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.

Purpose

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.

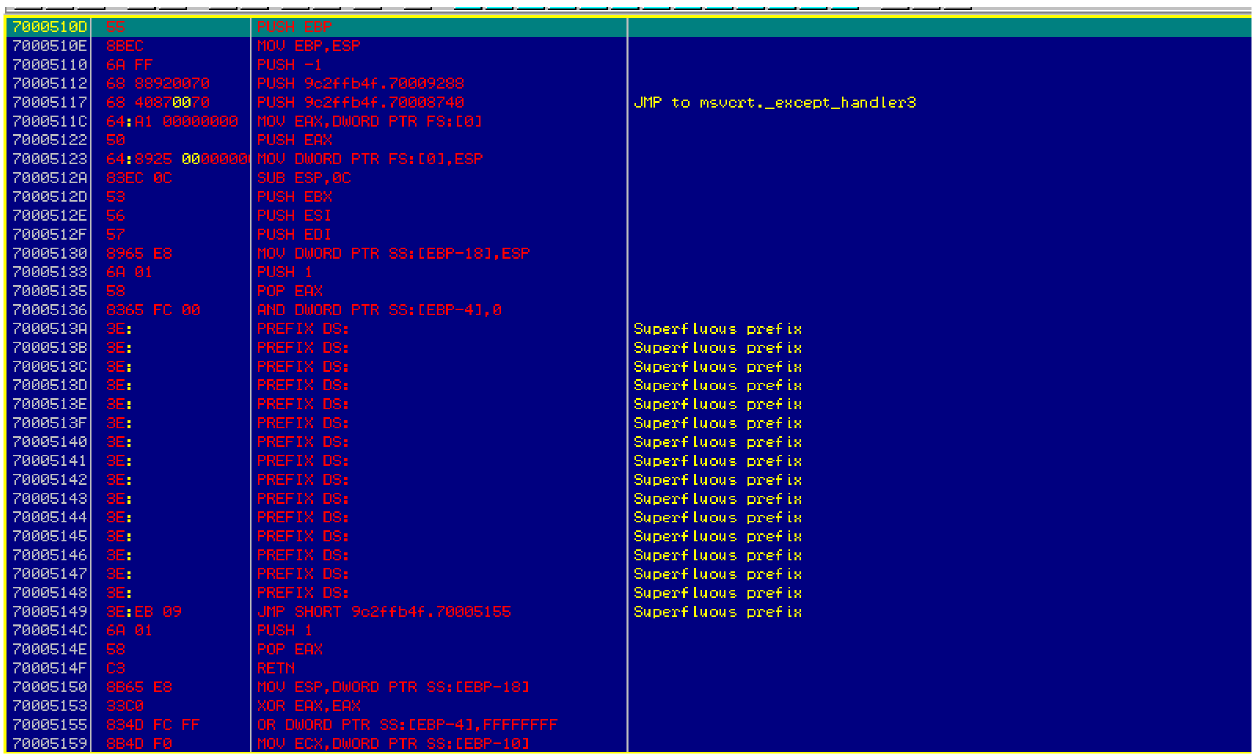
For instance, to detect the presence of VMWare there is a known documented technique which checks for the presence of VMX backdoor used by Hypervisor to communicate with the guest operating system:

```
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.

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))
{
    printf("There was an error in retrieving the module information\n");
    exit(0);
}

while(Module32Next(psnap, &me))
{
    if(strcmp(me.szModule, "kernel32.dll") != 0)
    {
        if(strcmp(me.szModule, "ntdll.dll") != 0)
        {
            hModule = GetModuleHandle(me.szModule);
            if(FreeLibrary(hModule) != 0)
            {
                printf("successfully unloaded injected module, %s\n",
me.szModule);
            }
        }
    }
}

return 0;
}
```

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

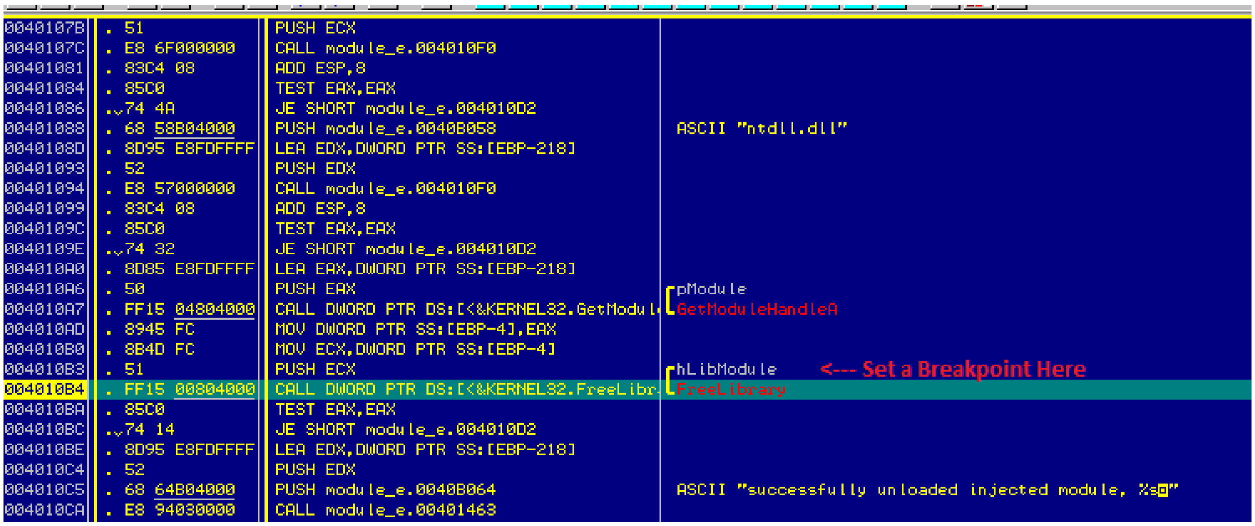
If the above technique is used by a virus to identify the extra loaded module and unload it using FreeLibrary(), we can protect from this using several methods.

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

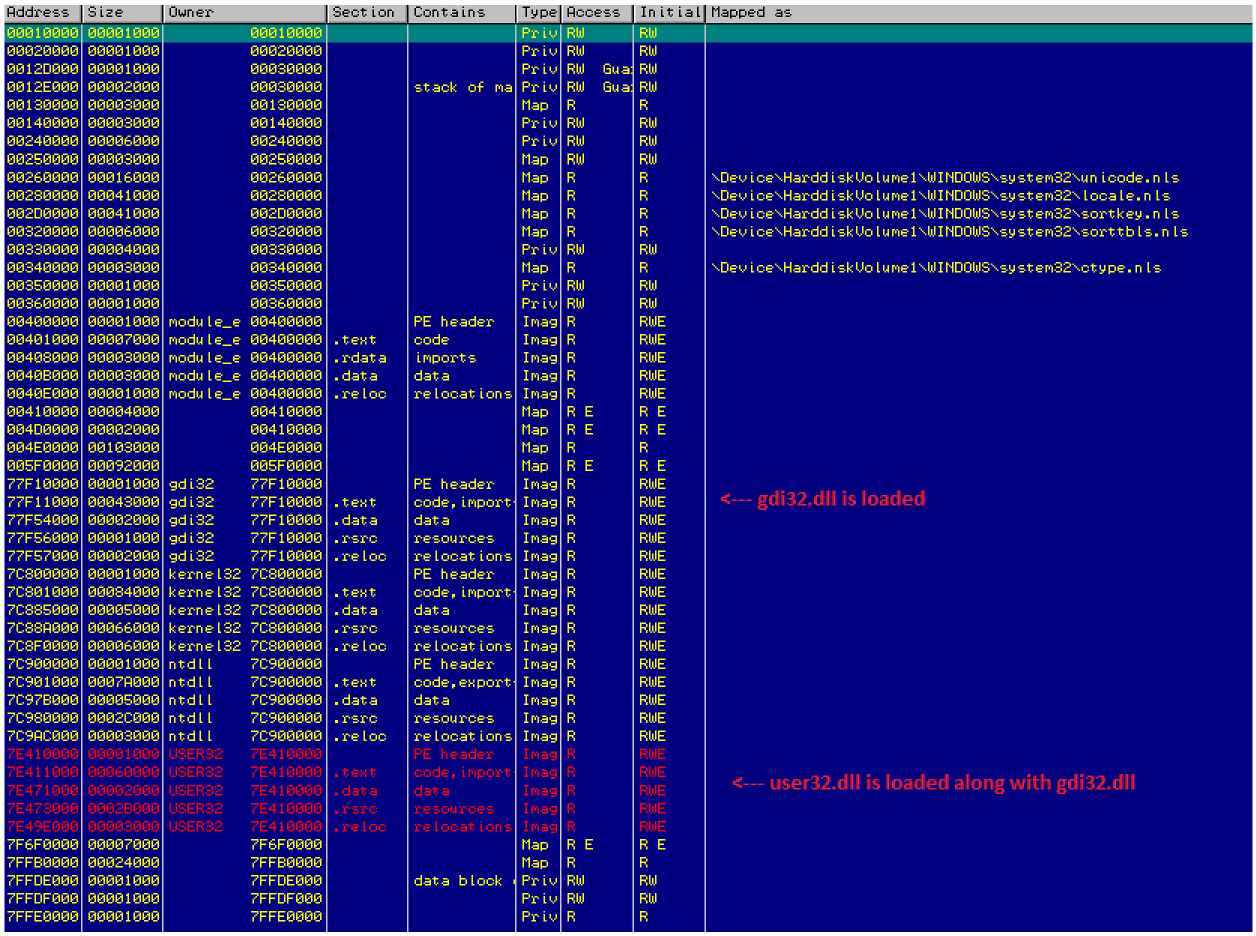
Also, the reference count of a loaded module can be incremented by calling LoadLibrary(). Each time we call LoadLibrary(), it increments the reference count of loaded module and each time we call FreeLibrary(), it decrements the reference count.

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

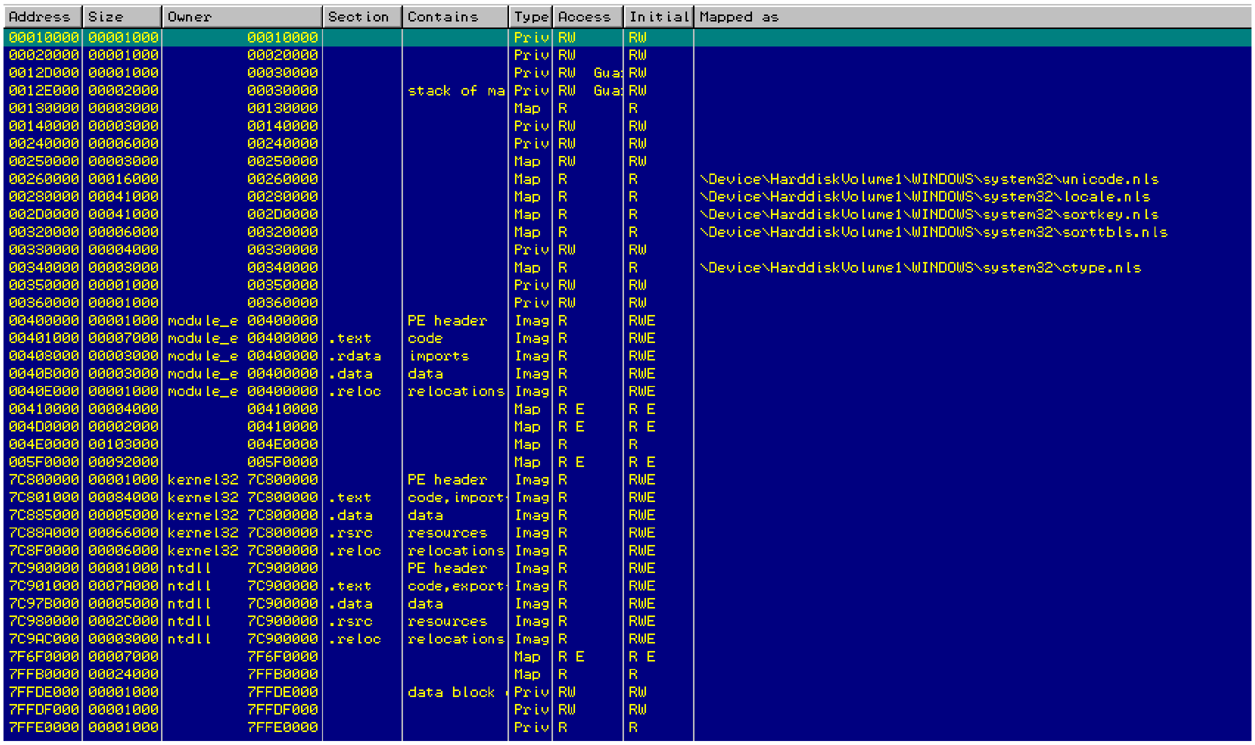


Before executing this call instruction, let us view the loaded modules in Olly Debugger. We can see that both, gdi32.dll and user32.dll are loaded in the process address space.



Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00010000	00001000	00010000			Priv	RW	RW	
00020000	00001000	00020000			Priv	RW	RW	
0012D000	00001000	00030000			Priv	RW Gua:	RW	
0012E000	00002000	00030000		stack of ma	Priv	RW Gua:	RW	
00130000	00003000	00130000			Map	R	R	
00140000	00003000	00140000			Priv	RW	RW	
00240000	00006000	00240000			Priv	RW	RW	
00250000	00003000	00250000			Map	RW	RW	
00260000	00016000	00260000			Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\unicode.nls
00280000	00041000	00280000			Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\locale.nls
002D0000	00041000	002D0000			Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\sortkey.nls
00320000	00006000	00320000			Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\sorttbls.nls
00330000	00004000	00330000			Priv	RW	RW	
00340000	00003000	00340000			Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\ctype.nls
00350000	00001000	00350000			Priv	RW	RW	
00360000	00001000	00360000			Priv	RW	RW	
00400000	00001000	module_e 00400000		PE header	Imag	R	RWE	
00401000	00007000	module_e 00400000	.text	code	Imag	R	RWE	
00408000	00003000	module_e 00400000	.rdata	imports	Imag	R	RWE	
0040B000	00003000	module_e 00400000	.data	data	Imag	R	RWE	
0040E000	00001000	module_e 00400000	.reloc	relocations	Imag	R	RWE	
00410000	00004000	00410000			Map	R E	R E	
004D0000	00002000	00410000			Map	R E	R E	
004E0000	00103000	004E0000			Map	R	R	
005F0000	00092000	005F0000			Map	R E	R E	
77F10000	00001000	gdi32 77F10000		PE header	Imag	R	RWE	
77F11000	00043000	gdi32 77F10000	.text	code,import	Imag	R	RWE	
77F54000	00002000	gdi32 77F10000	.data	data	Imag	R	RWE	
77F56000	00001000	gdi32 77F10000	.rsrc	resources	Imag	R	RWE	
77F57000	00002000	gdi32 77F10000	.reloc	relocations	Imag	R	RWE	
7C800000	00001000	kernel32 7C800000		PE header	Imag	R	RWE	
7C801000	00084000	kernel32 7C800000	.text	code,import	Imag	R	RWE	
7C885000	00005000	kernel32 7C800000	.data	data	Imag	R	RWE	
7C88A000	00066000	kernel32 7C800000	.rsrc	resources	Imag	R	RWE	
7C8F0000	00006000	kernel32 7C800000	.reloc	relocations	Imag	R	RWE	
7C900000	00001000	ntdll 7C900000		PE header	Imag	R	RWE	
7C901000	0007A000	ntdll 7C900000	.text	code,export	Imag	R	RWE	
7C97B000	00005000	ntdll 7C900000	.data	data	Imag	R	RWE	
7C980000	0002C000	ntdll 7C900000	.rsrc	resources	Imag	R	RWE	
7C9AC000	00003000	ntdll 7C900000	.reloc	relocations	Imag	R	RWE	
7E410000	00001000	USER32 7E410000		PE header	Imag	R	RWE	
7E411000	00060000	USER32 7E410000	.text	code,import	Imag	R	RWE	
7E471000	00002000	USER32 7E410000	.data	data	Imag	R	RWE	
7E473000	0002B000	USER32 7E410000	.rsrc	resources	Imag	R	RWE	
7E49E000	00003000	USER32 7E410000	.reloc	relocations	Imag	R	RWE	
7F6F0000	00007000	7F6F0000			Map	R E	R E	
7FFB0000	00024000	7FFB0000			Map	R	R	
7FFDE000	00001000	7FFDE000		data block	Priv	RW	RW	
7FFDF000	00001000	7FFDF000			Priv	RW	RW	
7FFE0000	00001000	7FFE0000			Priv	R	R	

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	Type	Access	Initial	Mapped as
00010000	00001000	00010000			Priv	RW	RW	
00020000	00001000	00020000			Priv	RW	RW	
0012D000	00001000	00030000			Priv	RW Gua:	RW	
0012E000	00002000	00030000		stack of ma	Priv	RW Gua:	RW	
00130000	00003000	00130000			Map	R	R	
00140000	00003000	00140000			Priv	RW	RW	
00240000	00006000	00240000			Priv	RW	RW	
00250000	00003000	00250000			Map	RW	RW	
00260000	00016000	00260000			Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\unicode.nls
00280000	00041000	00280000			Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\locale.nls
002D0000	00041000	002D0000			Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\sortkey.nls
00320000	00006000	00320000			Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\sorttbls.nls
00330000	00004000	00330000			Priv	RW	RW	
00340000	00003000	00340000			Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\ctype.nls
00350000	00001000	00350000			Priv	RW	RW	
00360000	00001000	00360000			Priv	RW	RW	
00400000	00001000	module_e 00400000		PE header	Imag	R	RWE	
00401000	00007000	module_e 00400000	.text	code	Imag	R	RWE	
00408000	00003000	module_e 00400000	.rdata	imports	Imag	R	RWE	
0040B000	00003000	module_e 00400000	.data	data	Imag	R	RWE	
0040E000	00001000	module_e 00400000	.reloc	relocations	Imag	R	RWE	
00410000	00004000	00410000			Map	R E	R E	
004D0000	00002000	00410000			Map	R E	R E	
004E0000	00103000	004E0000			Map	R	R	
005F0000	00092000	005F0000			Map	R E	R E	
7C800000	00001000	kernel32 7C800000		PE header	Imag	R	RWE	
7C801000	00084000	kernel32 7C800000	.text	code,import	Imag	R	RWE	
7C885000	00005000	kernel32 7C800000	.data	data	Imag	R	RWE	
7C88A000	00066000	kernel32 7C800000	.rsrc	resources	Imag	R	RWE	
7C8F0000	00006000	kernel32 7C800000	.reloc	relocations	Imag	R	RWE	
7C900000	00001000	ntdll 7C900000		PE header	Imag	R	RWE	
7C901000	0007A000	ntdll 7C900000	.text	code,export	Imag	R	RWE	
7C97B000	00005000	ntdll 7C900000	.data	data	Imag	R	RWE	
7C980000	0002C000	ntdll 7C900000	.rsrc	resources	Imag	R	RWE	
7C9AC000	00003000	ntdll 7C900000	.reloc	relocations	Imag	R	RWE	
7F6F0000	00007000	7F6F0000			Map	R E	R E	
7FFB0000	00024000	7FFB0000			Map	R	R	
7FFDE000	00001000	7FFDE000		data block	Priv	RW	RW	
7FFDF000	00001000	7FFDF000			Priv	RW	RW	
7FFE0000	00001000	7FFE0000			Priv	R	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++;
}
```

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.

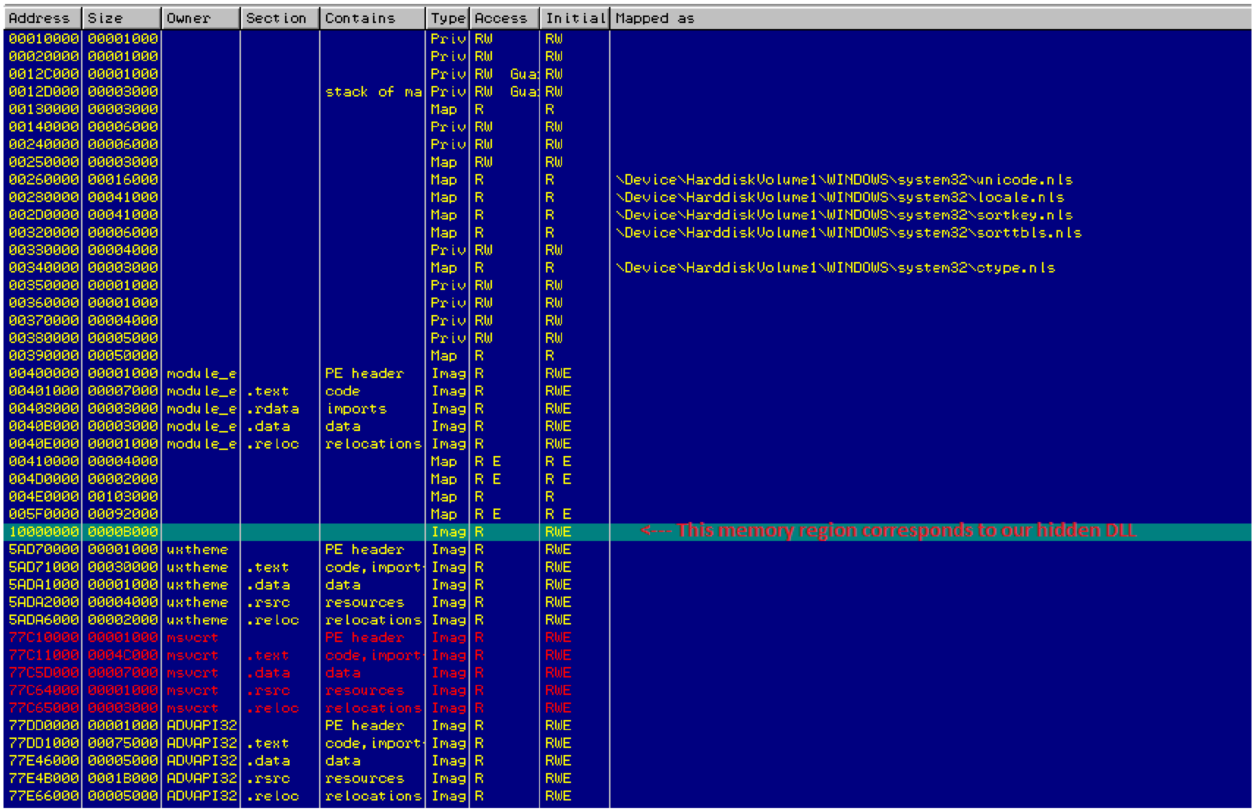
When LoadLibrary() is called, it invokes the DllMain() function of DLL which in turn will call HideDll() function that unlinks the module from PEB.

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	Type	Access	Initial	Mapped as
00010000	00001000				Priv	RW	RW	
00020000	00001000				Priv	RW	RW	
0012C000	00001000				Priv	RW Gua:	RW	
0012D000	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
002D0000	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	
004D0000	00002000				Map	R E	R E	
004E0000	00103000				Map	R	R	
005F0000	00092000				Map	R E	R E	
10000000	0000B000				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	
5ADA1000	00001000	uxtheme	.data	data	Imag	R	RWE	
5ADA2000	00004000	uxtheme	.rsrc	resources	Imag	R	RWE	
5ADA6000	00002000	uxtheme	.reloc	relocations	Imag	R	RWE	
77C10000	00001000	msvcrt		PE header	Imag	R	RWE	
77C11000	0004C000	msvcrt	.text	code,import	Imag	R	RWE	
77C5D000	00007000	msvcrt	.data	data	Imag	R	RWE	
77C64000	00001000	msvcrt	.rsrc	resources	Imag	R	RWE	
77C65000	00003000	msvcrt	.reloc	relocations	Imag	R	RWE	
77DD0000	00001000	ADVAPI32		PE header	Imag	R	RWE	
77DD1000	00075000	ADVAPI32	.text	code,import	Imag	R	RWE	
77E46000	00005000	ADVAPI32	.data	data	Imag	R	RWE	
77E4B000	0001B000	ADVAPI32	.rsrc	resources	Imag	R	RWE	
77E66000	00005000	ADVAPI32	.reloc	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:

1. Short jmp - opcode 0xeb
2. 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
cmp     byte ptr [eax],0E9h
je      dest1
cmp     byte ptr [eax],0EBh
jne     dest2
dest1:
cmp     dword ptr [eax+5],90909090h
je      dest2
mov     edi,edi
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    jmp     qword ptr [kernel32!_imp_Sleep (00000000`7745d7a0)]
00000000`773e28ee 90              nop
00000000`773e28ef 90              nop
00000000`773e28f0 90              nop
00000000`773e28f1 90              nop
00000000`773e28f2 90              nop
00000000`773e28f3 90              nop
kernel32!Wow64DisableWow64FsRedirection:
00000000`773e28f4 90              nop
```

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:

_CIlog
_CIpow
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 Name	Non Standard Prolog	Default Inline Hook by OS	Total Number of Exported 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.
```

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.
```

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]
```

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.

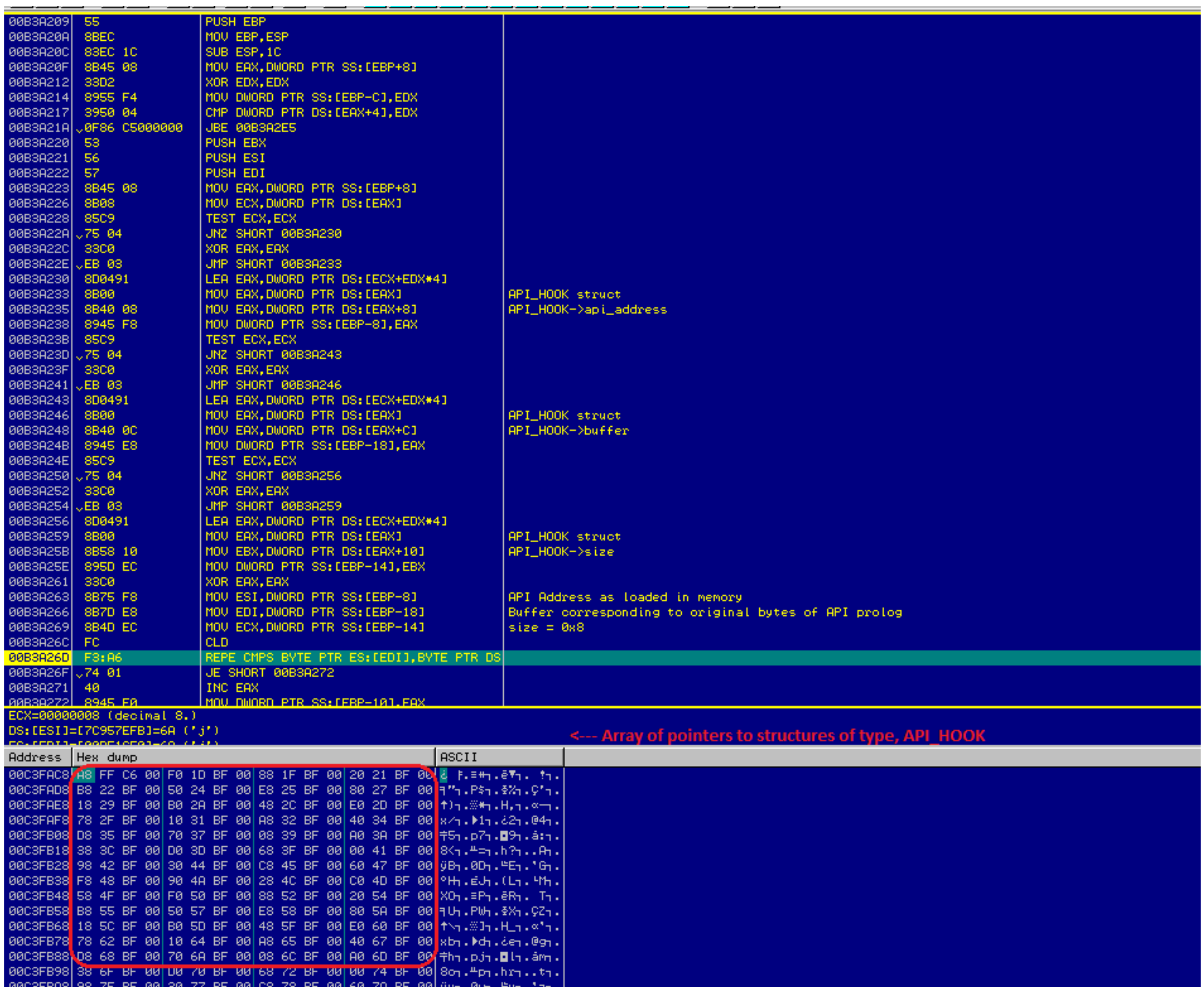
3. After loading the module in memory, it then locates and parses the **Export Data Directory**. Using the **AddressOfOrdinals**, **AddressOfFunctions** and **AddressOfNames** arrays in the Export Directory, it forms a structure for each of the exported API as shown below:

```
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.

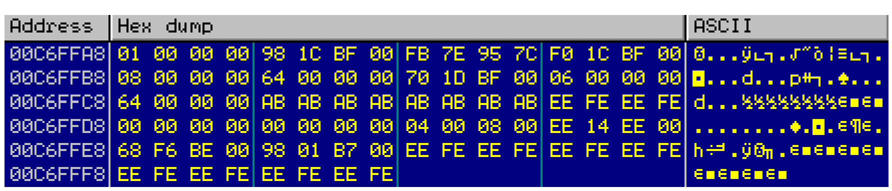
Below screenshot shows the function used for this purpose. The first parameter of this function is a pointer to a pointer to an array of pointers to structures of type **API_HOOK** as mentioned above.



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

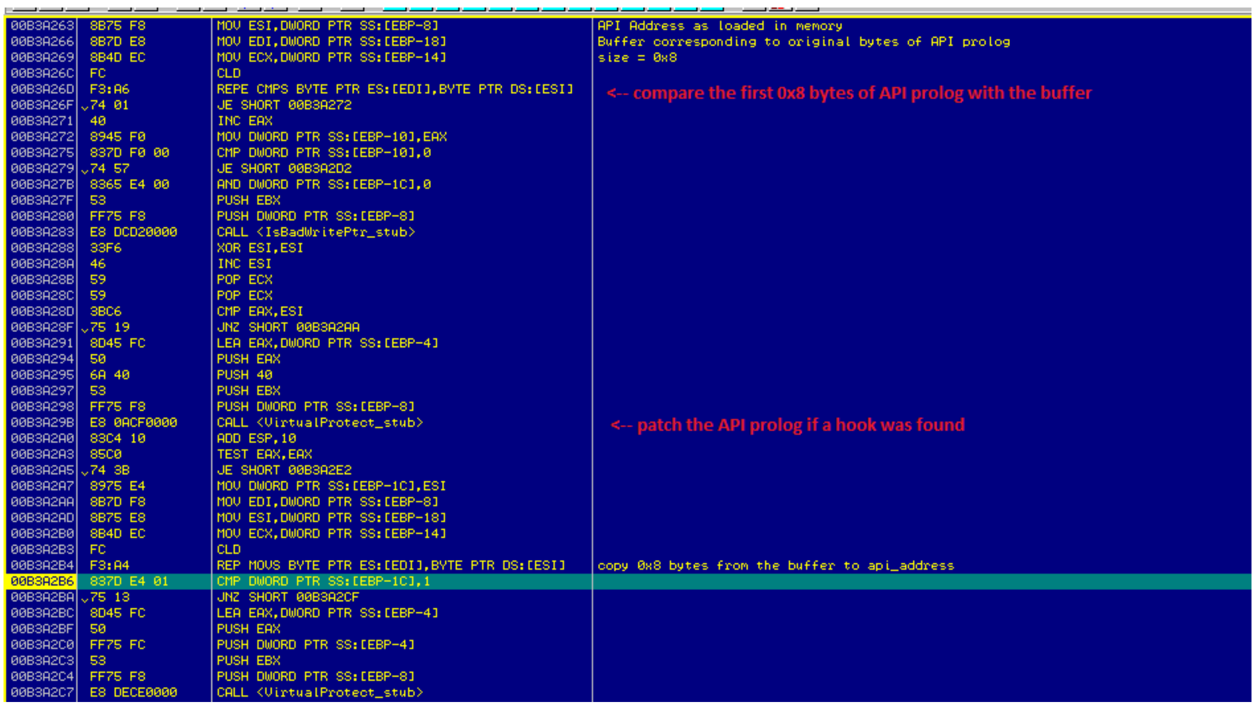
Each of these structures correspond to an API imported from ntdll.dll

Let us check the structure at 0x00C6FFA8



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**.

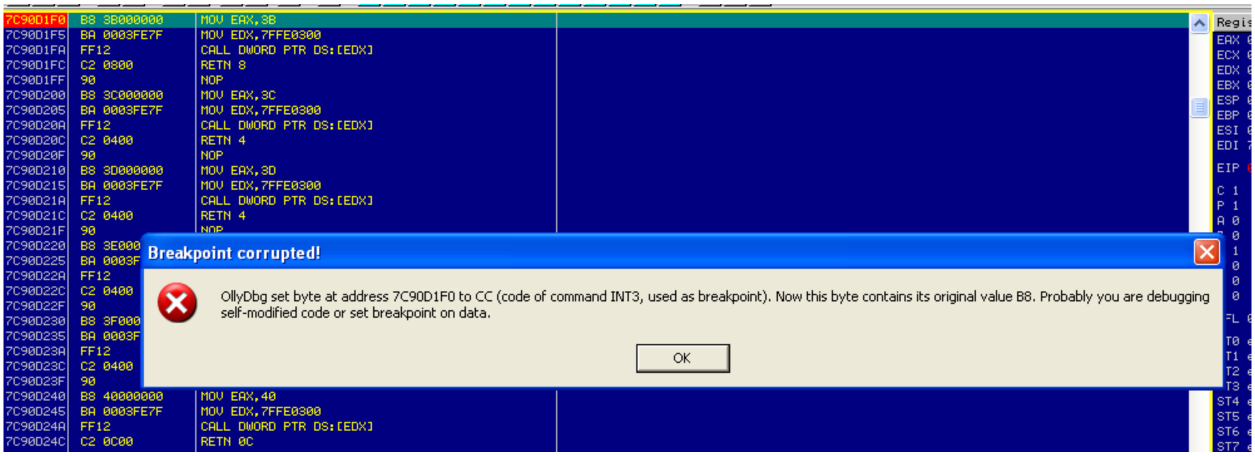


In order to test this algorithm, let us set a breakpoint (INT3) at a native API like ZwDelayExecution.

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.

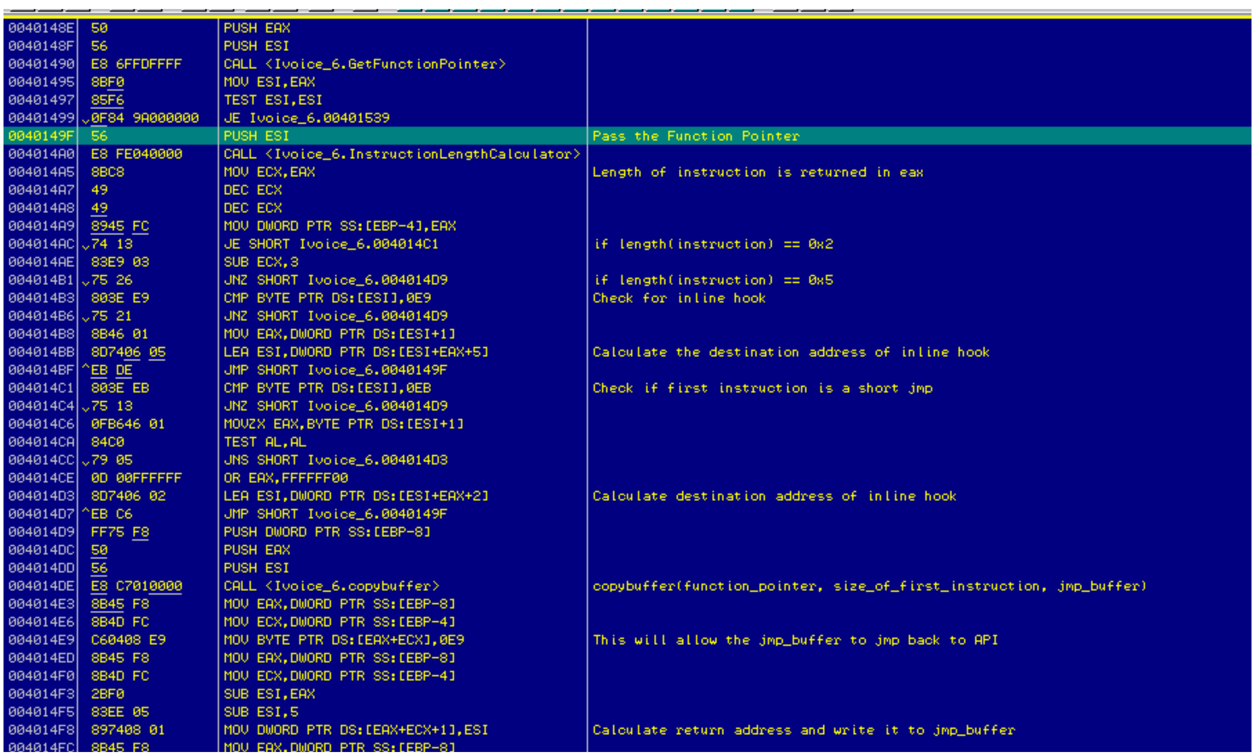
3. After this, it copies the first X bytes of the API prolog to a buffer. Here X refers to the length of the first instruction. It then calculates the address of instruction after X bytes in the API prolog and writes that into the buffer prefixed by a near jmp opcode (0xE9)

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.

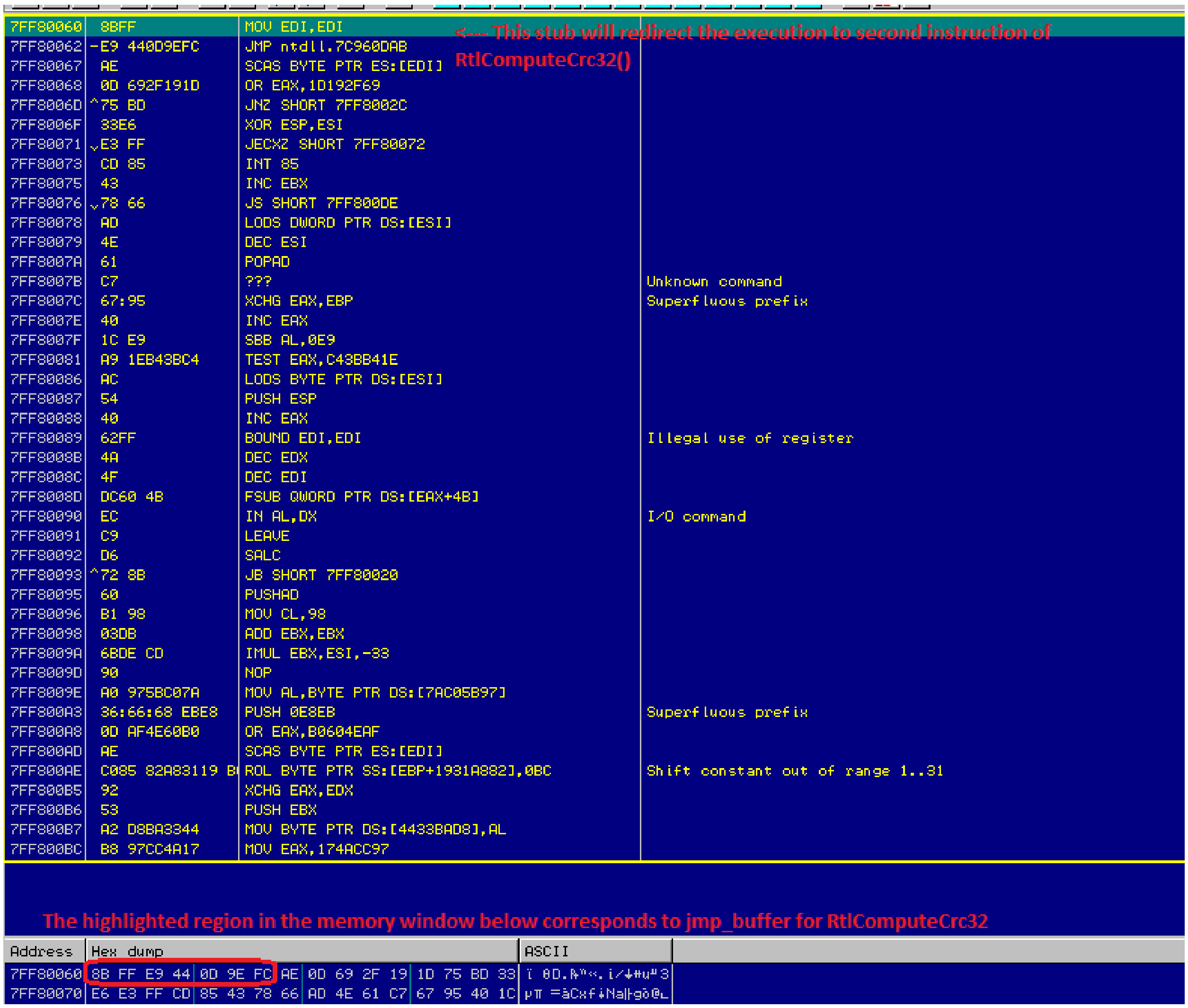
Below screenshot shows the Algorithm along with relevant comments.



```
0040148E   50              PUSH EAX
0040148F   56              PUSH ESI
00401490   E8 6FFDFFFF     CALL <Ivoice_6.GetFunctionPointer>
00401495   8BF0            MOV ESI,EAX
00401497   85F6            TEST ESI,ESI
00401499  ,0F84 9A000000   JE Ivoice_6.00401539
0040149F   56              PUSH ESI                                      Pass the Function Pointer
004014A0   E8 FE040000     CALL <Ivoice_6.InstructionLengthCalculator>
004014A5   8BC8            MOV ECX,EAX                                   Length of instruction is returned in eax
004014A7   49              DEC ECX
004014A8   49              DEC ECX
004014A9   8945 FC         MOV DWORD PTR SS:[EBP-4],EAX
004014AC  ,74 13           JE SHORT Ivoice_6.004014C1                    if length(instruction) == 0x2
004014AE   83E9 03         SUB ECX,3
004014B1  ,75 26           JNZ SHORT Ivoice_6.004014D9                   if length(instruction) == 0x5
004014B3   803E E9         CMP BYTE PTR DS:[ESI],0E9                     Check for inline hook
004014B6  ,75 21           JNZ SHORT Ivoice_6.004014D9
004014B8   8B46 01         MOV EAX,DWORD PTR DS:[ESI+1]
004014BB   8D7406 05       LEA ESI,DWORD PTR DS:[ESI+EAX+5]              Calculate the destination address of inline hook
004014BF  ^EB DE           JMP SHORT Ivoice_6.0040149F
004014C1   803E EB         CMP BYTE PTR DS:[ESI],0EB                     Check if first instruction is a short jmp
004014C4  ,75 13           JNZ SHORT Ivoice_6.004014D9
004014C6   0FB646 01       MOVZX EAX,BYTE PTR DS:[ESI+1]
004014CA   84C0            TEST AL,AL
004014CC  ,79 05           JNS SHORT Ivoice_6.004014D3
004014CE   0D 00FFFFFF     OR EAX,FFFFFF00
004014D3   8D7406 02       LEA ESI,DWORD PTR DS:[ESI+EAX+2]              Calculate destination address of inline hook
004014D7  ^EB C6           JMP SHORT Ivoice_6.0040149F
004014D9   FF75 F8         PUSH DWORD PTR SS:[EBP-8]
004014DC   50              PUSH EAX
004014DD   56              PUSH ESI
004014DE   E8 C7010000     CALL <Ivoice_6.copybuffer>                    copybuffer(function_pointer, size_of_first_instruction, jmp_buffer)
004014E3   8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]
004014E6   8B4D FC         MOV ECX,DWORD PTR SS:[EBP-4]
004014E9   C60408 E9       MOV BYTE PTR DS:[EAX+ECX],0E9                 This will allow the jmp_buffer to jmp back to API
004014ED   8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]
004014F0   8B4D FC         MOV ECX,DWORD PTR SS:[EBP-4]
004014F3   2BF0            SUB ESI,EAX
004014F5   83EE 05         SUB ESI,5
004014F8   897408 01       MOV DWORD PTR DS:[EAX+ECX+1],ESI              Calculate return address and write it to jmp_buffer
004014FC   8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]
```

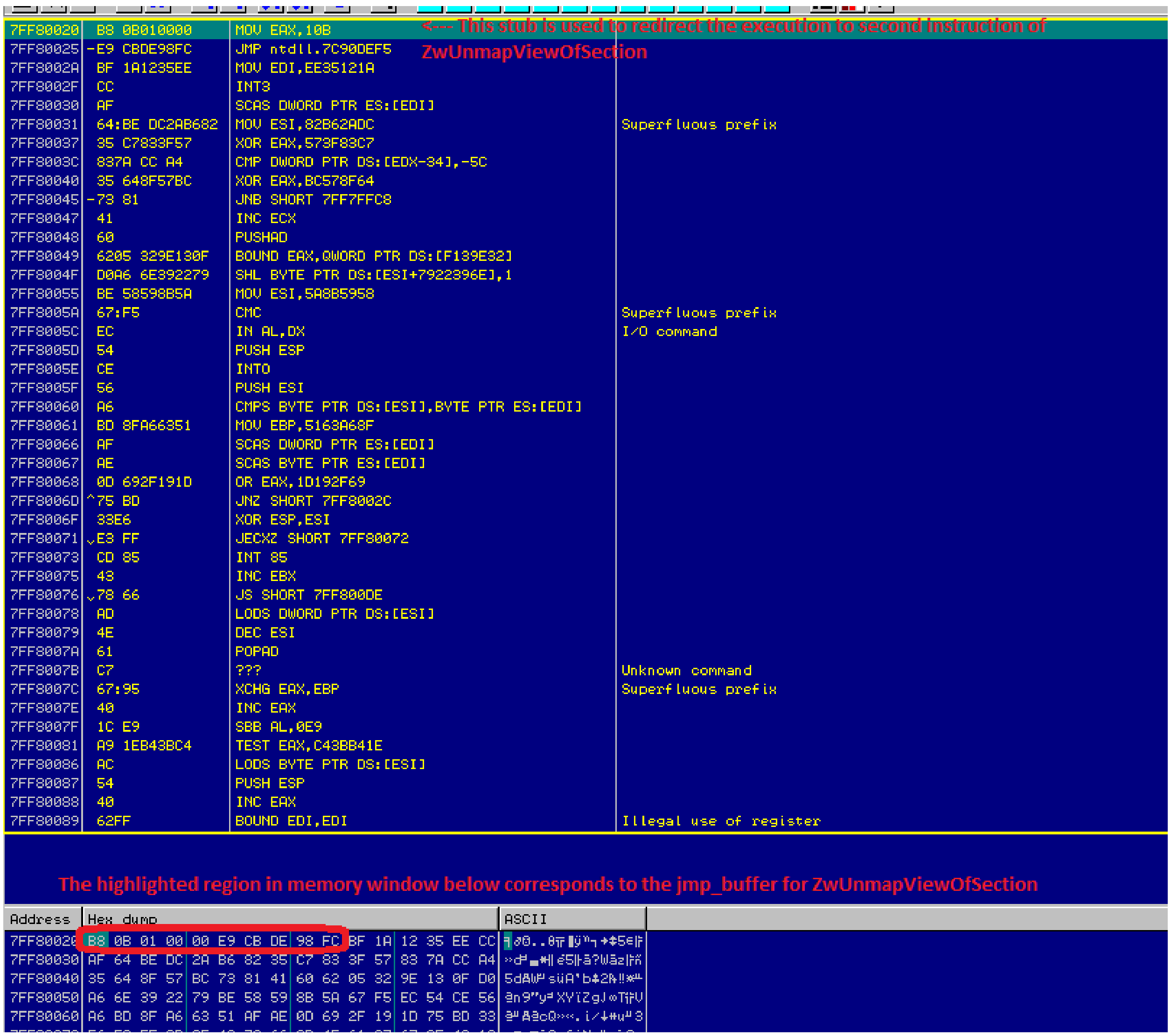
Below is an example of **jmp_buffer** stub for an API with 0x2 bytes as the length of the first instruction:

RtlComputeCrc32:



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

ZwUnmapViewOfSection:



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:

```
7FF90EDC   55              PUSH EBP
7FF90EDD   8BEC            MOV EBP,ESP
7FF90EDF   51              PUSH ECX
7FF90EE0   68 A805F97F     PUSH 7FF905A8                        ASCII "KiFastSystemCall"
7FF90EE5   68 2805F97F     PUSH 7FF90528                        ASCII "ntdll.dll"
7FF90EEA   FF15 DC00F97F   CALL DWORD PTR DS:[7FF900DC]         Call to GetModuleHandleA through stub
7FF90EF0   50              PUSH EAX
7FF90EF1   FF15 7800F97F   CALL DWORD PTR DS:[7FF90078]         Call to GetProcAddress through Stub
7FF90EF7   85C0            TEST EAX,EAX
7FF90EF9  ˅74 48           JE SHORT 7FF90F43
7FF90EFB   8038 EB         CMP BYTE PTR DS:[EAX],0EB            Check for a short jmp at KiFastSystemCall
7FF90EFE  ˅75 43           JNZ SHORT 7FF90F43
7FF90F00   0FB648 01       MOVZX ECX,BYTE PTR DS:[EAX+1]
7FF90F04   894D FC         MOV DWORD PTR SS:[EBP-4],ECX
7FF90F07   84C9            TEST CL,CL
7FF90F09  ˅79 09           JNS SHORT 7FF90F14
7FF90F0B   81C9 00FFFFFF   OR ECX,FFFFFF00
7FF90F11   894D FC         MOV DWORD PTR SS:[EBP-4],ECX
7FF90F14   8D4408 02       LEA EAX,DWORD PTR DS:[EAX+ECX+2]     Follow the short jmp
7FF90F18   8038 68         CMP BYTE PTR DS:[EAX],68             Check for a Push Instruction
7FF90F1B  ˅75 21           JNZ SHORT 7FF90F3E
7FF90F1D   56              PUSH ESI
7FF90F1E   8B70 01         MOV ESI,DWORD PTR DS:[EAX+1]         Read the argument of Push instruction
7FF90F21   8D45 FC         LEA EAX,DWORD PTR SS:[EBP-4]
7FF90F24   50              PUSH EAX
7FF90F25   6A 40           PUSH 40
7FF90F27   6A 05           PUSH 5
7FF90F29   56              PUSH ESI
7FF90F2A   FF15 9C00F97F   CALL DWORD PTR DS:[7FF9009C]         Call to VirtualProtect through Stub
7FF90F30   56              PUSH ESI
7FF90F31   6A 05           PUSH 5
7FF90F33   68 A005F97F     PUSH 7FF905A0
7FF90F38   E8 5D0C0000     CALL <CopyBuffer>                    Copy the original bytes of KiFastSystemCall to buffer
7FF90F3D   5E              POP ESI
7FF90F3E   33C0            XOR EAX,EAX
7FF90F40   40              INC EAX
7FF90F41   C9              LEAVE
7FF90F42   C3              RETN
7FF90F43   33C0            XOR EAX,EAX                          ntdll.KiFastSystemCall
7FF90F45   C9              LEAVE
7FF90F46   C3              RETN
```

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 execution to Betabot's SystemCall
7C90E4FA  C3             RETN                 checking subroutine
7C90E4FB  90             NOP
7C90E4FC  8D6424 00      LEA ESP,DWORD PTR SS:[ESP]
7C90E500  8D5424 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)
    {
        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)
    {
        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, 0xa
```

```
        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", "qemu", "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)
        {
            if(strstr(buffer, vm_names[k]) != NULL)
            {
                printf("Found Device Name: %s\n", buffer);
            }
            k++;
        }

        i++;
    }

    return;
}
```

```c
void drivers_check()
{
    char buffer[256];
    char *basedir="c:\\windows\\system32\\drivers\\";
    char
*driver_names[]={"vmci.sys","vmhgfs.sys","vmmouse.sys","vmscsi.sys","vmusbmou
se.sys","vmx_svga.sys","vmxnet.sys","VBoxMouse.sys"};
    int i=0;

    while(i < 8)
    {
        memset(buffer,'\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.

```c
#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)
    {
        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)
    {
    __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);
    }
    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)
    {
        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;
}
}
```