Anti-Debug: Misc

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

Contents

Misc

1. FindWindow()

This technique includes the simple enumeration of window classes in the system and comparing them with known windows classes of debuggers.

The following functions can be used:

- `user32!FindWindowW()`
- `user32!FindWindowA()`
- `user32!FindWindowExW()`
- `user32!FindWindowExA()`

C/C++ Code

```
const std::vector<std::string> vWindowClasses = {
    "OLLYDBG",
    "WinDbgFrameClass", // WinDbg
    "ID",               // Immunity Debugger
    "Zeta Debugger",
    "Rock Debugger",
    "ObsidianGUI",
};

bool IsDebugged()
{
    for (auto &sWndClass : vWindowClasses)
    {
        if (NULL != FindWindowA(sWndClass.c_str(), NULL))
            return true;
    }
    return false;
}
```

2. Parent Process Check

Normally, a user-mode process is executed by double-clicking on a file icon. If the process is executed this way, its parent process will be the shell process (**"explorer.exe"**).

The main idea of the two following methods is to compare the PID of the parent process with the PID of **"explorer.exe"**.

2.1. NtQueryInformationProcess()

This method includes obtaining the shell process window handle using `user32!GetShellWindow()` and obtaining its process ID by calling `user32!GetWindowThreadProcessId()`.

Then, the parent process ID can be obtained from the `PROCESS_BASIC_INFORMATION` structure by calling `ntdll!NtQueryInformationProcess()` with the `ProcessBasicInformation` class.

C/C++ Code

```
bool IsDebugged()
{
    HWND hExplorerWnd = GetShellWindow();
    if (!hExplorerWnd)
        return false;

    DWORD dwExplorerProcessId;
    GetWindowThreadProcessId(hExplorerWnd, &dwExplorerProcessId);

    ntdll::PROCESS_BASIC_INFORMATION ProcessInfo;
    NTSTATUS status = ntdll::NtQueryInformationProcess(
        GetCurrentProcess(),
        ntdll::PROCESS_INFORMATION_CLASS::ProcessBasicInformation,
        &ProcessInfo,
        sizeof(ProcessInfo),
        NULL);
    if (!NT_SUCCESS(status))
        return false;

    return (DWORD)ProcessInfo.InheritedFromUniqueProcessId != dwExplorerProcessId;
}
```

2.2. CreateToolhelp32Snapshot()

The parent process ID and the parent process name can be obtained using the `kernel32!CreateToolhelp32Snapshot()` and `kernel32!Process32Next()` functions.

C/C++ Code

```
DWORD GetParentProcessId(DWORD dwCurrentProcessId)
{
    DWORD dwParentProcessId = -1;
    PROCESSENTRY32W ProcessEntry = { 0 };
    ProcessEntry.dwSize = sizeof(PROCESSENTRY32W);

    HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if(Process32FirstW(hSnapshot, &ProcessEntry))
    {
        do
        {
            if (ProcessEntry.th32ProcessID == dwCurrentProcessId)
            {
                dwParentProcessId = ProcessEntry.th32ParentProcessID;
                break;
            }
        } while(Process32NextW(hSnapshot, &ProcessEntry));
    }

    CloseHandle(hSnapshot);
    return dwParentProcessId;
}

bool IsDebugged()
{
    bool bDebugged = false;
    DWORD dwParentProcessId = GetParentProcessId(GetCurrentProcessId());

    PROCESSENTRY32 ProcessEntry = { 0 };
    ProcessEntry.dwSize = sizeof(PROCESSENTRY32W);

    HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if(Process32First(hSnapshot, &ProcessEntry))
    {
        do
        {
            if ((ProcessEntry.th32ProcessID == dwParentProcessId) &&
                (strcmp(ProcessEntry.szExeFile, "explorer.exe")))
            {
                bDebugged = true;
                break;
            }
        } while(Process32Next(hSnapshot, &ProcessEntry));
    }

    CloseHandle(hSnapshot);
    return bDebugged;
}
```

3. Selectors

Selector values might appear to be stable, but they are actually volatile in certain circumstances, and also depending on the version of Windows. For example, a selector value can be set within a thread, but it might not hold that value for very long. Certain events might cause the selector value to be changed back to its default value. One such event is an exception. In the context of a debugger, the single-step exception is still an exception, which can cause some unexpected behavior.

x86 Assembly

```
    xor  eax, eax
    push fs
    pop  ds
l1: xchg [eax], cl
    xchg [eax], cl
```

On the 64-bit versions of Windows, single-stepping through this code will cause an access violation exception at `l1` because the DS selector will be restored to its default value even before `l1` is reached. On the 32-bit versions of Windows, the `DS` selector will not have its value restored, unless a non-debugging exception occurs. The version-specific difference in behaviors expands even further if the `SS` selector is used. On the 64-bit versions of Windows, the `SS` selector will be restored to its default value, as in the `DS` selector case. However, on the 32-bit versions of Windows, the `SS` selector value will not be restored, even if an exception occurs.

x86-64 Assembly

```
    xor  eax, eax
    push offset l2
    push d fs:[eax]
    mov  fs:[eax], esp
    push fs
    pop  ss
    xchg [eax], cl
    xchg [eax], cl
l1: int  3 ;force exception to occur
l2: ;looks like it would be reached
    ;if an exception occurs
    ...
```

then when the “`int 3`” instruction is reached at `l1` and the breakpoint exception occurs, the exception handler at `l2` is not called as expected. Instead, the process is simply terminated.

A variation of this technique detects the single-step event by simply checking if the assignment was successful.

```
push 3
pop  gs
mov  ax, gs
cmp  al, 3
jne  being_debugged
```

The `FS` and `GS` selectors are special cases. For certain values, they will be affected by the single-step event, even on the 32-bit versions of Windows. However, in the case of the `FS` selector (and, technically, the `GS` selector), it will be not restored to its default value on the 32-bit versions of Windows, if it was set to a value from zero to three. Instead, it will be set to zero (the `GS` selector is affected in the same way, but the default value for the `GS` selector is zero). On the 64-bit versions of Windows, it (they) will be restored to its (their) default value.

This code is also vulnerable to a race condition caused by a thread-switch event. When a thread-switch event occurs, it behaves like an exception, and will cause the selector values to be altered, which, in the case of the `FS` selector, means that it will be set to zero.

A variation of this technique solves that problem by waiting intentionally for a thread-switch event to occur.

```
    push 3
    pop  gs
l1: mov  ax, gs
    cmp  al, 3
    je   l1
```

However, this code is vulnerable to the problem that it was trying to detect in the first place, because it does not check if the original assignment was successful. Of course, the two code snippets can be combined to produce the desired effect, by waiting until the thread-switch event occurs, and then performing the assignment within the window of time that should exist until the next one occurs. [Ferrie]

C/C++ Code

```
bool IsTraced()
{
    __asm
    {
        push 3
        pop  gs

    __asm SeclectorsLbl:
        mov  ax, gs
        cmp  al, 3
        je   SeclectorsLbl

        push 3
        pop  gs
        mov  ax, gs
        cmp  al, 3
        jne  Selectors_Debugged
    }

    return false;

Selectors_Debugged:
    return true;
}
```

4. DbgPrint()

The debug functions such as `ntdll!DbgPrint()` and `kernel32!OutputDebugStringW()` cause the exception `DBG_PRINTEXCEPTION_C` (`0x40010006`). If a program is executed with an attached debugger, then the debugger will handle this exception. But if no debugger is present, and an exception handler is registered, this exception will be caught by the exception handler.

C/C++ Code

```
bool IsDebugged()
{
    __try
    {
        RaiseException(DBG_PRINTEXCEPTION_C, 0, 0, 0);
    }
    __except(GetExceptionCode() == DBG_PRINTEXCEPTION_C)
    {
        return false;
    }

    return true;
}
```

5. DbgSetDebugFilterState()

The functions `ntdll!DbgSetDebugFilterState()` and `ntdll!NtSetDebugFilterState()` only set a flag which will be checked be a kernel-mode debugger if it is present. Therefore, if a kernel debugger is attached to the system, these functions will succeed. However, the functions can also succeed because of side-effects caused by some user-mode debuggers. These functions require administrator privileges.

C/C++ Code

```
bool IsDebugged()
{
    return NT_SUCCESS(ntdll::NtSetDebugFilterState(0, 0, TRUE));
}
```

6. NtYieldExecution() / SwitchToThread()

This method is not really reliable because it only shows if there a high priority thread in the current process. However, it could be used as an anti-tracing technique.

When an application is traced in a debugger and a single-step is executed, the context can't be switched to other thread. This means that `ntdll!NtYieldExecution()` returns `STATUS_NO_YIELD_PERFORMED` (`0x40000024`), which leads to `kernel32!SwitchToThread()` returning zero.

The strategy of using this technique is that there is a loop which modifies some counter if `kernel32!SwitchToThread()` returns zero, or `ntdll!NtYieldExecution()` returns `STATUS_NO_YIELD_PERFORMED`. This can be a loop which decrypts strings or some other loop which is supposed to be analyzed manually in a debugger. If the counter has the expected value (expected i.e. the value if all `kernel32!SwitchToThread()` returned zero) after leaving the loop, we consider that the debugger is present.

In the example below, we define a one-byte counter (initialized with 0) which shifts one bit to the left if `kernel32!SwitchToThread` returns zero. If it shifts 8 times, then the value of the counter will become 0 and the debugger is considered to be present.

C/C++ Code

```
bool IsDebugged()
{
    BYTE ucCounter = 1;
    for (int i = 0; i < 8; i++)
    {
        Sleep(0x0F);
        ucCounter <<= (1 - SwitchToThread());
    }

    return ucCounter == 0;
}
```

Mitigations

During debugging: Fill anti-debug pr anti-traced checks with `NOP`s.

For anti-anti-debug tool development:

1. For `FindWindow()`: Hook `user32!NtUserFindWindowEx()`. In the hook, call the original `user32!NtUserFindWindowEx()` function. If it is called from the debugged process and the parent process looks suspicious, then return unsuccessfully from the hook.

2. For Parent Process Checks: Hook `ntdll!NtQuerySystemInformation()`. If `SystemInformationClass` is one of the following values:
   - `SystemProcessInformation`
   - `SystemSessionProcessInformation`
   - `SystemExtendedProcessInformation`

   and the process name looks suspicious, then the hook must modify the process name.

3. For Selectors: No mitigations.

4. For `DbgPrint`: you have to implement a plugin for a specific debugger and change the behavior of event handler which is triggered after the `DBG_PRINTEXCEPTION_C` exception has arrived.

5. For `DbgSetDebugFilterState()`: Hook `ntdll!NtSetDebugFilterState()`. If the process is running with debug privileges, return unsuccessfully from the hook.

6. For `SwitchToThread`: Hook `ntdll!NtYieldExecution()` and return an unsuccessful status from the hook.