Anti-Debug: Direct debugger interaction

anti-debug.checkpoint.com/techniques/interactive.html

Contents

Direct debugger interaction

The following techniques let the running process manage a user interface or engage with its parent process to discover inconsistencies that are inherent for a debugged process.

1. Self-Debugging

There are at least three functions that can be used to attach as a debugger to a running process:

- `kernel32!DebugActiveProcess()`
- `ntdll!DbgUiDebugActiveProcess()`
- `ntdll!NtDebugActiveProcess()`

As only one debugger can be attached to a process at a time, a failure to attach to the process might indicate the presence of another debugger.

In the example below, we run the second instance of our process which tries to attach a debugger to its parent (the first instance of the process). If `kenel32!DebugActiveProcess()` finishes unsuccessfully, we set the named event which was created by the first instance. If the event is set, the first instance understands that a debugger is present.

C/C++ Code

```
#define EVENT_SELFDBG_EVENT_NAME L"SelfDebugging"

bool IsDebugged()
{
    WCHAR wszFilePath[MAX_PATH], wszCmdLine[MAX_PATH];
    STARTUPINFO si = { sizeof(si) };
    PROCESS_INFORMATION pi;
    HANDLE hDbgEvent;

    hDbgEvent = CreateEventW(NULL, FALSE, FALSE, EVENT_SELFDBG_EVENT_NAME);
    if (!hDbgEvent)
        return false;

    if (!GetModuleFileNameW(NULL, wszFilePath, _countof(wszFilePath)))
        return false;

    swprintf_s(wszCmdLine, L"%s %d", wszFilePath, GetCurrentProcessId());
    if (CreateProcessW(NULL, wszCmdLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))
    {
        WaitForSingleObject(pi.hProcess, INFINITE);
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);

        return WAIT_OBJECT_0 == WaitForSingleObject(hDbgEvent, 0);
    }

    return false;
}

bool EnableDebugPrivilege()
{
    bool bResult = false;
    HANDLE hToken = NULL;
    DWORD ec = 0;

    do
    {
        if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES, &hToken))
            break;

        TOKEN_PRIVILEGES tp;
        tp.PrivilegeCount = 1;
        if (!LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &tp.Privileges[0].Luid))
            break;

        tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
        if( !AdjustTokenPrivileges( hToken, FALSE, &tp, sizeof(tp), NULL, NULL))
            break;

        bResult = true;
    }
    while (0);
    if (hToken)
        CloseHandle(hToken);

    return bResult;
}

int main(int argc, char **argv)
{
    if (argc < 2)
    {
        if (IsDebugged())
            ExitProcess(0);
    }
    else
    {
        DWORD dwParentPid = atoi(argv[1]);
        HANDLE hEvent = OpenEventW(EVENT_MODIFY_STATE, FALSE,
EVENT_SELFDBG_EVENT_NAME);
        if (hEvent && EnableDebugPrivilege())
        {
            if (FALSE == DebugActiveProcess(dwParentPid))
                SetEvent(hEvent);
            else
                DebugActiveProcessStop(dwParentPid);
        }
        ExitProcess(0);
    }

    // ...

    return 0;
}
```

2. GenerateConsoleCtrlEvent()

When a user presses Ctrl+C or Ctrl+Break and a console window is in the focus, Windows checks if there is a handler for this event. All console processes have a default handler function that calls the `kernel32!ExitProcess()` function. However, we can register a custom handler for these events which neglects the Ctrl+C or Ctrl+Break signals.

However, if a console process is being debugged and CTRL+C signals have not been disabled, the system generates a `DBG_CONTROL_C` exception. Usually this exception is intercepted by a debugger, but if we register an exception handler, we will be able to check whether `DBG_CONTROL_C` is raised. If we intercepted the `DBG_CONTROL_C` exception in our own exception handler, it may indicate that the process is being debugged.

C/C++ Code

```
bool g_bDebugged{ false };
std::atomic<bool> g_bCtlCCatched{ false };

static LONG WINAPI CtrlEventExeptionHandler(PEXCEPTION_POINTERS pExceptionInfo)
{
    if (pExceptionInfo->ExceptionRecord->ExceptionCode == DBG_CONTROL_C)
    {
        g_bDebugged = true;
        g_bCtlCCatched.store(true);
    }
    return EXCEPTION_CONTINUE_EXECUTION;
}

static BOOL WINAPI CtrlHandler(DWORD fdwCtrlType)
{
    switch (fdwCtrlType)
    {
    case CTRL_C_EVENT:
        g_bCtlCCatched.store(true);
        return TRUE;
    default:
        return FALSE;
    }
}

bool IsDebugged()
{
    PVOID hVeh = nullptr;
    BOOL bCtrlHadnlerSet = FALSE;

    __try
    {
        hVeh = AddVectoredExceptionHandler(TRUE, CtrlEventExeptionHandler);
        if (!hVeh)
            __leave;

        bCtrlHadnlerSet = SetConsoleCtrlHandler(CtrlHandler, TRUE);
        if (!bCtrlHadnlerSet)
            __leave;

        GenerateConsoleCtrlEvent(CTRL_C_EVENT, 0);
        while (!g_bCtlCCatched.load())
            ;
    }
    __finally
    {
        if (bCtrlHadnlerSet)
            SetConsoleCtrlHandler(CtrlHandler, FALSE);

        if (hVeh)
            RemoveVectoredExceptionHandler(hVeh);
    }
    return g_bDebugged;
}
```

3. BlockInput()

The function `user32!BlockInput()` can block all mouse and keyboard events, which is quite an effective way to disable a debugger. On Windows Vista and higher versions, this call requires administrator privileges.

We can also detect whether a tool that hooks the `user32!BlockInput()` and other anti-debug calls is present. The function allows the input to be blocked only once. The second call will return `FALSE`. If the function returns `TRUE` regardless of the input, it may indicate that some hooking solution is present.

C/C++ Code

```
bool IsHooked ()
{
    BOOL bFirstResult = FALSE, bSecondResult = FALSE;
    __try
    {
        bFirstResult = BlockInput(TRUE);
        bSecondResult = BlockInput(TRUE);
    }
    __finally
    {
        BlockInput(FALSE);
    }
    return bFirstResult && bSecondResult;
}
```

4. NtSetInformationThread()

The function `ntdll!NtSetInformationThread()` can be used to hide a thread from a debugger. It is possible with a help of the undocumented value `THREAD_INFORMATION_CLASS::ThreadHideFromDebugger` (`0x11`). This is intended to be used by an external process, but any thread can use it on itself.

After the thread is hidden from the debugger, it will continue running but the debugger won't receive events related to this thread. This thread can perform anti-debugging checks such as code checksum, debug flags verification, etc.

However, if there is a breakpoint in the hidden thread or if we hide the main thread from the debugger, the process will crash and the debugger will be stuck.

In the example, we hide the current thread from the debugger. This means that if we trace this code in the debugger or put a breakpoint to any instruction of this thread, the debugging will be stuck once `ntdll!NtSetInformationThread()` is called.

C/C++ Code

```
#define NtCurrentThread ((HANDLE)-2)

bool AntiDebug()
{
    NTSTATUS status = ntdll::NtSetInformationThread(
        NtCurrentThread,
        ntdll::THREAD_INFORMATION_CLASS::ThreadHideFromDebugger,
        NULL,
        0);
    return status >= 0;
}
```

5. EnumWindows() and SuspendThread()

The idea of this technique is to suspend the owning thread of the parent process.

First, we need to verify whether the parent process is a debugger. This can be achieved by enumerating all top-level windows on the screen (using `user32!EnumWindows()` or `user32!EnumThreadWindows()`), searching the window for which process ID is the ID of the parent process (using `user32!GetWindowThreadProcessId()`), and checking the title of this window (by `user32!GetWindowTextW()`). If the window title of the parent process looks like a debugger title, we can suspend the owning thread using `kernel32!SuspendThread()` or `ntdll!NtSuspendThread()`.

C/C++ Code

```
DWORD g_dwDebuggerProcessId = -1;

BOOL CALLBACK EnumWindowsProc(HWND hwnd, LPARAM lParam)
{
    DWORD dwProcessId = *(PDWORD)lParam;

    DWORD dwWindowProcessId;
    GetWindowThreadProcessId(hwnd, &dwWindowProcessId);

    if (dwProcessId == dwWindowProcessId)
    {
        std::wstring wsWindowTitle{
string_heper::ToLower(std::wstring(GetWindowTextLengthW(hwnd) + 1, L'\0')) };
        GetWindowTextW(hwnd, &wsWindowTitle[0], wsWindowTitle.size());

        if (string_heper::FindSubstringW(wsWindowTitle, L"dbg") ||
            string_heper::FindSubstringW(wsWindowTitle, L"debugger"))
        {
            g_dwDebuggerProcessId = dwProcessId;
            return FALSE;
        }
        return FALSE;
    }

    return TRUE;
}

bool IsDebuggerProcess(DWORD dwProcessId) const
{
    EnumWindows(EnumWindowsProc, reinterpret_cast<LPARAM>(&dwProcessId));
    return g_dwDebuggerProcessId == dwProcessId;
}

bool SuspendDebuggerThread()
{
    THREADENTRY32 ThreadEntry = { 0 };
    ThreadEntry.dwSize = sizeof(THREADENTRY32);

    DWORD dwParentProcessId =
process_helper::GetParentProcessId(GetCurrentProcessId());
    if (-1 == dwParentProcessId)
        return false;

    HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD,
dwParentProcessId);
    if(Thread32First(hSnapshot, &ThreadEntry))
    {
        do
        {
            if ((ThreadEntry.th32OwnerProcessID == dwParentProcessId) &&
IsDebuggerProcess(dwParentProcessId))
            {
                HANDLE hThread = OpenThread(THREAD_SUSPEND_RESUME, FALSE,
ThreadEntry.th32ThreadID);
                if (hThread)
                    SuspendThread(hThread);
                break;
            }
        } while(Thread32Next(hSnapshot, &ThreadEntry));
    }

    if (hSnapshot)
        CloseHandle(hSnapshot);

    return false;
}
```

6. SwitchDesktop()

Windows supports multiple desktops per session. It is possible to select a different active desktop, which has the effect of hiding the windows of the previously active desktop, and with no obvious way to switch back to the old desktop.

Further, the mouse and keyboard events from the debugged process desktop will no longer be delivered to the debugger, because their source is no longer shared. This obviously makes debugging impossible.

C/C++ Code

```
BOOL Switch()
{
    HDESK hNewDesktop = CreateDesktopA(
        m_pcszNewDesktopName,
        NULL,
        NULL,
        0,
        DESKTOP_CREATEWINDOW | DESKTOP_WRITEOBJECTS | DESKTOP_SWITCHDESKTOP,
        NULL);
    if (!hNewDesktop)
        return FALSE;

    return SwitchDesktop(hNewDesktop);
}
```

7. OutputDebugString()

This technique is deprecated as it works only for Windows versions earlier than Vista. However, this technique is too well known to not mention here.

The idea is simple. If a debugger is not present and `kernel32!OutputDebugString` is called, then an error will occur.

C/C++ Code

```
bool IsDebugged()
{
    if (IsWindowsVistaOrGreater())
        return false;

    DWORD dwLastError = GetLastError();
    OutputDebugString(L"AntiDebug_OutputDebugString");
    return GetLastError() != dwLastError;
}
```

Mitigations

During debugging, it is better to skip suspicious function calls (e.g. fill them with `NOP`s).

If you write an anti-anti-debug solution, all the following functions can be hooked:

- `kernel32!DebugActiveProcess`
- `ntdll!DbgUiDebugActiveProcess`
- `ntdll!NtDebugActiveProcess`
- `kernel32!GenerateConsoleCtrlEvent()`
- `user32!NtUserBlockInput`
- `ntdll!NtSetInformationThread`
- `user32!NtUserBuildHwndList` (for filtering `EnumWindows` output)
- `kernel32!SuspendThread`
- `user32!SwitchDesktop`
- `kernel32!OutputDebugStringW`

Hooked functions can check input arguments and modify the original function behavior.