Study of the Anti-Debugging Techniques and their Mitigations

Muhammad Saad and Muhammad Taseer
School of Electrical Engineering and Computer Sciences, NUST, Islamabad, Pakistan
Corresponding author: 12msccmsuleman@seecs.edu.pk

Abstract:
The major goal of this study is to provide anti-debugging and anti-reversing strategies/techniques employed by executables, DLLs, and packers/protectors, as well as to examine strategies that can be utilized to bypass or disable these protections. Anti-debugging techniques are designed to make sure that a program is not being executed inside a debugger. In most cases, the anti-debugging process slows down the reverse engineering [1] process but doesn't stop it. This information will allow malware analysts and researchers to identify the techniques used by the malware. This information may also be used by security researchers, reverse engineers those want to slow down the process of reverse engineering in order to add security [2] to their software. It causes some difficulties for a reverse engineer, but, of course, nothing stops a skilled, knowledgeable, and committed reverse engineer.

Keywords: malware analysis, anti-debugging, anti-reversing, protectors, packers

1. Introduction

Prior to then, malware Development served as a showcase for malware coders. Malware analysts have used debuggers to run a malware program's instructions one by one, introducing modifications to memory spaces, settings as well as variable values. Debuggers are the most commonly used reverse engineering tools, such as Interactive Disassembler (IDA), x64dbg, and OllyDBG. If debugging is successful, it helps to understand malware behavior and its capabilities. This is something malware developers would like to avoid. That is why they must implement anti-debugging techniques. Anti-debugging techniques[3] can be used to merely detect the presence of a debugger, deactivate it, lose control of it, or even take advantage of a flaw in the debugger. Disabling or avoiding debugger checks can be done generally and specifically. However, you can exploit this vulnerability against specific debuggers. Furthermore, The Supervisory Control and Data Acquisition (SCADA)[4] system has a vulnerability, according to the Trend Micro report "Unseen Threats, Imminent losses," which is the part of industrial...
control systems (ICS)[5]. In addition, In many situations, knowing how to apply anti-debugging techniques to malicious code to prevent it from being tracked down and evaluated is also helpful. One of the main tools used by malware analysts and reverse engineers is the debugger. What is a debugger? A debugger is software that is used to evaluate and control the flow of execution of other executables or software. By using a debugger, we can execute each instruction step by step and can note down the changes that can be displayed on the stack, memory dumps, registers, etc. Most packers use these techniques to determine whether the system is running a debugger or if a process is being debugged. These debugger detection methods[6] include checks that are relatively basic all the way up to ones that are applicable to native Application Programming Interfaces (APIs) and kernel objects[7]. This section discusses how anti-debugging techniques work. Each process's user space contains a data structure called a Process Environment Block (PEB), which holds information about the related process. Each process's user space contains a data structure called a Process Environment Block (PEB), which holds information about the related process. It is intended to access Windows API (WinAPI)It is intended to access Windows API (WinAPI) but access is not restricted by this. Process Environment Block (PEB) can be accessed directly from memory. Checking the value of the Process Environment Block (PEB) structure that has been debugged is a relatively straightforward implementation and technique. As we know that there are so many Applications Programmable Interfaces (APIs) which are documented and undocumented. For example, IsDebuggerPresent, which we will discuss later in this paper. To enhance, we can also check the APIs manually. The fs segment register can access the Process Environment Block (PEB) at fs: [30]. On an x86 [8] computer, this register corresponds to a Thread Information Block (TIB). There is also a flag below the Process Environment Block (PEB) that indicates whether the first memory space of the process was created in debug mode. Provide an offset of 0x18 in the Process Environment Block (PEB). So, here I break down the anti-debugging techniques into two categories: static anti-debugging and dynamic anti-debugging [9], as seen in the Table 1 below.

### Table 1. Static Vs Dynamic Techniques

<table>
<thead>
<tr>
<th>Difference</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Difficulty Level</td>
<td>Easy, Medium</td>
<td>Hard</td>
</tr>
<tr>
<td>Main Idea</td>
<td>Use System Info</td>
<td>Reverse and exploit</td>
</tr>
<tr>
<td>Target</td>
<td>Detect Debugger</td>
<td>Hide its own code and data</td>
</tr>
<tr>
<td>Time Point</td>
<td>When debugging start</td>
<td>While debugger is running</td>
</tr>
<tr>
<td>Defend Method(s)</td>
<td>API Hook, debugger plugin</td>
<td>API hook, Debugger Plugin</td>
</tr>
<tr>
<td>Example(s)</td>
<td>PEB, TEB, TLS</td>
<td>Breakpoints (INT3), TimingCheck</td>
</tr>
</tbody>
</table>

In our research we will discuss we will discuss some of the main anti-debugging techniques and how a reverse engineer can be able to identify them easily for example in this paper we will discuss about the IsDebuggerPresent, TimeChecks, NtQueryInformationProcess, NtSetInformationThread, SwitchDesktops, SeDebugPrivilege, ParentProcess, DebuggerWindow, DeviceDrivers etc.

**Anti-Debugging Techniques Mechanism:**

Anti-debugging[10] is the implementation of
one or more techniques in computer code that make it difficult to reverse engineer or debug the target process. These techniques are ways for a program to detect whether it is running under the control of a debugger[11]. If a debugger is detected, the malware will execute arbitrary code, usually code to terminate. The anti-debugging process slows down the reverse engineering process but doesn't stop it.

2. Is Debugger Present:

The easiest debugger detection technique is to check the BeingDebugged flag in the Process Environment Block (PEB). The kernel32!IsDebuggerPresent() function was introduced in Windows 95, and the Application Programmable Interface (API) checks the value of this flag to identify the process whether it is in the user-mode debugger. This code (same 32-bit or 64-bit Windows environment) can be used for verification to check the 32-bit or 64-bit Windows environment. As we can see the assembly code of the IsDebuggerPresent() in Figure 1.

![Figure 1. Assembly code of IsDebuggerPresent()](image)

**C/C++ Code:**

As we can see in the example if IsDebuggerPresent() in Figure 2.

```c
if (IsDebuggerPresent())
    ExitProcess(-1);
```

![Figure 2. C/C++ code of IsDebuggerPresent()](image)

**Solution:**

This technique can be easily bypassed by manually patching the Process Environment Block (PEB). BeingDebugged flag with the value 0x00 in the bytes.

3. Nt Query Information Process () / Check Remote Debugger Present ()

CheckRemoteDebuggerPresent() is another a debugger should be attached to a process? Use this Check Remote DebuggerPresent() to decide. The API calls ntdll!ProcessDebugPort inside the kernel A value that is not zero in the DebugPort field tells that the process is being debugged in user mode by the debugger. If so, ProcessInformation will be set to 0xFFFFFFFF, otherwise the value of ProcessInformation will be 0x0. The CheckRemoteDebuggerPresent()[12] function in Kernel32 is functional. On either the 32-bit or 64-bit version of Windows, the check can be made by using this 32-bit code to look at the 32-bit window environment. The Function The function CheckRemoteDebuggerPresent() takes 2 parameters; the first parameter is the (PID), and the A pointer to a Boolean variable serves as the second parameter. That will hold TRUE if the process is being debugged. As we can see from the C/C++ code in Figure 3.

![Figure 3 C/C++ Code for CheckRemoteDebuggerPresent()](image)
ProcessInformation class is set to as ProcessDebugPort as we can see C/C++ code in the Figure 4.

![Figure 4. C/C++ Code for NtQueryInformationProcess()](image)

This example shows how the call to the CheckRemoteDebuggerPresent() and To see whether the current process is being debugged, utilize the NtQueryInformationProcess function, as we can see in Figure 5 and Figure 6.

![Figure 5 Assembly code of CheckRemoteDebuggerPresent()](image)

![Figure 6. Assembly code of NtQueryInformationProcess()](image)

Solution:
One solution is to set NtQueryInformationProcess(return y)`s value is a breakpoint. ProcessInformation is patched to a DWORD value of 0 when the breakpoint is reached of 0.

4. Nt SetInformation Thread:

NtSetInformationThread()[13] is usually used to set the priority of a thread. It can also be used to hide threads from the debugger. It can also be done with the help of a non-documented value, which is not documented but can be used. THREAD_INFORMATION_CLASS::ThreadHideFromDebugger (0x11). When a thread is hidden in the debugger, it will not be informed of anything pertaining to that thread not be informed of anything pertaining to that thread. The thread is also capable of anti-debugging methods, such as examining debug flags, code checksums, etc. If there are hidden breakpoints in the thread, if we try to keep the main thread hidden from the debugger, either the process will crash or the debugger will get stuck. An example of calling the NtSetInformationThread would be like this, as we can see in Figure 7.

![Figure 7. Assembly code of NtSetInformationThread()](image)

C/C++ Code:
As we can see, C/C++ code in Figure 8.

![Figure 8. C/C++ code of NtSetInformationThread()](image)

Solution:
The breakpoint is set to ntdll!NtSetInformationThread(), and when the breakpoint is hit, reverse engineers can modify the EIP, to prevent the API calls from reaching the kernel and being called from other functions.
5. **SwitchDesktop()**

Platforms based on Windows NT allow for multiple desktop sessions. The windows of the previous active desktop can be hidden by choosing a different active desktop, but there is no visible way to return to the previous desktop. The mouse and keyboard events won’t be sent to the debugger from the debugger's desktop.[13], they no longer divulge their source, either. Debugging could become impossible as a result. Both the 32-bit and 64-bit versions of Windows can be used to make this call. Here is an example of a 32-bit version of Windows as we can see in Figure 9.

![Figure 9. Assembly code of SwitchDesktop()](image)

**C/C++ Code:**

As we can see the C/C++ code in the Figure 10.

![Figure 10 C/C++ code of SwitchDesktop()](image)

6. **Execution Time / Timing Checks**

When a reverse engineer tries to debug a process and uses a single step in code, there is a significant delay between the execution of the individual’s instructions[13]. The process is running under a debugger if the amount of time required is excessive compared to a typical execution. Here is a list of some instructions that can be used to increase the execution time of the instruction.

a. RDTSC (Read Time-Stamp Counter)

b. RDPMC (Read Performance-Monitoring Counters)

c. GetLocalTime

d. GetSystemTime

e. GetTickCount

Now we will take an example of a timing check.

As we can see in Figure 11.

![Figure 11 Assembly Code of GetTickCount()](image)

We check the synchronization using the kernel32 GetTickCount() API or manually verify that the SharedUserData structure’s TickCountLow and TickCountMultiplier entries are always set to 0xe. Identifying these timing techniques can be challenging, especially when RDTSC is used as spam, when other obscure techniques are used to mask them.
Solution:
One of the solutions is to identify where the time checks are and try to avoid stepping into them. Also, the code between these time checks. Reverse Engineers can place a breakpoint before that delta and execute instead of steps until a breakpoint is reached or a breakpoint is reached. We can also set a breakpoint in GetTickCount() to specify where to call it or to change its return value. Mitigations During Debugging: just fill time checks with NOPs and set the result of these checks to the appropriate value. For anti-debugging solution development: there is no great need to do anything with it, as time checks are not very reliable, but you can still hook timing functions and accelerate the time between calls.

Mitigations:
- During Debugging, just fill time checks with NOPs and set the result of these checks to the appropriate value.
- For anti-debugging solution development: there is no great need to do anything with it, as time checks are not very reliable, but you can still hook timing functions and accelerate the time between calls.

7. SeDebugPrivilege:

By default, the SeDebugPrivilege permission is disabled for the process access token. When a debugger like x32dbg, OllyDBG, etc. loads a process, SeDebugPrivilege permission is enabled. This is because these debuggers keep trying. SeDebugPrivilege permissions are inherited.

If the process can open the CSRSS.EXE process, then SeDebugPrivilege is active when the process is accessed.

Token pointing to the process being debugged. The test is valid for the following reasons: The Process Security Descriptor CSRSS.EXE allows the system access to the process.

However, if the process has SeDebugPrivilege privilege, other processes have independent access to the Security Descriptor. This permission is only granted to administrative groups by default, as we can see in Figure 12.

![Figure 12 Assembly Code of SeDebugPrivilege()](image)

This control uses ntdll! The CSRSS.exe GetProcessId() API gets the Process ID (PID) from CSRSS.EXE. You can get it manually by looking at the Process ID CSRSS.EXE processes. If OpenProcess() succeeds, SeDebugPrivilege is activated, indicating that the process is currently running and debugging, too.

Solution:
The ntdll breakpoint can be hit by setting a breakpoint as a solution. Returns from NtOpenProcess(). If PID passed by CSRSS.exe is CSRSS.exe, set the EX-value to 0xC0000022 (STATUS_ACCESS_DENIED).
**Parent Process:**
Users launch apps by clicking on the executable's icon that the shell process displays (Explorer.exe). By clicking on the executable's icon that the shell process displays, users can launch apps (Explorer.exe). Due to this, Explorer.exe becomes the parent process of the active process. This will show that the program was created by someone else and suggest that you can debug it.

1. Using Process32First/Next(), it will list every process and note explorer.exe. PROCESS32.szExeFile and the PROCESSENTRY32.th32parentProcessID are the two files that provide the process ID and the parent process ID of the current process, respectively.

2. The target is being debugged if the Process ID (PID) of the parent process differs from the Process ID (PID) of the explorer.exe.

**Solution:**
We need to patch the element of Kernel32!Process32NextW() that contains the code that performs a return after setting the value of EAX to 0.

**8. Debugger Window:**
The presence of the debug window is a flag that the debugger is running system[13]. Because the debugger creates windows with special class names (OllyDBG for OLLYDBG and WinDbgFrameClass for WinDbg), user32 can easily identify these debug windows! FindWindow()` or User32! findWindowEx().

**Solution:**
One solution is to set breakpoints in FindWindow() and FindWindowEx() When the breakpoint is hit, modify the value of the lpClassName string parameter to prevent the API from functioning. Setting the return value to NULL is another option.

**9. Debugger Process:**
List all the processes on the system and see whether the process name matches the name of the debugger to find out if it is currently running (for example, OLLYDBG.EXE, windbg.exe, etc.). Simple to implement; just use Process32First / Next() after confirming that the image name corresponds to the name of the debugger.

Sometimes these methods also use Kernel32 ReadProcessMemory() to read process memory and then look for debugger-related strings such as "x64dbg", "IDA", "OllyDBG", etc. to reverse engineer the debugger. To implement. After getting the debugger. The malware will stop his execution and silently exit or terminates the process.

**Solution:**
Another solution is to check the main process, including patching the kernel 32 patch! Process32NextW() always fails and prevents the developer from enumerating the process.

**10. Device Drivers**
An old technique is to verify that the debugger is running in a Kernel Mode in the system and try to, access device drivers. This technique is very simple and consists of simply making a call to the against well-known device names
used by kernel-mode debuggers, such as SoftICE, using Kernel32!CreateFile(). Some versions of Soft-ICE also add numbers to the device name, making it to check. The reversing forum's suggested technique is to brute force the corresponding digits until the right device name is discovered[14]. The new packer also uses device driver detection techniques to detect system monitors such as “Process Monitor” etc.

Solution:
Establishing a breakpoint in kernel32 is the simple fix. When the breakpoint is reached, CreateFileFileW() should either handle the FileName parameter or alter its return value to INVALID HANDLE VALUE (0xFFFFFFFF).

Process Memory:
A process can check or interact with its own memory for the presence of a debugger. This section includes anti-hitch methods[15] such as process memory and thread context checking, breakpoint DETECTION, PATCHING function and debugging functions.

11. Breakpoint and Patching Detection:

To verify if our code has any software breakpoints, we may still inspect the process memory, and we can also check the CPU debug registers to see if any hardware breakpoints have been set.

12. Software Breakpoints Detection:

Software breakpoints are defined as breakpoints that are created by altering the code at the target location and replacing it with the byte value 0xCC (INT3 / Breakpoint Interrupt)[17]. Finding the byte 0xCC in the API code and protector code will help you locate software breakpoints as seen by the example of assembly code in Figure 13.

![Figure 13 Assembly Code of Software Breakpoint Detection](image1)

C/C++ Code:
As we can C/C++ code in the Figure 14.

![Figure 14. C/C++ code of Software Breakpoint Detection](image2)

Solution:
Hardware breakpoints can be reverse engineered if software breakpoints are identified. If you need to set a breakpoint in the API code, and when the packer tries to find a breakpoint in the API code, reverse engineering the UNICODE API version allows for the setting of breakpoints. That eventually calls the ANSI version, such as LoadLibraryExW LoadLibrar-
yA or the native API corresponding to Load-
DLL to replace.

13. Hardware Breakpoints:

DR0, DR1, DR2, and DR3 are debug registers
that can be obtained from the thread context.
Debug registers 0-3 are used to store virtual
address of the so-called hardware breakpoints.
C/C++ Code:

As we can see C/C++ code in the figure 15.

![Figure 15 C/C++ code of Hardware
Breakpoints](image)

14. Memory Checks:

This section includes methods for directly
inspecting or modifying a process's virtual
memory in order to spot and stop debug-
ging[18].

15. Nt Query Virtual Memory ():

The memory page of the process in which the
code is located is shared by all processes prior
to the page being written. Then the OS creates
a replica of this page and allocates it to the
process's virtual memory[19], so the page is no
longer "shared". Now we can see how to
declare NTDLL, as we can see in figure 16.

![Figure 16 NTDLL Deceleration of
NtQueryVirtualMemory()](image)

C/C++ Code:
As we can see the C/C++ code in the Figure
17.

![Figure 17 C/C++ Code for Hardware
Breakpoints](image)

16. Detecting A function Patch:

Calling kernel32 is a common approach to find
a debugger. IsDebuggerPresent(). By altering
the outcome in the EAX register or hacking the
kernel32, you may easily get around this
check! IsDebuggerPresent(). Instead of
looking for breakpoints in the process memory, we can check to see if kernel32IsDebuggerPresent() has been altered[20]. The first few bytes of this function can be read and compared to the same function's bytes from other processes. Windows libraries are loaded at the same base address throughout the process, even if the Address Space Layout Randomization (ASLR) feature is enabled. The base address only changes across reboots but remains the same for the duration of the session.

Mitigations:

- During Debugging: Enter the function that conducts the Step-Over check and run it till the end(Ctrl + F9).
- Finding the specific check and either path it with NOPs or setting the return to a value that permits the application to keep running are the best ways to mitigate all "memory" techniques, including anti-step over.

Conclusion:
To defend itself against reverse engineering analysis, the malware employs anti-debugging techniques. Debug analysis can be avoided by anti-debugging techniques. Reverse engineers need advanced debuggers and knowledge to analyze malware using anti-debugging techniques. By applying common sense and slowly debugging the process, it is possible to identify the majority of anti-debugging techniques. For example, if you see that the code is terminating too rapidly in a conditional jump, which could mean preventing debugging technical. The most widely used anti-debugging methods involve fs access: [30h] by using a Windows API or performing a time check.

Of course, as with all malware analysis, the best way to learn how to stop it by using debugging techniques by continuously testing malware. Malware developers are constantly coming up with new techniques to evade debuggers and keep security researchers like you on their toes.

17. References:


