VMDE
Virtual Machines
Detection Enhanced

N. Rin
EP_XOFF

November 2013
Abstract

This document contains short overview of existing and exploited by WinNT malicious software (malware) methods (AntiVM) that help malware detect execution in the controlled environment such as virtual machine (VM) or/and sandbox. However, this is not complete R&D of each malware with AntiVM, we will focus only on most popular and often seen methods. Unlike other articles, we will show other methods that detect presence of the software VM. We assume readers are familiar with Windows NT based malware, Windows NT core components, x86 architecture and programming languages.

VM detection code criteria:

- User mode execution only;
- No requirement for elevation of privileges;
- Detection must work with all popular virtualization products.

Target software (only popular malware targeted WinNT software):

- VMware Workstation;
- Parallels Desktop;
- Oracle VirtualBox;
- Sandboxie;
- Microsoft Virtual PC (former Connectix Virtual PC), also known as Windows XP Mode in the Windows 7 release.
1. Overview

Virtual machines detection based mostly on execution artifacts with predicted behavior. They can be software-based and hardware-based. Typical example of hardware-based detection is timing attack. This is a hardware specific method based on prediction that code inside virtual machine will execute slower than on real. As example: we calculate time between start and end of code block execution using \texttt{rdtsc} instruction. If the returned value exceeds a predetermined barrier then this used as strong indication of VM presence.

Next malware authors are trying to incorporate hardware-based detections, most of them developed years ago as proof-of-concepts. All methods here based on execution of specific instructions:

- \texttt{slidt}
- \texttt{sgdt}
- \texttt{sidt} \(^1\)
- \texttt{str} \(^2\)
- \texttt{smsw} \(^3\)

After execution, malware performs analysis of the returned data. As for now, virtualization software made a big step forward and these methods simple do not work at all. Malware looks for a specific user names, loaded libraries, registry keys and values, process names, Windows product IDs, filenames. Let us take a brief overview of most notorious malware families with AntiVM. An example of attempt to use hardware-based detection is \textit{Win32/Conficker}, which abuses \texttt{slidt} instruction trick to detect VM. In continuation, we have \textit{Win32/Simda}, which contains special VM detection mechanisms: \textit{Simda} uses blacklist of Product IDs to ban most popular online sandboxing services like Anubis or JoeBox, scans Windows registry for specific installed software and check if the several VM service processes are running. \textit{Win32/Zeus} clone known as \textit{Citadel} brings special signature scanning to detect running on the virtual environment and depending on that fact change it execution flow. It implemented as scanning of image resources for specific string patterns such as “VMware”, “virtualbox” and others. One of \textit{WinNT/Alureon} (family also known as \textit{TDSS/TDLX}) members, a fork of \textit{TDL4} project \textit{MaxSS} implements VM detection by using documented WMI queries\(^4\). It selects information from various WMI providers such as \texttt{Win32_Processor}, \texttt{Win32_SCSIController}, \texttt{Win32_ComputerSystem}, \texttt{Win32_DiskDrive}, \texttt{Win32_BIOS}, and then compares data with it blacklist. Yet another example of well-known malware family with VM detection is \textit{Win32/Sirefef}. Some of it droppers were using specific VM detection code based on documented VM features (querying VMware hypervisor port), and not documented but known VM behavior (Virtual PC way of communication with host). Number of ransomware trojans, as for example \textit{Win32/CBeplay} family is widely using VM detection tricks to complicate analysis. Some of malware may abuse different fact of virtual machines, such as limited virtual disk size,

\footnotesize
\begin{itemize}
  \item \(^1\)http://web.archive.org/web/20070911024318/http://invisiblethings.org/papers/redpill.html
  \item \(^3\)http://www.offensivecomputing.net/dc14/vmdetect.cpp
  \item \(^4\)http://social.technet.microsoft.com/wiki/contents/articles/942.hyper-v-how-to-detect-if-a-computer-is-a-vm-using-script.aspx
\end{itemize}
known Ethernet adapters MAC addresses, hard drive/videocard vendor names, and logical bombs.

To detect execution in the sandbox (e.g. Sandboxie) malware may perform self-checking for API splicing. Another good self-explaining example found inside leaked Win32/Carberp trojan source code, shown on Figure 1-1.

![Fragment of Win32/Carberp code.](image)

**Win32/Caphaw** implements detection by using files and processes scanning. On Figure 1-2 shown Delphi module, code from it widely used in low cost malware such as ransomware Win32/Kovter and various IRC bots (original author comments unmodified).
Win32/Gamarue also known as Andromeda performs several checks to determine execution environment. It employs Sandboxie detection using almost documented way - checking presence of loaded SBIEDLL.DLL⁵ and detects VM by comparing disk ID strings with predefined product blacklist. Example of usual Sandbox detection seen in malware shown on Figure 1-3 (original author comments unmodified).

Win32/Winwebsec is infamous Fake AV that implements detection by using \texttt{cpuid} instruction to get hypervisor vendor name and then compare it with blacklisted VMware vendor string\textsuperscript{6}. Additionally some variants of Winwebsec may use Setup API\textsuperscript{7} to query specific hardware information, code fragment shown Figure 1-4 (original author comments unmodified). Malware detecting usual set of popular VM and QEMU emulator because this software used by several online sandbox services such as Anubis.

\textsuperscript{6} http://kb.VMware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1009458

\textsuperscript{7} http://msdn.microsoft.com/en-us/library/cc185682(v=vs.85).aspx
All the above methods widely used in various malware families and malware obfuscation software (VirTool/Obfuscator). Summarized VM / Sandbox detection methods listed in Table 1-1.

FIGURE 1-4. Win32/Winwebsec VM detection routine.
### TABLE 1-1. Summary of most popular VM / Sandbox detection methods.

<table>
<thead>
<tr>
<th><strong>Method</strong></th>
<th><strong>VM</strong></th>
<th><strong>Sandbox</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU instructions (sidt, sgdt, sldt, str, smsw, cpuid, rdtsc, in)</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>User/Computer names</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Running processes</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Running services</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Loaded modules and memory scanning</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Loaded drivers and their devices</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Registry keys and their values</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Files/Directories</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Named system objects (Devices, Drivers, Mutants, Semaphores, Events, Sections, Ports)</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>MS Hydra GUI subsystem (window names, class names)</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>IOCTL requests</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Hardware information (Disk serial numbers, HDD size, HWID’s, vendor names, Ethernet MAC addresses)</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Windows Product ID’s</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>WMI queries</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Own code modification detection (API hooking detection)</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Logical bombs (user desktop contents, mouse movements)</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

Conclusion: many malware currently has VM detection on board in various implementations of the same methods. This slowdowns analysis and sample may behave completely different inside virtual machine if it detects it.
2. Detection

Now let us take a look how we can implement comprehensive VM detection. The question here “Is it possible at all”? Short answer – no. However, based on our knowledge and set of software we must detect we will try to implement generic methods for VM detection. However, they will not be heuristic, as we cannot rely on software implementation bugs and matching our criteria from Abstract. It is always better abuse things that cannot be simple fixed without product code revision.

Which possible VM detection vectors we decided to use:

- Software environment (2.1);
- Emulated hardware (2.2);
- Additional execution artifacts (2.3).

VM environment usually set up by installing inside VM guess-to-host virtualization support software. It is VMware Tools, VirtualBox Guess Additions, and Virtual PC VM Additions. Each of them contains kernel mode drivers and this fact gives us a good point to start.

**What we decided to throw away:**

A. Process names, driver names, file names, known registry values and names, computer or user names;

B. VM detection proof-of-concepts;

C. GUI based detections;

D. Hardware access related detections (MAC addresses, HDD related info);

E. WMI;

F. Logical bombs.

**Why:**

A. The above methods are unstable and depend on initial VM configuration. If user decided to not use VM additions all these methods will fail;

B. They do not work with modern hardware supported virtualization, do not work at all or too specific;

C. Is too specific and almost the same as A case;

D. Noticeable and unreliable;

E. Significant code increase only to perform basic queries;

F. In most cases, they are unreliable.

We created special tool called VMDE designed to fulfill our criteria and next we will take a closer look on it with examples from VMDE code.
2.1. Software environment

2.1.1. Documented and semi-official detection ways

VMware provides official documented ways of VM detection. First way is based on using hypervisor port 0x5658 ("VX") and hypervisor magic DWORD 0x564D5868 which stands for "VMXh". This method is very old known and widely used by malware. Figure 2-1 shows method implementation inside our VMDE.

The above detection method (as well as used in ScoopyNG\(^8\)) can be simple turned off by VM reconfiguration through %vmname%.vmx file\(^9\) or by additional filtering\(^10\). However, this backdoor usually required for normal work of VMware Tools.

Second documented method described by both VMware and Microsoft. It is testing the CPUID hypervisor (HV) present bit. Quote from official VMware document:

Intel and AMD CPUs have reserved bit 31 of ECX of CPUID leaf 0x1 as the hypervisor present bit. This bit allows hypervisors to indicate their presence to the guest operating system. Hypervisors set this bit and physical CPUs (all existing and future CPUs) set this bit to zero. Guest operating systems can test bit 31 to detect if they are running inside a virtual machine.

Quote from official Microsoft documentation\(^11\):

Before it uses any hypervisor interface functions, software should first determine whether it runs within a virtualized environment. On x64 platforms, software verifies that it runs within a virtualized environment by executing the CPUID

\(^8\) http://www.trapkit.de/research/vmm/scoopyng/

\(^9\) http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf

\(^10\) http://www.securitylab.ru/analytics/427855.php

instruction with an input (EAX register) value of 1. When the CPUID instruction is executed, code should check bit 31 of register ECX. Bit 31 is the hypervisor-present bit. If the hypervisor-present bit is set, the hypervisor is present. In a non-virtualized environment, the hypervisor-present bit is clear.

Example of this detection shown on Figure 2-2.

```c
BOOL Ishypervisor(
    VOID
)
{
    int CPUInfo[4] = {-1};

    /*
    query hypervisor presence
    be aware this detection can be bogus
    */

    __cpuid(CPUInfo, 1);
    if (((CPUInfo[2] >> 31) & 1) { 
        return TRUE;
    }

    return FALSE;
}
```

FIGURE 2-2. HV detection by ECX bit.

If HV presence confirmed then it is good to know which type of HV we have. Quote:

Intel and AMD have also reserved CPUID leaves 0x40000000 - 0x400000FF for software use. Hypervisors can use these leaves to provide an interface to pass information from the hypervisor to the guest operating system running inside a virtual machine. The hypervisor bit indicates the presence of a hypervisor and that it is safe to test these additional software leaves. VMware defines the 0x40000000 leaf as the hypervisor CPUID information leaf. Code running on a VMware hypervisor can test the CPUID information leaf for the hypervisor signature. VMware stores the string "VMwareVMware" in EBX, ECX, EDX of CPUID leaf 0x40000000.

The above works for different HV as well.

```c
BYTE GetHypervisorType(
    VOID
)
{
    int CPUInfo[4] = {-1};

    char HvProductName[0x40];

    __cpuid(CPUInfo, 0x4000000);  
    RtlSecureZeroMemory(HvProductName, sizeof(HvProductName));
    memcpy(HvProductName, CPUInfo + 1, 12);

    if (_strcmpiA(HvProductName, "Microsoft Hv") == 0) {
        return 1;
    }

    if (_strcmpiA(HvProductName, "VMwareVMware") == 0) {
        return 2;
    }

    /* Parallels VM ids */
    if (_strcmpiA(HvProductName, "prl hypervr") == 0) {
        return 3;
    }

    return 0;
}
```

FIGURE 2-3. Detecting type of HV.
The above `cpuid` method used in Win32/Winwebsec malware.

Microsoft Virtual PC formerly known as Connectix Virtual PC can be detected by abusing method it uses to communicate in its guest-to-host model. It is not documented anywhere, but so widely used and so old known, so we can say it is “semi-documented” way\(^\text{12}\). Virtual PC uses a bunch of invalid instructions to allow the interfacing between the virtual machine and the Virtual PC software. The detection code is very popular in ITW malware and malware authors usually blindly copy-past it even without changing set of invalid instructions. Usual set is the following: 0x0F 0x3F 0x07 0x0B. However exists more than 1000 combinations of 0x0F 0x3F 0xXX 0xXX that can be used to detect Virtual PC.

![Virtual PC detection by illegal instruction.](http://www.codeproject.com/Articles/9823/Detect-if-your-program-is-running-inside-a-Virtual)

\(^{12}\) [http://www.codeproject.com/Articles/9823/Detect-if-your-program-is-running-inside-a-Virtual](http://www.codeproject.com/Articles/9823/Detect-if-your-program-is-running-inside-a-Virtual)
2.1.2. Abusing named system objects

All software selected by our criteria has a bunch of named system objects. If user decides not install guess-to-host support VM software (Tools, Additions) the following detection will fail. However, usually these slowdowns VM work as emulated virtual devices work without proper drivers. Named system objects are perfect targets for the VM detection. Why objects are so important? Because Windows operation system object-based. Drivers, devices, communication channels, sections, directories, symbolic links, synchronization mechanisms, jobs, etc, they all represented as objects in kernel mode. The Windows Object Manager controls objects that are part of the kernel-mode operating system. Objects are stored in the corresponding objects directory, for example, KnownDLLs keeps section type objects of the system dll’s and Device directory keeps device type objects created by system or third party drivers. Figure 2-5 shows typical object directory.


![Device directory](http://msdn.microsoft.com/en-us/library/windows/hardware/ff557755(v=vs.85).aspx)
In a particular, we are interested in the following object types:

- Device names
- Mutants (Win32 API alias Mutex)
- Communication Ports

How malware usually use this? Mostly by calling Win32 API services like `CreateFile` or `CreateMutex`. Let us look how we can do the same on native level. As example, we will take detection of Virtual PC Windows XP Mode. Despite the fact that it is almost the same Virtual PC 2007, it has some new features that may be used to detect it exactly as "XP Mode". We only need to check if the mutex

```
MicrosoftVirtualPC7UserServiceMakeSureWe'reTheOnlyOneMutex
```

exists.

![FIGURE 2-6. VMDE mutant checker routine.](image)

The same method used to detect Sandboxie presence. Malware checks for

```
Sandboxie_SingleInstanceMutex_Control
```

mutex presence. This method can be replaced the bellow described technique. Another way to detect Sandboxie is to check presence of `SbieSvcPort` object inside `\RPC Control` objects directory. You can add here any Sandboxie system object that has name and permanently available. While work Sandboxie allocates new directory in objects directory root called `Sandbox`. It keeps virtualized objects for each sandbox created in Sandboxie. They used to deceive sandboxed applications by replacing global system constants. Note:
the system objects allocated first only when something is running in selected sandbox. Figure 2-7 shows example of Sandboxie directory.

```
lkd> !object \Sandbox
Object: 98628650 Type: (839b4e90) Directory
ObjectHeader: 98628638 (new version)
HandleCount: 0 PointerCount: 2
Directory Object: 87405e28 Name: Sandbox

  Hash  Address  Type       Name
   ----  --------  ------   -----
      30  99d65d48  Directory  TestPC

lkd> !object \Sandbox\TestPC
Object: 99d85d48 Type: (839b4e90) Directory
ObjectHeader: 99d85d30 (new version)
HandleCount: 0 PointerCount: 4
Directory Object: 98528650 Name: TestPC

  Hash  Address  Type       Name
   ----  --------  ------   -----
      06  92cd95f8  Directory  12345
      11  87cd7f8  Directory  Sandbox_3
      20  8fa54f58  Directory  DefaultBox
```

FIGURE 2-7. Sandboxie object directory.

To detect presence of Sandboxie is enough simple check if the above-described directory allocated. Note that if Sandboxie was installed but not executed these objects not allocated as well.

```
if ( IsSb == FALSE ) {
    /* not found or error, check sandbox object directory presence */
    RtInitUString(anszName, DIRECTORY_SANDBOX);
    InitializeObjectAttributes(&attr, auszName, OBJ_CASE_INSENSITIVE, NULL, NULL);
    Status = NtOpenDirectoryObject(&hObject, DIRECTORY_QUERY, &attr);
    if (NT_SUCCESS(Status)) {
        IsSb = TRUE;

        #ifdef _DEBUG
        DebugLog(TEXT("Sandbox::ObjectDirectory"));
        #endif

        NtClose(hObject);
    }
}
```

FIGURE 2-8. Detection of Sandboxie object directory.

Malware VM devices hunting usually performed by CreateFile Win32 API calls with usage of symbolic links. Disadvantages of this method are the following: you need to keep VM devices symbolic links database, Win32 API may be easily filtered by upper level user mode sandboxing software.
FIGURE 2-9. Typical VirtualBox detection by VM Guest Additions used by malware.

How can we change this part? We will switch from obtaining a handle of the given device by its link (and thus triggering possible alerts) to checking presence of this device object. In other words, we will query object via scanning objects directory of the given object type. We decided to use simplified enumerator with callback routine, which will perform objects names comparison. Callback procedure is simple and shown on Figure 2-10.

```c
if(CreateFile(TEXT("\\\\.\\\\\BoxMiniRd-DN"), GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL) != INVALID_HANDLE_VALUE) {

    NTSTATUS NTAPI DetectObjectCallback(
    POBJECT_DIRECTORY_INFORMATION Entry,
    PVOID CallbackParam
    )
    {
    POBJSCANPARAM Param;

    if (!ARGUMENT_PRESENT(CallbackParam))
        return STATUS_MEMORY_NOT_ALLOCATED;
    Param = (POBJSCANPARAM)CallbackParam;

    __try {
    if (Param->Buffer == NULL || Param->BufferSize == 0)
        return STATUS_MEMORY_NOT_ALLOCATED;

    if (Entry->Name.Buffer) {
        if (!_strcmpw(Entry->Name.Buffer, Param->Buffer) == 0) {
            return STATUS_SUCCESS;
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        /* nothing */
    }
    return STATUS_UNSUCCESSFUL;
    }

    NTSTATUS NTAPI NtOpenDirectoryObject(
    PVOID DirObjectHandle,
    POBJECT_DIRECTORY_INFORMATION Entry,
    PVOID CallbackParam
    )
    {
    POBJSCANPARAM Param;

    if (!ARGUMENT_PRESENT(CallbackParam))
        return STATUS_MEMORY_NOT_ALLOCATED;
    Param = (POBJSCANPARAM)CallbackParam;

    __try {
    if (Param->Buffer == NULL || Param->BufferSize == 0)
        return STATUS_MEMORY_NOTALLOCATED;

    if (Entry->Name.Buffer) {
        if (!_strcmpw(Entry->Name.Buffer, Param->Buffer) == 0) {
            return STATUS_SUCCESS;
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        /* nothing */
    }
    return STATUS_UNSUCCESSFUL;
    }
```

FIGURE 2-10. VMDE query object callback.

Next let us start with enumerator routine, first acquire object directory handle by calling `NtOpenDirectoryObject`. Notice that as most of Native API of this kind it support supplying not only directory object name, but also a handle to the root directory. Important note: several object directories access requires rights elevation. This applies mostly to “Drivers” directory (as we do not walk all object directories, just a few), but if you have administrator rights this is not a problem. “Device”, “BaseNamedObjects” directory in any case will be accessible, otherwise Windows API will fail itself.
FIGURE 2-11. VMDE opening objects directory.

Once we have object directory handle we need to walk directory by calling `NtQueryDirectoryObject` until it will not return `STATUS_NO_MORE_ENTRIES`. We use simplified memory allocation, usually it is more than enough but you may consider querying exact size of required memory. Figure 2-12 shows rest of `EnumSystemObjects` routine.
Finally the identification routine, IsObjectExists it is very short and simple, see Figure 2-13.

In the Appendix, you will find examples of this routine usage.
2.2. Emulated Hardware

Every virtual machine provides a set of specially emulated hardware. Usually they are video, network adapters, sound cards or virtual controllers. To be able fully use them user need to install set of VM vendor utilities and drivers (VMware Tools, VirtualBox Guest Additions, Virtual PC VM Additions etc). However even nothing installed, emulated hardware is still here and working. We will abuse this fact and target virtual machine hardware. Usually malware detects suspicious hardware by using documented WMI requests, Setup API or by querying hardware specific registry data without deep knowledge what exactly they are querying. If the virtual machine has been heavily customized all the above tricks will fail.

Some of hardware detection methods implemented relying on fact how VM emulates CPU behavior. For example, using cache instructions, like \texttt{wbinvd} and \texttt{invd}\footnote{http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf}. Unfortunately, they are privileged\footnote{http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf}, which means we cannot use them in user mode and that’s break our criteria.

To detect VM presence we will analyze the following data:

- Video BIOS
- System Management BIOS
- PCI bus devices

To get video BIOS and SMB data we will use feature, unavailable in client versions of Windows prior to Vista. Starting from Windows 2003 SP1 as server and Windows Vista RTM as client operation system provides documented\footnote{http://msdn.microsoft.com/en-us/library/windows/desktop/ms724379(v=vs.85).aspx} way to read system components firmware. We will go little deeper and show you it implementation. Because we are going at internal Native API level, nothing from this documented (and presumably never), everything can be subject of change, so you use this as always on your own risk.

![FIGURE 3-1. Structure definitions.](image)

\footref{http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf}
\footref{http://msdn.microsoft.com/en-us/library/windows/desktop/ms724379(v=vs.85).aspx}
However, what to do with legacy Windows versions such as almost dead Windows XP SP3? On Windows XP SP3 the above info class unavailable, but all required information located inside Client/Server Runtime Subsystem (CSRSS) memory space, stored here for Virtual DOS Machine (VDM) purposes, at few fixed addresses. Access to the csrss process requires SeDebugPrivilege enabled for caller process. That is a restriction, but since all Windows XP systems by default running everything with admin rights, it is does not matter in this case. Figure 3-3 shows memory map of csrss with selected memory region that hold Video BIOS data.
Memory region with address 0xC0000 holds raw firmware data, and memory region with address 0xE0000 holds raw SMBIOS data. Dump it from there as shown on Figure 3-4. After dumping contents of these tables all is left run signature scanning, locating VM specific data. Note that Microsoft Virtual PC video card is emulated S3 Trio64 adapter with complete S3 Trio64 VGA BIOS. Ironically, only fact of old S3 Graphics based adapter is a good indication of virtual machine.
FIGURE 3-4. The rest of VMDE GetFirmwareTable.

Next few figures demonstrate contents of firmware data used by popular virtual machines.
FIGURE 3-5. Part of Virtual PC firmware dump, S3 VGA BIOS.

FIGURE 3-6. Part of Virtual PC RSMB dump.
FIGURE 3-7. Part of VMware firmware dump.

FIGURE 3-8. Part of VMware RSMB dump.
FIGURE 3-9. Part of Virtual Box firmware dump.

FIGURE 3-10. Part of Virtual Box RSMB dump.
This information is not depending on VM tools installation but as you might guess, heavy manually customized VM can bypass that check, but not all VM may be reconfigured that way without execution problem. Another way to detect VM presence is scanning PCI bus devices. Figure 3-13 shows PCI bus connected devices on the Virtual PC.
FIGURE 3-13. PCI bus connected devices.

Usually malware does not take this information in credit and only scans registry for specific vendor names like *VBOX*, *VMWARE* and so on. We will detect VM presence in a different way by enumerating PCI bus devices and querying their Hardware vendor ID (HWID). The problem is again that we must do this from user mode and if possible, without elevating privileges, so we cannot use \SetcbPrivilege and \ProcessUserModeIOPL to list PCI devices manually. Instead, we will use system information stored inside

\REGISTRY\MACHINE\SYSTEM\CurrentControlSet\Enum\PCI

We even do not need to read anything from it, it is enough to simple enumerate the root key. Each key here represent a PCI bus connected device and has the following device instance format:

VEN_XXXX&DEV_XXXX&SUBSYS_XXXXXXXXX&REV_XX

Where VEN stands for Vendor ID in hexadecimal view, DEV stands for Device ID in hexadecimal view. While dumping content of the root key we will process each entry and extract VID value from it. See Appendix for a examples of usage.
FIGURE 3-14. VMDE HWID enumeration routine.

Once we know all PCI bus devices we can walk through ID table and compare with known VM vendors ID. Table 3-1 contains list of known vendors.

TABLE 3-1. Virtual Machine known hardware vendor ID.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Vendor ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMware</td>
<td>0x15AD</td>
</tr>
<tr>
<td>Oracle</td>
<td>0x80EE</td>
</tr>
<tr>
<td>Parallels</td>
<td>0x1AB8</td>
</tr>
</tbody>
</table>

In case of Virtual PC you can query S3 VID which is 0x5333, but take care as this can trigger false positives.
2.3. Additional execution artifacts.

While running certain virtualization software always produces artifacts that cannot occur outside VM/Sandbox environment. They can be different: mismatches between hardware information returned by CPU and operation system (number of cores, suspicious CPU vendor names), incorrectly emulated instructions, traces of virtualization software code execution. The perfect example of these artifacts, we can say artifacts generator, is Sandboxie. Some of it detection methods already mentioned in previous chapters and in-the-wild malware uses only few of them. What Sandboxie actually does:

- Isolates selected application from the rest of the system by heavy kernel mode modifications to process affiliated system structures;
- Provides transparent layer of API call redirects and filtering, controlling application execution behavior.

How secure this compared to execution on the VM? The both sandbox and virtual machine might have many in common goals but very different in implementation and result. Therefore, we would say they could not be compared because code runs at different conditions. However, instead of pure VM Sandboxie will always suffer from critical implementation bugs\(^{18}\) that may alter behavior of real system and/or even compromise it security. We did not say that VM is free from implementation bugs\(^{19}\) \(^{20}\), but their usage and effect is incomparable to bugs in the sandbox software that shares same host resources and session. In other words, sandboxing software is mostly for casual users, while virtual machines are more advanced and more user skill dependent. Here we must distinguish sandboxing software oriented for end-users and sandboxing as implementation used for example in antivirus products and laboratories.

We will abuse Sandboxie transparent API layer to detect if our application is being sandboxed. There are multiple detection ways, but we will show only most obvious. One of the ways is that how Sandboxie implements program isolation through virtualized registry. Big role here plays helper library SBIEDLL.DLL that injected in every sandboxed process. This dll itself is a great indication of Sandboxie and this fact widely used by malware. Nevertheless, exist helper applications that hide presence of this dll\(^{21}\). Virtualized registry only present in sandboxed applications so all we need is to establish fact of such virtualization. Each sandbox has its own dedicated fake registry entries (just like in case of system named objects). Figure 3-15 shows example.


\(^{19}\) [https://www.virtualbox.org/ticket/10947](https://www.virtualbox.org/ticket/10947)


\(^{21}\) [http://bsa.isoftware.nl/](http://bsa.isoftware.nl/)
To make this virtualization transparent to the sandboxed application SBIEDDLL.DLL component perform massive API filtering in the user mode, faking results of several APIs. Below is list of massive code modifications found in the sandboxed process running under Windows XP SP3. Note that under different platforms this list may look different. These standard libraries modifications itself by the way can be used to detect sandboxing.

To figure out the exact changes, one can use tools like Process Monitor or similar. The listed changes are inline code modifications that are injected into the sandboxed application.

Here is a list of some of the changes found in the sandboxed process running under Windows XP SP3:

- **ntdll.dll**
  - `LdrInitializeThunk` Inline - Relative Jump 0x7C901166--->7E420008 [unknown_code_page]
  - `LdrLoadDll` Inline - Relative Jump 0x7C9163A3--->7D2538E0 [SbieDll.dll]
  - `LdrQueryImageFileExecutionOptions` Inline - Relative Jump 0x7C91CC83--->7D2539E0 [SbieDll.dll]
  - `LdrUnloadDll` Inline - Relative Jump 0x7C91736B--->7D253940 [SbieDll.dll]
  - `NtAcceptConnectPort` Code Mismatch 0x7C90CE40 + 4 [18 E9 16 32 B1 01]
  - `NtAccessCheck` Code Mismatch 0x7C90CE50 + 4 [20 E9 06 32 B1 01]
  - `NtAccessCheckAndAuditAlarm` Code Mismatch 0x7C90CE60 + 4 [2C E9 F6 31 B1 01]
  - `NtAccessCheckByType` Code Mismatch 0x7C90CE70 + 4 [2C E9 E6 31 B1 01]
  - `NtAccessCheckAndAuditAlarmInline` - Relative Jump 0x7C90CE84--->7E420060 [unknown_code_page]
  - `NtAccessCheckByTypeResultList` Code Mismatch 0x7C90CE90 + 4 [2C E9 C6 31 B1 01]
  - `NtAccessCheckByTypeResultListAndAuditAlarm Inline` - Relative Jump 0x7C90CEA4--->7E420060 [unknown_code_page]
  - `NtAccessCheckByTypeResultListAndAuditAlarmByHandle` Inline - Relative Jump 0x7C90CEB4--->7E420060 [unknown_code_page]
  - `NtAddAtom` Code Mismatch 0x7C90CEC0 + 4 [0C E9 96 31 B1 01]
  - `NtAddBootEntry` Code Mismatch 0x7C90CED0 + 4 [08 E9 86 31 B1 01]
  - `NtAdjustGroupsToken` Inline - Relative Jump 0x7C90CF15--->7E420060 [SbieDll.dll]
  - `NtAllocateLocallyUniqueId` Code Mismatch 0x7C90CF20 + 4 [04 E9 36 31 B1 01]
  - `NtAllocateUserPhysicalPages` Code Mismatch 0x7C90CF30 + 4 [0C E9 26 31 B1 01]
  - `NtAllocateUuids` Code Mismatch 0x7C90CF40 + 4 [10 E9 16 31 B1 01]
  - `NtAllocateVirtualMemory` Code Mismatch 0x7C90CF50 + 4 [18 E9 06 31 B1 01]
  - `NtAreMappedFilesTheSame` Code Mismatch 0x7C90CF60 + 4 [08 E9 F6 30 B1 01]
  - `NtAssignProcessToJobObject` Inline - Relative Jump 0x7C90CF70--->7D260930 [SbieDll.dll]
  - `NtAssignProcessToJobObject` Inline - Relative Jump 0x7C90CF75--->7E420060 [unknown_code_page]
  - `NtAlertResumeThread` Code Mismatch 0x7C90CF80 + 4 [08 E9 56 31 B1 01]
  - `NtAlertThread` Code Mismatch 0x7C90CF90 + 4 [04 E9 46 31 B1 01]
  - `NtAllocateLocallyUniqueId` Code Mismatch 0x7C90CFB0 + 4 [04 E9 36 31 B1 01]
  - `NtAllocateUserPhysicalPages` Code Mismatch 0x7C90CFD0 + 4 [08 E9 A6 30 B1 01]
  - `NtAllocateUuids` Code Mismatch 0x7C90CFE0 + 4 [08 E9 86 31 B1 01]
  - `NtAllocateVirtualMemory` Code Mismatch 0x7C90CFF0 + 4 [10 E9 16 31 B1 01]
  - `NtAreMappedFilesTheSame` Code Mismatch 0x7C90CFF4 + 4 [18 E9 06 31 B1 01]
  - `NtAssignProcessToJobObject` Inline - Relative Jump 0x7C90CFF70--->7D260930 [SbieDll.dll]
  - `NtAssignProcessToJobObject` Inline - Relative Jump 0x7C90CFF5--->7E420060 [unknown_code_page]
  - `NtCancelDeviceWakeupRequest` Code Mismatch 0x7C90CF90 + 4 [04 E9 C6 30 B1 01]
  - `NtCancelIoFile` Code Mismatch 0x7C90CFB0 + 4 [08 E9 B6 30 B1 01]
  - `NtCancelTimer` Code Mismatch 0x7C90CFB0 + 4 [08 E9 A6 30 B1 01]
  - `NtClearEvent` Code Mismatch 0x7C90CFE0 + 4 [04 E9 96 30 B1 01]
  - `NtClose` Inline - Relative Jump 0x7C90CFFD--->7D233C70 [SbieDll.dll]
  - `NtClose` Inline - Relative Jump 0x7C90CFD5--->7E420060 [unknown_code_page]
  - `NtCloseObjectAuditAlarm` Inline - Relative Jump 0x7C90CFF5--->7C90D013C [SbieDll.dll]
  - `NtCompactKeys` Code Mismatch 0x7C90CFF0 + 4 [08 E9 66 30 B1 01]
  - `NtCompareTokens` Code Mismatch 0x7C90CFD0 + 4 [0C E9 56 30 B1 01]
ntdll.dll-->NtQueryDebugFilterState Code Mismatch 0x7C90D720 + 4 [08 E9 36 29 B1 01]
ntdll.dll-->NtQueryDefaultLocale Code Mismatch 0x7C90D730 + 4 [08 E9 26 29 B1 01]
ntdll.dll-->NtQueryDefaultUILanguage Code Mismatch 0x7C90D740 + 4 [04 E9 16 29 B1 01]
ntdll.dll-->NtQueryDirectoryFile Inline - RelativeJump 0x7C90D750-->7D23A990 [SbieDll.dll]
ntdll.dll-->NtQueryDirectoryFile Inline - RelativeJump 0x7C90D770-->7E420060 [unknown_code_page]
ntdll.dll-->NtQueryDirectoryObject Code Mismatch 0x7C90D760 + 4 [1C E9 F6 28 B1 01]
ntdll.dll-->NtQuery EaFile Code Mismatch 0x7C90D770 + 4 [24 E9 E6 28 B1 01]
ntdll.dll-->NtQueryEvent Code Mismatch 0x7C90D780 + 4 [14 E9 D6 28 B1 01]
ntdll.dll-->NtQueryFullAttributesFile Inline - RelativeJump 0x7C90D790-->7D23BF0C [SbieDll.dll]
ntdll.dll-->NtQueryFullAttributesFile Inline - RelativeJump 0x7C90D795-->7E420060 [unknown_code_page]
ntdll.dll-->NtQueryInformationAtom Code Mismatch 0x7C90D7A0 + 4 [14 E9 B6 28 B1 01]
ntdll.dll-->NtQueryInformationFile Inline - RelativeJump 0x7C90D7B0-->7D23E450 [SbieDll.dll]
ntdll.dll-->NtQueryInformationFile Inline - RelativeJump 0x7C90D7B5-->7E420060 [unknown_code_page]
ntdll.dll-->NtQueryInformationJobObject Code Mismatch 0x7C90D7C0 + 4 [14 E9 96 28 B1 01]
ntdll.dll-->NtQueryInformationPort Code Mismatch 0x7C90D7D0 + 4 [14 E9 86 28 B1 01]
ntdll.dll-->NtQueryInformationProcess Inline - RelativeJump 0x7C90D7E4-->7C90D814 [ntdll.dll]
ntdll.dll-->NtQueryInformationThread Code Mismatch 0x7C90D7F0 + 4 [14 E9 66 28 B1 01]
ntdll.dll-->NtQueryInformationToken Mismatch 0x7C90D800 + 4 [14 E9 56 28 B1 01]
ntdll.dll-->NtQueryInstallUILanguage Code Mismatch 0x7C90D810 + 4 [04 E9 46 28 B1 01]
ntdll.dll-->NtQueryIntervalProfile Code Mismatch 0x7C90D820 + 4 [08 E9 36 28 B1 01]
ntdll.dll-->NtQueryIoCompletion Code Mismatch 0x7C90D830 + 4 [14 E9 26 28 B1 01]
ntdll.dll-->NtQueryKey Inline - RelativeJump 0x7C90D840-->7D252C70 [SbieDll.dll]
ntdll.dll-->NtQueryKey Inline - RelativeJump 0x7C90D845-->7E420060 [unknown_code_page]
ntdll.dll-->NtQueryMultipleValueKey Inline - RelativeJump 0x7C90D850-->7D252620 [SbieDll.dll]
ntdll.dll-->NtQueryMultipleValueKey Inline - RelativeJump 0x7C90D855-->7E420060 [unknown_code_page]
ntdll.dll-->NtQueryObject Inline - RelativeJump 0x7C90D860 + 4 [14 E9 F6 27 B1 01]
ntdll.dll-->NtQueryObject Inline - RelativeJump 0x7C90D870-->7D254F00 [SbieDll.dll]
ntdll.dll-->NtQueryObject Inline - RelativeJump 0x7C90D875-->7E420060 [unknown_code_page]
ntdll.dll-->NtQueryOpenSubKeys Code Mismatch 0x7C90D880 + 4 [08 E9 D6 27 B1 01]
ntdll.dll-->NtQueryPerformanceCounter Code Mismatch 0x7C90D890 + 4 [08 E9 C5 27 B1 01]
ntdll.dll-->NtQueryPortInformationProcess Inline - RelativeJump 0x7C90D9F5-->7E420060 [unknown_code_page]
ntdll.dll-->NtQueryQuotaInformationFile Code Mismatch 0x7C90D9A0 + 4 [24 E9 B6 27 B1 01]
ntdll.dll-->NtQuerySection Code Mismatch 0x7C90D9B0 + 4 [14 E9 A6 27 B1 01]
ntdll.dll-->NtQuerySecurityObject Inline - RelativeJump 0x7C90D9C0-->7D25E080 [SbieDll.dll]
ntdll.dll-->NtQuerySecurityObject Inline - RelativeJump 0x7C90D9C5-->7E420060 [unknown_code_page]
ntdll.dll-->NtQuerySemaphore Code Mismatch 0x7C90D9D0 + 4 [14 E9 56 28 B1 01]
ntdll.dll-->NtQuerySymbolicLinkObject Inline - RelativeJump 0x7C90D9E0-->7C90D913 [ntdll.dll]
ntdll.dll-->NtQuerySystemEnvironmentValue Code Mismatch 0x7C90D9F0 + 4 [10 E9 66 27 B1 01]
ntdll.dll-->NtQuerySystemEnvironmentValueEx Code Mismatch 0x7C90D9F0 + 4 [14 E9 56 27 B1 01]
ntdll.dll-->NtQuerySystemInformation Inline - RelativeJump 0x7C90D980-->7D23DF07 [SbieDll.dll]
ntdll.dll-->NtQuerySystemInformation Inline - RelativeJump 0x7C90D995-->7E420060 [unknown_code_page]
ntdll.dll-->NtQuerySystemTime Code Mismatch 0x7C90D920 + 4 [04 E9 36 27 B1 01]
ntdll.dll-->NtQueryTimer Code Mismatch 0x7C90D930 + 4 [14 E9 26 27 B1 01]
ntdll.dll-->NtQueryTimerResolution Code Mismatch 0x7C90D940 + 4 [04 E9 16 27 B1 01]
ntdll.dll-->NtQueryValueKey Inline - RelativeJump 0x7C90D950-->7D252210 [SbieDll.dll]
ntdll.dll-->NtQueryValueKey Inline - RelativeJump 0x7C90D955-->7E420060 [unknown_code_page]
ntdll.dll-->NtQueryVirtualMemory Inline - RelativeJump 0x7C90D960-->7D2550F0 [SbieDll.dll]
ntdll.dll-->NtQueryVirtualMemory Inline - RelativeJump 0x7C90D965-->7E420060 [unknown_code_page]
ntdll.dll-->NtQueryVolumeInformationFile Inline - RelativeJump 0x7C90D970-->7D23CE50 [SbieDll.dll]
ntdll.dll-->NtQueryVolumeInformationFile Inline - RelativeJump 0x7C90D975-->7E420060 [unknown_code_page]
ntdll.dll-->NtQueueApcThread Code Mismatch 0x7C90D990 + 4 [08 E9 36 29 B1 01]
ntdll.dll-->NtRaiseHardError Code Mismatch 0x7C90D9A0 + 4 [18 E9 56 28 B1 01]
ntdll.dll-->NtReadFile Inline - RelativeJump 0x7C90D9B0-->7D233290 [SbieDll.dll]
ntdll.dll-->NtReadFile Inline - RelativeJump 0x7C90D9B5-->7E420060 [unknown_code_page]
ntdll.dll-->NtReadFileScatter Code Mismatch 0x7C90D9C0 + 4 [24 E9 96 26 B1 01]
ntdll.dll-->NtReadRequestData Code Mismatch 0x7C90D9D0 + 4 [18 E9 96 26 B1 01]
ntdll.dll-->NtReadVirtualMemory Inline - RelativeJump 0x7C90D9E0-->7D23D912 [ntdll.dll]
ntdll.dll-->NtRegisterThreadTerminatePort Code Mismatch 0x7C90D9F0 + 4 [04 E9 66 26 B1 01]
ntdll.dll-->NtReleaseApcEvent Code Mismatch 0x7C90DF00 + 4 [10 E9 86 20 B1 01]
ntdll.dll-->NtReleaseApcEvent Code Mismatch 0x7C90DF00 + 4 [24 E9 96 26 B1 01]
ntdll.dll-->NtRemoveProcessDebug Code Mismatch 0x7C90D9A0 + 4 [08 E9 56 26 B1 01]
ntdll.dll-->NtRenameKey Inline - RelativeJump 0x7C90D9A0-->7D25D340 [SbieDll.dll]
ntdll.dll-->NtRenameKey Inline - RelativeJump 0x7C90D9A5-->7E420060 [unknown_code_page]
ntdll.dll-->NtReplaceKey Code Mismatch 0x7C90D9A0 + 4 [0C E9 06 26 B1 01]
ntdll.dll-->NtReplaceObject Mismatch 0x7C90D9B0 + 4 [08 E9 F6 25 B1 01]
ntdll.dll-->NtReplyWaitReceivePort Code Mismatch 0x7C90D9A0 + 4 [10 E9 E6 25 B1 01]
ntdll.dll-->NtReplyWaitReceivePort Code Mismatch 0x7C90D9A0 + 4 [14 E9 D6 25 B1 01]
Due to lazy implementation, SBIEDLL does multiple improper modifications to the API call results, which however does not affect their work (not always, about a year ago author fixed one bug related to improper data modification – improper truncating result of some API call), but can be used as Sandboxie detection flags. For example, SBIEDLL truncates result of registry querying to hide from sandboxed application registry.
virtualization fact. Ironically, exactly this will be used to detect virtualization, as shown on Figure 3-16.

The following code used to detect if caller application is sandboxed, but only after caller make sure Sandboxie is present, so this code must work in complex with other detection methods to eliminate possible false positives. See Appendix for example of usage. Next, we will abuse helper-to-the-driver communication mechanism and use it to detect if the caller process is being sandboxed.

Another method based on how SBIEDLL communicates with Sandboxie driver that performs kernel mode modifications (and until recent versions driver was incompatible with Kernel Patch Protection due to obvious reasons). SBIEDLL does this through specially created device object. Internally SBIEDLL opens handle to it in every sandboxed process and uses it during NtDeviceIoControlFile calls, passing various IOCTL codes to the main driver. This device name is constant value:

**SandboxieDriverApi**

Following device name can be used to detect Sandboxie presence on a target machine (see part 2.1.2 of this document), because this device is permanent. We will enumerate caller process handle table and verify objects to belong to Sandboxie. To improve performance the verifying objects will be filtered by file object type. See Figure 3-17 for implementation details.
Another way is to perform process memory scanning and searching for Sandboxie injected executable memory. How can we distinguish it from other kind of caller executable code? By tag Sandboxie uses for allocated memory. Yes, it is author nickname.

![Command - Local kernel - WinDbg:6.11.0001.404 AN](image)

**FIGURE 3-18.** Sandboxie memory tag.
Now scan process memory, filter regions by allocation protect and inspect suspicious memory areas for known signature.

All the above detection methods must work only if Sandboxie presence confirmed by other detection methods and must work together to back up each other.

As a short conclusion: Sandboxie does so many modifications to the sandboxed process so detecting fact of sandboxing is a trivial task even if the above described methods will be out-dated.
2.4. Conclusion.

Malware more often use various VM detection tricks, targeting popular virtualization software. Currently these detection methods are primitive and blindly copy-pasted over various malware families. The virtualization software needs additional configuration to able to handle such cases. Unfortunately, none of the observed products has this feature out of the box. From all the listed products only VirtualBox and VMware has ability to additional reconfiguration for bypassing VM checks. As for Sandboxie, with its current implementation it cannot be hidden at all and not recommended as platform for malware research. More to say we would not recommend using this product on real PC on the regular basis as well, and if you still plan use it for malware research then do this only inside additionally controlled environment such as, you might already guess, virtual machine.
3. Appendix

3.1. Virtual PC detection

```c
BOOL IsVirtualPC(
    VOID
)
{
    BYTE IsVM = 0;
    ULONG dwDataSize = 0L;
    PSYSTEM_FIRMWARE_TABLE_INFORMATION pSIF = NULL;
    /* devs of XP Mode we're so kind so they added special mutex, check it */
    IsVM = IsMutexExist(MUTEX_VPCXPMODE);
    #ifdef _DEBUG
    if (IsVM == 1) DebugLog(TEXT("VPC::XPMode"));
    #endif
    /*
       use well-known trick with illegal instructions, but be creative and don't use the
       same as here, there are numbers of them actually (> 1000), not only one set
    */
    if (IsVM == 0) {
        __try {
            __asm push   ebx
            __asm mov    ebx, 0
            __asm mov    eax, 1
            __asm __emit 0Fh
            __asm __emit 3Fh
            __asm __emit 0Dh
            __asm __emit 0h
            __asm test   ebx, ebx
            __asm setz   [IsVM]
            __asm pop    ebx
        }
        __except(VPCExceptionHandler(GetExceptionInformation())) {  }
    }
    #ifdef _DEBUG
    if (IsVM == 1) DebugLog(TEXT("VPC::Backdoor"));
    #endif
    /* query virtual pc device */
    if (IsVM == 0) {
        IsVM = (IsObjectExists(DEVICELINK, DEVICE_VIRTUALPC));
    }
    #ifdef _DEBUG
    if (IsVM == 1) DebugLog(TEXT("VPC::VMSvc"));
    #endif
    /* query virtual pc driver, reg. rights elevation */
    if (IsVM == 0) {
        IsVM = (IsObjectExists(DRIVERLINK, DRIVER_VIRTUALPC));
    }
    #ifdef _DEBUG
    if (IsVM == 1) DebugLog(TEXT("VPC::VmDriver"));
    #endif
    /* scan raw firmware for specific string patterns */
    if (IsVM == 0) {
        pSIF = (PSYSTEM_FIRMWARE_TABLE_INFORMATION)GetFirmwareTable(&dwDataSize, FIRM,
        0xC0000);
        if (pSIF != NULL && dwDataSize > 0) {
            IsVM = ScanDump((CHAR*)pSIF, dwDataSize, VENDOR_VPC, _strlenA(VENDOR_VPC));
        }
    }
    #ifdef _DEBUG
    if (IsVM == 1) DebugLog(TEXT("VPC::FirmwareS3MS"));
    #endif
}
```
RtlFreeHeap(RtlProcessHeap(), 0, pSIF);
}

/* scan raw sbios data for specific string patterns */
if (IsVM == 0) {
    pSIF = (PSYSTEM_FIRMWARE_TABLE_INFORMATION)GetFirmwareTable(&dwDataSize, RSMB, 0);
    if (pSIF != NULL && dwDataSize > 0) {
        IsVM = ScanDump((CHAR*)pSIF, dwDataSize, SMB_VPC, strlenA(SMB_VPC));
    }
}

#ifdef _DEBUG
    if (IsVM == 1) DebugLog(TEXT("VPC::VM"));
#endif

RtlFreeHeap(RtlProcessHeap(), 0, pSIF);
}

/* query S3 VID on PCI bus devices */
if (IsVM == 0) {
    if (vIsInList(VID_S3MS) != NULL) IsVM = 1;
#endif
    if (IsVM == 1) DebugLog(TEXT("VPC::VID_S3"));
#endif

return IsVM;
)
3.2. VMware detection

BOOL IsVMware(
    VOID
)
{
    BYTE IsVM = 0;
    ULONG dwDataSize = 0L;
    PSYSTEM_FIRMWARE_TABLE_INFORMATION pSIF = NULL;
    /* query VMware additions device presence */
    IsVM = IsObjectExists(DEVICELINK, DEVICE_VMWARE);
    #ifdef _DEBUG
    if (IsVM == 1) DebugLog(TEXT("VMware::Memctrl"));
    #endif

    if (IsVM == 0) {
        /* query VMware presence by hypervisor port */
        */
        __try {
            __asm push   edx
            __asm push   ecx
            __asm push   ebx
            __asm mov    eax, 'VMXh'
            __asm mov    ebx, 0
            __asm mov    ecx, 10
            __asm mov    edx, 'VX'
            __asm in     eax, dx
            __asm cmp    ebx, 'VMXh'
            __asm setz   [IsVM]
            __asm pop    ebx
            __asm pop    ecx
            __asm pop    edx
        }
        __except(EXCEPTION_EXECUTE_HANDLER) {
            IsVM = 0;
        }
    }
    #ifdef _DEBUG
    if (IsVM == 1) DebugLog(TEXT("VMware::HVPort"));
    #endif

    /* scan raw firmware for specific string patterns */
    if (IsVM == 0) {
        pSIF = (PSYSTEM_FIRMWARE_TABLE_INFORMATION)GetFirmwareTable(&dwDataSize, FIRM, 0xC0000);
        if (pSIF != NULL && dwDataSize > 0) {
            IsVM = ScanDump((CHAR*)pSIF, dwDataSize, VENDOR_VMWARE, _strlenA(VENDOR_VMWARE));
        }
    }
    #ifdef _DEBUG
    if (IsVM == 1) DebugLog(TEXT("VMware::Firmware_VMware"));
    #endif

    RtlFreeHeap(RtlProcessHeap(), 0, pSIF);

    /* scan raw SMBIOS firmware table for specific string patterns */
    if (IsVM == 0) {
        pSIF = (PSYSTEM_FIRMWARE_TABLE_INFORMATION)GetFirmwareTable(&dwDataSize, RSMB, 0);
        if (pSIF != NULL && dwDataSize > 0) {
            IsVM = ScanDump((CHAR*)pSIF, dwDataSize, VENDOR_VMWARE, _strlenA(VENDOR_VMWARE));
        }
    }
    #ifdef _DEBUG
    if (IsVM == 1) DebugLog(TEXT("VMware::SMB_VMware"));
    #endif

    IsVM = ScanDump((CHAR*)pSIF, dwDataSize, SMB_VMWARE, _strlenA(SMB_VMWARE));
    #ifdef _DEBUG

if (IsVM == 1) DebugLog(TEXT("VMware::SMBIOS_VMware"));
}
#ifdefined _DEBUG
if (IsVM == 1) DebugLog(TEXT("VMware::VID_VMware"));
#endif
#else
RtlFreeHeap(RtlProcessHeap(), 0, pSIF);
#endif
/* query VMware VID on PCI bus devices */
if ( IsVM == 0 ) {
    if ( vIsInList(VID_VMWARE) != NULL ) IsVM = 1;
}
#if defined _DEBUG
    if (IsVM == 1) DebugLog(TEXT("VMware::VID_VMware"));
#endif
return IsVM;
3.3. VirtualBox detection

BOOL IsVirtualBox(
    VOID
) {
    BYTE IsVM = 0;
    ULONG dwDataSize = 0L;
    PSYSTEM_FIRMWARE_TABLE_INFORMATION pSIF = NULL;

    /* query vbox additions guest device */
    IsVM = (IsObjectExists(DEVICELINK, DEVICE_VIRTUALBOX1));
    #ifdef _DEBUG
    if (IsVM == 1) DebugLog(TEXT("VBox::VBoxGuest"));
    #endif

    /* query vbox additions device */
    if (IsVM == 0) {
        IsVM = (IsObjectExists(DEVICELINK, DEVICE_VIRTUALBOX2));
        #ifdef _DEBUG
        if (IsVM == 1) DebugLog(TEXT("VBox::VBoxMiniRdrDN"));
        #endif
    }

    /* query vbox additions video driver, reg. rights elevation*/
    if (IsVM == 0) {
        IsVM = (IsObjectExists(DRIVERLINK, DRIVER_VIRTUALBOX1));
        #ifdef _DEBUG
        if (IsVM == 1) DebugLog(TEXT("VBox::VBoxVideo"));
        #endif
    }

    /* query vbox additions mouse driver, reg. admin rights elevation */
    if (IsVM == 0) {
        IsVM = (IsObjectExists(DRIVERLINK, DRIVER_VIRTUALBOX2));
        #ifdef _DEBUG
        if (IsVM == 1) DebugLog(TEXT("VBox::VBoxMouse"));
        #endif
    }

    /* scan raw firmware for specific string patterns */
    if (IsVM == 0) {
        pSIF = (PSYSTEM_FIRMWARE_TABLE_INFORMATION)GetFirmwareTable(&dwDataSize, FIRM, 0xC0000);
        if (pSIF != NULL && dwDataSize > 0) {
            IsVM = ScanDump((CHAR*)pSIF, dwDataSize, VENDOR_VBOX, _strlenA(VENDOR_VBOX));
            #ifdef _DEBUG
            if (IsVM == 1) DebugLog(TEXT("VBox::FirmwareVBox"));
            #endif
        }
        if (IsVM == 0) {
            IsVM = ScanDump((CHAR*)pSIF, dwDataSize, VENDOR_ORACLE, _strlenA(VENDOR_ORACLE));
            #ifdef _DEBUG
            if (IsVM == 1) DebugLog(TEXT("VBox::FirmwareOracle"));
            #endif
        }
    }
    #ifdef _DEBUG
    if (IsVM == 0) {
        IsVM = ScanDump((CHAR*)pSIF, dwDataSize, VENDOR_INNOTEK, _strlenA(VENDOR_INNOTEK));
        #ifdef _DEBUG
        if (IsVM == 1) DebugLog(TEXT("VBox::FirmwareInnotek"));
        #endif
    }
    #endif
}
RtlFreeHeap(RtlProcessHeap(), 0, pSIF);

/* scan raw SMBIOS firmware table for specific string patterns */
if (IsVM == 0) {
    pSIF = (PSYSTEM_FIRMWARE_TABLE_INFORMATION)GetFirmwareTable(&dwDataSize, RSMB, 0);
    if (pSIF != NULL & dwDataSize > 0) {
        IsVM = ScanDump((CHAR*)pSIF, dwDataSize, VENDOR_VBOX,
            _strlenA(VENDOR_VBOX));
#ifdef _DEBUG
        if (IsVM == 1) DebugLog(TEXT("VBox::SMBIOS_VBox"));
#endif
        if (IsVM == 0) {
            IsVM = ScanDump((CHAR*)pSIF, dwDataSize, VENDOR_ORACLE,
                _strlenA(VENDOR_ORACLE));
#ifdef _DEBUG
        if (IsVM == 1) DebugLog(TEXT("VBox::SMBIOS_Oracle"));
#endif
        }
        if (IsVM == 0) {
            IsVM = ScanDump((CHAR*)pSIF, dwDataSize, VENDOR_INNOTEK,
                _strlenA(VENDOR_INNOTEK));
#ifdef _DEBUG
        if (IsVM == 1) DebugLog(TEXT("VBox::SMBIOS_Innotek"));
#endif
        }
        RtIFreeHeap(RtlProcessHeap(), 0, pSIF);
    }
}

/* query oracle VID on PCI bus devices */
if (IsVM == 0) {
    if (vIsInList(VID_ORACLE) != NULL) IsVM = 1;
#ifdef _DEBUG
        if (IsVM == 1) DebugLog(TEXT("VBox::VID_Oracle"));
#endif
}
return IsVM;
3.4. Parallels detection

BOOL IsParallels(
    VOID
)
{
    BYTE IsVM = 0;
    ULONG dwDataSize = 0L;
    PSYSTEM_FIRMWARE_TABLE_INFORMATION pSIF = NULL;
    /* query parallels additions device presence */
    IsVM = IsObjectExists(DEVICELINK, DEVICE_PARALLELS1);

    #ifdef _DEBUG
    if (IsVM == 1) DebugLog(TEXT("Parallels::prl_pv"));
    #endif

    if (IsVM == 0)
    {
        IsVM = IsObjectExists(DEVICELINK, DEVICE_PARALLELS2);

        #ifdef _DEBUG
        if (IsVM == 1) DebugLog(TEXT("Parallels::prl_tg"));
        #endif

        if (IsVM == 0)
        {
            IsVM = IsObjectExists(DEVICELINK, DEVICE_PARALLELS3);

            #ifdef _DEBUG
            if (IsVM == 1) DebugLog(TEXT("Parallels::prl_time"));
            #endif
        }
    }

    /* scan raw firmware for specific string patterns */
    if (IsVM == 0)
    {
        if (pSIF != NULL && dwDataSize > 0)
        {
            IsVM = ScanDump((CHAR*)pSIF, dwDataSize, VENDOR_PARALLELS,
                _strlenA(VENDOR_PARALLELS));

            #ifdef _DEBUG
            if (IsVM == 1) DebugLog(TEXT("Parallels::Firmware_Parallels"));
            #endif
        }
    }
}

RtlFreeHeap(RtlProcessHeap(), 0, pSIF);

/* scan raw SMBIOS firmware table for specific string patterns */
if (IsVM == 0)
{
    pSIF = (PSYSTEM_FIRMWARE_TABLE_INFORMATION)GetFirmwareTable(&dwDataSize, FIRM,
        0xCD0000);
    if (pSIF != NULL && dwDataSize > 0) { IsVM = ScanDump((CHAR*)pSIF, dwDataSize, SMB_PARALLELS, _strlenA(SMB_PARALLELS));
        #ifdef _DEBUG
        if (IsVM == 1) DebugLog(TEXT("Parallels::SMBIOS_Parallels"));
        #endif
    }
}

RtlFreeHeap(RtlProcessHeap(), 0, pSIF);

/* query Parallels on PCI bus devices */
if (IsVM == 0)
{
    if (vIsInList(VID_PRLS) != NULL) IsVM = 1;

    #ifdef _DEBUG
    if (IsVM == 1) DebugLog(TEXT("VPC::VID_PRLS"));
    #endif
}
return IsVM;
3.5. Sandboxie detection

BOOL IsSandboxiePresent(
    VOID
)
{
    BYTE IsSB = 0;
    OBJECT_ATTRIBUTES attr;
    UNICODE_STRING ustrName;
    NTSTATUS Status;
    HANDLE hObject = NULL;
    WCHAR szSandboxieMutex[MAX_PATH] = {0};
    /* first check sandboxie device */
    IsSB = (IsObjectExists(DEVICELINK, DEVICE_SANDBOXIE));
#ifdef _DEBUG
    if (IsSB == 1) DebugLog(TEXT("Sandboxie::ApiDevice"));
#endif
    if (IsSB == 0)
    {
        /* not found or error, check sandbox object directory presence */
        RtlInitUnicodeString(&ustrName, DIRECTORY_SANDBOXIE);
        InitializeObjectAttributes(&attr, &ustrName, OBJ_CASE_INSENSITIVE, NULL, NULL);
        Status = NtOpenDirectoryObject(&hObject, DIRECTORY_QUERY, &attr);
        if (NT_SUCCESS(Status))
        {
            IsSB = 1;
#ifdef _DEBUG
            DebugLog(TEXT("Sandboxie::ObjectDirectory"));
#endif
            NtClose(hObject);
        }
    }
    /* query sandboxie mutex */
    if (IsSB == 0) { IsSB = IsMutexExist(MUTEX_SANDBOXIE); }
#ifdef _DEBUG
    if (IsSB == 1) DebugLog(TEXT("Sandboxie::Mutex"));
#endif
    /* query sandboxie rpc port presence*/
    if (IsSB == 0) { IsSB = (IsObjectExists(RPCCONTROLLINK, PORT_SANDBOXIE)); }
#ifdef _DEBUG
    if (IsSB == 1) DebugLog(TEXT("Sandboxie::Port"));
#endif
    /* query driver object, reg. rights elevation */
    if (IsSB == 0) { IsSB = (IsObjectExists(DRIVERLINK, DRIVER_SANDBOXIE)); }
#ifdef _DEBUG
    if (IsSB == 1) DebugLog(TEXT("Sandboxie::Driver"));
#endif
    return IsSB;
}
3.6. Sandboxie virtualization detection

```c
BOOL IsSandboxieVirtualRegistryPresent(
    VOID
)
{
    BYTE IsSB = 0;
    ULONG hashA, hashB;
    HANDLE hKey;
    NTSTATUS Status;
    UNICODE_STRING ustrRegPath;
    OBJECT_ATTRIBUTES obja;

    WCHAR szObjectName[MAX_PATH * 2] = {0};
    hashA = HashFromStrW(REGSTR_KEY_USER);
    RtlInitUnicodeString(&ustrRegPath, REGSTR_KEY_USER);
    InitializeObjectAttributes(&obja, &ustrRegPath, OBJ_CASE_INSENSITIVE, NULL, NULL);
    Status = NtOpenKey(&hKey, MAXIMUM_ALLOWED, &obja);
    if (NT_SUCCESS(Status)) {
        if (QueryObjectName((HKEY)hKey, &szObjectName, MAX_PATH * 2, TRUE)) {
            hashB = HashFromStrW(szObjectName);
            if (hashB != hashA) IsSB = 1;
        }
        NtClose(hKey);
    }
    return IsSB;
}

BOOL AmISandboxed(
    VOID
)
{
    BYTE IsSB = 0;
    ULONG uLength = 0L;
    NTSTATUS Status;
    HANDLE hDummy;
    ULONG_PTR k, i, FileID = 0xFFFFFFFF, OurID = GetCurrentProcessId();
    SYSTEM_OBJECTTYPE_INFORMATION TypeInfo = {0};
    MEMORY_BASIC_INFORMATION RegionInfo = {0};
    WCHAR szObjectName[MAX_PATH * 2] = {0};

    __try {
        /* find sandboxie api device inside our handle table */
        hDummy = OpenDevice(L"\Device\Null", GENERIC_READ, &Status);
        if (hDummy == NULL) __leave;
        HandleTable = (PSYSTEM_HANDLE_INFORMATION)NativeAllocateInfoBuffer(SystemHandleInformation, &uLength);
        if (HandleTable == NULL) __leave;

        for (k=0; k<2; k++) {
            for (i=0; i<HandleTable->NumberOfHandles; i++) {
                if (HandleTable->Handles[i].UniqueProcessId == OurID ) {
                    if (k == 0) {
                        if (HandleTable->Handles[i].HandleValue == (USHORT)hDummy ) {
                            FileID = HandleTable->Handles[i].ObjectTypeIndex;
                            break;
                        }
                    } else {
                        if (HandleTable->Handles[i].ObjectTypeIndex == FileID ) {
                            if (QueryObjectName((HANDLE)HandleTable->Handles[i].HandleValue, &szObjectName, MAX_PATH * 2, TRUE) ) {
                                if (strstrW(szObjectName, VENDOR_SANDBOXIE) != NULL) {
                                    #ifdef _DEBUG
                                    /* do something */
                                    #endif
                                }
                            }
                        }
                    }
                    if (QueryObjectName((HANDLE)HandleTable->Handles[i].HandleValue, &szObjectName, MAX_PATH * 2, TRUE) ) {
                        if (strstrW(szObjectName, VENDOR_SANDBOXIE) != NULL) {
                            #ifdef _DEBUG
                            /* do something */
                            #endif
                        }
                    }
                }
            }
        }
    }
```
DebugLog(TEXT("Sandboxie::HandleTable"));
#endif
    IsSB = 1;
    break;
  }
}
}

/* brute-force memory to locate sandboxie injected code and locate sandboxie tag */
if ( IsSB == 0 ) {
  i = (ULONG_PTR)g_siSysInfo.lpMinimumApplicationAddress;
  do {
    Status = NtQueryVirtualMemory(NtCurrentProcess(), (PVOID)i,
         MemoryBasicInformation,
         &RegionInfo, sizeof(MEMORY_BASIC_INFORMATION), &uLength);
    if (NT_SUCCESS(Status)) {
      if (IsExecutableCode(RegionInfo.AllocationProtect,
          RegionInfo.State)) {
        for (k=i; k<i+RegionInfo.RegionSize;
           k+=sizeof(DWORD)) {
          if (*((PDWORD)k) == 'kuzt') {
            IsSB = 1;
            break;
          }
        }
      }
    } else {
      i += PAGE_SIZE;
    }
  } while (i<(ULONG_PTR)g_siSysInfo.lpMaximumApplicationAddress);
}

/* abuse sandboxie sbiedll.dll bug/lame behaviour */
if ( IsSB == 0 ) {
  IsSB = IsSandboxieVirtualRegistryPresent();
#endif
    if (IsSB == 1) DebugLog(TEXT("Sandboxie::VirtualRegistry"));
#endif
}

__finally {
    mmfree(HandleTable);
    if ( hDummy != NULL ) NtClose(hDummy);
}

return IsSB;
}
3.7. Hypervisor detection

BOOL IsHypervisor(
    VOID
)
{
    int CPUInfo[4] = {-1};
    /*
     * query hypervisor presence
     * be aware this detection can be bogus
     */
    __cpuid(CPUInfo, 1);
    if ((CPUInfo[2] >> 31) & 1) {
        return TRUE;
    }
    return FALSE;
}

BYTE GetHypervisorType(
    VOID
)
{
    int CPUInfo[4] = {-1};
    char HvProductName[0x40];
    __cpuid(CPUInfo, 0x40000000);
    RtlSecureZeroMemory(HvProductName, sizeof(HvProductName));
    memcpy(HvProductName, CPUInfo + 1, 12);
    if (_strcmpiA(HvProductName, "Microsoft Hv") == 0) {
        return 1;
    }
    /*
    if (_strcmpiA(HvProductName, "VMwareVMMware") == 0) {
        return 2;
    }
    /* Parallels VM ids */
    if (_strcmpiA(HvProductName, "prl hyperv") == 0) {
        return 3;
    }
    return 0;
}
3.8. VMDE support routines

 PVOID GetFirmwareTable(
    PULONG pdwDataSize,
    DWORD dwSignature,
    DWORD dwTableID
)
{
    NTSTATUS Status;
    ULONG Length;
    HANDLE hProcess = NULL;
    ULONG uAddress;
    PSYSTEM_FIRMWARE_TABLE_INFORMATION pSIF = NULL;
    SIZE_T memIO = 0;
    CLIENT_ID cid;
    OBJECT_ATTRIBUTES attr;
    MEMORY_REGION_INFORMATION memInfo;

    /* use documented GetSystemFirmwareTable instead, this is its raw implementation */
    if (g_osVer.dwMajorVersion > 5) {
        Length = 0x1000;
        pSIF = (PSYSTEM_FIRMWARE_TABLE_INFORMATION)RtlAllocateHeap(RtlProcessHeap(),
            HEAP_ZERO_MEMORY, Length);
        if (pSIF != NULL) {
            pSIF->Action = SystemFirmwareTable_Get;
            pSIF->ProviderSignature = dwSignature;
            pSIF->TableID = dwTableID;
            pSIF->TableBufferLength = Length;
            /* query if info class available and if how many memory we need */
            Status = NtQuerySystemInformation(SystemFirmwareTableInformation, pSIF,
                Length, &Length);
            if (Status == STATUS_INVALID_INFO_CLASS || Status ==
                STATUS_INVALID_DEVICE_REQUEST) {
                RtlFreeHeap(RtlProcessHeap(), 0, pSIF);
                return NULL;
            }
            if (!NT_SUCCESS(Status) || Status == STATUS_BUFFER_TOO_SMALL) {
                pSIF = (PSYSTEM_FIRMWARE_TABLE_INFORMATION)RtlAllocateHeap(RtlProcessHeap(),
                    HEAP_ZERO_MEMORY, Length);
                if (pSIF != NULL) {
                    pSIF->Action = SystemFirmwareTable_Get;
                    pSIF->ProviderSignature = dwSignature;
                    pSIF->TableID = dwTableID;
                    pSIF->TableBufferLength = Length;
                    Status = NtQuerySystemInformation(SystemFirmwareTableInformation, pSIF,
                        Length, &Length);
                    if (!NT_SUCCESS(Status)) {
                        RtlFreeHeap(RtlProcessHeap(), 0, pSIF);
                        return NULL;
                    }
                    if (ARGUMENT_PRESENT(pdwDataSize))
                        *pdwDataSize = Length;
                }
            } else {
                if (ARGUMENT_PRESENT(pdwDataSize))
                    *pdwDataSize = Length;
            }
        } else {
            /* on pre Win2k3 SP1 systems the above info class unavailable, but all required
                information can be found inside csrss memory space (stored here for VDM purposes) at few fixed
                addresses */
            if ((dwSignature != FIRM) && (dwSignature != RSMB)) return NULL;
            /* we are interested only in two memory regions */
            switch (dwTableID) {
                case FIRM:
                    uAddress = 0xC0000; /* FIRM analogue */
                    break;
                case RSMB:
                    uAddress = 0xE0000; /* RSMB analogue */
                    break;
            }
        }
    } else {
        /* use documented GetSystemFirmwareTable instead, this is its raw implementation */
        if (ARGUMENT_PRESENT(pdwDataSize))
            *pdwDataSize = Length;
    }
break;
  default:
    return NULL;
    break;
  
  Length = 0;
  cid.UniqueProcess = (HANDLE)CsrGetProcessId();
  cid.UniqueThread = 0;
  InitializeObjectAttributes(&attr, NULL, 0, 0, NULL);
  /* open csrss, reg. client debug privilege set */
  Status = NtOpenProcess(hProcess, PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
    &attr, &cid);
  
  if (NT_SUCCESS(Status)) {
    /* get memory data region size for buffer allocation */
    Status = NtQueryVirtualMemory(hProcess, (PVOID)uAddress,
      MemoryRegionInformation, &memInfo, sizeof(MEMORY_REGION_INFORMATION), &memIO);
    if (NT_SUCCESS(Status)) {
      pSIF = (PSYSTEM_FIRMWARE_TABLE_INFORMATION)RtlAllocateHeap(RtlProcessHeap(), HEAP_ZERO_MEMORY,
        memInfo.RegionSize);
      if (pSIF != NULL) {
        /* read data to our allocated buffer */
        Status = NtReadVirtualMemory(hProcess, (PVOID)uAddress,
          pSIF, memInfo.RegionSize, &memIO);
        if (NT_SUCCESS(Status)) {
          if (ARGUMENT_PRESENT(pdwDataSize))
            *pdwDataSize = memInfo.RegionSize;
        } else {
          RtlFreeHeap(RtlProcessHeap(), 0, pSIF);
          return NULL;
        }
      }
      NtClose(hProcess);
    }
  
  return pSIF;
}

BOOL QueryObjectName(
  HKEY hKey,
  PVOID Buffer,
  ULONG BufferSize,
  BOOL IsUnicodeCall
) {

  POBJECT_NAME_INFORMATION pObjName;
  NTSTATUS Status;
  ULONG ReturnLength;
  BOOL bResult;
  ULONG len;
  LPSTR StrA = NULL;
  LPWSTR StrW = NULL;
  pObjName = NULL;
  ReturnLength = 0;
  bResult = FALSE;

  if ( IsUnicodeCall ) {
    StrW = (LPWSTR)Buffer;
  } else {
    StrA = (LPSTR)Buffer;
  }

  __try {
    NtQueryObject(hKey, ObjectNameInformation, NULL, ReturnLength, &ReturnLength);
    pObjName = (POBJECT_NAME_INFORMATION)mmalloc(ReturnLength);
    if ( pObjName == NULL )
      __leave;
    Status = NtQueryObject(hKey, ObjectNameInformation, pObjName, ReturnLength, NULL);
    if (NT_SUCCESS(Status)) {

if ( pObjName->Name.Buffer != NULL && pObjName->Name.Length > 0 ) { }

    bResult = TRUE;
    len = (ULONG)_strlenW(pObjName->Name.Buffer);
    if (len > BufferSize) len = BufferSize;
    if ( IsUnicodeCall ) {
        _strncpyW(StrW, BufferSize, pObjName->Name.Buffer,
            BufferSize);
    } else {
        WideCharToMultiByte(CP_ACP, 0, pObjName->Name.Buffer,
            len, StrA, len, 0, 0);
    }
} __finally {
    if ( pObjName != NULL) mmfree(pObjName);
} return bResult;

VOID EnumPCIDevReg(
    VOID
) {
    HANDLE hKey = NULL;
    DWORD dwKeySubIndex = 0;
    ULONG ResultLength = 0;
    NTSTATUS Status = STATUS_UNSUCCESSFUL;
    UNICODE_STRING ustrKeyName;
    OBJECT_ATTRIBUTES obja;
    PKEY_BASIC_INFORMATION pKeyInfo = NULL;
    PVIDOR_ENTRY entry;
    WCHAR szTempBuf[MAX_PATH] = {0};
    RtlInitUnicodeString(&ustrKeyName, REGSTR_KEY_PCIENUM);
    InitializeObjectAttributes(&obja, &ustrKeyName, OBJ_CASE_INSENSITIVE, NULL, NULL);
    __try {
        Status = NtOpenKey(&hKey, KEY_ENUMERATE_SUB_KEYS, &obja);
        if ( hKey == NULL && !NT_SUCCESS(Status)) __leave ;
        do {
            NtEnumerateKey(hKey, dwKeySubIndex, KeyBasicInformation, NULL, 0,
                &ResultLength);
            pKeyInfo = (PKEY_BASIC_INFORMATION)RtlAllocateHeap(RtlProcessHeap(),
                HEAP_ZERO_MEMORY, ResultLength);
            if ( pKeyInfo == NULL) __leave;
            Status = NtEnumerateKey(hKey, dwKeySubIndex, KeyBasicInformation, pKeyInfo,
                ResultLength, &ResultLength);
            if (NT_SUCCESS(Status)) {
                entry = (PVENDOR_ENTRY)mmalloc(sizeof(VENDOR_ENTRY));
                if ( entry ) { _strncpyW(entry->VendorFullName, MAX_PATH, pKeyInfo->Name,
                    pKeyInfo->NameLength / sizeof(WCHAR));
                    vExtractID(entry);
                    InsertHeadList(&VendorsListHead, &entry->ListEntry);
                }
                RtlFreeHeap(RtlProcessHeap(), 0, pKeyInfo);
                pKeyInfo = NULL;
            }
            dwKeySubIndex++;
        } while ( NT_SUCCESS(Status) );
    } __finally {
        if (hKey != NULL) NtClose(hKey);
        if ( pKeyInfo != NULL) RtlFreeHeap(RtlProcessHeap(), 0, pKeyInfo);
    }
}
All the code provided for education purposes, we are not responsible for it usage or any bugs it may have.

Microsoft, Windows, VMware, VirtualBox, Parallels (and other vendor), product names are registered trademarks or may be registered trademarks of their respective owners.

This document converted from slides.