TECHNICAL FEATURE

DOC – ANSWERING THE HIDDEN 'CALL' OF A VIRUS

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

Malicious programs use obfuscations to hide information about the system calls they make. Detector of Obfuscated Calls, or DOC, is a prototype tool which demonstrates a technique for detecting obfuscated calls and returns in binaries. DOC identifies several types of obfuscations statically, promising to speed up the process of determining whether or not 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, there may be good reason to consider it likely to be malicious.

The set (or sequence) of system calls a program makes is referred to as its behaviour. The behaviour of a program may be determined either by static analysis or by dynamic analysis.

In static analysis, a program is analysed (by humans and/or tools) without running or simulating it. In dynamic analysis, a program's behaviour is observed, often by trapping the calls or sniffing network activity.

Malware writers have developed obfuscation techniques that make it difficult, using static analysis techniques, to identify the calls made by their program. Effectively, these programs make a call without actually using the call instruction (see Peter Ször and Peter Ferrie, *Virus Bulletin Conference 2001*). Doing this increases the difficulty of analysing a program not least because it defeats the methods that typical disassemblers use to identify procedure entry and exit points.

Therefore, anti-virus companies tend to rely on dynamic methods for determining a program's behaviour. 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 or not it is malicious.

However, while dynamic analyses are helpful and often necessary, they have a tendency to be cumbersome, time-consuming and fallible.

Malware authors already have 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 behaviour only in particular circumstances.

For these reasons alone static analysis is still a critical component of anti-virus strategies, but methods for overcoming obfuscation obstacles are extremely desirable.

In this article we present the results obtained by using a new tool called DOC (Detector of Obfuscated Calls) to analyse the virus W32/Evol. DOC identifies statically several types of obfuscations related to the call and return instructions.

Technical details of the method used by DOC have been described elsewhere (Lakhotia and Kumar 2004, *Fourth IEEE International Workshop on Source Code Analysis and Manipulation*). In this article will review call/return obfuscations, describe DOC and how it was applied to W32/Evol, and summarise 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.

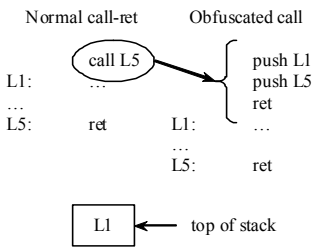


Figure 1. Call obfuscation.

In the left-hand column is a normal call instruction. In the right-hand 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 would be able to discover the obfuscations, the time that it takes to make the discovery can be very precious when the malware is spreading actively.

Substituting call instructions, in particular, breaks most automated methods for detecting a virus since these methods depend on recognizing call instructions both to identify the kernel functions used by the program and to

identify procedures in the code. As is shown later, *IDA Pro*, a disassembler used very widely in the anti-virus industry, gives incorrect and misleading results in the presence of call/return obfuscations.

ABOUT DOC

DOC is implemented in Java as a plug-in to the Eclipse Platform (see http://www.eclipse.org/). Figure 2 shows a screenshot of DOC when opening an assembly file (.asm extension).

DOC allows any number of projects to be opened at the same time. The navigator view (on the left-hand side) is used to browse and open files in a project. The files are displayed in the file view (shown on the right).

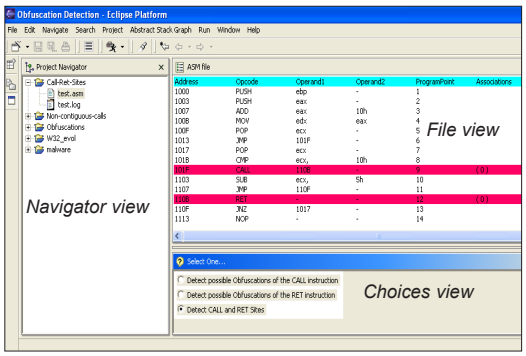


Figure 2. DOC user interface.

DOC takes as its input an assembly file or a disassembled binary obtained from a disassembler such as *IDA Pro*. The user may select 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. Although 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 performed instead 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 analyses the abstract stack to match call-ret instructions, detect obfuscated calls, and detect obfuscated returns.

W32/EVOL – REVEALING THE HIDDEN

It was our efforts at analysing W32/Evol statically that led us to develop DOC. It all started a few years ago as a result of our first attempt at developing an anti-virus scanner based on formal, static analysis. We had implemented a behaviour-based analyser using model checking – however, our analyser failed miserably when we exposed it to W32/Evol.

A closer analysis revealed that the virus was obfuscating all system calls, and our analyser made the assumption that *IDAPro* would detect system calls correctly in disassembled code. It failed and, as is so common in developing new technologies, its failure 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 (for example GetTickcount) in a *Windows* environment is as follows:

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

Here, addr1 and addr2 are pointers to the 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

```
:0040153F                mov     dword ptr [eax], 'TteG'
:00401545                mov     dword ptr [eax+4], 'Ckci'
:0040154C                mov     dword ptr [eax+8], 'tnuo'
:00401553                mov     byte ptr [eax+0Ch], 0
:00401557                push    eax
:00401558                call    sub_401280
:0040155D                push    eax
:0040155E                call    sub_4012A7
:00401563                mov     [ebp+0], eax
:00401566                add     esp, 10h
:00401569                pop     ebp
:0040156A                retn
:0040156A sub_401530     endp ; sp = -0Ch
```

Figure 3. W32/Evol code with multiple obfuscations.

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.

IDA Pro assumes that the retn instruction at address 0040156A actually returns from the procedure. Thus, it deems this statement as ending the procedure that has an 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 is modified such that when the retn instruction is executed, it transfers control to GetTickCount.

```
0040153F mov dword ptr ds:[eax], 54746547 ;'TteG'
00401545 mov dword ptr ds:[eax+4], 436B6369 ;'Ckci'
0040154C mov dword ptr ds:[eax+8], 746E756F ;'tnuo'
00401553 mov byte ptr ds:[eax+c], 0;  '\0'
00401557 push eax; ptr to "GetTickCount".
00401558 call 00401280; gets base address of
                        kernel32.dll base.
0040155D push eax;
0040155E call 004012A7; obfuscated call to
                        GetProcAddress()
00401563 mov dword ptr ss:[ebp], eax; addr of
                                      GetTickCount().
00401566 add esp, 10
00401569 pop ebp
0040156A retn; transfer control to GetTickCount().
```

Figure 4. Annotated code of Figure 3.

This can be verified by analysing the virus manually 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().

0040153F	MOV	DS:[EAX],	54746547	443	
00401545	MOV	DS:[EAX+4],	436B6369	444	
0040154C	MOV	DS:[EAX+8],	746E756F	445	
00401553	MOV	DS:[EAX+C],	0	446	
00401557	PUSH	EAX	-	447	(0)
00401558	CALL	00401280	-	448	
0040155D	PUSH	EAX	-	449	
0040155E	CALL	004012A7	-	450	
00401563	MOV	SS:[EBP],	EAX	451	
00401566	ADD	ESP,	10	452	
00401569	POP	EBP	-	453	
0040156A	RETN	-	-	454	(0)

Figure 5. Using DOC to detect obfuscated call.

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.

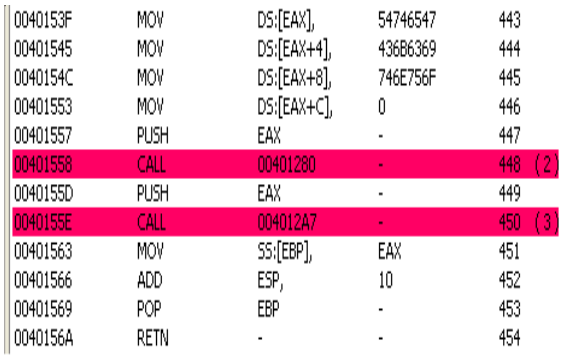
W32/Evol uses similar code to make system calls in 25 locations. *IDA Pro* misses all of these calls, whereas DOC highlights every such retn instruction as making a call.

Matching call-retn instructions

Figure 6 shows the same code as that shown in Figure 3, but it also shows of the results of running DOC's analysis for matching call-retn instructions.

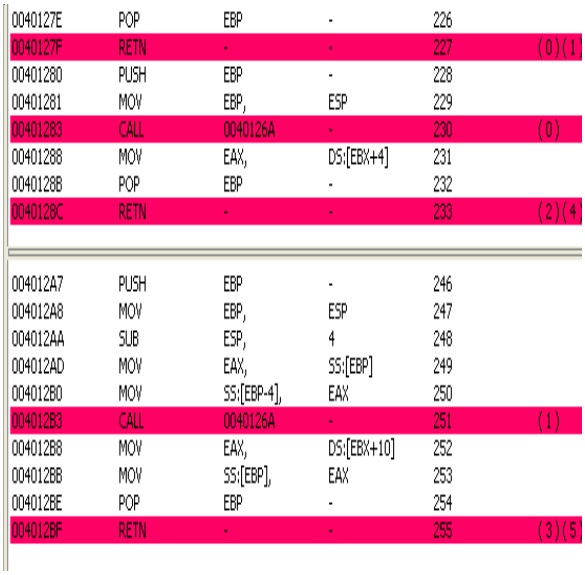
The two call instructions at addresses 00401558 and 0040155E are highlighted and are annotated '(2)' and '(3)', respectively. These numbers are arc labels in the effective call graph.

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 labelled. 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



0040153F	MOV	DS:[EAX],	54746547	443
00401545	MOV	DS:[EAX+4],	436B6369	444
0040154C	MOV	DS:[EAX+8],	746E756F	445
00401553	MOV	DS:[EAX+C],	0	446
00401557	PUSH	EAX	-	447
00401558	CALL	00401280	-	448 (2)
0040155D	PUSH	EAX	-	449
0040155E	CALL	004012A7	-	450 (3)
00401563	MOV	SS:[EBP],	EAX	451
00401566	ADD	ESP,	10	452
00401569	POP	EBP	-	453
0040156A	RETN	-	-	454

Figure 6. Using DOCs to detect valid calls.



0040127E	POP	EBP	-	226	
0040127F	RETN	-	-	227	(0)(1)
00401280	PUSH	EBP	-	228	
00401281	MOV	EBP,	ESP	229	
00401283	CALL	0040126A	-	230	(0)
00401288	MOV	EAX,	DS:[EBX+4]	231	
0040128B	POP	EBP	-	232	
0040128C	RETN	-	-	233	(2)(4)
004012A7	PUSH	EBP	-	246	
004012A8	MOV	EBP,	ESP	247	
004012AA	SUB	ESP,	4	248	
004012AD	MOV	EAX,	SS:[EBP]	249	
004012B0	MOV	SS:[EBP-4],	EAX	250	
004012B3	CALL	0040126A	-	251	(1)
004012B8	MOV	EAX,	DS:[EBX+10]	252	
004012BB	MOV	SS:[EBP],	EAX	253	
004012BE	POP	EBP	-	254	
004012BF	RETN	-	-	255	(3)(5)

Figure 7. Using DOC to detect valid call-ret sites.

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 inW32/Evol. We believe its techniques could become an important part of an anti-virus researcher's toolkit, and that they can significantly speed up analysis of obfuscated binaries.

DOC does have a number of 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 currently in progress in our laboratory.