Code (De)Obfuscation

Matias Madou*, Bertrand Anckaert*,
Koen De Bosschere*,[1],

* *ELIS, Ghent University, Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium*

ABSTRACT

Measuring the security of code obfuscation is difficult. A novel obfuscation transformation is in some cases only measured in terms of code expansion and speed, which are in fact only side effects of the transformation. A first step to define a security value to an obfuscation transformation could be having a look at what a cracker is able to reveal from an obfuscated program. This abstract first of all gives a short overview of existing techniques to obfuscate. Then, we describe existing techniques that can be used to deobfuscate, which were sometimes originally meant for other purposes, and new techniques which we are working on to deobfuscate.

KEYWORDS: obfuscation; reverse engineering

1 Introduction

Reverse engineering techniques are mostly used to make program improvements such as platform optimizations, bug-fixes, or virus removal. Unfortunately, these techniques are also used for unlawful purposes, for example searching for vulnerabilities in binaries and finding malware, which makes legal protection of the program void. Protection against exploiting vulnerabilities, unauthorized access or protection of the intellectual properties justifies making specific binaries hard to reverse engineer. For example, a company wants to protect their unique algorithm from a competing company.

Technical protection of a program can be done in a few ways. Technical protection by using special hardware, such as the palladium is a well known technique [LTM+00]. The use of the palladium is effective but software written for this piece of hardware is no longer executable on any machine. Another solution is code obfuscation which is a program transformation technique that makes a transformed program harder to understand and/or difficult to reverse engineer while maintaining its functionality.

The goal of code obfuscation is making it harder for a cracker to understand the program. Therefore measuring code obfuscation should be in terms of security. As side effects code expansion and speed could be measured. Nevertheless, most code obfuscation transformations are measured only in terms of code expansion and slowdown. There is also no entire understanding of what is more difficult for a cracker. The only attempt of an obfuscation taxonomy is done by Collberg *et al.*[CTL97] but is inefficient.

[1]E-mail: {mmadou,banckaer,kdb}@elis.UGent.be

A new approach to have some idea of the security of an obfuscated program is having a look at the cracker's side. What does the cracker need to deobfuscate the program? By deobfuscation is meant the transformation from the obfuscated program to a program that is "similar to" the original program.

2 Existing techniques

In this section, we describe shortly the existing obfuscation transformations and applications.

The most popular control flow obfuscation technique is described in the dissertation of Chenxi Wang[Wan00] and is called control flow flattening. The idea is that all basic blocks of a function appear to have the same set of predecessors and successors. This obfuscation transformation takes a central part in an industrial obfuscation tool by Cloakware Inc.[CGJZ01]. It is a technique which is not only used for obfuscation, but also for watermarking [WS04, Clo03].

Linn and Debray [LD03] introduce branch functions and apply other control flow transformations with the aim of thwarting a static disassembly of executable code. Branch functions are functions that do not return to the caller, but by use of the return-address pushed on the stack a new address is computed. In particular, they try thwarting the advanced disassembly heuristics discussed by Schwarz et al. [BS02].

ObjObf[2], created by Team-Teso, is the only x86 object obfuscator. The obfuscation transformations implemented in this tool are swapping of instructions, splitting basic blocks, insertion of junk instructions, and entangle basic blocks. Hackers used a program obfuscated with this tool to breach a Debian server and make it as hard as possible for the administrator to have any idea what the malware was doing[3].

Wroblewski [Wro02] used instruction reordering, block reordering, exchange of fragments and insertion of code to obfuscate the code. This technique is very simple, and hence not very powerful. It offers the advantage of not depending on any specific property of the target computer architecture.

Another well-known technique for obfuscation [CTL98] (and also for watermarking [Arb02]) is the use of opaque predicates in programs. An opaque predicate has a property that is known at obfuscation time, but is hard to know afterwards. This property can be used to guide the execution of a program.

The first transformation using self-modifying code to obfuscate the code is described by Kanzaki *et al.*[KMNiM03]. In this paper the novel idea of using self-modifying code as obfuscation technique is quite interesting, although the obfuscation transformation itself is not very powerful.

While these transformations look quite useful and natural, all share the same principal shortcoming: a lack of any theoretical foundations for realistic attack models which guarantee their obfuscating effectiveness.

---

[2]Object-Obfuscator x86/Linux ELF, http://www.team-teso.net/projects/objobf/

[3]http://www.theregister.co.uk/2003/12/02/hackers_used_unpatched_server/

3 Deobfuscation

An evaluation of three variants of the flattening algorithm is done by us in cooperation with Sharath Udupah and Saumya Debray and is described in a draft[UDM05]. By use of static and dynamic methods we tried to deobfuscate a flattened binary. The results indicate that it is not too hard to deobfuscate this transformation. Although, Wang claims that the problem is NP-hard for a simple attack model.

To go around the obfuscation technique proposed by Linn and Debray to thwart the static disassembly, can be done by a new disassembly technique proposed by Kruegel *et al.*[KRVV04]. This technique is able to reveal about 97% of the original instructions. Another possibility to crack the disassembly obfuscation transformation is by use of dynamic tracing and static rewriting. We are working on this total new approach and expect some good results. The idea is to start with a dynamic trace and locate the executed parts in the static program. We are sure about these parts and now the idea is to extend these parts by use of static analyses. The advantage is of this approach is that the disassembly process starts with known parts. Our technique is able to work in cooperation with Kruegel's disassembly technique.

As ObjObf obfuscates object files, it is easy to test the implemented transformations by use of a link-time optimization tool, such as Diablo[4]. Amazingly, Diablo is able to deobfuscate nearly the whole program. The parts that were not deobfuscated are on the one hand due to bugs in ObjObf itself and on the other hand, parts were really obfuscated. When going into details, only duplication of basic blocks and making them different seems to be interesting but the original technique should be changed to be robust against optimization attacks.

A good obfuscation transformation is one that even if you know every detail about the transformation, it is still impossible to remove the obfuscation from the program. With opaque predicates, the database containing the predicates is seen as the key of the transformation. Keeping secrets is not the way obfuscation works and this technique can be seen as security through obscurity. Of course, when the database containing the opaque predicates is known, we are able to remove all opaque predicates statically.

Using simple self-modifying code, such as proposed by Kanzaki *et al.*, as an obfuscation technique is useful if the cracker has no experience with cracking software. It is true that a static attack will not be able to reveal the code, but that is always the case if self-modifying code is used. Using a dynamic attack won't have any problems with the obfuscation. Even worse, using the technique reveals more about the program than the program without obfuscation, because the transformation can only be inserted in places where three conditions hold. These conditions are easy to determine at obfuscation time, but difficult ones the program is written out. When attacking the program and finding such obfuscation transformations, the cracker is sure that the conditions hold, which he couldn't be sure about if he was trying to reverse engineer the original binary.

References

[Arb02] Geneviere Arboit. A method for watermarking java programs via opaque predicates, October 2002.

[4]http://www.elis.ugent.be/diablo

[BS02] Gregory Andrews Benjamin Schwarz, Saumya K. Debray. Disassembly of executable code revisited, 2002.

[CGJZ01] S. Chow, Y. Gu, H. Johnson, and V. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. *In G. Davida and Y. Frankel, editors, Information Security*, ISC 2001, volume 2200 of Lectures Notes in Computer Science (LNCS):144– 155 Springer–Verlag, 2001. 68, 2001.

[Clo03] Introduction to the cloakware/transcoder, version 2.0. Cloakware Corporation, July 2003.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, The University of Auckland, New Zealand, 1997.

[CTL98] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, pages 184–196, 1998.

[KMNiM03] Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken ichi Matsumoto. Exploiting self-modification mechanism for program protection. In *COMPSAC '03: Proceedings of the 27th Annual International Conference on Computer Software and Applications*, page 170, Washington, DC, USA, 2003. IEEE Computer Society.

[KRVV04] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of USENIX Security 2004*, pages 255–270, San Diego, CA, August 2004.

[LD03] Cullen Linn and Saumya K. Debray. Obfuscation of executable code to improve resistance to static disassembly. *Proc. 10th. ACM Conference on Computer and Communications Security (CCS 2003)*, Oct 2003.

[LTM+00] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.

[UDM05] Sharath Udupah, Saumya Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code, May 2005. http://www.cs.arizona.edu/ debray/papers/unflatten.ps.

[Wan00] Chenxi Wang. *A Security Architecture for Survivability Mechanisms*. PhD thesis, Department of Computer Science, University of Virginia, October 2000.

[Wro02] G. Wroblewski. *General Method of Program Code Obfuscation*. PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.

[WS04] Kelce Steven Wilson and Jason Dean Sattler. Software control flow watermarking, Aug 2004. Baker and Botts, US2005/0055312 A1.