6

Navigating Anti-Virtual Machine Strategies

Anti-virtual machine techniques are predominantly found in widely spread malware, such as bots, scareware, and spyware, mainly because honeypots often use virtual machines and these types of malware generally target the average user's computer, which is unlikely to be running a virtual machine. In this chapter, you will learn how to employ **anti-virtual machine** (**anti-VM**) strategies to counteract attempts at analysis.

In this chapter, we're going to cover the following main topics:

- Filesystem detection techniques
- Approaches to hardware detection
- Time-based sandbox evasion techniques
- Identifying VMs through the registry

Technical requirements

In this book, I will use the Kali Linux (**https://www.kali.org/**) and Parrot Security OS (**https://www.parrotsec.org/**) VMs for development and demonstration, and Windows 10 (**https://www.microsoft.com/en-us/software-download/windows10ISO**) as the victim's machine.

The next thing we'll want to do is set up our development environment in Kali Linux. We'll need to make sure we have the necessary tools installed, such as a text editor and compiler.

I use NeoVim (**https://github.com/neovim/neovim**) with syntax highlighting as a text editor. Neovim is a great choice for a lightweight, efficient text editor, but you can use another that you like – for example, VS Code (**https://code.visualstudio.com/**).

As far as compiling our examples, I use MinGW (**https://www.mingw-w64.org/**) for Linux, which is installed in my case via the following command:

```
$ sudo apt install mingw-*
```

Filesystem detection techniques

All filesystem detection methods conform to the following principle – *such files and directories do not exist on a typical host, but they do exist in virtual environments and sandboxes. If such an artifact is present, it can be detected as virtualized.*

Let's check whether specific files exist.

VirtualBox machine detection

If the target system has the following files, then the target system is most likely a VirtualBox VM:

- **`c:\windows\system32\drivers\VBoxMouse.sys`**
- **`c:\windows\system32\drivers\VBoxGuest.sys`**
- **`c:\windows\system32\drivers\VBoxSF.sys`**
- **`c:\windows\system32\drivers\VBoxVideo.sys`**
- **`c:\windows\system32\vboxdisp.dll`**
- **`c:\windows\system32\vboxhook.dll`**
- **`c:\windows\system32\vboxservice.exe`**
- **`c:\windows\system32\vboxtray.exe`**

A practical example

This filesystem detection technique method makes use of the file differences between a typical host system and virtual environments. There are numerous file artifacts in virtual environments that are unique to these types of systems. On typical host systems where no virtual environment is installed, these files are absent.

Let's create code that will check the system for the presence of these artifacts. Here is the full C code with a function named **`checkVM`** that checks for the existence of the specified paths:

```
/*
 * Malware Development for Ethical Hackers
 * hack.c - Anti-VM tricks
 * check filesystem
 * author: @cocomelonc
*/
#include <windows.h>
#include <stdio.h>
BOOL checkVM() {
  // Paths to check
  LPCSTR path1 = "c:\\windows\\system32\\drivers\\VBoxMouse.sys";
  LPCSTR path2 = "c:\\windows\\system32\\drivers\\VBoxGuest.sys";
  // Use GetFileAttributes to check if the first file exists
  DWORD attributes1 = GetFileAttributes(path1);
  // Use GetFileAttributes to check if the second file exists
  DWORD attributes2 = GetFileAttributes(path2);
  // Check if both files exist
  if ((attributes1 != INVALID_FILE_ATTRIBUTES && !(attributes1 & FILE_ATTRIBUTE_DIRECTORY)) ||
        (attributes2 != INVALID_FILE_ATTRIBUTES && !(attributes2 & FILE_ATTRIBUTE_DIRECTORY))) {
        // At least one of the files exists
        return TRUE;
  } else {
        // Both files do not exist or are directories
        return FALSE;
  }
}
int main() {
  if (checkVM()) {
    printf("The system appears to be a virtual machine.\n");
  } else {
    printf("The system does not appear to be a virtual machine.\n");
    printf("hacking...");
  }
  return 0;
}
```

The full source code of this logic can be found here: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter06/01-filesystem/hack.c.

As you can see, for simplicity, this function checks only two paths:

- `c:\windows\system32\drivers\VBoxMouse.sys`
- `c:\windows\system32\drivers\VBoxGuest.sys`

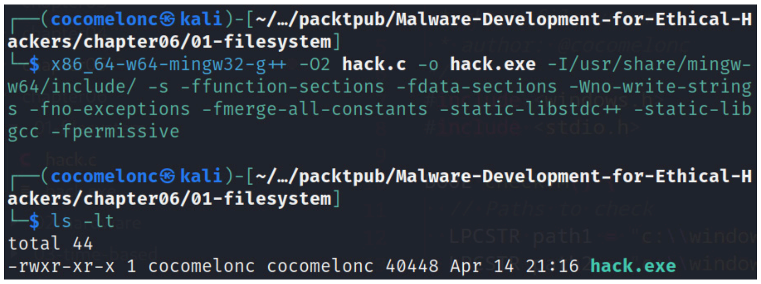
However, you can update this logic to check other artifacts as well.

Demo

As usual, we will compile our example in a Kali VM:

```
$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sect
```

On Kali Linux, it looks like this:



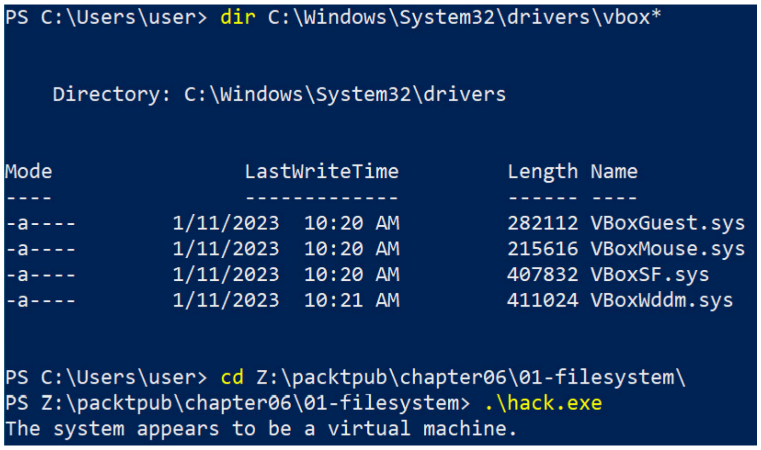
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-H
ackers/chapter06/01-filesystem]
└─$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-
w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-string
s -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-lib
gcc -fpermissive

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-H
ackers/chapter06/01-filesystem]
└─$ ls -lt
total 44
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Apr 14 21:16 hack.exe
```

Figure 6.1 – Compiling the hack.c example

Then, we will run it on our target Windows VirtualBox machine:

```
> .\hack.exe
```

On a Windows 10 VM, it looks like this:



```
PS C:\Users\user> dir C:\Windows\System32\drivers\vbox*


    Directory: C:\Windows\System32\drivers


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         1/11/2023  10:20 AM         282112 VBoxGuest.sys
-a----         1/11/2023  10:20 AM         215616 VBoxMouse.sys
-a----         1/11/2023  10:20 AM         407832 VBoxSF.sys
-a----         1/11/2023  10:21 AM         411024 VBoxWddm.sys


PS C:\Users\user> cd Z:\packtpub\chapter06\01-filesystem\
PS Z:\packtpub\chapter06\01-filesystem> .\hack.exe
The system appears to be a virtual machine.
```

Figure 6.2 – Running hack.exe in the VM

As you can see from the preceding screenshot, the specified files are actually present on the target Windows VirtualBox machine.

Of course, checking the filesystem may not be enough, so the next method is checking the hardware.

Approaches to hardware detection

Virtual environments imitate hardware devices and leave specific traces in their descriptions, which can be queried to determine the non-host OS.

Checking the HDD

One of the techniques is verifying that the HDD vendor ID has a specific value. For this logic, the following function is used:

```
BOOL DeviceIoControl(
  HANDLE       hDevice,
  DWORD        dwIoControlCode,
  LPVOID       lpInBuffer,
  DWORD        nInBufferSize,
  LPVOID       lpOutBuffer,
  DWORD        nOutBufferSize,
  LPDWORD      lpBytesReturned,
  LPOVERLAPPED lpOverlapped
);
```

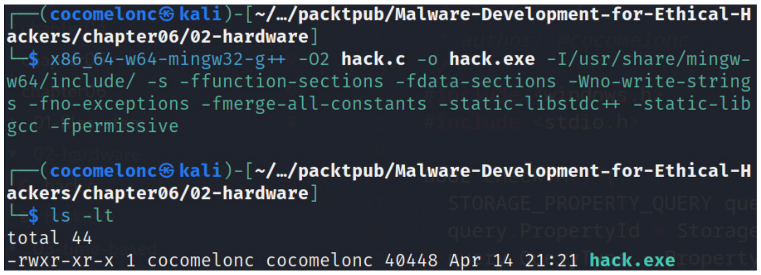
The full source code of this logic can be found here: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter06/02-hardware/hack.c.

Demo

Let's compile our example:

```
$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sect
```

On Kali Linux, it looks like this:



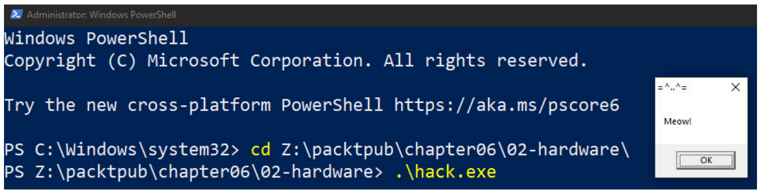
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter06/02-hardware]
└─$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter06/02-hardware]
└─$ ls -lt
total 44
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Apr 14 21:21 hack.exe
```

Figure 6.3 – Compiling the hack.c example

Then, we will run it on our target's Windows VirtualBox machine:

```
> .\hack.exe
```

On a Windows 10 machine, it looks like this:



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Windows\system32> cd Z:\packtpub\chapter06\02-hardware\
PS Z:\packtpub\chapter06\02-hardware> .\hack.exe
```

Figure 6.4 – Running hack.exe in the VM

As you can see, everything worked as expected.

However, checking the hardware in some cases will not bring the desired results, so let's show a technique based on time.

Time-based sandbox evasion techniques

Sandbox emulation is typically brief because sandboxes are typically filled with thousands of samples. Rarely does emulation time exceed three to five minutes. Malware can, therefore, take advantage of this fact to avoid detection by delaying its malicious actions for an extended period of time.

Sandboxes can incorporate features that manipulate time and execution delays to counteract this. **Cuckoo Sandbox**, for instance, has a sleep-skipping feature that replaces delays with a very brief value. This should compel the malware to initiate its malicious behavior prior to the expiration of the analysis timer.

A simple example

Delaying execution may circumvent sandbox analysis by exceeding the sample execution's duration limit. Nonetheless, it is not as simple as `Sleep(1000000)`.

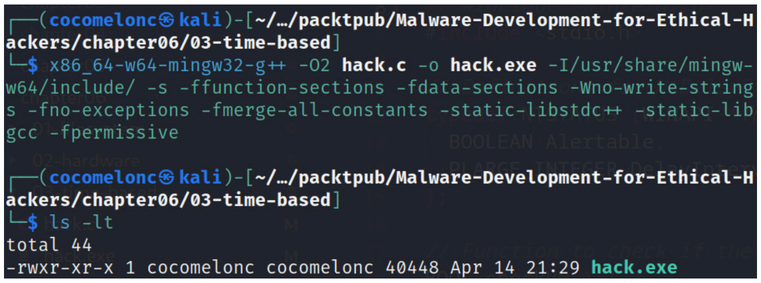
We can check the uptime of the system before and after sleeping. Additionally, we can use a lower-level userland API for sleeping (there is a slightly smaller possibility that it is hooked by AV). This necessitates dynamically obtaining the function's address; it will be used more broadly during the API call obfuscation described in one of the following chapters. Additionally, the **NtDelayExecution** function requires a distinct format for the sleep time parameter rather than **Sleep**. The code can be found here: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter06/03-time-based/hack.c.

This preceding code is a crude **proof of concept** (**PoC**); for simplicity, we print messages and show message boxes.

Let's examine everything in action. We'll compile our PoC source code:

```
$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sect
```

On Kali Linux, it looks like this:



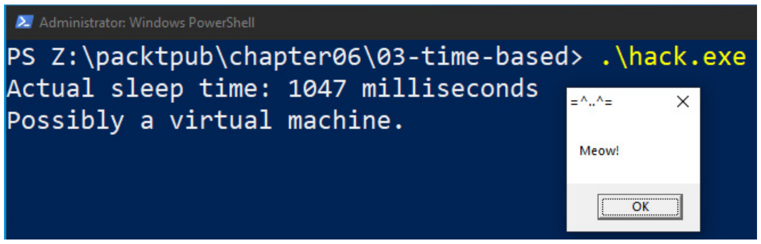
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter06/03-time-based]
└─$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter06/03-time-based]
└─$ ls -lt
total 44
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Apr 14 21:29 hack.exe
```

Figure 6.5 – Compiling our "malware"

Then, we'll execute it on the victim's computer:

```
> .\hack.exe
```

On a Windows 10 x64 VM, it looks like this:



```
PS Z:\packtpub\chapter06\03-time-based> .\hack.exe
Actual sleep time: 1047 milliseconds
Possibly a virtual machine.
```

Figure 6.6 – Running our malware

Absolutely perfect!

There are also other methods that are based on time – for example, deferred execution using Task Scheduler, sleep-skipping detection (which is used in most cases to detect Cuckoo Sandbox), or measuring time intervals using different methods. But we won't consider them; that would require a whole chapter. I just provided simple examples so that you can understand the concept.

Identifying VMs through the registry

The underlying principle of all registry detection methods is that such registry keys and values do not exist on a typical host. Nevertheless, they exist in specific virtual environments.

The presence of VM artifacts on a typical system that has VMs installed can occasionally result in false positives when these tests are performed. In contrast to virtual environments, this system is treated cleanly in all other respects.

The first technique verifies the existence of specified registry paths. I can verify this using the following logic:

```
int registryKeyExist(HKEY rootKey, char* subKeyName) {
  HKEY registryKey = nullptr;
  LONG result = RegOpenKeyExA(rootKey, subKeyName, 0, KEY_READ, &registryKey);
  if (result == ERROR_SUCCESS) {
        RegCloseKey(registryKey);
        return TRUE;
  }
  return FALSE;
}
```

As you can see, I simply verify the existence of the registry key path. **`TRUE`** is returned if the value exists; otherwise, **`FALSE`** is returned.

Another trick involves determining whether a particular registry key contains a value. For example, consider the following reasoning:

```
int compareRegistryKeyValue(HKEY rootKey, char* subKeyName, char* registryValue, char* comparisonV
  HKEY registryKey = nullptr;
  LONG result;
  char retrievedValue[1024];
  DWORD size = sizeof(retrievedValue);
  result = RegOpenKeyExA(rootKey, subKeyName, 0, KEY_READ, &registryKey);
  if (result == ERROR_SUCCESS) {
        RegQueryValueExA(registryKey, registryValue, NULL, NULL, (LPBYTE)retrievedValue, &size);
        if (result == ERROR_SUCCESS) {
        if (strcmp(retrievedValue, comparisonValue) == 0) {
        return TRUE;
        }
        }
  }
  return FALSE;
}
```

This function's logic is similarly straightforward. We verify the value of the registry key via **`RegQueryValueExA`**, in which the result of the **`RegOpenKeyExA`** function is the first parameter.

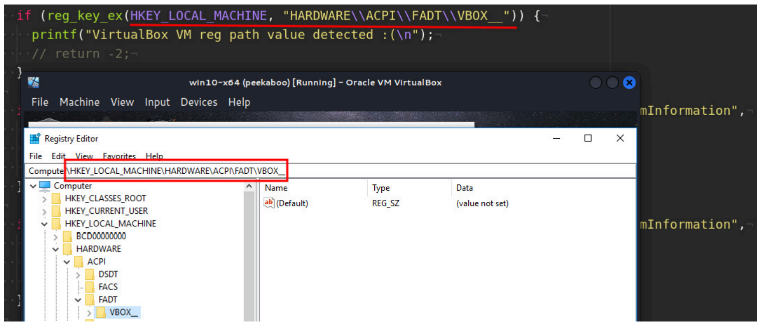
I'll evaluate only Oracle VirtualBox. For additional VMs/sandboxes, the same techniques apply.

A practical example

Consequently, let's look at a practical example. Let's inspect the full source code, which you can find on our GitHub repo: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter06/04-registry/hack.c.

As you can see, this is basically a typical payload injection attack, with VM VirtualBox detection tricks via the Windows Registry.

Let's check the **`HKLM\HARDWARE\ACPI\FADT\VBOX_`** path:

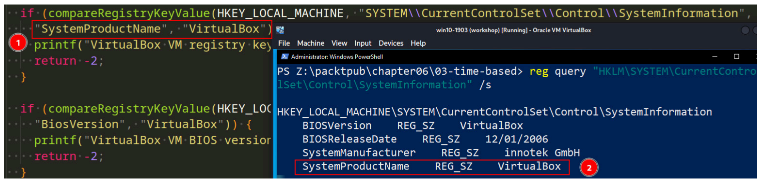


```
if (reg_key_ex(HKEY_LOCAL_MACHINE, "HARDWARE\\ACPI\\FADT\\VBOX__")) {
  printf("VirtualBox VM reg path value detected :(\n");
  // return -2;
}
```

win10-x64 (peekaboo) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Registry Editor
File Edit View Favorites Help
Computer\HKEY_LOCAL_MACHINE\HARDWARE\ACPI\FADT\VBOX__
Computer
HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
BCD00000000
HARDWARE
ACPI
DSDT
FACS
FADT
VBOX__
Name Type Data
(Default) REG_SZ (value not set)

Figure 6.7 – Checking HKLM\HARDWARE\ACPI\FADT\VBOX_

Enumerate the **`SystemProductName`** registry key from **`HKLM\SYSTEM\CurrentControlSet\Control\SystemInformation`** and compare it with the VirtualBox string:



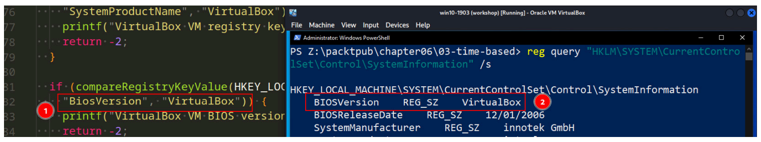
```
  if (compareRegistryKeyValue(HKEY_LOCAL_MACHINE, "SYSTEM\\CurrentControlSet\\Control\\SystemInformation",
    "SystemProductName", "VirtualBox")
    printf("VirtualBox VM registry key
    return -2;
  }

  if (compareRegistryKeyValue(HKEY_LOC
    "BiosVersion", "VirtualBox")) {
    printf("VirtualBox VM BIOS versior
    return -2;
  }
```

win10-1903 (workshop) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Administrator: Windows PowerShell
1
2

```
PS Z:\packtpub\chapter06\03-time-based> reg query "HKLM\SYSTEM\CurrentContro
lSet\Control\SystemInformation" /s

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SystemInformation
    BIOSVersion    REG_SZ    VirtualBox
    BIOSReleaseDate    REG_SZ    12/01/2006
    SystemManufacturer    REG_SZ    innotek GmbH
    SystemProductName    REG_SZ    VirtualBox
```

Figure 6.8 – Checking ...\Control\SystemInformation

Enumerate a BIOS version key, **`BiosVersion`**, from the same location:



```
    "SystemProductName", "VirtualBox")
    printf("VirtualBox VM registry key
    return -2;
  }

  if (compareRegistryKeyValue(HKEY_LOC
    "BiosVersion", "VirtualBox")) {
    printf("VirtualBox VM BIOS versior
    return -2;
```

win10-1903 (workshop) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Administrator: Windows PowerShell
1
2

```
PS Z:\packtpub\chapter06\03-time-based> reg query "HKLM\SYSTEM\CurrentContro
lSet\Control\SystemInformation" /s

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SystemInformation
    BIOSVersion    REG_SZ    VirtualBox
    BIOSReleaseDate    REG_SZ    12/01/2006
    SystemManufacturer    REG_SZ    innotek GmbH
```

Figure 6.9 – Checking BiosVersion

IMPORTANT NOTE

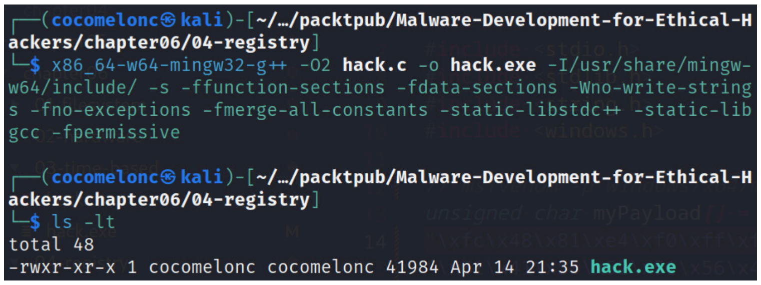
Note that key names are always case-insensitive.

Demo

Let's examine everything in action. We'll compile our example on the machine of the attacker (the Kali Linux x64 or Parrot Security OS):

```
$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sect
```

On Kali Linux, it looks like this:



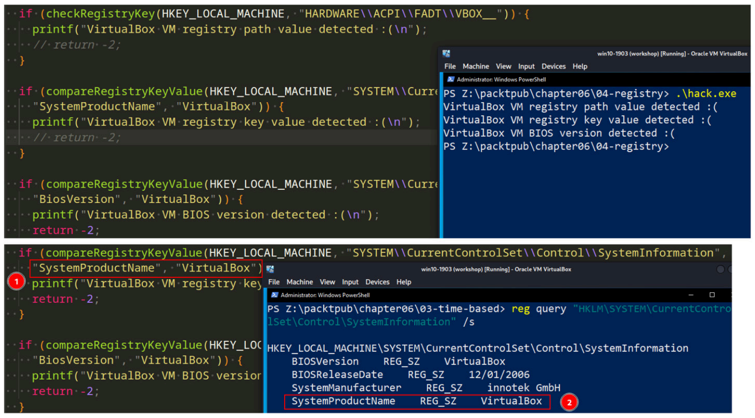
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter06/04-registry]
└─$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter06/04-registry]
└─$ ls -lt
total 48
-rwxr-xr-x 1 cocomelonc cocomelonc 41984 Apr 14 21:35 hack.exe
```

Figure 6.10 – Compiling PoC code

Then, on the victim's machine (Windows 10 x64 in my case), execute it:

```
> .\hack.exe
```

On Windows 10 x64 VM, it looks like this:



```
  if (checkRegistryKey(HKEY_LOCAL_MACHINE, "HARDWARE\\ACPI\\FADT\\VBOX__")) {
    printf("VirtualBox VM registry path value detected :(\n");
    // return -2;
  }

  if (compareRegistryKeyValue(HKEY_LOCAL_MACHINE, "SYSTEM\\Curre
    "SystemProductName", "VirtualBox")) {
    printf("VirtualBox VM registry key value detected :(\n");
    // return -2;
  }

  if (compareRegistryKeyValue(HKEY_LOCAL_MACHINE, "SYSTEM\\Curre
    "BiosVersion", "VirtualBox")) {
    printf("VirtualBox VM BIOS version detected :(\n");
    return -2;
```

win10-1903 (workshop) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Administrator: Windows PowerShell

```
PS Z:\packtpub\chapter06\04-registry> .\hack.exe
VirtualBox VM registry path value detected :(
VirtualBox VM registry key value detected :(
VirtualBox VM BIOS version detected :(
PS Z:\packtpub\chapter06\04-registry>
```

```
  if (compareRegistryKeyValue(HKEY_LOCAL_MACHINE, "SYSTEM\\CurrentControlSet\\Control\\SystemInformation",
    "SystemProductName", "VirtualBox")
    printf("VirtualBox VM registry key
    return -2;
  }

  if (compareRegistryKeyValue(HKEY_LOC
    "BiosVersion", "VirtualBox")) {
    printf("VirtualBox VM BIOS versior
    return -2;
  }
```

1
win10-1903 (workshop) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Administrator: Windows PowerShell
2

```
PS Z:\packtpub\chapter06\03-time-based> reg query "HKLM\SYSTEM\CurrentControlSet\Control\SystemInformation" /s

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SystemInformation
    BIOSVersion    REG_SZ    VirtualBox
    BIOSReleaseDate    REG_SZ    12/01/2006
    SystemManufacturer    REG_SZ    innotek GmbH
    SystemProductName    REG_SZ    VirtualBox
```

Figure 6.11 – Running the "malware"

If we delve into the investigation of real-world malware and scenarios, we will undoubtedly discover numerous other specified registry paths and keys.

For example, in the **`HKLM\SYSTEM\ControlSet001\Services\Disk\Enum`** registry path, **`DeviceDesc`** and **`FriendlyName`** are equal to **`VBOX`**, and in the **`HKLM\SYSTEM\CurrentControlSet\Control\SystemInformation`** path, **`SystemProductName`**'s value is **`VIRTUAL`** or **`VIRTUALBOX`**.

In specific scenarios, malware might iterate through sub-keys and verify whether the name of the subkey contains a particular string, rather than checking the existence of the specified key directly.

Of course, I have given the simplest examples here so that you can easily re-implement such logic in practice in a local laboratory or during pen-tests. In real malware, the logic of the checks is the same, but the steps can be more confusing and sophisticated.

All these and many other methods are used by adversaries in real attacks. You can study them in more detail on this page: https://attack.mitre.org/techniques/T1497/.

Summary

This chapter delved into the complex world of anti-VM strategies, acknowledging their prevalence in malware that targets common users. As VMs become commonplace in cybersecurity analysis, malware developers employ sophisticated methods to avoid detection in these environments. The discussed techniques, which are prevalent in malware, scareware, and spyware, play a crucial role in evading VM-based honeypots. By averting analysis within VMs, these types of malware increase their chances of infiltrating the systems of unsuspecting users.

Throughout the chapter, you were provided with a variety of applicable skills. Through meticulous analysis of filesystem artifacts, you acquired an in-depth understanding of filesystem detection techniques and learned to decipher VMs and sandboxes. In addition, you mastered the art of hardware detection, gaining the ability to recognize VMs based on nuanced hardware data. The chapter also delved into time-based sandbox evasion techniques, providing you with insights into strategies employed by malware to thwart analysis within time-constrained environments. Lastly, you were instructed on how to identify VMs using registry keys, a crucial skill for developing malware attempting to conceal itself.

In the next chapter, we will cover how anti-disassembly uses specially crafted code or data in a program to cause disassembly analysis tools to produce an incorrect program listing.