# **Breaking the Sandbox**

### **Author: Sudeep Singh**

### **Introduction**

In this paper, I would like to discuss various existing and interesting techniques which are used to evade the detection of a virus in Sandbox. We will also look at ways a sandbox can be hardened to prevent such evasion techniques.

This paper is targeted towards those who have an experience with Windows OS internals, reverse engineering viruses as well as those who are interested in developing detection mechanisms for viruses in a Sandboxed environment.

A deep understanding of the evasion techniques used by viruses in the wild helps us in implementing better detection mechanisms.

### <u>Purpose</u>

New offensive techniques give rise to innovative detection mechanisms, as has always been the case in cyber security.

Nowadays, it is becoming increasingly common for malware analysts to leverage sandboxes for automation of malware analysis. However, most techniques used in viruses to evade such sandboxes are targeted towards commercial and well-known VMs like VMWare Workstation, VMWare Fusion, Virtual Box, Virtual PC, Qemu and some sandboxes like CWSandbox, Anubis and so on. These techniques can prevent analysis in a sandbox based only on known information.

However, there is a lot of scope for improvement in the techniques used by viruses to detect and evade a Virtualized Environment. This encourages the development of more hardened sandboxes which are as close to VMM transparency as possible.

# Anti VM techniques

Before we discuss some new ways of preventing analysis in sandboxes, let us discuss some of the commonly used anti sandbox/anti vm techniques.

**Process Names:** Till date, viruses rely on looking for known process names running on the machine to identify the presence of a virtual machine. By enumerating the process names using Process32First()/Process32Next(), they look for running process such as vmsrvc.exe,

vmusrvc.exe, vboxtray.exe, vmtoolsd.exe, df5serv.exe, vboxservice.exe and so on. These are known values of process names that may be running in a sandbox used by malware analysts.

**Registry Artifacts:** It is also common for viruses to access specific registry entries to check for presence of sandboxes. One of the most common registry entry accessed for this purpose is:

HKEY\_LOCAL\_MACHINE\SYSTEM\ControlSet001\Services\Disk\Enum

Subkey: "0"

Then the value of above subkey is parsed for the presence of substrings like vmware, qemu, xen and so on.

This is again based on known values.

**Module Names:** Sandboxes inject modules in a process to log the activities performed by it. Since some of the module names used by security sandboxes are known publicly, they are used by the virus to detect the presence of sandbox the following way:

Call GetModuleHandleA() on the module name. If the handle returned is Non Null, this indicates the module is loaded. It is unloaded by calling FreeLibrary(). Once this is done, the sandbox will not log any activity performed by the malware. Common module names often searched by the viruses are **dbghelp.dll**, **sbiedll.dll** and so on.

**Backdoor Detection:** In most virtual machine softwares like VMWare Workstation, VMWare Fusion, Virtual PC, Parallels and so on, Hypervisor uses a backdoor to communicate with the guest operating system.

We can detect the presence of virtualization software and also identify its type using this backdoor technique.

For instance, to detect the presence of VMWare there is a known documented technique which checks for the presence of VMX backdoor used by Hypervisor to communicate with the guest operating system:

mov eax,0x564d5868 mov ecx, 0xa mov dx, 0x5658 in eax, dx cmp ebx, 0x564d5868

While these techniques are good, they are again directed towards specific virtualization softwares.

**Long Opcode Instructions:** This technique was documented by jaelanicu in 2009. It is not used so often in viruses, however it is a unique technique. It is based on the fact that virtualized CPU does not have a limitation on the length of an instruction unlike a real x86 CPU. When an instruction of length greater than 0x15 bytes is executed on a real CPU, it will trigger an exception however in a virtual CPU it will not trigger an exception. This difference in the result is used to detect the presence of virtualization.

70005100	55	PUSH EBP	
7000510E	8BEC	MOV EBP,ESP	
70005110		PUSH -1	
70005112		PUSH 9c2ffb4f.70009288	
70005117	68 40870070	PUSH 9c2ffb4f.70008740	JMP to msvortexcept_handler3
7000511C		MOV EAX, DWORD PTR FS: [0]	
70005122		PUSH EAX	
70005123	64:8925 000000	MOV DWORD PTR FS:[0],ESP	
7000512A		SUB ESP,0C	
7000512D		PUSH EBX	
7000512E		PUSH ESI	
7000512F		PUSH EDI	
70005130		MOV DWORD PTR SS:[EBP-18],ESP	
70005133		PUSH 1	
70005135		POP EAX	
70005136		AND DWORD PTR SS:[EBP-4],0	
7000513A	8E:	PREFIX DS:	Superfluous prefix
7000513B	8E:	PREFIX DS:	Superfluous prefix
7000513C	8E:	PREFIX DS:	Superfluous prefix
7000513D	8E:	PREFIX DS:	Superfluous prefix
7000513E	8E:	PREFIX DS:	Superfluous prefix
7000513F	3E:	PREFIX DS:	Superfluous prefix
70005140	3E:	PREFIX DS:	Superfluous prefix
70005141	3E:	PREFIX DS:	Superfluous prefix
70005142	3E:	PREFIX DS:	Superfluous prefix
70005143	3E:	PREFIX DS:	Superfluous prefix
70005144	3E:	PREFIX DS:	Superfluous prefix
70005145	3E:	PREFIX DS:	Superfluous prefix
70005146	3E:	PREFIX DS:	Superfluous prefix
70005147	3E:	PREFIX DS:	Superfluous prefix
70005148	8E:	PREFIX DS:	Superfluous prefix
70005149	3E:EB 09	JMP SHORT 9c2ffb4f.70005155	Superfluous prefix
7000514C		PUSH 1	
7000514E		POP EAX	
7000514F		REIN	
70005150		MOV ESP, DWORD PTR SS:[EBP-18]	
70005153	3300	XUR EHX, EHX	
70005155		UR DWURD PIR SSILEBP-41, FFFFFFF	
70005159	884U FØ	MUV ECX,UWUKU PIN SS:[EBP-10]	

It was observed that Qakbot uses this technique as shown in the screenshot below:

Please note that this technique may not work reliably on recent versions of Virtualization Softwares.

**Number of Cores:** It is common for malware analysts to allocate a single processer core to the sandbox. However, in a real world case today, most processors will have multiple cores.

Malwares can use several techniques to find the number of CPU cores and then decide if they are running inside a virtual machine on the basis of the result. One of the easiest ways of doing this is by checking the Process Environment Block:

Mov eax, dword ptr fs:[0x30] Mov eax, dword ptr ds:[eax+0x64] Cmp eax, 0x1 Je vm\_detected This technique may appear to be trivial but it can be effective in some cases.

**Data structures:** There are certain structures like IDT, GDT and LDT, which are at different locations between Host and Guest OS. This concept was used in techniques such as Red Pill to detect the presence of virtualization software. Since SIDT is a sensitive unprivileged instruction, VMM performs binary translation for it to return a different result than the host OS. Credits for Red Pill to Joanna Rutkowska.

Please note that on a multi processor machine the behavior of SIDT is not consistent. I will be testing this on various virtualization softwares/processor configurations and including a consistent code in my VM Buster program (Appendix I).

**Device Information:** It is also possible to detect the presence of virtualization software by enumerating the device details using APIs like SetupDiGetClassDevsA, SetupDiEnumDeviceInfo and SetupDiGetDeviceRegistryPropertyA. After enumerating, it can be compared with known values used in Sandboxes like VMware Pointing, VMware Accelerated, VMware SCSI, VMware SVGA, VMware Replay, VMware server memory, CWSandbox, Virtual HD, QEMU and so on.

**File System Artifacts**: There are some system drivers specific to the virtualization software, which can be located in the path: %windir%\system32\drivers\. It was observed that there are a few viruses, which check for the presence of these files as well.

Some of the driver names to look for: vmci.sys, vmhgfs.sys, vmmouse.sys, vmscsi.sys, vmusbmouse.sys, vmx\_svga.sys, vmxnet.sys, VBoxMouse.sys.

**Network Adapter MAC Address**: The vendor of Network Adapter can be identified from the first 3 bytes of a Mac Address.

Example: 00-0C-29-B4-0A-15

00-0c-29 is specific to VMWare.

**Sensitive Instructions:** We know that the x86 processor architecture cannot be completely virtualized. VMWare introduced the concept of full virtualization using binary translation for sensitive unprivileged instructions like SIDT, SLDT, SGDT, VERR, VERW and others. Fortunately, these instructions exhibit a different behavior for a Guest OS and the Host OS due to this binary translation performed by the VMM.

Malwares in the past have used instructions such as VERR/VERW to detect the presence of virtualization softwares like VMWare.

Please note that the newer versions of VMWare are not impacted by it. Also, you can harden your Virtual Environment from these techniques by disabling the Acceleration option provided by your VMM software.

I have written a C program, which will use almost all of the above methods for various virtualization softwares to detect their presence. It is scalable and can be modified to support more virtualization softwares by adding more artifacts information.

The program can be found in Appendix I.

As can be seen, it is really easy to detect the presence of Virtual Environment for a virus. One must harden their sandbox by modifying the default configuration of a Guest Operating System to protect themselves from such Anti VM techniques.

# **Drawbacks of Common Anti VM techniques**

We looked at some of the commonly used techniques for detecting the presence of a sandbox. While these techniques are effective against few virtualization softwares, they rely on known data.

As the usage of sandboxes for detecting the malicious binaries is increasing and security organizations are leveraging these sandboxes for detection mechanisms, attackers will explore new evasion techniques.

If we have a sandbox which has an unknown list of running processes, unknown file system and registry artifacts, no guest VM tools, multiple processor cores, unknown injected module name, unknown hypervisor port, then almost all of the above commonly used anti vm/anti sandbox techniques are rendered ineffective.

# **Essentials of Sandbox Based Detection**

A sandbox, which is used to automatically analyze the behavior of a binary and conclude its maliciousness, has to monitor the activities performed by the binary. After studying closely various sandboxes used for automation of malware analysis, it was found that almost all these sandboxes have below common attributes:

- 1. They inject a module into the process address space of the binary being analyzed.
- 2. The injected module will perform API hooking in user mode to log the API calls and the parameters passed.

# **Detect and Unload**

We know that a module is injected into the address space of our malicious binary to log the activities.

How do we detect its presence?

As a malware author, we are aware of the modules that will be loaded by our binary during the course of its execution. We can enumerate over the list of loaded modules and identify the injected DLL. Below is an example code to do this.

Let us consider a binary, which loads only ntdll.dll and kernel32.dll by default. For the purpose of demonstration, I have used LoadLibrary() to load an extra module, gdi32.dll. In a real world scenario, the extra module would be injected by an external entity like a kernel mode driver.

```
#include <windows.h>
#include <stdio.h>
#include <TlHelp32.h>
/*
Author: Sudeep Singh
*/
int main(int argc, char **argv)
{
HANDLE psnap;
HMODULE hModule;
MODULEENTRY32 me;
me.dwSize = sizeof(MODULEENTRY32);
psnap = CreateToolhelp32Snapshot(TH32CS SNAPMODULE, 0);
if(!Module32First(psnap, &me))
Ł
    printf("There was an error in retrieving the module information\n");
    exit(0);
}
while (Module32Next (psnap, &me))
Ł
    if(strcmp(me.szModule, "kernel32.dll") != 0)
    Ł
        if(strcmp(me.szModule, "ntdll.dll") != 0)
        {
            hModule = GetModuleHandle(me.szModule);
            if(FreeLibrary(hModule) != 0)
            Ł
                printf("successfully unloaded injected module, %s\n",
me.szModule);
            }
        }
    }
}
return 0;
ł
```

We are enumerating over the modules using Module32First()/Module32Next() and doing a basic string comparison to identify the extra loaded modules. Once we find the injected DLL, we can unload it using a call to FreeLibrary().

Please note that even though this technique might appear to be easy, it can render the entire sandbox analysis mechanism ineffective once the injected DLL is unloaded.

What happens if the module is unloaded?

You might ask, what is the impact of unloading the injected DLL? Since all the API hooks are applied by your injected DLL as soon as the module is loaded into the address space of virus.

While the API hooks remain intact, their functionality is rendered ineffective. As an example, consider an inline hook placed by your injected DLL on an API, Sleep() imported from kernel32.dll

The function prolog of Sleep() after inline hook looks like:

jmp <into\_module\_address\_space>
push 0
push dword ptr ds:[ebp+0x8]

After the module is unloaded, when Sleep() API is invoked by the virus, it will try to follow the inline hook into the module address space. However, since the module is unloaded, this would result in a crash (since it does not point to a valid memory address range). As a result of this, the binary would not be analyzed in the sandbox.

### **Protect from Unload**

If the above technique is used by a virus to identify the extra loaded module and unload it using FreeLibrary(), we can protect from this using several methods.

**Reference Count of DLL:** We know that FreeLibrary() will unload a module from the process address space only if the reference count is 0.

Also, the reference count of a loaded module can be incremented by calling LoadLibrary(). Each time we call LoadLibrary(), it increments the reference count of loaded module and each time we call FreeLibrary(), it decrements the reference count.

As an example, let us consider the code mentioned above. We compile it into a binary and run it inside a debugger.

Set a breakpoint at a call to FreeLibrary() and run the program. When FreeLibrary() is called the first time, it is trying to unload the module, gdi32.dll

0040107B	. 51	PUSH ECX	
0040107C	. E8 6F000000	CALL module_e.004010F0	
00401081	. 8304 08	ADD ESP,8	
00401084	. 8500	TEST EAX,EAX	
00401086	74 48	JE SHORT module_e.004010D2	
00401088	. 68 58804000	PUSH module_e.0040B058	ASCII "ntdll.dll"
0040108D	. 8D95 E8FDFFFF	LEA EDX,DWORD PTR SS:[EBP-218]	
00401093	. 52	PUSH EDX	
00401094	. ES 57000000	CALL module_e.004010F0	
00401099	. 8304 08	ADD ESP,8	
0040109C	. 8500	TEST EAX,EAX	
0040109E	74 32	JE SHORT module_e.004010D2	
004010A0	. 8D85 E8FDFFFF	LEA EAX,DWORD PTR SS:[EBP-218]	
004010A6	. 50	PUSH EAX	rpModule
004010A7	. FF15 04804000	CALL DWORD PTR DS:E<&KERNEL32.GetModul	GetModuleHandleA
004010AD	. 8945 FC	MOV DWORD PTR SS:[EBP-4],EAX	
004010B0	. 884D FC	MOV ECX, DWORD PTR SS:[EBP-4]	
004010B3	. 51	PUSH ECX	rhLibModule < Set a Breakpoint Here
004010B4	. FF15 <u>00804000</u>	CALL DWORD PTR DS:I<&KERNEL32.FreeLibr	FreeLibrary
004010BA	. 8500	TEST EAX,EAX	
004010BC	74 14	JE SHORT module_e.004010D2	
004010BE	. 8D95 E8FDFFFF	LEA EDX,DWORD PTR SS:[EBP-218]	
004010C4	. 52	PUSH EDX	
004010C5	. 68 <u>64804000</u>	PUSH module_e.0040B064	ASCII "successfully unloaded injected module, %so"
004010CA	. E8 94030000	CALL module_e.00401463	

Before executing this call instruction, let us view the loaded modules in Olly Debugger. We can see that both, gdi32.dll and user32.dll are loaded in the process address space.

Address	Size	Owner		Section	Contains	Туре	Access	5 J	Initial	Mapped as
00010000	00001000		00010000			Priv	RW	F	RW	
00020000	00001000		00020000			Priv	RW	F	RW	
0012D000	00001000		00030000			Priv	RW Gu	ia: F	RW	
0012E000	00002000		00030000		stack of ma	Priv	RW Gu	ia: F	RW	
00130000	00003000		00130000			Map	R	F	۹ ا	
00140000	00003000		00140000			Priv	RW	F	RW	
00240000	00006000		00240000			Priv	RW	F	RW	
00250000	00003000		00250000			Map	RW	F	RW	
00260000	00016000		00260000			Map	R	F	R	NDeviceNHarddiskVolume1NWINDOWSNsystem32Nunicode.nls
00280000	00041000		00280000			Map	R	F	۹ ا	NDeviceNHarddiskVolume1NWINDOWSNsystem32Nlocale.nls
00200000	00041000		00200000			Map	R	F	R	NDeviceNHarddiskUolume1NWINDOWSNsystem32Nsortkey.nls
00320000	00006000		00320000			Map	R	F	R	NDeviceNHarddiskUolume1NWINDOWSNsystem32Nsorttbls.nls
00330000	00004000		00330000			Priv	RW	F	RW	
00340000	00003000		00340000			Map	R	F	R	\Device\HarddiskUolume1\WINDOWS\system32\ctype.nls
00350000	00001000		00350000			Priv	RW	F	RW	
00360000	00001000		00360000			Priv	RW	F	RW	
00400000	00001000	module_e	00400000		PE header	Imag	R	F	RWE	
00401000	00007000	module_e	00400000	.text	code	Imag	R	F	RWE	
00408000	00003000	module_e	00400000	.rdata	imports	Imag	R	F	RWE	
00408000	00003000	module_e	00400000	.data	data	Imag	R	F	RWE	
0040E000	00001000	module_e	00400000	.reloc	relocations	Imag	R	F	RWE	
00410000	00004000		00410000			Map	R E	F	R E 👘	
00400000	00002000		00410000			Map	R E	F	R E 👘	
004E0000	00103000		004E0000			Map	R	F	3	
005F0000	00092000		005F0000			Map	R E	F	R E 👘	
77F10000	00001000	gdi32	77F10000		PE header	Imag	R	F	RWE	
77F11000	00043000	gdi32	77F10000	.text	code, import	Imag	R	F	RWE	< gdi32.dll is loaded
77F54000	00002000	gdi32	77F10000	.data	data	Imag	R	F	RWE	
77F56000	00001000	gdi32	77F10000	.rsrc	resources	Imag	R	F	RWE	
77F57000	00002000	gdi32	77F10000	.reloc	relocations	Imag	R	F	RWE	
70800000	00001000	kernel32	70800000		PE header	Imag	R	F	RWE	
70801000	00084000	kernel32	70800000	.text	code, import	Imag	R	F	RWE	
70885000	00005000	kernel32	70800000	.data	data	Imag	R	F	RWE	
70888000	00066000	kernel32	70800000	.rsrc	resources	Imag	R	F	RWE	
7C8F0000	00006000	kernel32	70800000	.reloc	relocations	Imag	R	F	RWE	
70900000	00001000	ntdll	70900000		PE header	Imag	R	F	RWE	
70901000	00078000	ntdll	70900000	.text	code, export	Imag	R	F	RWE	
7C97B000	00005000	ntdll	70900000	.data	data	Imag	R	F	RWE	
70980000	00020000	ntdll	70900000	.rsrc	resources	Imag	R	F	RWE	
70980000	00003000	ntdil	70900000	.reloc	relocations	Imag	R	F	RWE	
7E410000		USER32	7E410000		PE header	Imag	R	F	RWE	
7E411000	00060000	USER32	7E410000	.text	code, import	Imag	R	F	RWE	c user22 dll is loaded along with gdi22 dll
7E471000		USER32	7E410000	.data	data	Imag	R	F	RWE	with guiszian is loaded along with guiszian
7E473000		USER32	7E410000	.rsrc	resources	Imag	R	ľ	RWE	
7E49E000	00003000	USER32	7E410000	.reloc	relocations	Imag	R	ľ	RWE	
7F6F0000	00007000		7F6F0000			Map	RE	F	R E	
7FFB0000	00024000		7FFB0000			Map	R	F	1	
7FFDE000	00001000		7FFDE000		data block	Priv	RW	F	RW	
7FFDF000	00001000		7FFDF000			Priv	RW	F	RW	
7FFE0000	00001000		7FFE0000			Priv	R	F	<	

Now, we execute the call and notice that these modules are unloaded. This can be confirmed by viewing the Memory Window in Olly Debugger once again as shown below:

Address	Size	Owner		Section	Contains	Туре	Acces	s I	Initial	Mapped as
00010000	00001000		00010000			Priv	R₩	F	RW	
00020000	00001000		00020000			Priv	R₩	F	RW	
00120000	00001000		00030000			Priv	RW 6	iua: F	RW	
0012E000	00002000		00030000		stack of ma	Priv	RW 6	iua: F	RW	
00130000	00003000		00130000			Map	R	F	R	
00140000	00003000		00140000			Priv	R₩	F	RW	
00240000	00006000	- I	00240000			Priv	RW	F	RW	
00250000	00003000		00250000			Map	RW	F	RW	
00260000	00016000		00260000			Map	R	F	R	\Device\HarddiskVolume1\WINDOWS\system32\unicode.nls
00280000	00041000		00280000			Map	R	F	R	\Device\HarddiskVolume1\WINDOWS\system32\locale.nls
00200000	00041000		00200000			Map	R	F	R	\Device\HarddiskVolume1\WINDOWS\system32\sortkey.nls
00320000	00006000		00320000			Map	R	F	R	\Device\HarddiskVolume1\WINDOWS\system32\sorttbls.nls
00330000	00004000		00330000			Priv	RW	F	RW	
00340000	00003000		00340000			Map	R	F	R	\Device\HarddiskVolume1\WINDOWS\system32\ctype.nls
00350000	00001000		00350000			Priv	R₩	F	RW	
00360000	00001000		00360000			Priv	RW	F	RW	
00400000	00001000	module_e (	00400000		PE header	Imag	R	F	RWE	
00401000	00007000	module_e (	00400000	.text	code	Imag	R	F	RWE	
00408000	00003000	module_e (	00400000	.rdata	imports	Imag	R	F	RWE	
00408000	00003000	module_e (	00400000	.data	data	Imag	R	F	RWE	
0040E000	00001000	module_e (	00400000	.reloc	relocations	Imag	R	F	RWE	
00410000	00004000		00410000			Map	R E	F	R E	
00400000	00002000		00410000			Map	R E	F	R E	
004E0000	00103000		004E0000			Map	R	F	R	
005F0000	00092000		005F0000			Map	R E	F	RE	
70800000	00001000	kernel32	70800000		PE header	Imag	R	F	RWE	
70801000	00084000	kernel32	70800000	.text	code, import	Imag	R	F	RWE	
70885000	00005000	kernel32	70800000	.data	data	Imag	R	F	RWE	
7C88A000	00066000	kernel32	70800000	.rsro	resources	Imag	R	F	RWE	
7C8F0000	00006000	kernel32	70800000	.reloc	relocations	Imag	R	F	RWE	
70900000	00001000	ntdll	70900000		PE header	Imag	R	F	RWE	
70901000	00074000	ntdll	70900000	.text	code, export	Imag	R	F	RWE	
70978000	00005000	ntdll	70900000	.data	data	Imag	R	F	RWE	
70980000	00020000	ntdll	70900000	.rsrc	resources	Imag	R	F	RWE	
7C9AC000	00003000	ntdll	70900000	.reloc	relocations	Imag	R	F	RWE	
7F6F0000	00007000		7F6F0000			Map	RE	F	RE	
7FFB0000	00024000		7FFB0000			Map	R	F	R	
7FFDE000	00001000		7FFDE000		data block (	Priv	RW	F	RW	
7FFDF000	00001000		7FFDF000			Priv	RW	F	RW	
7FFE0000	00001000		7FFE0000			Priv	R	F	R	

Let us modify the previous code by calling LoadLibrary() more than 1 time as shown below:

```
int i=0;
while(i<0x2)
{
    LoadLibraryA("gdi32.dll");
    i++;
}</pre>
```

After compiling this into a binary and attaching the debugger, we once again check the Memory Window after executing the call to FreeLibrary(). This time, we observe that even though FreeLibrary() returns a non zero value, the module is still loaded in the address space.

This is a very trivial method to prevent your module from being unloaded. A virus author could check the reference count of a module before calling FreeLibrary().

I wrote the following inline assembly, which can be used to find the reference count of any loaded module. We could then modify the reference count using inline assembly and call FreeLibrary().

\_\_\_asm { pushad mov ebx, hModule

```
mov eax, dword ptr fs:[0x18]
mov eax, dword ptr ds:[eax+0x30]
mov eax, dword ptr ds:[eax+0xc] ; _PEB_LDR_DATA
add eax, 0xc
mov ecx, dword ptr ds:[eax] ; pointer to InLoadOrderModuleList
repeat:
mov edx, ecx
cmp dword ptr ds:[edx+0x8], 0
mov ecx, dword ptr ds:[ecx]
je repeat
cmp ebx, dword ptr ds:[edx+0x18]
jnz repeat
mov eax, dword ptr ds:[edx+0x38]
mov ref_count, eax
popad
}
```

Above code will find the reference count (LoadCount) of the module that we want to unload. We find the LoadCount by parsing the Process Environment Block.

This will allow the attacker to unload the injected module even if the reference count was modified by calling LoadLibrary() multiple times.

**Prevent Enumeration of Modules**: If a sandbox is relying on DLL injection to analyze the behavior of a binary, it is essential to hook APIs such as **Module32First()/Module32Next()** which could be used to enumerate the loaded modules. However, based on the study of some sandboxes, it was found that these APIs are not hooked in the user mode.

**Hiding the module in PEB**: It is possible to hide the injected module in the Process Environment Block. This way, it would not show up in the list of loaded modules. Such techniques are encouraged and should be used by sandboxes.

When a process loads a module, information specific to the DLL is stored in the Process Environment Block. Below are some structures specific to PEB, which allow us to access DLL information:

0:001> dt nt!\_PEB @\$peb ntdll!\_PEB +0x000 InheritedAddressSpace : 0 '' +0x001 ReadImageFileExecOptions : 0 '' +0x002 BeingDebugged : 0x1 '' +0x003 SpareBool : 0 '' +0x004 Mutant : 0xffffffff Void +0x008 ImageBaseAddress : 0x01000000 Void +0x00c Ldr : 0x001a1e90 \_PEB\_LDR\_DATA **PEB\_LDR\_DATA** structure has 3 linked lists, which store information about all the loaded modules.

0:001> dt nt!\_PEB\_LDR\_DATA 0x001a1e90 ntdll!\_PEB\_LDR\_DATA +0x000 Length : 0x28 +0x004 Initialized : 0x1 '' +0x008 SsHandle : (null) +0x00c InLoadOrderModuleList : \_LIST\_ENTRY [ 0x1a1ec0 - 0x1a2bc0 ] +0x014 InMemoryOrderModuleList : \_LIST\_ENTRY [ 0x1a1ec8 - 0x1a2bc8 ] +0x01c InInitializationOrderModuleList : \_LIST\_ENTRY [ 0x1a1f28 - 0x1a2bd0 ]

The 3 linked lists are highlighted above. If we can unlink the information of our injected module from these 3 linked lists, our module will be hidden.

This means,

GetModuleHandle() would return NULL for our module name. As a result of this, the FreeLibrary() trick for unloading our module will not work.

Module32First()/Module32Next() will not show our DLL in the list of loaded modules. This is because these Windows APIs also use the information stored in PEB to enumerate the loaded modules.

What code we need to add to our DLL?

In order to unlink our module from the PEB, we need to add a function which will be called when the reason code, **DLL\_PROCESS\_ATTACH** is passed to our **DllMain()** as shown below:

```
BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul_reason_for_call, LPVOID
lpReserved)
{
    if(ul_reason_for_call == DLL_PROCESS_ATTACH)
    {
        HideDll((ULONG_PTR)hModule);
        MessageBoxA(NULL,"DLL Hidden", "Hide the DLL", MB_OK);
    }
return 1;
}
```

The complete code for unlinking the DLL from PEB can be found in Appendix III. Credits to Pnluck from OpenRCE for this.

When LoadLibrary() is called, it invokes the DllMain() function of DLL which in turn will call HideDll() function that unlinks the module from PEB.

In order to confirm that our method works, let us use the program discussed previously to enumerate the modules using Module32First()/Module32Next() to load our new modified module.

We will set a breakpoint at a call to LoadLibrary().

When we return from LoadLibrary(), we can see the base address of our module as 0x10000000 in eax.

Let us view the list of loaded modules in Memory Window of Olly Debugger. We can see that a memory region is mapped at address 0x10000000 however no module name is shown.

Address	Size	Owner	Section	Contains	Туре	Access	Initial	Mapped as
00010000	00001000				Priv	RW	RW	
00020000	00001000				Priv	RW	RW	
00120000	00001000				Priv	R⊌ Gua	RW	
00120000	00003000			stack of ma	Priv	RW Gua	RW	
00130000	00003000				Map	R	R	
00140000	00006000				Priv	RW	RW	
00240000	00006000				Priv	RW	RW	
00250000	00003000				Map	RW	RW	
00260000	00016000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\unicode.nls
00280000	00041000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\locale.nls
00200000	00041000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\sortkey.nls
00320000	00006000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\sorttbls.nls
00330000	00004000				Priv	RW	RW	
00340000	00003000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\ctype.nls
00350000	00001000				Priv	RW	RW	
00360000	00001000				Priv	RW	RW	
00370000	00004000				Priv	RW	RW	
00380000	00005000				Priv	RW	RW	
00390000	00050000				Map	R	R	
00400000	00001000	module_e		PE header	Imag	R	RWE	
00401000	00007000	module_e	.text	code	Imag	R	RWE	
00408000	00003000	module_e	.rdata	imports	Imag	R	RWE	
0040B000	00003000	module_e	.data	data	Imag	R	RWE	
0040E000	00001000	module_e	.reloc	relocations	Imag	R	RWE	
00410000	00004000				Map	R E	R E	
00400000	00002000				Map	R E	R E	
004E0000	00103000				Map	R	R	
005F0000	00092000				Map	RE	RE	
1000000	00008000				Imag	R	RWE	< This memory region corresponds to our hidden DLL
5AD70000	00001000	uxtheme		PE header	Imag	R	RWE	
5AD71000	00030000	uxtheme	.text	code, import	Imag	R	RWE	
5HDH1000	00001000	uxtheme	.data	data	Imag	R	RWE	
5ADA2000	00004000	uxtheme	.rsro	resources	Imag	R	RWE	
SADA6000	00002000	uxtheme	.reloc	relocations	Imag	R	RWE	
77010000	00001000	msvort		PE header	Imag	R	RWE	
77011000	00040000	msvert	.text	code, import	Imag	K	RWE	
77060000	00007000	msvort	.data	data	Imag	K	RWE	
77064000	00001000	msvort	.rsrc	resources	Imag	K	RWE	
77065000	00003000	nsvort	.reloc	relocations	imag	R .	RWE	
77000000	00001000	HDVHP132		PE header	Imag	R	RWE	
77001000	00075000	HDUHP132	.text	code, import	Imag	R .	RWE	
77E46000	00005000	HDUHP132	.data	data	Imag	R .	RWE	
77E4B000	00018000	ADVAP132	.rsrc	resources	Imag	R .	RWE	
77566000	00005000	HDVHP132	.retoc	relocations	Imag	R.	RWE	

This means, it is possible to hide our injected module from the Debugger as well.

### API Hooking

So far we looked at methods of detecting the injected DLL and how one can prevent that DLL from being unloaded by a virus.

Now, let us target another essential functionality of an automated malware analysis sandbox. In order to log the activities performed by the virus, there are API hooks placed by the injected DLL. It is becoming increasingly common these days for malwares to detect the API hooks in a sandbox. However, it was observed that most malwares only check for inline API hooks.

We will look at some of the viruses found in the wild and understand the API hook detection techniques used by them.

### **Detect and Skip**

As we know, in Windows, some APIs when invoked will in turn invoke other low level APIs. A good example of this is APIs imported from kernel32.dll. These APIs in turn invoke functions from ntdll.dll

For instance,

Sleep() from kernel32.dll calls SleepEx() from kernel32.dll

SleepEx() in turn calls NtDelayExection() from ntdll.dll

We also know that in Microsoft Windows, most of the wrapper APIs have a 0x5 byte stub at function prolog which looks like shown below:

mov edi, edi push ebp mov ebp, esp

This stub has a size of 0x5 bytes and since we require 0x5 bytes to place an inline API hook, it makes it very convenient for sandboxes to apply an inline API hook for such APIs.

An inline hook for Sleep() API would look like:

jmp <into module address space> push 0 push dword ptr ds:[ebp+0x8] call kernel32!SleepEx Since an API hook on wrapper API can be bypassed by a virus by calling lower level APIs, an API is hook is placed on SleepEx() as well which has a different function prolog.

Options for inline API hook:

- Short jmp opcode 0xeb
   Near jmp opcode 0xe9
- 3. Call opcode 0xe8

As there is only a limited number of ways in which a sandbox can apply an inline hook, it is trivial to bypass them by checking the first byte of the API.

If the malware calls the APIs through a stub which first checks the API prolog and then skips it if an inline hook is detected, the sandbox would not be able to log any activity of the malware.

Example stub:

```
api address = GetProcAddress(hModule, api name);
 asm
{
mov
      eax, api address
cmp
      byte ptr [eax],0E8h
je
      dest1
      byte ptr [eax],0E9h
cmp
je
      dest1
      byte ptr [eax],0EBh
cmp
      dest2
jne
dest1:
      dword ptr [eax+5],90909090h
cmp
       dest2
je
       edi,edi
mov
push
       ebp
mov
      ebp,esp
lea
      eax,[eax+5]
dest2:
jmp
      eax
}
```

In the above stub, we check if the first byte of the API prolog is 0xe8 or 0xe9. If so, then we jump to the location, dest1. At dest1, we check if the inline hook is followed by 4 NOP instructions. This check is to ensure that the inline hook is not a default hook placed by OS since from Windows 7 onwards; the calls from kernel32.dll are redirected to kernelbase.dll as shown below:

0:001> u kernel32!Sleep	
kernel32!Sleep:	
00000000`773e28e8 ff25b2ae0700 j	<pre>jmp qword ptr [kernel32!_imp_Sleep (00000000`7745d7a0)]</pre>
00000000`773e28ee 90 n	lop
00000000`773e28ef 90 n	lop
00000000`773e28f0 90 n	юр
00000000`773e28f1 90 n	lop
00000000`773e28f2 90 n	юр
00000000`773e28f3 90 n	юр
kernel32!Wow64DisableWow64FsRedirec	stion:
00000000`773e28f4 90 n	lop

If we detect an inline API hook, then we execute the standard prolog instructions stored in our stub (0x5 byte stub). This is followed by adding 0x5 to the API address to skip over the function prolog and resume execution from the 4th instruction.

This way, we do not affect the functionality of the API and also bypass any user mode inline hook applied on an API with a standard prolog.

Now, one might ask, what if the first instruction of an API is a jmp or a call instruction by default in the OS. We have included a check for Win 7 OS in our API hook checking stub above, when a jmp instruction is followed by 4 NOP instructions.

However, there are also some APIs imported from kernel32.dll, ntdll.dll and user32.dll, which have the first instruction as a jmp/call.

In order to find out the API names, I have written a C Program which enumerates all the APIs in the export directory of a module, calculates its address and checks the function prolog for control transfer instruction opcodes (0xe8, 0xe9 and 0xeb).

Based on the results for Win XP SP3, we have:

kernel32.dll - 4 ntdll.dll - 8 advapi32.dll - 0 user32.dll - 1 ws2 32.dll - 0

Let us check which functions in these modules have the first instruction as jmp/call by default in the OS:

#### kernel32.dll:

CloseProfileUserMapping DebugBreak GetUserDefaultLangID UnregisterConsoleIME

#### ntdll.dll:

\_Cllog \_Clpow atan ceil floor log pow

#### user32.dll:

AnyPopup

As you can see in the list above, fortunately a virus rarely uses these APIs and we don't need to check for an inline API hook for these.

The API hook checking stub mentioned above works good for the purpose of virus.

Also, please note that, this API hook checking stub can also be utilized in a shellcode. The importance of using this in a shellcode is:

Some security products like **EMET** detect **ROP** payload execution by checking for stack pivot. These checks are done by monitoring a specific set of APIs, which are often called by ROP payloads like VirtualAlloc, VirtualProtect, CreateFile and so on.

Once, they detect a call to these APIs, they perform a check on the stack pointer to ensure that it is within the limits as mentioned in the TIB.

#### TIB->StackLimit < esp < TIB->StackBase

If a ROP payload calls all the above APIs through an API hook checking stub as mentioned above, it can bypass the exploit code detections such as stack pivot as used in some security products like EMET.

### **Function Prolog Analysis**

So far, we have discussed APIs, which have the default function prolog with a size of 0x5 bytes, which makes it very convenient for the sandboxes to apply an API hook without altering the functionality of the API.

I modified my previous C Program to calculate the number of APIs imported from various modules on Windows XP SP3 which have a non standard prolog (First 0x5 bytes are not equal to 0x8b, 0xff, 0x55, 0x8b, 0xec).

The code for this can be found in Appendix II.

Below are the results:

Module	Non Standard	Default Inline Hook by	Total Number of Exported
Name	Prolog	OS	Functions
kernel32.dll	215	4	953
ntdll.dll	767	8	1315
advapi32.dll	243	0	676
user32.dll	204	1	732
ws2_32.dll	33	0	117

Which are the other types of function prologs?

#### Prolog #1:

0:002> u kernel32!SleepEx kernel32!SleepEx: 7c8023a0 6a2c push 2Ch 7c8023a2 686024807c push 7c802460 7c8023a7 e82a010000 call kernel32!\_SEH\_prolog (7c8024d6)

The size of first 2 instructions is 0x7 bytes followed by a CALL instruction.

#### Prolog #2:

0:002> u kernel32!CreateRemoteThread kernel32!CreateRemoteThread: 7c8104bc 6810040000 push 410h 7c8104c1 689806817c push 7c810698 7c8104c6 e80b20ffff call kernel32!\_SEH\_prolog (7c8024d6) The size of first 2 instructions is 0xa bytes followed by a CALL instruction.

#### Prolog #3:

0:002> u ntdll!ZwDelayExecution ntdll!ZwDelayExecution: 7c90d1f0 b83b000000 mov eax,3Bh 7c90d1f5 ba0003fe7f mov edx,7ffe0300 7c90d1fa ff12 call dword ptr [edx]

The size of first instruction is 0x5 bytes.

#### Prolog #4:

0:002> u kernel32!CloseProfileUserMapping kernel32!CloseProfileUserMapping: 7c82c865 e80efdfeff call 7c81c578 7c82c86a 833dd450887c00 cmp dword ptr [7c8850d4],0

The size of first instruction is 0x5 bytes, which is a CALL instruction by default by the OS.

What do we conclude from the above Prologs?

We saw previously that Prolog #4 is uncommon and it is present only for few APIs, which are rarely used by the virus.

Regarding the other 3 API prologs, we can see that it is still convenient for a sandbox to place an inline hook.

Let us discuss each type of prolog one by one:

**Prolog #1**: Besides the standard prolog of 0x5 byte stub, the second most common prolog in Windows is this type of prolog.

Here, two parameters are passed to **\_SEH\_prolog** function. Since these parameters are constants for a specific API, we can easily copy them to our buffer and redirect the control flow to the third instruction in the prolog after our hook has completed the logging activity.

Taking the example of SleepEx() above, our hook would now look like:

jmp <into\_module\_address\_space>
nop
nop
call kernel32!\_SEH\_Prolog <-- sandbox API hook will return here.</pre>

Note the addition of 2 NOP instructions since in this case we have a 0x7 byte prolog.

**Prolog #2**: This prolog is similar to the above, however here both the first 2 instructions have a size of 0x5 bytes. So, we need to copy 0xa bytes to our buffer.

Taking the example of CreateRemoteThread above, our hook would look like:

jmp <into\_module\_address\_space>
nop
nop
nop
nop
nop
call kernel32!\_SEH\_Prolog << sandbox API hook will return here.</pre>

**Prolog #3**: This type of function prolog is specific to Native APIs imported from ntdll.dll. As we know, APIs from kernel32.dll will call the native APIs from ntdll.dll, a sandbox might place an inline hook at a native API as well.

All these Native APIs have a similar function prolog. They place the system service number in eax, move the pointer to **SystemCallStub** in edx and call it.

The size of first instruction in this prolog is 0x5 bytes, which makes it convenient to place an inline hook. Also, the first instruction in this prolog is a constant specific to the API, so we can copy it to our buffer without affecting the functionality of the API.

Taking the example of **ZwDelayExecution** above, our hook would look like:

jmp <into\_module\_address\_space>
mov edx, 0x7ffe0300 <-- sandbox API hook would return here.
call dword ptr [edx]</pre>

As we can see from the above function prolog analysis, the method of detecting an inline hook remains consistent. Also, it is highly likely that a sandbox would place an inline hook at any one of these stages.

# **Detect and Exit**

In some cases, if a virus detects an API hook placed by a sandbox, it might exit or crash to prevent analysis in a sandbox.

However, these days, viruses would not want to exit, as there is a high likelihood that such API hooks are also present on a real world endpoint due to endpoint security protection mechanisms. This makes it necessary for the virus to bypass the hooks in addition to detecting them.

# **Detect and Patch**

There are some viruses in the wild which will detect the API hook and instead of skipping it, they will patch it. As an example, let us analyze the algorithm used by a virus found in the wild to patch the API hooks.

The main hook checking algorithm works as follows:

1. It opens the system DLLs like ntdll.dll, kernel32.dll and advapi32.dll from the path, %windir%\system32 using CreateFileA.

2. It maps these DLLs to memory by parsing their PE Header. It loads each section (.text, .data, .rsrc and .reloc) manually into memory. It uses multiple calls to SetFilePointer and ReadFile to perform these functions.

3. After loading the module in memory, it then locates and parses the **Export Data Directory**. Using the **AddressOfOrdinals**, **AddressOfFunctions** and **AddressOfNames** arrays in the Export Directory, it forms a structure for each of the exported API as shown below:

```
struct API_HOOK
{
    DWORD APIOrdinal;
    char *api_name;
    void *api_address;
    BYTE *buffer;
    int size;
} *API_HOOK
```

This structure stores the API ordinal, pointer to API name, the actual API address (as loaded in the memory) and pointer to a buffer which contains the first 0x8 bytes of the API prolog for the API. The size member of the above structure is always set to 0x8.

4. It then calls the function for checking any differences in the API prolog of APIs imported from the corresponding module.

Below screenshot shows the function used for this purpose. The first parameter of this function is a pointer to a pointer to an array of pointers to structures of type **API\_HOOK** as mentioned above.

00B3A209	55	PUSH EBP	
00B3A20A	8BEC	MOV EBP.ESP	
00838200	83EC 1C	SUB ESP.1C	
0083820E	8845.08	MOU FAX, DWORD PTR SS: [FBP+8]	
00830212	3302	YOR FOX FOX	
000000212	00EE EA	MOLL DWORD DTD CONFERDANCE ENV	
00000017	0700 F4	OWD DWORD FIR SSILEDFTCJ,EDA	
00838217	3950 04	UNP DWORD FIR DS:LEHAT4J,EDA	
0083H21H	V0F86 C5000000	JBE 00B3HZE5	
00B3H220	58	PUSH EBX	
00B3A221	56	PUSH ESI	
00B3A222	57	PUSH EDI	
00B3A223	8B45 08	MOV EAX, DWORD PTR SS:[EBP+8]	
00B3A226	8BØ8	MOV ECX, DWORD PTR DS: [EAX]	
00B3A228	8509	TEST ECX, ECX	
00B3A22A	J75 04	JNZ SHORT 00B3A230	
00B3822C	3300	XOR EAX.EAX	
00B3922E	EB 03	IMP SHORT 00B30233	
00830230	800491	LEG EGY DHORD PTR DS+FECX+EDX+41	
000000200	9900	MOLL FOY DWORD PTP DS+FEOV1	ORT HOOK struct
00000200	0000	MOULEON DWORD PTP DevEENY+01	DPI MORE bei address
000000200	0045 50	MOULDWORD DTD CONFERD OF FOU	HET_HOOK_ABPT_BOOK_SS
000000200	0740 FO	TEOT FOW FOW	
0083H23B	8509		
00B3H23D	v75 Ø4	JNZ SHURT 00B3H243	
00B3A23F	3300	XOR EAX,EAX	
00B3A241	↓EB 03	JMP SHORT 00B3A246	
00B3A243	800491	LEA EAX,DWORD PTR DS:[ECX+EDX+4]	
00B3A246	8800	MOV EAX, DWORD PTR DS: [EAX]	API_HOOK struct
00B3A248	8840 OC	MOV EAX, DWORD PTR DS:[EAX+C]	API_HOOK->buffer
00B3A24B	8945 E8	MOV DWORD PTR SS:[EBP-18],EAX	
00B3A24E	8509	TEST ECX.ECX	
00B3A250	075-04	JNZ SHORT ØØB3A256	
ØØB38252	3300	XOR FAX.FAX	
00830254	FB 03	IMP SHORT 00830259	
000000000	909491	I CO COV DWORD DTD DC+FECV+EDV+41	
000000000	000401	MOLL EON DWORD DTP DE. FEONI	ORT HOOK struct
000000202	0000 10	MOULERY RUORD FIR DOLLERAL	
000000000	0000 10	HOV EDA, DWORD FIR DOLLEHATION	HFI_HOOK=2st2e
00B3H25E	8950 EL	HOV DWORD FIR SSILEBF-141,EBX	
00B3H261	3300	XUR EHX,EHX	
00B3H263	8875 F8	MUV ESI, DWURD PIR SS: LEBP-81	HP1 Hddress as loaded in memory
00B3H266	887D E8	MOV EDI, DWORD PTR SS:[EBP-18]	Buffer corresponding to original bytes of API prolog
00B3A269	8B4D EC	MOV ECX, DWORD PTR SS:[EBP-14]	size = 0x8
00B3A26C	FC	CLD	
00B3A26D	F3:A6	REPE CMPS BYTE PTR ES: [EDI], BYTE PTR DS	
00B3A26F	V74 01	JE SHORT 00B3A272	
00B3A271	40	INC EAX	
00B38272	8945 FR	MOU DWORD PTR SS:FERP-101.F9X	
ECX=00000	1008 (decimal 8.)		
DS:[ESI]=	:[7C957EFB]=6A (*,		< Array of pointers to structures of type, API HOOK
Oddress	Hey dump		
00025000			
AGC SEADO	DO 22 DE 00 E0 2	1  DE  00  CO  10  DE  00  CO  20  CI  00  CO  10  CI  00  CO  10  CI  00  CI	
COCOFIE	10 00 DF 00 D0 00	+ DF 00 E0 20 DF 00 00 27 DF 00 1 1.F31.	
DECOPHED	10 27 BF 00 B0 20	1 DF 00 40 20 DF 00 E0 20 DF 00 171	
00C3FHF8	78 2F BF 00 10 3:	I BF 00 H8 32 BF 00 40 34 BF 00 x⁄₁.▶1₁.	627 • 847 •
00C3FB08	D8 35 BF 00 70 37	/ BF 00 08 39 BF 00 H0 3H BF 00 ‡5₁.p7₁.	47.817.
00C3FB18	38 30 8F 00 00 30	J BF 00 68 3F BF 00 00 41 BF 00 8<₁.#=₁.	Difference and the second s
00C3FB28	98 42 BF 00 30 44	+ BF 00 C8 45 BF 00 60 47 BF 00 ÿΒ <sub>1</sub> .0D <sub>1</sub> .	En . 16.
00C3FB38	F8 48 BF 00 90 40	н BF 00 28 4C BF 00 C0 4D BF 00 °H.ел.	(L1 M).
00C3FB48	58 4F BF 00 F0 50	ט BF 00 88 52 BF 00 20 54 BF 00 X01.≡P1.	eRa. Ta.
00C3FB58	B8 55 BF 00 50 57	7 BF 00 E8 58 BF 00 80 5A BF 00 ╕UႹ.PѠႹ.	ž×1.ĢZ1.
00C3FB68	18 5C BF 00 B0 50	) BF 00 48 5F BF 00 E0 60 BF 00 ↑\٦.∭]٦.	H_1.«'1.
00C3FB78	78 62 BF 00 10 64	4 BF 00 A8 65 BF 00 40 67 BF 00 xb₁.≯d₁.	خ <del>تا، قاتا، المعالمة المعالم</del>
00C3FB88	D8 68 BF 00 70 66	А ВЕ 00 08 6C ВЕ 00 А0 6D ВЕ 00 †hา.рјา.	ala.ám.
00C3FB98	38 6F BF 00 D0 70	и вн ий[68-72 вн ий]ий 74 вн ий 8о <mark>л."</mark> рл.!	hrqtq.

For instance, we can see the array of pointers to structures of type, **API\_HOOK** at address, 0x00C3FAC8

Each of these structures correspond to an API imported from ntdll.dll

Let us check the structure at 0x00C6FFA8

Address	He	< du	amp.														ASCII
					_				_				_			_	
<b>AACGEEBS</b>	Ø1	·юю	·ЮЮ	юø	98	10	BE	юø	FB	7F	95	-70	FЙ	10	BE	юø	8ບັບສຸມປີໃດ່!≣ບສຸມ
000011110	× •	~~		~~			- Con	~~							- Con	~~	ourselie oneri
00C4FFR9	0.2	юø	aa	юø	64	00	00	юø	70	10	RE	юø	06	юø	юø	юø	🗖 d nata 🔺
000011.00	00	00	- 00	- 00	94	00	00	- 00	10	- 10	- Di	- 00	00	00	00	- 00	Been deserballe zone
00020000	CA.	00	00	00	OD.	OD.	OD.	OD.	OD.	OD.	OD.	OD.	CC.	EE.	EE.	CC.	A BEBBBBBBBBBBBB
DOCOFFCO	04	00	00	00	но	но	но	но	но	но	но	но				F E	n***33333336eee
aaccentra	00	00	00	00	00	00	00	00	0.4	00	00	00	EE.	4.4		00	A 🗖 c 🖉 c
000666608	66	66	66	66	99	66	90	66	64	66	69	96	EE	14	EE	99	••••••••••••••••••••••••••••••••••••••
<b>OOCAEEEO</b>	20	EC.	DE	00	00	0.1	07	00	EE	EE	EE	EE	EE	EE	EE	EE	had do chonchon
DOCOFFES	00	гь.	DE	90	20	61	DC	90	EE.	TE.	EE.	E E	EE.	TE.	EE	TE.	H yon . energiese
ooccepto.	E.E.					<b>FF</b>											
0006FFF8	EE	FE.	EE	FE	EE	FE.	EE	FE									elelele

5. Now, it compares the first 0x8 bytes of the API prolog (as loaded in memory) with the original first 0x8 bytes.

If it finds a difference, then it concludes that there was an API hook placed in the function prolog. It proceeds to mark the first 0x8 bytes of the function prolog as **PAGE\_EXECUTE\_READWRITE** using **VirtualProtect**, copies the original bytes from the buffer to api\_address and restores the protection of the memory region to **PAGE\_EXECUTE\_READ**.

00B3A263	8875 F8	MOV ESI, DWORD PTR SS: [EBP-8]	API Address as loaded in memory
00B3A266	887D E8	MOV EDI, DWORD PTR SS:[EBP-18]	Buffer corresponding to original bytes of API prolog
00B3A269	8840 EC	MOV ECX, DWORD PTR SS:[EBP-14]	size = 0x8
00B3A26C	FC	CLD	
00B3A26D	F3:A6	REPE CMPS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]	$\leq$ compare the first 0x8 bytes of API prolog with the huffer
00B3A26F	V74 01	JE SHORT 00B3A272	s compare the motoxy bytes of All prolog with the burlet
00B3A271	40	INC EAX	
00B3A272	8945 FØ	MOV DWORD PTR SS:[EBP-10],EAX	
00B3A275	837D FØ 00	CMP DWORD PTR SS:[EBP-10],0	
00B3A279	V74 57	JE SHORT ØØB3A2D2	
00B3A27B	8365 E4 00	AND DWORD PTR SS:[EBP-1C],0	
00B3A27F	58	PUSH EBX	
00B3A280	FF75 F8	PUSH DWORD PTR SS:[EBP-8]	
00B3A283	E8 DCD20000	CALL <isbadwriteptr_stub></isbadwriteptr_stub>	
00B3A288	33F6	XOR ESI,ESI	
00B3A28A	46	INC ESI	
00B3A28B	59	POP ECX	
00B3A28C	59	POP ECX	
00B3A28D	3BC6	CMP EAX,ESI	
00B3A28F	075-19	JNZ SHORT ØØB3A2AA	
00B3A291	8D45 FC	LEA EAX,DWORD PTR SS:[EBP-4]	
00B3A294	50	PUSH EAX	
00B3A295	6A 40	PUSH 40	
00B3A297	53	PUSH EBX	
00B3A298	FF75 F8	PUSH DWORD PTR SS:[EBP-8]	
00B3A29B	E8 0ACF0000	CALL (VirtualProtect_stub)	< patch the API prolog if a book was found
00838280	83C4 10	ADD ESP.10	paten the ran protog in a nook mas round
00838283	8500	TEST EAX,EAX	
00B3A2A5	074 3B	JE SHORT ØØB3A2E2	
00B3A2A7	8975 E4	MOV DWORD PTR SS:[EBP-1C],ESI	
00B3A2AA	887D F8	MOV EDI, DWORD PTR SS: [EBP-8]	
00B3A2AD	8875 E8	MOV ESI, DWORD PTR SS: [EBP-18]	
00B3A2B0	8840 EC	MOV ECX, DWORD PTR SS: [EBP-14]	
00B3A2B3	FC	CLD	
00B3A2B4	F3:A4	REP MOUS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]	copy 0x8 bytes from the buffer to api_address
00B3A2B6	837D E4 01	CMP DWORD PTR SS:[EBP-1C],1	
00B3A2BA	075-13	JNZ SHORT ØØB3A2CF	
00B3A2BC	8D45 FC	LEA EAX,DWORD PTR SS:[EBP-4]	
00B3A2BF	50	PUSH EAX	
00B3A2C0	FF75 FC	PUSH DWORD PTR SS:[EBP-4]	
00B3A2C3	53	PUSH EBX	
00B3A2C4	FF75 F8	PUSH DWORD PTR SS:[EBP-8]	
00B3A2C7	E8 DECE0000	CALL (VirtualProtect_stub)	

In order to test this algorithm, let us set a breakpoint (INT3) at a native API like ZwDelayExecution.

When the above hook checking algorithm detects a difference in the API prolog, it copies the original 0x8 bytes to **ZwDelayExecution**.

Now, let us go to the API **ZwDelayExecution** in Olly Debugger. It still shows us the breakpoint. However, this breakpoint has already been corrupted since the byte 0xCC was overwritten by 0xB8 by our algorithm.

We can confirm this by trying to set a breakpoint at **ZwDelayExecution** once again. Olly Debugger lets us know that the breakpoint was corrupted.



How can we detect the Hook Patching activity?

Since the virus needs to modify the protection of memory region corresponding to API prolog prior to patching the hook, the sandbox can hook **VirtualProtect()** and monitor calls to it. If a binary is attempting to call **VirtualProtect()** on an API imported from system DLL, it is a good indicator that this binary is malicious.

This is also the reason a virus should prefer to skip the hooks rather than patch them. The hook checking algorithm mentioned above could be modified to call APIs through a stub which contains the first N instructions of the API prolog (if a hook is detected). It can use an x86 generic disassembler to calculate the length of instructions.

# **Real World Examples**

Now that we have discussed the various points at which a sandbox can apply inline hooks, let us look at a virus, which was found in the wild, and see how it attempts to detect the API hooks and bypass them.

Below is the algorithm used by the virus:

1. Gets the Function Pointer and passes it to a Generic x86 Disassembler which calculates the length of the first instruction.

2. If the length of the first instruction is 0x2 bytes, then it checks whether the first opcode is **0xeb** (corresponding to a short jmp). If it finds a short jmp, it follows the short jmp and once again calculates the length of first instruction.

If the length of the first instruction is 0x5 bytes, then it checks whether the first opcode is **0xe9** (corresponding to a near jmp). If it finds a near jmp, it follows the near jmp and once again calculates the length of the first instruction.

It keeps repeating the above steps till it finds that the first opcode is not 0xeb or 0xe9 depending upon the length of the first instruction.

3. After this, it copies the first X bytes of the API prolog to a buffer. Here X refers to the length of the first instruction. It then calculates the address of instruction after X bytes in the API prolog and writes that into the buffer prefixed by a near jmp opcode (0xE9)

The jmp\_buffer looks like:

#### [first X bytes of the API prolog][E9][offset to instruction after API prolog].

It repeats these steps for all the APIs it calls to perform malicious activities.

Below screenshot shows the Algorithm along with relevant comments.



Below is an example of **jmp\_buffer** stub for an API with 0x2 bytes as the length of the first instruction:

### RtlComputeCrc32:

7FF80060	SBFF	MOV EDI, EDI	direct the execution to second instruction of
7FF80062	-E9 440D9EFC	JMP ntdll.7C960DAB	
7FF80067	AE	SCAS BYTE PTR ES: [EDI] RUCOMPUTECTC32()	
7FF80068	0D 692F191D	OR EAX,1D192F69	
7FF8006D	^75 BD	JNZ SHORT 7FF8002C	
7FF8006F	33E6	XOR ESP,ESI	
7FF80071	VE3 FF	JECXZ SHORT 7FF80072	
7FF80073	CD 85	INT 85	
7FF80075	43	INC EBX	
7FF80076	V78 66	JS SHORT 7FF800DE	
7FF80078	AD	LODS DWORD PTR DS:[ESI]	
7FF80079	4E	DEC ESI	
7FF8007A	61	POPAD	
7FF8007B	07	???	Unknown command
7FF8007C	67:95	XCHG EAX,EBP	Superfluous prefix
7FF8007E	40	INC EAX	
7FF8007F	10 E9	SBB AL,0E9	
7FF80081	A9 1EB43BC4	TEST EAX,C43BB41E	
7FF80086	AC	LODS BYTE PTR DS:[ESI]	
7FF80087	54	PUSH ESP	
7FF80088	40	INC EAX	
7FF80089	62FF	BOUND EDI,EDI	Illegal use of register
7FF8008B	4A	DEC EDX	
7FF8008C	4F	DEC EDI	
7FF8008D	DC60 4B	FSUB QWORD PTR DS:[EAX+4B]	
7FF80090	EC	IN AL,DX	I/O command
7FF80091	09	LEAVE	
7FF80092	D6	SALC	
7FF80093	^72_8B	JB SHORT 7FF80020	
7FF80095	60	PUSHAD	
7FF80096	B1 98	MOV CL,98	
7FF80098	03DB	ADD EBX,EBX	
7FF8009A	6BDE CD	IMUL EBX,ESI,-33	
7FF8009D	90	NOP	
7FF8009E	AØ 975BC07A	MOV AL, BYTE PTR DS:[7AC05B97]	
7FF800A3	36:66:68 EBE8	PUSH ØE8EB	Superfluous prefix
7FF800A8	0D AF4E60B0	OR EAX, B0604EAF	
7FF800AD	AE	SCAS BYTE PTR ES:[EDI]	
7FF800AE	C085 82A83119 B	ROL BYTE PTR SS:[EBP+1931A882],0BC	Shift constant out of range 131
7FF800B5	92	XCHG EAX,EDX	
7FF800B6	53	PUSH EBX	
7FF800B7	A2 D8BA3344	MOV BYTE PTR DS:[4433BAD8],AL	
7FF800BC	B8 97CC4A17	MOV EAX,174ACC97	

The highlighted region in the memory window below corresponds to jmp\_buffer for RtlComputeCrc32

Address	Hex	du	mp														AS	CI	I				
7FF80060	8B	FF	E9	44 ।	0D	9E	FC	AE	ØD	69	2F	19	1D	75	BD	33	ï	θD.	. An	«. i	∕∔#u <sup>µ</sup>	3	
7FF80070	E6	E3	FF	CD :	85	43	78	66	AD.	4E	61	C7	67	95	40	10	υπ	=;	àCx-	£ ≩N	alFað@	i, I	

Below is an example of **jmp\_buffer** stub for an API with 0x5 bytes as the length of the first instruction:

#### ZwUnmapViewOfSection:

_ن								
7FF80020	B8 0B010000	MOV EAX,10B	< This stub is used t	o redirect the execution to second instruction of				
7FF80025	-E9 CBDE98FC	JMP ntdll.7C90DEF5	7wlInmanViewOfSect	ion				
7FF8002A	BF 1A1235EE	MOV EDI,EE35121A	2110111110101010					
7FF8002F	cc	INTS						
7FF80030	AF	SCAS DWORD PTR ES: LED	[]					
7FF80031	64:BE DC288682	MOU ESI.82862ADC		Superfluous prefix				
7FF80037	35 C7833F57	XOR E8X.573F83C7						
7FF8003C	8378 CC 84	CMP DWORD PTR DS: LEDX-	-34150					
7FF80040	35 648F57BC	XOR EAX. BC578F64						
7FF80045	-73 81	JNB SHORT ZEEZEECS						
7FF80047	41	INC ECX						
7FF80048	60	PUSHOD						
7FF80049	6205 329E130E	BOUND FRX. QUORD PTR D	3: FE139E321					
7FF8004F	D096 6E392279	SHI BYTE PTR DS: [EST+]	7922396E1.1					
75580055	BE 58598850	MOULEST 50885958						
75580050	67.E5	CMC		Superfluous prefix				
7FF8005C	FC			1/0 command				
75580050	54	PUSH ESP		1/0 connaite				
75590055	05	INTO						
75590055	54	PUCH EST						
75590060	06	CMDC DUTE DTD DC.FECT						
75500041		MOLLERD E162060E	, bite tin 25, (2011					
75500044	05	SCOR DWORD DTD FR. FED	11					
75500067		SCAS DWORD FIN ES. LED	1					
75500007		OR FOY IDIOSEG	1					
75500000	00 072F1710	INT CUODE ZEEOOOOC						
75500000	20050	VOD FOD FOT						
75500051	5000	JECVZ CUODI ZEEGOGZO						
75500071		THE OF						
7550073	40	101 85						
75500075	70.66							
75500070	vro 00	LODE DWODD DTD DE FEE						
75500070		DEC FOT						
75500079	40	DEC EST						
7550078	01	POPHD		the loss of the second s				
76680078	67-05	YOUG DOW DDD		Unknown command				
76680070	67:95	AUNG ENA, EBP		Superfluous prefix				
7FF8007E	40							
75500075		TECT FOX CAODDAAF						
75500001	H9 10043004	LODO DUTE DTD DO FEOT						
75500000		DUCH SCD	1					
75500007	40	TUSH ESP						
76680088	40	THU EHA		There is a structure of the structure of				
7668089	6277	BOOND EDI,EDI		Illegal use of register				
The highlighted region in memory window below corresponds to the imp buffer for ZwUnmapViewOfSection								
Oddaoga	Hou dump		LOSCI I					
ZEEGGGGG								
7F700022 B0 05 01 00 02 CB 0E 73 FU BF 1H 12 35 EE CU 1000FT 97 7455FT								
76680030	HF 64 BE DC 2H B	6 82 35 U7 83 3F 57 83						
76680040	35 64 8F 57 BC 7	5 61 41 60 62 05 32 9E	TA OF TA POR SUN DE SAMP SUN D	287				
7FF500501H6 5E 39 22 79 5E 58 59 8B 5H 67 F5 EC 54 CE 55 979"2477(23)*****								
7FF80060 A6 BD 8F A6 63 51 AF AE 0D 69 2F 19 1D 75 BD 33 ª"A≧cQ∾≪.i/↓#u"3								

Flaws in this algorithm:

At first the above algorithm looks convincing at bypassing the API hooks of a sandbox. However, if you look closely at the algorithm, there are several shortcomings, which would result in the virus not being able to bypass the API hooks.

1. It does not check for all the possible control transfer instruction opcodes (there is no check for 0xe8).

2. When it finds an opcode 0xeb or 0xe9 at the start of API prolog, it follows the hooked routine address. Even if it now skips the first instruction of hooked routine address, the execution will still be redirected to API hook of the sandbox.

Example:

Let's consider, **ZwDelayExecution** with an inline hook from the Sandbox:

jmp <hooked\_routine>
mov edx, 0x7ffe0300
call dword ptr [edx]

hooked\_routine:

push ebp mov ebp, esp

a) It detects the inline hook and calculates the address of hooked\_routine.b) It follows the hooked\_routine and now calculates the length of first instruction of hooked\_routine which is 0x1 in this case.

c) Now the length of first instruction is neither 0x2 nor 0x5, so it proceeds to copy the first byte from hooked\_routine to its jmp\_buffer and calculate address of second instruction of hooked\_routine.

The next time, virus calls ZwDelayExecution() through its jmp\_buffer, the execution would still be redirected to the hooked\_routine of the sandbox.

As a result of this, the API hook checking algorithm used above is ineffective in bypassing the Sandbox API hooks.

Surprisingly, this algorithm was used in a large number of viruses recently. This shows that the algorithms used for evading sandbox API hooks still need improvement.

# KiFastSystemCall Hook

This virus also used another API hook checking algorithm for **KiFastSystemCall** stub. Interestingly, this algorithm is correct and also used not so often in viruses.

Below is the algorithm:

1. Calculate address of KiFastSystemCall.

2. Check for a short jmp (opcode: 0xeb) at KiFastSystemCall. We will see later in more detail the reason why it checks only for 0xeb and not 0xe9 or 0xe8.

3. Once it finds a short jmp, it follows the short jmp and checks for a push instruction (opcode: 0x68).

4. If it finds a push instruction, it marks memory region pointed to by the argument of push instruction as PAGE\_EXECUTE\_READWRITE.

5. Now, it copies the 0x5 bytes corresponding to KiFastSystemCall stub to the above memory region.

As a result of this, even though the KiFastSystemCall hook remains intact, the execution would still be redirected to code specific to KiFastSystemCall.

Below is the screenshot specific to the algorithm mentioned above with relevant comments:

7FF90EDD8BECHOU EBP,ESP7FF90EDFSIPUSH ECX7FF90EDFSIPUSH 7FF905887FF90EES68 8085F97FPUSH 7FF905887FF90EES68 2805F97FPUSH 7FF905287FF90EES68 2805F97FPUSH 7FF905287FF90EERFF15 DC00F97FCALL DWORD PTR D5:[7FF9080C]7FF90EF850PUSH EAX7FF90EF1FF15 7800F97FCALL DWORD PTR D5:[7FF90878]7FF90EF7SSC0TEST ERX,ERX7FF90EF88638EB7FF90EF9\$74 48JE SHORT 7FF90F437FF90EF9\$75 43JN2 SHORT 7FF90F437FF90EF9\$75 43JN2 SHORT 7FF90F437FF90EF9\$75 43JN2 SHORT 7FF90F437FF90F090FB648 01MOUZX ECX,BYTE PTR D5:[EAX+1]7FF90F098940 FCMOU DWORD PTR S5:[EEP-4],ECX7FF90F09\$109 00FFFFFFOR ECX,FFFFFF047FF90F188048 02LEA EAX,DWORD PTR D5:[EAX+ECX+2]7FF90F14804408 02LEA EAX,DWORD PTR D5:[EAX+ECX+2]7FF90F188038 68C/the WTE PTR D5:[EAX],687FF90F188038 68C/the WTE PTR D5:[EAX],687FF90F18804408 02LEA EAX,DWORD PTR D5:[EAX]7FF90F18804408 02LEA EAX,DWORD PTR D5:[EAX]7FF90F188038 68C/the WTE PTR		PUSH EBP	55	7FF90EDC
7FF90EDF51PUSH ECX7FF90EE068 A005P97FPUSH 7FF90528ASCII "KiFastSystenCall"7FF90EE668 2005P97FPUSH 7FF90528ASCII "ntdll.dl"7FF90EF758PUSH 7FF90528ASCII "ntdll.dl"7FF90EF858PUSH EAXCall to GetModuleHandleA through stub7FF90EF7S50PUSH EAXCall to GetProcAddress through Stub7FF90EF78508TEST EAX.ERXCall to GetProcAddress through Stub7FF90EF88038 EBCHE BYTE PTR DS: (FFF90678)Call to GetProcAddress through Stub7FF90EF9~74 48JE SHORT 7FF90F43Check for a short jmp at KiFastSystemCall7FF90EF80858 EBCHE BYTE PTR DS: (EAX+1)Check for a short jmp at KiFastSystemCall7FF90EF6~75 43JN2 SHORT 7FF90F43Check for a short jmp at KiFastSystemCall7FF90E70085648 01MOUZX ECX.BYTE PTR DS: (EAX+1)FF90F747FF90E798409FEST CL, CLFEST CL, CL7FF90E798409JNS SHORT 7FF90F14FOIlow the short jmp7FF90E748940 PCMOU DWORD PTR S: (EEP+4), ECXFollow the short jmp7FF90F14804408 02LEA EAX, DWORD PTR S: (EEX+ECX+2)Follow the short jmp7FF90F14804408 02LEA EAX, DWORD PTR S: (EEX+ECX+2)Follow the short jmp7FF90F14804408 02LEA EAX, DWORD PTR DS: (EAX+ECX+2)Follow the short jmp7FF90F14804408 02LEA EAX, DWORD PTR DS: (EAX+ECX+2)Follow the short jmp7FF90F15RECX, FFFFF6RECheck for a Push Instruction<		MOV EBP, ESP	8BEC	7FF90EDD
7FF90EE068 R005F97FPUSH 7FF905R8ASCII "KiFastSystemCall"7FF90EE568 2005F97FPUSH 7FF90528ASCII "ntdil.dli"7FF90EF6FF15 DC00F97FCRLL DWORD PTR DS:C7FF900DC3Call to GetModuleHandleR through stub7FF90EF7S50PUSH EAXCall to GetProcAddress through Stub7FF90EF7S500TEST EAX,EAXCall to GetProcAddress through Stub7FF90EF7S74 48JE SHORT 7FF90F43Call to GetProcAddress through Stub7FF90EF88038 EBCHP BYTE PTR DS:CEAX;0EBCheck for a short jmp at KiFastSystemCall7FF90EF88038 EBCHP BYTE PTR DS:CEAX+11Check for a short jmp at KiFastSystemCall7FF90F960FB648 01M002X ECX,BYTE PTR DS:CEAX+11Check for a short jmp at KiFastSystemCall7FF90F968409TEST CL,CLFST CL,CL7FF90F988109 00FFFFFFOR ECX,FFFFF087FF90F14S940 FCM00 DWORD PTR SS:CEBX+41,ECX7FF90F188038 62LER EAX,DWORD PTR DS:CEAX+227FF90F188048 02LER EAX,DWORD PTR DS:CEAX+217FF90F188088 63CMP BYTE PTR DS:CEAX+227FF90F188088 63CMP BYTE PTR DS:CEAX+647FF90F188088 63CMP BYTE PTR DS:CEAX+647FF90F14SHOR 864 <td< td=""><td></td><td>PUSH ECX</td><td>51</td><td>7FF90EDF</td></td<>		PUSH ECX	51	7FF90EDF
7FF90EES68 2005F97FPUSH 7FF90528RSCII "htdll.dll"7FF90EEAFFIS DC00F97FCALL DWORD PTR DS:[7FF900DC]Call to GetHoduleHandleA through stub7FF90EF1FFIS 7800F97FCALL DWORD PTR DS:[7FF90078]Call to GetProcAddress through Stub7FF90EF28500TEST ERX, EAXCall to GetProcAddress through Stub7FF90EF38500TEST ERX, EAXCall to GetProcAddress through Stub7FF90EF48500TEST ERX, EAXCall to GetProcAddress through Stub7FF90EF58638 EBCHP BVTE PTR DS:[EAX],0EBCheck for a short jmp at KiFastSystemCall7FF90EF68648 01MOUZX ECX, BYTE PTR DS:[EAX+1]7FF90E7694C9TEST CL, CL7FF90E7694C9TEST CL, CL7FF90E7881C9 00FFFFFF7FF90F7081C9 00FFFFFF007FF90F14804408 027FF90F14804408 027FF90F35Check for a Push Instruction7FF90F368038 667FF90F37MC Sh00R TFF90F36	ASCII "KiFastSystemCall"	05F97F PUSH 7FF905A8	68 A805F97F	7FF90EE0
7FF908ER       FF15 DC00F97F       CRLL DWORD PTR DS:[7FF908DC]       Call to GetModuleHandleR through stub         7FF908EF8       50       PUSH ERX       Call to GetModuleHandleR through stub         7FF908EF7       SE00       TEST EAX,ERX       Call to GetProcAddress through Stub         7FF908EF7       SSC0       TEST EAX,ERX       Call to GetProcAddress through Stub         7FF908EF8       S808       CHP BVTE PTR DS:[ERX],0EB       Check for a short jmp at KiFastSystemCall         7FF908F8       S940 FC       MOUZX ECX,8VTE PTR DS:[EAX+1]       Check for a short jmp at KiFastSystemCall         7FF908F8       S940 FC       MOUZX ECX,8VTE PTR DS:[EBP-4],ECX       Check for a short jmp at KiFastSystemCall         7FF908F9       796968       S1C9 00FFFFFF       OR ECX,FFFFF604       Check for a short jmp         7FF908F14       S0408 PC       HOU DWORD PTR SS:[EEP-4],ECX       Follow the short jmp         7FF908F14       S04408 02       LEA EAX,DWORD PTR DS:[EAX+ECX+2]       Follow the short jmp         7FF908F14       S04408 02       LEA EAX,DWORD PTR DS:[EAX+ECX+2]       Follow the short jmp         7FF908F14       S04408 02       LEA EAX,DWORD PTR DS:[EAX+ECX+2]       Follow the short jmp         7FF908F14       S04408 02       LEA EAX,DWORD PTR DS:[EAX+ECX+2]       Follow the short jmp         <	ASCII "ntdll.dll"	05F97F PUSH 7FF90528	68 2805F97F	7FF90EE5
7FF90EF0       50       PUSH ERX       Call to GetProcAddress through Stub         7FF90EF1       FF15       75800F97F       CALL DWORD PTR DS:[7FF90078]       Call to GetProcAddress through Stub         7FF90EF5       SS00       TEST ERX,ERX       TEST ERX,ERX         7FF90EF9       74 48       JE SHORT 7FF90F43       Check for a short jmp at KiFastSystemCall         7FF90EF8       8038 EB       CHP BYTE PTR DS:[ERX],0EB       Check for a short jmp at KiFastSystemCall         7FF90EF4       754 43       JN2 SHORT 7FF90F43       TEST CL,CL         7FF90E74       8940 FC       HOU DWORD PTR SS:[EBP-4],ECX       TEST CL,CL         7FF90E79       7409       JNS SHORT 7FF90F14       TEST CL,CL         7FF90E79       79 99       JNS SHORT 7FF90F14       TEST CL,CL         7FF90E71       8940 FC       HOU DWORD PTR SS:[EBP-4],ECX       TEST CL,CL         7FF90E71       8940 FC       HOU DWORD PTR SS:[EBP-4],ECX       TEST CL,CL         7FF90E71       8940 FC       HOU DWORD PTR SS:[EBP-4],ECX       TEST CL,CL         7FF90E74       804FFFF       RECX,FFFF808       Follow the short jmp         7FF90E71       8940 FC       HOU DWORD PTR DS:[ERX+ECX+2]       Follow the short jmp         7FF90E718       8088 68       CMP BYTE PTR DS:[ER	Call to GetModuleHandleA through stub	DC00F97F CALL DWORD PTR DS:[7FF900DC]	FF15 DC00F97F	7FF90EEA
7FF90EF1       FF15       7800F97F       CRLL DWORD PTR DS:(7FF90078)       Call to GetProcAddress through Stub         7FF90EF7       SC0       TEST EAX,EAX       TEST EAX,EAX         7FF90EF7       V74 48       JE SHORT 7FF90F43       Check for a short jmp at KiFastSystemCall         7FF90EF8       8088 EB       CHP BYTE PTR DS:(EAX),0EB       Check for a short jmp at KiFastSystemCall         7FF90E78       V75 43       JN2 SHORT 7FF90F43       Check for a short jmp at KiFastSystemCall         7FF90E78       0FB648 01       M0U2X ECX,BYTE PTR DS:(EAX+1)       Check for a short jmp at KiFastSystemCall         7FF90E78       8609       TEST CL,CL       Check for a short jmp at KiFastSystemCall       Check for a short jmp at KiFastSystemCall         7FF90E78       8409       TEST CL,CL       Check for a short jmp at KiFastSystemCall       Check for a short jmp at KiFastSystemCall         7FF90E79       V79       909       JNS SHORT 7FF90F14       Check for a short jmp at KiFastSystemCall       Check for a short jmp at KiFastSystemCall         7FF90E78       8109       00FFFFFF       OR ECX,FFFFF00       Check for a f		PUSH EAX	50	7FF90EF0
7FF90EF7         SSC0         TEST ERX,ERX           7FF90EF9         ~74         48         JE SHORT 7FF90F43           7FF90EF8         8038         CHP BVTE PTR DS:LERX!J0EB         Check for a short jmp at KiFastSystemCall           7FF90EF8         ~75         43         JN2 SHORT 7FF90F43         Check for a short jmp at KiFastSystemCall           7FF90E78         0FB648         01         MOUZX ECX,BVTE PTR DS:LERX+11         FFF90F04           7FF90F04         0840 FC         MOUZX ECX,BVTE PTR DS:LERX+11         FFF90F04           7FF90F07         8409         TEST CL,CL         FF90F14           7FF90F18         S010 00FFFFFF         OR ECX,FFFFFF04           7FF90F14         804408         02         LER ERX,DWORD PTR DS:LERX+ECX+21         Follow the short jmp           7FF90F18         8088 68         CHP BVTE PTR DS:LERX).68         Check for a Push Instruction           7FF90F18         8088 68         CHP BVTE PTR DS:LERX).68         Check for a Push Instruction	Call to GetProcAddress through Stub	7800F97F CALL DWORD PTR DS:[7FF90078]	FF15 7800F97F	7FF90EF1
7FF90EF9       74 48       JE SHORT 7FF90F43         7FF90EF8       8038 EB       CHP BVTE PTR DS:LEAX1,0EB       Check for a short jmp at KiFastSystemCall         7FF90EF8       75 43       JN2 SHORT 7FF90F43       Check for a short jmp at KiFastSystemCall         7FF90E76       75 43       HOUZX ECX,BVTE PTR DS:LEAX1,10       Check for a short jmp at KiFastSystemCall         7FF90F04       045648 01       HOUZX ECX,BVTE PTR DS:LEAX1,11       Check for a short jmp at KiFastSystemCall         7FF90F04       045649       TEST CL, CL       FST CL, CL       FST CL, CL         7FF90F08       3(29 00FFFFF       OR ECX,FFFFF00       FSILEEP-41,ECX         7FF90F14       8940 62       LEA EAX,DWORD PTR DS:LEAX+ECX+21       Follow the short jmp         7FF90F14       804408 02       LEA EAX,DWORD PTR DS:LEAX+ECX+21       Follow the short jmp         7FF90F18       8088 68       CHP BVTE PTR DS:LEAX1,68       Check for a Push Instruction         7FF90F18       8088 68       CHP BVTE PTR DS:LEAX1,68       Check for a Push Instruction		TEST EAX,EAX	8500	7FF90EF7
7FF90EFB     8038 EB     CHP BVTE PTR DS:(EAX),0EB     Check for a short jmp at KiFastSystemCall       7FF90EFE     75 43     JN2 SHORT 7FF90F43       7FF90F00     0FB648 01     MOUZX ECX,BVTE PTR DS:(EAX+1)       7FF90F04     8940 FC     MOU DWORD PTR SS:(EBP+4),ECX       7FF90F069     769     JNS SHORT 7FF90F14       7FF90F069     799     JNS SHORT 7FF90F14       7FF90F071     8040 FC     MOU DWORD PTR SS:(EBP+4),ECX       7FF90F08     81C9 00FFFFFF     OBFECK,FFFFF00       7FF90F14     804488 02     LEA EAX,DWORD PTR DS:(EAX+ECX+2)       7FF90F18     80488 62     CMP BVTE PTR DS:(EAX),68       7FF90F18     L75 21     JNC SHORT 7FF90F2		JE SHORT 7FF90F43	v74 48	7FF90EF9
7FF906FE     \75     \	Check for a short jmp at KiFastSystemCall	EB CMP BYTE PTR DS:[EAX],0EB	8038 EB	7FF90EFB
7FF90F00         0FB648         01         MOUZX ECX,BVTE PTR DS:[EBX+1]           7FF90F04         8940 FC         MOU DWORD PTR SS:[EBP-4],ECX           7FF90F07         84C9         TEST CL,CL           7FF90F08         JNS SHORT 7FF90F14           7FF90F08         81C9 00FFFFF           08         EXAMPLE           7FF90F14         8940 FC           7FF90F14         8940 FC           7FF90F14         894498           7FF90F14         804408           7FF90F14         804408           7FF90F18         8038 68           C1P BVTE PTR DS:[EBX],68           Check for a Push Instruction           7FF90F18         SHORT 7FF90F26		JNZ SHORT 7FF90F43	V75 43	7FF90EFE
7FF90F04         894D FC         MOU DWORD PTR SS:LEBP-4],ECX           7FF90F07         84C9         TEST CL,CL           7FF90F09         79 09         JNS SHORT 7FF90F14           7FF90F08         81C9         00FFFFFF           7FF90F08         81C9         00FFFFFF           7FF90F11         894D FC         MOU DWORD PTR SS:LEBP-4],ECX           7FF90F14         804408 02         LER ERX,DWORD PTR DS:LERX+ECX+2]           7FF90F14         804408 02         LER ERX,DWORD PTR DS:LERX+ECX+2]           7FF90F18         8084 68         CMP BYTE PTR DS:LERX],68           7FF90F18		8 01 MOVZX ECX, BYTE PTR DS:[EAX+1]	0FB648 01	7FF90F00
7FF90F07         84C9         TEST CL_CL           7FF90F09         .79         09         JNS SHORT 7FF90F14           7FF90F08         81C9         00FFFFFF         OR ECX,FFFFFP00           7FF90F08         81C9         00FFFFFF         OR ECX,FFFFFP00           7FF90F14         804408         02         LEA EAX,DWORD PTR DS:LEEX+22         Follow the short jmp           7FF90F18         808468         CMP BYTE PTR DS:LEEX1,68         Check for a Push Instruction           7FF90F18		FC MOV DWORD PTR SS:[EBP-4],ECX	894D FC	7FF90F04
7FF90F09     7F90F09     7F90F14       7FF90F08     81C9     00FFFFF     0R       7FF90F14     8940     FC     MOU DU0RD PTR       7FF90F14     804408     02     LEA       7FF90F14     804408     02     LEA       7FF90F14     804408     02     LEA       7FF90F18     8088     68     C1HP BYTE PTR       7FF90F18     8088     68     C1HP BYTE PTR       7FF90F18     8088     68     C1HP BYTE PTR       7FF90F18     8088     68     Check for a Push Instruction		TEST CL,CL	8409	7FF90F07
7FF90F08         81C9         00FFFFFF         OR         ECX,FFFFF00           7FF90F11         8940         FC         MOU DW0R0         FTR <ss:lebp-4],ecx< td="">           7FF90F14         804408         02         LEA         EAX,DW0R0         PTR<ds:leax+ecx+2]< td="">         Follow the short jmp           7FF90F18         8038         68         CMP         BYTE         PTR         DS:LEAX],68         Check for a Push Instruction           7FF90F18        </ds:leax+ecx+2]<></ss:lebp-4],ecx<>		JNS SHORT 7FF90F14	V79-09	7FF90F09
7FF90F11         894D FC         MOU DWORD PTR SS:[EBP-4],ECX           7FF90F14         804408 02         LEA EAX,DWORD PTR DS:[EAX+ECX+2]         Follow the short jmp           7FF90F18         8088 68         CMP BYTE PTR DS:[EAX],68         Check for a Push Instruction           7FF90F18		00FFFFFF OR ECX,FFFFFF00	81C9 00FFFFFF	7FF90F0B
7FF90F14         804408         02         LER ERX,DWORD PTR DS:[ERX+ECX+2]         Follow the short jmp           7FF90F18         8088 68         CMP BYTE PTR DS:[ERX],68         Check for a Push Instruction           7FF90F18		FC MOV DWORD PTR SS:[EBP-4],ECX	8940 FC	7FF90F11
7FF90F18 8038 68 CMP BYTE PTR DS:[EAX],68 Check for a Push Instruction 7FF90F1875 21 JNZ SHORT 7FF90F3E	Follow the short jmp	8 02 LEA EAX, DWORD PTR DS: [EAX+ECX+2]	804408 02	7FF90F14
ZEF90F18 J75 21 JNZ SHORT ZEF90F3E	Check for a Push Instruction	68 CMP BYTE PTR DS:[EAX],68	8038 68	7FF90F18
		JNZ SHORT 7FF90F3E	V75 21	7FF90F1B
7FF90F1D 56 PUSH ESI		PUSH ESI	56	7FF90F1D
7FF90F1E 8B70 01 MOV ESI,DWORD PTR DS:[EAX+1] Read the argument of Push instruction	Read the argument of Push instruction	01 MOV ESI, DWORD PTR DS: [EAX+1]	8B70 01	7FF90F1E
7FF90F21 8D45 FC LEA EAX,DWORD PTR SS:[EBP-4]		FC LEA EAX, DWORD PTR SS: [EBP-4]	8D45 FC	7FF90F21
7FF90F24 50 PUSH EAX		PUSH EAX	50	7FF90F24
7FF90F25 6A 40 PUSH 40		PUSH 40	6A 40	7FF90F25
7FF90F27 6A 05 PUSH 5		PUSH 5	6A 05	7FF90F27
7F90F29 56 PUSH ESI		PUSH ESI	56	7FF90F29
7FF90F2A FF15 9C00F97F CALL DWORD PTR DS:[7FF9009C] Call to VirtualProtect through Stub	Call to VirtualProtect through Stub	9C00F97F CALL DWORD PTR DS:[7FF9009C]	FF15 9C00F97F	7FF90F2A
7FF90F30 56 PUSH ESI		PUSH ESI	56	7FF90F30
7F90F31 6A 05 PUSH 5		PUSH 5	6A 05	7FF90F31
7F90F33 68 A005F97F PUSH 7FF905A0		05F97F PUSH 7FF905A0	68 A005F97F	7FF90F33
7FF90F38 E8 SD0C0000 CALL <copybuffer> Copy the original bytes of KiFastSystemCall to buffer</copybuffer>	Copy the original bytes of KiFastSystemCall to buffer	0C0000 CALL <copybuffer></copybuffer>	E8 5D0C0000	7FF90F38
7FF90F3D SE POP ESI		POP ESI	SE	7FF90F3D
7FF90F3E 33C0 XOR EAX,EAX		XOR EAX,EAX	3300	7FF90F3E
7F90F40 40 INC EAX		INC EAX	40	7FF90F40
7F90F41 C9 LEAVE		LEAVE	C9	7FF90F41
7FF90F42 C3 RETN		RETN	C3	7FF90F42
7FF90F43 33C0 XOR EAX,EAX ntdll.KiFastSystemCall	ntdll.KiFastSystemCall	XOR EAX,EAX	3300	7FF90F43
7F90F45 C9 LEAVE		LEAVE	C9	7FF90F45
7F90F46 C3 RETN		RETN	C3	7FF90F46

### **Betabot Hook**

Now, let us discuss why the previous **KiFastSystemCall** hook checking algorithm was only checking for a short jmp at KiFastSystemCall.

To understand this better, let us first analyze the SystemCallStub.

On Windows XP SP3, the SystemCallStub looks like:

mov edx, esp

sysenter retn

This has a size of 0x5 bytes. Now, you might ask why we cannot apply a simple inline hook?

The reason being, if we overwrite the above SystemCallStub with an inline hook, we would end up overwriting the **KiFastSystemCallRet** as well.

What is the consequence of overwriting KiFastSystemCallRet?

When the program enters kernel mode after execution of sysenter, it finds the address of **KiFastCallEntry** using the **SYSENTER\_EIP\_MSR** (0x176).

This part of user mode to kernel mode transition is not impacted even if we overwrite **KiFastSystemCallRet** instruction in user mode.

However, after completing the execution in kernel mode, when the control is returned to user mode, sysexit is triggered.

Kernel mode knows where to return in the user mode based on the value of **SystemCallRet** member of the **\_KUSER\_SHARED\_DATA** structure.

The address of **KiFastSystemCallRet** is stored at offset 0x304 in the \_KUSER\_SHARED\_DATA structure. Also, this structure is not writable in the user mode, so it is not possible to modify it.

```
0:007> dt nt!_KUSER_SHARED_DATA 0x7ffe0000
ntdll!_KUSER_SHARED_DATA
.....
+0x300 SystemCall : 0x7c90e4f0
+0x304 SystemCallReturn : 0x7c90e4f4
```

So, how can we hook the KiFastSystemCall stub?

To understand this better, let us look at the method used by Betabot to apply this hook.

7C90E4F0	↓EB 03	JMP SHORT ntdll.7C90E4F5					
7C90E4F2	0F34	SYSENTER					
7C90E4F4	C3	RETN					
7C90E4F5	68 24F5F37F	PUSH 7FF3F524 < This redirects execu	tion to Betabot's SystemCall				
7C90E4FA	C3	RETN checking subrouting					
7C90E4FB	90	NOP					
7C90E4FC	8D6424 00	LEA ESP, DWORD PTR SS: [ESP]					
7C90E500	805424 08	LEA EDX,DWORD PTR SS:[ESP+8]					
7C90E504	CD 2E	INT 2E					
7C90E506	C3	RETN					

As shown in the screenshot above, it places a short jmp at the first instruction of KiFastSystemCall which redirects the control to KiFastSystemCall + 0x5.

Here, it places a sequence of push and retn instruction to simulate a jmp instruction to redirect the control flow to its system call checking subroutine.

We were able to place a short jmp at the start of KiFastSystemCall conveniently because what follows a SystemCallStub is the old mechanism of user mode to kernel mode transition, which is not used in modern operating systems.

This is a good method of applying hooks at the System Call level. While in this case, this method was used by Betabot, it is also possible for a Sandbox to use similar technique for API hooking.

The KiFastSystemCall hook detection routine we discussed previously can easily bypass such System Call hooks.

### **Conclusion**

After reading this paper, the reader should be able to comprehend the various evasion techniques which are being used by viruses in the wild as well as methods which can be used to prevent such evasion techniques.

As can be seen from the various topics we discussed in this paper, the usage of evasion techniques in viruses are still improving and the importance of evading automated sandbox analysis is increasing.

### Appendix I

The code below can be used to detect the presence of virtualization software and also identify its type. I will be maintaining this code on github here: https://gist.github.com/c0d3inj3cT/c68a203c2c1224df55b3.

Methods for detecting more virtualization softwares like Virtual PC and Virtual Box need to be added.

```
#include <windows.h>
#include <stdio.h>
#include <TlHelp32.h>
#include <Setupapi.h>
#include <string.h>
/*
VM Buster
Author: Sudeep Singh
*/
```

```
void vmx check();
void process name check();
void class name check();
void cpuid check();
void cpu cores check();
void registry check();
void devices check();
void drivers check();
int main(int argc, char **argv)
{
process name check();
class_name check();
vmx_check();
cpuid check();
cpu_cores_check();
registry check();
devices check();
drivers check();
return 0;
}
void process name_check()
{
    HANDLE psnap;
    PROCESSENTRY32 pe;
    int i=0;
    char *process name[] = {"regshot.exe", "wireshark.exe", "vmtoolsd.exe",
"vboxtray.exe", "vboxservice.exe", "filemon.exe", "procmon.exe",
"vmacthlp.exe"};
    pe.dwSize = sizeof(PROCESSENTRY32);
    psnap = CreateToolhelp32Snapshot(TH32CS SNAPPROCESS, 0);
    if(!Process32First(psnap, &pe))
    {
        printf("There was an error in retrieving the process information\n");
        return;
    }
    while(Process32Next(psnap, &pe))
    {
        i=0;
        while(i != 8)
        {
            if(lstrcmpi(process name[i], pe.szExeFile) == 0)
            {
                printf("Found process: %s\n", pe.szExeFile);
            }
            i++;
        }
    }
    return;
}
```

```
void cpu cores check()
{
    int i=0;
     asm
    {
       pushad
       mov eax, dword ptr fs:[0x18];
       mov eax, dword ptr ds:[eax+0x30]
        mov eax, dword ptr ds:[eax+0x64];
        cmp eax, 0x1
        jnz done
       xor eax, eax
       inc eax
        mov i, eax
        done:
       popad
    }
    if(i==1)
    {
        printf("Only 1 CPU core assigned to the VM\n");
    }
   return;
}
void cpuid check()
{
   int i=0;
    __asm
    {
       pushad
       mov eax, 0x1
       cpuid
        and ecx, 0x1
       cmp ecx, 0x1
       jnz done
       xor eax, eax
        inc eax
        mov i, eax
        done:
        popad
    }
    if(i == 1)
    {
        printf("Hypervisor found\n");
    }
   return;
}
void class name check()
{
```

```
char *window names[] = {"VMDisplayChangeControlClass",
"VMwareDragDetWndClass", "vmtoolsdControlWndClass", "VMwareTrayIcon"};
    int i=0;
    while(i < 5)</pre>
    {
        if(FindWindow(window names[i], NULL) != NULL)
        {
            printf("Found window name: %s\n", window names[i]);
        }
        i++;
    }
    return;
}
void registry check()
{
    HKEY hkey;
    char *buffer;
    int i=0, j=0;
    int size = 256;
    char *vm names[] = {"vmware", "qemu", "xen"};
    buffer = (char *) malloc(sizeof(char) * size);
    RegOpenKeyEx (HKEY LOCAL MACHINE,
"SYSTEM\\ControlSet001\\Services\\Disk\\Enum", 0, KEY READ, &hkey);
    RegQueryValueEx(hkey, "0", NULL, NULL, buffer, &size);
    while(*(buffer+i))
    {
        *(buffer+i) = (char) tolower(*(buffer+i));
        i++;
    }
    while(j < 3)</pre>
    {
        if(strstr(buffer, vm names[j]) != NULL)
        {
            printf("Found string %s in Registry\n", vm names[j]);
        }
        j++;
    }
    return;
}
void vmx check()
{
    int i=0;
     asm
    {
        pushad
        mov eax, 0x564d5868
        mov edx, 0x5658
        mov ecx, Oxa
```

```
in eax, dx
        cmp ebx, 0x564d5868
        jnz done
        xor eax, eax
        inc eax
        mov i, eax
        done:
        popad
    }
    if(i == 1)
    {
        printf("Found VMX backdoor\n");
    }
    return;
}
void devices check()
{
    HDEVINFO devinfo;
    DWORD size;
    char *buffer;
    char *vm names[] = {"vmware", "gemu", "xen"};
    int i=0,j=0,k=0;
    SP DEVINFO DATA DeviceInfoData;
    DeviceInfoData.cbSize = sizeof(SP DEVINFO DATA);
    devinfo = SetupDiGetClassDevs(0,0,0,6);
    while (SetupDiEnumDeviceInfo(devinfo, i, &DeviceInfoData) != 0)
    {
        j=k=0;
        SetupDiGetDeviceRegistryProperty(devinfo, &DeviceInfoData, 0, 0, 0,
0, &size);
        buffer = (char *) calloc(0x40, size);
        SetupDiGetDeviceRegistryProperty(devinfo, &DeviceInfoData, 0, 0,
buffer, size, 0);
        while(*(buffer+j))
        {
            *(buffer+j) = (char) tolower(*(buffer+j));
            j++;
        }
        while(k < 3)</pre>
        Ł
            if(strstr(buffer, vm names[k]) != NULL)
            {
                printf("Found Device Name: %s\n", buffer);
            }
            k++;
        }
        i++;
    }
    return;
}
```

```
void drivers check()
{
    char buffer[256];
    char *basedir="c:\\windows\\system32\\drivers\\";
    char
*driver names[]={"vmci.sys", "vmhqfs.sys", "vmmouse.sys", "vmscsi.sys", "vmusbmou
se.sys", "vmx svga.sys", "vmxnet.sys", "VBoxMouse.sys"};
    int i=0;
    while(i < 8)</pre>
    {
        memset (buffer, ' \setminus 0', 256);
        strcpy(buffer,basedir);
        strcat(buffer,driver_names[i]);
        if (GetFileAttributes (buffer) != INVALID FILE ATTRIBUTES)
        {
             printf("Found driver: %s\n",driver names[i]);
        }
        i++;
    }
    return;
}
```

### Appendix II

The code below can be used to parse the export directory of a module, enumerate all the exported functions and find their addresses. For each function, we could perform some operations like check the function prolog and identify if it has a standard prolog. This code could also be modified to identify any API which has been hooked by checking for presence of opcodes 0xe8, 0xe9 or 0xeb at API prolog.

```
#include <windows.h>
#include <stdio.h>
/*
Export Directory Parser
Author: Sudeep Singh
*/
int main(int argc, char **argv)
{
    HANDLE hModule;
    DWORD address;
    char prolog[] = {0x8b, 0xff, 0x55, 0x8b, 0xec};
    char *prolog_address = prolog;
    BYTE *buffer;
    int i=0, j=0, num=0, result=0;
    char *api name="";
```

```
buffer = (BYTE *) malloc(sizeof(BYTE) * 5);
    if(argc < 2)</pre>
    {
        printf("usage: export parser.exe <module name>\n");
        exit(0);
    }
    hModule = LoadLibraryA(argv[1]);
     asm
    {
       pushad
       mov eax, hModule
        mov ebx, dword ptr ds:[eax+0x3c]
        add ebx, eax
        add eax, dword ptr ds:[ebx+0x78]
        mov edx, dword ptr ds:[eax+0x18]
        mov num, edx
        popad
    }
    printf("Total number of functions imported from %s are %x\n", argv[1],
num);
    while(i < num)</pre>
    {
     _asm
    {
        pushad
       mov edx, i
       mov eax, hModule
        mov ecx, eax
        mov ebx, dword ptr ds:[eax+0x3c]
        add ebx, eax
        add eax, dword ptr ds:[ebx+0x78]
        mov ebx, dword ptr ds:[eax+0x20]
        mov eax, ecx
        add ebx, eax
        add eax, dword ptr ds:[ebx+edx*4]
        mov api name, eax
        popad
    }
    address = (DWORD) GetProcAddress(hModule, api name);
    memcpy(buffer, (BYTE *)address, 5);
    result = 0;
    __asm
    {
        pushad
       mov eax, buffer
       mov ebx, prolog address
        xor ecx, ecx
        xor edx, edx
```

```
xor esi, esi
    repeat:
    mov cl, byte ptr ds:[eax+esi]
    mov dl, byte ptr ds:[ebx+esi]
    cmp cl, dl
    jnz done
    inc esi
    cmp esi, 0x5
    jnz repeat
    xor esi, esi
    inc esi
    mov result, esi
    done:
    popad
}
if(result == 1)
{
    j++;
    printf("%s | %x\n", api name, address);
}
i++;
}
printf("Number of functions with a standard prolog: %x\n", j);
return 0;
```

### Appendix III

The code below can be used to unlink any module from Process Environment Block. This would result in the module not showing up in the list of loaded modules in a Debugger, as well as Window APIs such as Module32First()/Module32Next() and GetModuleHandle() will not be able to find the module.

Credit for this code goes to Pnluck from OpenRCE.

}

```
#include <windows.h>
#ifndef UNICODE_STRING
typedef struct _UNICODE_STRING {
   USHORT Length;
   USHORT MaximumLength;
   PWSTR Buffer;
} UNICODE_STRING, *PUNICODE_STRING;
#endif
#ifndef LDR_MODULE
typedef struct _LDR_MODULE {
 LIST_ENTRY InLoadOrderModuleList;
 LIST_ENTRY InMemoryOrderModuleList;
```

```
LIST ENTRY InInitializationOrderModuleList;
PVOID BaseAddress;
PVOID EntryPoint;
ULONG SizeOfImage;
UNICODE STRING FullDllName;
UNICODE STRING BaseDllName;
ULONG Flags;
SHORT LoadCount;
SHORT TlsIndex;
LIST ENTRY HashTableEntry;
ULONG TimeDateStamp;
} LDR MODULE, *PLDR MODULE;
#endif
#ifndef PEB LDR DATA
typedef struct PEB LDR DATA
{
         ULONG Length;
         UCHAR Initialized;
         PVOID SsHandle;
         LIST ENTRY InLoadOrderModuleList;
         LIST ENTRY InMemoryOrderModuleList;
         LIST ENTRY InInitializationOrderModuleList;
         PVOID EntryInProgress;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
#endif
BOOL APIENTRY DllMain (HMODULE hModule, DWORD ul reason for call, LPVOID
lpReserved)
{
    if (ul reason for call == DLL PROCESS ATTACH)
    {
        HideDll((ULONG PTR)hModule);
        MessageBoxA (NULL, "DLL Hidden", "Hide the DLL", MB OK);
    ł
    return 1;
}
BOOL HideDll (ULONG PTR DllHandle)
{
ULONG PTR ldr addr;
PEB LDR DATA* ldr data;
LDR MODULE *modulo, *prec, *next;
 try
{
 asm
{
   mov eax, fs:[0x30]
    add eax, 0xc
   mov eax, [eax]
   mov ldr addr, eax
}
ldr data = (PEB LDR DATA*)ldr addr;
```

```
modulo = (LDR MODULE*)ldr data->InLoadOrderModuleList.Flink;
while (modulo->BaseAddress != 0)
{
    if( (ULONG PTR)modulo->BaseAddress == DllHandle)
    ł
        if (modulo->InInitializationOrderModuleList.Blink == NULL)
        {
            return 0;
        }
        prec = (LDR MODULE*) (ULONG PTR) ((ULONG PTR) modulo-
>InInitializationOrderModuleList.Blink - 16);
        next = (LDR MODULE*) (ULONG PTR) ((ULONG PTR) modulo-
>InInitializationOrderModuleList.Flink - 16);
        prec->InInitializationOrderModuleList.Flink = modulo-
>InInitializationOrderModuleList.Flink;
        next->InInitializationOrderModuleList.Blink = modulo-
>InInitializationOrderModuleList.Blink;
        prec = (LDR MODULE*)modulo->InLoadOrderModuleList.Blink;
        next = (LDR MODULE*)modulo->InLoadOrderModuleList.Flink;
        prec->InLoadOrderModuleList.Flink = modulo-
>InLoadOrderModuleList.Flink;
        prec->InMemoryOrderModuleList.Flink = modulo-
>InMemoryOrderModuleList.Flink;
        next->InLoadOrderModuleList.Blink = modulo-
>InLoadOrderModuleList.Blink;
        next->InMemoryOrderModuleList.Blink = modulo-
>InMemoryOrderModuleList.Blink;
       return 1;
    }
   modulo = (LDR MODULE*)modulo->InLoadOrderModuleList.Flink;
}
}
 except (EXCEPTION EXECUTE HANDLER)
{
    return 0;
}
}
```