Control Flow Obfuscations in Malwares

Author: Sudeep Singh

Introduction

In this paper I will discuss about the control flow obfuscations used in malwares. The purpose of using these control-flow obfuscations, how they are done and how they are used to deter reverse engineering will be discussed.

The term control flow obfuscation is used in this article to indicate code sections in the binary, which are added in order to make the comprehension of program more difficult.

After this, I also present a pintool I have written to help detect some important sequence of instructions, which will be of interest to the virus analyst.

Note: You may need to zoom into the screenshots of disassembly included to view them clearly.

Purpose of Control Flow Obfuscations

The two main reasons of using control flow obfuscations in malwares are:

1. To deter the static reverse engineering of malwares. It becomes more difficult to target the code sections of interest.
2. To defeat the static signatures used by AV vendors, which rely on specific byte sequences in the binary to detect them.

Application Defined Callback Functions

There are certain APIs provided by Microsoft, which allow us to register a Callback Function. These can be used by malwares to hide the main logic of their code. They can pass a pointer to the malicious subroutine as the callback function parameter for the API.

Window Procedure

Using `RegisterClassExA()`, a Window Procedure can be registered for a specific Class Name. All the windows with that class name will have the same Window Procedure.

When a window is created using `CreateWindowA()`, the Window Procedure is invoked with certain default window messages like `WM_CREATE`, `WM_NCCREATE` and so on.

However, the main virus code will be executed only when a particular windows message is received.

Let us take as an example a virus which calls malicious subroutine indirectly:

After unpacking the malware, the first thing it does is to register a Window Class with the name, "Runtime Check" with the Window Procedure subroutine at address, 00402680. It then creates the
Window. During the creation of the Window, the Window Procedure is invoked which handles the initial window messages like WM_CREATE.

After the Window is created, it retrieves the message from the Thread’s queue using GetMessage() and dispatches it to the Window Procedure using DispatchMessage().

Inside the Window Procedure, it reads the code of the Window Message from the stack and stores it in the EAX register. It then checks whether the window message code is greater than 0xF. If it is equal to 0x113, then it sets up a Timer that elapses after 1 second. Since the last parameter to the SetTimer() function is NULL, the system will post a WM_TIMER message to the queue every time the timer elapses. Each time a WM_TIMER message is retrieved from the application thread’s message queue using GetMessage(), it increments a counter. Once the counter is equal to 5, it calls the malicious subroutine. Since the timer is set to elapse after 1 second, so overall delay introduced is approximately, 5 seconds.
Window Procedure:

Set the Timer:

Check the Window Message Code:
Below are the corresponding sections of code:

https://gist.github.com/c0d3inj3cT/7611371#file-wmtimer-asm

And here is the code rewritten in C:

```c
if(wind_code > 0xF)
{
    if(wind_code == 0x113)
    {
        counter++;
        if(counter == 0x5)
        {
            call malicious_code;
        }
    }
}
else if(wind_code == 0xF)
{
    // code for handling the WM_PAINT message
}
else if(wind_code == 0x1)
{
    SetTimer(hWnd, 1, 0x3e8, 0)
}
```

As can be seen, this method can be used to introduce any amount of delay in execution. Since, most automated sandboxes detect the delays in execution by checking for Sleep()/SleepEx()/NtDelayExecution() API calls, this method would bypass such detections.

**DialogBoxParamA()**:  
This is another API, which takes the address of the Window Procedure as one of the input parameters. Below is an example of a virus that executes the main code section only when it receives the WM_COMMAND window message.

```
0401395E | 54 444400 | PUSH DWORD PTR DS:[4444400]  | hWnd = NULL
04013961 | 55 00    | POP ESI             | hWnd = NULL
04013962 | E8 590000 | CALL DWORD PTR DS:[590000]  | hWnd = NULL
04013967 | 8B EF    | LEA EAX, DWORD PTR DS:[EBP-4]  | hWnd = NULL
04013969 | 44 85C04400 | CALL DWORD PTR DS:[EBP-4]  | hWnd = NULL
0401396C | 55 00    | POP ESI             | hWnd = NULL
0401396D | E8 594400 | CALL DWORD PTR DS:[594400]  | hWnd = NULL
04013970 | 8B EF    | LEA EAX, DWORD PTR DS:[EBP-4]  | hWnd = NULL
04013972 | 44 85C04400 | CALL DWORD PTR DS:[EBP-4]  | hWnd = NULL
04013975 | 55 00    | POP ESI             | hWnd = NULL
04013976 | 8B ED    | LEA EAX, DWORD PTR DS:[EBP-3]  | hWnd = NULL
04013978 | 55 00    | POP ESI             | hWnd = NULL
04013979 | 0F B7444400 | CALL DWORD PTR DS:[594400]  | hWnd = NULL
```

<--- Callback Function

```
0401394E | 89 C8    | MOV ECX, EAX      | hWnd = NULL
04013951 | 55 00    | POP ESI             | hWnd = NULL
04013952 | 0F B7444400 | CALL DWORD PTR DS:[594400]  | hWnd = NULL
```

<--- ExitProc

```
0401394E | 89 C8    | MOV ECX, EAX      | hWnd = NULL
04013951 | 55 00    | POP ESI             | hWnd = NULL
04013952 | 0F B7444400 | CALL DWORD PTR DS:[594400]  | hWnd = NULL
```

<--- ExitProc
Here is another example of a Windows API, which takes an application defined callback function as one of the input parameters.

By passing the pointer to malicious subroutine as the callback function, we can invoke it indirectly through `EnumSystemLocalesA()` as shown below:

```
EnumSystemLocalesA():

Also, it can be seen that there is a control flow obfuscation which finally redirects the execution to the address 0x4013A7 which is in between two assembled instructions. This would result in updating the view of Debugger since the disassembly changes.

The main impact of using this technique is that the code will be executed if we step over the call to these APIs. As a result of this, we need to set a breakpoint at the callback function just before the API is invoked. We will break at the callback function in the debugger as soon as the API is executed, this way we can continue stepping through the code.

While this technique may appear to be easy for a seasoned reverse engineer, its usage is becoming increasingly common among malwares these days.
There are several other Window APIs provided which accept an application defined callback function as one of the input parameters.

**Execution through Exception Handlers**

Malwares could also redirect the execution to the malicious subroutine by triggering an exception. In order to do this, they first register an exception handler using either `RtlAddVectoredExceptionHandler()` or by registering a new Structured Exception Handler.

The exception can be invoked using either of the following:

1. Triggering a memory access violation (0xc0000005) by attempting to write to a memory address to which there is no write access or by attempting to call an invalid memory address.
2. Executing a privileged instruction like STI or CLI, which would result in a Privileged Exception in protected mode (0xc0000096).
3. Performing a division by zero to trigger the exception (0xC0000094).

**Execution through Exception Handler for 0xc0000096:**

Below is an example of a malware, which calls the malicious code by triggering a Privileged Instruction exception.

It first registers an exception handler. Then it decrypts the code of that exception handler.

```
00401748  94 43  XOR $44, $43
0040174b  D5 BB  JMP SHORT got3frame.0040173c
0040174e  50 4d  JMP SHORT gotifjdf.0040174c
00401751  89 74 FF  MOV PTR DS:$EC$x2+401752
00401754  50 4d  JMP SHORT gotifjdf.0040174c
00401757  82 56 FF  UMP SHORT gotifjdf.00401745
0040175a  89 4c FF  LOOPO SHORT gotifjdf.00401741
0040175d  51 FF  CLI
```

```
0040175f  89 54 5D  MOV EDI, [ESI+0x1610]
00401763  82 56 FF  UMP SHORT gotifjdf.00401745
00401766  89 4c FF  LOOPO SHORT gotifjdf.00401741
00401769  50 FF  MOV ESP, [EIP]
0040176c  50 FF  MOV ESP, [EIP]
```

Once this is done, it triggers an exception by executing a privileged instruction like CLI or STI (both these instructions are privileged in the protected mode).

Since an exception is triggered, the corresponding exception handler from the SEH chain will be invoked. This is a control flow obfuscation trick. Below screenshots show an exception triggered after executing the CLI instruction. On the stack we can see the exception handler address as: 0x00401610.
To continue the analysis in Olly Debugger, we can press, Shift + F9 and pass the exception to the exception handler or we can just set the EIP to 0x00401610.

<table>
<thead>
<tr>
<th>00001658</th>
<th>CALL g4f2f2f.0000161f</th>
<th>&lt;--- Resume Execution from the Exception Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001658</td>
<td>5A CF500002</td>
<td></td>
</tr>
<tr>
<td>0000165B</td>
<td>2D B1</td>
<td></td>
</tr>
<tr>
<td>0000165D</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>0000165E</td>
<td>0000000000000000000000</td>
<td></td>
</tr>
<tr>
<td>00001660</td>
<td>DB</td>
<td>ADD EAX,EBX</td>
</tr>
<tr>
<td>00001661</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>00001662</td>
<td>BCE</td>
<td></td>
</tr>
<tr>
<td>00001663</td>
<td>DXE</td>
<td>FSR STK,ST</td>
</tr>
<tr>
<td>00001664</td>
<td>60</td>
<td>ADD EAX,EBX</td>
</tr>
</tbody>
</table>

### Execution through Vectored Exception Handler:

Below is an example of a malware, which calls the malicious subroutine through a Vectored Exception Handler.

The handler checks only for memory access violation (0xc0000005) exception. It retrieves the address of the faulting instruction from EXCEPTION_RECORD structure and compares it with the address it expects. If they are equal it will set the value of EIP in the CONTEXT structure to malicious subroutine address (0x00401f03 in this case) so that execution resumes there after exception handling completes.

### Execution through RaiseException:

There are also some cases where debuggers like Olly Debugger do not pause at the exception Handler when an exception is triggered and instead run the code.

One such case is when we trigger an exception by calling `RaiseException()` with the exception code, 0x80000003.

It first registers an exception handler, which has the malicious subroutine code and then triggers the exception by calling RaiseException.

In this case, we can manually set the EIP to 0x0040126D (Structured Exception Handler) and continue debugging from there.
Execution through Exception Handler for 0xC0000094:

In the case below, the virus redirects execution to exception handler by triggering the exception, division by zero.

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x40B65F</td>
<td>PUSH EBP</td>
<td>Calculate Address of Instruction after Faulting Instruction</td>
</tr>
<tr>
<td>0x40B660</td>
<td>MOV EBP, ESP</td>
<td></td>
</tr>
<tr>
<td>0x40B661</td>
<td>ADD ESP, 0x10000000</td>
<td></td>
</tr>
<tr>
<td>0x40B662</td>
<td>LEA ES, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B663</td>
<td>LEA DS, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B664</td>
<td>LEA DS, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B665</td>
<td>LEA ES, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B666</td>
<td>LEA ES, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B667</td>
<td>LEA DS, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B668</td>
<td>LEA ES, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B669</td>
<td>LEA DS, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B66A</td>
<td>LEA ES, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B66B</td>
<td>LEA DS, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B66C</td>
<td>LEA ES, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B66D</td>
<td>LEA DS, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B66E</td>
<td>LEA ES, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B66F</td>
<td>LEA DS, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
</tbody>
</table>

Inside the Exception Handler, it sets the address to resume execution from in the CONTEXT Record as the address right after execution point of exception (in our case, 0x40B665)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x40B665</td>
<td>PUSH EBP</td>
<td>Establish Frame</td>
</tr>
<tr>
<td>0x40B666</td>
<td>MOV EBP, ESP</td>
<td>Context Record</td>
</tr>
<tr>
<td>0x40B667</td>
<td>ADD ESP, 0x10000000</td>
<td>Update Context Record with address to resume execution from</td>
</tr>
<tr>
<td>0x40B668</td>
<td>LEA ES, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B669</td>
<td>LEA ES, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B66A</td>
<td>LEA ES, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B66B</td>
<td>LEA ES, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B66C</td>
<td>LEA ES, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B66D</td>
<td>LEA ES, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B66E</td>
<td>LEA ES, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
<tr>
<td>0x40B66F</td>
<td>LEA ES, DWORD PTR DS:[40000000]</td>
<td></td>
</tr>
</tbody>
</table>

**Execution Slide**

There are certain special instructions or sequence of instructions which when executed in the debugger change the default behavior of the debugger (to trap at every instruction).

Below are a few examples:

**INT 2D Instruction:** INT 2D has a special behavior in Olly Debugger. Olly will skip the next byte in execution as a result of which the control flow is obfuscated. This technique is often referred to as byte scission.

It also has a dynamic behavior under different environments (different combinations of user mode/kernel mode debuggers and in case of no debuggers).

**Overwrite RETN:** This is a special behavior observed in Olly Debugger. If we overwrite the RETN instruction with the opcode, 0xC3 (which is the opcode of RETN) just before executing RETN, the debugger does not pause at the RETN address but instead runs the code inside debugger.

Below is a proof of concept I have written for this:
; Overwrite RETN opcode
; Control Flow Obfuscation
; Sudeep Singh

include \masm32\include\masm32rt.inc

.data
hMod dd 0

.code
start:
push cfm$("RETN -- 0xc3 Overwrite\n")
call crt_printf
push cfm$("Make the code section writable\n")
call crt_printf
call nextaddr
nextaddr: pop eax
mov ebx, eax
push 4
    call crt_malloc
mov esi, eax
invoke LoadLibrary, chr$("kernel32.dll")
mov hMod, eax
invoke GetProcAddress, hMod, chr$("VirtualProtect")
mov ecx, eax
push esi
push 040h
push 0100h
push ebx
call ecx
pushad
push cfm$("Enter the proof of concept routine\n")
call crt_printf
call label1
popad ; Debugger will not trap here and instead execute the code
mov eax, 01h
shl eax, 08h
push eax
push cfm$("2 ^ 8 is: %#0x\n")
call crt_printf
call ExitProcess
label1:
call label2
label3: retn
label2:
    retn

end start

Trap Flag Check: We can recover the true value of the Trap Flag bit which is used by Debuggers for single stepping by making the processor suspend the interrupts for the next instruction to be executed.
This can be done by writing to the Stack Segment register using either of the following pairs of instructions:

```
Push SS
Pop SS
PUSHF
```

Or

```
Mov ax, ss
Mov ss, ax
PUSHF
```

This will allow us to recover the true value of EFLAGS register and check for the Trap Flag bit in it. This method has been known for quite some time however not used so often in malwares.

**Junk Instructions**

There are several Polymorphic Engines which are used by malware authors to generate modified versions of their binary which perform the same activities on the machine however their code is modified.

This is often used to bypass static signatures written for malwares by security vendors.

One of the important features of a Polymorphic Engine is the junk instruction generator. Junk instructions are sequence of instructions that do not impact the overall logic of the code in anyway but are placed to deter reverse engineering.

Between every useful instruction, several junk bytes are placed.

The main reasons for injecting junk bytes into the code section are:

1. These junk bytes could correspond to complete instructions which do not alter the overall logic of the code. They increase the size of code section and deter reverse engineering since even though these instructions appear to be legitimate, they have no impact on the main behavior of virus.
2. The junk bytes injected into the code section correspond to partial instructions. This is done to confuse the disassemblers which rely on algorithms like **Linear Sweep** and **Recursive Traversals**.
3. The code can be obfuscated even further by using **opaque predicates** which can be combined with Windows APIs that will always return a fixed value.

Let us now look at each of the above methods by taking real world virus examples:

At first, let us look at a simple example which places a lot of junk bytes at the Entry Point of the Program which correspond to NOPs:
In this case, by combining an easy sequence of instructions like PUSH/POP, a long chain of NOPs is generated. However, once such a pattern is identified, it becomes easy for the reverse engineer to skip such sections of code.

Now, let us look at an example where Window APIs are used in such a way that their return value is constant. By combining multiple calls to Window APIs in this way, a sequence of junk instructions can be generated:

1. **LoadIconA()** is called with an invalid Resource Name so that its return value is always 0x0. As a result of this, the conditional test that follows it becomes an opaque predicate.
2. **GetCurrentThread**() will always return the value 0xffffffff as a result of which Z flag will be set by the conditional test.

3. **GetStockObject**() is called in such a way that return value is always 0x0 so that it falls through the next conditional test.

Here is another example of using Windows APIs along with some junk instruction sequences:

```
In this case we can see that a bit of variation is added by calling GetStockObject() twice, once such that it always returns 0x0 and the second time it is called with a valid parameter (WHITE_BRUSH), so that it returns a non-zero value.

A PUSH/RET sequence is used to jump to the next address.
```

Even though this sequence of instructions might appear to be easy to analyze, when a lot of such sequences are combined together, it can help deter analysis to an extent.

Now, we will look at a sequence of instructions where opaque predicates are created without using Window APIs:

```
Let us now look at examples where control flow is obfuscated by injecting junk bytes in such a way that they form partial instructions and are never executed.
```
Below example shows the disassembly produced by Olly Debugger when the EIP is at the address 00401610. It is important to note that Linear Sweep algorithm is used in this case to generate the disassembly (without the “Analyze Code” option). So, it keeps disassembling the bytes to x86 instructions in sequence as and when it is able form a valid instruction.

```
00401610  E8 04000000  CALL 00401619
00401615  BA DCFE0068  MOV EDX,6800FEDC
0040161A  2016  AND BYTE PTR DS:[ESI],DL
0040161C  40  INC EAX
0040161D  00EB  ADD BL,CH
0040161F  04 BA  ADD AL,0BA
00401621  DCEE  FSUB ST(6),ST
00401623  0059 EB  ADD BYTE PTR DS:[ECX-15],BL
```

The actual control flow for above code when executed is:

```
00401610  E8 04000000  CALL 00401619
00401619  68 20164000  PUSH 00401620
0040161E  EB 04  JMP SHORT 00401624
00401624  59  POP ECX
```

Let us now understand how the junk bytes were injected and how they confused the disassembler.

There were 4 bytes injected in between the valid instructions at addresses, 00401610 and 00401619.

4 junk bytes injected = BA DC FE 00

BA = opcode of instruction, **mov edx, <DWORD>**

This is a 5 byte instruction. However, we can see that only 4 bytes are injected which makes the instruction incomplete.

The last byte required to complete the instruction is used from the valid instruction at address, 00401619. The byte in this case corresponds to the PUSH instruction at 00401619.

Since the disassembler is making use of Linear Sweep algorithm, it disassembles the 5 bytes to:

**MOV EDX,6800FEDC**

As a result of this, the remaining bytes are disassembled incorrectly as well.

Now, let us look at this code in Olly debugger. When we step through the instructions, debugger will follow the proper control flow. However, since the initial disassembly displayed was not as per the control flow of the code, it will be updated each time we step through it as shown below:
Observe how the disassembly changes each time we step through the code and every time the disassembly changes, the view is updated and instruction at EIP will be at the top of the view.

Olly Debugger is capable of using a **Recursive Traversal** algorithm for disassembling the code as well. It provides us an option to use the “Analyze Code” feature which will disassemble the code based on the control flow. Let us use this feature and apply it to the above code.

![Disassembly Example]

We can see that though recursive traversal algorithm is better than linear sweep algorithm at identifying the junk bytes, it is still susceptible to disassembly errors.
The "?" symbol next to the opcodes seen above in Olly Debugger indicates that these instructions were not disassembled properly.

Also, when injecting junk bytes in the code section, we have to make sure that these junk bytes are not executed. In order to do this, unconditional jump instructions are placed before the junk bytes.

Below is an example which shows the initial disassembly and the actual control flow:

```
00401610  E8 04000000 CALL 00401619
00401615  BA DCFE0068 MOV EDX,6800FEDC
0040161A  2016 AND BYTE PTR DS:[ESI],DL
0040161C  40 INC EAX
0040161D  00EB ADD BL,CH
0040161F  04 BA ADD AL,0BA
00401621  DCEE FSUB ST(6),ST
00401623  0059 EB ADD BYTE PTR DS:[ECX-15],BL
00401626  05 EB8B09EB ADD EAX,EB098EB
0040162B  03EB ADD EBP,EBX
0040162D  FA CLI
0040162E  74 58 JE SHORT 00401688
00401630  EB 05 JMP SHORT 00401637
00401632 ^EB 8B JMP SHORT 004016BF
00401634  00EB ADD BL,CH
00401636  03EB ADD EBP,EBX
00401638  FA CLI
00401639 ^74 EB JE SHORT 00401626
0040163B  05 7529C8EB ADD EAX,EBC82975
00401640  04 EB ADD AL,0EB
00401642  FA CLI
00401643 -0F85 C1E008EB JNZ EB48F70A
00401645  07 POP ES
0040164A  3B05 4A164000 CMP EAX,DWORD PTR DS:[40164A]
00401650  7D 6A JGE SHORT 004016BC
00401652  030F ADD ECX,DWORD PTR DS:[EDI]
00401654  C8 EB05EB ENTER 5EB,0EB
```

The actual control flow:

```
00401610  E8 04000000 CALL 00401619
00401615  68 20164000 PUSH 00401620
00401619  EB 04 JMP SHORT 00401624
0040161E  59 POP ECX
00401624  EB 05 JMP SHORT 0040162C
00401625 ^EB FA JMP SHORT 00401628
00401626  8B09 MOV ECX,DWORD PTR DS:[ECX]
00401628  EB 03 JMP SHORT 0040162F
00401629  58 POP EAX
0040162A  EB 05 JMP SHORT 00401637
0040162F ^EB FA JMP SHORT 00401633
00401632  8B00 MOV EAX,DWORD PTR DS:[EAX]
00401633  EB 03 JMP SHORT 0040163A
00401635  EB 05 JMP SHORT 00401641
0040163A  ^EB FA JMP SHORT 0040163D
0040163C  29C8 SUB EAX,ECX
0040163D  04 EB JMP SHORT 00401645
00401643  C1E0 08 SHL EAX,8
00401645  EB 07 JMP SHORT 00401651
00401648  6A 03 PUSH 3
```
You can observe the excessive use of unconditional jumps to prevent the junk bytes from executing.

**Detection of Interesting Instructions using Pintool**

Now, let us look at the pintool, which I have written to detect interesting sequence of instructions in malwares.

The reason I wrote a Pintool to do this is because if we rely on Static Byte Signatures, then we are limited to static analysis of the binary (on disk). If the binary is packed then we might not be able to detect the interesting instructions, which would be executed after the binary is unpacked in memory.

Since pintool allows us to perform Dynamic Binary Instrumentation, it would be good to make use of it for this purpose.

Please note that this pintool is not specifically related to control flow obfuscations.

It can be used to detect the following:

1. Obfuscated code sections of the malware.
2. Encryption/Decryption Routines.
3. Function Name Hash generation routines.
4. Junk Instructions inserted by Polymorphic Engines.
5. Privileged Instructions
6. Some methods like GetPC, which are often used by shellcode to be position independent.
7. Execution of special instructions like SIDT, SLDT, SGDT, which indicate the usage of Anti VM tricks.
8. Execution of RDTSC, which may indicate the usage of Anti Debugging Tricks.
9. And some more interesting instructions can be discovered.

I wrote this tool to help me while analyzing malwares and also to discover interesting viruses in the wild. This is more of a concept at present and it can be extended to discover more malware attributes at an instruction level.

Please note that some of the characteristics mentioned above will also be observed in known packers like UPX, ASPack and so on. You can quickly identify the known packers with PEiD and a good database of known packers byte signatures.

Interestingly, if you run this pintool against a benign binary, you will observe very little to almost no output. As a result of this, it can also be used to detect malicious binaries based on the type of instructions executed.
Below is the code written:

/*
Instruction Tracer to identify
interesting sequence of instructions
in malwares.
Sudeep Singh
*/

#include <stdio.h>
#include <iostream>
#include "pin.H"

VOID Instruction(INS ins, VOID *v)
{
    if(INS_Opcode(ins) == XED_ICLASS_XOR && INS_Address(ins) < 0x3d930000)
    {
        if(INS_MaxNumRRegs(ins) == 1)
        {
            cout << hex << INS_Address(ins) << " : " << INS_Disassemble(ins) << endl;
        }
        else
        {
            string regRead;
            string regWrite;
            regWrite = REG_StringShort(INS_RegW(ins, 0));
            regRead = REG_StringShort(INS_RegR(ins, 0));
            if(regRead.compare(regWrite) != 0 && regRead.compare("ebp") != 0 && regWrite.compare("ebp") != 0)
            {
                cout << hex << INS_Address(ins) << " : " << INS_Disassemble(ins) << endl;
            }
        }
    }
    else if(INS_Opcode(ins) == XED_ICLASS_ADD && INS_Address(ins) < 0x3d930000)
    {
        if(INS_MaxNumRRegs(ins) == 1 && INS_RegWContain(ins, REG_ESP) == 0 &&
            (INS_OperandImmediate(ins, 1) & 0x00ff0000) != 0 &&
            (INS_OperandImmediate(ins, 1) & 0x00ffff00) != 0)
        {
            cout << hex << INS_Address(ins) << " : " << INS_Disassemble(ins) << endl;
        }
        else
        {
            string regRead;
            string regWrite;
            regWrite = REG_StringShort(INS_RegW(ins, 0));
            regRead = REG_StringShort(INS_RegR(ins, 0));
            if(regRead.compare(regWrite) != 0 && regRead.compare("ebp") != 0 && regWrite.compare("ebp") != 0 && regWrite.compare("esp") != 0)
            {
                cout << hex << INS_Address(ins) << " : " << INS_Disassemble(ins) << endl;
            }
        }
    }
}

else if(INS_Opcode(ins) == XED_ICLASS_SIDT || INS_Opcode(ins) == XED_ICLASS_SGDT) {
    cout << hex << INS_Address(ins) << " : " << INS_Disassemble(ins) << endl;
}
else if(INS_Opcode(ins) == XED_ICLASS_STI || INS_Opcode(ins) == XED_ICLASS_CLI) {
    cout << hex << INS_Address(ins) << " : " << INS_Disassemble(ins) << endl;
}
else if(INS_Opcode(ins) == XED_ICLASS_SUB && INS_MaxNumRegs(ins) == 1 &&
    INS_RegWContains(ins, REG_ESP) == 0 && (INS_OperandImmediate(ins, 1) & 0x0000ff00) != 0 &&
    INS_Address(ins) < 0x3d930000) {
    cout << hex << INS_Address(ins) << " : " << INS_Disassemble(ins) << endl;
}
else if(INS_Opcode(ins) == XED_ICLASS_LOOP && INS_Address(ins) < 0x3d930000) {
    cout << hex << INS_Address(ins) << " : " << INS_Disassemble(ins) << endl;
}
else if(INS_Opcode(ins) == XED_ICLASS_ROR && INS_MaxNumRegs(ins) == 1 &&
    INS_Address(ins) < 0x3d930000) {
    cout << hex << INS_Address(ins) << " : " << INS_Disassemble(ins) << endl;
}
else if(INS_IsCall() && INS_IsIndirectBranchOrCall(ins) == 0) {
    if(INS_DirectBranchOrCallTargetAddress(ins) == INS_Address(ins) + 0x5) {
        cout << hex << INS_Address(ins) << " : " << INS_Disassemble(ins) << " --> GetPC " << endl;  
    }
}
else if(INS_Opcode(ins) == XED_ICLASS_RDTSC) {
    cout << hex << INS_Address(ins) << " : " << INS_Disassemble(ins) << endl;
}
else if(INS_Opcode(ins) == XED_ICLASS_INT || INS_Opcode(ins) == XED_ICLASS_INT1 ||
    INS_Opcode(ins) == XED_ICLASS_INT3) {
    cout << hex << INS_Address(ins) << " : " << INS_Disassemble(ins) << "<-- INT instruction" << endl;
}

VOID Fini(INT32 code, VOID *v) {
    printf("Instrumentation has completed!\n");
}
INT32 Usages() {
    return -1;
}
```c
int main(int argc, char * argv[]) {
    if (PIN_Init(argc, argv))
        return Usage();

    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();

    return 0;
}
```

Now, let us run it against some of the viruses discussed previously and understand the output generated.

Below is the output from the pintool for one of the viruses:

```
401000 : call 0x401000 --> GetPC
401000 : sub ecx, 0x000000
401018 : loop 0x401142
40107e : sub dword ptr [0x418b22f], 0x2a02e82f
401090 : sub dword ptr [0x418b22c], 0x016c1347
4010a4 : sub dword ptr [0x418b22f], 0x13677b11
4010b8 : sub dword ptr [0x418b22c], 0xe412e931
4010d0 : sub dword ptr [0x418b22f], 0x376f5f4b
4010e4 : sub dword ptr [0x418b22c], 0xc6e9f8c5
4010f8 : sub dword ptr [0x418b22c], 0x01c2c3f1
401112 : sub dword ptr [0x418b22c], 0x0510b12d
401126 : sub dword ptr [0x418b22c], 0x08f12e25
401140 : sub dword ptr [0x418b22c], 0x5f3f7f73
401154 : mov ecx, 0x55
401179 : ror ecx, 0x55
401194 : cmp ecx, 0x018d954
4011aa : loop 0x4010c9
4011c6 : call 0x4010c6 --> GetPC
4011cc : sub edi, 0x406f161
4011e8 : add byte ptr [eax], bl
40120a : loop 0x401027
401226 : add byte ptr [eax], bl
401242 : loop 0x401027
40125e : call 0x40147f --> GetPC
401277 : sub edi, 0x004925
401294 : call 0x40147f --> GetPC
401309 : xor dword ptr [eax], esi
401325 : loop 0x401655
401341 : xor dword ptr [eax], esi
401357 : loop 0x401655
250050 : loop 0x255004c
250050 : loop 0x255004c
253142 : call 0x38581547 --> GetPC
253148 : sub edi, 0x404925
25314e : loop 0x255004c
25314e : loop 0x255004c
40174c : loop 0x40174c
401758 : clc
```

We have the addresses corresponding to the instructions of interest. Let us now look at the code sections which have these instructions in the debugger.
The instructions in the pintools output can be used to identify the Function Name hash generation routine as shown below:

Let us label the subroutine at 00401110 as “GetFunctionNameHash()”.

If we look up the instruction at address, 004010d4, it brings us to the subroutine used to calculate the Function Pointer.
Let us label the subroutine at address, 004010A5 as **GetFunctionPointer()**

We will look up the instruction at address, 00401176 in debugger:

If we trace the code to the shellcode at address, 0040b008 we can see that the pintool identified the decryption routine correctly.

By putting all this together we have the flow as:

1. The code manually crafts a 0x40 bytes shellcode at address 0x0040b00e using a sequence of Sub instructions.
2. It calculates the function pointer of VirtualProtect() using a precalculated function name hash and by parsing the export directory of kernel32.dll
3. It calls VirtualProtect() to mark 0x1000 bytes at address, 0x0040b00e as PAGE_EXECUTE_READWRITE since this region of code will be self modified and then executed.
4. Transfers the control flow to 0x0040b00e.
5. Uses GetPC to identify the address of code to be decrypted.
6. Uses a one byte ADD key, 0xDB to decrypt 0xA5 bytes of code and then continues executing the decrypted code.

This way, we can see how the pintool helped us quickly identify the useful sections of code. This will help us in performing an indepth analysis of the control flow of the code, to understand the packer used and the decryption routines used as well.

**Conclusion**

After reading this paper you will have an understanding of the various techniques used by viruses in the real world to obfuscate the code to deter reverse engineering.
This should help in analyzing viruses which use similar techniques as it is becomingly increasingly common for viruses to prevent the analysis of their code.

References

MSDN: http://msdn.microsoft.com/
OllyDbg: http://www.ollydbg.de/
RaiseException Reference: http://waleedassar.blogspot.in/2012/11/ollydbg-raiseexception-bug.html