Breaking Eggs And Making Omelettes: Intelligence Gathering For Open Source Software Development

Mike Melanson (mike@multimedia.cx)

There is an ongoing information war raging on in the software world. Despite free software developers' best efforts, new proprietary software continues to proliferate. New techniques must be developed to efficiently reverse engineer closed data formats so that free, interoperable solutions can be deployed under Linux. This paper focuses on intelligence-gathering for developing software algorithms, regardless of their difficulty or perceived legality.

Introduction

There is an ongoing information war raging on in the software world. Despite free software developers' best efforts, new proprietary software continues to proliferate. New techniques must be developed to efficiently reverse engineer closed data formats so that free, interoperable solutions can be deployed under Linux.

Software reverse engineering occurs on various levels. It may be necessary to study a piece of poorly-written, poorly-commented code developed in a high-level language such as C++ and understand what the original program was supposed to accomplish. It may also be necessary to disassemble a program that has been compiled into machine language and express it as a higher-level language so that the underlying algorithms can eventually be expressed as higher-level concepts in a human language. After obtaining an algorithmic description via reverse engineering, the algorithm can be reimplemented for any language on any computing platform.

Scope

This paper discusses the technical issues and challenges surrounding software reverse engineering. This has always been the subject of much legal and ethical controversy, particularly with respect to such intellectual property ideas such as patents, trade secrets, and software ownership. However, such discussion is outside the scope of this paper.

Motivation

The Linux operating system, as well as free and open source software in general, has made extraordinary strides in the computer world in the past decade. Once confined to the back office server, Linux has become a more viable option for desktop computing. However, in order to create a desktop (and in some respects, a server), that is acceptable to much of the world, the computer must be able to interoperate with existing computers and the dominant software that runs on them.

This has traditionally been a substantial problem in the multimedia arena. Many popular multimedia files available on the internet are encoded with proprietary software algorithms that commercial companies do not share.

Thus, it often becomes necessary to reverse engineer the algorithms in order to use them on an open source Linux desktop.

Alternatives To Binary Reverse Engineering

It is important to view binary software reverse engineering as strictly a last resort. It is not an easy task and it is certainly not something that a programmer should want to do if there might be an alternative. Given that the end goal of reverse engineering tends to be understanding data formats and data manipulation algorithms, a programmer should search far and wide to learn if the reverse engineering target is already available via other means.

Perhaps the algorithm was opened up by the creating company. In some cases, another reverse engineer has taken an interest in the same algorithm, figured out the code, and released it as open source. Sometimes, that same programmer has figured out the code but opted not to release the source code, only the compiled binary, but for no specific reason. Upon reasoned request, such programmers may be willing to make the source code available.

If all other methods are fruitless then it may be time to employ some reverse engineering techniques.

Requirements

There are some minimum requirements that an individual should meet if they endeavor to reverse engineer software.

- A Goal-- Of course, the reverse engineer should have a specific goal in mind.
- Time-- A reverse engineer should have plenty of time to devote to the craft. Preferably, the time should be in large continuous blocks (better for concentration).
- Patience-- Anything that takes a lot of time will also require a lot of patience.
- Tenacity-- A reverse engineer must be willing to stick to the goal to the very end.
- Tools-- A well-stocked toolbox is essential, as is knowledge in their proper use.
- Programming knowledge-- A software reverse engineer should at least know C programming and have exposure to assembly language. Generally, Intel

x86 assembly language is important as much proprietary software is targeted at the platform. Note that it is adequate to have a beginning familiarity with the assembly language that the target is composed in; the study and practice of reverse engineering tends to increase a programmer's assembly aptitude many times over.

- Domain knowledge-- When reverse engineering a multimedia algorithm, for example, it is useful for the reverse engineer to have some exposure to multimedia coding theory in order to understand what the target might look like.
- Capacity to believe the unbelievable and accept the unacceptable-- A reverse engineer will undoubtedly encounter many odd situations and methods of solving problems, and will need to be able to conceptualize how other programmers might have solved similar problems.
- Creative thinking-- Reverse engineering is a very open field with few standard methods of operation. A reverse engineer will often need to think of innovative methods for understanding software.

Reverse Engineering Without Program Disassembly

Disassembling a closed binary program may not be entirely necessary in order to discover an algorithm or format. If there are alternatives to disassembling a binary program and tracing through instructions, those alternatives are often preferable.

One common problem area for reverse engineering is file formats. For example, in the multimedia arena, encoded audio and video data are packed into some kind of file format. Many multimedia file formats have similar properties and patterns. A programmer with adequate domain knowledge in multimedia file formats can often examine sample files a hex editor and figure out most or all the information necessary to take apart the whole file. Typically, no binary reverse engineering is necessary.

Another problem area is network communications protocols. For this type of reverse engineering, a network protocol analyzer (a.k.a. sniffer) is an indespensible tool. Occasionally, binary disassembly may be required in order to discover the finer points of a network protocol like the way to unravel some lightweight transport encryption. Generally, by analyzing patterns and sequences in subsequent data packets, and understanding what type of data payload the packets are already carrying, much intelligence can be gathered without binary disassembly.

Example Disassembly

When it becomes inevitable that a binary program will have to be torn apart in order to discover an algorithm, the first usual step is to disassemble the binary into assembly language instructions. In the case of a program compiled for the Intel i386 platform, a disassembly will yield thousands or even hundreds of thousands of lines that resemble these:

```
1000 8B442404                   mov eax, dword[esp+04]
1004 85C0                       test eax, eax
1006 740D                       je 1015
1008 8B08                       mov ecx, dword[eax]
100A 83F93C                     cmp ecx, 0000003C
100D 7506                       jne 1015
100F B801000000                 mov eax, 00000001
1014 C3                         ret
1015 C70500900090040000000      mov dword[nnnn9000], 00000004
101F 33C0                       xor eax, eax
1021 C3                         ret
```

The first column shows program addresses. The second column lists machine opcodes. The remainder of a line is the assembly language instruction mnemonic. Note that although Intel i386 family processors use a 32-bit address space, the top 16 bits of these addresses have been omitted for simplicity.

Basic Reverse Engineering

How does a reverse engineer interpret assembly code such as the code presented in the previous section? This section will discuss and illustrate a few basic techniques.

Breaking Down The Problem

As in any complicated problem, it is important to break down the problem into manageable pieces. A reverse engineer should never attempt to understand the entire program as a whole. It is more advantageous to view the program as most programmers are trained: as a series of smaller pieces which typically represent inidividual functions.

A good disassembler should be able to use various heuristics to determine and visually indicate function boundaries. In the example presented in the previous section, the disassembler correctly marked the sequence of lines as a single function. Assembly language instructions that indicate a return from a subroutine call are often good clues.

Context Analysis

Examination of the context in which a function is called often yields useful intelligence, particularly regarding a function's parameter list and return type. A typical invocation of the example function above looks like this:

```
1031 8B742408                   mov esi, dword[esp+08]
1035 56                         push esi
1036 E8C5FFFFFF                 call nnnn1000
```

```
103B 83C404                  add esp, 00000004
103E 85C0                    test eax, eax
1040 7405                    je nnnn1047
```

Since the standard C convention is to pass parameters to a function by pushing them on the stack in right to left order (right-most parameter is pushed first, left-most parameter is pushed last), So the calling context reveals that the function has one parameter (only one data item is pushed onto the stack). Further, the context reveals that the function returns a value (via the eax register) that is acted upon (code branch if eax is zero).

Context analysis also applies to any extra knowledge that a binary program module provides. Very few programs exist in a vacuum; in other words, many programs depend on shared libraries in order to operate. A compiled program that relies on external libraries must carry information about which modules must be loaded by the operating system and which function calls must be located inside the various modules. It is very difficult for the compiled program to obscure the fact that it makes calls out to these external library functions. For example, this statement was disassembled from the same module as the preceding assembly language instructions:

```
53BA FF15FCA00090            call dword[nnnnA0FC]
                            ;;call KERNEL32.ReadFile
```

The disassembler informs the reader that the the program calls out to the standard Windows 32-bit API function called ReadFile() to read data from the disk. Consulting any Windows API guide reveals that this function takes 5 parameters and returns a Boolean value. Further, it defines what all of the parameters and return value indicate which can be used to determine the meanings of more variables.

Call Trees

Provided that a high-level language program is not written as one large function littered with GOTO statements, it is a fair assumption that many programs can be represented by call trees. A C program begins with a main() function. A non-trivial C program will call other functions and those functions will likely call other functions. For example:

```
main()
  +-function1()
    +-function2()
    +-function2()
  +-function3()
```

A call tree is another way to view the reverse engineering problem. In this simple example, main() calls function1() and function3(). Function1() calls function2() twice. In reverse engineering, it is sometimes useful to identify leaf functions that are called repeatedly (like function2()) and understand them first. Doing so can help with context analysis. As a practical example, many multimedia audio and video coding algorithms pack data as bit streams rather than byte streams. Many audio and video decoders need to read variable amounts of bits from the bitstream. So a programmer will often write a small bit reading function and call it many times throughout a decoder. Understanding this early on in the reverse engineering process can be useful.

There are exceptions to every rule. It is always possible for a program to use some clever construct where two functions call each other. This is a corner case that a reverse engineer may need to deal with eventually.

Reverse Engineering By Hand

Reverse engineering by hand is a phenomenally tedious chore. This is where the reverse engineer examines the disassembled code and pieces together original algorithms by understanding the operation of each individual line. To apply the practice to the example code fragment used in this paper:

```
' move the function parameter from the stack into register eax
1000 mov eax, dword[esp+04]
' logically AND the function parameter in eax with itself; do
' not store the result but modify the CPU flags
1004 test eax, eax
' jump if equal (the CPU's zero flag is set) to address 1015
1006 je 1015
' move the 32-bit doubleword pointed at by eax into register ecx
1008 mov ecx, dword[eax]
' compare (subtract) 0x3C from value in ecx; do not store
' difference but modify the CPU flags
100A cmp ecx, 0000003C
' jump if not equal (the CPU's zero flag is clear) to address 1015
100D jne 1015
' move the constant value 1 into the eax register
100F mov eax, 00000001
' return to the calling function
1014 ret
' move the constant value 4 into a global 32-bit variable
' indicated by [nnnn9000]
1015 mov dword[nnnn9000], 00000004
' logically eXclusive OR the eax register with itself
' this has the net effect of setting the register to 0
101F xor eax, eax
' return to calling function
1021 ret
```

Observe that from the very beginning of the function, the comments take advantage of substitution. After the function parameter is moved into the eax register at address 1000, it is safe to express that the function parameter, and not just eax, is being tested at address 1004. Since the address inside eax is dereferenced at address 1008, it is also safe to assume that the lone function parameter is a memory pointer. The function has two paths of returning to the calling function (addresses 1014 and 1021) and it explicitly sets eax prior to leaving. In Intel i386 programs, it is customary to return function values through the eax register. Thus, the function takes a pointer parameter and returns a 32-bit integer:

```
unsigned int functionA(unsigned char *pointer);
```

And the algorithm inside the function can be expressed as:

```
{
  if (pointer == NULL) {
    /* set a global variable at address 0xnnnn9000 to 4 */
    return 0;
  }

  if (*((unsigned int*)pointer) != 0x3C) {
    /* set a global variable at address 0xnnnn9000 to 4 */
    return 0;
  }

  return 1;
}
```

To sum up this function's algorithm, make sure the pointer parameter is non-NULL and that the first 4 bytes that it points to (in little-endian/Intel format) are 0x0000003C. If these two conditions are true, return 1. If either of these conditions are false, set a global variable at address 0xnnnn9000 to 4 and return 0.

Now that the general purpose of the function has been determined (checking the first four bytes of a buffer), the function can be renamed everywhere it is called in the disassembly which will help in context analysis.

Tracing

A traditional method of reverse engineering is tracing through code, The basic idea is to use a code debugger to step through code, line-by-line, understanding the operation of an algorithm by watching the sequence of instructions executing, the code branches taken, and the data moved in and out of registers. A step debugger that can set breakpoints can also help in finding interesting portions of code in the first place.

By itself, however, tracing can turn into an even more tedious chore than disassembly by hand.

Tools For Reverse Engineering

Clearly, reverse engineering even the simplest function from binary software is a non-trivial task. Anyone who spends a serious amount of time reverse engineering binary software will notice that the whole process is tedious, timeconsuming, boring, and highly error-prone. Fortunately, tasks that fit those criteria have traditionally been prime candidates for automation.

Boomerang

Boomerang is an open source project that works to ease the pain of reverse engineering by searching for patterns in binary code and replacing them with equivalent C constructs. It uses a series of algorithms that convert assembly language instructions to C code and then make automatic substitutions throughout. Ideally, all the is left for the reverse engineer is to rename variable and function identifiers. Boomerang can also accept a set of hints that specify the names of known data structures so that the program can automatically replace those names as they are seen in the decompiled code.

Boomerang is an interesting and developing projeect. It is not perfect as software methods can only do so much without human intervention. In particular, case-switch constructs have long been the bane of software revere engineers. Due to the way such constructs intermingle code and data in a compiled program, it is a very difficult problem to automatically determine how the original case-switch construct was laid out in the original source code.

See the references for more information regarding the Boomerang project.

IDA PRO

IDA Pro is a famous tool designed for reverse engineering code. Heavily geared toward the IT security specialist, the IDA Pro features an impressive variety of tools to lighten to burden of reverse engineering such as variable naming substitution: when it becomes evident that [esp+04] represents value *foo*, IDA Pro can rename instances of that variable; call trees: IDA Pro can track call tree structures; control flow: IDA Pro has facilities for graphing control flow such as if-else constructs.

IDA Pro is a proprietary, closed source tool. However, the creating company, DataRescue, makes a free Windows version available which runs under Wine in Linux. Recently, DataRescue has also made a native Linux version available.

See the references for more information regarding the IDA Pro product.

Ad-Hoc Tools

Many novice reverse engineers seem to develop grand plans to create a full-featured, generalized reverse engineering tool. But due to the highly experimental nature of reverse engineering, it is often useful to develop creative new ideas and test them carefully in a limited environment rather than trying to find a magical approach to reverse engineering all software.

This section explores other possibly useful reverse engineering techniques and tools.

Execution Profiling

Code profiling is used during software development to understand where a program spends the majority of its execution time. The goal of collecting such execution data is to discover places where the code ought to be optimized.

In reverse engineering, profiling can provide useful intelligence to understand where a program spends its time. Combined with domain knowledge regarding the specific application, a reverse engineer can gain an understanding of specific areas of code that contain the most important algorithms.

However, it is rather difficult to use standard profiling tools on binary programs that lack any debugging information. It is possible to bypass this limitation by creating special tools to monitor a program's execution activity.

See the references for more information about this type of tool.

Automated Call Tree Generation

Many disassemblers to a very good job of automatically determining function boundaries when outputting textual disassembly. Using programming languages that are reputed to be very good at manipulating text (Perl, for example), it is possible to create tools that automatically generate call tree data from raw disassembly listings.

See the references for more information about this type of tool.

Reverse Engineering Other Languages

Not all computer software is written in C and C++ and subsequently compiled into machine language for distribution and execution. Unix operating systems such as Linux have a rich tradition of programming languages. Programs written for many of these languages, such as Perl and Python, are distributed as source code and interpreted and/or compiled at run time. Rarely are they compiled into binary modules for distribution.

However, recent years have seen efforts such as Sun's Java and Microsoft's dot-NET framework as a method of compiling source code into machine-independent bytecode for distribution. Indeed, this is sometimes vaunted as a selling point for such languages.

In Java's case, however, it is ironically possible to decompile a compiled bytecode program file (called a class file in Java parlance) and obtain the complete source code with all control structures and identifier names intact. Such a disassembly will even include intact case-switch code constructs. Any comments in the original file will have been stripped away during compilation and will not be available in the decompilation.

Since a Java class file retains so much information about the original source code, a market has sprung up for Java source code obfuscators. These programs operate by looking through Java source code and replacing identifiers (such as function and variable names) with random, unintuitive, or confusing names. However, there is little that such obfuscators can do to make the actual source code harder to decompile. A Java decompiler will still be able to interpret all of the original code structures simply because the Java run-time engine needs to be able to do the same to run a Java program.

Granted, the obfuscation can make decompiled code more difficult to interpret. But the same techniques and programs that are used to obfuscate the code in the first place can be subverted to partially deobfuscate the same code. In this process, the unintuitive identifiers are replaced with random, but more logical words. For example, replace obfuscated variable names with random animal names and obfuscated method names with random verbs. Ideally, the list of animal names and verbs used should be selected from the reverse engineer's native language. The logic behind this method is that it should be easier, psychologically, to understand source code with better identifiers.

For example, a Java source file may have a class constructor with the following declaration:

```
MainClassConstructor(int width, int height);
```

After being run through a code obfuscator, the declaration may appear as:

```
_mthelse(int aQ, int v);
```

After deobfuscation, the method name will be replaced with a random verb and the function parameters will be random nouns:

```
streamline(int sparrow, int bullDog);
```

Such names actually are easier to deal with than the obfuscated equivalent, particularly since they are easier to search and replace on a global text file level.

Conclusion

This paper has presented a rather broad overview of the equally broad topic of software reverse engineering. The topic is still open to much research and much potential improvement. It is something about which more programmers need to think seriously and talk openly. There is still a whole world of proprietary software algorithms and data structures at large that have yet to be understood.

References

- Breaking Eggs And Making Omelettes Blog: http://multimedia.cx/eggs/

  This weblog tracks various experiments related to the art of software reverse engineering as well as its application towards multimedia techology.

- Practical Reverse Engineering: http://multimedia.cx/pre/

  This page features various reverse engineering experiments such as automated call tree generation, execution profiling, and partial automated

Java de-obfuscation.

- Boomerang: http://boomerang.sourceforge.net/

  Web site for the Boomerang project.
- IDA Pro: http://www.datarescue.com/

  Web site for the IDA Pro product.