

Anti-Debug Protection Techniques: Implementation and Neutralization



Apriorit Inc, Kulchytskyy Oleg, 7 Apr 2016 [CPOL](#)



4.97 (31 votes)

Several approaches of anti-debug protection and some tips about how they are usually neutralized by reversers

In terms of software, reverse engineering is the process of researching programs to obtain some closed information and algorithms of their work. While software reversing results can be used for [legal purposes](#), it is generally considered that they are used by hackers for illegal activity. For example, researched algorithms can be the base to develop a license key generator (program crack). In the modern world, almost any software uses network data exchange. Such software product can be researched for vulnerabilities, which can be then used to get unauthorized access to a remote computer.

There are several approaches to analyze software:

1. Data exchange analysis. Researcher uses packet sniffer to analyze network data exchange.
2. Software binary code disassembling to get its listing in assembly language.
3. Decompilation of the binary or byte-code to re-create the source code in the high-level programming language.

This article considers popular anti-cracking, anti reverse engineering protection techniques, namely anti-debug methods in Windows OS. I want to mention right at the beginning that it is impossible to completely protect from reversing. The main goal of various anti reverse engineering techniques is to complicate reversers' life as much as possible.

Contents

Who this article is intended for

Anti-debug method introduction

IsDebuggerPresent

PEB (Process Environment Block)

How to neutralize the IsDebuggerPresent check

TLS Callback

NtGlobalFlag

How to neutralize the NtGlobalFlag check

NtGlobalFlag and IMAGE_LOAD_CONFIG_DIRECTORY

Heap Flags and ForceFlags

How to neutralize the Heap Flags and ForceFlags checks

Trap Flag Check

How to neutralize the TF check

CheckRemoteDebuggerPresent and NtQueryInformationProcess

How to neutralize CheckRemoteDebuggerPresent and NtQueryInformationProcess

Other techniques of anti-debug protection based on NtQueryInformationProcess

How to neutralize the NtQueryInformationProcess checks

Breakpoints: Software and Hardware ones

SEH (Structured Exception Handling)

How to neutralize SEH checks

VEH (Vectored Exception Handler)

How to neutralize hardware breakpoint check and VEH

NtSetInformationThread – hiding thread from debugger

How to neutralize thread hiding from debugger

NtCreateThreadEx

How to neutralize NtCreateThreadEx

Handle Tracing

Stack Segment Manipulation

Conclusion

References

Who this article is intended for

This article is for all software developers and reverse engineers, who are interested with the topic. To understand all described examples and technologies you need Assembler knowledge, some Windbg work experience, and experience of development for Windows using API functions.

Anti-debug method introduction

The best way to be prepared for an attack is to know where it would come from. This article presents popular anti-debug techniques, starting from the simplest ones, and notes about how to neutralize them. I won't consider different theories of building software protection, only practical examples.

IsDebuggerPresent

Maybe the simplest method is calling the [IsDebuggerPresent](#) function. This function detects if the calling process is being debugged by a user-mode debugger. The code below represents an elementary protection example:

```
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 such code:

```
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:

```
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
```

There is 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:

```
0:000< dt _PEB
```

```

ntdll!_PEB
+0x000 InheritedAddressSpace : UChar
+0x001 ReadImageFileExecOptions : UChar
+0x002 BeingDebugged      : UChar

```

i.e. the **IsDebuggerPresent** function reads the value of the **BeingDebugged** field. If the process is debugged, the value is 1, if not it will be 0.

PEB (Process Environment Block)

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

```

// 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 the x64 system, and another one PEB structure is created. Here you have an example of obtaining the PEB structure pointer in WOW64 environment:

```

// Get PEB for WOW64 Process
PVOID GetPEB64()
{
    PVOID pPeb = 0;
#ifdef _WIN64
    // 1. There are two copies of PEB - PEB64 and PEB32 in WOW64 process
    // 2. PEB64 follows after PEB32
    // 3. This is true for version less 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 of the function to check the OS version is below:

```

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; }

```

How to neutralize the IsDebuggerPresent check

To do so, one needs to put 0 to **BeingDebugged** before the checking code is executed. For example, DLL injection can be used for it:

```

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

```

For x64 process:

```

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

```

TLS Callback

Checking the presence of a debugger in the **main** function is not the best idea, as it is the first place a reverser looks at when viewing the 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. A good place to perform debugger presence check is TLS Callback. The Callback function will be called before the executable module entry point call.

```

#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;

```

NtGlobalFlag

In Windows NT, there is a set of flags, which is stored in the global variable **NtGlobalFlag**, common for the whole system. At the system start, the **NtGlobalFlag** global system variable is initialized with the value from the system registry key:

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\GlobalFlag]
```

The variable value is used for system tracing, debug, and control. The variable flags are undocumented, but the SDK includes the gflags utility, which allows 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 debug, such flags are set in the **NtGlobalFlag** field:

```

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

```

To check if a process is started with debugger, you should check the value of the NtGlobalFlag field of 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.

```

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

```

For x64 process:

```

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

```

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

```

#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)
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 neutralize the NtGlobalFlag check

One should just perform the actions inverse to the check ones before the check is performed, i.e. put 0 the **NtGlobalFlag** field of the PEB structure of the debugged process before this value is checked by the anti-debug protection.

NtGlobalFlag and 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 that 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 has a non-zero value, then there is 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:

```

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].VirtualAddress -
pImageSectionHeader->VirtualAddress)));
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 address of loading of the currently running process and checks the **GlobalFlagsClear** field value. If it is not 0, then the process is likely being debugged. The **CheckGlobalFlagsClearInFile** function performs the same check but for the executable on the disk.

Heap Flags and ForceFlags

The PEB structure contains the pointer to the process heap – the **_HEAP** structure:

```

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:

```

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 fields **Flags** and **ForceFlags** have specific debug values:

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

It's worth mentioning that the **_HEAP** structure is undocumented and the values of offsets of the **Flags** and **ForceFlags** fields can differ depending on the OS version. The following code represents the anti-debug protection sample based on the heap flag check:

```

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 neutralize the Heap Flags and ForceFlags checks

To neutralize anti-debug protection based on the heap flag check, one should set the **HEAP_GROWABLE** flag for the **Flags** field and set the value of the **ForceFlags** field to 0. Obviously, the redefining of the field values should be performed before the heap flag check.

Trap Flag Check

TF (Trap Flag) is inside **EFLAGS** register. If TF is set to 1, 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:

```

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 &ndash; 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 neutralize the TF check

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

CheckRemoteDebuggerPresent and NtQueryInformationProcess

Unlike the **IsDebuggerPresent** function, [CheckRemoteDebuggerPresent](#) checks if a process is being debugged by another parallel process. Here you have an example of **CheckRemoteDebuggerPresent**-based anti debugging technique:

```

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 of **CheckRemoteDebuggerPresent**, the **NtQueryInformationProcess** function is called:

```

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 gets the `DebugPort` value, as the `ProcessInformationClass` parameter value (the second one) is 7. The following anti debugging code sample is based on calling `NtQueryInformationProcess`:

```
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,
"NtQueryInformationProcess");
        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 neutralize CheckRemoteDebuggerPresent and NtQueryInformationProcess

One should substitute the value returned by the `NtQueryInformationProcess` function. [mhook](#) can be used. To set up a hook, one can inject DLL into the debugged process and set up a hook in DLLMain using mhook. Here is an example of mhook usage:

```
#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,
"NtQueryInformationProcess");
        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;
}

```

Other techniques of anti-debug protection based on NtQueryInformationProcess

There are several more technologies of debugger detection using information provided by the `NtQueryInformationProcess` function:

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

Let's consider them in detail.

ProcessDebugObjectHandle

Starting for Windows XP, the «debug object» is created for the debugged process. Here is an example of checking the «debug object» of the current process:

```

status = NtQueryInformationProcess(
    GetCurrentProcess(),
    ProcessDebugObjectHandle,
    &hProcessDebugObject,
    sizeof(HANDLE),
    NULL);
if (0x00000000 == status && NULL != hProcessDebugObject)

```

```

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

```

If there is a debug object, the process is being debugged.

ProcessDebugFlags

When checking this flag, it returns the inversed 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 is an example of such anti debugging check:

```

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:

```

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 the popular debuggers. Here is the listing of such anti-debug check:

```

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 neutralize the NtQueryInformationProcess checks

Neutralization is pretty simple. The values returned by the **NtQueryInformationProcess** function should be changed to those, which do not indicate debugger presence:

1. Set **ProcessDebugObjectHandle** to 0
2. Set **ProcessDebugFlags** to 1

3. For **ProcessBasicInformation**, change the **InheritedFromUniqueProcessId** value to the another process ID, e.g. explorer.exe

Breakpoints

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

1. Software breakpoints
2. Hardware breakpoints

It is very hard to reverse without breakpoints. So a popular anti reverse engineering tactics is based on detecting breakpoints, providing a series of the corresponding anti-debug methods.

Software Breakpoints

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

```
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 work if the `/INCREMENTAL:NO` linker option is set, otherwise when getting the function address to calculate the checksum, we will get the relative jump address:

```

DebuggeeFunction:
013C16DB jmp      DebuggeeFunction (013C4950h)

```

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

How to neutralize software breakpoint check

I do not know a universal approach. To neutralize this protection, one should find the code calculating 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 is a set of debug registers used by developers when checking and debugging the code. These registers allow to interrupt program execution and transfer control to debugger when accessing memory to read or write. Debug registers are a privileged resource and can be used by a program only in the real mode or safe mode with the privilege level CPL=0. There are 8 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 the 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, execute. Here is an example of hardware breakpoint check:

```

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 is also possible to reset hardware breakpoints by means of the **SetThreadContext** function. This is the hardware breakpoint reset example:

```

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

```

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

How to neutralize hardware breakpoint check and reset

If we look inside the **GetThreadContext** function, we'll understand that it calls the **NtGetContextThread** function:

```

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, one should 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** inside. The following example shows how to neutralize hardware breakpoint check and reset:

```

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);
}

```

SEH (Structured Exception Handling)

Structured Exception Handling is a mechanism provided by operating system to an application allowing it to receive notifications about exceptional situations like division by zero, reference to nonexistent pointer, or execution of a restricted instruction. This mechanism allows to handle exceptions inside the application, without OS involvement. If an exception is not handled, it will result in the abnormal program termination. Developers usually locate pointers to SEH in the stack, they 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:

```

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, SEH handler should return one of the `_EXCEPTION_DISPOSITION` members:

```

typedef enum _EXCEPTION_DISPOSITION {
    ExceptionContinueExecution,
    ExceptionContinueSearch,

```

```

    ExceptionNestedException,
    ExceptionCollidedUnwind
} EXCEPTION_DISPOSITION;

```

If handler returns **ExceptionContinueSearch**, the system continues execution from the instruction, which triggered the exception. If handler does not know what to do with an exception, it should return **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:

```

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, system handler goes to the registry to get the key

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\AeDebug
```

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 in the very beginning of the stack memory allocated for the process. The SEH handler function signature looks as follows:

```

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

```

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

```

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, SEH handler is set. The pointer to it is put to 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 **g_isDebuggerPresent** value will be set to **FALSE**. The **ContextRecord->Eip += 1** line changes 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 original SEH handler, clears the stack, and checks if debugger is present.

How to neutralize SEH checks

I do not know a universal approach, but still there are some techniques to make the reverser's life easier. Let's take a look at the call stack, which led to the SEH handler call:

```

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 the call came from **ntdll!ExecuteHandler2**. This function is the starting point for calling any SEH handler. Breakpoint can be set to the call instruction:

```

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

```

Then one should analyze the code of each called SEH handler. If protection is based on the multiple calls of SEH handlers, reverser will really sweat over it.

VEH (Vectored Exception Handler)

VEH was introduced starting from 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, SEH chain is not affected as the list of VEH handlers is stored in the **ntdll!LdrpVectorHandlerList** non-exported variable. VEH and SEH mechanisms are pretty similar, the only difference is that documented functions are used to set up and remove VEH handler. The signatures of the functions of adding and removing VEH handler as well as VEH handler function itself are as follows:

```

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 handler, the system collects the current process context and passes it via the **ContextRecord** parameter. Here you have an anti-debug protection code sample using vector exception handling:

```
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 generate interruption (it is not necessary int 1h). At the interruption generation, an exception appears and control is transferred to the VEH handler. If a hardware breakpoint is set, program execution is stopped. If there is no hardware breakpoints, the EIP register value is increased by 2 to continue execution after the int 1h generation instruction.

How to neutralize hardware breakpoint check and VEH

Let's take a look at the call stack which led to the VEH handler:

```
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:

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

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

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

The next code sample represents neutralization of the hardware breakpoint check applying the `KiUserExceptionDispatcher` function hook:

```
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,
"KiUserExceptionDispatcher");
    Mhook_SetHook((PVOID*)&g_origKiUserExceptionDispatcher, HookKiUserExceptionDispatcher);
    return 0;
}

```

In this example, the values of the DRx registers are reset in the **HookKiUserExceptionDispatcher** function, i.e. before the VEH handler call.

NtSetInformationThread – hiding thread from debugger

In Windows 2000, a new class of thread information transferred to the **NtSetInformationThread** function appeared – **ThreadHideFromDebugger**. It is one of the first anti-debug techniques provided by Windows, and it is very powerful. If this flag is set for a thread, it stops sending notifications about the debug events. These events include breakpoints and notification about the program finish, if **ThreadHideFromDebugger** was set for the main thread. The value of this flag is stored in the **HideFromDebugger** field of the **_ETHREAD** structure:

```

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

```

Here is an example of setting **ThreadHideFromDebugger**:

```

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 neutralize thread hiding from debugger

To prevent an application from hiding the thread from debugger, it is required to hook the **NtSetInformationThread** function call. Here you have a hook code example:

```
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 in a right way, **STATUS_SUCCESS** will be returned without transferring control to the original **NtSetInformationThread** function.

NtCreateThreadEx

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

```
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 such flags:

```

#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 debugger at its creation. It is the same **ThreadHideFromDebugger**, which is set up by the **NtSetInformationThread** function. The code, responsible for security tasks, can be executed in the thread with the **THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER** flag set.

How to neutralize NtCreateThreadEx

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

Handle Tracing

Starting from Windows XP, the system has the mechanism of kernel object handle tracing. When the tracing mode is on, all operations with handlers are saved to the circular buffer, and also when trying to use nonexistent handler, e.g. close it using the **CloseHandle** function, the **EXCEPTION_INVALID_HANDLE** exception will be generated. If a process is started not from debugger, the **CloseHandle** function will return **FALSE**. The following example shows the anti-debug protection based on **CloseHandle**:

```

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
}

```

Stack Segment Manipulation

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

```

__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
}

```

Conclusion

This article describes a series of anti reverse engineering techniques, in particular anti-debug methods, starting with the simplest ones, as well as the ways to neutralize them. It is just a limited list, as there are much more technologies and approaches you can continue investigating yourself:

1. Self-debugging process
2. Debugger detection using the `FindWindow` function

3. Time calculation approach (see for example [this article](#))
4. **NtQueryObject**
5. **BlockInput**
6. **NtSetDebugFilterState**
7. Self-modifying code

While we focused on the anti-debug protection methods, there are other anti-reverse engineering approaches, like anti-dumping, obfuscation, and others.

I want to stress again that even the best anti reverse engineering technique can't protect software completely from being reversed. The main task of anti-debug technologies is to make the reversers' life harder and complicate their work as much as possible.

References

1. [This article in the Apriorit Blog](#)
2. <https://msdn.microsoft.com/library>
3. <http://www.infosecinstitute.com/>
4. <http://pferrie.tripod.com/>
5. <http://www.openrce.org/articles/>
6. <http://www.nynaeve.net/>
7. <http://stackoverflow.com/>
8. <http://x86.renejeschke.de/>

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Authors

apriorit

Apriorit Inc




ApriorIT is a Software Research and Development company that works in advanced knowledge-intensive scopes.

Company offers integrated research&development services for the software projects in such directions as Corporate Security, Remote Control, Mobile Development, Embedded Systems, Virtualization, Drivers and others.

Official site <http://www.apriorit.com>

Group type: Organisation

Kulchytskyy Oleg

Ukraine 

No Biography provided