Understanding Obfuscated Code

Matias Madou, Ludo Van Put and Koen De Bosschere
Ghent University
St Pietersnieuwstraat 41
9000 Ghent
{mmadou,lvanput,kdb}@elis.ugent.be

Abstract

Code obfuscation makes it harder for a security analyst to understand the malicious payload of a program. In most cases an analyst needs to study the program at the machine code level, with little or no extra information available, apart from his experience. An unexperienced analyst is confronted with a steep learning curve, as understanding unobfuscated machine code already requires some skills. We have built LOCO, *a graphical, interactive environment to help a security analyst improving his skills in understanding obfuscated code.*

1. Introduction

Code obfuscation can serve two goals. On the one hand, it can be applied to protect a company's technology from being copied. On the other hand, obfuscation can be used to hide malicious code in a program. For example, an exploit to hack a Debian server was obfuscated, to make it difficult for a security analyst to find the malicious code [1]. In this demonstration we present Loco; a tool to speed up the learning process of a security analyst to understand obfuscated code. The tool can apply a series of obfuscation transformations to a program, after which it shows a graphical presentation of the obfuscated program. A security analyst can make use of well-known analyses such as dominator analysis, liveness analysis and other predefined analyses included in the tool to help him deobfuscate the code.

LOCO is an extension of the graphical user interface LANCET[8], combined with an obfuscation infrastructure in the underlying link-time program rewriter DIABLO[4][2], which allows us to do fine-grained code obfuscation[7]. LOCO is freely available from the DIABLO website.

The remainder of this tool presentation is organized as follows. Section 2 presents the underlying structure of LOCO, one obfuscation transformation and the running example. Section 3 describes the deobfuscation features. Section 4 gives a brief overview of the demo. In Section 5 we discuss the limitations of our tool and give directions for future work. Conclusions are drawn in Section 6.

[1]http://www.theregister.co.uk/2003/12/02/hackers_used_unpatched_server/
[2]http://www.elis.ugent.be/diablo

2 Obfuscation transformations

DIABLO is a multi-platform link-time binary rewriting framework from which we will use the x86 backend and ELF object file format. LANCET, a graphical user interface on top of Diablo, can visualize the call graph of a program and control flow graphs (CFGs) of the procedures and lets a user easily navigate through them. Furthermore, LANCET provides means to edit the graphs. LOCO not only consists of a library with obfuscation transformations, but contains also features for deobfuscation.

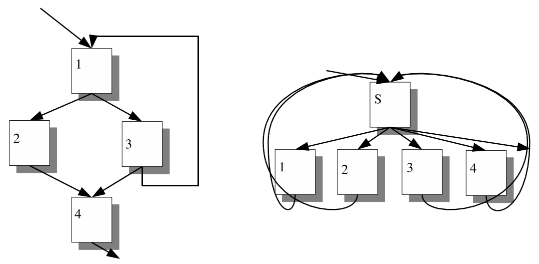
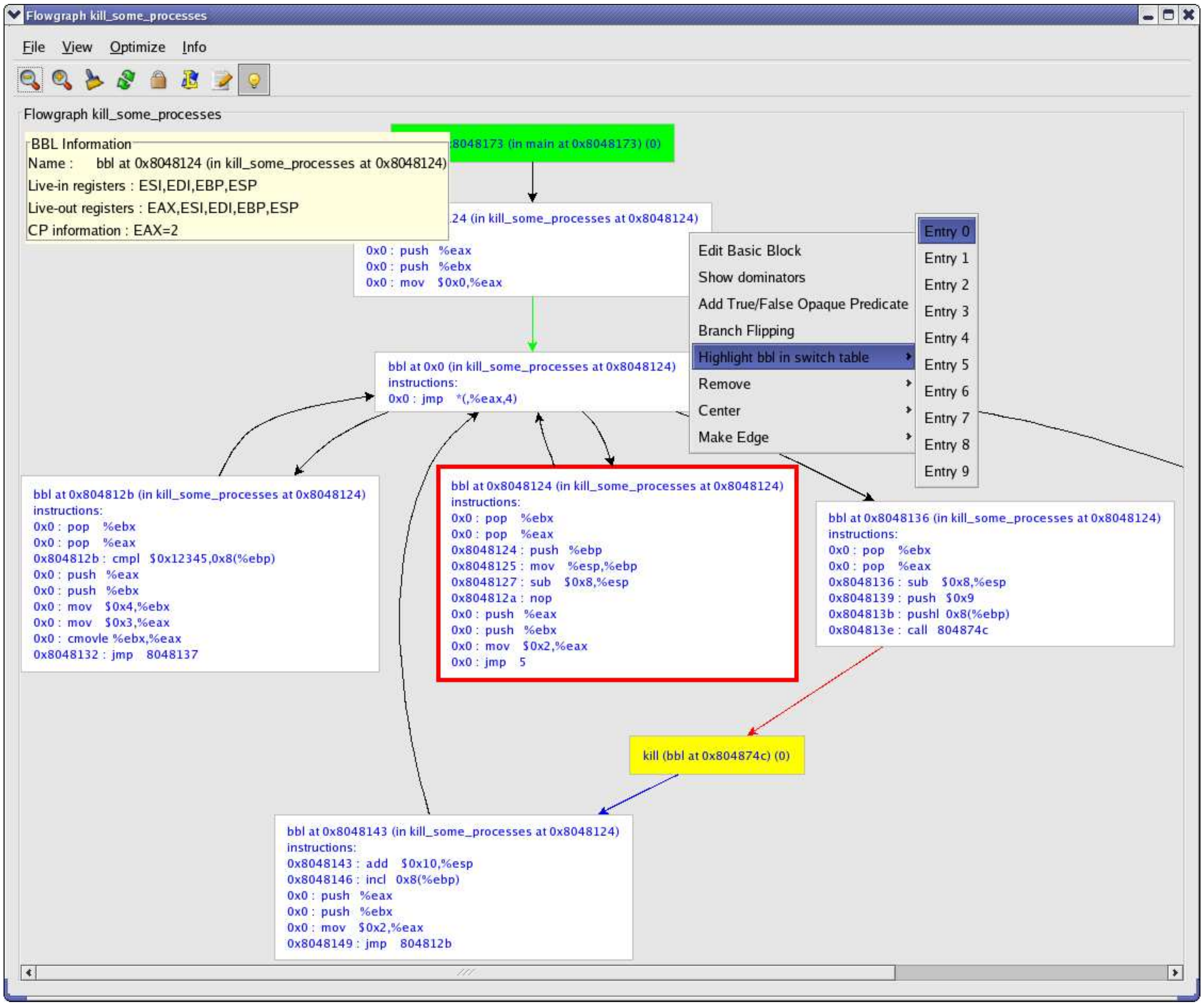


Figure 1. Control flow flattening

The insertion of *opaque predicates* [3] and the *flattening of the control flow graph* [9] are the most known obfuscation transformations. An opaque predicate such as $\forall x, y \in \mathbb{Z}: \ 7y^2 - 1 \neq x^2$ [1] exploits a property which is known at obfuscation time, but that is hard to derive afterwards. Insertion of such predicate introduces new (seemingly executable) paths through the program that have the



BBL Information
Name : bbl at 0x8048124 (in kill_some_processes at 0x8048124)
Live-in registers : ESI,EDI,EBP,ESP
Live-out registers : EAX,ESI,EDI,EBP,ESP
CP information : EAX=2

```
8048173 (in main at 0x8048173) (0)
```

```
24 (in kill_some_processes at 0x8048124)
0x0 : push  %eax
0x0 : push  %ebx
0x0 : mov   $0x0,%eax
```

```
bbl at 0x0 (in kill_some_processes at 0x8048124)
instructions:
0x0 : jmp   *(,%eax,4)
```

```
bbl at 0x804812b (in kill_some_processes at 0x8048124)
instructions:
0x0 : pop   %ebx
0x0 : pop   %eax
0x804812b : cmpl  $0x12345,0x8(%ebp)
0x0 : push  %eax
0x0 : push  %ebx
0x0 : mov   $0x4,%ebx
0x0 : mov   $0x3,%eax
0x0 : cmovle %ebx,%eax
0x8048132 : jmp   8048137
```

```
bbl at 0x8048124 (in kill_some_processes at 0x8048124)
instructions:
0x0 : pop   %ebx
0x0 : pop   %eax
0x8048124 : push  %ebp
0x8048125 : mov   %esp,%ebp
0x8048127 : sub   $0x8,%esp
0x804812a : nop
0x0 : push  %eax
0x0 : push  %ebx
0x0 : mov   $0x2,%eax
0x0 : jmp   5
```

```
bbl at 0x8048136 (in kill_some_processes at 0x8048124)
instructions:
0x0 : pop   %ebx
0x0 : pop   %eax
0x8048136 : sub   $0x8,%esp
0x8048139 : push  $0x9
0x804813b : pushl 0x8(%ebp)
0x804813e : call  804874c
```

```
kill (bbl at 0x804874c) (0)
```

```
bbl at 0x8048143 (in kill_some_processes at 0x8048124)
instructions:
0x8048143 : add   $0x10,%esp
0x8048146 : incl  0x8(%ebp)
0x0 : push  %eax
0x0 : push  %ebx
0x0 : mov   $0x2,%eax
0x8048149 : jmp   804812b
```

Figure 2. The obfuscated procedure *kill_some_processes* **with additional screens to make it easier for a security analyst to deobfuscate the code.**

purpose to mislead the attacker. Another obfuscation transformation which changes the control flow of the program is control flow flattening. It is the most popular control flow obfuscation technique and is first described by Wang [9]. As can be seen in Figure1, a CFG is transformed such that all basic blocks of a procedure appear to have the same set of predecessors and successors. This obfuscation technique is the key technology in an industrial obfuscation tool by Cloakware Inc.[2]. This technique was developed to be applied on Java programs, but can easily be applied on an x86 program. Control flow flattening is also used for watermarking [11] and tamper resistance [10].

Using a simple example, we will show how LOCO helps a security analyst in understanding obfuscated code. We will obfuscate a simple piece of code containing a malicious content; killing some processes. The assembly code generated for the source code shown below is rather easy to understand.

```
int kill_some_processes(int i)
{
  for (;i<=0x12345;i++)
    kill(i,9);
}
```

Obfuscating this code with the standard control flow flattening makes it less comprehensive. The obfuscated x86 assembly code can be seen in Figure 2. A security analyst could try to understand the obfuscated code as such, or he could first try to simplify the code. For this simplification he can use some features of LOCO to assist him. In the next section, we discuss those features.

3 Deobfuscation infrastructure

There are two difficulties in analyzing an obfuscated program: creating a CFG representation of the code and extracting the functionality of the code. In our case, we only focus on the latter and assume that the analyst has been able to build, at least partially, a CFG of the program under study.

In the case of LOCO, all the necessary information is available to allow the analyst to modify the CFG, without the need to take care of adapting address calculations or linearizing the CFG. As such, an analyst can create an executable version of the code at every moment and test if the transformations applied so far are semantics-preserving.

Transformations in LOCO can be applied manually or automatically. In a basic block, instructions can be inserted, deleted, moved or changed. Sometimes, changing instructions will change the control flow as well, so LOCO allows a user to modify the control flow by adding, removing or retargeting edges. In case there are obvious side effects of some actions, the side effects can be applied automatically, like e.g. removing a fallthrough path when a jump instruction is made unconditional.

Manual inspection of the code could be a good starting point for the deobfuscation process. Starting from the entry basic block of a function, a security analyst could dig into the code to reveal superfluous paths, inefficient code or malicious content. A function has a certain structure and suspicious paths will be detected much faster after gaining some experience. For example, it is very uncommon that the entry basic block of a function does not start with the two instructions: *push %ebp* and *mov %esp,%ebp*, which is the case in our obfuscated example in Figure 2.

LOCO helps the security analyst by providing useful information about the program internals. In the current tool, the security analyst has liveness analysis, constant propagation and dominator analysis at his disposal to extract information from the control flow graph. In our case e.g. a *dispatcher variable*[3] is used for control flow flattening. Constant propagation could help the analyst finding the value of the dispatcher variable when some path in the CFG is taken.

Editing the CFG by changing edges and instructions in a basic block can make other instructions become superfluous. An analyst can reuse analysis and optimizations originally developed for program compaction to automatically remove (part of) this superfluous code. Examples of this are dead code removal, branch forwarding, unreachable code removal etc.

During the deobfuscation process, a security analyst might experience some shortcomings in the available analyses and deobfuscation transformations or find himself repetitively applying the same set of transformations by hand. In this case, the security analyst can implement his newly found transformation into the LOCO framework to further automate the deobfuscation process.

Besides support for deobfuscation, LOCO can point an analyst to the interesting parts of the program under study. A program can contain a lot of functions, which are mostly not relevant for the analyst. Using some predefined metrics, the procedures in a program can be sorted by their degree of suspicion. These metrics can be e.g. the absence of a procedure prologue or epilogue, overwriting the return address, etc. As such, an analyst can find the obscure parts of a program more efficiently.

4 Demo

In the demo, we will start from an obfuscated procedure, as shown in Figure 2 and deobfuscate it. During the deobfuscation process we will make use of manual graph modifications and instruction edits. Constant propagation will be used to find the value of the dispatcher variable (although this is trivial in this oversimplified example). Using dead code removal, the resulting code will be cleaned. In this example we will end up with several superfluous stack operations. We will explain how an analyst can add his own transformation to automate the removal of such instruction sequences.

After the deobfuscation, the result is compared with the original procedure. We will then show how this code has been obfuscated using the built-in obfuscation transformations in LOCO. We will also show the provided functionality to scan a program for suspicious code fragments.

5 Limitations and Future Work

LOCO is developed as an experimental environment to help a security analyst in understanding obfuscated code and is not yet able to deobfuscate or reverse engineer real-world malicious software such as obfuscated viruses. The infrastructure currently assumes the existence of a control flow graph derived from the malicious program. Disassembling an obfuscated malicious program and deriving a control flow graph from it is not in the scope of this paper, although we are currently working on this. We are developing a new frontend which will produce first of all a disassembly

[3] a variable that is used as an offset in a switch table that will steer the control fbw

from an obfuscated binary, based on the novel binary analysis technique proposed by Kruegel *et al.* [5]. Afterwards, a control flow graph will be derived from the disassembled instructions. The resulting CFG might be overly conservative and contain a lot of unrealizable paths. However, even with extra information available, it is nearly impossible to construct the most accurate CFG. More details on the construction of a CFG without information external to the binary can be found in Madou et al. [6].

We think that LOCO is an ideal tool to develop new (de)obfuscation transformations and we think that it can be easily extended to be applicable for other (de)obfuscation scenarios. The tool is the first and currently the only x86 (de)obfuscator and is free to use and modify. This enables other security analysts to extend the tool with their own analysis and transformations.

6 Conclusion

We will demonstrate LOCO, a graphical, interactive, easy-to-use experimental environment to help a security analyst in understanding obfuscated code. With LOCO it is possible to interactively deobfuscate a program using underlying analysis. The program modifications during deobfuscation can be tested on correctness by producing an executable version of the code and verifying the functionality. LOCO reduces the learning effort for unexperienced program analysts and is a good experimentation platform to test obfuscation and deobfuscation techniques.

Acknowledgments

The authors would like to thank the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) and the Fund for Scientific Research Flanders (FWO) for their financial support. This research is also partially supported by Ghent University and by the HiPEAC network

References

[1] G. Arboit. A method for watermarking java programs via opaque predicates. In *Proceedings of ICECR-5*, October 2002.

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

[3] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages (POPL'98)*, pages 184–196.

[4] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere. Link-time optimization of ARM binaries. In *Proc. of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*.

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

[6] M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. In *Proceedings of the 5th ACM Workshop on Digital Rights Management*, pages 75–82. ACM Press, 2005.

[7] M. Madou, L. Van Put, and K. De Bosschere. Loco: An interactive code (de)obfuscation tool. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'06)*.

[8] L. Van Put, B. De Sutter, M. Madou, B. De Bus, D. Chanet, K. Smits, and K. De Bosschere. Lancet: A nifty code editing tool. In *Proc. 6th ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM Press, 2005.

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

[10] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, 12 2000.

[11] K. S. Wilson and J. D. Sattler. Software control fbw watermarking, Aug 2004. Baker and Botts, US2005/0055312 A1.