Anti-Debug: Object Handles


anti-debug.checkpoint.com/techniques/object-handles.html

Contents

Object Handles

The following set of techniques represents the checks which use kernel objects handles to detect a debugger presence. Some WinAPI functions that accept kernel object handles as their parameters can behave differently under debugging or cause side-effects that emerge because of debuggers' implementation. Moreover, there are specific kernel objecst that are created by the operation system when debugging begins.

1. OpenProcess()

Some debuggers can be detected by using the `kernel32!OpenProcess()` function on the **csrss.exe** process. The call will succeed only if the user for the process is a member of the administrators group and has debug privileges.

C/C++ Code

```
typedef DWORD (WINAPI *TCsrGetProcessId)(VOID);

bool Check()
{
    HMODULE hNtdll = LoadLibraryA("ntdll.dll");
    if (!hNtdll)
        return false;

    TCsrGetProcessId pfnCsrGetProcessId = (TCsrGetProcessId)GetProcAddress(hNtdll,
"CsrGetProcessId");
    if (!pfnCsrGetProcessId)
        return false;

    HANDLE hCsr = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pfnCsrGetProcessId());
    if (hCsr != NULL)
    {
        CloseHandle(hCsr);
        return true;
    }
    else
        return false;
}
```

2. CreateFile()

When the `CREATE_PROCESS_DEBUG_EVENT` event occurs, the handle of the debugged file is stored in the CREATE_PROCESS_DEBUG_INFO structure. Therefore, debuggers can read the debug information from this file. If this handle is not closed by the debugger, the file won't be opened with exclusive access. Some debuggers can forget to close the handle.

This trick uses `kernel32!CreateFileW()` (or `kernel32!CreateFileA()`) to exclusively open the file of the current process. If the call fails, we can consider that the current process is being run in the presence of a debugger.

C/C++ Code

```
bool Check()
{
    CHAR szFileName[MAX_PATH];
    if (0 == GetModuleFileNameA(NULL, szFileName, sizeof(szFileName)))
        return false;

    return INVALID_HANDLE_VALUE == CreateFileA(szFileName, GENERIC_READ, 0, NULL,
OPEN_EXISTING, 0, 0);
}
```

3. CloseHandle()

If a process is running under a debugger and an invalid handle is passed to the `ntdll!NtClose()` or `kernel32!CloseHandle()` function, then the `EXCEPTION_INVALID_HANDLE` (`0xC0000008`) exception will be raised. The exception can be cached by an exception handler. If the control is passed to the exception handler, it indicates that a debugger is present.

C/C++ Code

```
bool Check()
{
    __try
    {
        CloseHandle((HANDLE)0xDEADBEEF);
        return false;
    }
    __except (EXCEPTION_INVALID_HANDLE == GetExceptionCode()
                ? EXCEPTION_EXECUTE_HANDLER
                : EXCEPTION_CONTINUE_SEARCH)
    {
        return true;
    }
}
```

4. LoadLibrary()

When a file is loaded to process memory using the `kernel32!LoadLibraryW()` (or `kernel32!LoadLibraryA()`) function, the `LOAD_DLL_DEBUG_EVENT` event occurs. The handle of the loaded file will be stored in the LOAD_DLL_DEBUG_INFO structure. Therefore, debuggers can read the debug information from this file. If this handle is not closed by the debugger, the file won't be opened with exclusive access. Some debuggers can forget to close the handle.

To check for the debugger presence, we can load any file using `kernel32!LoadLibraryA()` and try to exclusively open it using `kernel32!CreateFileA()`. If the `kernel32!CreateFileA()` call fails, it indicates that the debugger is present.

C/C++ Code

```
bool Check()
{
    CHAR szBuffer[] = { "C:\\Windows\\System32\\calc.exe" };
    LoadLibraryA(szBuffer);
    return INVALID_HANDLE_VALUE == CreateFileA(szBuffer, GENERIC_READ, 0, NULL,
OPEN_EXISTING, 0, NULL);
}
```

5. NtQueryObject()

When a debugging session begins, a kernel object called **“debug object”** is created, and a handle is associated with it. Using the `ntdll!NtQueryObject()` function, it is possible to query for the list of existing objects, and check the number of handles associated with any debug object that exists.

However this technique can’t say for sure if the current process is being debugged right now. It only shows if the debugger is running on the system at all since the system’s boot.

C/C++ Code

```
typedef struct _OBJECT_TYPE_INFORMATION
{
    UNICODE_STRING TypeName;
    ULONG TotalNumberOfHandles;
    ULONG TotalNumberOfObjects;
} OBJECT_TYPE_INFORMATION, *POBJECT_TYPE_INFORMATION;

typedef struct _OBJECT_ALL_INFORMATION
{
    ULONG NumberOfObjects;
    OBJECT_TYPE_INFORMATION ObjectTypeInformation[1];
} OBJECT_ALL_INFORMATION, *POBJECT_ALL_INFORMATION;

typedef NTSTATUS (WINAPI *TNtQueryObject)(
    HANDLE                   Handle,
    OBJECT_INFORMATION_CLASS ObjectInformationClass,
    PVOID                    ObjectInformation,
    ULONG                    ObjectInformationLength,
    PULONG                   ReturnLength
);

enum { ObjectAllTypesInformation = 3 };

#define STATUS_INFO_LENGTH_MISMATCH 0xC0000004

bool Check()
{
    bool bDebugged = false;
    NTSTATUS status;
    LPVOID pMem = nullptr;
    ULONG dwMemSize;
    POBJECT_ALL_INFORMATION pObjectAllInfo;
    PBYTE pObjInfoLocation;
    HMODULE hNtdll;
    TNtQueryObject pfnNtQueryObject;

    hNtdll = LoadLibraryA("ntdll.dll");
    if (!hNtdll)
        return false;

    pfnNtQueryObject = (TNtQueryObject)GetProcAddress(hNtdll, "NtQueryObject");
    if (!pfnNtQueryObject)
        return false;

    status = pfnNtQueryObject(
        NULL,
        (OBJECT_INFORMATION_CLASS)ObjectAllTypesInformation,
        &dwMemSize, sizeof(dwMemSize), &dwMemSize);
    if (STATUS_INFO_LENGTH_MISMATCH != status)
        goto NtQueryObject_Cleanup;

    pMem = VirtualAlloc(NULL, dwMemSize, MEM_COMMIT, PAGE_READWRITE);
```

```
    if (!pMem)
        goto NtQueryObject_Cleanup;

    status = pfnNtQueryObject(
        (HANDLE)-1,
        (OBJECT_INFORMATION_CLASS)ObjectAllTypesInformation,
        pMem, dwMemSize, &dwMemSize);
    if (!SUCCEEDED(status))
        goto NtQueryObject_Cleanup;

    pObjectAllInfo = (POBJECT_ALL_INFORMATION)pMem;
    pObjInfoLocation = (PBYTE)pObjectAllInfo->ObjectTypeInformation;
    for(UINT i = 0; i < pObjectAllInfo->NumberOfObjects; i++)
    {

        POBJECT_TYPE_INFORMATION pObjectTypeInfo =
            (POBJECT_TYPE_INFORMATION)pObjInfoLocation;

        if (wcscmp(L"DebugObject", pObjectTypeInfo->TypeName.Buffer) == 0)
        {
            if (pObjectTypeInfo->TotalNumberOfObjects > 0)
                bDebugged = true;
            break;
        }

        // Get the address of the current entries
        // string so we can find the end
        pObjInfoLocation = (PBYTE)pObjectTypeInfo->TypeName.Buffer;

        // Add the size
        pObjInfoLocation += pObjectTypeInfo->TypeName.Length;

        // Skip the trailing null and alignment bytes
        ULONG tmp = ((ULONG)pObjInfoLocation) & -4;

        // Not pretty but it works
        pObjInfoLocation = ((PBYTE)tmp) + sizeof(DWORD);
    }

NtQueryObject_Cleanup:
    if (pMem)
        VirtualFree(pMem, 0, MEM_RELEASE);

    return bDebugged;
}
```

Mitigations

The simplest way to mitigate these checks is to just manually trace the program till a check and then skip it (e.g. patch with `NOP`s or change the instruction pointer or change the Zero Flag after the check).

If you write an anti-anti-debug solution, you need to hook the listed functions and change return values after analyzing their input:

- `ntdll!OpenProcess`: Return `NULL` if the third argument is the handle of **csrss.exe**.
- `ntdll!NtClose`: You can check if it is possible to retrieve any information about the input handle using `ntdll!NtQueryObject()` and not throw an exception if the handle is invalid.
- `ntdll!NtQueryObject`: Filter debug objects from the results if the `ObjectAllTypesInformation` class is queried.

The following techniques should be handled without hooks:

- `ntdll!NtCreateFile`: Too generic to mitigate. However, if you write a plugin for a specific debugger, you can ensure that the handle of the debugged file is closed.
- `kernel32!LoadLibraryW/A`: No mitigation.