Next-Generation Protection Against Reverse Engineering

4 February 2005

Chris Coakley, Jay Freeman, Robert Dick

ccoakley@anacapasciences.com; saurik@saurik.com; radick@anacapasciences.com

Purpose

This white paper describes a next-generation, software-protection technique that prevents certain classes of automated reverse engineering tools[1] from successfully attacking compiled software to expose underlying code. This paper:

- Presents the issues
- Describes the new protection technique
- Proves the technique in a critical test
- Provides all proof-of-concept code
- Identifies a strategy to extend this protection to national-interest software

Audience

This paper is written for software protection managers, as well as software development professionals. Managers may want to review the background sections below before examining the proof-of-concept description and the strategy for protecting national-interest software. Software professionals may want to scan the proof-of-concept description, then examine the code provided on-line at http://www.anacapasciences.com/projects/reverseengineering/index.html.

The Software Protection Arms Race

Provably, the arms race between software protection and software attack (reverse engineering) cannot be won by either side. The race is a never-ending spiral, first favoring one side and then the other. In the arena of automated reverse engineering, the attack side has been moving into the favored position, due to the success of emergent, next-generation tools called “de-obfuscating disassemblers”. Our next-generation protection technique, however, neutralizes these tools and moves protection back into a favored position.

History

Software engineers originally developed the first reverse engineering tools to help automate the process of debugging their software. These tools – called “disassemblers” – are still an indispensable part of modern software engineering.

As disassemblers became more powerful, however, people began using them to automatically reverse engineer software developed by other parties. The motivation for exposing underlying code is often to steal intellectual property, circumvent anti-piracy techniques, steal information, or compromise essential systems.

In an effort to neutralize the threat from illegitimate reverse engineering, software engineers developed techniques to obfuscate (hide) code before it was shipped to customers. They still relied, of course, on disassemblers to debug non-obfuscated versions of their software.

The early obfuscation techniques took advantage of certain ***accidental*** weaknesses of legacy disassemblers. These early protection techniques only worked against the generation of disassemblers that contained the accidental weaknesses.

---

[1] This paper focuses on protection against automated disassemblers that can rapidly reverse engineer large sections of program logic. We do not address protection against emulators, which involve a manual, labor-intensive process only practical for reverse engineering small sections of code.

Research by academics and by software pirates ultimately yielded new approaches to building disassemblers that did not have these accidental weakness. Although the research motivations for the two groups differed greatly, both avenues have led to very similar kinds of new reverse engineering tools, the de-obfuscating disassemblers.

Status

The new disassemblers have had remarkable success against current protection techniques [1]. That success threatens the security of many types of essential software.

Fortunately, however, academic researchers have recently identified an avenue for neutralizing emerging next-generation disassemblers [1]. This avenue takes advantage of certain ***inherent*** weaknesses in all approaches to automated reverse engineering, both new and old. The new approach can be used to create "heavy obfuscators" that resist a broad range of disassemblers, including those of the next generation.

The Software Protection Battleground

Threat Model

We are concerned here about threats in which:

- A hostile party may acquire the executable (binary) code for an important, complex application.
- The hostile party must correctly disassemble large portions of the binary to meet his objectives.
- The application is sufficiently complex that using an emulator and extensive human-software interactive techniques to disassemble the binary are impractical.
- The performance requirements of the application are sufficiently stringent that heavy encryption of the binary is impractical.

The protection objective, therefore, is to heavily obfuscate the binary so it is impervious to automatic disassembly, but without incurring a large reduction in performance.

Software Architectural Context

The adage, "Choose your battles," is true in software protection - some battles cannot be won. For example, reverse engineering is fundamentally unstoppable on certain modern computer architectures. These architectures include PowerPC, UltraSPARC, MIPS, and other architectures that use Fixed Length Instructions (FLI). We do not deal with these architectures.

The battleground of interest is protecting against automated reverse engineering of software for architectures that use Variable Length Instructions (VLI). These architectures are used by Intel, AMD, and Transmeta. On these VLI architectures, reverse engineering is, in fact, stoppable. Because a good deal of national-interest software runs on these architectures, notably Intel's, this battleground is important.

The context for the battle involves the method that VLI architectures use to encode binary files, summarized here:

- Computer instructions occupy between 1 and 8 bytes.
- Instructions follow one after another with no padding.
- Each encoding of instruction sequences has a unique starting point for correct decoding.
- All instruction sequences in a program are in serial order with arbitrary padding between them.
- The padding can be any length and may contain any content (including program data), or contain nothing.
- The padding content may be "garbage" designed to look like data or instructions.

The strategy for attack has these characteristics:

- All disassemblers begin with a binary file and attempt to decode the file to end with human-readable code.
- Whether the human-readable code is correct or not is dependent upon decoding the file from correct begin- and end-points for each of the many sequences of variable length instructions in the binary file.
- The inherent difficulty is that the begin- and end-points can only be identified correctly by analyzing code *after* it has been decoded.
- Disassemblers partially resolve this difficulty by systematically guessing the begin- and end-points.
- The newly emerging disassemblers have become efficient and powerful in homing in on correct guesses.

The strategy for protection relates to some of the same characteristics:

- The only begin-point that a disassembler tool knows with certainty is the first begin-point.
- All end-points (and all other begin-points) must be determined by code analysis.
- While guessing these points can be made efficient in emergent disassemblers, deriving absolute knowledge about these points through some algorithm is a fundamentally unsolvable problem
- *Until now, no one on the protection side has exploited the fundamentally unsolvable attack problem to neutralize correct guessing by disassemblers.*

More About Reverse Engineering Methods

Before describing the new protection strategy, it's helpful to present a little more background information about current and emerging attack strategies.

Disassemblers work by automatically guessing where to begin decoding, using one of the techniques below. Hopefully (from the attacker's viewpoint) the guessing leads to human-readable code that makes sense with minimum additional manual analysis.

Linear Sweep

Linear sweep starts at the beginning of the software binary file and decodes instructions in sequence until it generates an error. Then it tries to find the next nearest place to re-start decoding. This technique is very fast, but it does not account for the fact that jumps in code can land at arbitrary locations. With jump instructions, the next nearest place to begin decoding is not always the best.

Recursive Descent

Recursive descent is "control flow aware." As with linear sweep, recursive descent disassemblers will start at the beginning of a file. However, if a jump statement is encountered, the disassembler will stop decoding at the current location and re-start decoding from the jump target location. The difficulty with this approach is that some jump targets are computed based on data input. The inability to guess the jump target when analyzing a static binary (without data processing for computing jump points) results in large segments of code that remain encoded (and obfuscated).

Hybrid Approaches

Hybrid approaches first use recursive descent to decode as much of a binary file as possible. This leaves some sections of the binary file still encoded. Then the hybrid approach switches to linear sweep for the sections that were still encoded. The linear sweep helps identify new code blocks that were missed during the recursive descent. After completing the linear sweep, the hybrid approach then switches back to recursive descent mode. This alternating process continues until it fails to identify new blocks of code.

Next-generation de-obfuscating disassemblers use these hybrid approaches, and they are becoming extremely effective against current-generation protection techniques.

Next-Generation, Broad-Spectrum Protection

Emergence of the new breed of de-obfuscating disassemblers has spawned research in methods to protect against them. We have examined published and unpublished research on ways to exploit inherent weaknesses in both legacy disassemblers and the new de-obfuscating disassemblers. We've adapted and combined selected exploitations to create a powerful hybrid exploitation that compiles heavily obfuscated C binaries.

Before presenting our heavy-obfuscation compiler, we'll summarize the exploitive techniques we employ. Each technique prevents an attacking disassembler from producing human-readable code.

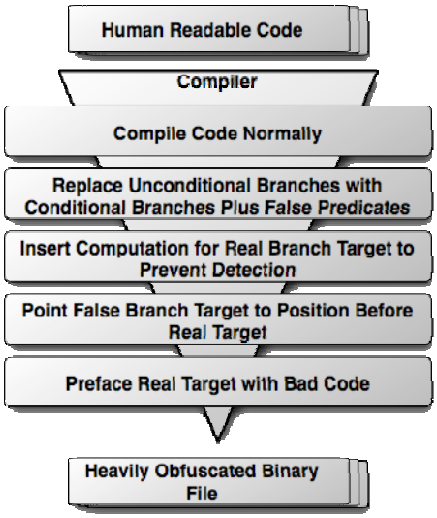
- Branch Point Obfuscation – In this technique, the target of a branch statement is prefaced with a one byte jump statement. This preface causes the next four bytes (the valid instructions) to be erroneously decoded as a new jump target. An attacking disassembler would not be able to identify the four bytes of valid instructions.
- Computed Branch Target – In this technique, the target address of a branch statement is obfuscated by requiring that the address value be computed. The required computation can be arbitrarily difficult – as protection engineers, we would always choose a level of difficulty that exceeds current and emerging disassembler capabilities. When the attacking disassembler miscalculates the address value, it will incorrectly identify the next code segment's starting location. This incorrect identification will result in decoding invalid code as if it were valid.
- False predicates – In this technique, the normal function of a conditional branching statement is subverted. All non-conditional branches are replaced with conditional branches, which require a disassembler to examine both branches. Each of our new conditional branches are coupled to a false predicate. A false predicate always yields the same result, meaning that one side of the conditional branch is always taken, making the conditional branch, in effect, a non-conditional branch. This causes additional, but unfruitful work for an attacking disassembler. Whatever lies on the non-taken branch, be it data or padding, will be decoded as if it were valid instructions.
- Combined defense – This technique combines elements of the above techniques to mutually strengthen each other. For example, the second branch of a false predicated conditional branch is combined with branch point obfuscation. We make the branch always taken require a computed branch target. We make the second, not taken, half of the branch point to the special one byte jump statement inserted using Branch Point Obfuscation. To an attacking disassembler, this makes the invalid code indistinguishable from valid code and provides no incentive (and an additional disincentive) to find the valid code.

Proof-of-Concept, Next-Generation Obfuscation Compiler

We used all of the exploitation techniques above in developing a proof-of-concept, next-generation, heavy-obfuscation compiler that protects binaries against both legacy and emergent disassemblers.

Approach

We modified a freely available C compiler – called tcc [3] – to produce heavily obfuscated executable binaries. We used tcc because it is very small and particularly easy to modify. Our new obfuscation technology, however, will work with any compiler that targets a VLI machine.



The figure to the right shows the compilation steps produced by the modified version of tcc. The source code for the

modified tcc is available for download at
http://www.anacapasciences.com/projects/reverseengineering/index.html

Test

Testing involved the following steps:

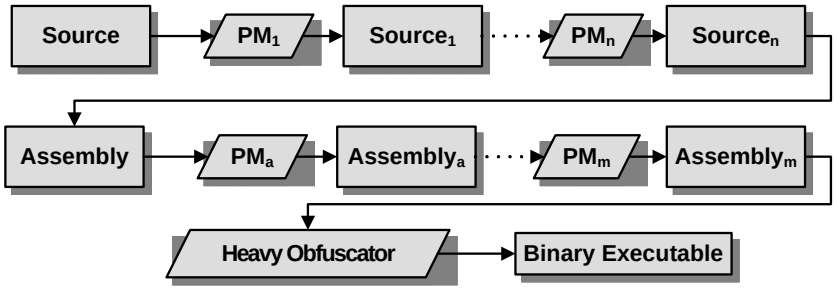
1. We developed three examples of program code, representing increasing levels of complexity.
2. We ran the three programs through an ordinary compiler.
3. We used a current-technology disassembler – objdump – to reverse engineer the programs. Objdump was successful in reverse engineering the simplest program, partially successful on the moderately complex program, and unsuccessful on the most complex program.
4. We also obtained a copy of a next-generation, de-obfuscating disassembler from the University of California at Santa Barbara (UCSB). This disassembler currently holds the record in Linn and Debray's benchmark test of disassembler effectiveness [2]. UCSB's disassembler was successful in reverse engineering all three programs.
5. We then ran the three test programs through the modified tcc compiler to obfuscate them heavily.
6. All three of the heavily-obfuscated programs completely resisted reverse engineering by objdump and UCSB's disassembler. The disassemblers could not even determine the main program entry points for any of the obfuscated programs.

Proof

A complete listing of the code for each of the three programs, the correct assembly, and each of the disassembler outputs are available for downloaded and examination at http://www.anacapasciences.com/projects/reverseengineering/index.html.

Application

Our heavy obfuscation technology can be used in conjunction with other protection techniques to achieve broad-spectrum protection. If multiple protection measures (PMs) are used inline, our heavy obfuscator is inserted at the last point in the chain, prior to generation of the binary, as shown in the figure below.



Strategy for Protecting National-Interest Software

We have developed and demonstrated new, broad-spectrum, heavy-obfuscation technology for protecting software binaries for Intel x86 machines, which run much of the national-interest software. Currently, the new technology is in the form of a heavy-obfuscation compiler based on tcc, a small compiler that is convenient for proof-of-concept demonstrations.

Primary Thrusts

To be broadly useful, however, the new principles of heavy obfuscation should be adapted to develop at least three different sets of developer tools to protect software on a wide front:

- A heavy-obfuscation version of gcc, which is used to compile nearly 100% of all Linux executables. The advantages of gcc over tcc are the existence of multiple front ends

(Fortran, C++, and other languages) and gcc's renowned code optimizers. Creating a heavy-obfuscation version of gcc will involve a scaled-up, somewhat-modified version of the process we used to create the heavy-obfuscation version of tcc.

- A heavy obfuscator tool set to protect Microsoft Windows binary files. Creating this tool set is a multi-step process that employs a mixture of off-the-shelf compilers and custom-written assemblers and linkers. The resulting heavy obfuscator tools will protect binaries produced from C, C++, ADA, and Fortran. In fact, the resulting heavy obfuscator tools will enable protection for any existing compiler that can produce x86 assembly output.
- Heavy-obfuscation tools for protecting other VLI architectures, especially the ARM processors used in embedded, mission critical systems. Creation of these tools is more complicated due to the fact that other architectures will require different instruction selection and substitution criteria from the x86 architecture this research focuses on.

Recommendations

We recommend two courses of action to protect national-interest software from next-generation, already-powerful, de-obfuscating disassemblers:

1. Because a significant segment of national-interest software involves Microsoft Windows binary files running on x86 machines, we recommend the immediate development of a heavy-obfuscation tool set for this domain. A detailed workplan for producing an initial version of this tool set is available from the authors on request.

2. Because a growing segment of national-interest software will likely involve Linux executables over the next few years, we recommend development of a heavy obfuscation version of gcc, starting within the next six months. A detailed workplan for producing a preliminary version of this tool is also available on request.

Acknowledgements

We would like to thank Professor Giovanni Vigna and Dr. Chris Kruegel of UCSB for lending us the code to their de-obfuscating disassembler. Without their cooperation, the stringent test of our new protection technology would not have been possible.

References

[1] C. Kruegel, W. Robertson, F. Valeur, G. Vigna, "Static Disassembly of Obfuscated Binaries", to appear in Proceedings of the 13th USENIX Security Symposium, San Diego, CA, August 2004.

[2] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly", Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS), pages 290-299, Washington, DC, October 2003.

[3] Tiny C Compiler homepage: http://fabrice.bellard.free.fr/tcc/

