DOC – ANSWERING THE HIDDEN “CALL” OF A VIRUS

Uday Kumar Eric, Aditya Kapoor, Arun Lakhotia
University of Louisiana at Lafayette

SUMMARY
DOC is a prototype to demonstrate a method for detecting obfuscated calls and returns in binaries. Malicious programs use such obfuscations to hide information about the system calls they make. DOC statically identifies several types of obfuscations, promising to speed up the process of determining whether a program is malicious.

INTRODUCTION
One of the first steps in determining whether a program is malicious is to identify the system calls it makes. If the program performs certain collections of file operations, registry operations, or network operations, it may be good reason to consider it potentially malicious.

The set (sometimes, the sequence) of system calls a program makes we refer to as its behavior. The behavior of a program may be determined by either static analysis or by dynamic analysis. In static analysis, a program is analyzed (by humans and/or tools) without running or simulating it. In dynamic analysis, a program’s behavior is observed, often by trapping the calls or sniffing network activity.

Malware writers have developed obfuscation techniques that make it difficult to statically identify the calls made by their program. These programs effectively make a call without actually using the call instruction (Szor, Ferrie 2001, Virus Bulletin Conference). Doing so increases the difficulty of analyzing a program not least because it defeats the methods typical disassemblers use to identify procedure entry and exit points.

AV companies, therefore, rely on dynamic methods for determining a program’s behavior. For instance, Symantec’s Bloodhound technology executes a program in a sandbox (or an emulator), traps the calls made by the program, and then determines whether it is malicious.

While dynamic analyses are helpful and often necessary, they are often cumbersome, time-consuming and fallible. Malware authors already know many methods for defeating detection through dynamic analysis, including detecting the dynamic analysis method, introducing delay loops to bypass stopping heuristics, and executing their malicious behavior in only particular circumstances. For these reasons alone static analysis is still a critical component of AV strategies, but methods for overcoming obfuscation obstacles are extremely desirable.

In this paper we present the results of using a new tool called DOC (Detector of Obfuscated Calls) to analyze W32.Evol. DOC statically identifies several types of obfuscations related to the call and return instructions. Technical details of the method used by DOC has been described elsewhere (Lakhotia and Kumar 2004, Fourth IEEE International Workshop on Source Code Analysis and Manipulation-SCAM’2004). We will review call/return obfuscations, describe DOC and how it was applied to W32.Evol, and close by summarizing some of the successes and limitations of the approach.

CALL/RETN OBFUSCATIONS

Figure 1 shows a classic example of call obfuscation used by viruses, most notably W32.Evol and Netsky.Z. In the left column is a normal call instruction. In the right column is code containing a sequence of two push instructions and a retn instruction. These three instructions do exactly the same work as the call instruction. They are semantically equivalent.

Other related obfuscations include the substitution of retn instructions and the use of non-contiguous function bodies. For instance a retn may be replaced with a pop ip instruction. Non-contiguous procedure bodies can be created by intertwining a procedure’s code with the code of other procedures, thus making it difficult to match a call instruction to its corresponding retn instructions.
Such obfuscations take away important cues that are used during both automated and manual analysis. While a determined, experienced programmer can discover the obfuscations; the time spent in making the discovery can be precious when the malware is actively spreading.

Substituting call instructions, in particular, breaks most automated methods for detecting a virus since these methods depend on recognizing call instructions to (a) identify the kernel functions used by the program and (b) to identify procedures in the code. As is shown later, IDA Pro, a disassembler used very widely in the industry, gives incorrect and misleading results in the presence of call/return obfuscations.

**ABOUT DOC**

DOC is implemented Java as a plug-in to the Eclipse Platform (www.eclipse.org). Figure 2 shows a screenshot DOC when opening an assembly file (.asm extension). DOC provides the ability to open any number of projects at the same time. The navigator view is used to browse and open files in a project. The files are displayed in the file view.

DOC takes as input an assembly file or a disassembled binary obtained from a disassembler such as IDA Pro. A user may choose any of the following three analyses:

- Match call-ret instructions
- Detect obfuscated calls
- Detect obfuscated returns

DOC returns its results by highlighting and annotating the assembly. The annotations contain links to related code when there are multiple occurrences of the same type of obfuscation.

**INSIDE DOC**

DOC uses abstract interpretation, a technique commonly used in static analysis. In this technique a program is interpreted using abstract values, instead of real values. The key challenge in using abstract interpretation is in choosing the right abstraction.

DOC creates an abstraction of the stack and its contents. A specific instance of a real stack is represented as an abstract stack. Further, the set of all possible abstract stacks for all possible executions of a program is represented as an abstract stack graph. Though the set of all abstract stacks (or real stacks) for all possible executions of a program may be infinite, the abstract stack graph is finite.

The abstract stack graph for a given assembly program is constructed by interpreting each instruction of the program. The operations performed by the instruction on a real stack are instead performed on an abstract stack graph. Each instruction is interpreted at most once.

Once the abstract interpretation terminates, the abstract stack graph contains an abstraction of all possible stacks at each statement. DOC analyzes the abstract stack to match call-ret instructions, detect obfuscated calls, and detect obfuscated returns.

**W32.EVOL – REVEALING THE HIDDEN**

Our efforts at statically analyzing W32.Evol, in fact, led us to developing DOC. It all started a few years ago as a result of our first attempt at developing an AV scanner based on formal, static analysis. We had implemented behavior-based analyzer using model checking. Our analyzer failed miserably when we exposed it to W32.Evol.
A closer analysis revealed that the virus was obfuscating all system calls, and our analyzer assumed that IDA Pro would correctly detect system calls in disassembled code. It failed and, as is so common in developing new technologies, the failures provided the impetus to explore new methods. Here we describe some of the causes for disassembly failure and show how DOC can detect these.

**Call/Ret Obfuscation in W32.Evol**

The common sequence of instructions to make a system call, say `GetTickCount`, on a Windows environment is as follows:

```
push add1 ; "kernel32.dll"
push add2 ; "GetTickCount"
call GetProcAddress
call [eax] ; "call GetTickCount"
```

Here addr1 and addr2, respectively, are pointers to strings "kernel32.dll" and "GetTickCount" located in the data segment. The addresses of these strings are pushed on the stack. The kernel32.dll function GetProcAddress is called, which returns the address of the function "GetTickCount" in the eax register. The program then does an indirect call to the address in eax, effectively making a call to `GetTickCount`.

Disassemblers, such as IDA Pro, can detect such patterns of call and aid in detecting system calls. Figure 3 shows a code fragment from W32.Evol for calling the function `GetTickCount`. This code has multiple obfuscations, none of which are detected by IDA Pro. The reasons for this are instructive.

IDAPro assumes that the `retn` instruction at address 0040156A actually returns from the procedure. Thus, it deems this statement as ending the procedure that has entry at address 00401530. IDA Pro indicates the end of a procedure by introducing the dummy directive `endp`. Thus it deduces that the `retn` statement matches “call 00401530” instructions.

The `retn` instruction, it turns out, is performing a call. The value returned from `GetProcAddress` is moved to the stack, and the stack pointer modified such that when the `retn` instruction is executed, it transfers control to `GetTickCount`. This can be verified by manually analyzing the virus in a debugger such as OllyDbg. Figure 4 presents the code of Figure 3 with annotations created by such a manual analysis.

**Detecting call obfuscations**

Figure 5 shows a portion of the code where DOC detects the obfuscated call to the kernel function `GetTickCount()`. The `push` instruction at address 00401557 and the `retn` instruction at address 0040156A are instrumental in obfuscating the call to `GetTickCount()`. This is indicated by highlighting these instructions in red. The annotation “(0)” at the end of these instructions indicates that the two belong to the same call obfuscation.

W32.Evol uses similar code to make system calls in 25 locations. IDA Pro misses all of these calls, where as DOC highlights every such `retn` instruction as making a call.

**Matching call-retn instructions**

Figure 6 shows the same code as Figure 3, but it also shows the results of running DOC’s analysis for matching call-retn instructions. The two call instructions at addresses 00401557 and 0040156A are highlighted and are annotated “(2)” and “(3)”, respectively. These numbers are arc
labels in the effective call graph. Figure 7 shows return sites corresponding to these statements. These statements are annotated with the numbers “(2)” and “(3)”, which are matched to the call sites so labeled. This figure also shows retn statements matching call sites annotated as “(0)” and “(1)”. As is expected, one retn statement may match multiple call sites.

DOC correctly found matching retn statements for all 33 call statements of W32.Evol. In several instances the procedure code was not contiguous.

CONCLUSIONS

DOC is efficient, being linear in both space and time. And it is demonstrably effective in finding the sort of call/retn obfuscations found in W32.Evol. We believe its techniques can be an important part of an AV researcher’s toolkit, and can significantly speed up analysis of obfuscated binaries. DOC has limitations. It is restricted solely to detecting call obfuscations, and cannot handle some of these, including manual stack manipulation. Efforts to overcome some of these limitations are in progress in our laboratory.