

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.

0040113E	6A 6C	PUSH 6C	
00401140	50	PUSH EAX	
00401141	C745 F0 06000000	MOV DWORD PTR SS:[EBP-10],6	
00401148	C745 F4 6D000000	MOV DWORD PTR SS:[EBP-C],6D	
0040114F	C745 F8 30C54400	MOV DWORD PTR SS:[EBP-8],2e3d7d16.0044C530	UNICODE "Runtime Check"
00401156	FFD6	CALL ESI	
00401158	8D4D D0	LEA ECX,DWORD PTR SS:[EBP-30]	
0040115B	51	PUSH ECX	
0040115C	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX	
0040115F	FF15 68714000	CALL DWORD PTR DS:[407168]	USER32.RegisterClassExW
00401165	5E	POP ESI	
00401166	8BE5	MOV ESP,EBP	
00401168	5D	POP EBP	
00401169	C3	RETN	
0040116A	CC	INT3	
0040116B	CC	INT3	
0040116C	CC	INT3	
0040116D	CC	INT3	
0040116E	CC	INT3	
0040116F	CC	INT3	
00401170	55	PUSH EBP	
00401171	8BEC	MOV EBP,ESP	
00401173	51	PUSH ECX	
00401174	8B4D 08	MOV ECX,DWORD PTR SS:[EBP+8]	
00401177	8B01	MOV EAX,DWORD PTR DS:[ECX]	
00401179	8B49 04	MOV ECX,DWORD PTR DS:[ECX+4]	
0040117C	53	PUSH EBX	
0040117D	56	PUSH ESI	
0040117E	0FC8	BSWAP EAX	Window Procedure
00401180	0FC9	BSWAP ECX	
DS:[00407168]=7E41AF7F (USER32.RegisterClassExW)			
Address	Hex dump	ASCII	
0012FED4	30 00 00 00 03 00 00 00 80 26 40 00 00 00 00 00	0...*...Ç&@.....	
0012FEE4	00 00 00 00 00 00 40 00 AB 00 C4 00 11 00 01 00@.%.-.4.0.	

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:

00402680	55	PUSH EBP	
00402681	8BEC	MOV EBP,ESP	
00402683	83E4 F8	AND ESP,FFFFFFF8	
00402686	83EC 4C	SUB ESP,4C	
00402689	A1 04A04000	MOV EAX,DWORD PTR DS:[40A004]	
0040268E	33C4	XOR EAX,ESP	
00402690	894424 48	MOV DWORD PTR SS:[ESP+48],EAX	
00402694	8B45 0C	MOV EAX,DWORD PTR SS:[EBP+C]	
00402697	56	PUSH ESI	
00402698	8B75 08	MOV ESI,DWORD PTR SS:[EBP+8]	
0040269B	83F8 0F	CMP EAX,0F	←-- WM_PAINT
0040269E	✓77 75	JA SHORT 2e3d7d16.00402715	
004026A0	✓74 47	JE SHORT 2e3d7d16.004026E9	
004026A2	8BC8	MOV ECX,EAX	
004026A4	49	DEC ECX	
004026A5	✓74 1E	JE SHORT 2e3d7d16.004026C5	
004026A7	49	DEC ECX	



Read Window
Message Code

Set the Timer:

004026A8	✓75 7A	JNZ SHORT 2e3d7d16.00402724	
004026AA	51	PUSH ECX	
004026AB	FF15 44714000	CALL DWORD PTR DS:[407144]	USER32.PostQuitMessage
004026B1	33C0	XOR EAX,EAX	
004026B3	5E	POP ESI	
004026B4	8B4C24 48	MOV ECX,DWORD PTR SS:[ESP+48]	
004026B8	33CC	XOR ECX,ESP	
004026BA	E8 5D010000	CALL 2e3d7d16.0040281C	
004026BF	8BE5	MOV ESP,EBP	
004026C1	5D	POP EBP	
004026C2	C2 1000	RETN 10	
004026C5	6A 00	PUSH 0	
004026C7	68 E8030000	PUSH 3E8	
004026CC	6A 01	PUSH 1	
004026CE	56	PUSH ESI	
004026CF	FF15 5C714000	CALL DWORD PTR DS:[40715C]	USER32.SetTimer
004026D5	33C0	XOR EAX,EAX	
004026D7	5E	POP ESI	
004026D8	8B4C24 48	MOV ECX,DWORD PTR SS:[ESP+48]	

Check the Window Message Code:

00402715	8BC8	MOV ECX,EAX	
00402717	81E9 11010000	SUB ECX,111	
0040271D	✓74 57	JE SHORT 2e3d7d16.00402776	←-- WM_COMMAND
0040271F	83E9 02	SUB ECX,2	
00402722	✓74 22	JE SHORT 2e3d7d16.00402746	←-- WM_TIMER
00402724	8B55 14	MOV EDX,DWORD PTR SS:[EBP+14]	
00402727	8B4D 10	MOV ECX,DWORD PTR SS:[EBP+10]	
0040272A	52	PUSH EDX	
0040272B	51	PUSH ECX	
0040272C	50	PUSH EAX	
0040272D	56	PUSH ESI	
0040272E	FF15 58714000	CALL DWORD PTR DS:[407158]	USER32.DefWindowProcW
00402734	5E	POP ESI	
00402735	8B4C24 48	MOV ECX,DWORD PTR SS:[ESP+48]	
00402739	33CC	XOR ECX,ESP	
0040273B	E8 DC000000	CALL 2e3d7d16.0040281C	
00402740	8BE5	MOV ESP,EBP	
00402742	5D	POP EBP	
00402743	C2 1000	RETN 10	
00402746	A1 C4C64400	MOV EAX,DWORD PTR DS:[44C6C4]	
0040274B	40	INC EAX	←-- Increment Counter
0040274C	A3 C4C64400	MOV DWORD PTR DS:[44C6C4],EAX	
00402751	83F8 05	CMP EAX,5	
00402754	✓75 67	JNZ SHORT 2e3d7d16.004027BD	
00402756	E8 75FBFFFF	CALL 2e3d7d16.004022D0	←-- Call Main Subroutine if Counter == 0x5
0040275B	56	PUSH ESI	
0040275C	FF15 58714000	CALL DWORD PTR DS:[407158]	USER32.DestroyWindow
00402762	33C0	XOR EAX,EAX	
00402764	5E	POP ESI	

Below are the corresponding sections of code:

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

And here is the code rewritten in 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.

004013E1	. A3 40444000	MOV DWORD PTR DS:[404440],EAX	
004013E6	. 6A 00	PUSH 0	
004013E8	. 68 0D114000	PUSH c92c7f70.00401100	lParam = NULL
004013ED	. 6A 00	PUSH 0	DlgProc = c92c7f70.00401100 <-- Callback Function
004013EF	. 68 E0300000	PUSH 3E8	hOwner = NULL
004013F4	. FF35 40444000	PUSH DWORD PTR DS:[404440]	pTemplate = 3E8
004013FA	. FF15 54304000	CALL DWORD PTR DS:[&USER32.DialogBoxPa	hInst = NULL
00401400	. 50	PUSH EAX	DialogBoxParamA
00401401	. FF15 10304000	CALL DWORD PTR DS:[&KERNEL32.ExitProce	ExitCode
			ExitProcess

00401100	. 55	PUSH EBP	
0040110E	. 8BEC	MOV EBP,ESP	
00401110	. 8B45 0C	MOV EAX,DWORD PTR SS:[EBP+C] <-- Read Window Message Code from Stack	
00401113	. 3D 10010000	CMP EAX,110	
00401118	..0F85 4C010000	JNZ c92c7f70.0040126A <-- if(wind_code) == WM_INITDIALOG	
0040111E	. 8B45 08	MOV EAX,DWORD PTR SS:[EBP+8]	
00401121	. A3 44444000	MOV DWORD PTR DS:[404444],EAX	
00401126	. 6A 3C	PUSH 3C	
00401128	. 68 50444000	PUSH c92c7f70.00404450	
0040112D	. E8 B5FFFFFF	CALL c92c7f70.004010E7	
00401132	. B8 6C444000	MOV EAX,c92c7f70.0040446C	
00401137	. 68 34444000	PUSH c92c7f70.00404434	
0040113C	. 50	PUSH EAX	
0040113D	. FF15 40304000	CALL DWORD PTR DS:[&&KERNEL32.lstrcpyA]	String2 = "Arial" String1 => c92c7f70.0040446C lstrcpyA
00401143	. C705 50444000	MOV DWORD PTR DS:[404450],12	

0040126A	> 3D 11010000	CMP EAX,111	
0040126F	..75 2D	JNZ SHORT c92c7f70.0040129E <-- if(wind_code) == WM_COMMAND	
00401271	. B8 9C444000	MOV EAX,c92c7f70.0040449C	
00401276	. 50	PUSH EAX	
00401277	. 50	PUSH EAX	
00401278	. FF15 1C304000	CALL DWORD PTR DS:[&&KERNEL32.GetSystemTime]	pSystemTime => c92c7f70.0040449C GetSystemTime
0040127E	. 5A	POP EDI	
0040127F	. 33C0	XOR EAX,EAX	
00401281	. 66:8B42 02	MOV AX,WORD PTR DS:[EDI+2] <-- Store current month in AX to perform timing checks	
00401285	. 05 FB140000	ADD EAX,c92c7f70.004041FB	
0040128A	. FF35 3C304000	PUSH DWORD PTR DS:[&&KERNEL32.GetModuleHandleA]	kerne l32.GetModuleHandleA
00401290	. 68 A9154000	PUSH c92c7f70.004015A9	Entry address
00401295	. E8 DE050000	CALL c92c7f70.00401878 <-- Call Main Subroutine	
0040129A	. 5D	POP EBP	
0040129B	. C2 1000	RETN 10	

EnumSystemLocalesA():

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:

0040115A	FF15 1D104000	CALL DWORD PTR DS:[40101D]	kerne l32.GetVolumeInformationA
00401160	817D 00 401ACD00	CMP DWORD PTR SS:[EBP],0CD1A40	
00401167	74 09	JE SHORT gutfifjh.00401172	
00401169	817D 00 46461470	CMP DWORD PTR SS:[EBP],70144646	
00401170	75 05	JNZ SHORT gutfifjh.00401177	
00401172	E9 DA010000	JMP gutfifjh.00401351	
00401177	892D 434E4000	MOV DWORD PTR DS:[404E43],EBP	
0040117D	6A 02	PUSH 2 <-- dwFlags parameter of EnumSystemLocalesA	
0040117F	E8 0E000000	CALL gutfifjh.00401192 <-- This Call will Push the Return Address (0x401184) on the Stack	
00401184	8B2D 434E4000	MOV EBP,DWORD PTR DS:[404E43] <-- Address of Callback Function	
0040118A	68 9D114000	PUSH gutfifjh.0040119D	
0040118F	58	POP EAX	
00401190	FFD0	CALL EAX <-- Call 40119d (eax is set to this value using Push/Pop)	
00401192	FF15 2D104000	CALL DWORD PTR DS:[40102D]	kerne l32.EnumSystemLocalesA
00401198	E8 B4010000	CALL gutfifjh.00401351	
0040119D	58	POP EAX <-- Execution is Resumed Here	
0040119E	E8 04020000	CALL gutfifjh.004013A7 <-- Call 4013A7 which is in between two assembled instructions	
004011A3	83F8 00	CMP EAX,0	
004011A6	74 05	JE SHORT gutfifjh.004011AD	
004011A8	E9 A4010000	JMP gutfifjh.00401351	
004011AD	E8 08000000	CALL gutfifjh.004011BA	

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.

0040174C	34 43	XOR AL,43	<i><-- Single Byte XOR Decryption of the Exception Handler</i>
0040174E	EB 0A	JMP SHORT gvtfifjh.0040175A	
00401750	EB FA	JMP SHORT gvtfifjh.0040174C	
00401752	8D844B 52174000	LEA EAX,DWORD PTR DS:[EBX+ECX*2+401752]	
00401759	7A AA	JPE SHORT gvtfifjh.00401705	
0040175B	E2 E4	LOOPD SHORT gvtfifjh.00401741	
0040175D	FA	CLI	
0040175A	AA	STOS BYTE PTR ES:[EDI]	<i><-- Write the decrypted byte to the Exception Handler code</i>
0040175B	E2 E4	LOOPD SHORT gvtfifjh.00401741	
0040175D	FA	CLI	<i><-- Trigger the 0xc0000096 exception here</i>
0040175E	4C	DEC ESP	
0040175F	017416 1C	ADD DWORD PTR DS:[ESI+EDX+1C],ESI	

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.

0006FFC0	00401610	SE handler	<i><-- Exception Handler registered at top of stack</i>
0006FFC4	7C817077	RETURN to kernel[32.7C817077]	
0006FFC8	7C910228	ntdll.7C910228	
0006FFCC	FFFFFFFF		
0006FFD0	7FFD8000		
0006FFD4	80544CFD		

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.

00401610	E8 04000000	CALL gvtfifjh.00401619	<-- Resume Execution from the Exception Handler
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	

Execution through Vectored Exception Handler:

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

00401ABE	. 55	PUSH EBP	
00401ABF	. 8BEC	MOV EBP,ESP	
00401AC1	. 53	PUSH EBX	
00401AC2	. 8B45 00	MOV EAX,DWORD PTR SS:[EBP+8]	EXCEPTION_POINTERS
00401AC5	. 8B58 04	MOV EBX,DWORD PTR DS:[EAX+4]	CONTEXT_RECORD
00401AC8	. 8B00	MOV EAX,DWORD PTR DS:[EAX]	EXCEPTION_RECORD
00401ACA	. 8138 05000000	CMPL DWORD PTR DS:[EAX],C0000005	Check for memory access violation
00401AD0	~> 75 34	JNZ SHORT virus.00401B06	
00401AD2	. 8178 0C ED1E4	CMPL DWORD PTR DS:[EAX+C],virus.00401EED	Check the faulting instruction address
00401AD9	~> 75 2B	JNZ SHORT virus.00401B06	
00401ADB	. 8B93 C4000000	MOV EDX,DWORD PTR DS:[EBX+C4]	Set ESP in Context Structure
00401AE1	. C702 40000000	MOV DWORD PTR DS:[EDX],40	
00401AE7	. C742 04 B0204	MOV DWORD PTR DS:[EDX+4],virus.004020B0	
00401AEE	. C742 08 A41A4	MOV DWORD PTR DS:[EDX+8],virus.00401AA4	
00401AF5	. C783 B8000000	MOV DWORD PTR DS:[EBX+B8],virus.00401F03	Set EIP in Context Structure
00401AFF	. B8 FFFFFFFF	MOV EAX,-1	
00401B04	~> EB 05	JMP SHORT virus.00401B0B	
00401B06	> B8 00000000	MOV EAX,0	
00401B0B	> 5B	POP EBX	
00401B0C	. C9	LEAVE	
00401B0D	. C2 0400	RETN 4	

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.

0006F324	00401240	CALL to RaiseException
0006F328	80000003	ExceptionCode = 80000003
0006F32C	00000000	ExceptionFlags = EXCEPTION_CONTINUE
0006F330	00000001	nArguments = 1
0006F334	00000000	pArguments = NULL
0006F338	0006F83C	Pointer to next SEH record
0006F33C	0040126D	SE handler

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.

0040B62C	55	PUSH EBP	
0040B62D	8BEC	MOV EBP,ESP	
0040B62F	8D35 22B64000	LEA ESI,DWORD PTR DS:[40B622]	
0040B635	8DB6 9AB54000	LEA ESI,DWORD PTR DS:[ESI+40B59A]	
0040B63B	55	PUSH EBP	
0040B63C	B8 6A000000	MOV EAX,6A	
0040B641	05 FBB54000	ADD EAX,9cd951bb.0040B5F8	<-- Calculate Address of Instruction after Faulting Instruction
0040B646	50	PUSH EAX	
0040B647	05 3DFFFFFF	ADD EAX,-0C3	
0040B64C	50	PUSH EAX	<-- Calculate Address of Exception Handler
0040B64D	B8 78FFFFFF	MOV EAX,-88	
0040B652	BB 22B64000	MOV EBX,9cd951bb.0040B622	
0040B657	03DB	ADD EBX,EBX	
0040B659	03C3	ADD EAX,EBX	
0040B65B	2BF0	SUB ESI,EAX	<-- Set ESI to 0x0
0040B65D	64:FF36	PUSH DWORD PTR FS:[ESI]	
0040B660	64:8926	MOV DWORD PTR FS:[ESI],ESP	<-- Register the Exception Handler
0040B663	F7F6	DIV ESI	<-- Trigger the Exception
0040B665	64:8F05 000000	POP DWORD PTR FS:[0]	
0040B66C	83C4 0C	ADD ESP,0C	
0040B66F	EB 03	JMP SHORT 9cd951bb.0040B674	
0040B671	58	POP EAX	
0040B672	EB 05	JMP SHORT 9cd951bb.0040B679	
0040B674	E8 F8FFFFFF	CALL 9cd951bb.0040B671	
0040B679	05 1DFFFFFF	ADD EAX,-0E3	

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)

0040B5A2	55	PUSH EBP	
0040B5A3	8BEC	MOV EBP,ESP	
0040B5A5	60	PUSHAD	
0040B5A6	8B45 0C	MOV EAX,DWORD PTR SS:[EBP+C]	<-- EstablisherFrame
0040B5A9	8B75 10	MOV ESI,DWORD PTR SS:[EBP+10]	<-- Context Record
0040B5AC	8B50 08	MOV EDX,DWORD PTR DS:[EAX+8]	
0040B5AF	8996 B8000000	MOV DWORD PTR DS:[ESI+B8],EDX	<-- Update Context Record with address to resume execution from
0040B5B5	8B50 0C	MOV EDX,DWORD PTR DS:[EAX+C]	
0040B5B8	8996 B4000000	MOV DWORD PTR DS:[ESI+B4],EDX	
0040B5BE	8986 C4000000	MOV DWORD PTR DS:[ESI+C4],EAX	
0040B5C4	61	POPAD	
0040B5C5	B8 00000000	MOV EAX,0	
0040B5CA	C9	LEAVE	
0040B5CB	C2 1000	RETN 10	

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:
pop eax
sub eax, offset label3
lea esi, dword ptr [eax+label3]
lea edi, dword ptr [eax+label4]
mov ecx, 1
rep movs byte ptr [edi], byte ptr [esi]
label4: 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:

00401000	83E1 FF	AND ECX,FFFFFFFF
00401003	83E8 00	SUB EAX,0
00401006	8BC9	MOV ECX,ECX
00401008	83EB 00	SUB EBX,0
0040100B	90	NOP
0040100C	60	PUSHAD
0040100D	61	POPAD
0040100E	90	NOP
0040100F	60	PUSHAD
00401010	61	POPAD
00401011	90	NOP
00401012	83E8 00	SUB EAX,0
00401015	9C	PUSHFD
00401016	9D	POPFD
00401017	8BF6	MOV ESI,ESI
00401019	90	NOP
0040101A	90	NOP
0040101B	83F1 00	XOR ECX,0
0040101E	83E3 FF	AND EBX,FFFFFFFF
00401021	90	NOP
00401022	90	NOP
00401023	52	PUSH EDX
00401024	5A	POP EDX
00401025	9C	PUSHFD
00401026	9D	POPFD
00401027	83EB 00	SUB EBX,0
0040102A	52	PUSH EDX
0040102B	5A	POP EDX
0040102C	83F1 00	XOR ECX,0
0040102F	52	PUSH EDX
00401030	5A	POP EDX
00401031	8BC0	MOV EAX,EAX
00401033	83EB 00	SUB EBX,0
00401036	52	PUSH EDX
00401037	5A	POP EDX
00401038	EB 00	JMP SHORT 1e952b6.0040103A
0040103A	90	NOP

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:

00435D60	55	PUSH EBP	
00435D61	8BEC	MOV EBP,ESP	
00435D63	83EC 50	SUB ESP,50	
00435D66	A1 C0234400	MOV EAX,DWORD PTR DS:[&GDI32.GetStockObject]	
00435D6B	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX	
00435D6E	833D D4274400	CMPL DWORD PTR DS:[4427D4],0	
00435D75	75 15	JNZ SHORT 2e3d7d16.00435D8C	<<-- This jump will never occur since Z flag is set before it
00435D77	68 19100000	PUSH 1019	
00435D7C	6A 00	PUSH 0	
00435D7E	FF15 AC234400	CALL DWORD PTR DS:[&USER32.LoadIconA]	[RsrcName = 4121. hInst = NULL LoadIconA
00435D84	85C0	TEST EAX,EAX	
00435D86	74 04	JE SHORT 2e3d7d16.00435D8C	<<-- This jump will always take place since LoadIconA returns 0x0
00435D88	33C0	XOR EAX,EAX	
00435D8A	EB 34	JMP SHORT 2e3d7d16.00435D8C	
00435D8C	FF15 40234400	CALL DWORD PTR DS:[&KERNEL32.GetCurrentThread]	[GetCurrentThread
00435D92	83F8 FE	CMPL EAX,-2	
00435D95	74 04	JE SHORT 2e3d7d16.00435D9B	<<-- This jump will always take place
00435D97	33C0	XOR EAX,EAX	
00435D99	EB 25	JMP SHORT 2e3d7d16.00435D8C	
00435D9B	68 22000100	PUSH 10022	UNICODE "RPROFILE=C:\Documents and Settings\All Use
00435DA0	FF55 FC	CALL DWORD PTR SS:[EBP-4]	<<-- Call GetStockObject
00435DA3	85C0	TEST EAX,EAX	
00435DA5	75 11	JNZ SHORT 2e3d7d16.00435DB8	<<-- This jump will never take place since Z flag is set before it
00435DA7	B9 00104000	MOV ECX,2e3d7d16.00401000	
00435DAC	81E9 15160500	SUB ECX,51615	<<-- This value (0x51615) will be added back to eax once we return
00435DB2	890D D8274400	MOV DWORD PTR DS:[4427D8],ECX	
00435DB8	A1 D8274400	MOV EAX,DWORD PTR DS:[4427D8]	
00435DBD	83C0 04	ADD EAX,4	
00435DC0	8BE5	MOV ESP,EBP	
00435DC2	5D	POP EBP	
00435DC3	C3	RET	

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:

00435E10	> 55	PUSH EBP	
00435E11	. 8BEC	MOV EBP,ESP	
00435E13	. 83EC 14	SUB ESP,14	
00435E16	. C705 E0274400	MOV DWORD PTR DS:[4427E0],0CC79	
00435E20	. A1 E0274400	MOV EAX,DWORD PTR DS:[4427E0]	
00435E25	. 2D 79C00000	SUB EAX,0CC79	<-- Set eax to 0x0
00435E2A	. A3 E0274400	MOV DWORD PTR DS:[4427E0],EAX	
00435E2F	. FF15 44234400	CALL DWORD PTR DS:[&KERNEL32.GetOEMCP]	[GetOEMCP
00435E35	. 85C0	TEST EAX,EAX	
00435E37	. 0F84 E4000000	JE 2e3d7d16.00435F21	
00435E3D	. FF15 8C214400	CALL DWORD PTR DS:[44218C]	[GetCurrentThread
00435E43	. 83F8 FE	CMP EAX,-2	
00435E46	. 0F85 D5000000	JNZ 2e3d7d16.00435F21	<-- This jump will never take place
00435E4C	. 68 83841700	PUSH 178483	[ObjType = 1541251.
00435E51	. FF15 90214400	CALL DWORD PTR DS:[442190]	[GetStockObject
00435E57	. 85C0	TEST EAX,EAX	
00435E59	. 0F85 C2000000	JNZ 2e3d7d16.00435F21	
00435E5F	. 6A 00	PUSH 0	[ObjType = WHITE_BRUSH
00435E61	. FF15 90214400	CALL DWORD PTR DS:[442190]	[GetStockObject
00435E67	. 85C0	TEST EAX,EAX	
00435E69	. 0F84 B2000000	JE 2e3d7d16.00435F21	<-- This jump will never take place
00435E6F	. 68 755E4300	PUSH 2e3d7d16.00435E75	
00435E74	. C3	RETN	RET used as a jump to 00435E75
00435E75	> 8B0D E0274400	MOV ECX,DWORD PTR DS:[4427E0]	<-- PUSH/RET instruction above will redirect execution here
00435E7B	. 3B4D 0C	CMP ECX,DWORD PTR SS:[EBP+C]	
00435E7E	. 72 05	JB SHORT 2e3d7d16.00435E85	
00435E80	. E9 9C000000	JMP 2e3d7d16.00435F21	
00435E85	> C705 80214400	MOV DWORD PTR DS:[442188],0D13A	
00435E8F	. 8B55 08	MOV EDX,DWORD PTR SS:[EBP+8]	
00435E92	. 0315 E0274400	ADD EDX,DWORD PTR DS:[4427E0]	
00435E98	. 8B02	MOV EAX,DWORD PTR DS:[EDX]	

1. 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.
2. 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:

004360A3	> A1 EC274400	MOV EAX,DWORD PTR DS:[4427EC]	
004360A8	. A3 00284400	MOV DWORD PTR DS:[442800],EAX	
004360AD	. C705 04284400	MOV DWORD PTR DS:[442804],23C912	
004360B7	. 8B0D 04284400	MOV ECX,DWORD PTR DS:[442804]	
004360BD	. 81E9 12C92300	SUB ECX,23C912	<-- Set ecx to 0x0
004360C3	. 890D 04284400	MOV DWORD PTR DS:[442804],ECX	
004360C9	. 8B15 04284400	MOV EDX,DWORD PTR DS:[442804]	
004360CF	. 8915 08284400	MOV DWORD PTR DS:[442808],EDX	
004360D5	> A1 04284400	MOV EAX,DWORD PTR DS:[442804]	<-- Set eax to 0x0
004360DA	. 3B05 EC274400	CMP EAX,DWORD PTR DS:[4427EC]	
004360E0	. 0F83 82000000	JNB 2e3d7d16.00436168	<-- This jump will never take place
004360E6	. 8B0D 00284400	MOV ECX,DWORD PTR DS:[442800]	
004360EC	. 51	PUSH ECX	
004360ED	. 8B15 C0204400	MOV EDX,DWORD PTR DS:[4420C0]	
004360F3	. 52	PUSH EDX	
004360F4	. E8 17030000	CALL 2e3d7d16.00436410	<-- Call the function with necessary arguments
004360F9	. 83C4 08	ADD ESP,8	
004360FC	. A3 0C284400	MOV DWORD PTR DS:[44280C],EAX	
00436101	. A1 0C284400	MOV EAX,DWORD PTR DS:[44280C]	

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:



Address	Disassembly	Comment
00401619	68 20164000	PUSH gutfifjh.00401620
0040161E	EB 04	JMP SHORT gutfifjh.00401624 <-- EIP
00401620	BA DCEE0059	MOV EDX,5900EEDC
00401625	EB 05	JMP SHORT gutfifjh.0040162C
00401627	EB 0B	JMP SHORT gutfifjh.004015B4
00401629	09EB	OR EBX,EBP
0040162B	03EB	ADD EBP,EBX
0040162D	FA	CLI
0040162E	74 58	JE SHORT gutfifjh.00401688

Address	Disassembly	Comment
00401624	59	POP ECX
00401625	EB 05	JMP SHORT gutfifjh.0040162C <-- EIP
00401627	EB 0B	JMP SHORT gutfifjh.004015B4
00401629	09EB	OR EBX,EBP
0040162B	03EB	ADD EBP,EBX
0040162D	FA	CLI
0040162E	74 58	JE SHORT gutfifjh.00401688

Address	Disassembly	Comment
0040162C	EB FA	JMP SHORT gutfifjh.00401628 <-- EIP
0040162E	74 58	JE SHORT gutfifjh.00401688
00401630	EB 05	JMP SHORT gutfifjh.00401637

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.

00401610	. E8 04000000	CALL gutfifjh.00401619	
00401615	BA	DB BA	<-- Junk bytes injected
00401616	DC	DB DC	
00401617	. FE00	INC BYTE PTR DS:[EAX]	
00401619	. 68 20164000	PUSH gutfifjh.00401620	<-- Correct Disassembly by tracing the control flow
0040161E	EB 04	JMP SHORT gutfifjh.00401624	<-- Incorrect Disassembly
00401620	? BA DCEE0059	MOV EDX,5900EEDC	
00401625	? EB 05	JMP SHORT gutfifjh.0040162C	
00401627	? EB 0B	JMP SHORT gutfifjh.004015B4	
00401629	? 09EB	OR EBX,EBP	
0040162B	. 03EB	ADD EBP,EBX	<-- Disassembly errors continue
0040162D	. FA	CLI	
0040162E	74 58	JE SHORT gutfifjh.00401688	
00401630	EB 05	JMP SHORT gutfifjh.00401637	
00401632	EB	DB EB	
00401633	> 8B00	MOV EAX,DWORD PTR DS:[EAX]	
00401635	EB 03	JMP SHORT gutfifjh.0040163A	
00401637	> EB FA	JMP SHORT gutfifjh.00401633	
00401639	74	DB 74	CHAR 't'
0040163A	EB 05	JMP SHORT gutfifjh.00401641	
0040163C	75	DB 75	CHAR 'u'
0040163D	> 29C8	SUB EAX,ECX	
0040163F	EB 04	JMP SHORT gutfifjh.00401645	
00401641	> EB FA	JMP SHORT gutfifjh.0040163D	
00401643	0F	DB 0F	
00401644	85	DB 85	
00401645	> C1E0 08	SHL EAX,8	
00401648	EB 07	JMP SHORT gutfifjh.00401651	
0040164A	3B	DB 3B	CHAR ';'
0040164B	05	DB 05	
0040164C	4A	DB 4A	CHAR 'J'
0040164D	16	DB 16	
0040164E	40	DB 40	CHAR '@'
0040164F	00	DB 00	
00401650	7D	DB 7D	CHAR ')'

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,EB098BEB
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 004015BF
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
00401649 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
00401619 68 20164000 PUSH 00401620
0040161E EB 04 JMP SHORT 00401624
00401624 59 POP ECX
00401625 EB 05 JMP SHORT 0040162C
0040162C ^EB FA JMP SHORT 00401628
00401628 8B09 MOV ECX,DWORD PTR DS:[ECX]
0040162A EB 03 JMP SHORT 0040162F
0040162F 58 POP EAX
00401630 EB 05 JMP SHORT 00401637
00401637 ^EB FA JMP SHORT 00401633
00401633 8B00 MOV EAX,DWORD PTR DS:[EAX]
00401635 EB 03 JMP SHORT 0040163A
0040163A EB 05 JMP SHORT 00401641
00401641 ^EB FA JMP SHORT 0040163D
0040163D 29C8 SUB EAX,ECX
0040163F EB 04 JMP SHORT 00401645
00401645 C1E0 08 SHL EAX,8
00401648 EB 07 JMP SHORT 00401651
00401651 6A 03 PUSH 3
```

```
00401653  0FC8          BSWAP EAX
00401655  EB 05         JMP SHORT 0040165C
```

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) ^ 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 && regRead.compare("esp") != 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 || INS_Opcode(ins) == XED_ICLASS_SLDT)
    {
        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_MaxNumRRegs(ins) == 1 &&
INS_RegWContain(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_CMP && INS_MaxNumRRegs(ins) == 1 &&
INS_Size(ins) > 0x3 && INS_IsMemoryRead(ins) == 0 && (INS_OperandImmediate(ins,
1) & 0xff000000) != 0 && ((INS_OperandImmediate(ins, 1) & 0x00ffff00) ^
0x00ffff00) != 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_MaxNumRRegs(ins) == 1 &&
INS_Address(ins) < 0x3d930000)
    {
        cout << hex << INS_Address(ins) << " : " << INS_Disassemble(ins) << endl;
    }
    else if(INS_IsCall(ins) && 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 Usage()
{
    return -1;
}

```

```

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:

```

40108b : call 0x401090 --> GetPC      <-- GetPC used to XOR ecx
401091 : sub ecx, 0x401090
401158 : loop 0x401142
401176 : sub dword ptr [0x40b028], 0x3a6e82fa
401180 : sub dword ptr [0x40b00c], 0x616e3347
40118a : sub dword ptr [0x40b024], 0x13677b11
401194 : sub dword ptr [0x40b020], 0x643c4351
40119e : sub dword ptr [0x40b010], 0x33665bd2 <-- Write 0x40 bytes corresponding to self
4011a8 : sub dword ptr [0x40b02c], 0x6c9f8e5c   <-- modifying shellcode at address, 0x40b008
4011b2 : sub dword ptr [0x40b01c], 0x1abc58b1
4011bc : sub dword ptr [0x40b018], 0x5b1d812d
4011c6 : sub dword ptr [0x40b014], 0x6f120a5a
4011d0 : sub dword ptr [0x40b008], 0x5f5fd7f3
401120 : ror ecx, 0x5
401120 : ror ecx, 0x5      <-- Correspond to Function Name Hash Generation
4010d4 : cmp eax, 0xf18d954
40110a : loop 0x4010c9
40b009 : call 0x40b00e --> GetPC <-- Self modifying code which uses an ADD Key to
40b00f : sub edi, 0x406061   <-- decrypt the code
40b027 : add byte ptr [eax], b1
40b02a : loop 0x40b027
40b027 : add byte ptr [eax], b1
40b02a : loop 0x40b027
40b476 : call 0x40b47b --> GetPC
40b47c : sub edi, 0x404925
40b142 : call 0x40b147 --> GetPC
40b65b : xor dword ptr [eax], esi
40b65e : loop 0x40b656
40b65b : xor dword ptr [eax], esi
40b65e : loop 0x40b656
2a50050 : loop 0x2a5004c
2a50050 : loop 0x2a5004c
2a50342 : call 0x2a50347 --> GetPC
2a50348 : sub edi, 0x404925
2a5046a : loop 0x2a50462
2a5046a : loop 0x2a50462
40175b : loop 0x401741
40175d : cli      <-- Use of a Privileged Instruction to trigger Exception

```

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.

00401176	. 812D 28B04000	SUB DWORD PTR DS:[40B028],3A6E82FA	
00401180	. 812D 0CB04000	SUB DWORD PTR DS:[40B00C],616E3347	
0040118A	. 812D 24B04000	SUB DWORD PTR DS:[40B024],13677B11	<- Write 0x40 bytes corresponding to self modifying shellcode at 0040b008
00401194	. 812D 20B04000	SUB DWORD PTR DS:[40B020],643C4351	
0040119E	. 812D 10B04000	SUB DWORD PTR DS:[40B010],33665B02	
004011A8	. 812D 2CB04000	SUB DWORD PTR DS:[40B02C],6C9F8E5C	
004011B2	. 812D 1CB04000	SUB DWORD PTR DS:[40B01C],1ABC58B1	
004011BC	. 812D 18B04000	SUB DWORD PTR DS:[40B018],5B1D812D	
004011C6	. 812D 14B04000	SUB DWORD PTR DS:[40B014],6F120A5A	
004011D0	. 812D 08B04000	SUB DWORD PTR DS:[40B008],5F5FD7F3	
004011DA	. 60	PUSHAD	
004011DB	. E8 C5FEFFFF	CALL gvtfifjh.004010A5	
004011E0	. 68 04B04000	PUSH gvtfifjh.0040B004	
004011E5	. 6A 40	PUSH 40	
004011E7	. 68 00100000	PUSH 1000	
004011EC	. 68 08B04000	PUSH gvtfifjh.0040B008	
004011F1	. FFD0	CALL EAX	
004011F3	. 61	POPAD	
004011F4	.-E9 0F9E0000	JMP gvtfifjh.0040B008	<- Transfer control to the Shellcode

The instructions in the pintools output can be used to identify the Function Name hash generation routine as shown below:

00401110	55	PUSH EBP	
00401111	. 8BEC	MOV EBP,ESP	
00401113	. 56	PUSH ESI	
00401114	. 8B75 08	MOV ESI,DWORD PTR SS:[EBP+8]	<- Function Name is passed as an argument
00401117	. 33C9	XOR ECX,ECX	
00401119	> AC	LODS BYTE PTR DS:[ESI]	<- Read a byte from the Function Name
0040111A	. 0AC0	OR AL,AL	
0040111C	. 74 07	JE SHORT gvtfifjh.00401125	
0040111E	. 32C8	XOR CL,AL	<- Function Name hash will be stored in ECX
00401120	. C1C9 05	ROR ECX,5	
00401123	. EB F4	JMP SHORT gvtfifjh.00401119	
00401125	> 8BC1	MOV EAX,ECX	<- Return the Function Name Hash
00401127	. 5E	POP ESI	
00401128	. C9	LEAVE	
00401129	. C2 0400	RETN 4	

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.

004010A5	53	PUSH EBX	
004010A6	. 56	PUSH ESI	
004010A7	. 57	PUSH EDI	
004010A8	. 64:A1 30000000	MOV EAX,DWORD PTR FS:[30]	
004010AE	. 8B40 0C	MOV EAX,DWORD PTR DS:[EAX+C]	
004010B1	. 8B70 1C	MOV ESI,DWORD PTR DS:[EAX+1C]	
004010B4	. AD	LODS DWORD PTR DS:[ESI]	
004010B5	. 8B58 08	MOV EBX,DWORD PTR DS:[EAX+8]	
004010B8	. 8B7B 3C	MOV EDI,DWORD PTR DS:[EBX+3C]	
004010BB	. 8B7C1F 78	MOV EDI,DWORD PTR DS:[EDI+EBX+78]	
004010BF	. 03FB	ADD EDI,EBX	
004010C1	. 8B4F 18	MOV ECX,DWORD PTR DS:[EDI+18]	
004010C4	. 8B77 20	MOV ESI,DWORD PTR DS:[EDI+20]	
004010C7	. 03F8	ADD ESI,EBX	
004010C9	> 60	PUSHAD	
004010CA	. 8B36	MOV ESI,DWORD PTR DS:[ESI]	
004010CC	. 03F8	ADD ESI,EBX	
004010CE	. 56	PUSH ESI	
004010CF	. E8 3C000000	CALL <gvtfifjh.GetFunctionNameHash>	
004010D4	. 3D 54D9180F	CMP EAX,0F18D954	<- Compare with the Hash of VirtualProtect()
004010D9	. 75 2B	JNZ SHORT gvtfifjh.00401106	
004010DB	. 61	POPAD	
004010DC	. 8B57 20	MOV EDX,DWORD PTR DS:[EDI+20]	
004010DF	. 03D3	ADD EDX,EBX	
004010E1	. 2BF2	SUB ESI,EDX	
004010E3	. D1EE	SHR ESI,1	
004010E5	. 8B47 24	MOV EAX,DWORD PTR DS:[EDI+24]	
004010E8	. 03C3	ADD EAX,EBX	
004010EA	. 03C6	ADD EAX,ESI	
004010EC	. B9 00000000	MOV ECX,0	
004010F1	. 66:8B08	MOV CX,WORD PTR DS:[EAX]	
004010F4	. C1E1 02	SHL ECX,2	
004010F7	. 8B47 1C	MOV EAX,DWORD PTR DS:[EDI+1C]	
004010FA	. 03C3	ADD EAX,EBX	
004010FC	. 03C1	ADD EAX,ECX	<- Return the Function Pointer of VirtualProtect()
004010FE	. 8B00	MOV EAX,DWORD PTR DS:[EAX]	

Let us label the subroutine at address, 004010A5 as **GetFunctionPointer()**

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

00401176	. 812D 28B04000	SUB DWORD PTR DS:[40B028],3A6E82FA	
00401180	. 812D 0CB04000	SUB DWORD PTR DS:[40B00C],616E3347	
0040118A	. 812D 24B04000	SUB DWORD PTR DS:[40B024],13677B11	
00401194	. 812D 20B04000	SUB DWORD PTR DS:[40B020],643C4351	
0040119E	. 812D 10B04000	SUB DWORD PTR DS:[40B010],336658D2	
004011A8	. 812D 2CB04000	SUB DWORD PTR DS:[40B02C],6C9F8E5C	
004011B2	. 812D 1CB04000	SUB DWORD PTR DS:[40B01C],1ABC58B1	<-- Write the 0x40 bytes of shellcode at 0040b008
004011BC	. 812D 18B04000	SUB DWORD PTR DS:[40B018],5B1D812D	
004011C6	. 812D 14B04000	SUB DWORD PTR DS:[40B014],6F120A5A	
004011D0	. 812D 08B04000	SUB DWORD PTR DS:[40B008],5F5FD7F3	
004011DA	. 60	PUSHAD	
004011DB	. E8 C5FEFFFF	CALL <gutfifjh.GetFunctionPointer>	<-- Returns Function Pointer of VirtualProtect()
004011E0	. 68 04B04000	PUSH gutfifjh.0040B004	
004011E5	. 6A 40	PUSH 40	<-- Mark 0x1000 bytes in self modifying code region as PAGE_EXECUTE_READWRITE
004011E7	. 68 00100000	PUSH 1000	
004011EC	. 68 08B04000	PUSH gutfifjh.0040B008	
004011F1	. FFD0	CALL EAX	kernel32.VirtualProtect
004011F3	. 61	POPAD	
004011F4	.-E9 0F9E0000	JMP gutfifjh.0040B008	<-- Control Flow transfer to the Shellcode

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

0040B008	60	PUSHAD	
0040B009	E8 00000000	CALL gutfifjh.0040B00E	<-- GetPC
0040B00E	5F	POP EDI	
0040B00F	81EF 61604000	SUB EDI,gutfifjh.00406061	
0040B015	8D05 83604000	LEA EAX,DWORD PTR DS:[406083]	
0040B01B	03C7	ADD EAX,EDI	
0040B01D	B9 550A0000	MOV ECX,0A55	<-- 0xA55 bytes of code will be decrypted
0040B022	BB DB000000	MOV EBX,0DB	
0040B027	0018	ADD BYTE PTR DS:[EAX],BL	<-- Decryption Routine identified by the pintool
0040B029	40	INC EAX	
0040B02A	^E2 FB	LOOPD SHORT gutfifjh.0040B027	
0040B02C	61	POPAD	
0040B02D	90	NOP	

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 0xA55 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

Pintool: <http://software.intel.com/sites/landingpage/pintool/docs/49306/Pin/html/>

MSDN: <http://msdn.microsoft.com/>

OllyDbg: <http://www.ollydbg.de/>

RaiseException Reference: <http://waleedassar.blogspot.in/2012/11/ollydbg-raiseexception-bug.html>