

5

Anti-Debugging Tricks

The sections in this chapter demonstrate how an analyst may identify whether the application is being debugged or inspected. There are numerous debugging detection techniques; some of them will be covered in this chapter. Obviously, an analyst is capable of mitigating any technique; nevertheless, certain techniques present greater complexity than others.

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

- Detecting debugger presence
- Spotting breakpoints
- Identifying flags and artifacts

Technical requirements

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

As far as compiling our examples, I use MinGW (<https://www.mingw-w64.org/>) for Linux, which I install via the following command:

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

Also, in this chapter, we are using <https://github.com/x64dbg/x64dbg> in our practical cases.

Detecting debugger presence

The first thing that must be done is to determine whether or not the application is being run with a debugger attached to it. There are a lot of different approaches to debugging detection, and we are going to go over some of them. A malware analyst may, of course, reduce the risk posed by any methodology; nevertheless, some methods are more difficult to implement than others.

It is possible to *ask* the operating system whether or not a debugger is attached. The `IsDebuggerPresent` function is responsible for checking whether or not the `BeingDebugged` flag is set in the **process environment block (PEB)**:

```
B00L IsDebuggerPresent();
```

You can find relevant documentation here:

<https://learn.microsoft.com/en-us/windows/win32/api/debugapi/nf-debugapi-isdebuggerpresent>.

Practical example 1

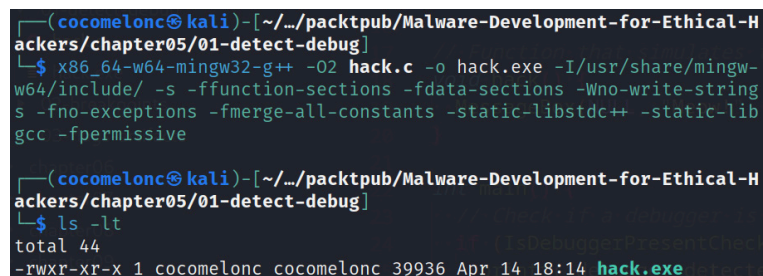
The full source code of the **proof of concept (PoC)** looks like this:

```
/*
 * Malware Development for Ethical Hackers
 * hack.c - Anti-debugging tricks
 * detect debugger
 * author: @cocomelonc
 */
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
// Function to check if a debugger is present
bool IsDebuggerPresentCheck() {
    return IsDebuggerPresent() == TRUE;
}
// Function that simulates the main functionality
void hack() {
    MessageBox(NULL, "Meow!", "=^..^=", MB_OK);
}
int main() {
    // Check if a debugger is present
    if (IsDebuggerPresentCheck()) {
        printf("debugger detected! exiting...\n");
        return 1; // exit if a debugger is present
    }
    // Main functionality
    hack();
    return 0;
}
```

Let's examine everything in action. Compile our PoC source code:

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

The result of running this command on Kali Linux looks like this:



```
(cocomelonc@kali) - [~/packtpub/Malware-Development-for-Ethical-Hackers/chapter05/01-detect-debug]
$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive

(cocomelonc@kali) - [~/packtpub/Malware-Development-for-Ethical-Hackers/chapter05/01-detect-debug]
$ ls -lt
total 44
-rwxr-xr-x 1 cocomelonc cocomelonc 39936 Apr 14 18:14 hack.exe
```

Figure 5.1 – Compiling our “malware”

Then, open it with x64dbg. For example, on the Windows 10 x64 VM, in our case, it looks like this:

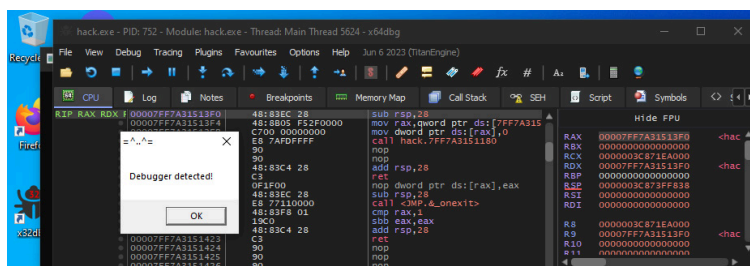


Figure 5.2 – Running our malware via x64dbg

The anti-debugging logic worked as expected. Absolutely perfect!

Another trick with checking debuggers is using another function. The **CheckRemoteDebuggerPresent()** function checks whether a debugger (in a different process on the same machine) is attached to the current process.

Practical example 2

The logic of checking looks like this:

```
// Function to check if a debugger is present
bool DebuggerCheck() {
    BOOL result;
    CheckRemoteDebuggerPresent(GetCurrentProcess(), &result);
    return result;
}

int main() {
    // Check if a debugger is present
    if (DebuggerCheck()) {
        MessageBox(NULL, "Bow-wow!", "=^..^=", MB_OK);
        return 1; // exit if a debugger is present
    }
    // Main functionality
    // something hacking
    return 0;
}
```

Let's examine everything in action. Compile our PoC source code:

```
$ x86_64-w64-mingw32-g++ -O2 hack2.c -o hack2.exe -I/usr/share/mingw-w64/include/ -s -ffunction-se
```

The result of running this command on Kali Linux looks like this:

```
(cocomelon@kali)~/.../packtpub/Malware-Development-for-Ethical-Hackers/chapter05/01-detect-debug
$ x86_64-w64-mingw32-g++ -O2 hack2.c -o hack2.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fd
merge-all-constants -static-libstdc++ -static-libgcc -fpermissive -w

(cocomelon@kali)~/.../packtpub/Malware-Development-for-Ethical-Hackers/chapter05/01-detect-debug
$ ls -lt
total 40
-rwxr-xr-x 1 cocomelon cocomelon 14848 Apr 19 11:50 hack2.exe
```

Figure 5.3 – Compiling our malware

Then, open it with the x64dbg debugger. On the Windows 10 x64 VM, in our case, it looks like this:

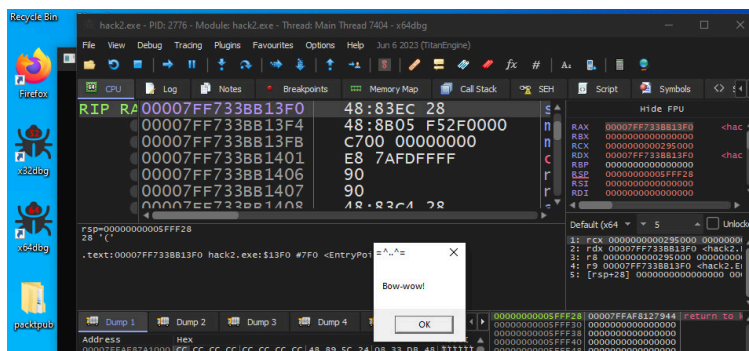


Figure 5.4 – Running our malware via x64dbg

If you run it:

```
> .\hack2.exe
```

The result of running this command on Windows looks like this:

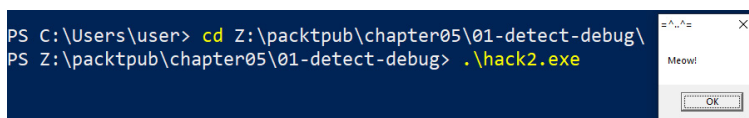


Figure 5.5 – Running the malware without attaching it to the debugger

As you can see, the anti-debugging logic worked as expected. Absolutely perfect!

Spotting breakpoints

The procedure of examining memory page permissions can aid in identifying program breakpoints set by a debugger. Initially, it is necessary to ascertain the total count of pages within the process working set and allocate a sufficiently large buffer to store all relevant information. Subsequently, the task involves iterating through memory pages and inspecting the permissions associated with each, with a specific focus on executable pages. We analyze each executable page to determine whether its **IF** statement is utilized by processes other than the current one. By default, memory pages are shared among all concurrently running programs. However, when a write operation occurs (e.g., inserting an **INT 3** instruction into the code), a copy of the page is mapped to the process's virtual memory. This copy-on-write mechanism results in the page no longer being shared after a write operation.

Practical example

The following is a simple PoC code in C that demonstrates the logic of checking memory page permissions to detect breakpoints:

<https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter05/02-breakpoints/hack.c>

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

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

The result of running this command on Kali Linux looks like this:

```
(cocomelonc@kali) - [~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter05/02-breakpoints]
$ i686-w64-mingw32-gcc -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -Wl,--disable-stdcall-fixup
(cocomelonc@kali) - [~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter05/02-breakpoints]
$ ls -lt
total 20
-rwxr-xr-x 1 cocomelonc cocomelonc 15360 Apr 14 18:41 hack.exe
```

Figure 5.6 – Compiling the PoC code

Then, on the victim's machine (Windows 10 x64 in my instance), start the debugger:

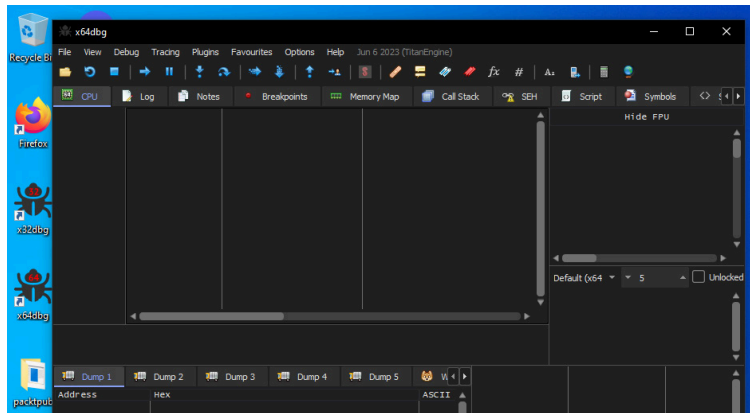


Figure 5.7 – Starting the debugger

Then, attach our `hack.exe` process:

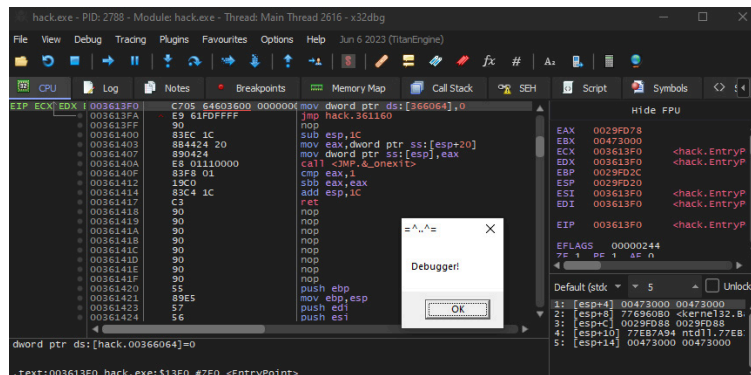


Figure 5.8 – Attaching `hack.exe`

As you can see, our logic has worked; the debugger is detected:

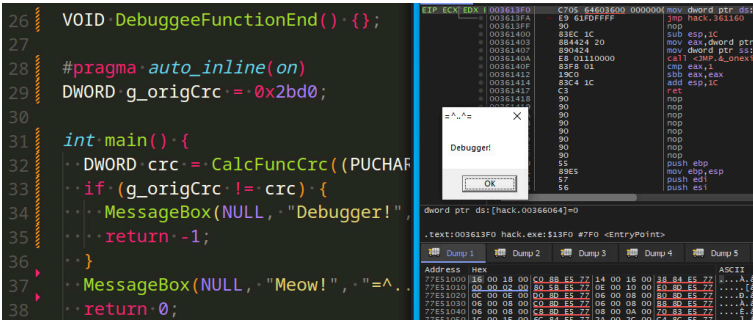


Figure 5.9 – Printing the “Debugger detected” message

Run our PoC code with the following command line:

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

As usual, on the Windows 10 x64 VM, it looks like this:

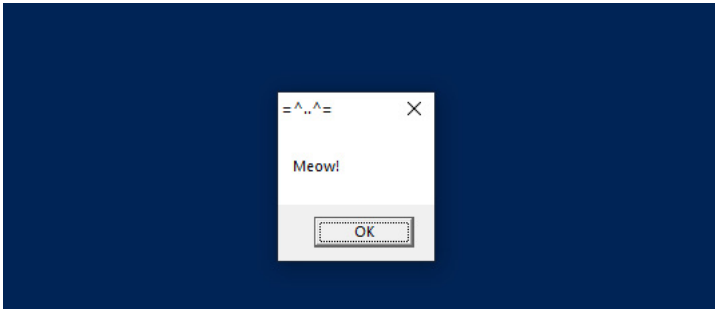


Figure 5.10 – Running hack.exe

The debugger is not detected, and the logic is working perfectly as expected.

IMPORTANT NOTE

You can use any other debugger instead of x64dbg (or x86dbg for 32-bit “malware”).

Let’s go research other techniques.

Identifying flags and artifacts

By default, the `0` value is stored in the `NtGlobalFlag` field of the Process Environment Block (located at offset `0x68` on 32-bit Windows and `0xBC` on 64-bit Windows):

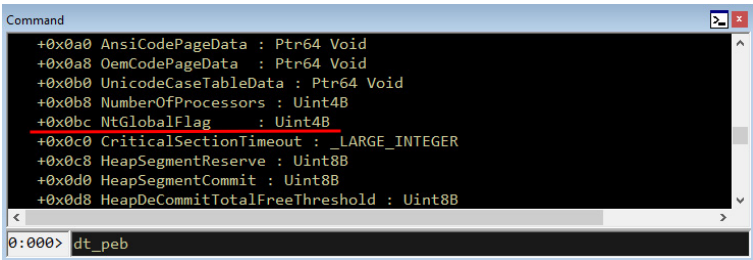


Figure 5.11 – NtGlobalFlag

The value of the **NtGlobalFlag** variable is unaffected by the attachment of a debugger. On the other hand, if a debugger was responsible for creating the process, the following flags will be set:

- **FLG_HEAP_ENABLE_TAIL_CHECK** (0x10)
- **FLG_HEAP_ENABLE_FREE_CHECK** (0x20)
- **FLG_HEAP_VALIDATE_PARAMETERS** (0x40)

To check whether a process has been started with a debugger, check the value of the **NtGlobalFlag** field in the PEB structure.

Practical example

Let's observe the practical implementation and demonstration via a straightforward PoC code for anti-debugging:

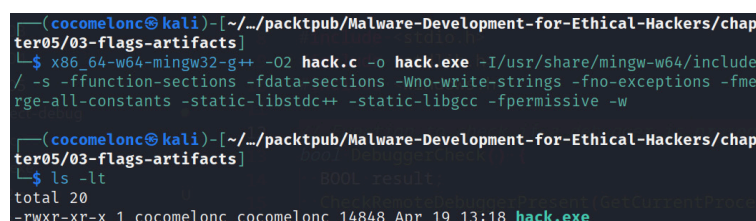
```
/*
 * Malware Development for Ethical Hackers
 * hack.c - Anti-debugging tricks
 * detect debugger via NtGlobalFlag
 * author: @cocomelonc
 */
#include <windows.h>
#include <stdio.h>
#define FLG_HEAP_ENABLE_TAIL_CHECK 0x10
#define FLG_HEAP_ENABLE_FREE_CHECK 0x20
#define FLG_HEAP_VALIDATE_PARAMETERS 0x40
#define NT_GLOBAL_FLAG_DEBUGGED (FLG_HEAP_ENABLE_TAIL_CHECK | FLG_HEAP_ENABLE_FREE_CHECK | FLG_HEAP_VALIDATE_PARAMETERS)
DWORD checkNtGlobalFlag() {
    PPEB ppeb = (PPEB)__readgsqword(0x60);
    DWORD myNtGlobalFlag = *(PDWORD)((PBYTE)ppeb + 0xBC);
    MessageBox(NULL, myNtGlobalFlag & NT_GLOBAL_FLAG_DEBUGGED ? "Bow-wow!" : "Meow-meow!", "=^..^=",
        return 0;
}
int main(int argc, char* argv[]) {
    DWORD check = checkNtGlobalFlag();
    return 0;
}
```

As you can see, the logic is not particularly complicated; all we do is check various flag combinations.

Compile it on Kali Linux (or any Linux machine with MinGW):

```
$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive -w
```

The result of running this command on Kali Linux looks like this:



```
(cocomelonc@kali)~/.../packtpub/Malware-Development-for-Ethical-Hackers/chapter05/03-flags-artifacts$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive -w
(cocomelonc@kali)~/.../packtpub/Malware-Development-for-Ethical-Hackers/chapter05/03-flags-artifacts$ ls -lt
total 20
-rwxr-xr-x 1 cocomelonc cocomelonc 14848 Apr 19 13:18 hack.exe
```

Figure 5.12 – Compiling our malicious PoC code

Run it and attach it to the x64dbg debugger:

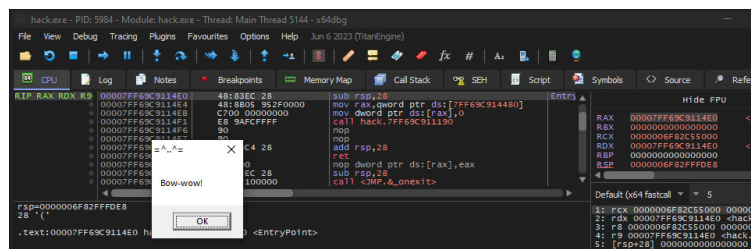


Figure 5.13 – Running hack.exe via x64dbg

Then, run **hack.exe** from the command prompt:

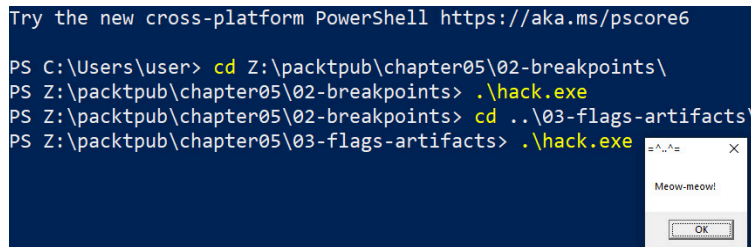


Figure 5.14 – Running hack.exe without the debugger

As you can see, everything is working perfectly.

ProcessDebugFlags

The next interesting trick with flags is the following: **EPROCESS**, a kernel structure that describes a process object, contains the field **NoDebugInherit**. The inverse value of this field can be obtained from the undocumented **ProcessDebugFlags (0x1f)** class. As a result, if the return value is **0**, the debugger is active.

Practical example

Let's observe the practical implementation and demonstration via a simple PoC code for this anti-debugging technique.

The full source code is available on GitHub at the following link:

<https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter05/03-flags-artifacts/hack2.c>

The main logic looks like the following function:

```
bool DebuggerCheck() {
    BOOL result;
    DWORD rProcDebugFlags, returned;
    const DWORD ProcessDebugFlags = 0x1f;
    HMODULE nt = LoadLibraryA("ntdll.dll");
    fNtQueryInformationProcess myNtQueryInformationProcess = (fNtQueryInformationProcess)
    GetProcAddress(nt, "NtQueryInformationProcess");
    myNtQueryInformationProcess(GetCurrentProcess(), ProcessDebugFlags, &rProcDebugFlags, sizeof(DWORD)
    result = BOOL(rProcDebugFlags == 0);
    return result;
}
```


Compile it as follows:

```
$ x86_64-w64-mingw32-g++ -O2 hack2.c -o hack2.exe -I/usr/share/mingw-w64/include/ -s -ffunction-se
```

The result of running this command on Kali Linux looks like this:

```
(cocomelonc@kali)~/packtpub/Malware-Development-for-Ethical-Hackers/chapter05/03-flags-artifacts$ x86_64-w64-mingw32-g++ -O2 hack2.c -o hack2.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive -w
(cocomelonc@kali)~/packtpub/Malware-Development-for-Ethical-Hackers/chapter05/03-flags-artifacts$ ls -lt
total 40
-rwxr-xr-x 1 cocomelonc cocomelonc 14848 Apr 19 14:24 hack2.exe
```

Figure 5.15 – Compiling our malicious PoC code

Run it and attach it to the x64dbg debugger:

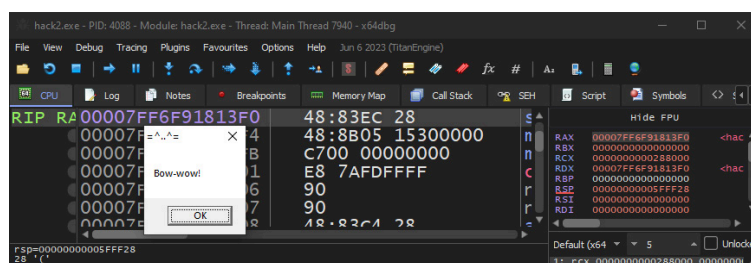


Figure 5.16 – Running hack2.exe via x64dbg

Then, run **hack2.exe** from the command prompt:

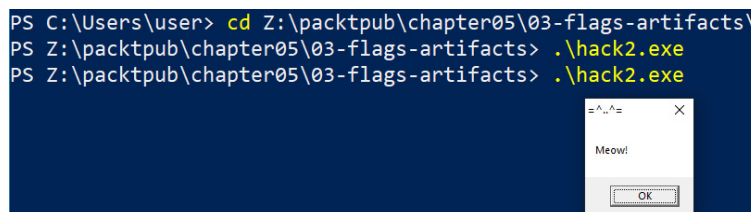


Figure 5.17 – Running hack2.exe without the debugger

As you can see, everything is working perfectly.

In conclusion of this chapter, I would like to note that all the described techniques work well with other debuggers, and can also create problems during manual analysis for malware analysis specialists of any level, from beginners to professionals, and therefore are actively used by attackers in real-world examples – for example, malicious software such as **AsyncRAT**, **DRATzarus**, and those actively used by the Lazarus APT group (we will look at APT groups in more detail in the final part of our book).

Summary

Throughout the chapter, readers delved into the intricate realm of detecting debugger presence, spotting breakpoints, and identifying flags and artifacts indicative of malware analysis.

The first skill empowers readers to discern whether their malware is operating under the scrutiny of an attached debugger, a critical insight for evading detection and analysis. The second skill introduces techniques to identify the presence of breakpoints, crucial elements in the arsenal of malware analysts. This knowledge is paramount for developers seeking to build resilient malicious software that can operate undetected.

Then, we took a deeper dive into the nuanced indicators that reveal malware under analysis. Understanding specific flags that betray the watchful eye of a malware analyst is essential for crafting sophisticated and evasive malware. Each skill is accompanied by a practical example, ensuring a hands-on learning experience that solidifies theoretical concepts.

In this chapter, we have unraveled the intricacies of anti-debugging methods, recognizing that while every technique may be subject to analyst mitigation, some prove more formidable than others. Aspiring ethical hackers, armed with the insights gained from this chapter, are better prepared to navigate the complex landscape of malware development.

In the next chapter, we will discuss anti-virtual machine techniques and how to use them in practice.