

CONTENTS

- Patching
- API Hooking
- Anti-VM Techniques
- Code Obfuscation

PATCHING A BINARY

Patching is a mechanism to modify its binary code by identifying the buggy segments through disassembling the code.

When does it make sense to make binary patches?

When you have very few releases, a very big product, very few changes, and a very large user-base of people who don't make other changes to your product.

Whether it's to circumvent an anti-analysis check (anti analysis check: malware attempt to detect when they are being analyzed by AV systems, allowing them to change their behavior so as not to trigger any alarms.), or simply a bug that needs to be fixed, patching a binary is a useful technique to have in the reverse engineer's toolbox. Normally patching refers to a process where a patch file is generated and then applied to a binary. But today we will be patching by simply editing a few bytes in the binary with vim. It's quick, it's dirty, and it works.

Vulnerable Program:

```
#include<string.h>
#include<stdio.h>

int main(int argc, char *argv[]){
    if(argc==2){
        printf("Checking license: %s\n", argv[1]);
        if(strcmp(argv[1], "AAAA-Z10N-42-OK")==0) {
            printf("Access Granted!\n");
        } else {
            printf("WRONG!\n");
        }
    }else {
        printf("Usage: <key>\n");
    }
}
```

It is a simple license checking program.

We can see that the license is checked by strcmp function. if the strings match strcmp returns 0 and hence "access granted" is printed. This is vulnerability of patch.c

Examining its behavior:

```
kali@kali:~$ gcc patch.c -o patch
kali@kali:~$ ./patch
Usage: <key>
kali@kali:~$ ./patch AAAA
Checking license: AAAA
WRONG!
```

Finding the vulnerable point where you can patch it:

Open gdb and disassemble main

```
gdb -q patch
disass main
```

```
0x00000000000001184 <+47>: callq 0x1040 <printf@plt> " checking license"
```

```
0x00000000000001189 <+52>: mov -0x10(%rbp),%rax
```

```
0x0000000000000118d <+56>: add $0x8,%rax
```

```
0x00000000000001191 <+60>: mov (%rax),%rax
```

```
0x00000000000001194 <+63>: lea 0xe7f(%rip),%rsi # 0x201a
```

```
0x0000000000000119b <+70>: mov %rax,%rdi
```

```
0x0000000000000119e <+73>: callq 0x1050 <strcmp@plt> storing 0 in eax if password is correct
```

```
0x000000000000011a3 <+78>: test %eax,%eax
```

```
0x000000000000011a5 <+80>: jne 0x11b5 <main+96> jumps to 11b5 if its wrong
```

```
0x000000000000011a7 <+82>: lea 0xe7c(%rip),%rdi # 0x202a
```

```
0x000000000000011ae <+89>: callq 0x1030 <puts@plt> "Access Granted!"
```

```
0x000000000000011b3 <+94>: jmp 0x11cf <main+122>
```

```
0x000000000000011b5 <+96>: lea 0xe7e(%rip),%rdi # 0x203a
```

```
0x000000000000011bc <+103>: callq 0x1030 <puts@plt> "WRONG!"
```

```
0x000000000000011c1 <+108>: jmp 0x11cf <main+122>
```

```
0x000000000000011c3 <+110>: lea 0xe77(%rip),%rdi # 0x2041
```

```
0x000000000000011ca <+117>: callq 0x1030 <puts@plt> "usage key"
```

Observe that:

Jne is determining if we are jumping to the wrong case or the access granted case.

So if we would invert jne and make it a je, then when we supply a wrong key it should jump to access granted case.

```
0x00000000000011a5 <+80>: jne 0x11b5 <main+96>
0x00000000000011a7 <+82>: lea 0xe7c(%rip),%rdi
```

Jne starts at 11a5 and next is at 11a7. So the jne instruction is 2 bytes long.

So let us examine those 2 bytes.

```
(gdb) x/2b 0x00000000000011a5
0x11a5 <main+80>: 0x75 0x0e
```

The first byte indicates that it is a jne instruction.

The value of the second byte(e) is added to the next instruction hex value(11a7), and the result gives you at which address it would jump to (11b5).

<https://miniwebtool.com/hex-calculator/?number1=11a7&operate=1&number2=e>

$$11a7 + e = 11b5$$

So the second byte controls where it is going to jump.

<https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf>

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
75 cb	JNE rel8	D	Valid	Valid	Jump short if not equal (ZF=0).
74 cb	JE rel8	D	Valid	Valid	Jump short if equal (ZF=1).

Jne is 75 and je is 74.

So we only need to change this byte from 75 to 74.

To easily locate these bytes let us keep a pattern in mind: 75 0e 48 8d 3d 7c 00

```
(gdb) x/8b 0x00000000000011a5
0x11a5 <main+80>: 0x75 0x0e 0x48 0x8d 0x3d 0x7c 0x0e 0x00
```

Let's patch it up:

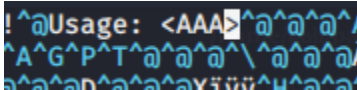
```
vim patch
```

You will get the binary file output of patch.c

- To search a string (eg: Usage):

```
/Usage
```

- Press enter and escape and i to enter insert mode.



I replaced key with AAA. Notice that when we delete a character, everything shifts. You must replace with same number of characters and must not move the binary around. Else, everything breaks. Thus, you can only change binaries in place.

- Escape and save by

```
:wq
```

- To see if it works, run it with no input.

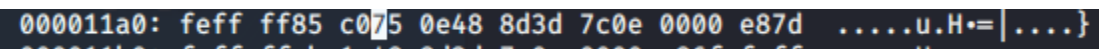
You will get Usage: <AAA> instead of the original Usage: <key>.

We have to find jne now.

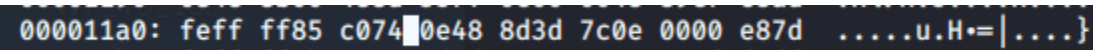
- Type the following to run the whole file through x60 which gives us a hex dump.

```
:%!xxd
```

- To find the pattern, type /75 and enter and keep pressing n till you find the pattern.



- To enter insert mode and change 75 to 74: Escape and i



- To return back to the binary file format:

```
Escape and :%!xxd -r
```

- Save it by

```
:wq
```

- Run the file with the wrong input:

```
kali@kali:~$ ./patch wrongkey  
Checking license: wrongkey  
Access Granted!
```

API HOOKING

<https://www.infosecurity-magazine.com/opinions/api-hooking-evading-detection/>

Traditional malware detection and forensic investigation techniques typically focus on detecting malicious native executables to disk, and performing disk forensics to uncover evidence of historical actions on a system.

In response, many threat actors have shifted their offensive techniques to avoid writing to disk, staying resident only in memory. Consequently, the ability to effectively analyze live memory for evidence of compromise and to gather additional forensic evidence has become increasingly important.

API hooking is one of the memory-resident techniques cyber-criminals are increasingly using. The process involves intercepting function calls in order to monitor and/or change the information passing back and forth between them. There are many reasons, both legitimate and malicious, why using this might be desirable. In the case of malware, the API hooking process is commonly considered to be 'rootkit' functionality and is mostly used to hide evidence of its presence on the system from other processes, and to spy on sensitive data.

How are the cyber-criminals using API hooking?

There are two common use cases for the malicious use of API hooking. Firstly, it can be used to spy on sensitive information and so they use it to intercept sensitive data, such as communications with the keyboard to log keystrokes including passwords that are typed by a user, or sensitive network communications before they are transmitted. This includes the ability to intercept data encrypted using protocols such as Transport Layer Security (TLS) prior to the point at which they are protected, in order to capture passwords and other sensitive data before it is transmitted.

Secondly, they modify the results returned from certain API calls in order to hide the presence of their malware. This commonly may involve file-system or registry related API calls to remove entries used by the malware, to hide its presence from other processes. Not only can cyber-criminals implement API hooking in a number of ways, the technique can also be deployed across a wide range of processes on a targeted system.

Tackling malicious API hooking

One way cybersecurity teams can detect the hidden traces of API hooking and other similar techniques is through memory analysis frameworks such as [Volatility](#). Volatility is an open source framework and the de facto standard toolset for performing memory analysis techniques against raw system memory images, useful in forensic investigations and malware analysis.

While memory analysis can be an incredibly powerful and useful technique, it does not come without its challenges. One hurdle to consider when deploying memory analysis is the labor intensity it requires: memory analysis is a highly skilled and time-intensive technique typically performed on one image at a time. This can be very effective when performing a dedicated investigation of a serious compromise, where the systems involved are known and relatively small in number.

However, the challenge arises when trying to use memory analysis at scale to detect compromises on a large enterprise network in the absence of any other evidence.

Another obstacle to be aware of when implementing memory analysis is legitimate 'bad' behavior. There are plenty of examples of hooking techniques being used by malware for malicious purposes. Nevertheless, there are also many cases of these techniques being used for legitimate, above-board purposes. In particular, technologies such as data loss prevention and anti-virus often target the same functions for hooking as malware does. Without the techniques and experience to quickly separate legitimate injection and hooking from malicious behavior, a great deal of time can be wasted.

Example: <https://www.apriorit.com/dev-blog/160-apihooks>

Anti-Virtual Machine Techniques

Malware authors sometimes use anti-virtual machine (anti-VM) techniques to thwart attempts at analysis. With these techniques, the malware attempts to detect whether it is being run inside a virtual machine. If a virtual machine is detected, it can act differently or simply not run. This can, of course, cause problems for the analyst.

Anti-VM techniques are most commonly found in malware that is widely deployed, such as bots, scareware, and spyware (mostly because honeypots often use virtual machines and because this malware typically targets the average user's machine, which is unlikely to be running a virtual machine).

The popularity of anti-VM malware has been going down recently, and this can be attributed to the great increase in the usage of virtualization. Traditionally, malware authors have used anti-VM techniques because they thought only analysts would be running the malware in a virtual machine. However, today both administrators and users use virtual machines in order to make it easy to rebuild a machine (rebuilding had been a tedious process, but virtual machines save time by allowing you to go back to a snapshot). Malware authors are starting to realize that just because a machine is a virtual machine does not necessarily mean that it isn't a valuable victim. As virtualization continues to grow, anti-VM techniques will probably become even less common.

Because anti-VM techniques typically target VMware, in this chapter, we'll focus on anti-VMware techniques. We'll examine the most common techniques and how to defeat them by tweaking a couple of settings, removing software, or patching an executable.

VMware Artifacts

The VMware environment leaves many artifacts on the system, especially when VMware Tools is installed. Malware can use these artifacts, which are present in the filesystem, registry, and process listing, to detect VMware.

For example, [Figure 17-1](#) shows the process listing for a standard VMware image with VMware Tools installed. Notice that three VMware processes are running: *VMwareService.exe*, *VMwareTray.exe*, and *VMwareUser.exe*. Any one of these can be found by malware as it searches the process listing for the VMware string.

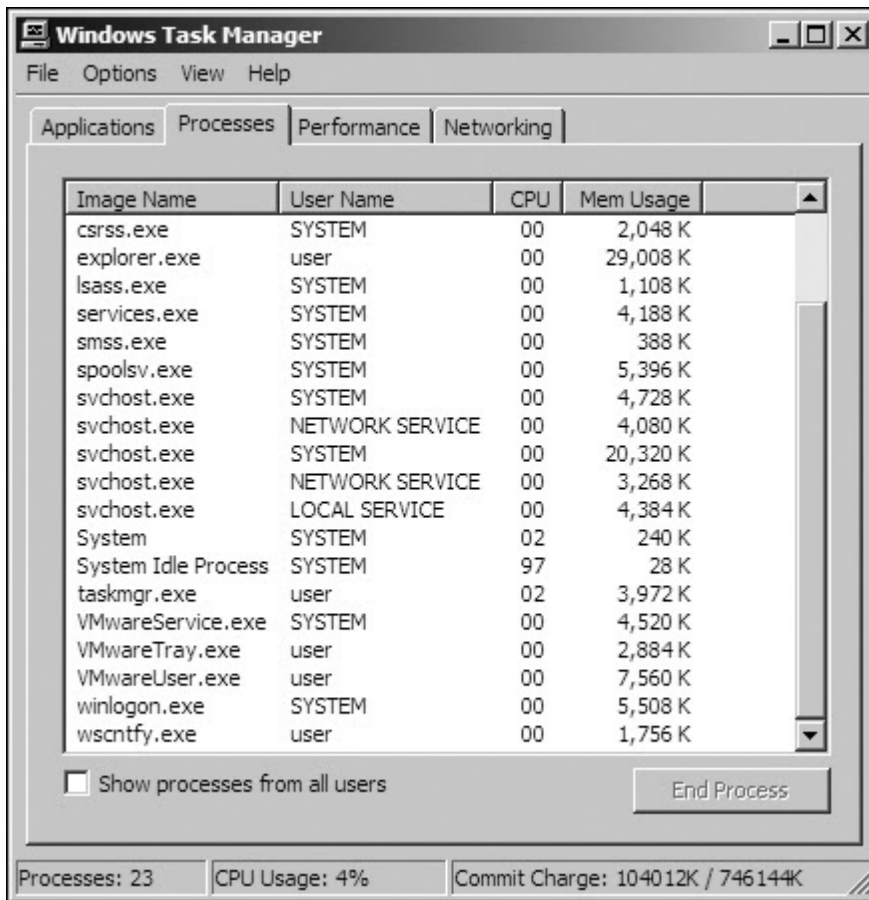


Figure 17-1. Process listing on a VMware image with VMware Tools running

VMwareService.exe runs the VMware Tools Service as a child of *services.exe*. It can be identified by searching the registry for services installed on a machine or by listing services using the following command:

```
C:\> net start | findstr VMware
```

```
VMware Physical Disk Helper Service
```

```
VMware Tools Service
```

The VMware installation directory *C:\Program Files\VMware\VMware Tools* may also contain artifacts, as can the registry. A quick search for “VMware” in a virtual machine’s registry might find keys like the following, which are entries that include information about the virtual hard drive, adapters, and virtual mouse.

```
[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0]
```

```
"Identifier"="VMware Virtual IDE Hard Drive"
```

```
"Type"="DiskPeripheral"
```

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Reinstall\0000]
```

```
"DeviceDesc"="VMware Accelerated AMD PCNet Adapter"
```

```
"DisplayName"="VMware Accelerated AMD PCNet Adapter"
```

```
"Mfg"="VMware, Inc."
```

```
"ProviderName"="VMware, Inc."
```

```
[HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Class\{4D36E96F-E325-11CE-BFC1-08002BE10318}\0000]
```

```
"LocationInformationOverride"="plugged into PS/2 mouse port"
```

```
"InfPath"="oem13.inf"
```

```
"InfSection"="VMMouse"
```

```
"ProviderName"="VMware, Inc."
```

You can connect your virtual machine to a network in a variety of ways, all of which allow the virtual machine to have its own virtual network interface card (NIC). Because VMware must virtualize the NIC, it needs to create a MAC address for the virtual machine, and, depending on its configuration, the network adapter can also identify VMware usage.

The first three bytes of a MAC address are typically specific to the vendor, and MAC addresses starting with 00:0C:29 are associated with VMware. VMware MAC addresses typically change from version to version, but all that a malware author needs to do is to check the virtual machine's MAC address for VMware values.

Malware can also detect VMware by other hardware, such as the motherboard. If you see malware checking versions of hardware, it might be trying to detect VMware. Look for the code that checks MAC addresses or hardware versions, and patch the code to avoid the check.

The most common VMware artifacts can be easily eliminated by uninstalling VMware Tools or by trying to stop the VMware Tools Service with the following command:

```
net stop "VMware Tools Service"
```

You may also be able to prevent malware from searching for artifacts. For example, if you find a single VMware-related string in malware—such as `net start | findstr VMware`, `VMMouse`, `VMwareTray.exe`, or `VMware Virtual IDE Hard Drive`—you know that the malware is attempting to detect VMware artifacts. You should be able to find this code easily in IDA Pro using the references to the strings. Once you find it, patch it to avoid detection while ensuring that the malware will function properly.

Bypassing VMware Artifact Searching

Defeating malware that searches for VMware artifacts is often a simple two-step process: identify the check and then patch it. For example, say we run `strings` against the malware `vmt.exe`. We notice that the binary contains the string `"VMwareTray.exe"`, and we discover a cross-reference from the code to this string. We follow this cross-reference to 0x401098, as shown in the disassembly in [Example 17-1](#) at ❶.

Example 17-1. Disassembly snippet from vmt.exe showing VMware artifact detection

```
0040102D      call ds:CreateToolhelp32Snapshot
00401033      lea ecx, [ebp+processentry32]
00401039      mov ebx, eax
0040103B      push ecx          ; lppe
0040103C      push ebx          ; hSnapshot
```

```

0040103D      mov [ebp+processentry32.dwSize], 22Ch
00401047      call ds:Process32FirstW
0040104D      mov esi, ds:WideCharToMultiByte
00401053      mov edi, ds:strncmp
00401059      lea esp, [esp+0]
00401060 loc_401060:      ; CODE XREF: sub_401000+B7j
00401060      push 0          ; lpUsedDefaultChar
00401062      push 0          ; lpDefaultChar
00401064      push 104h       ; cbMultiByte
00401069      lea edx, [ebp+Str1]
0040106F      push edx         ; lpMultiByteStr
00401070      push 0FFFFFFFFh ; cchWideChar
00401072      lea eax, [ebp+processentry32.szExeFile]
00401078      push eax         ; lpWideCharStr
00401079      push 0          ; dwFlags
0040107B      push 3          ; CodePage
0040107D      call esi ; WideCharToMultiByte
0040107F      lea eax, [ebp+Str1]
00401085      lea edx, [eax+1]
00401088 loc_401088:      ; CODE XREF: sub_401000+8Dj
00401088      mov cl, [eax]
0040108A      inc eax
0040108B      test cl, cl
0040108D      jnz short loc_401088
0040108F      sub eax, edx
00401091      push eax         ; MaxCount
00401092      lea ecx, [ebp+Str1]
00401098      push offset Str2 ; "VMwareTray.exe" ❶
0040109D      push ecx         ; Str1
0040109E      call edi ; strcmp ❷
004010A0      add esp, 0Ch
004010A3      test eax, eax
004010A5      jz short loc_4010C0
004010A7      lea edx, [ebp+processentry32]
004010AD      push edx         ; lppe
004010AE      push ebx         ; hSnapshot
004010AF      call ds:Process32NextW
004010B5      test eax, eax
004010B7      jnz short loc_401060
...
004010C0 loc_4010C0:      ; CODE XREF: sub_401000+A5j
004010C0      push 0          ; Code
004010C2      call ds:exit

```

Analyzing this code further, we notice that it is scanning the process listing with functions like `CreateToolhelp32Snapshot`, `Process32Next`, and so on. The `strcmp` at ❷ is comparing the `VMwareTray.exe` string with the result of converting `processentry32.szExeFile` to ASCII to determine if the process name is in the process listing. If `VMwareTray.exe` is discovered in the process listing, the program will immediately terminate, as seen at `0x4010c2`.

There are a couple of ways to avoid this detection:

- Patch the binary while debugging so that the jump at 0x4010a5 will never be taken.
- Use a hex editor to modify the `VMwareTray.exe` string to read `XXXareTray.exe` to make the comparison fail since this is not a valid process string.
- Uninstall VMware Tools so that *VMwareTray.exe* will no longer run.

Checking for Memory Artifacts

VMware leaves many artifacts in memory as a result of the virtualization process. Some are critical processor structures, which, because they are either moved or changed on a virtual machine, leave recognizable footprints.

One technique commonly used to detect memory artifacts is a search through physical memory for the string `VMware`, which we have found may detect several hundred instances.

Vulnerable Instructions

The virtual machine monitor program monitors the virtual machine's execution. It runs on the host operating system to present the guest operating system with a virtual platform. It also has a couple of security weaknesses that can allow malware to detect virtualization.

In kernel mode, VMware uses binary translation for emulation. Certain privileged instructions in kernel mode are interpreted and emulated, so they don't run on the physical processor. Conversely, in user mode, the code runs directly on the processor, and nearly every instruction that interacts with hardware is either privileged or generates a kernel trap or interrupt. VMware catches all the interrupts and processes them, so that the virtual machine still thinks it is a regular machine.

Some instructions in x86 access hardware-based information but don't generate interrupts. These include `sidt`, `sgdt`, `sldt`, and `cpuid`, among others. In order to virtualize these instructions properly, VMware would need to perform binary translation on every instruction (not just kernel-mode instructions), resulting in a huge performance hit. To avoid huge performance hits from doing full-instruction emulation, VMware allows certain instructions to execute without being properly virtualized. Ultimately, this means that certain instruction sequences will return different results when running under VMware than they will on native hardware.

The processor uses certain key structures and tables, which are loaded at different offsets as a side effect of this lack of full translation. The *interrupt descriptor table (IDT)* is a data structure internal to the CPU, which is used by the operating system to determine the correct response to interrupts and exceptions. Under x86, all memory accesses pass through either the *global descriptor table (GDT)* or the *local descriptor table (LDT)*. These tables contain segment descriptors that provide access details for each segment, including the base address, type, length, access rights, and so on. IDT (IDTR), GDT (GDTR), and LDT (LDTR) are the internal registers that contain the address and size of these respective tables.

Note that operating systems do not need to utilize these tables. For example, Windows implements a flat memory model and uses only the GDT by default. It does not use the LDT. Three sensitive instructions—`sidt`, `sgdt`, and `sldt`—read the location of these tables, and all store the respective register into a memory location. While these instructions are typically used by the operating system, they are not privileged in the x86 architecture, and they can be executed from user space.

An x86 processor has only three registers to store the locations of these three tables. Therefore, these registers must contain values valid for the underlying host operating system and will diverge from values expected by the virtualized (guest) operating system. Since the `sidt`, `sgdt`, and `sldt` instructions can be invoked at any time by user-mode code without being trapped and properly virtualized by VMware, they can be used to detect its presence.

Using the Red Pill Anti-VM Technique

Red Pill is an anti-VM technique that executes the `sidt` instruction to grab the value of the IDTR register. The virtual machine monitor must relocate the guest's IDTR to avoid conflict with the host's IDTR. Since the virtual machine monitor is not notified when the virtual machine runs the `sidt` instruction, the IDTR for the virtual machine is returned. The Red Pill tests for this discrepancy to detect the usage of VMware.

[Example 17-2](#) shows how Red Pill might be used by malware.

Example 17-2. Red Pill in malware

```
push    ebp
mov     ebp, esp
sub     esp, 454h
push    ebx
push    esi
push    edi
push    8                ; Size
push    0                ; Val
lea     eax, [ebp+Dst]
push    eax              ; Dst
call    _memset
add     esp, 0Ch
lea     eax, [ebp+Dst]
1 sidt    fword ptr [eax]
mov     al, [eax+5]
cmp     al, 0FFh
jnz     short loc_401E19
```

The malware issues the `sidt` instruction at **1**, which stores the contents of IDTR into the memory location pointed to by EAX. The IDTR is 6 bytes, and the fifth byte offset contains the start of the base memory address. That fifth byte is compared to 0xFF, the VMware signature. Red Pill succeeds only on a single-processor machine. It won't work consistently against multicore processors because each processor (guest or host) has an IDT assigned to it. Therefore, the result of the `sidt` instruction can vary, and the signature used by Red Pill can be unreliable.

To thwart this technique, run on a multicore processor machine or simply NOP-out the `sldt` instruction.

Using the No Pill Technique

The `sgdt` and `sldt` instruction technique for VMware detection is commonly known as No Pill. Unlike Red Pill, No Pill relies on the fact that the LDT structure is assigned to a processor, not an operating system. And because Windows does not normally use the LDT structure, but VMware provides virtual support for it, the table will differ predictably: The LDT location on the host machine will be zero, and on the virtual machine, it will be nonzero. A simple check for zero against the result of the `sldt` instruction does the trick.

The `sldt` method can be subverted in VMware by disabling acceleration. To do this, select **VM ► Settings ► Processors** and check the **Disable Acceleration** box. No Pill solves this acceleration issue by using the `smsw` instruction if the `sldt` method fails. This method involves inspecting the undocumented high-order bits returned by the `smsw` instruction.

Querying the I/O Communication Port

Perhaps the most popular anti-VMware technique currently in use is that of querying the I/O communication port. This technique is frequently encountered in worms and bots, such as the Storm worm and Phatbot.

VMware uses virtual I/O ports for communication between the virtual machine and the host operating system to support functionality like copy and paste between the two systems. The port can be queried and compared with a magic number to identify the use of VMware.

The success of this technique depends on the x86 `in` instruction, which copies data from the I/O port specified by the source operand to a memory location specified by the destination operand. VMware monitors the use of the `in` instruction and captures the I/O destined for the communication channel port 0x5668 (VX). Therefore, the second operand needs to be loaded with VX in order to check for VMware, which happens only when the EAX register is loaded with the magic number 0x564D5868 (VMXh). ECX must be loaded with a value corresponding to the action you wish to perform on the port. The value 0xA means “get VMware version type,” and 0x14 means “get the memory size.” Both can be used to detect VMware, but 0xA is more popular because it may determine the VMware version.

Phatbot, also known as Agobot, is a botnet that is simple to use. One of its features is its built-in support of the I/O communication port technique, as shown in [Example 17-3](#).

Example 17-3. Phatbot’s VMware detection

```
004014FA      push    eax
004014FB      push    ebx
004014FC      push    ecx
004014FD      push    edx
004014FE      mov     eax, 'VMXh' ❶
00401503      mov     ebx, [ebp+var_1C]
```



```

00401506      mov     ecx, 0xA
00401509      mov     dx, 'VX' ❷
0040150E      in      eax, dx
0040150F      mov     [ebp+var_24], eax
00401512      mov     [ebp+var_1C], ebx
00401515      mov     [ebp+var_20], ecx
00401518      mov     [ebp+var_28], edx
...
0040153E      mov     eax, [ebp+var_1C]
00401541      cmp     eax, 'VMXh' ❸
00401546      jnz     short loc_40155C

```

The malware first loads the magic number 0x564D5868 (VMXh) into the EAX register at ❶. Next, it loads the local variable `var_1c` into EBX, a memory address that will return any reply from VMware. ECX is loaded with the value 0xA to get the VMware version type. At ❷, 0x5668 (VX) is loaded into DX, to be used in the following `in` instruction to specify the VMware I/O communication port.

Upon execution, the `in` instruction is trapped by the virtual machine and emulated to execute it. The `in` instruction uses parameters of EAX (magic value), ECX (operation), and EBX (return information). If the magic value matches VMXh and the code is running in a virtual machine, the virtual machine monitor will echo that back in the memory location specified by the EBX register.

The check at ❸ determines whether the code is being run in a virtual machine. Since the get version type option is selected, the ECX register will contain the type of VMware (1=Express, 2=ESX, 3=GSX, and 4=Workstation).

The easiest way to overcome this technique is to NOP-out the `in` instruction or to patch the conditional jump to allow it regardless of the outcome of the comparison.

Using the `str` Instruction

The `str` instruction retrieves the segment selector from the task register, which points to the task state segment (TSS) of the currently executing task. Malware authors can use the `str` instruction to detect the presence of a virtual machine, since the values returned by the instruction may differ on the virtual machine versus a native system. (This technique does not work on multiprocessor hardware.)

[Figure 17-2](#) shows the `str` instruction at 0x401224 in malware known as *SNG.exe*. This loads the TSS into the 4 bytes: `var_1` through `var_4`, as labeled by IDA Pro. Two comparisons are made at 0x40125A and 0x401262 to determine if VMware is detected.

Anti-VM x86 Instructions

We've just reviewed the most common instructions used by malware to employ anti-VM techniques. These instructions are as follows:

- `sidt`

- `sgdt`
- `sldt`
- `smsw`
- `str`
- `in` (with the second operand set to `VX`)
- `cpuid`

Malware will not typically run these instructions unless it is performing VMware detection, and avoiding this detection can be as easy as patching the binary to avoid calling these instructions. These instructions are basically useless if executed in user mode, so if you see them, they're likely part of anti-VMware code. VMware describes roughly 20 instructions as "not virtualizable," of which the preceding are the most commonly used by malware.

Highlighting Anti-VM in IDA Pro

You can search for the instructions listed in the previous section in IDA Pro using the IDAPython script shown in [Example 17-4](#). This script looks for the instructions, highlights any in red, and prints the total number of anti-VM instructions found in IDA's output window.

[Figure 17-2](#) shows a partial result of running this script against *SNG.exe* with one location (`str` at 0x401224) highlighted by the bar. Examining the highlighted code in IDA Pro will allow you to quickly see if the instruction found is involved in an anti-VM technique. Further investigation shows that the `str` instruction is being used to detect VMware.

```

00401210 sub_401210 proc near ; CODE XREF: _main+39↓p
00401210
00401210 var_4= byte ptr -4
00401210 var_3= byte ptr -3
00401210 var_2= byte ptr -2
00401210 var_1= byte ptr -1
00401210
00401210     push ebp
00401211     mov ebp, esp
00401213     push ecx
00401214     mov [ebp+var_4], 0
00401218     mov [ebp+var_3], 0
0040121C     mov [ebp+var_2], 0
00401220     mov [ebp+var_1], 0
00401224     str word ptr [ebp+var_4]
00401228     push offset aTest4Str ; "\n[+] Test 4: STR\n"
0040122D     call printf
00401232     add esp, 4
00401235     movzx eax, [ebp+var_1]
00401239     push eax
0040123A     movzx ecx, [ebp+var_2]
0040123E     push ecx
0040123F     movzx edx, [ebp+var_3]
00401243     push edx
00401244     movzx eax, [ebp+var_4]
00401248     push eax
00401249     push offset aStrBase0x02x02 ; "STR base: 0x%02x%02x%02x%02x\n"
0040124E     call printf
00401253     add esp, 14h
00401256     movzx ecx, [ebp+var_4]
0040125A     test ecx, ecx
0040125C     jnz short loc_401276
0040125E     movzx edx, [ebp+var_3]
00401262     cmp edx, 40h
00401265     jnz short loc_401276
00401267     push offset aResultVmware_2 ; "Result : VMware detected\n\n"
0040126C     call printf
00401271     add esp, 4
00401274     jmp short loc_401283
00401276 ; -----
00401276
00401276 loc_401276: ; CODE XREF: sub_401210+4C↑j sub_401210+55↑j
00401276     push offset aResultNative_2 ; "Result : Native OS\n\n"
0040127B     call printf
00401280     add esp, 4
00401283
00401283 loc_401283: ; CODE XREF: sub_401210+64↑j
00401283     mov esp, ebp
00401285     pop ebp
00401286     retn

```

Figure 17-2. The str anti-VM technique in SNG.exe

Example 17-4. IDA Pro script to find anti-VM instructions

```

from idautils import *
from idc import *

heads = Heads(SegStart(ScreenEA()), SegEnd(ScreenEA()))
antiVM = []
for i in heads:
    if (GetMnem(i) == "sidt" or GetMnem(i) == "sgdt" or GetMnem(i) == "sldt" or
        GetMnem(i) == "smsw" or GetMnem(i) == "str" or GetMnem(i) == "in" or
        GetMnem(i) == "cpuid"):
        antiVM.append(i)

```

```
print "Number of potential Anti-VM instructions: %d" % (len(antiVM))
for i in antiVM:
    SetColor(i, CIC_ITEM, 0x0000ff)
    Message("Anti-VM: %08x\n" % i)
```

Using ScoopyNG

ScoopyNG (<http://www.trapkit.de/>) is a free VMware detection tool that implements seven different checks for a virtual machine, as follows:

- The first three checks look for the `sidt`, `sgdt`, and `sldt` (Red Pill and No Pill) instructions.
- The fourth check looks for `str`.
- The fifth and sixth use the backdoor I/O port `0xa` and `0x14` options, respectively.
- The seventh check relies on a bug in older VMware versions running in emulation mode.

For a disassembled version of ScoopyNG's fourth check, see [Figure 17-2](#).

Tweaking Settings

We have discussed a number of ways to thwart VMware detection throughout this chapter, including patching code, removing VMware Tools, changing VMware settings, and using a multiprocessor machine.

There are also a number of undocumented features in VMware that can help mitigate anti-VMware techniques. For example, placing the options in [Example 17-5](#) into the virtual machine's `.vmx` file will make the virtual machine less detectable.

Example 17-5. VMware's .vmx file undocumented options used to thwart anti-VM techniques

```
isolation.tools.getPtrLocation.disable = "TRUE"
isolation.tools.setPtrLocation.disable = "TRUE"
isolation.tools.setVersion.disable = "TRUE"
isolation.tools.getVersion.disable = "TRUE"
monitor_control.disable_directexec = "TRUE"
monitor_control.disable_chksimd = "TRUE"
monitor_control.disable_ntreloc = "TRUE"
monitor_control.disable_selfmod = "TRUE"
monitor_control.disable_reloc = "TRUE"
monitor_control.disable_btinout = "TRUE"
monitor_control.disable_btmemspace = "TRUE"
monitor_control.disable_btpriv = "TRUE"
monitor_control.disable_btseg = "TRUE"
```

The `directexec` parameter causes user-mode code to be emulated, instead of being run directly on the CPU, thus thwarting certain anti-VM techniques. The first four settings are used

by VMware backdoor commands so that VMware Tools running in the guest cannot get information about the host.

These changes will protect against all of ScoopyNG's checks, other than the sixth, when running on a multiprocessor machine. However, we do not recommend using these settings in VMware, because they disable the usefulness of VMware Tools and they may have serious negative effects on the performance of your virtual machines. Add these options only after you've exhausted all other techniques. These techniques have been mentioned for completeness, but modifying a *.vmx* file to try to catch ten of the potentially hundreds of ways that VMware might be detected can be a bit of a wild-goose chase.

Escaping the Virtual Machine

VMware has its vulnerabilities, which can be exploited to crash the host operating system or even run code in it.

Many publicized vulnerabilities are found in VMware's shared folders feature or in tools that exploit the drag-and-drop functionality of VMware Tools. One well-publicized vulnerability uses shared folders to allow a guest to write to any file on the host operating system in order to modify or compromise the host operating system. Although this particular technique doesn't work with the current version of VMware, several different flaws have been discovered in the shared folders feature. Disable shared folders in the virtual machine settings to prevent this type of attack.

Another well-publicized vulnerability was found in the virtual machine display function in VMware. An exploit for this vulnerability is known as Cloudburst, and it is publicly available as part of the Canvas penetration-testing tool (this vulnerability has also been patched by VMware).

Certain publicly available tools assist in exploiting VMware once the host has been infected, including VMchat, VMcat, VMftp, VMdrag-n-hack, and VMdrag-n-splloit. These tools are of little use until you have escaped the virtual machine, and you shouldn't need to worry about them if malware is being run in the virtual machine.

Conclusion

This chapter introduced the most popular anti-VMware techniques. Because malware authors use these techniques to slow down analysis, it's important to be able to recognize them. We have explained these techniques in detail so that you can find them in disassembly or debugging, and we've explored ways to overcome them without needing to modify malware at the disassembly level.

When performing basic dynamic analysis, you should always use a virtual machine. However, if your subject malware doesn't seem to run, consider trying another virtual machine with VMware Tools uninstalled before debugging or disassembling the malware in search of virtual machine detection. You might also run your subject malware in a different virtual environment (like VirtualBox or Parallels) or even on a physical machine.

As with anti-debugging techniques, anti-VM techniques can be spotted using common sense while slowly debugging a process. For example, if you see code terminating prematurely at a

conditional jump, it may be doing so as a result of an anti-VM technique. As always, be aware of these types of issues and look ahead in the code to determine what action to take.

<https://learning.oreilly.com/library/view/practical-malware-analysis/9781593272906/ch18.html>

CODE OBFUSCATION

<https://www.intertrust.com/blog/top-seven-code-obfuscation-techniques-for-code-protection/>

Code obfuscation uses a variety of techniques to make source code confusing and unreadable. Knowing how to obfuscate code may help mitigate the risk from decompilers and frustrate hackers whose intent is to reverse engineering a program, since the decompiled code is rendered unintelligible.

Code obfuscation techniques

1. Data Transformation

<https://www.sciencedirect.com/science/article/pii/S1877050915032780>

One of the most important elements of code obfuscation is transforming data processed by the program into another form, which has a minimal effect on the performance of the code, but makes it hard for hackers to break down or reverse engineer it.

Examples:

1)

By data transformation:

Program-1 (Original Code)	Program-1 (Obfuscated Code)
<pre>.model small .code Mov DH, 65 Mov AH, 2 Mov DL, 75 Int 21h Mov AH, 4CH Int 21h end</pre>	<pre>.model small .code Mov DH, 65 Mov AH, 2 db 10110010 b Dec BX Int 21h Mov AH, 4CH Int 21h end</pre>

In the obfuscated code, the machine code is written for 'Mov DL'. The binary form of the number 75 is 01001011. For decrementing the content a register, the instruction is '01001reg' and the register BX represented as '011'. So the binary number can be replaced as the assembly code 'Dec BX'.

2)

vi) By combining binary and decimal numbers with Assembly code instructions:

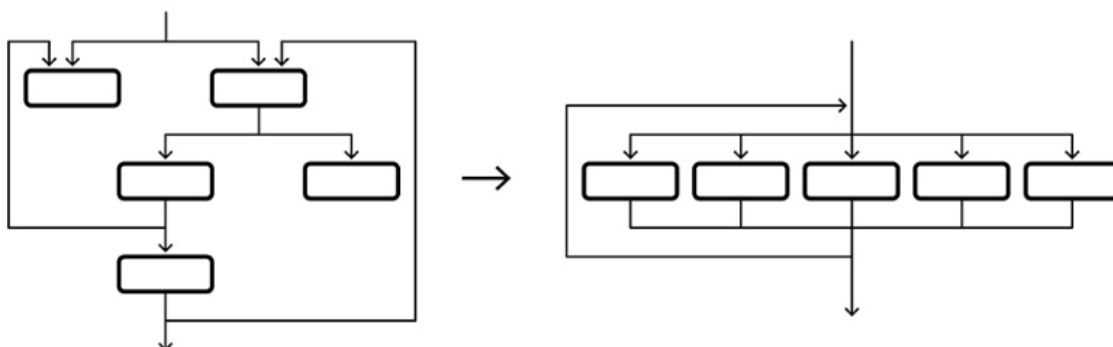
Program-6 (Original Code)	Program-6 (Obfuscated Code)
<pre> .model small .code Mov BL, 80 Mov BH, 85 Mov AH, 2 Mov BL, 178 Mov BH, 98 Mov DL, BH Int 21h Mov DL, BL Int 21h Mov AH, 76 Int 21h end </pre>	<pre> .model small .code Mov BL, 80 Mov BH, 85 Mov AH, 2 <i>db 10110011b</i> <i>Mov DL, 183</i> <i>db 98d</i> Mov DL, BH Int 21h Mov DL, BL Int 21h Mov AH, 76 Int 21h end </pre>

In the original program, according to the instruction 'Mov BL,178', the binary code of 'Mov BL' is 10110011, which is mentioned in the obfuscated code. The binary form of 178 is 10110010, which can be elaborated as '1011- 0-010' and it is 'Mov DL' as the assembly code. So, there is a binary instruction, then an assembly code instruction and after that a decimal number is there in the obfuscated program, which is similar to the italic assembly code instructions of the original program.

2. Code Flow Obfuscation

By changing the control flow of the code, the orientation of the code is changed. This means that although the final results are the same, it takes a lot longer to understand why the code takes a certain direction or where it is going.

Control flow obfuscation can be performed by altering the order of program execution statements, changing the control graph by inserting arbitrary jump instructions, and converting tree-like conditional constructs into flat switch statements as shown in the following diagram.



Control aggregation – change the group of program statements; for instance, using the inline methods instead of method calls. (inline methods: To replace a function call with a copy of the function's code during compilation.)

Control ordering – change the order in which the program statements are executed;

Control computation – change the control flow in the program as:

- Inserting dead code in the code execution flow to hide the true control flow; the code complexity is increased;
- Inserting uncontrolled jump instructions to change the control graph of the program;
- Inserting alternative structures having a controlled evaluation of the condition to broke a statement block in two statement sub-blocks at least.

3. Address Obfuscation

This technique alters the addresses of program data and code to create unpredictability and make it more difficult to exploit. When an application is built, the obfuscation algorithm randomizes the absolute locations of some code and data in memory, and the relative distances between different data items. This not only reduces the likelihood of successful attacks, it also means that even if a hacker is successful on one application or device, they will not be able to replicate it on others, reducing the benefit of reverse engineering a program.

Methods to perform it

1) Randomize the base addresses of memory regions. By changing the base addresses of code and data segments by a random amount, we can alter the absolute locations of data resident in each segment. If the randomization is over a large range, say, between 1 and 100 million, the virtual addresses of code and data objects become highly unpredictable

Randomize the base address of the stack. This transformation has the effect of randomizing all the addresses on the stack. A classical stack-smashing attack requires the return address on the stack to be set to point to the beginning of a stack-resident buffer into which the attacker has injected his/her code. This becomes very difficult when the attacker cannot predict the address of such a buffer due to randomization of stack addresses. . Stack-address randomization can be implemented by subtracting a large random value from the stack pointer at the beginning of the program execution.

2) Randomize the base address of the heap. This transformation randomizes the absolute locations of data in the heap, and can be performed by allocating a large block of random size from the heap

Randomize the starting address of dynamically linked libraries. This transformation has the effect of randomizing the location of all code and static data associated with dynamic libraries. This will prevent existing code attacks (also called return-into-libc attacks), where the attack causes a control flow transfer to a location within the library that is chosen by the attacker.

3) Permute the order of variables/routines. Attacks that exploit relative distances between objects, such as attacks that overflow past the end of a buffer to overwrite adjacent data that is subsequently used in a security-critical operation, can be rendered difficult by a random permutation of the order in which the variables appear.

There are three possible rearrangement transformations:

1. Permute the order of local variables in a stack frame
2. Permute the order of static variables
3. Permute the order of routines in shared libraries or the routines in the executable

4) Introduce random gaps between objects. For some objects, it is not possible to rearrange their relative order.

a) Introduce random padding into stack frames. The primary purpose of this transformation is to randomize the distances between variables stored in different stack frames, which makes it difficult to craft attacks that exploit relative distances between stack resident data. The size of the padding should be relatively small to avoid a significant increase in memory utilization.

b) Introduce random padding between successive malloc allocation requests.

c) Introduce random padding between variables in the static area.

d) Introduce gaps within routines, and add jump instructions to skip over these gaps.

Effectiveness Address obfuscation is not a fool proof defense against all memory error exploits, but is instead a probabilistic technique which increases the amount of work required before an attack (or sequence of attacks) succeeds. Hence, it is critical to have an estimate of the increase in attacker work load.

4. Regular Renewal of Obfuscated Code

This technique proactively prevents attacks by regularly issuing updates of obfuscated software, frustrating hacker attempts to crack the system. By occasionally replacing existing software with newly obfuscated instances, an attacker is forced to abandon their existing analysis. In the end, the effort of breaking code exceeds the value gained.

5. Obfuscation of Assembly Code Instructions

Transforming and altering the assembly code can render it more difficult to reverse-engineer. One such method is to use overlapping assembly instructions (also known as “jump-in-the-middle” method) that hide code within other code causing a disassembler to produce incorrect output. Assembly code can also be strengthened against penetration through the inclusion of unnecessary control statements and garbage code.

6. Obfuscating Debug Information

Debug information can be used for reverse engineering a program to discover its source code thorough decompiling and recompiling a program’s code. That’s why it’s important to block unauthorized access and debugging. This can be accomplished by changing line numbers and file names in debug data, or removing debug information altogether.

Obfuscation Example:

Obs.c

```
#include<stdio.h>
int main(){
//some floating point numbers for testing
float b[]={1.1431391224375825e+27, 6.6578486920496456e+28, 7.7392930965627798e-19, 3.2512161851627752e-9};
//print all numbers in array b
puts((char*)b);
return 0;
}
```

```
kali@kali:~$ gcc obs.c
kali@kali:~$ ./a.out
Hello World!
```

Each of the floats is 4 bytes. Each character is 1 byte. Therefore a float value cast to char will give us 4 characters.

So when you print one of the values you will get corresponding 4 chars:

```
kali@kali:~$ gcc obs.c
kali@kali:~$ ./a.out
rld!
```

We can say that we have used the data transformation technique to obfuscate this hello world program. By seeing the program it is difficult to determine that it prints out Hello World!

How to obfuscate a code using python

PyArmor is a command line tool used to obfuscate python scripts, bind obfuscated scripts to fixed machine or expire obfuscated scripts.

Install:

```
pip install pyarmor
```

Obfuscate scripts:

```
pyarmor obfuscate name_of_file.py
```

The obfuscated script is stored in a folder called dist inside the top directory.

Example:

example.py

```
5 # In[6]:
6
7 def hello():
8     print('Hello world!')
9
10 def sum2(a, b):
11     return a + b
12
13 def main():
14     hello()
15     print('1 + 1 = %d' % sum2(1, 1))
16
17 main()
18 # In[ ]:
```

Run in command line

Pyarmor obfuscate example.py

\xd6\x29\xb9\xd9\x34\x31\x08\x41\xc8\x99\xdb\xd4\x19\xeb\xeb\x0b\xc1\x02\xa2\x7c\xc8\xd0\xc0\x6c\x59\x42\x17\x5e\xae\x6c\xda\xa2\xdc\x72\xab\x62\xf1\x22\xfd\x62\x13\xad\x2e\x67\xf1\xc9\xe6\x96\xd3\xc3\x09\x10\xfe\x4d\xbe\x8b\xf2\xac\xdb\x25\xee\xbe\xeb\x78\x6e\x65\x76\xd6\xac\xa0\xc7\x9a\x9e\x2f\x9b\x53\xdb\x3c\x71\x91\x4c\x9c\xf3\x15\x5e\x9a\x86\x53\x00\xf0\xc8\xf6\x79\x9c\xbc\x04\x42\xf3\x7a\x08\x5e\xe6\x28\xe8\xe6\xe8\xbb\xeb\x46\x8a\xe6\x34\x1d\xe1\x96\xbf\x32\x25\x36\xe7\xe9\x28\xfd\xd9\x5f\xa4\x61\xcc\xa2\xc1\xf1\x77\x57\x01\x62\x31\x28\xfa\x8e\xf3\x03\xec\x6e\x89\x38\x7a\x6e\x88\x80\x6d\x9a\x24\x16\x1d\x84\x4b\x80\x96\xa7\x2d\x36\x29\x25\x47\xd4\x8a\x95\x0f\xdb\xd9\xa0\x26\x99\x42\xa5\xc3\xcc\xeb\xbd\x71\x59\x4e\x1c\x1c\xd1\x0f\x1b\x0a\xbc\x52\x73\x66\xe5\xce\xe8\x0a\xee\x98\x20\x18\x17\x3e\x99\xe9\x6e\x9d\x59\x1b\x3b\x4a\x53\x50\x3e\x5c\x0a\x9f\x07\x63\x10\x54\x4b\x11\x0a\x54\x5c\x35\xc8\x8b\x2c\xa4\x4a\xd2\x74\x24\x90\xc7\x71\x39\x4a\x2b\xed\x75\x23\x94\x42\xa1\x40\x4e\x7b\xd6\x27\xc3\x17\x6a\x68\x45\x85\x2c\x1a\xfb\xc2\x85\xb7\x01\x2d\x81\x52\x42\x46\xab\xeb\xfd\x2f\x03\xbb\xea\xcc\xe5\xc6\x70\x60\x5c\x75\xc6\x1d\x0e\x4f\xce\x94\x43\x7a\xd6\x1e\x8e\x4c\xe6\x20\xe8\x72\x27\xa5\x6a\x8b\x70\xc3\x4e\x10\xa4\xd5\xd3\xdf\x6c\x2f\xfe\x54\x42\x46\x23\x97\x22\xa8\x3b\x2c\x97\x12\x26\xa2\x30\xaf\xce\x1a\x61\xdc\xc0\x9f\x0b\x18\xf3\xce\xb2\xbb\x28\x7d\x35\x14\x60\x00\x6d\x77\x87\x97\xdf\x9f\xbb\xf5\x66\x9b\x13\x4c\x85\x71\x57\xfc\x97\xa9\x9c\xf8\x5b\xe6\xc9\xfb\xd1\xc3\xd1\x7d\xa6\xdf\x23\x7e\x66\x27\xe3\xe6\xfc\xeb\x7a\x53\xb5\x80\xcb\xd3', 1)

Obfuscated code gives the same output

```
(base) C:\Users\shcil\Desktop\workdir\Untitled Folder>cd dist
(base) C:\Users\shcil\Desktop\workdir\Untitled Folder\dist>python example.py
Hello world!
1 + 1 = 2
(base) C:\Users\shcil\Desktop\workdir\Untitled Folder\dist>
```

How cyberattackers use code obfuscation?

<https://www.zdnet.com/article/a-question-of-security-what-is-obfuscation-and-how-does-it-work/>

Off-the-shelf malware can be purchased easily by anyone. While some of the most sophisticated forms of malware out there can fetch prices of \$7,000 and more, it is also possible to pick up exploit kits and more for far less, or even for free.

The problem with this so-called "commodity" malware is that antivirus companies are well aware of their existence and so prepare their solutions accordingly with signatures that detect the malware families before they can cause damage.

So, how do threat actors circumvent such protection?

This is known as obfuscation.

The goal of obfuscation is to anonymize cyberattackers, reduce the risk of exposure, and hide malware by changing the overall signature and fingerprint of malicious code -- despite the payload being a known threat.

Signatures very often are hashes, but they can also be some other brief representation of a unique bit of code inside a piece of malware."

Rather than attempt to create a new signature through changing malware itself, obfuscation instead focuses on delivery mechanisms in an attempt to dupe antivirus solutions which rely heavily on signatures.

Obfuscation can include a variety of techniques to hide malware

These techniques include:

- Packers: These software packages will compress malware programs to hide their presence, making original code unreadable.
- Crypters: Crypters may encrypt malware programs, or portions of software, to restrict access to code which could alarm an antivirus product to familiar signatures.
- Dead code insertion: Ineffective, useless code can be added to malware to disguise a program's appearance.
- Instruction changes: Threat actors may alter instruction codes in malware from original samples that end up changing the appearance of the code -- but not the behavior -- as well as change the order and sequence of scripts.
- Exclusive or operation (XOR): This common method of obfuscation hides data so it cannot be read unless trained eyes apply XOR values of 0x55 to code.
- ROT13: This technique is an ASM instruction for "rotate" which substitutes code for random letters.

While some antivirus products search for common obfuscating techniques so that they too may be blacklisted, this practice is not nearly as well established as the blacklisting of malware payload signatures

Threat actors are increasingly using obfuscation techniques in combination with commodity malware.

This trend runs counter to a widely-held assumption in the information security space which holds that highly customized malware paired with zero-day exploits are deserving of the most attention.