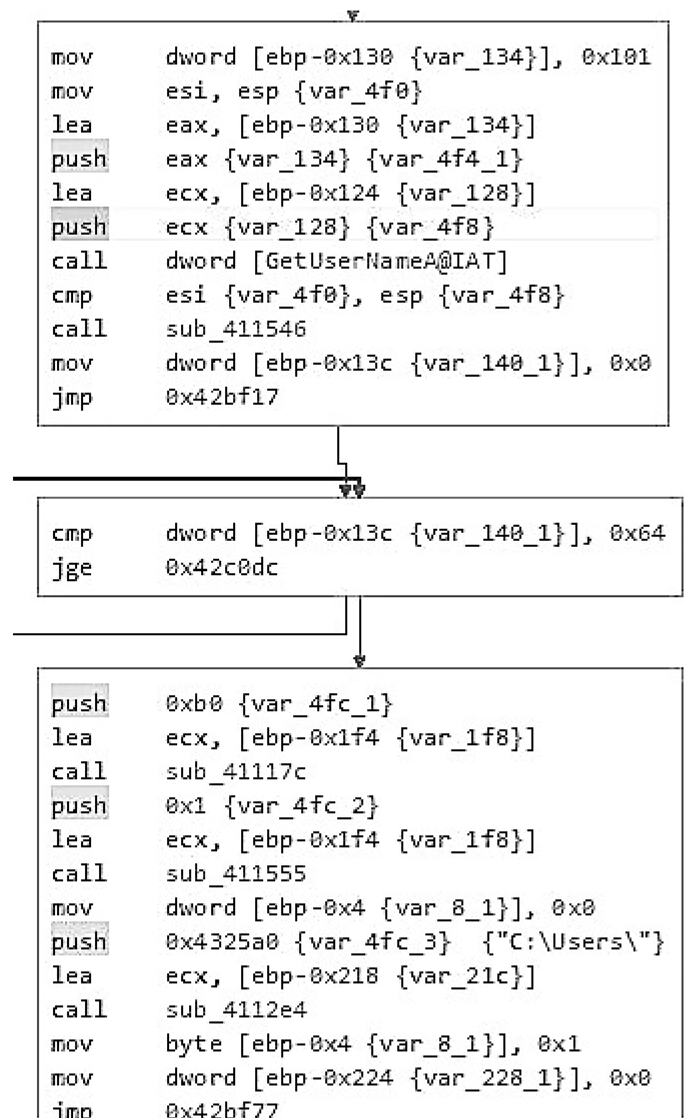


Figure 5.5: Screen clipping of code that would be executed if the letter ‘e’ was guessed.

The first statement places the value 101 in the address referenced by the value in ebp-130. This address is later dereferenced and its value placed in eax (not shown in image above), which is used as our counter. The comparison instruction *cmp dword [ebp-0x13]*, compares the value that is later placed in eax to 0x64h, which is 100; this is the comparison that controls the creation of the text files. If the counter is less than 100, another text file is created and placed on the user’s desktop. Notice that above the comparison, the instruction *call dword[GetUserName@IAT]* retrieves the username. The rest of the code displayed is involved in retrieving the path to the user’s desktop. At the bottom of the text file loop, eax is decremented, and its new value is stored in the address referenced to by ebp-0x013c, which is the same address that contains the value checked in the comparison for the loop. Figure 5.6 displays the set of instructions used to decrement eax and place

Copyright ©2020, American Society for Engineering Education 11

its value in the address at ebp-0x013c.

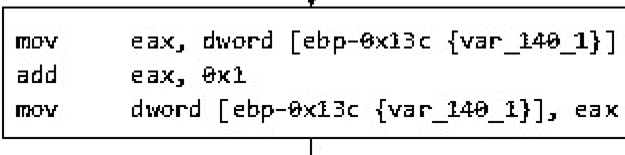


Figure 5.6: Screen clipping of instructions used to decrement eax and store the updated value.

5. *Delete error*

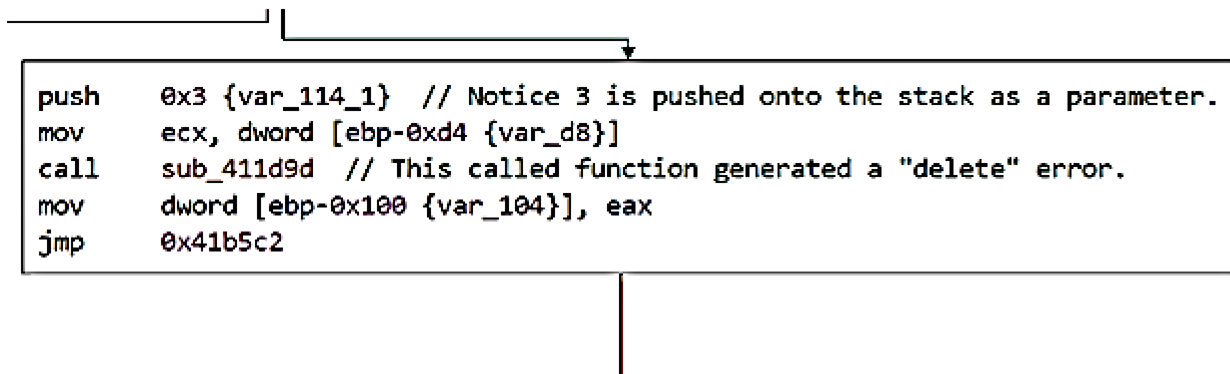
   Near the end of the program a “delete” error occurs. This error is generated in the fourth from the last called function (Figure 5.7). Stepping into another function within the outer function reveals code with two flows of execution. One flow of execution frees memory correctly, the other frees memory incorrectly. The original program code was designed to always flow through the code that frees memory incorrectly, generating a “delete” error (Figure 5.8). The code causing the flow of execution is an and statement with a result that is never zero followed by a jump statement that goes to the correct code only if the result of the ”and” statement is zero. Additionally, the ”and” statement is never zero because it “ands” the binary representation of 2 and 3. The result of which is always the binary representation of 2 (Figure 5.9). After analyzing the code producing a “delete” error, students determined that the code attempts to free memory it has not allocated. Because the program is trying to free memory it is not authorized to free, an error describing this behavior is thrown.

Copyright ©2020, American Society for Engineering Education 12

```
mov     dword [ebp-0x4 {var_8_2}], 0xffffffff  {0xffffffff}
lea     ecx, [ebp-0x2c {var_30}]
call    sub_411dac  // The function generates a "delete" error.
xor     eax, eax  {0x0}
push    edx {var_114_3}
mov     ecx, ebp {__saved_ebp}
push    eax {var_118}  {0x0}
lea     edx, [data_427c1c]
call    sub_4119a6
pop     eax {var_118}  {0x0}
pop     edx {var_114_3}
mov     ecx, dword [ebp-0xc {var_10}]
mov     dword [fs:0x0], ecx
pop     ecx {var_110}
pop     edi {var_10c}
pop     esi {var_108}
pop     ebx {__saved_ebx}  {0x0}
mov     ecx, dword [ebp-0x10 {var_14}]
xor     ecx, ebp {__saved_ebp}
call    sub_4119f6
add     esp, 0xfc
cmp     ebp {__saved_ebp}, esp {__saved_ebp}
call    sub_411546
mov     esp, ebp
pop     ebp {__saved_ebp}
retn     {__return_addr}
```

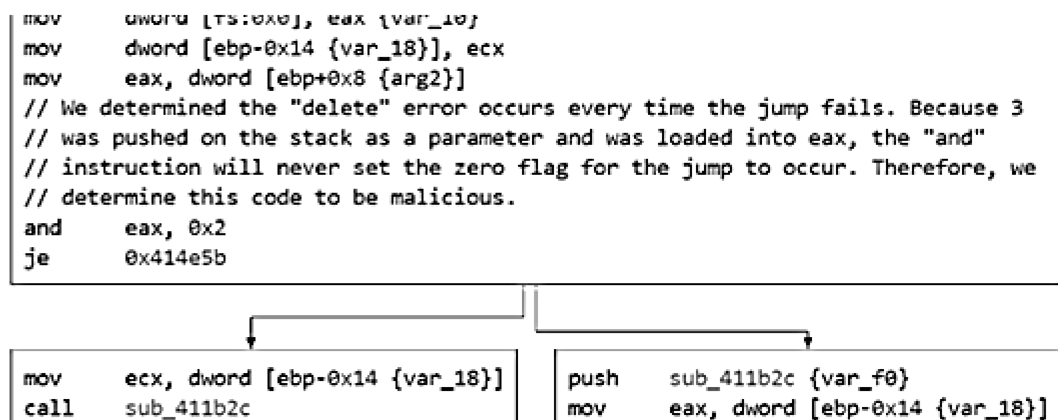
Figure 5.7: This image identifies the location of the function in main that generates a “delete” error.

Copyright ©2020, American Society for Engineering Education



```
push    0x3 {var_114_1}  // Notice 3 is pushed onto the stack as a parameter.
mov     ecx, dword [ebp-0xd4 {var_d8}]
call    sub_411d9d  // This called function generated a "delete" error.
mov     dword [ebp-0x100 {var_104}], eax
jmp     0x41b5c2
```

Figure 5.8: This image identifies the function where the “delete” error is generated. This image also displays the value 3 being passed to the function as a parameter, which is integral to the error being generated.



```
mov     dword [ebp-0x14 {var_18}], ecx
mov     eax, dword [ebp+0x8 {arg2}]
// We determined the "delete" error occurs every time the jump fails. Because 3
// was pushed on the stack as a parameter and was loaded into eax, the "and"
// instruction will never set the zero flag for the jump to occur. Therefore, we
// determine this code to be malicious.
and     eax, 0x2
je      0x414e5b

mov     ecx, dword [ebp-0x14 {var_18}]
call    sub_411b2c

push    sub_411b2c {var_f0}
mov     eax, dword [ebp-0x14 {var_18}]
```

Figure 5.9: This image displays the *and* statement that causes the program to always flow through the code that generates the “delete” error.

6. *Misspelled output*

   There were several misspellings in the original executable, so that the console would display messages such as “Unqessed letters” or “Enter you guess”.

6 Design: Modifications to malicious segments and functionality

There were several ways students tried to alter the code. Since BinaryNinja allowed inline editing in the graphical view with live disassembly, the most simple and preferable changes

Copyright ©2020, American Society for Engineering Education 14

to alter the program structure entailed editing existing instructions, such as turning a conditional jump into an unconditional jump. This prevented students from having to worry about overwriting surrounding object code that they did not wish to alter.

The students also explored altering the code by creating their own function and making a call to that function. In the executable, between the existing functions were segments that did not contain object code. In BinaryNinja, right-clicking on one of these segments would cause a drop-down menu to open, one of the options being 'Create Function Here'. Once the function was created, instructions could easily be entered by switching to the disassembly graph view, clicking on a specific instruction, and selecting 'Edit Current Line'. Then, students simply typed the instruction they wanted to insert in assembly language, and those instructions were automatically converted to object code. Once they had finished editing our function's contents, they could call the function from any part of the code.

The final way students explored editing the executable entailed figuring out the object code of the instructions they wished to introduce and editing the hexadecimal information of the object code directly. This method was used most often if they wanted to explore inserting only one or two lines of object code. They would figure out the encoding for the commands and locate exactly where in the executable file they wished to insert these commands. They would then shift all the bytes of object code that would follow our inserted commands down so that once they typed in their new commands, none of the previous data would be overwritten. Finally, they manually entered the new instructions. Regardless of the different approaches used, after any significant alteration or advancement was made, the other project partner was notified of the alteration and the new executable was made immediately available.

7 Implementation

This subsection contains a list of the problems they discovered (as described in the previous section) followed by a description their initial exploration of the executable revealed that it functioned as a hangman game with some malicious side effects, including stack usage errors and a large number of files being created and placed on the user's desktop. Of course, they aimed at further investigating any other malicious side effects and eliminating harmful behavior. Students also initially planned to alter the program so that a player would lose two guesses if they guessed a vowel not in the mystery word.

1. *Exhaustion of Heap*

   In order to nullify this behavior, students simply commented out the four instructions that performed the malicious actions. Figure 7.10 is an image of the modified executable.

Copyright ©2020, American Society for Engineering Education 15

```
nod_17.exe — Binary Ninja
ew  Tools  Help
Anom2718_mod_17.exe (PE Graph)
nt32_t __fastcall sub_426b60(int32_t* arg1, int32_t arg2 @ esi)
mov     eax, 0xcccccccc
rep stosd dword [edi]  {0x0}  {0xcccccccc}  {0xcccccccc}
pop     ecx {var_1b8}
mov     eax, dword [data_437014]
xor     eax, ebp {__saved_ebp}
mov     dword [ebp-0x10 {var_14}], eax
push    eax {var_1b8}
lea     eax, [ebp-0xc {var_10}]
mov     dword [fs:0x0], eax {var_10}
mov     dword [ebp-0x18 {var_1c}], ecx
// To solve the dynamically allocated memroy error we removed the code that
// caused the error. The "nop" statements correspond to four statements that
// were removed. The result of these modifications is that the program no longer
// has the error and the program functions properly.
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
mov     dword [ebp-0x11c {var_120}], eax {var_10}
mov     eax, dword [ebp-0x11c {var_120}]
mov     dword [ebp-0x24 {var_28}], eax
mov     esi, esp {var_1b8}
push    sub_411271 {var_1bc}
push    0x4324e4 {var_1c0}  {"Unguessed letters:"}
mov     eax, dword [std::cout@IAT]
push    eax {var_1c4}
call    sub_411bf4
add     esp, 0x8
mov     ecx, eax
call    dword [std::basic_ostream<char,...::char_traits<char> >::operator<<@IAT]
```

Figure 7.10: Screen clipping of code modified to prevent the heap exhaustion error.

2. *Lag time error*

   To nullify this behavior students replaced the call statement to this loop by moving -1 to eax, so that the same outcome would result without the time delay. Figure 7.11 is an image of our modified code.

Copyright ©2020, American Society for Engineering Education 16

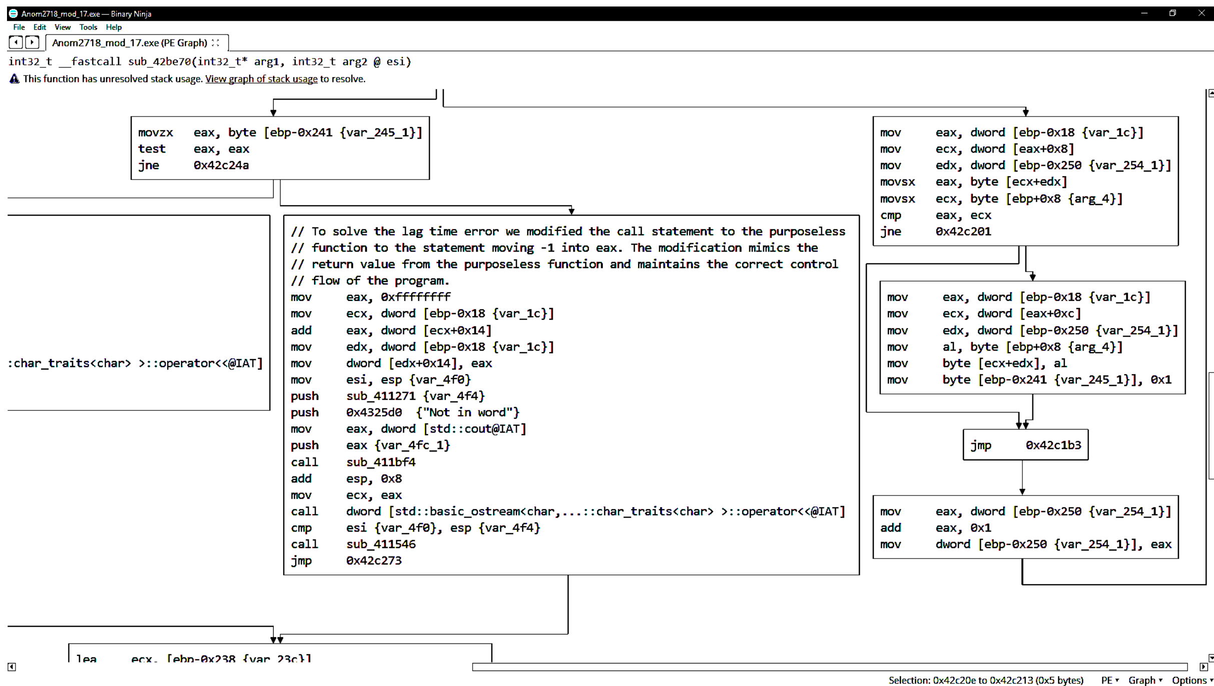


Figure 7.11: Screen clipping of the code modified so that the lag time error does not occur.

3. *Stack error upon guessing the letter 'a'*

   To eliminate this malicious behavior, they altered the comparison made, so instead of comparing the character in eax to 0x61 ('a'), it compares to 0x40 ('@'). Since the code before this function already checks that the character entered by the user is a lowercase alphabetic character, the jump will always be performed. Alternatively, the students could have removed the comparison or replaced the conditional jump with an unconditional jump to another section of the executable; however, they chose to make this alteration so that they could demonstrate different solutions to this problem. It is important to note that if solely this alteration was made, it would be possible that a stack error be generated if the code preceding this segment of the executable did not guard against the '@' symbol being entered as a valid guess. If the guard against special characters was removed and the user entered '@' then a stack error would result just as happened before the alteration when the character 'a' was entered. Figure 7.12 displays the executable with the students' alteration as well as their comments on the executable's behavior after they implemented this alteration but before they introduced any other alterations.

Copyright ©2020, American Society for Engineering Education 17

```
rep stosd dword [edi]  {0x0}  {0xcccccccc}  {0xcccccccc}
pop     ecx {var_4f0}
mov     eax, dword [data_437014]
xor     eax, ebp {__saved_ebp}
mov     dword [ebp-0x10 {var_14}], eax
push    eax {var_4f0}
lea     eax, [ebp-0xc {var_10}]
mov     dword [fs:0x0], eax {var_10}
mov     dword [ebp-0x18 {var_1c}], ecx
movsx   eax, byte [ebp+0x8 {arg_4}]
// Hypothesis: This is why we previously had an error thrown when we entered the
// letter 'a' as a guess.
// Test: change cmp to compare with ASCII symbol for '@' (hex value 40)
// Result: Correct! Changing this value to 40 prevents crashing upon entering
// the letter 'a' as a guess
// Misc. notes: After running it with this edit...
//    (1) the program crashed after 4 entries (UPDATE: this seems to be
// consistent behavior)
//    (2) it successfully updated the total number of guesses left until the 4th
// guess where that number did not update correctly (UPDATE: sometimes the
// function will update the total number of guesses correctly and sometimes it
// will not)
//    (3) UPDATE: it seems to be ok re-prompting for letters when they have been
// guessed before.
cmp     eax, 0x40
jmp     0x42c0dc
```

Figure 7.12: Screen clipping of code where the letter ‘a’ would trigger cause a stack error to be thrown.

4. *Malicious text file creation*

   The students resolved this problem by making an unconditional jump to the code following the portion of the executable that creates the text files. Figure 7.13 displays the executable flow after this change is implemented.

Copyright ©2020, American Society for Engineering Education 18

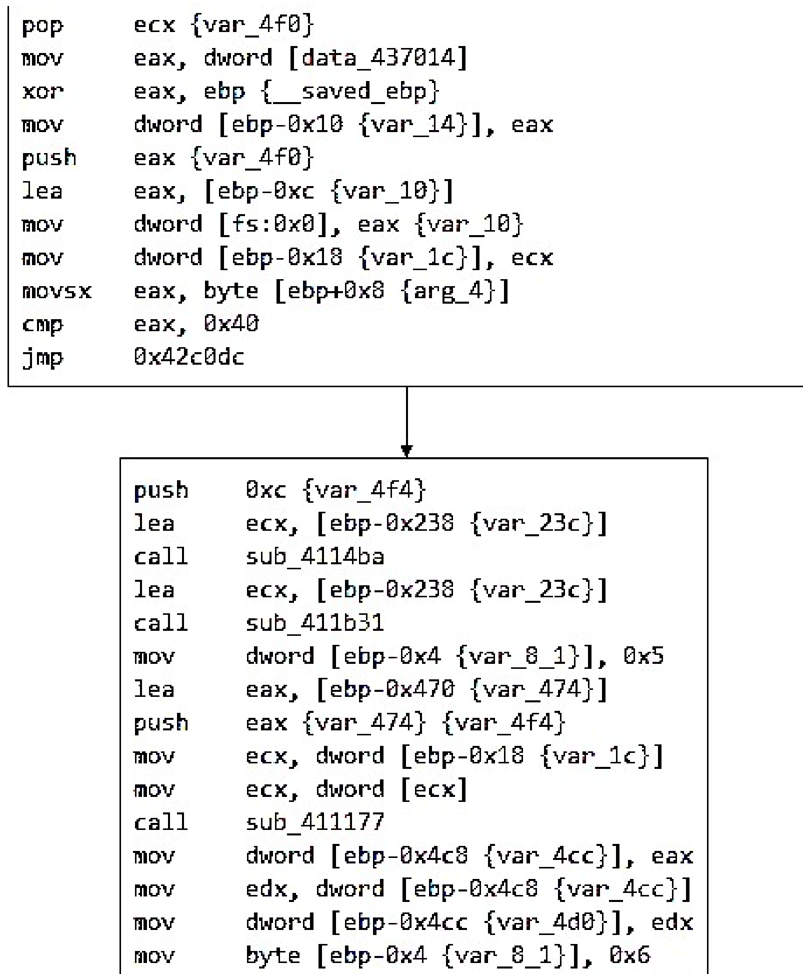


Figure 7.13: Screen clipping of program flow once the unconditional jump was implemented.

Significantly, the comparison, *cmp eax, 0x40*, no longer has an effect since an unconditional jump is made. Therefore, regardless of what character is guessed by the user, the displayed flow of the program will execute. As a result, notice that there is no longer a branch in the disassembly. The comparison instruction is retained simply to demonstrate that it has no effect.

5. *Delete error*

   To prevent the program from freeing memory it had not allocated, the students modified the ”and” statement so the result of the statement is always zero. They instruct the statement to “and” the value of 3 and 0, the result of which is always 0. This causes the subsequent jump statement to always go to the code that frees memory correctly. This modification ensures the malicious code is isolated and that memory is properly freed at the end of the program. Figure 7.14 displays the code after this modification.

Copyright ©2020, American Society for Engineering Education 19

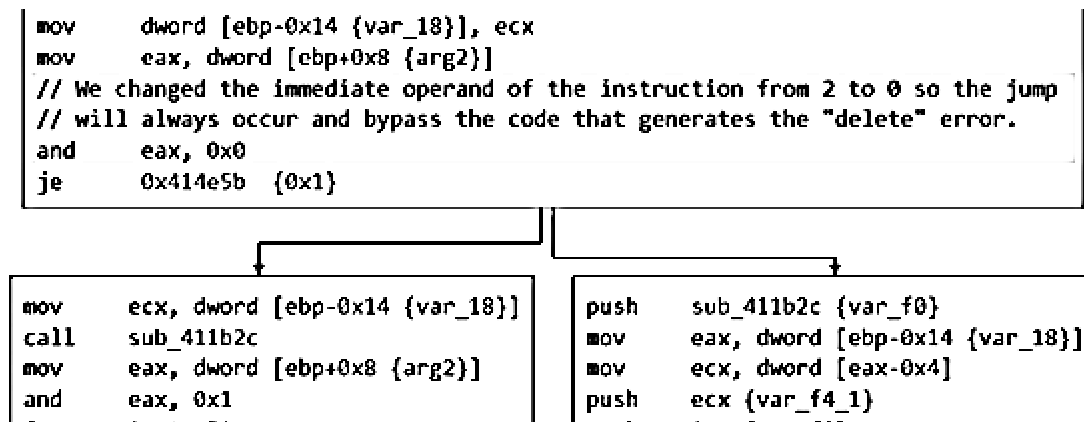


Figure 7.14: The *and* statement now results in the ZF being set which causes the program to always flow through the correct code.

6. *Misspelled output*

   Students fixed any intentional typos in the prompt calls.

After making these changes, the students re-considered their original plan for additional changes they wished to implement.

One of the ideas they considered was finding the word guess counter and modifying its behavior-either by changing the value of the counter or changing when it is decremented- so the player has an almost guaranteed chance of guessing the word. They determined that given the word typically ranges from 8 to 11 characters long, and the player already has 15 tries to guess the word. The counter is simply a value pushed onto the stack and then referenced repeatedly to decrement the counter as the user makes guesses. Students could repeatedly reset this counter to 15 (or another value) inside of one of the functions or eliminate the decrement of this value upon the user entering a guess, but they didn't want to eliminate the possibility of losing. Therefore, they decided to leave the counter as it was.

Another idea considered was to prevent the console window from closing immediately after the user either won or entered their last valid guess. They explored various ways of trying to accomplish this task, including trying to call system("push") and trying to prompt the user to press a key to signal they were finished looking at the result of the game. Due to the structure of the executable and the data in the executable, students were unable to implement this alteration. However, the exploration of possible implementations proved interesting.

The students tried three different methods to try to get the system to pause. The first was trying to make the function call *system("pause");*. This required placing the string "pause" somewhere in the executable, pushing the address of the first byte of this string onto the stack, and then calling the *system()* function with "pause" as its parameter. The students were able to place the string in the executable and push its address onto the stack. The difficulty came in trying to make the call to *system()*. They made a very simple program in Visual Studio that was a main function that only contained *system("pause")*. They then tried to use the disassembled

Copyright ©2020, American Society for Engineering Education 20

object code for the *system()* call to perform the call in the executable. However, this did not work because the object code created for the system call was created based off of values retrieved in a long series of previous function calls. Unfortunately, they could not figure out how to correctly call the correct library code from the executable.

The next method the students tried was trying to insert a set of instructions that would essentially print the message "Press a key to continue...", and then wait for the user to enter a key, which would be retrieved using cin. Again, they had no difficulty placing the string in memory or finding a spot to store the character captured by cin. They did struggle, however, in calling cin and cout to display the message and capture the character. Simply copying the object code from a previous segment of the executable would not work because the address to the system used by cin and cout will change from machine to machine, and thus cannot be hard coded, which is what results from simply copying the object code from previous cin and cout statements.

Finally, students tried to jump to the portion of the executable beneath "Enter your guess" that reads in the character entered by the user, and then jumping back to the end of the executable so the program exits once the user types another key stroke. The difficulty came in trying ensure that the executable would only skip to the end of the function if the final outcome of the game had finally been determined. They spent time exploring the possibility of storing a boolean value in memory in the executable that would be set only if the final outcome had been determined and then checking that value to determine if it should jump to the end of the executable or not. This idea seemed practical and may have been accomplished; however, there was not enough space between the instruction and function calls in the executable to insert the required instructions. Thus, this too proved a dead end.

There is one other option they considered briefly for causing the executable to pause. Students considered adding in a function that iterated over a loop a large number of times, which would cause the executable to lag for a brief amount of time before proceeding with the final instructions and closing the console window. However, they decided they didn't want to pursue this option because the performance across different machines would not be consistent. A slower machine would pause for a longer amount of time than a faster machine. They wished to find a solution that would have consistent performance across different platforms and hardware.

Finally, the students settled on implementing this additional change: they created an infinite loop that displays the text 'You Lost!' to the console window if the user is unsuccessful in guessing the word. There were several reasons why they decided that this was an additional change they wished to implement. One reason is that it serves as additional motivation for the user to correctly guess the randomly generated word. Another reason is that it allowed the students' to demonstrate their knowledge and understanding of how the original executable functioned while demonstrating how an executable could be modified to introduce malicious behavior. They introduced an unconditional jump that could only be reached if the player lost; this required understanding of both the original executable's flow and of how to edit the executable's contents.

Copyright ©2020, American Society for Engineering Education 21

8 Conclusions

For both teammates, this project was the first experience the students had reverse engineering a program from a binary executable. To succeed they had to discover what tools were available, what methods to use to understand the program, what new low-level instructions meant, how to alter the executable, and more. This learning process is filled with struggles and triumphs. Here are some of the lessons learned by the students that may be relevant for future reverse engineering projects.

1. *Find a decompiler.* For this project, students never utilized a decompiler to understand the code. A decompiler was not necessary because the complexity of this program was minimal. However, had the code been more involved, the higher level understanding that a decompiler provides would be necessary to analyze the program. Students did not find a suitable decompiler, but they did learn that a decompiler must be updated constantly to remain current with modern compilers and that there are plugins for decompilers that are compatible with common reverse engineering tools. The decompiler that is compatible with IDA Pro is a very attractive option although it would require purchasing the professional version of IDA.

2. *Use the debugger in IDA Demo.* If used intelligently, the debugger can significantly help students learn about how the unknown program functions. Additionally, by observing the registers, stack, and memory while debugging the program students can learn specifically how the program works. While stepping through the code, once errors are identified the debugger can help identify how the errors are caused and why they negatively impact the program.

3. *Writing and calling assembly in the executable is manageable, but calling library functions or any eternal code is not.* Students spent a combined 10 hours for this project trying to call code that allows the user to enter a character and still failed. To their knowledge the flow of code was set up correctly, they jumped to the code they wanted to call and then jumped back to the main program code when the call was complete, but the call to the cin method did not work. They tried to understand how the offsets worked and where the cin code was located in the file, but they could not get the code to execute. A deeper understanding of assembled code and executable files is required to understand the behavior of various library functions.

4. *Understand the program flow of the main program.* In the initial exploration phase of this project students spend several hours stepping into functions that were misleading rabbit trails. Once they were finally adept at identifying code that was relevant to the control flow of the main program, their time trying to understand insignificant code was minimized. This helped them streamline the process of understanding what kind of program they were analyzing and how to locate the errors.

5. *Document all knowledge gleaned about the mystery disassembly!* There are too many details, functions, and code in a disassembled executable file to remember everything.

Copyright ©2020, American Society for Engineering Education 22

Documenting all new knowledge gleaned from analysis is essential to minimizing time spent re-tracing over the same code again. Additionally, when analyzing code it is easy to work for several hours in a flow where one's short-term memory is well tuned to the program. However, upon returning to the program the next day, it is often difficult to recall those details. For this reason it is very important to continuously document the current state of analysis to save time.

6. *Students should start early and front-load the work.* The sooner students understand the disassembly of an executable they are trying to reverse engineer, the sooner they will make measurable progress. It is important to dedicate sufficient time upfront in their project timeline to get this important part of the project complete. All other aspects of the project depend on understand the code so having this analysis done will allow students to begin dividing up the subsequent work and make progress in a more efficient manner. For example, once the team grasped the main program control flow of the executable they were given, they divided up which errors each team member would correct and quickly worked to complete the given tasks.

9 Future Work

Reverse software engineering tools have become advanced enough that just a few layers of program obfuscation do not provide a sufficient challenge. With the release of the open-source NSA software Ghidra in April of 2019, reverse engineering has been simplified. Ghidra packages all the typical reverse engineering tools such as disassemblers, decompilers, binary editors, and more, allowing many students to rely on Ghidra alone to finish the class project. To encourage the use of more tools and more research, further layers of obfuscation that take advantage of the weaknesses of Ghidra will need to be developed into future class projects.

Malicious elements of future projects should be deeper and more complex. Using Ghidra and other modern reverse software engineering tools, students were able to find most of the malicious/obfuscating elements within the Fall 2019 project and quarantine them with simple binary code editing. In future iterations of the project, multiple malicious elements alongside obfuscation of those malicious elements will be necessary to provide a greater challenge to students.

Copyright ©2020, American Society for Engineering Education 23

10 References

[1] W. Gragido and J. Pirc, "Seven Commonalities of Subversive Multivector Threat," (Boston), 2011, 153 –175, doi: https://doi.org/10.1016/B978-1-59749-613-1.00009-1, http://www.sciencedirect.com/science/article/pii/B9781597496131000091.
[2] R. Brooks, S.B. Yun, and J. Deng, " Cyber-Physical Security of AutomotiveInformation Technology," ed. Sajal K. Das, Krishna Kant, and Nan Zhang (Boston: Morgan Kaufmann, 2012), 655–676, ISBN: 978-0-12-415815-3, doi: https://doi.org/10.1016/B978-0-12-415815-3.00026-1, http://www.sciencedirect.com/science/article/pii/B9780124158153000261; Alexandre Gazet et al.,Chapter 5 - Obfuscation(Wiley, 2014), 267 –340,ISBN: 978-1-118-78731-1.
[3] C. Cohen, "Semantic Code Analysis for Malware Code Deobfuscation," 2013, https://insights.sei.cmu.edu/sei blog/2013/07/semantic- code- analysis- for- malware- code- deobfuscation.html.
[4] M. Sewak, S.K. Sahay, and H. Rathor, "An investigation of a deep learning based malware-detection system," 2018.
[5] S. Chaki, "Using Machine Learning to Detect Malware Similarity," 2011, https://insights.sei.cmu.edu/sei blog/2011/09/using- machine- learning- to- detect- malware-similarity.html.
[6] D. Gavrilut et al., "Malware detection using machine learning," vol. 4 (November 2009), 735 –741, doi:10.1109/IMCSIT.2009.5352759.
[7] N. Peiravian and X. Zhu, "Machine Learning for Android Malware Detection Using Permission and APICalls," in2013 IEEE 25th International Conference on Tools with Artificial Intelligence(2013), 300–305.
[8] E. Laspe and J. Raber, "Deobfuscator: An Automated Approach to the Identification and Removal of Code Obfuscation", 2008, https://www.blackhat.com/presentations/bh-usa-08/Laspe Raber/BH US 08 Laspe Raber Deobfuscator.pdf.
[9] "Ghidra," National Security Agency, 2019, accessed April 28, 2019, https://www.nsa.gov/resources/everyone/ghidra/.
[10] C. Cimpanu, "NSA to release a free reverse engineering tool," 2019, https://www.zdnet.com/article/nsa-to-release-a-free-reverse-engineering-tool/.
[11] "Binary Ninja: Home," Binary Ninja, 2019, accessed March 25, 2019, https://binary.ninja.

Copyright ©2020, American Society for Engineering Education 24