Anti Debugging Protection Techniques with Examples

By alexey.erko@apriorit.com

Published: 2019-05-23 · Archived: 2026-04-06 00:18:36 UTC

> Key takeaways:
>
> - Anti-debugging techniques increase the effort and cost of debugging for attackers, protecting your software and intellectual property.
> - Combining multiple anti-debugging techniques makes your software stronger and more resilient.
> - Process-level hardening and obfuscation add extra layers of protection on top of standard code-level anti-debugging techniques.
> - With expert guidance, anti-debugging measures can be applied effectively without affecting performance.

While debuggers are legitimate development tools, hackers can use them to research an application's logic, uncover vulnerabilities, and access proprietary algorithms.

This is where anti-debugging techniques step in. These techniques detect and obstruct unwanted debugging attempts so that hackers need much more time and expertise — and many more resources — to analyze or tamper with your software.

In this article, we explore practical anti-debugging methods, from basic checks to advanced protections, and demonstrate how experienced reverse engineers attempt to bypass them. You'll gain insights into how these techniques work, when they should be applied, and how they can help you protect your software from unauthorized analysis and tampering.

This article will be useful for project and development leaders who want to fortify their software.

Best anti-debugging techniques from Apriorit experts

Software developers use anti-debugging techniques to prevent malicious actors from analyzing or modifying a program's code or data. The main goal of these techniques is to complicate the process as much as possible by detecting if a program is being run in a debugger — a specialized software tool for analyzing and troubleshooting code execution. Generally, software engineers use debuggers during their coding workflow to search for errors and make sure their applications work as intended. However, hackers can also use debuggers to get into your code, see how it works, and steal or harm it. Therefore, your team must be ready for such threats and apply anti-debugging measures to protect your software.

It's important to understand that anti-debugging techniques are not foolproof. Determined attackers with advanced skills may still be able to bypass these protections. However, a competent team can use anti-debugging techniques to significantly increase the time and effort required to reverse engineer or crack software, making your programs safe from casual attacks.

At Apriorit, we use a range of proven anti-debugging techniques to make software more resilient against unauthorized access and tampering. Even though determined attackers may persist, our goal is to make it harder for them to compromise software integrity.

When choosing which techniques to use, we analyze the project's specifics and the client's requirements. You could choose one or several techniques, depending on the task you want to solve. Let's explore the most commonly used techniques in detail, showing practical examples for each.

1. Calling the IsDebuggerPresent function

Perhaps the simplest anti-debugging method is calling the IsDebuggerPresent function. This function detects if the calling process is being debugged by a user-mode debugger. The code below shows an example of elementary protection:

C++

```
 int main()
{
    if (IsDebuggerPresent())
    {
        std::cout << "Stop debugging program!" << std::endl;
        exit(-1);
    }
    return 0;
}
```

If we take a look inside the `IsDebuggerPresent` function, we'll find the following code:

ShellScript

```
 0:000< u kernelbase!IsDebuggerPresent L3
KERNELBASE!IsDebuggerPresent:
751ca8d0 64a130000000    mov     eax,dword ptr fs:[00000030h]
751ca8d6 0fb64002        movzx   eax,byte ptr [eax+2]
751ca8da c3              ret
```

For x64 process:

ShellScript

```
 0:000< u kernelbase!IsDebuggerPresent L3
KERNELBASE!IsDebuggerPresent:
00007ffc`ab6c1aa0 65488b042560000000 mov   rax,qword ptr gs:[60h]
```

```
00007ffc`ab6c1aa9 0fb64002          movzx eax,byte ptr [rax+2]
00007ffc`ab6c1aad c3                ret
```

We see the PEB (Process Environment Block) structure by the 30h offset relative to the fs segment (the 60h offset relative to the gs segment for x64 systems). If we look by the 2 offset in the PEB, we'll find the `BeingDebugged` field:

ShellScript

```
 0:000< dt _PEB
ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
```

In other words, the `IsDebuggerPresent` function reads the value of the `BeingDebugged` field. If the process is being debugged, the value is 1 if not, it's 0.

How to bypass the IsDebuggerPresent check

To bypass the IsDebuggerPresent check, set `BeingDebugged` to 0 before the checking code is executed. DLL injection can be used to do this:

ShellScript

```
 mov eax, dword ptr fs:[0x30]
mov byte ptr ds:[eax+2], 0
```

For x64 process:

ShellScript

```
 DWORD64 dwpeb = __readgsqword(0x60);
*((PBYTE)(dwpeb + 2)) = 0;
```

2. Inspecting the Process Environment Block (PEB) for debugger flags

PEB is a closed structure used inside the Windows operating system. Depending on the environment, you need to get the PEB structure pointer in different ways. Below, you can find an example of how to obtain the PEB pointer for x32 and x64 systems:

C++

```
 // Current PEB for 64bit and 32bit processes accordingly
PVOID GetPEB()
{
#ifdef _WIN64
    return (PVOID)__readgsqword(0x0C * sizeof(PVOID));
#else
    return (PVOID)__readfsdword(0x0C * sizeof(PVOID));
#endif
}
```

The WOW64 mechanism is used for an x32 process started on an x64 system, and another PEB structure is created. Here's an example of how to obtain the PEB structure pointer in a WOW64 environment:

C++

```
 // Get PEB for WOW64 Process
PVOID GetPEB64()
{
    PVOID pPeb = 0;
#ifndef _WIN64
    // 1. There are two copies of PEB - PEB64 and PEB32 in WOW64 process
    // 2. PEB64 follows after PEB32
    // 3. This is true for versions lower than Windows 8, else __readfsdword returns address of real PEB64
    if (IsWin8OrHigher())
    {
        BOOL isWow64 = FALSE;
        typedef BOOL(WINAPI *pfnIsWow64Process)(HANDLE hProcess, PBOOL isWow64);
        pfnIsWow64Process fnIsWow64Process = (pfnIsWow64Process)
            GetProcAddress(GetModuleHandleA("Kernel32.dll"), "IsWow64Process");
        if (fnIsWow64Process(GetCurrentProcess(), &isWow64))
        {
            if (isWow64)
            {
                pPeb = (PVOID)__readfsdword(0x0C * sizeof(PVOID));
                pPeb = (PVOID)((PBYTE)pPeb + 0x1000);
            }
        }
    }
#endif
    return pPeb;
}
```

The code for the function to check the operating system version is below:

C++

```
 WORD GetVersionWord()
{
    OSVERSIONINFO verInfo = { sizeof(OSVERSIONINFO) };
    GetVersionEx(&verInfo);
    return MAKEWORD(verInfo.dwMinorVersion, verInfo.dwMajorVersion);
}
BOOL IsWin8OrHigher() { return GetVersionWord() >= _WIN32_WINNT_WIN8; }
BOOL IsVistaOrHigher() { return GetVersionWord() >= _WIN32_WINNT_VISTA; }
```

3. Using TLS callbacks to detect debugger activity

Checking for the presence of a debugger in the `main` function is not the best idea, as this is the first place a reverser will look when viewing a disassembler listing. Checks implemented in `main` can be erased by `NOP` instructions, thus disarming the protection. If the CRT library is used, the main thread will already have a certain call stack before transfer of control to the `main` function. Thus a good place to perform a debugger presence check is in the TLS Callback. Callback function will be called before the executable module entry point call.

C++

```
 #pragma section(".CRT$XLY", long, read)
__declspec(thread) int var = 0xDEADBEEF;
VOID NTAnopPI TlsCallback(PVOID DllHandle, DWORD Reason, VOID Reserved)
{
    var = 0xB15BADB0; // Required for TLS Callback call
    if (IsDebuggerPresent())
    {
        MessageBoxA(NULL, "Stop debugging program!", "Error", MB_OK | MB_ICONERROR);
        TerminateProcess(GetCurrentProcess(), 0xBABEFACE);
    }
}
__declspec(allocate(".CRT$XLY"))PIMAGE_TLS_CALLBACK g_tlsCallback = TlsCallback;
```

4. Checking the NtGlobalFlag for debugging indicators

In Windows NT, there's a set of flags that are stored in the global variable `NtGlobalFlag`, which is common for the whole system. At boot, the `NtGlobalFlag` global system variable is initialized with the value from the system registry key:

```
[HKEY_LOCAL_MACHINESYSTEMCurrentControlSetControlSession ManagerGlobalFlag]
```

This variable value is used for system tracing, debugging, and control. The variable flags are undocumented, but the SDK includes the gflags utility, which allows you to edit a global flag value. The PEB structure also includes

the `NtGlobalFlag` field, and its bit structure does not correspond to the `NtGlobalFlag` global system variable. During debugging, such flags are set in the `NtGlobalFlag` field:

C++

```
 FLG_HEAP_ENABLE_TAIL_CHECK (0x10)
FLG_HEAP_ENABLE_FREE_CHECK (0x20)
FLG_HEAP_VALIDATE_PARAMETERS (0x40)
```

To check if a process has been started with a debugger, check the value of the NtGlobalFlag field in the PEB structure. This field is located by the 0x068 and 0x0bc offset for the x32 and x64 systems respectively relative to the beginning of the PEB structure.

C++

```
 0:000> dt _PEB NtGlobalFlag @$peb
ntdll!_PEB
   +0x068 NtGlobalFlag : 0x70
```

For x64 process:

C++

```
 0:000> dt _PEB NtGlobalFlag @$peb
ntdll!_PEB
   +0x0bc NtGlobalFlag : 0x70
```

The following piece of code is an example of anti debugging protection based on the `NtGlobalFlag` flags check:

C++

```
 #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_PAR
void CheckNtGlobalFlag()
{
    PVOID pPeb = GetPEB();
    PVOID pPeb64 = GetPEB64();
    DWORD offsetNtGlobalFlag = 0;
#ifdef _WIN64
    offsetNtGlobalFlag = 0xBC;
#else
```

```
    offsetNtGlobalFlag = 0x68;
#endif
    DWORD NtGlobalFlag = *(PDWORD)((PBYTE)pPeb + offsetNtGlobalFlag);
    if (NtGlobalFlag & NT_GLOBAL_FLAG_DEBUGGED)
    {
        std::cout << "Stop debugging program!" << std::endl;
        exit(-1);
    }
    if (pPeb64)
    {
        DWORD NtGlobalFlagWow64 = *(PDWORD)((PBYTE)pPeb64 + 0xBC);
        if (NtGlobalFlagWow64 & NT_GLOBAL_FLAG_DEBUGGED)
        {
            std::cout << "Stop debugging program!" << std::endl;
            exit(-1);
        }
    }
}
```

How to bypass the NtGlobalFlag check

To bypass the NtGlobalFlag check, just performing reverse the actions that we took before the check; in other words, set the the `NtGlobalFlag` field of the PEB structure of the debugged process to 0 before this value is checked by the anti debugging protection.

5. Combining NtGlobalFlag checks with IMAGE_LOAD_CONFIG_DIRECTORY

The executable can include the `IMAGE_LOAD_CONFIG_DIRECTORY` structure, which contains additional configuration parameters for the system loader. This structure is not built into an executable by default, but it can be added using a patch. This structure has the `GlobalFlagsClear` field, which indicates which flags of the `NtGlobalFlag` field of the PEB structure should be reset. If an executable was initially created without the mentioned structure or with `GlobalFlagsClear = 0`, while on the disk or in the memory, the field will have a non-zero value indicating that there's a hidden debugger working. The code example below checks the `GlobalFlagsClear` field in the memory of the running process and on the disk thus illustrating one of the popular anti debugging techniques:

C++

```
 PIMAGE_NT_HEADERS GetImageNtHeaders(PBYTE pImageBase)
{
    PIMAGE_DOS_HEADER pImageDosHeader = (PIMAGE_DOS_HEADER)pImageBase;
    return (PIMAGE_NT_HEADERS)(pImageBase + pImageDosHeader->e_lfanew);
}
PIMAGE_SECTION_HEADER FindRDataSection(PBYTE pImageBase)
{
    static const std::string rdata = ".rdata";
```

```
    PIMAGE_NT_HEADERS pImageNtHeaders = GetImageNtHeaders(pImageBase);
    PIMAGE_SECTION_HEADER pImageSectionHeader = IMAGE_FIRST_SECTION(pImageNtHeaders);
    int n = 0;
    for (; n < pImageNtHeaders->FileHeader.NumberOfSections; ++n)
    {
        if (rdata == (char*)pImageSectionHeader[n].Name)
        {
            break;
        }
    }
    return &pImageSectionHeader[n];
}
void CheckGlobalFlagsClearInProcess()
{
    PBYTE pImageBase = (PBYTE)GetModuleHandle(NULL);
    PIMAGE_NT_HEADERS pImageNtHeaders = GetImageNtHeaders(pImageBase);
    PIMAGE_LOAD_CONFIG_DIRECTORY pImageLoadConfigDirectory = (PIMAGE_LOAD_CONFIG_DIRECTORY)(pImageBase
        + pImageNtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG].VirtualAddress);
    if (pImageLoadConfigDirectory->GlobalFlagsClear != 0)
    {
        std::cout << "Stop debugging program!" << std::endl;
        exit(-1);
    }
}
void CheckGlobalFlagsClearInFile()
{
    HANDLE hExecutable = INVALID_HANDLE_VALUE;
    HANDLE hExecutableMapping = NULL;
    PBYTE pMappedImageBase = NULL;
    __try
    {
        PBYTE pImageBase = (PBYTE)GetModuleHandle(NULL);
        PIMAGE_SECTION_HEADER pImageSectionHeader = FindRDataSection(pImageBase);
        TCHAR pszExecutablePath[MAX_PATH];
        DWORD dwPathLength = GetModuleFileName(NULL, pszExecutablePath, MAX_PATH);
        if (0 == dwPathLength) __leave;
        hExecutable = CreateFile(pszExecutablePath, GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL)
        if (INVALID_HANDLE_VALUE == hExecutable) __leave;
        hExecutableMapping = CreateFileMapping(hExecutable, NULL, PAGE_READONLY, 0, 0, NULL);
        if (NULL == hExecutableMapping) __leave;
        pMappedImageBase = (PBYTE)MapViewOfFile(hExecutableMapping, FILE_MAP_READ, 0, 0,
            pImageSectionHeader->PointerToRawData + pImageSectionHeader->SizeOfRawData);
        if (NULL == pMappedImageBase) __leave;
        PIMAGE_NT_HEADERS pImageNtHeaders = GetImageNtHeaders(pMappedImageBase);
        PIMAGE_LOAD_CONFIG_DIRECTORY pImageLoadConfigDirectory = (PIMAGE_LOAD_CONFIG_DIRECTORY)(pMappedImageBase
            + (pImageSectionHeader->PointerToRawData
                + (pImageNtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG].VirtualAddre
```

```
        if (pImageLoadConfigDirectory->GlobalFlagsClear != 0)
        {
            std::cout << "Stop debugging program!" << std::endl;
            exit(-1);
        }
    }
    __finally
    {
        if (NULL != pMappedImageBase)
            UnmapViewOfFile(pMappedImageBase);
        if (NULL != hExecutableMapping)
            CloseHandle(hExecutableMapping);
        if (INVALID_HANDLE_VALUE != hExecutable)
            CloseHandle(hExecutable);
    }
}
```

In this code sample, the `CheckGlobalFlagsClearInProcess` function finds the `PIMAGE_LOAD_CONFIG_DIRECTORY` structure by the loading address of the currently running process and checks the value of the `GlobalFlagsClear` field. If this value is not 0, then the process is likely being debugged. The `CheckGlobalFlagsClearInFile` function performs the same check but for the executable on the disk.

6. Verifying Heap Flags and ForceFlags for debugger detection

The PEB structure contains a pointer to the process heap (the `_HEAP` structure):

C++

```
 0:000> dt _PEB ProcessHeap @$peb
ntdll!_PEB
   +0x018 ProcessHeap : 0x00440000 Void
0:000> dt _HEAP Flags ForceFlags 00440000
ntdll!_HEAP
   +0x040 Flags      : 0x40000062
   +0x044 ForceFlags : 0x40000060
```

For x64:

C++

```
 0:000> dt _PEB ProcessHeap @$peb
ntdll!_PEB
   +0x030 ProcessHeap : 0x0000009d`94b60000 Void
0:000> dt _HEAP Flags ForceFlags 0000009d`94b60000
ntdll!_HEAP
```

```
   +0x070 Flags      : 0x40000062
   +0x074 ForceFlags : 0x40000060
```

If the process is being debugged, both the `Flags` and `ForceFlags` fields will have specific debug values:

1. If the `Flags` field does not have the `HEAP_GROWABLE (0x00000002)` flag set, then the process is being debugged.
2. If the value of `ForceFlags is not 0`, then the process is being debugged.

It's worth mentioning that the `_HEAP` structure is undocumented and that the values of offsets of the `Flags` and `ForceFlags` fields can differ depending on the operating system version. The following code shows an anti-debugging protection example based on the heap flag check:

C++

```
 int GetHeapFlagsOffset(bool x64)
{
    return x64 ?
        IsVistaOrHigher() ? 0x70 : 0x14: //x64 offsets
        IsVistaOrHigher() ? 0x40 : 0x0C; //x86 offsets
}
int GetHeapForceFlagsOffset(bool x64)
{
    return x64 ?
        IsVistaOrHigher() ? 0x74 : 0x18: //x64 offsets
        IsVistaOrHigher() ? 0x44 : 0x10; //x86 offsets
}
void CheckHeap()
{
    PVOID pPeb = GetPEB();
    PVOID pPeb64 = GetPEB64();
    PVOID heap = 0;
    DWORD offsetProcessHeap = 0;
    PDWORD heapFlagsPtr = 0, heapForceFlagsPtr = 0;
    BOOL x64 = FALSE;
#ifdef _WIN64
    x64 = TRUE;
    offsetProcessHeap = 0x30;
#else
    offsetProcessHeap = 0x18;
#endif
    heap = (PVOID)*(PDWORD_PTR)((PBYTE)pPeb + offsetProcessHeap);
    heapFlagsPtr = (PDWORD)((PBYTE)heap + GetHeapFlagsOffset(x64));
    heapForceFlagsPtr = (PDWORD)((PBYTE)heap + GetHeapForceFlagsOffset(x64));
    if (*heapFlagsPtr & ~HEAP_GROWABLE || *heapForceFlagsPtr != 0)
    {
```

```
        std::cout << "Stop debugging program!" << std::endl;
        exit(-1);
    }
    if (pPeb64)
    {
        heap = (PVOID)*(PDWORD_PTR)((PBYTE)pPeb64 + 0x30);
        heapFlagsPtr = (PDWORD)((PBYTE)heap + GetHeapFlagsOffset(true));
        heapForceFlagsPtr = (PDWORD)((PBYTE)heap + GetHeapForceFlagsOffset(true));
        if (*heapFlagsPtr & ~HEAP_GROWABLE || *heapForceFlagsPtr != 0)
        {
            std::cout << "Stop debugging program!" << std::endl;
            exit(-1);
        }
    }
}
```

How to bypass the Heap Flags and ForceFlags checks

To bypass anti-debugging protection based on the heap flag check, set the `HEAP_GROWABLE` flag for the `Flags` field as well as the value of the `ForceFlags` field to 0. Obviously, these field values should be redefined before the heap flag check.

7. Performing a Trap Flag check to catch single-step debugging

The Trap Flag (TF) is inside the EFLAGS register. If TF is set to 1, the CPU will generate INT 01h, or the “Single Step” exception after each instruction execution. The following anti-debugging example is based on the TF setting and exception call check:

C++

```
 BOOL isDebugged = TRUE;
__try
{
    __asm
    {
        pushfd
        or dword ptr[esp], 0x100 // set the Trap Flag
        popfd                    // Load the value into EFLAGS register
        nop
    }
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
    // If an exception has been raised – debugger is not present
    isDebugged = FALSE;
```

```
}
if (isDebugged)
{
    std::cout << "Stop debugging program!" << std::endl;
    exit(-1);
}
```

Here, TF is intentionally set to generate an exception. If the process is being debugged, the exception will be caught by the debugger.

How to bypass the TF check

To neutralize the TF flag check during debugging, pass the pushfd instruction not by single-stepping but by jumping over it, putting the breakpoint after it, and continuing the program execution. After the breakpoint, tracing can be continued.

8. Calling CheckRemoteDebuggerPresent and NtQueryInformationProcess

Unlike the `IsDebuggerPresent` function, CheckRemoteDebuggerPresent checks if a process is being debugged by another parallel process. Here's an example of anti-debugging technique based on `CheckRemoteDebuggerPresent`:

C++

```
 int main(int argc, char *argv[])
{
    BOOL isDebuggerPresent = FALSE;
    if (CheckRemoteDebuggerPresent(GetCurrentProcess(), &isDebuggerPresent ))
    {
        if (isDebuggerPresent )
        {
            std::cout << "Stop debugging program!" << std::endl;
            exit(-1);
        }
    }
    return 0;
}
```

Inside `CheckRemoteDebuggerPresent`, the `NtQueryInformationProcess` function is called:

ShellScript

```
 0:000> uf kernelbase!CheckRemotedebuggerPresent
KERNELBASE!CheckRemoteDebuggerPresent:
...
```

```
75207a24 6a00            push    0
75207a26 6a04            push    4
75207a28 8d45fc          lea     eax,[ebp-4]
75207a2b 50              push    eax
75207a2c 6a07            push    7
75207a2e ff7508          push    dword ptr [ebp+8]
75207a31 ff151c602775    call    dword ptr [KERNELBASE!_imp__NtQueryInformationProcess (7527601c)]
75207a37 85c0            test    eax,eax
75207a39 0f88607e0100    js      KERNELBASE!CheckRemoteDebuggerPresent+0x2b (7521f89f)
...
```

If we take a look at the NtQueryInformationProcess documentation, this Assembler listing will show us that the `CheckRemoteDebuggerPresent` function is assigned the `DebugPort` value, as the `ProcessInformationClass` parameter value (the second one) is 7. The following anti-debugging code example is based on calling the `NtQueryInformationProcess`:

C++

```
 typedef NTSTATUS(NTAPI *pfnNtQueryInformationProcess)(
    _In_      HANDLE           ProcessHandle,
    _In_      UINT             ProcessInformationClass,
    _Out_     PVOID            ProcessInformation,
    _In_      ULONG            ProcessInformationLength,
    _Out_opt_ PULONG           ReturnLength
    );
const UINT ProcessDebugPort = 7;
int main(int argc, char *argv[])
{
    pfnNtQueryInformationProcess NtQueryInformationProcess = NULL;
    NTSTATUS status;
    DWORD isDebuggerPresent = 0;
    HMODULE hNtDll = LoadLibrary(TEXT("ntdll.dll"));

    if (NULL != hNtDll)
    {
        NtQueryInformationProcess = (pfnNtQueryInformationProcess)GetProcAddress(hNtDll, "NtQueryInformationProc
        if (NULL != NtQueryInformationProcess)
        {
            status = NtQueryInformationProcess(
                GetCurrentProcess(),
                ProcessDebugPort,
                &isDebuggerPresent,
                sizeof(DWORD),
                NULL);
            if (status == 0x00000000 && isDebuggerPresent != 0)
            {
```

```
                std::cout << "Stop debugging program!" << std::endl;
                exit(-1);
            }
        }
    }
    return 0;
}
```

How to bypass CheckRemoteDebuggerPresent and NtQueryInformationProcess

To bypass CheckRemoteDebuggerPresent and NTQueryInformationProcess, substitute the value returned by the `NtQueryInformationProcess` function. You can use mhook to do this. To set up a hook, inject DLL into the debugged process and set up a hook in DLLMain using mhook. Here's an example of mhook in use:

C++

```
 #include <Windows.h>
#include "mhook.h"
typedef NTSTATUS(NTAPI *pfnNtQueryInformationProcess)(
    _In_      HANDLE           ProcessHandle,
    _In_      UINT             ProcessInformationClass,
    _Out_     PVOID            ProcessInformation,
    _In_      ULONG            ProcessInformationLength,
    _Out_opt_ PULONG           ReturnLength
    );
const UINT ProcessDebugPort = 7;
pfnNtQueryInformationProcess g_origNtQueryInformationProcess = NULL;
NTSTATUS NTAPI HookNtQueryInformationProcess(
    _In_      HANDLE           ProcessHandle,
    _In_      UINT             ProcessInformationClass,
    _Out_     PVOID            ProcessInformation,
    _In_      ULONG            ProcessInformationLength,
    _Out_opt_ PULONG           ReturnLength
    )
{
    NTSTATUS status = g_origNtQueryInformationProcess(
        ProcessHandle,
        ProcessInformationClass,
        ProcessInformation,
        ProcessInformationLength,
        ReturnLength);
    if (status == 0x00000000 && ProcessInformationClass == ProcessDebugPort)
    {
        *((PDWORD_PTR)ProcessInformation) = 0;
    }
    return status;
```

```
}
DWORD SetupHook(PVOID pvContext)
{
    HMODULE hNtDll = LoadLibrary(TEXT("ntdll.dll"));
    if (NULL != hNtDll)
    {
        g_origNtQueryInformationProcess = (pfnNtQueryInformationProcess)GetProcAddress(hNtDll, "NtQueryInformat
        if (NULL != g_origNtQueryInformationProcess)
        {
            Mhook_SetHook((PVOID*)&g_origNtQueryInformationProcess, HookNtQueryInformationProcess);
        }
    }
    return 0;
}
BOOL WINAPI DllMain(HINSTANCE hInstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
    switch (fdwReason)
    {
    case DLL_PROCESS_ATTACH:
        DisableThreadLibraryCalls(hInstDLL);
        CreateThread(NULL, NULL, (LPTHREAD_START_ROUTINE)SetupHook, NULL, NULL, NULL);
        Sleep(20);
    case DLL_PROCESS_DETACH:
        if (NULL != g_origNtQueryInformationProcess)
        {
            Mhook_Unhook((PVOID*)&g_origNtQueryInformationProcess);
        }
        break;
    }
    return TRUE;
}
```

9. Other anti-debugging techniques based on NtQueryInformationProcess

There are several techniques for debugger detection using information provided by the `NtQueryInformationProcess` function:

1. `ProcessDebugPort 0x07` – discussed above
2. `ProcessDebugObjectHandle 0x1E`
3. `ProcessDebugFlags 0x1F`
4. `ProcessBasicInformation 0x00`

Let’s consider numbers two through four in detail.

ProcessDebugObjectHandle

Starting with Windows XP, a “debug object” is created for a debugged process. Here’s an example of checking for a “debug object” in the current process:

C++

```
 status = NtQueryInformationProcess(
            GetCurrentProcess(),
            ProcessDebugObjectHandle,
            &hProcessDebugObject,
            sizeof(HANDLE),
            NULL);
if (0x00000000 == status && NULL != hProcessDebugObject)
{
    std::cout << "Stop debugging program!" << std::endl;
    exit(-1);
}
```

If a debug object exists, then the process is being debugged.

ProcessDebugFlags

When checking this flag, it returns the inverse value of the `NoDebugInherit` bit of the `EPROCESS` kernel structure. If the returned value of the `NtQueryInformationProcess` function is 0, then the process is being debugged. Here’s an example of such an anti-debugging check:

C++

```
 status = NtQueryInformationProcess(
    GetCurrentProcess(),
    ProcessDebugObjectHandle,
    &debugFlags,
    sizeof(ULONG),
    NULL);
if (0x00000000 == status && NULL != debugFlags)
{
    std::cout << "Stop debugging program!" << std::endl;
    exit(-1);
}
```

ProcessBasicInformation

When calling the `NtQueryInformationProcess` function with the `ProcessBasicInformation` flag, the `PROCESS_BASIC_INFORMATION` structure is returned:

C++

```
 typedef struct _PROCESS_BASIC_INFORMATION {
    NTSTATUS ExitStatus;
    PVOID PebBaseAddress;
    ULONG_PTR AffinityMask;
    KPRIORITY BasePriority;
    HANDLE UniqueProcessId;
    HANDLE InheritedFromUniqueProcessId;
} PROCESS_BASIC_INFORMATION, *PPROCESS_BASIC_INFORMATION;
```

The most interesting thing in this structure is the `InheritedFromUniqueProcessId` field. Here, we need to get the name of the parent process and compare it to the names of popular debuggers. Let's see how to create an anti-debugger:

C++

```
 std::wstring GetProcessNameById(DWORD pid)
{
    HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (hProcessSnap == INVALID_HANDLE_VALUE)
    {
        return 0;
    }
    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(PROCESSENTRY32);
    std::wstring processName = L"";
    if (!Process32First(hProcessSnap, &pe32))
    {
        CloseHandle(hProcessSnap);
        return processName;
    }
    do
    {
        if (pe32.th32ProcessID == pid)
        {
            processName = pe32.szExeFile;
            break;
        }
    } while (Process32Next(hProcessSnap, &pe32));

    CloseHandle(hProcessSnap);
    return processName;
}
status = NtQueryInformationProcess(
```

```
    GetCurrentProcess(),
    ProcessBasicInformation,
    &processBasicInformation,
    sizeof(PROCESS_BASIC_INFORMATION),
    NULL);
std::wstring parentProcessName = GetProcessNameById((DWORD)processBasicInformation.InheritedFromUniqueProcessId)
if (L"devenv.exe" == parentProcessName)
{
    std::cout << "Stop debugging program!" << std::endl;
    exit(-1);
}
```

How to bypass the NtQueryInformationProcess checks

Bypassing the NtQueryInformation Process checks is simple. Just change the values returned by the `NtQueryInformationProcess` function to values that don't indicate the presence of a debugger:

1. Set `ProcessDebugObjectHandle` to 0
2. Set `ProcessDebugFlags` to 1
3. For `ProcessBasicInformation`, change the `InheritedFromUniqueProcessId` value to the ID of another process, e.g. explorer.exe

10. Detecting software and hardware breakpoints

Breakpoints are the main tool provided by debuggers. Breakpoints allow you to interrupt program execution at a specified place. There are two types of breakpoints:

1. Software breakpoints
2. Hardware breakpoints

It's very hard to reverse engineer software without breakpoints. Popular anti-reverse engineering tactics are based on detecting breakpoints, providing a series of corresponding anti-debugging methods.

Software Breakpoints

In the IA-32 architecture, there's a specific instruction – int 3h with the 0xCC opcode – that is used to call the debug handle. When the CPU executes this instruction, an interruption is generated and control is transferred to the debugger. To get control, the debugger has to inject the int 3h instruction into the code. To detect a breakpoint, we can calculate the checksum of the function. Here's an example:

C++

```
 DWORD CalcFuncCrc(PUCHAR funcBegin, PUCHAR funcEnd)
{
    DWORD crc = 0;
```

```
    for (; funcBegin < funcEnd; ++funcBegin)
    {
        crc += *funcBegin;
    }
    return crc;
}
#pragma auto_inline(off)
VOID DebuggeeFunction()
{
    int calc = 0;
    calc += 2;
    calc <<= 8;
    calc -= 3;
}
VOID DebuggeeFunctionEnd()
{
};
#pragma auto_inline(on)
DWORD g_origCrc = 0x2bd0;
int main()
{
    DWORD crc = CalcFuncCrc((PUCHAR)DebuggeeFunction, (PUCHAR)DebuggeeFunctionEnd);
    if (g_origCrc != crc)
    {
        std::cout << "Stop debugging program!" << std::endl;
        exit(-1);
    }
    return 0;
}
```

It is worth mentioning that this will only work if the `/INCREMENTAL:NO` linker option is set, otherwise, when getting the function address to calculate the checksum, we'll get the relative jump address:

ShellScript

```
 DebuggeeFunction:
013C16DB  jmp         DebuggeeFunction (013C4950h)
```

The `g_origCrc` global variable contains crc already calculated by the `CalcFuncCrc` function. To detect the end of the function, we use the stub function trick. As the function code is located sequentially, the end of the `DebuggeeFunction` is the beginning of the `DebuggeeFunctionEnd` function. We also used the `#pragma auto_inline(off)` directive to prevent the compiler from making functions embedded.

How to bypass a software breakpoint check

There’s no universal approach for bypassing a software breakpoint check. To bypass this protection, you should find the code calculating the checksum and substitute the returned value with a constant, as well as the values of all variables storing function checksums.

Hardware Breakpoints

In the x86 architecture, there’s a set of debug registers used by developers when checking and debugging code. These registers allow you to interrupt program execution and transfer control to a debugger when accessing memory to read or write. Debug registers are a privileged resource and can be used by a program only in real mode or safe mode with privilege level CPL=0. There are eight debug registers DR0-DR7:

1. DR0-DR3 – breakpoint registers
2. DR4 & DR5 – reserved
3. DR6 – debug status
4. DR7 – debug control

DR0-DR3 contain linear addresses of breakpoints. Comparison of these addresses is performed before physical address translation. Each of these breakpoints is separately described in the DR7 register. The DR6 register indicates which breakpoint is activated. DR7 defines the breakpoint activation mode by the access mode: read, write, or execute. Here’s an example of a hardware breakpoint check:

C++

```
 CONTEXT ctx = {};
ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
if (GetThreadContext(GetCurrentThread(), &ctx))
{
    if (ctx.Dr0 != 0 || ctx.Dr1 != 0 || ctx.Dr2 != 0 || ctx.Dr3 != 0)
    {
        std::cout << "Stop debugging program!" << std::endl;
        exit(-1);
    }
}
```

It’s also possible to reset hardware breakpoints by means of the `SetThreadContext` function. Here’s an example of a hardware breakpoint reset:

C++

```
 CONTEXT ctx = {};
ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
SetThreadContext(GetCurrentThread(), &ctx);
```

As we can see, all DRx registers are set to 0.

How to bypass a hardware breakpoint check and reset

If we look inside the `GetThreadContext` function, we'll see that it calls the `NtGetContextThread` function:

ShellScript

```
 0:000> u KERNELBASE!GetThreadContext L6
KERNELBASE!GetThreadContext:
7538d580 8bff            mov     edi,edi
7538d582 55              push    ebp
7538d583 8bec            mov     ebp,esp
7538d585 ff750c          push    dword ptr [ebp+0Ch]
7538d588 ff7508          push    dword ptr [ebp+8]
7538d58b ff1504683975    call    dword ptr [KERNELBASE!_imp__NtGetContextThread (75396804)]
```

To make the protection receive zero values in Dr0-Dr7, reset the `CONTEXT_DEBUG_REGISTERS` flag in the `ContextFlags` field of the `CONTEXT` structure and then restore its value after the original `NtGetContextThread` function call. As for the `GetThreadContext` function, it calls `NtSetContextThread`. The following example shows how to bypass a hardware breakpoint check and reset:

C++

```
 typedef NTSTATUS(NTAPI *pfnNtGetContextThread)(
    _In_  HANDLE             ThreadHandle,
    _Out_ PCONTEXT           pContext
    );
typedef NTSTATUS(NTAPI *pfnNtSetContextThread)(
    _In_ HANDLE              ThreadHandle,
    _In_ PCONTEXT            pContext
    );
pfnNtGetContextThread g_origNtGetContextThread = NULL;
pfnNtSetContextThread g_origNtSetContextThread = NULL;
NTSTATUS NTAPI HookNtGetContextThread(
    _In_  HANDLE              ThreadHandle,
    _Out_ PCONTEXT            pContext)
{
    DWORD backupContextFlags = pContext->ContextFlags;
    pContext->ContextFlags &= ~CONTEXT_DEBUG_REGISTERS;
    NTSTATUS status = g_origNtGetContextThread(ThreadHandle, pContext);
    pContext->ContextFlags = backupContextFlags;
    return status;
}
NTSTATUS NTAPI HookNtSetContextThread(
    _In_ HANDLE              ThreadHandle,
    _In_ PCONTEXT            pContext)
```

```
{
    DWORD backupContextFlags = pContext->ContextFlags;
    pContext->ContextFlags &= ~CONTEXT_DEBUG_REGISTERS;
    NTSTATUS status = g_origNtSetContextThread(ThreadHandle, pContext);
    pContext->ContextFlags = backupContextFlags;
    return status;
}
void HookThreadContext()
{
         HMODULE hNtDll = LoadLibrary(TEXT("ntdll.dll"));
         g_origNtGetContextThread = (pfnNtGetContextThread)GetProcAddress(hNtDll, "NtGetContextThread");
         g_origNtSetContextThread = (pfnNtSetContextThread)GetProcAddress(hNtDll, "NtSetContextThread");
         Mhook_SetHook((PVOID*)&g_origNtGetContextThread, HookNtGetContextThread);
         Mhook_SetHook((PVOID*)&g_origNtSetContextThread, HookNtSetContextThread);
}
```

11. Using Structured Exception Handling (SEH) for debugger detection

Structured exception handling is a mechanism provided by the operating system to an application allowing it to receive notifications about exceptional situations like division by zero, reference to a nonexistent pointer, or execution of a restricted instruction. This mechanism allows you to handle exceptions inside an application, without operating system involvement. If an exception is not handled, it will result in abnormal program termination. Developers usually locate pointers to SEH in the stack, which are called SEH frames. The current SEH frame address is located by the 0 offset relative to the FS selector (or GS selector for the x64 systems). This address points to the ntdll! `_EXCEPTION_REGISTRATION_RECORD` structure:

ShellScript

```
0:000> dt ntdll!_EXCEPTION_REGISTRATION_RECORD
  +0x000 Next             : Ptr32 _EXCEPTION_REGISTRATION_RECORD
  +0x004 Handler          : Ptr32 _EXCEPTION_DISPOSITION
```

When an exception is initiated, control is transferred to the current SEH handler. Depending on the situation, this SEH handler should return one of the `_EXCEPTION_DISPOSITION` members:

C++

```
typedef enum _EXCEPTION_DISPOSITION {
   ExceptionContinueExecution,
   ExceptionContinueSearch,
   ExceptionNestedException,
   ExceptionCollidedUnwind
} EXCEPTION_DISPOSITION;
```

If the handler returns `ExceptionContinueSearch`, the system continues execution from the instruction, that triggered the exception. If the handler doesn't know what to do with an exception, it returns `ExceptionContinueSearch` and then the system moves to the next handler in the chain. You can browse the current exception chain using the `!exchain` command in the WinDbg debugger:

ShellScript

```
 0:000> !exchain
00a5f3bc: AntiDebug!_except_handler4+0 (008b7530)
  CRT scope  0, filter: AntiDebug!SehInternals+67 (00883d67)
                func:   AntiDebug!SehInternals+6d (00883d6d)
00a5f814: AntiDebug!__scrt_stub_for_is_c_termination_complete+164b (008bc16b)
00a5f87c: AntiDebug!_except_handler4+0 (008b7530)
  CRT scope  0, filter: AntiDebug!__scrt_common_main_seh+1b0 (008b7c60)
                func:   AntiDebug!__scrt_common_main_seh+1cb (008b7c7b)
00a5f8e8: ntdll!_except_handler4+0 (775674a0)
  CRT scope  0, filter: ntdll!__RtlUserThreadStart+54386 (7757f076)
                func:   ntdll!__RtlUserThreadStart+543cd (7757f0bd)
00a5f900: ntdll!FinalExceptionHandlerPad4+0 (77510213)
```

The last in the chain is the default handler assigned by the system. If none of the previous handlers managed to handle the exception, then the system handler goes to the registry to get the key

```
HKEY_LOCAL_MACHINESoftwareMicrosoftWindows NTCurrentVersionAeDebug
```

Depending on the `AeDebug` key value, either the application is terminated or control is transferred to the debugger. The debugger path should be indicated in `Debugger REG_SZ`.

When creating a new process, the system adds the primary SEH frame to it. The handler for the primary SEH frame is also defined by the system. The primary SEH frame itself is located almost at the very beginning of the memory stack allocated for the process. The SEH handler function signature looks as follows:

C++

```
typedef EXCEPTION_DISPOSITION (*PEXCEPTION_ROUTINE) (
   __in struct _EXCEPTION_RECORD *ExceptionRecord,
   __in PVOID EstablisherFrame,
   __inout struct _CONTEXT *ContextRecord,
   __inout PVOID DispatcherContext
   );
```

If an application is being debugged, after the int 3h interruption generation, control will be intercepted by the debugger. Otherwise, control will be transferred to the SEH handler. The following code example shows SEH frame-based anti-debugging protection:

C++

```
 BOOL g_isDebuggerPresent = TRUE;
EXCEPTION_DISPOSITION ExceptionRoutine(
    PEXCEPTION_RECORD ExceptionRecord,
    PVOID             EstablisherFrame,
    PCONTEXT          ContextRecord,
    PVOID             DispatcherContext)
{
    g_isDebuggerPresent = FALSE;
    ContextRecord->Eip += 1;
    return ExceptionContinueExecution;
}
int main()
{
    __asm
    {
        // set SEH handler
        push ExceptionRoutine
        push dword ptr fs:[0]
        mov  dword ptr fs:[0], esp
        // generate interrupt
        int  3h
        // return original SEH handler
        mov  eax, [esp]
        mov  dword ptr fs:[0], eax
        add  esp, 8
    }
    if (g_isDebuggerPresent)
    {
        std::cout << "Stop debugging program!" << std::endl;
        exit(-1);
    }
    return 0
}
```

In this example, an SEH handler is set. The pointer to this handler is put at the beginning of the handler chain. Then the int 3h interruption is generated. If the application is not being debugged, control will be transferred to the SEH handler and the value of `g_isDebuggerPresent` will be set to `FALSE`. The `ContextRecord->Eip += 1` line also changes the address of the next instruction in the execution flow, which will result in the execution of the instruction after int 3h. Then the code returns the original SEH handler, clears the stack, and checks if a debugger is present.

How to bypass SEH checks

There’s no universal approach to bypassing SEH checks, but still there are some techniques to make a reverser’s life easier. Let’s take a look at the call stack that led to the SEH handler call:

ShellScript

```
 0:000> kn
 # ChildEBP RetAddr
00 0059f06c 775100b1 AntiDebug!ExceptionRoutine
01 0059f090 77510083 ntdll!ExecuteHandler2+0x26
02 0059f158 775107ff ntdll!ExecuteHandler+0x24
03 0059f158 003b11a5 ntdll!KiUserExceptionDispatcher+0xf
04 0059fa90 003d7f4e AntiDebug!main+0xb5
05 0059faa4 003d7d9a AntiDebug!invoke_main+0x1e
06 0059fafc 003d7c2d AntiDebug!__scrt_common_main_seh+0x15a
07 0059fb04 003d7f68 AntiDebug!__scrt_common_main+0xd
08 0059fb0c 753e7c04 AntiDebug!mainCRTStartup+0x8
09 0059fb20 7752ad1f KERNEL32!BaseThreadInitThunk+0x24
0a 0059fb68 7752acea ntdll!__RtlUserThreadStart+0x2f
0b 0059fb78 00000000 ntdll!_RtlUserThreadStart+0x1b
```

We can see that this call came from `ntdll!ExecuteHandler2`. This function is the starting point for calling any SEH handler. A breakpoint can be set at the call instruction:

ShellScript

```
 0:000> u ntdll!ExecuteHandler2+24 L3
ntdll!ExecuteHandler2+0x24:
775100af ffd1            call    ecx
775100b1 648b2500000000  mov     esp,dword ptr fs:[0]
775100b8 648f0500000000  pop     dword ptr fs:[0]
0:000> bp 775100af
```

After setting the breakpoint, you should analyze the code of each called SEH handler. If protection is based on multiple calls to SEH handlers, a reverser will really sweat over bypassing it.

12. Using Vectored Exception Handlers (VEH) to detect debuggers

VEH was introduced in Windows XP and is a variation of SEH. VEH and SEH do not depend on each other and work simultaneously. When a new VEH handler is added, the SEH chain is not affected as the list of VEH handlers is stored in the `ntdll!LdrpVectorHandlerList` non-exported variable. The VEH and SEH mechanisms are pretty similar, the only difference being that documented functions are used to set up and remove a VEH handler. The signatures of the functions for adding and removing a VEH handler as well as the VEH handler function itself are as follows:

C++

```
 PVOID WINAPI AddVectoredExceptionHandler(
    ULONG                       FirstHandler,
    PVECTORED_EXCEPTION_HANDLER VectoredHandler
);
ULONG WINAPI RemoveVectoredExceptionHandler(
    PVOID Handler
);
LONG CALLBACK VectoredHandler(
    PEXCEPTION_POINTERS ExceptionInfo
);
The _EXCEPTION_POINTERS structure looks like this:
typedef struct _EXCEPTION_POINTERS {
  PEXCEPTION_RECORD ExceptionRecord;
  PCONTEXT          ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

After receiving control in the handler, the system collects the current process context and passes it via the `ContextRecord` parameter. Here's a sample of anti-debugging protection code using vector exception handling:

C++

```
 LONG CALLBACK ExceptionHandler(PEXCEPTION_POINTERS ExceptionInfo)
{
    PCONTEXT ctx = ExceptionInfo->ContextRecord;
    if (ctx->Dr0 != 0 || ctx->Dr1 != 0 || ctx->Dr2 != 0 || ctx->Dr3 != 0)
    {
        std::cout << "Stop debugging program!" << std::endl;
        exit(-1);
    }
    ctx->Eip += 2;
    return EXCEPTION_CONTINUE_EXECUTION;
}
int main()
{
    AddVectoredExceptionHandler(0, ExceptionHandler);
    __asm int 1h;
    return 0;
}
```

Here we set up a VEH handler and generated an interruption (int 1h is not necessary). When the interruption is generated, an exception appears and control is transferred to the VEH handler. If a hardware breakpoint is set,

program execution is stopped. If there are no hardware breakpoints, the EIP register value is increased by 2 to continue execution after the int 1h generation instruction.

How to bypass a hardware breakpoint check and VEH

Let’s take a look at the call stack that led to the VEH handler:

ShellScript

```
 0:000> kn
 # ChildEBP RetAddr
00 001cf21c 774d6822 AntiDebug!ExceptionHandler
01 001cf26c 7753d151 ntdll!RtlpCallVectoredHandlers+0xba
02 001cf304 775107ff ntdll!RtlDispatchException+0x72
03 001cf304 00bf4a69 ntdll!KiUserExceptionDispatcher+0xf
04 001cfc1c 00c2680e AntiDebug!main+0x59
05 001cfc30 00c2665a AntiDebug!invoke_main+0x1e
06 001cfc88 00c264ed AntiDebug!__scrt_common_main_seh+0x15a
07 001cfc90 00c26828 AntiDebug!__scrt_common_main+0xd
08 001cfc98 753e7c04 AntiDebug!mainCRTStartup+0x8
09 001cfcac 7752ad1f KERNEL32!BaseThreadInitThunk+0x24
0a 001cfcf4 7752acea ntdll!__RtlUserThreadStart+0x2f
0b 001cfd04 00000000 ntdll!_RtlUserThreadStart+0x1b
```

As we can see, control was transferred from `main+0x59` to `ntdll!KiUserExceptionDispatcher`. Let’s see what instruction in `main+0x59` resulted in this call:

ShellScript

```
 0:000> u main+59 L1
AntiDebug!main+0x59
00bf4a69 cd02            int     1
```

Here’s the instruction that generated the interruption. The `KiUserExceptionDispatcher` function is one of the callbacks that the system calls from kernel mode to user mode. This is its signature:

C++

```
 VOID NTAPI KiUserExceptionDispatcher(
    PEXCEPTION_RECORD pExcptRec,
    PCONTEXT ContextFrame
);
```

The next code sample Shows how to bypass the hardware breakpoint check by applying the `KiUserExceptionDispatcher` function hook:

C++

```
 typedef  VOID (NTAPI *pfnKiUserExceptionDispatcher)(
    PEXCEPTION_RECORD pExcptRec,
    PCONTEXT ContextFrame
    );
pfnKiUserExceptionDispatcher g_origKiUserExceptionDispatcher = NULL;
VOID NTAPI HandleKiUserExceptionDispatcher(PEXCEPTION_RECORD pExcptRec, PCONTEXT ContextFrame)
{
    if (ContextFrame && (CONTEXT_DEBUG_REGISTERS & ContextFrame->ContextFlags))
    {
        ContextFrame->Dr0 = 0;
        ContextFrame->Dr1 = 0;
        ContextFrame->Dr2 = 0;
        ContextFrame->Dr3 = 0;
        ContextFrame->Dr6 = 0;
        ContextFrame->Dr7 = 0;
        ContextFrame->ContextFlags &= ~CONTEXT_DEBUG_REGISTERS;
    }
}
__declspec(naked) VOID NTAPI HookKiUserExceptionDispatcher()
// Params: PEXCEPTION_RECORD pExcptRec, PCONTEXT ContextFrame
{
    __asm
    {
        mov eax, [esp + 4]
        mov ecx, [esp]
        push eax
        push ecx
        call HandleKiUserExceptionDispatcher
        jmp g_origKiUserExceptionDispatcher
    }
}
int main()
{
    HMODULE hNtDll = LoadLibrary(TEXT("ntdll.dll"));
    g_origKiUserExceptionDispatcher = (pfnKiUserExceptionDispatcher)GetProcAddress(hNtDll, "KiUserExceptionDispa
    Mhook_SetHook((PVOID*)&g_origKiUserExceptionDispatcher, HookKiUserExceptionDispatcher);
    return 0;
}
```

It this example, the values of the DRx registers are reset in the `HookKiUserExceptionDispatcher` function, in other words before the VEH handler call.

13. Hiding threads from a debugger with NtSetInformationThread

In Windows 2000, a new class of thread information transferred to the `NtSetInformationThread` function appeared – `ThreadHideFromDebugger`. This was one of the first anti-debugging techniques provided by Windows in Microsoft's search for how to prevent reverse engineering, and it's very powerful. If this flag is set for a thread, then that thread stops sending notifications about debug events. These events include breakpoints and notifications about program completion. The value of this flag is stored in the `HideFromDebugger` field of the `_ETHREAD` structure:

ShellScript

```
 1: kd> dt _ETHREAD HideFromDebugger 86bfada8
ntdll!_ETHREAD
   +0x248 HideFromDebugger : 0y1
```

Here's an example of how to set `ThreadHideFromDebugger`:

C++

```
 typedef NTSTATUS (NTAPI *pfnNtSetInformationThread)(
    _In_ HANDLE ThreadHandle,
    _In_ ULONG  ThreadInformationClass,
    _In_ PVOID  ThreadInformation,
    _In_ ULONG  ThreadInformationLength
    );
const ULONG ThreadHideFromDebugger = 0x11;
void HideFromDebugger()
{
    HMODULE hNtDll = LoadLibrary(TEXT("ntdll.dll"));
    pfnNtSetInformationThread NtSetInformationThread = (pfnNtSetInformationThread)
        GetProcAddress(hNtDll, "NtSetInformationThread");
    NTSTATUS status = NtSetInformationThread(GetCurrentThread(),
        ThreadHideFromDebugger, NULL, 0);
}
```

How to bypass thread hiding from debugger

To prevent an application from hiding the thread from a debugger, you need to hook the `NtSetInformationThread` function call. Here's a hook code example:

C++

```
 pfnNtSetInformationThread g_origNtSetInformationThread = NULL;
NTSTATUS NTAPI HookNtSetInformationThread(
```

```
    _In_ HANDLE ThreadHandle,
    _In_ ULONG  ThreadInformationClass,
    _In_ PVOID  ThreadInformation,
    _In_ ULONG  ThreadInformationLength
    )
{
    if (ThreadInformationClass == ThreadHideFromDebugger &&
        ThreadInformation == 0 && ThreadInformationLength == 0)
    {
        return STATUS_SUCCESS;
    }
    return g_origNtSetInformationThread(ThreadHandle,
        ThreadInformationClass, ThreadInformation, ThreadInformationLength
}

void SetHook()
{
    HMODULE hNtDll = LoadLibrary(TEXT("ntdll.dll"));
    if (NULL != hNtDll)
    {
        g_origNtSetInformationThread = (pfnNtSetInformationThread)GetProcAddress(hNtDll, "NtSetInformationThread
        if (NULL != g_origNtSetInformationThread)
        {
            Mhook_SetHook((PVOID*)&g_origNtSetInformationThread, HookNtSetInformationThread);
        }
    }
}
```

In the hooked function, when calling it correctly, `STATUS_SUCCESS` will be returned without transferring control to the original `NtSetInformationThread` function.

14. Using NtCreateThreadEx to evade debugging

Windows Vista introduced the `NtCreateThreadEx` function, whose signature is as follows:

C++

```
NTSTATUS NTAPI NtCreateThreadEx (
    _Out_    PHANDLE              ThreadHandle,
    _In_     ACCESS_MASK          DesiredAccess,
    _In_opt_ POBJECT_ATTRIBUTES   ObjectAttributes,
    _In_     HANDLE               ProcessHandle,
    _In_     PVOID                StartRoutine,
    _In_opt_ PVOID                Argument,
    _In_     ULONG                CreateFlags,
    _In_opt_ ULONG_PTR            ZeroBits,
```

```
    _In_opt_ SIZE_T              StackSize,
    _In_opt_ SIZE_T              MaximumStackSize,
    _In_opt_ PVOID               AttributeList
);
```

The most interesting parameter is `CreateFlags`. This parameter gets flags such as:

C++

```
 #define THREAD_CREATE_FLAGS_CREATE_SUSPENDED 0x00000001
#define THREAD_CREATE_FLAGS_SKIP_THREAD_ATTACH 0x00000002
#define THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER 0x00000004
#define THREAD_CREATE_FLAGS_HAS_SECURITY_DESCRIPTOR 0x00000010
#define THREAD_CREATE_FLAGS_ACCESS_CHECK_IN_TARGET 0x00000020
#define THREAD_CREATE_FLAGS_INITIAL_THREAD 0x00000080
```

If a new thread gets the `THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER` flag, it will be hidden from the debugger when it's created. This is the same `ThreadHideFromDebugger`, that's set up by the `NtSetInformationThread` function. The code that's responsible for security tasks, can be executed in the thread with the `THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER` flag set.

How to bypass NtCreateThreadEx

This technique can be bypassed by hooking the `NtCreateThreadEx` function, in which `THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER` will be reset.

15. Tracing handles to spot debugger interference

Starting with Windows XP, Windows systems have had a mechanism for kernel object handle tracing. When the tracing mode is on, all operations with handlers are saved to the circular buffer. Also, when trying to use a nonexistent handler, for instance, to close it using the `CloseHandle` function, the `EXCEPTION_INVALID_HANDLE` exception will be generated. If a process is started not from the debugger, the `CloseHandle` function will return `FALSE`. The following example shows anti-debugging protection based on `CloseHandle`:

C++

```
 EXCEPTION_DISPOSITION ExceptionRoutine(
    PEXCEPTION_RECORD ExceptionRecord,
    PVOID             EstablisherFrame,
    PCONTEXT          ContextRecord,
    PVOID             DispatcherContext)
{
    if (EXCEPTION_INVALID_HANDLE == ExceptionRecord->ExceptionCode)
    {
```

```
        std::cout << "Stop debugging program!" << std::endl;
        exit(-1);
    }
    return ExceptionContinueExecution;
}
int main()
{
    __asm
    {
        // set SEH handler
        push ExceptionRoutine
        push dword ptr fs : [0]
        mov  dword ptr fs : [0], esp
    }
    CloseHandle((HANDLE)0xBAAD);
    __asm
    {
        // return original SEH handler
        mov  eax, [esp]
        mov  dword ptr fs : [0], eax
        add  esp, 8
    }
    return 0
}
```

16. Manipulating the stack segment to confuse debuggers

When manipulating the ss stack segment register, the debugger skips the instruction tracing. In the next example, the debugger will immediately move to the xor edx, edx instruction, while the previous instruction will be executed:

C++

```
 __asm
{
    push ss
    pop  ss
    mov  eax, 0xC000C1EE // This line will be traced over by debugger
    xor  edx, edx        // Debugger will step to this line
}
```

Since Windows 10, the implementation of the *OutputDebugString* function has changed to a simple *RaiseException* call with the specific parameters. So, debug output exception must be now handled by the debugger.

There are two exception types: *DBG_PRINTEXCEPTION_C (0x40010006)* and *DBG_PRINTEXCEPTION_W(0x4001000A)*, which can be used to detect the debugger presence.

C++

```
 #define DBG_PRINTEXCEPTION_WIDE_C 0x4001000A
WCHAR * outputString = L"Any text";
ULONG_PTR args[4] = {0};
args[0] = (ULONG_PTR)wcslen(outputString) + 1;
args[1] = (ULONG_PTR)outputString;
__try
{
    RaiseException(DBG_PRINTEXCEPTION_WIDE_C, 0, 4, args);
    printf("Debugger detected");
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
    printf("Debugger NOT detected");
}
```

So, in case the exception is unhandled, it means there's no debugger attached.

DBG_PRINTEXCEPTION_W is used for wide-char output, and *DBG_PRINTEXCEPTION_C* is used for ansi. In means that in case of *DBG_PRINTEXCEPTION_C args[0]* will hold strlen() result, and args[1] – the pointer to ansi string (char *).

18. Applying Protected Process Light (PPL) as an additional protection layer

While it's not strictly an anti-debugging tool, **Protected Process Light (PPL)** is a Windows security mechanism designed to prevent unauthorized interference with protected processes. PPL restricts what an operation's external processes can do, including opening handles, injecting code, or modifying memory. PPL was originally intended to protect antivirus solutions and other security-critical components.

There are some limitations that make PPL unsuitable as a universal protection method:

1. **It requires Microsoft signing.** To run a process in PPL mode, it must be signed with a special Microsoft certificate. This is typically accessible only through official partnerships, which makes PPL unrealistic for most independent developers.
2. **It is not effective against kernel-level debugging.** PPL operates entirely in user mode. A kernel debugger or driver with sufficient privileges can still access and manipulate a PPL-protected process.
3. **It is not designed specifically for anti-debugging.** PPL restricts some forms of interference, but it doesn't provide the detection or evasion mechanisms commonly expected from dedicated anti-debugging techniques.

As you can see, PPL offers an additional hardening layer for products already operating within the Windows security ecosystem. On the other hand, in the case of typical commercial software, it is difficult to adopt and should not be considered standalone protection against debugging or reverse engineering.

Protect your software from unauthorized reverse engineering with Apriorit

Don't think of modern anti-reverse engineering techniques as a set of isolated tricks. Anti-reverse engineering is a strategic layer that helps safeguard your intellectual property, protect sensitive logic, and maintain product integrity. At Apriorit, we bring together more than two decades of experience in **cybersecurity, low-level development,** and **ethical reverse engineering** to help companies build resilient products that can withstand increasingly sophisticated analysis attempts.

When you partner with Apriorit, you gain a team that is experienced in:

- **Implementing various anti-debugging techniques**, from classic debugger detection to advanced multi-layered protection
- **Improving existing software with additional security layers**, including code obfuscation, integrity checks, anti-tampering mechanisms, and environment detection
- **Auditing and strengthening current protections**, identifying weak points in your defensive logic, and recommending improvements tailored to your architecture
- **Integrating protection into the development lifecycle**, making sure that security features work reliably across platforms and don't disrupt performance or maintainability

Apriorit specialists continually track evolving reverse-engineering and debugging techniques, allowing us to design solutions that resist both common tools and expert-level analysis. We'll help you secure a new product or reinforce an existing one by building a robust protection strategy.

Conclusion

Safeguarding your software from malicious reverse engineering activities is complex and tricky. To mitigate such threats, your team must acquire strong cybersecurity knowledge, apply the most suitable tools, and use proven techniques. One valuable technique against malicious activity is anti-debugging. It helps your team make reversers' lives harder and complicate their work as much as possible.

To get the most out of anti-debugging techniques, involve professional developers with relevant skills. At Apriorit, we have expert reverse engineering teams with extensive knowledge and experience in successfully applying anti-debugging protection methods. We are ready to help you safeguard your software, enhance project security, and solve non-trivial tasks.

FAQ

Anti-debugging techniques are protective mechanisms built into software to spot and interfere with debugging tools. Depending on the approach, they might:

- Verify process states

- Inspect system structures
- Trigger unusual execution flows
- Exploit quirks of debugging environments to reveal or disrupt a debugger’s presence

These methods help prevent analysts or attackers from stepping through code or observing internal behavior.

Anti-debugging and anti-tampering have different goals, but they are usually combined to create a more resilient layer of protection against illegal reverse engineering. Anti-debugging focuses on detecting or obstructing tools that inspect a program at runtime. Anti-tampering techniques, meanwhile, are designed to expose or block attempts to modify the application’s code or memory.

A debugger attaches to a running process and communicates with the operating system to control it. With the help of this connection, the debugger can:

- Pause process execution
- Inspect variables and memory
- Set breakpoints
- Step through instructions

Developers use this visibility to troubleshoot issues, but attackers can use the same capabilities to study and manipulate an application.

There are many popular approaches, including:

- API checks
- PEB flag inspection
- Exception-based tricks
- Timing checks to spot delays caused by breakpoints
- Hardware breakpoint detection

Many applications combine several of these techniques for stronger protection.

Your team can examine OS-level flags and behaviors. For example, software may query functions like IsDebuggerPresent, inspect process fields such as BeingDebugged in the PEB, monitor timing anomalies, or use exception handlers to catch unusual debugger-related behavior. You may also look for debugger-specific windows, handles, or loaded modules.

The appropriate response will depend on the sensitivity of your application. Some solutions immediately stop execution to block further inspection. Others deliberately alter behavior, restrict access to protected functionality, or log the incident for later analysis. More advanced systems may trigger alerts or activate additional verification steps to protect critical code paths.
