x86 Software Reverse-Engineering, Cracking, and Counter-Measures

Lab: Obfuscation

Environment Needed:

- Linux Virtual Machine (Ubuntu Recommended) Machine

Introduction

- This lab explores a variety of common obfuscation techniques, their effectiveness, and how to defeat them. Several different challenges are presented, but, as always, the primary goal is to solve the first; the rest are provided if you have extra time, or want to explore in more depth outside of class. If you have time for the bonuses, challenges I and V are the critical exercises; however, you should read the instructions for each other challenge in order.
- Each of the challenges implements a different simple key check algorithm that has been obfuscated with an off-the-shelf obfuscating compiler. Each challenge takes a key of the format XXXX-XXXXX-XXXX, and passes it as an array of digits to a check() function. check() returns true if the key check passes, and false if it fails.
- These challenges are best tackled with GDB and a text editor (of your choice). We'll use GDB to step through the code and better understand complex flows. In the text editor, we'll paste the program assembly, and make notes on control flow, obfuscations, dead code, and program simplifications.
- As we get more advanced with GDB, it helps to have a more assistive interface. This can be accomplished through a variety of plugins and initialization scripts. For this lab, we'll use the popular "GDB Dashboard" init script (https://github.com/cyrus-and/gdb-dashboard). Copy the file .gdbinit from the lab directory into your home directory (cp .gdbinit ~/).

= Part I - Substitution Obfuscations =

This challenge is a simple key check function, compiled with the popular Obfuscator LLVM compiler with the substitution obfuscation flag. Recall that substitution obfuscations transform a simple expression into a more complex, harder to read version.

1. Locate the file check_sub
2. Use objdump to dump the program assembly. Copy the assembly into a text editor of your choice, and save the file. Use the text editor to take notes on the code as you reverse engineer it. Some helpful approaches are:
3. Add blank lines to separate logically related lines of assembly into "blocks".

X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES
ISBN: **1394199880**

4. When you find and reverse engineer an obfuscated block of assembly, add comments to the top of the block to indicate what it does.
5. When you find dead code or bogus control flow (code that doesn't do anything, jumps that are never taken), indent them to remind yourself that they are not relevant.
6. Find the check() function in your disassembly file. Begin your initial reverse engineering. What are the input parameters? What are the local variables? What control flow constructs are being used? What condition is being checked at the end of the function?
7. Hint: check() is passed an array. Arrays in x86 are commonly accessed with [base + index * size], where base (a register) is the address of the array, index (another register) is the index of the element we want to access, and siz (a constant) is the size of each element of the array. In check(), this looks like [ecx + edx * 1] - ecx holds the array address, edx the array index, and each element of the array is a 1 byte digit.
8. Look for sequential assembly instructions that can be rewritten in a simpler form. These are the obfuscated transformations in the program. As a first example, the nearby "xor 0xffffffff" instructions stand out as something unusual. See if you can determine what this block of code does. Either use GDB to step through and watch the input values (ecx, eax) and output values (esi), or use pen and paper to evaluate the code. Once solved, make sure to annotate the code in your text document to remember what it does.
9. Find and simplify the other substitution obfuscations (there are two more).
10. With the obfuscations simplified, reverse engineer the overall key check.
11. Using pen and paper or a programming language of your choice, derive a valid key for the program.

= Bonus Challenge I - Bogus Control Flow =

This challenge is a simple key check function, compiled with the popular Obfuscator LLVM compiler with the bogus control flow obfuscation flag. Obfuscator LLVM's bogus control flow generates opaque predicates to make the control flow of the program harder to derive. Recall that opaque predicates are branches that always take the same path, but which make it difficult to determine which path will be taken.

1. Locate the file check_bcf
2. Begin the RE process as you did in the first challenge. Use objdump to get the program disassembly, and copy the assembly into a text editor. Save this file.
3. Unlike instruction substitution, opaque predicates (bogus control flow) do not replace existing code, but rather insert dead code (code that will never execute) and jump instructions that will always take the same path. When reverse engineering this program, use your text editor to indent code blocks that are never executed, and conditional blocks that always resolve to the same value. This will help keep track of what code is relevant, and what is not.

4. Carefully examine the first conditional jump in the check() function. What are the conditions that lead up to this jump? What values are being used in the multiplication? What do those values tell you about the result? Without gdb or dynamic analysis, can you determine which direction this jump with take?
5. Once you determine which direction the jump will take, use your text editor to mark unnecessary code. Indent code that is unused, and code that results in pre-determined jumps.
6. Repeat this process throughout the check() function. Isolate only the minimum necessary code for the function to execute.
7. Reverse engineer the essential code to derive the original key check function.
8. Using pen and paper or a programming language of your choice, derive a valid key for the program.

= Bonus Challenge II - Code Flattening =

This challenge is a simple key check function, compiled with the popular Obfuscator LLVM compiler with the code flattening obfuscation flag. Recall that code flattening transforms the normal control flow of a program into a switch statement.

1. Locate the file check_fla
2. Begin the RE process as you did in the first challenge. Use objdump to get the program disassembly, and copy the assembly into a text editor. Save this file.
3. Find the switch statement construct in the program. Which variable is being "switched"?
4. Find the basic blocks of the switch statement. Mark each block with a unique identifier. Decipher the logic of each block. At the end of each block, what change occurs to the switch variable?
5. Using pen and paper, reconstruct the overall control flow of the program; refer the overall flow back to the identifier of each block.
6. Reverse engineer the overall code to derive the original key check function.
7. Using pen and paper or a programming language of your choice, derive a valid key for the program.

= Bonus Challenge III - All of the Above =

This challenge is a simple key check function, compiled with the popular Obfuscator LLVM compiler with _all_ obfuscation flags.

1. Locate the file check_all
2. Begin the RE process as you did in the previous challenges. Use objdump to get the program disassembly, and copy the assembly into a text editor. Save this file.
3. Use the techniques from the previous challenges to decipher the algorithm in challenge III.

4. Using pen and paper or a programming language of your choice, derive a valid key for the program.

= Bonus Challenge IV - Hard Mode =

This challenge is a simple key check function, compiled with the popular Obfuscator LLVM compiler with all obfuscation flags turned up to a maximum. The 6 line C key checker took 5 minutes and 7 gigabytes of memory to compile.

1. Locate the file check_hard_mode
2. At over a million lines of assembly, this program is not solveable by hand. It requires automated scripts to reduce/remove the instruction substitution, bogus control flow, and code flattening employed by the obfuscating compiler.
3. Using your knowledge of instruction substitution, bogus control flow, code flattening, and the O-LLVM compiler, recover the original key check algorithm.
4. Derive a valid key for the program.

= Bonus Challenge V - Something Different =

A wide variety of very diverse obfuscating compilers exist. This challenge uses the M/o/Vfuscator compiler, which compiles C programs to only x86 mov instructions. The original C code remains a simple key check function.

5. Locate the file moving
6. Many methods of reverse engineering M/o/Vfuscated code exist. Some options are:
7. Open source tools that recover the control flow of M/o/Vfuscated programs.
8. Examining data access paterns of the mov instructions. For example, even without understanding the underlying logic, if the mov instructions load the byte values 'm' 'y' 'k' 'e' 'y' from memory, this reveals valuable information about the underlying logic.
9. Full reverse engineering of the instruction stream to recover the original program. This approach was first pursued by a professor of malware research at a large American university, but took hundreds of hours.
10. In some cases of reverse engineering, alternative approaches are necessary. Consider the common C function: "strcmp". Using your own insights or the internet, what weaknesses might this function have? How would you explore these weaknesses on a program? Could an obfuscated program be susceptible to such weaknesses?
11. Instead of reverse engineering the challenge, find an alternative attack on the program to derive a valid key.
12. Which of the previous challenges does this attack work on?