

Deobfuscator: an Automated Approach to the Identification and Removal of Code Obfuscation

Jason Raber and Eric Laspe
Riverside Research Institute
softwaresecurity@rri-usa.org

Abstract

The Deobfuscator is an IDA Pro plug-in that neutralizes anti-disassembly code and transforms obfuscated code to simplified code in the actual binary.

This plug-in is used in conjunction with a binary injector to remove obfuscated code and replace it with a simplified, transformed equivalent. We developed this tool in assessing strengths of protections and malware analysis for DoD government entities and commercial companies.

1. Introduction

Malware authors and owners of proprietary software algorithms often use code obfuscation techniques to hinder users from gaining understanding about the integral parts of their applications. Simple instruction sequences are obscured, control flow is disorganized, and unnecessary instructions are introduced to confuse disassembly tools, and the reverse engineer.

The Deobfuscator combines instruction emulation and pattern recognition to determine code control flow, interpret the intended results of obfuscated code, and transform instruction sequences to enhance the readability of code where all states are known.

2. Recognition & Transformation

The Deobfuscator processes a user-defined address range and automatically steps through instructions either sequentially (by address) or logically (following jumps and calls) depending on the selected operation mode.

Pattern recognition is triggered by reading a specific type of instruction. Each pattern is sequentially compared with each instruction examined by the Deobfuscator. Every pattern consists of simple predicates that determine if the most recently examined instruction matches its next expected instruction type

or value(s). If the instruction sequence is of the expected form, the pattern is matched and the function generates the byte code necessary to simplify the instructions. If the pattern's expected instruction sequence does not match the types or value(s) read by the Deobfuscator, a comparison will fail and the next pattern will be called until a pattern is matched or the list of patterns is exhausted.

The plug-in can recognize numerous obfuscation patterns as it steps through an address range. Most of the patterns are generic in nature and are not limited to simple peep-hole optimizations (e.g. the Deobfuscator tracks register liveness and emulates the stack). Figure 1 shows an example of an instruction sequence recognized by the Deobfuscator under the *push_pop_math* pattern. The *push* and *pop* instructions are recognized by their types and can be located several instructions—even basic blocks—apart. The math portion of the pattern recognizes instructions from a predefined set of math instructions and can consist of any number of those operations on a given register. The plug-in will continue to read instructions sequentially until the destination register is stored to memory or a control flow break (*call*, *ret* or conditional jump) occurs—the plug-in does, however, track patterns through unconditional jumps. Figure 2 shows the transformed result of the sequence. It is easier to see in the “deobfuscated” disassembly that the value being stored in *edx* may be used as an address. Additionally, upon opening the new binary, IDA Pro's analysis adds this instruction to the list of data cross-references to the address 0x40107B, which aids further static analysis of the disassembly. Should an indirect jump to *edx* follow this sequence, a second run of the Deobfuscator would resolve the jump to that location.

00401064	push	0E39A3CC0h
00401069	pop	edx
0040106A	xor	edx, 0E3DA2C8Bh

Figure 1: Before “deobfuscation”

00401064	mov	edx, offset byte_40107B
00401069	nop	
0040106A	nop	
0040106B	nop	
0040106C	nop	
0040106D	nop	
0040106E	nop	
0040106F	nop	

Figure 2: After “deobfuscation”

3. Operation

The Deobfuscator plug-in is run from IDA Pro using a hotkey. The user specifies an address range within the disassembly, and a “deobfuscation mode.” The patterns the Deobfuscator searches for are dependent on the mode chosen by the user. The five modes available are:

- Anti-disassembly – replaces code that causes disassembler errors
- Passive – simple peep-hole optimizations
- Aggressive – uses aggressive assumptions about memory contents; tracks registers/stack
- Ultra – uses more aggressive assumptions
- Remove NOPs – adjusts control flow

Once a pattern is identified, the Deobfuscator generates two text files named “jmp.txt” and “math.txt.” The contents of the files look similar to the output in Figure 3. Each line of the text file has, in this order: a 4-digit (hex) file offset; the (decimal) number of opcode bytes in the current line; and the (hex) opcode bytes to be written, at the specified offset, to the binary.



Figure 3: Output text file

Next, the user runs the binary injector “deob_patch.pl” which overwrites the binary at the file offset locations with the new code. Finally, the binary can be reloaded in IDA Pro and the Deobfuscator rerun to find additional patterns. Some complex obfuscated instruction sequences can be resolved in multiple runs.

4. Usefulness & Limitations

The Deobfuscator can currently replace over 44 different obfuscation patterns with simplified code that improves disassembly and human-readability. The Deobfuscator can resolve: many forms of anti-disassembly such as jump chains, push-returns, call-returns, return folds, jump indirects, jumps into instructions; several types of move and stack manipulation obfuscations, which try to mask the flow

of data; and unnecessary operations having no net effect. In its “aggressive” and “ultra” modes, the Deobfuscator tracks single or multiple register liveness, respectively, and can replace “dead code” with *nop* instructions.

The Deobfuscator was designed and tested using empirical analyses of malware and commercial protections we have encountered in our professional activity. It has reduced the time required for removing obfuscation from hours to minutes, and increased the accuracy of those transformations.

While most obfuscation removal is straightforward using static analysis techniques [1], there are limitations to what types of obfuscation can be removed automatically—even by hybrid static-dynamic analysis techniques [2]. Code that employs dynamic pointer structures, for example, cannot be simplified using the Deobfuscator since it does not make assumptions about values that cannot be tracked statically. The most aggressive assumptions for reducing code complexity are based on instructions where immediates and static memory structures are introduced. This approach has proven effective in removing most obfuscation.

6. Conclusion

We created the Deobfuscator using obfuscation patterns we observed in widely available software. Its effectiveness has been tested throughout development and has aided our understanding of several real-world software applications. The tool’s simple structure makes implementing new patterns and features straightforward. We believe the Deobfuscator will aid reverse engineers by transforming anti-disassembly and other obfuscations into code that is more human-readable.

7. References

[1] Udupa, Debray, and Madou, “12th. IEEE Working Conference on Reverse Engineering, Pittsburgh, PA, November 2005, pp. 45-54.

[2] Madou, Anckaert, De Sutter, and De Bosschere, “Hybrid Static-Dynamic Attacks Against Software Protection Mechanisms,” 5th ACM Workshop on Digital Rights Management, Alexandria, VA, November 2005, pp 75-82.

[3] Aho, Sethi, and Ullman, *Compilers – Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1985.