8

EVADING SANDBOXES AND DISRUPTING ANALYSIS

In previous chapters, you've learned about several techniques that malware uses to establish context and better understand its current environment. When malware determines that it's running in an analyst's lab or in an otherwise hostile environment, it may take evasive measures, such as delaying its execution, creating decoys, or even actively impeding investigation efforts by interfering with the analyst's tools. This chapter will focus on these and other methods that malware uses to hide from and circumvent analysis tools.

Self-Termination

A simple and effective way in which malware can avoid analysis is *self-termination*. The malware can simply call Windows API functions such as `TerminateProcess` or `ExitProcess` to issue a "kill" command to its own process, like so:

```
is_vm = enumerate_reg_keys(keys)
if (is_vm)
{
    current_process = GetCurrentProcess()
    TerminateProcess(current_process, ...)
}
```

This malware pseudocode first calls its own internal `enumerate_reg_keys` function to enumerate some of the VM-related registry keys discussed in Chapter 4. (The details of the function aren't shown here.) Next, if `is_vm` returns true, the malware requests a handle to its own process (`GetCurrentProcess`) and then terminates itself by calling `TerminateProcess`. The `ExitProcess` function can be used in the same way, with a few trivial differences. Sometimes malware even calls both functions to ensure that it has successfully terminated.

This technique is especially effective against automated sandboxes, which can't monitor the behavior of a malware sample that has terminated itself. However, a sandbox could flag the function itself or detect that the sample terminated itself too soon. This approach can also be effective against a malware analyst interacting with the sample manually, as the analyst will have to walk backward through the code in a debugger or disassembler to determine how and why the malware terminated itself.

When you're analyzing a malware sample that's using this technique, setting a debugger breakpoint on `ExitProcess` and `TerminateProcess` may help you catch the malware before it has a chance to kill itself. This will allow you to inspect the call stack and the code leading up to the process termination, and hopefully to identify what caused it. Keep in mind, however, that these API functions might also be called during a crash, so the malware may not be invoking them directly for evasion purposes.

Delayed Execution

Imagine a typical automated malware analysis sandbox environment. This environment will boot up on demand, detonate a malware sample, monitor the malware's behaviors for a few minutes (depending on how the sandbox is configured), and then shut down. But what if the malware delays its own execution to "time out" the sandbox analysis process? For example, perhaps the malware executes a sleep routine in which it lies dormant for several minutes, outlasting the short life of the sandbox environment. It's not unheard of for advanced malware to delay its execution for hours or even

weeks at a time. This is an effective method of evading sandboxes and frustrating malware analysts' efforts.

Sleep Function Calls

Perhaps the most common form of delayed execution is malware simply invoking the `Sleep` function from the Windows API. `Sleep`, as well as its cousin, `SleepEx`, takes a parameter that represents the sleep time in milliseconds. The following assembly code shows a snippet of a malware sample calling the `Sleep` function:

```
push 493E0h ; 5 minutes
call Sleep
```

In this case, the `493E0h` parameter passed to `Sleep` is the time in hexadecimal, representing 300,000 milliseconds, or 5 minutes.

NOTE *For more information on the* `Sleep` *function and how malware can use it, see Chapter 7.*

To bypass this technique, you could put a breakpoint on `Sleep` and `SleepEx` function calls and then modify the `dwMilliseconds` parameter passed to it. Alternatively, you could `nop` out these `Sleep` instructions or jump over them in a debugger. These aren't always foolproof solutions, however; advanced malware may calculate the system time before and after the calls to `Sleep` to verify that the `Sleep` function executed correctly! Lastly, many modern sandboxes can intercept calls to `Sleep` and modify them, dramatically lowering the sample's total sleep time.

Timeouts

Malware can take a less traditional route to delay its execution by using Windows utilities, such as *ping.exe*, to cause a *timeout*. This approach often works better than the sleep method, since it's more difficult for sandboxes to interfere with. Another advantage is that it may confuse the analysis process, as the malware analyst must figure out why the malware sample is invoking a certain application.

In the following code snippet, a malware sample is executing *ping.exe* to ping Google 1,000 times. Depending on the network connection speed, this could create a long delay or even cause the sandbox to time out and stop analysis:

```
push eax ; "ping.exe google.com -n 1000"
push 0;
call CreateProcessA
```

Malware can also call the *timeout.exe* Windows tool, which is typically used in batch scripts to pause command processing, in order to delay execution. Be on the lookout for malware invoking these types of tools. Use code

analysis and debugging to understand why the malware might be executing this behavior.

Time and Logic Bombs

In a *time bomb*, the malware sets a specific time, such as a certain date or time of day, for when it will execute. For example, a malware sample may contain embedded code that executes only at 9 AM every morning, once every Saturday, or on December 26, 2024, at 5:55 PM. Unless the sandbox or malware analyst manually sets the date or time to trick the malware into running, the sample won't execute its malicious code.

Similar to a time bomb, in a *logic bomb,* the malware executes after a specific event (such as a certain file deletion or database transaction) has occurred on the host. Logic bombs may be even more effective than time bombs, since they can be very specific to the malware's operating environment.

The following simplified pseudocode demonstrates a time bomb technique in which the malware sample gets the current system date and compares it to a hardcoded date (in this case, 2024):

```
--snip--
GetSystemTime(&systemTime)

if (systemTime.wYear <=  '2024') {
    KillSelf()
}
```

If the malware determines that the current date is 2024 or earlier, it will fail to execute.

Sometimes a sandbox can identify whether malware is using these techniques, but they often fly under the radar. The best way to identify time and logic bombs is code analysis. Inspecting the malware sample in a disassembler or debugger may uncover the time, date, or logic that the malware is looking for. Once you identify this, you can simply set your analysis system time to match it or try to re-create the logic. Alternatively, you could modify the malware's code in a disassembler or debugger to bypass these checks.

It's important to note that, besides being used for evasion, time bomb techniques are used to control the malware's spread. Malware may be programmed to *not* execute after a specific date or time in order to better control it or otherwise limit its lifetime.

Dummy Code and Infinite Loops

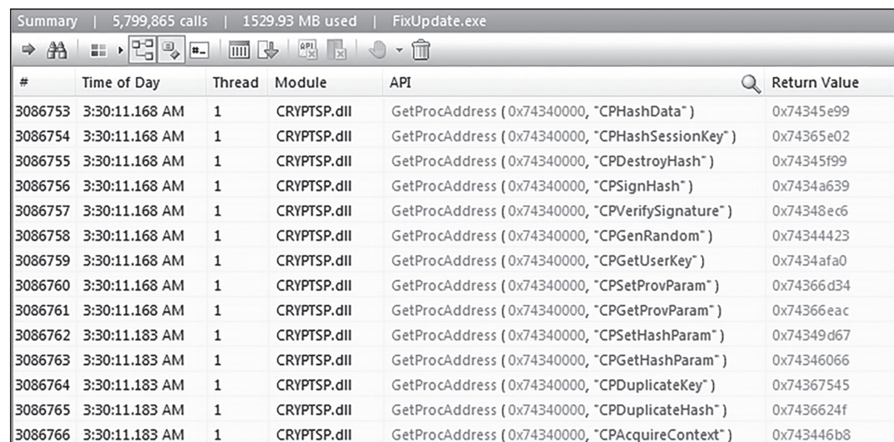
Some malware authors introduce *dummy code* into their malware that loops, possibly infinitely, calling CPU-intensive functions or functions that serve no purpose other than to time out the analysis. The dummy code usually

runs once the malware has detected a sandbox or VM environment. The following assembly code example shows what that might look like:

```
loop:
inc ecx
cmp ecx,ecx
je loop
```

In this basic `for` loop, the value of `ecx` is incremented by 1 and then compared to itself. If it's equal to itself (hint: it will be), the loop repeats. This simple code will stall the malware's execution indefinitely, or at least until the sandbox terminates or the malware analyst becomes frustrated and kills the process.

Similarly, some malware repeatedly calls Windows API functions to stall analysis. For example, it might call `RegEnumKey` to enumerate the host's entire registry, which will take a significant amount of time. Alternatively, the malware sample might repeatedly call `LoadLibrary` on nonexistent libraries. While writing this book, I analyzed a Dridex banking trojan sample that executes `GetProcAddress` over *five million times* to resolve addresses of functions it never uses (see Figure 8-1). This stalls analysis, uses up valuable sandbox memory and CPU resources, and sometimes results in a crash.



Summary | 5,799,865 calls | 1529.93 MB used | FixUpdate.exe

#	Time of Day	Thread	Module	API	Return Value
3086753	3:30:11.168 AM	1	CRYPTSP.dll	GetProcAddress (0x74340000, "CPHashData")	0x74345e99
3086754	3:30:11.168 AM	1	CRYPTSP.dll	GetProcAddress (0x74340000, "CPHashSessionKey")	0x74365e02
3086755	3:30:11.168 AM	1	CRYPTSP.dll	GetProcAddress (0x74340000, "CPDestroyHash")	0x74345f99
3086756	3:30:11.168 AM	1	CRYPTSP.dll	GetProcAddress (0x74340000, "CPSignHash")	0x7434a639
3086757	3:30:11.168 AM	1	CRYPTSP.dll	GetProcAddress (0x74340000, "CPVerifySignature")	0x74348ec6
3086758	3:30:11.168 AM	1	CRYPTSP.dll	GetProcAddress (0x74340000, "CPGenRandom")	0x74344423
3086759	3:30:11.168 AM	1	CRYPTSP.dll	GetProcAddress (0x74340000, "CPGetUserKey")	0x7434afa0
3086760	3:30:11.168 AM	1	CRYPTSP.dll	GetProcAddress (0x74340000, "CPSetProvParam")	0x74366d34
3086761	3:30:11.168 AM	1	CRYPTSP.dll	GetProcAddress (0x74340000, "CPGetProvParam")	0x74366eac
3086762	3:30:11.183 AM	1	CRYPTSP.dll	GetProcAddress (0x74340000, "CPSetHashParam")	0x74349d67
3086763	3:30:11.183 AM	1	CRYPTSP.dll	GetProcAddress (0x74340000, "CPGetHashParam")	0x74346066
3086764	3:30:11.183 AM	1	CRYPTSP.dll	GetProcAddress (0x74340000, "CPDuplicateKey")	0x74367545
3086765	3:30:11.183 AM	1	CRYPTSP.dll	GetProcAddress (0x74340000, "CPDuplicateHash")	0x7436624f
3086766	3:30:11.183 AM	1	CRYPTSP.dll	GetProcAddress (0x74340000, "CPAcquireContext")	0x743446b8

Figure 8-1: Delaying analysis by repeatedly executing `GetProcAddress`

Dridex has also been known to execute `OutputDebugString` in an infinite loop, which has the same effect as the `GetProcAddress` approach. The `OutputDebugString` function will be discussed in more detail in Chapter 10.

> **PERSISTENCE MECHANISMS FOR DELAYING EXECUTION**
>
> Oftentimes, malware attempts to establish persistence on the victim host in order to "survive" system shutdowns and reboots, but persistence is also a great delayed-execution tactic. Malware can configure a persistence mechanism in the form of a scheduled task on the victim host to execute its payload only after a certain event or amount of time. Chapter 15 will discuss several ways to achieve persistence.

Forcing Reboots and Logouts

Forcing a system shutdown, reboot, or logout can be an effective method of evasion, especially in sandboxes. It will promptly halt all analysis efforts, at least until the host is back up. Most modern sandboxes are able to deal with this, however, and if the sandbox senses a shutdown or logout has been issued, it will simply continue analysis after the machine is back up. But this can still negatively affect the malware analysis process. In the case of reboots, for example, artifacts that were once in memory may now be destroyed.

Malware can force a reboot or shutdown by invoking functions such as `InitiateShutdown`, `InitiateSystemShutdown`, and `InitiateSystemShutdownEx`. All three functions operate similarly and take a few key arguments, such as an option specifying whether to shut down or reboot the host, as well as a *timeout* value representing the duration between the function call and the reboot or shutdown. Another API function that malware might use is `ExitWindows` (or its sibling, `ExitWindowsEx`), which adds the option to log out the user, rather than simply rebooting or shutting down the host. Finally, the system can also be shut down using WMI, PowerShell, or the built-in Windows *shutdown.exe* tool.

Malware often uses this technique after it has established persistence, at which point it forces a reboot and then runs its actual payload. In this way, it successfully evades certain automated analysis sandboxes and confuses (or at least annoys) malware analysts trying to investigate the sample.

Decoys and Noise

Some malware authors take advantage of the fact that sandboxes operate in a predictable way. For example, sandboxes must capture a large amount of data to understand and assess a malware sample's behaviors, and malware can exploit this by generating lots of *noisy* or *decoy* data that can quickly overwhelm a sandbox or hamper analysis. This section covers a few different ways in which malware can do this.

API Hammering

When a sandbox detonates a malware sample, it logs the malware's behaviors and function calls. *API hammering* involves calling the same function many times (in some cases, hundreds of thousands of times) to quickly fill up the sandbox logs and flood the analysis environment with useless data. As a result, the sandbox may be unable to successfully analyze the sample due to too much noise and a full log. Furthermore, malware samples using API-hammering techniques will take a lot longer to fully execute in a sandbox since its logging behaviors introduce extra overhead. If the same sample were executed on a normal end-user system, it would execute much more quickly.

Nearly any Windows API function can be abused for this purpose. Two I've seen are `printf` (a C function that prints characters to the calling application) and `TlsGetValue`. The malware sample shown in Figure 8-2 called the `TlsGetValue` function over 30,000 times in a row!

260	9:15:24.054 PM	1	TlsGetValue (12)
261	9:15:24.054 PM	1	TlsGetValue (12)
262	9:15:24.054 PM	1	TlsGetValue (12)
263	9:15:24.054 PM	1	TlsGetValue (11)
264	9:15:24.054 PM	1	TlsGetValue (11)
265	9:15:24.054 PM	1	TlsGetValue (12)
266	9:15:24.054 PM	1	TlsGetValue (12)
267	9:15:24.054 PM	1	TlsGetValue (11)
268	9:15:24.054 PM	1	TlsGetValue (12)
269	9:15:24.054 PM	1	TlsGetValue (12)
270	9:15:24.054 PM	1	TlsGetValue (11)
271	9:15:24.054 PM	1	TlsGetValue (12)
272	9:15:24.054 PM	1	TlsGetValue (12)
273	9:15:24.054 PM	1	TlsGetValue (12)

Figure 8-2: Malware using API hammering by calling `TlsGetValue` multiple times

The malware families Nymaim and Trickbot both employ API-hammering techniques, as described in blog posts from Joe Security (*https://www.joesecurity.org*). At least one Nymaim variant makes over *half a million* benign Windows API function calls if the sample detects that it's running in a VM or sandbox environment! As you can imagine, this generates an enormous amount of data in a sandbox log. Some sandboxes, unable to handle that volume of data, would likely terminate the analysis early.

Many modern sandboxes can detect API hammering, however, and will flag such behavior as suspicious or even stop logging the questionable function altogether. A sandbox might also modify the running malware sample's

behavior or take other actions to prevent API hammering from interfering with analysis. But if left undetected, API hammering can severely impact the sandbox's ability to assess the malware.

Unnecessary Process Spawning

Like API hammering, unnecessary process spawning is a technique used to overwhelm sandboxes and malware analysts. The malware sample shown in Figure 8-3 spawns several hundred processes, all named *<xxxx>.tmp*, to hide its true activity.

Time of Day	Process Name	PID	Operation
2:45:01.3025261 PM	WinWord.exe	2100	Process Create
2:45:01.4770031 PM	4CC3.tmp	2068	Process Create
2:45:01.6280210 PM	4D50.tmp	1904	Process Create
2:45:01.7751459 PM	4DFC.tmp	2116	Process Create
2:45:01.8930741 PM	4E98.tmp	1936	Process Create
2:45:02.0401059 PM	4F15.tmp	2132	Process Create
2:45:02.1631946 PM	4FA1.tmp	2084	Process Create
2:45:02.3187729 PM	501E.tmp	1920	Process Create
2:45:02.4344296 PM	50BB.tmp	284	Process Create
2:45:02.5940949 PM	5138.tmp	2172	Process Create
2:45:02.7113267 PM	51D4.tmp	892	Process Create
2:45:02.8990254 PM	5251.tmp	1124	Process Create
2:45:03.0302509 PM	52ED.tmp	2176	Process Create
2:45:03.1843058 PM	5389.tmp	1784	Process Create
2:45:03.3156597 PM	5426.tmp	1980	Process Create
2:45:03.4443518 PM	54A3.tmp	1976	Process Create
2:45:03.5631673 PM	5520.tmp	2216	Process Create
2:45:03.6857043 PM	559D.tmp	2404	Process Create
2:45:03.8027174 PM	561A.tmp	2440	Process Create
2:45:04.3563369 PM	5687.tmp	2468	Process Create
2:45:04.5232190 PM	588B.tmp	2500	Process Create
2:45:04.8129748 PM	5946.tmp	2496	Process Create
2:45:04.9622704 PM	5A6F.tmp	2476	Process Create
2:45:05.2666865 PM	5B0B.tmp	2516	Process Create

Figure 8-3: Malware spawning a large number of "dummy" processes

Because of the staggering number of processes the malware creates, it's difficult for the analyst to identify which ones are worth investigating. Sandboxes may also be overwhelmed by all the data.

Decoy Network Communication

Some malware variants send fake or decoy network traffic to attempt to conceal the real malicious traffic. One malware family, Formbook, is well known for using this technique. Formbook connects to a randomized list of several decoy web addresses and one actual *command and control (C2)* address, which can confuse analysts and sandboxes. In some cases, these decoy addresses are real domains that can lead the malware analyst down

the wrong paths during the investigation. Figure 8-4 shows Formbook connecting to multiple decoy C2 addresses using normal HTTP `GET` requests.

```
08:53:07 AM [     HTTPListener80]   GET /sr2/?oN64tl4H=fc9xvltLhtrJIvGep5xcPuctYFycPA/rUy/N9oGuboQjhmixEWaiXiLqknym/EQdiU7MVg==&lBZDkH=xRK4RHWHT HTTP/1.1
08:53:07 AM [     HTTPListener80]   Host: www.typicalmaneuver.com
08:53:07 AM [     HTTPListener80]   Connection: close
08:53:07 AM [     HTTPListener80]
08:53:27 AM [     HTTPListener80]   GET /sr2/?oN64tl4H=76dXLJulqVo5KhXdF2rppVJLlykVgBEJNtIje33rPXbI26N0y35eaxzMKSC2qDZU9B+e+w==&lBZDkH=xRK4RHWHT HTTP/1.1
08:53:27 AM [     HTTPListener80]   Host: www.xingdonggan.com
08:53:27 AM [     HTTPListener80]   Connection: close
08:53:27 AM [     HTTPListener80]
08:53:46 AM [     HTTPListener80]   GET /sr2/?oN64tl4H=zgIrB9LpvHDE5AwSmeeDg6W8Zo7Tob8Jq1S0rcdaLNPO+prie6HzCjEi+kxPG3aADRAprQ==&lBZDkH=xRK4RHWHT HTTP/1.1
08:53:46 AM [     HTTPListener80]   Host: www.hartlandpoloclassic.net
08:53:46 AM [     HTTPListener80]   Connection: close
08:53:46 AM [     HTTPListener80]
08:54:06 AM [     HTTPListener80]   GET /sr2/?oN64tl4H=XdKzoqGQYJoINgcNaC8bxMa5jpBQnQUVqTdFcArOKi6f2ehdewn381e1OS136xYkvmqeuA==&lBZDkH=xRK4RHWHT HTTP/1.1
08:54:06 AM [     HTTPListener80]   Host: www.thecoachhousedouglas.com
08:54:06 AM [     HTTPListener80]   Connection: close
08:54:06 AM [     HTTPListener80]
08:54:27 AM [     HTTPListener80]   GET /sr2/?oN64tl4H=TamPOdRkAwn7lIGFyAJhulwQndSGrSy70Cj92KWYtFX2Wv1rE96dirp62bNRfUhGNAJm1w==&lBZDkH=xRK4RHWHT HTTP/1.1
08:54:27 AM [     HTTPListener80]   Host: www.tongtiejiche.com
08:54:27 AM [     HTTPListener80]   Connection: close
08:54:27 AM [     HTTPListener80]
08:54:47 AM [     HTTPListener80]   GET /sr2/?oN64tl4H=dTv3W1r0kuyf9g/ya/V0qbx1zVhdXJo8YsXsM16D1G4CsmaUu+P6NoqlpUmMzgubv72EVQ==&lBZDkH=xRK4RHWHT HTTP/1.1
08:54:47 AM [     HTTPListener80]   Host: www.startupworldcup.biz
08:54:47 AM [     HTTPListener80]   Connection: close
08:54:47 AM [     HTTPListener80]
08:55:13 AM [     HTTPListener80]   GET /sr2/?oN64tl4H=RP/YDtRpIq5sIhqKS4rDEgcdmaenKXnVUnQcoVx2dirZMYfnZyJRXdPrdqt54A+RLoqtow==&lBZDkH=xRK4RHWHT HTTP/1.1
08:55:13 AM [     HTTPListener80]   Host: www.tidalroyaltycorp.com
08:55:13 AM [     HTTPListener80]   Connection: close
08:55:13 AM [     HTTPListener80]
08:55:33 AM [     HTTPListener80]   GET /sr2/?oN64tl4H=2sdeVJJYpaWvmCQnoJIdsciD9GWstwHcf6eT4StdCpzYOvj/CMGXXohjdHMLYcdZ+WheNw==&lBZDkH=xRK4RHWHT HTTP/1.1
08:55:33 AM [     HTTPListener80]   Host: www.spatren.com
08:55:33 AM [     HTTPListener80]   Connection: close
```

Figure 8-4: Formbook connecting to decoy C2 addresses

As you can see, all of the traffic looks almost identical, but only one of these connections is for the real C2 server.

NOTE *You can download the Formbook malware from VirusTotal or MalShare using the following file hash:*

```
SHA256: 08ef1473879e6e8197f1eadfe3e51a9dbdc9c892e442b57a3186a64ecc9d1e41
```

Anti-hooking

Many malware analysis sandboxes and tools use *API hooking*, or simply *hooking*, to analyze malware behavior. This involves injecting a piece of code, called a *hook*, into the malware's memory space. The hook then intercepts API function calls, redirects them to a different function or modifies their behavior, and passes them on to the original function. This hook is often a module, typically in the form of a DLL, that then monitors the sample as it runs (see Figure 8-5).

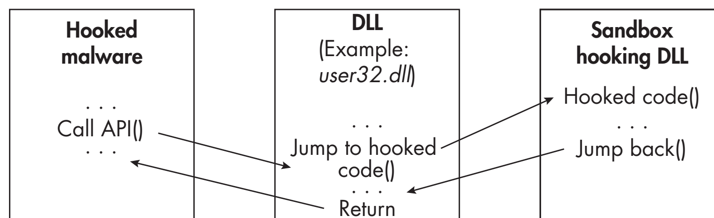


Figure 8-5: A sandbox hooking a running malware process

In this example, a sandbox has hooked the running malware's process (hooked malware) via DLL injection (the sandbox hooking DLL). The sandbox modifies the first few bytes of the function it's hooking (inside *user32.dll*) and inserts a jump (`jmp`) instruction. Now any calls to the function in the *user32.dll* library will jump to the hook code in the sandbox hooking DLL. The installed hook allows the sandbox to intercept and monitor function calls and potentially modify the function call parameters or return values.

To implement a hook, the sandbox agent inserts a jump statement into the beginning of a function it wishes to hook. The following assembly code excerpt shows the first few bytes of the `ReadFile` function after it has been hooked by a sandbox:

```
0x77000000  jmp hook_code
0x77000005  // Start of real ReadFile code
```

In this hooked code, the inserted jump statement will ensure that when the malware calls the `ReadFile` function, the execution flow will transfer to the sandbox hook code (`hook_code`) before executing the real `ReadFile` code. This type of hook is called *inline hooking*. Sandboxes use a technique called *process injection* to inject inline hooks into target processes. We'll discuss injection and various types of hooking in more detail in Chapter 12.

Some analysis tools, such as API Monitor and certain debugger plug-ins, use hooks for similar purposes. One example is the popular tool ScyllaHide, which can be used to circumvent anti-debugging techniques in malware. (Chapter 10 will cover ScyllaHide in greater detail.) In this section, we'll dig deeper into some of the ways in which malware can detect and circumvent hooking and monitoring.

Hook Detection

Before executing, malware will likely try to detect whether it's being hooked by a sandbox or an analysis tool by scanning its own memory for these injected hooking modules. In Chapter 7, you saw how malware can call functions such as `Module32First` and `Module32Next` to enumerate its loaded modules. For hook detection, the malware sample may keep track of which modules it will load, so if it enumerates its loaded modules and notices an anomalous loaded module, it may assume that it's being hooked or otherwise monitored.

Before executing its target function, malware can check whether a sandbox has modified that function's code in an attempt to hook it. In order to accomplish this, the malware invokes either `ReadProcessMemory` or `NtReadVirtualMemory` to read the memory where the suspect function resides, and then it inspects the first few bytes of the function. The malware will be on the lookout for anomalous jump instructions that have been inserted into the beginning of the function in question, a sure sign of hooking, as the following pseudocode illustrates:

```
handle = GetModuleHandle("ntdll.dll")
functionAddress = GetProcAddress(handle, "NtAllocateVirtualMemory")

ReadProcessMemory(GetCurrentProcess(), functionAddress, buffer, bufferSize, &bytesRead)

if (buffer[0] == 0xE9)
    {
         // Function hooked
           return true
    }
```

This malware's code first obtains a handle for `ntdll.dll` and the address for `NtAllocateVirtualMemory`. The code then invokes `ReadProcessMemory` to inspect the first byte of the `NtAllocateVirtualMemory` function. If the first byte is a jump instruction (hex `E9`), then the malware assumes that `NtAllocateVirtualMemory` is hooked and that it's being monitored by a sandbox or analysis tool.

We'll come back to this technique in a moment in "Performing Unaligned Function Calls" on page 140.

Hook Removal (Unhooking)

After detecting a hook, the malware sample can attempt to remove it by restoring the original data. There are a few ways in which malware can attempt to do this.

First, malware can manually unload any suspicious modules (injected hooking DLLs) that it determines have been loaded into its process address space. Once it detects an anomalous module, it can call the `FreeLibrary` function. `FreeLibrary` takes as a parameter the handle of the library module the malware wishes to unload.

A possibly better way for malware to accomplish this unhooking is by manually reloading Windows libraries that appear to be hooked. Malware can scan its loaded libraries for signs of a hooking module, and once it detects a hook, it can unload that DLL (using a function such as `FreeLibrary`) and then reload the fresh, unhooked library from disk. This effectively removes any function hooks installed by the sandbox or analysis tool.

Alternatively, once the malware detects that a function is hooked, it can simply rewrite the original code into the function, replacing the jump to the hooking code. To unhook an inline hook, the malware can simply remove the hooked bytes of the function (the jump statement) or overwrite them with something else, as the following pseudocode demonstrates:

```
handle = GetModuleHandle("ntdll.dll")
functionAddress = GetProcAddress(handle, "NtAllocateVirtualMemory")

VirtualProtect(functionAddress, size, PAGE_EXECUTE_READWRITE, &oldProtect)
memcpy(functionAddress, "\x4c\x8b\xd1\xb8", 4)

VirtualProtect(functionAddress, size, oldProtect, &newOldProtect)
```

In this code, the malware gets the address (`GetProcAddress`) of the library and function it wishes to unhook (in this case, `NtAllocateVirtualMemory`), then calls `VirtualProtect` to prepare the function for modification by giving it execute, read, and write permissions. Then, the malware copies (`memcpy`) the four bytes (`\x4c\x8b\xd1\xb8`) to the beginning of the target function's code. These bytes are the standard, unhooked, original bytes that would reside in the target function before they were hooked by the sandbox. Finally, the malware calls `VirtualProtect` again to change the memory permissions back to what they originally were.

Some sandboxes are aware that malware can try to unhook their installed function hooks and will be on the lookout for this. Similar to how malware scans its process memory for signs of hooking, sandboxes can periodically check whether their hooks are still in place and, if not, replace them. Or, sandboxes may monitor malware unhooking behaviors, such as by monitoring calls to `ReadProcessMemory`, `WriteProcessMemory`, `memcpy`, `FreeLibrary`, and others.

Next, let's discuss a subtler approach that malware can take to get around sandbox hooks: hook circumvention.

Hook Circumvention

As opposed to hook removal, *hook circumvention* bypasses or prevents hooking altogether. Examples of hook circumvention techniques include calling Windows functions in abnormal ways and manually loading code libraries (thus sidestepping the normal library-loading process). Since some sandboxes can detect whether their hooks are removed or altered, these methods can be less noisy and more difficult to detect.

Performing Unaligned Function Calls

In *unaligned* function calling, the malware indirectly calls functions by jumping over the sandbox hooking code, effectively skipping it entirely. Normally, malware will call a Windows API function, such as `ReadFile`, by using a call instruction (`call ReadFile`). This instruction will jump to the beginning of `ReadFile` (inside the *kernel32.dll* module) and execute this code. If the `ReadFile` function is hooked by a sandbox, however, the hooking code will be executed first, as discussed earlier in this chapter. In the following code, a hook has been injected into this function:

```
0x77000000  jmp hook_code
0x77000005  // Start of real ReadFile code
```

To implement an unaligned function call, the malware can directly jump to the `0x77000005` address by executing the instruction `jmp 0x77000005` (or adding 5 bytes to the base address, as in `jmp 0x77000000 + 0x5`), rather than calling `ReadFile` normally. This will skip the hooking `jmp` statement at `0x77000000` and directly execute the real `ReadFile` code starting at `0x77000005`.

One caveat here is that the malware must explicitly specify the function address, meaning it must know that address beforehand. One way the

malware can obtain the address is by calling `GetProcAddress`, as shown in this simplified assembly code:

```
--snip--
call GetProcAddress
mov  address, eax
cmp  [address], 0E9h
je   skip_hook
--snip--
skip_hook:
lea  eax, [address+5]
jmp  eax
```

The malware sample calls `GetProcAddress` to get the address of its desired target function, and it then stores that value in `address` (`mov address, eax`). The address points to the beginning of the function, where the malware is checking for hooks. Next, the malware compares the code at this address to the hex value `0E9h` (one of the assembly opcodes for `jmp`). If this opcode exists, the code jumps to the `skip_hook` function, which adds 5 bytes to the address of the target function and stores the pointer to this final address in EAX (`lea eax, [address+5]`). Finally, the code jumps to this new address (`jmp eax`), bypassing the hook.

Calling Low-Level and Uncommon Functions

To circumvent hooking behaviors in sandboxes and analysis tools, some malware invokes lower-level Native API calls, attempting to avoid the more commonly hooked higher-level calls. For example, malware can call the `NtProtectVirtualMemory` function directly, rather than calling `VirtualProtect` in an attempt to bypass any hooks on the latter.

Alternatively, malware can even make direct syscalls into the kernel, bypassing the normal WinAPI calling procedures. (We discussed syscalls in Chapter 1.) Some sandboxes may not monitor direct calls into the kernel, and that can leave blind spots in the analysis reports from these sandboxes. As this is also a technique used to circumvent endpoint defenses, we'll return to this topic in detail in Chapter 13.

Since automated sandboxes and some malware analysis tools hook or monitor the common Windows functions, malware may also use uncommon functions as a hook circumvention tactic. The Windows API contains a huge number of functions that cover nearly every task a program could want to complete, so inevitably, there are rarely used and near-duplicate functions. For example, the `SHEnumKeyEx` function is very similar to `RegEnumKey` and can also be used to enumerate registry keys, but it's far less commonly used. Thus, `SHEnumKeyEx` may receive less attention from automated sandboxes and analysts and may go unnoticed when used by malware to thwart hooking attempts.

Unfortunately, providing a list of all of these lesser-used functions is impossible since the Windows API is so extensive. However, it's important

to keep this tactic in mind when investigating malware and researching any API calls you're unfamiliar with.

SOCKETS

Most modern Windows applications use higher-level network communication libraries such as WinINet (*Wininet.dll*), WinHTTP (*Winhttp.dll*), and URLMon (*Urlmon.dll*). These are also some of the internet communication libraries most commonly loaded by malware; in fact, most of the malware examples throughout this book use these libraries. The primary benefit of these libraries for malware authors is their ease of use and simple implementation.

That said, some malware uses the lower-level library Winsock instead. With Winsock, malware authors have greater flexibility in the way they craft and manipulate their network connections. Additionally, because they operate at a lower level than the previously mentioned libraries, Winsock functions may fly under the radar of analysis tools like web proxies, and analysts can therefore miss some malware behaviors. The following pseudocode demonstrates how a malware sample might create a socket and send data to a remote server:

```
int sock = socket("AF_INET", 1, 6);
int connect(sock, *sockaddr, length);
send(sock, *data , strlen(*data) , 0 );
```

In this basic example, the malware sample creates a socket (`sock`) with parameters specifying that it should use IPv4 (`AF_INET`), connection-based byte streams (`1`), and the TCP protocol (`6`). Next, the malware attempts to connect to a remote server (`connect`), specifying `sock` and a pointer to the `sockaddr` table, which contains information about the remote service, such as the hostname and TCP port number. Finally, the malware sends data (`send`) to the remote server, specifying a pointer to `data`, which contains the data that the malware wishes to send to the remote server.

The details of sockets and how they work are beyond the scope of this book. For more information on sockets and all their possible parameters, MSDN is a great resource.

Manually Loading Libraries and Calling Functions

Malware can also manually load Windows libraries, rather than relying on the standard Windows loader. As you may recall from Chapter 1, the standard way in which Windows applications load libraries is by using functions such as `LoadLibrary`. The `LoadLibrary` function maps the requested library into memory, making for a quick and simple loading process, with the OS doing all the heavy lifting. The downside to this simplicity is that sandboxes

and other analysis tools can easily implement hooks within this library to intercept function calls.

To circumvent this, malware can manually map the library file into its process address space by using `NtMapViewOfSection`, as shown in this simplified pseudocode:

```
file_name = "C:\Windows\System32\Ntdll.dll"
NtCreateFile(file_handle, ..., file_name, ...)
NtCreateSection(section_handle, ..., file_handle)
NtMapViewOfSection(section_handle, process_handle, ...)
```

In this example, the malware uses `NtCreateFile` to get a handle to the file *C:\Windows\System32\Ntdll.dll*, which is the library it wishes to load. Next, the malware creates a section object using `NtCreateSection` and references the previously obtained file handle. A *section object* is a section of memory that can be shared with other processes, and it provides a method of mapping a file into this area of memory. After the section object is created, the malware maps the *ntdll.dll* file into it using `NtMapViewOfSection`. The `process_handle` variable represents the target process into which the file will be mapped. In this case, it's the malware's own process.

Another similar method is to read the file from disk, rather than mapping it into memory. To read *ntdll.dll* from disk, the malware can call `ReadFile` (or `NtReadFile`) and pass the target filename as a parameter. With either of these methods, once the library is mapped or read into memory, the malware can execute its intended functions by jumping to or calling the addresses in the target library. Note that these methods would not be effective "out of the box" and would require some additional work from the malware, such as properly locating the offsets of the functions within the DLL it wishes to call.

Writing Custom Functions

Finally, malware authors may choose to rewrite Windows functions entirely and include them in their malware samples to avoid hooking. This is often the most difficult hook circumvention technique to implement; many factors come into play, and the modified function must work perfectly with the victim host's operating system. It's quite rare to see this malware approach in practice.

Anti-hooking Toolsets

There are also tools written specifically for anti-hooking purposes. One example is the appropriately named anticuckoo project (*https://github.com/therealdreg/anticuckoo*), which detects potential sandbox hooking by using various methods. Additionally, the tool allows users to exploit the sandbox by modifying the hooked function's code and possibly causing a memory stack corruption, thus causing the sandbox to crash. This project doesn't seem to be maintained anymore, but it's a good example of research on the topic of sandbox anti-hooking. For additional information on this technique, read

the informative blog post "Prevalent Threats Targeting Cuckoo Sandbox Detection and Our Mitigation" at *https://www.fortinet.com/blog/threat-research/prevalent-threats-targeting-cuckoo-sandbox-detection-and-our-mitigation*.

Malware analysis is a cat-and-mouse game. Offensive-security researchers and malware authors consistently come up with new ways to detect and circumvent hooking, so malware analysts and sandbox developers must adapt. For example, the Cuckoo sandbox authors implemented several *anti-anti-hooking* techniques, such as preventing hooks from being overwritten by restricting memory protection modification. Many other commercial sandboxes have implemented similar functionalities.

Circumventing Sandbox Analysis

Because they're automated, sandboxes are susceptible to evasion tactics at the meta level, by which I mean the level of the sandbox product itself, not its implementation or the underlying OS. For example, certain sandboxes have a size limit on submitted files, so malware authors can simply artificially increase the size of the malware file to circumvent them. Other sandboxes can't process certain file types or scripts. It's becoming more common for malicious files to be delivered via email in an encrypted state, with the decryption password in the text of the email. An end user may happily enter this password, decrypt the file, and run the malware, but a sandbox has a much more difficult time with this!

Also, some sandboxes have trouble monitoring certain file types. At the time of this writing, many commercial and open source sandboxes don't fully support Microsoft .NET, which is a cross-platform development framework for Windows. Since .NET implements its own functions that differ from the native Windows and NT API functions, these sandboxes may miss important details about the malware's behaviors and functionalities.

These are just a few examples, and there are many other methods of tricking sandboxes into not executing the malware at all. Keep this in mind when analyzing malware in an automated sandbox, and always be on the lookout for the evasion techniques listed here. It's also important to properly evaluate a sandbox product to ensure it fits your needs before you deploy it in your environment.

Disrupting Manual Investigation

The techniques discussed in this chapter so far have focused on evading sandboxes, but malware can also directly interfere with manual analysis. For example, Chapter 4 described how malware can enumerate the processes running on a host so that it can detect a sandbox environment, a VM, or analysis tooling. However, along with detecting these tools, some malware can actively terminate them.

To terminate a target process, malware can iterate through the process tree by using `CreateToolhelp32Snapshot`, `Process32First`, and `Process32Next`, as you saw in Chapter 4. The malware can then call `OpenProcess` to obtain a handle to a victim process, followed by `TerminateProcess`. The following assembly code example demonstrates how a malware sample might terminate a remote process:

```
--snip--
push    [ebp+dwProcessId] ; PID of "wireshark.exe"
push    0  ; bInheritHandle
push    0x1  ; dwDesiredAccess
call    OpenProcess
mov     [ebp+ProcessHandle], eax
xor     eax, eax
--snip--
push    [ebp+ProcessHandle]
call    TerminateProcess
```

In this code snippet, the malware calls `OpenProcess` with parameters representing the `processID` of the target process (*wireshark.exe*, in this case), the `InheritHandle` value (which isn't important here), and the `dwDesiredAccess` value (the process access rights that the malware's process is requesting). In this case, the malware is requesting access rights `1` (`0x1` in hex), which equates to `PROCESS_TERMINATE` and allows a calling process (the malware) to terminate another process (*wireshark.exe*). Wireshark is, of course, just an example here. Malware can query and terminate any process if it has the correct permissions to do so.

NOTE *Sometimes renaming a malware analysis tool's executable file before launching it will trick simple malware that's employing this method. For example, renaming* wireshark.exe *to* krahseriw.exe *might prevent malware from "seeing" this process, thus preventing its termination. This solution won't work in all cases, however.*

Another tactic malware can use is disorienting the analyst. One interesting malware sample I've investigated creates a directory under *C:\Users\<user>\AppData\Local\Temp*. The malware names the directory a randomly generated number (for example, *21335493*) and writes temporary files that are necessary to its functionalities into it. In order to protect the directory, the malware constantly enumerates all open windows, looking specifically for windows that reference this temporary directory name, and issues a "kill" request for the window if there's a match.

Here's a simplified pseudocode example of this technique in action:

```
windows[] = EnumWindows()
for (i = 0; i < windows[].length; i++) {
    window_text = GetWindowText(windows[i])
    if (windows_text == "21335493") {
        PostMessage(windows[i], WM_CLOSE)
    }
}
```

This malware sample uses `EnumWindows` to enumerate all desktop windows and then loops through all the window title text, using `GetWindowText`, to look for `21335493`. If the code finds a window containing this text, the malware calls the `PostMessage` function with the `WM_CLOSE` parameter, forcing that window to close. Now, if the malware analyst tries to open the *21335493* temporary directory in, say, Explorer, it will be closed automatically before the analyst can inspect its contents.

These two examples only scratch the surface. Starting in Chapter 10, I'll discuss other interesting measures that malware authors can implement in their code to confuse and impede manual analysis.

Hypervisor Exploits and VM Escaping

The last technique we'll cover in this chapter may be the ultimate sandbox and VM evasion move: exploiting the hypervisor itself or escaping it entirely. While it's rarely seen in malware, there have been occasional uses of this technique in the wild, as well as the odd vulnerability discovered in products such as VMware and VirtualBox. One notable example is Cloudburst, an exploit developed in 2009 by Immunity Inc. that affected certain versions of VMware hypervisors. Playing a specially crafted video file on the Windows VM would exploit a flaw in VMware's display functions and possibly allow code to execute on the host OS itself.

Most known hypervisor vulnerabilities don't directly allow code execution on the host, meaning that complete "escape" from the sandbox environment is unlikely. For example, some of these vulnerabilities allow for writing files to the host or possibly reading files from the host, but they won't allow malicious files or code to be executed on the host. In addition, at the time of this writing, all of these discovered and reported vulnerabilities have been patched by their respective hypervisor vendors. As long as you, the malware analyst, are detonating malware on an updated and patched hypervisor, your host system is theoretically safe.

NOTE *I say "theoretically" here because there's always the possibility of zero-day vulnerabilities and unknown, unreported bugs in hypervisor code that malware could potentially exploit. There's always a risk when you're analyzing malware, but I believe any risk is outweighed by the benefits. In Appendix A, we'll discuss a few steps you can take to ensure you're working in the safest environment possible.*

Evasion Countermeasures

As mentioned earlier, there's a cat-and-mouse game between malware authors and malware researchers: authors invent a novel technique for detecting or bypassing analyst tools and sandboxes, and analysts and defensive-security researchers adapt. A great example of this is how far automated-analysis sandboxes have come. Many modern sandboxes have implemented countermeasures for the detection and evasion tactics mentioned throughout the past few chapters.

Sandboxes can alert malware analysts to detection and evasion attempts, providing a window into the malware internals and enabling the analysts to respond appropriately. You can manually circumvent many such techniques by attaching the process to a debugger, setting breakpoints on interesting function calls, and modifying the malware's code in the debugger itself or in a disassembler. These function calls can be `nop`'ed out, jumped over, or modified (by manipulating the function parameters or return values, as Chapter 3 explained). Finally, many of the techniques can be circumvented by properly configuring your VM and hypervisor. I'll discuss how to do so in Appendix A.

Summary

This chapter gave you an overview of the methods that malware might use to evade sandboxes, VM environments, and analysis tooling when it detects that it's being monitored. In Part III, you'll build on some of this knowledge as we begin to explore how malware uses anti-reversing techniques to interfere with disassemblers, detect and evade dynamic code analysis tools like debuggers, and misdirect malware analysts.