x86matthew - SelfDebug - A useless anti-debug trick by forcing a process to debug itself

x86matthew.com/view_post

Recently, I was looking at the internals of the Windows debugging functions and had an idea for an anti-debug trick. While this method is too much of a "hack" to be used in real-world situations, it's mildly interesting so I decided to document it anyway.

A process being debugged (debugee) communicates with the debugger using the DebugPort object. A process can only be attached to one DebugPort at a time, and any further attempts to attach a debugger to the process will fail. This can be observed in the DbgkpSetProcessDebugObject function, which returns the error code 0xC0000048 (STATUS_PORT_ALREADY_SET) if the target process already has an existing DebugPort object:

```
; CODE XREF: DbgkpSetProcessDebugObject+79↑j
cmp qword ptr [rsi+578h], 0 ; if(Process->DebugPort == NULL)
jz get_next_thread
mov edi, 0C0000048h ; error (STATUS_PORT_ALREADY_SET)
```

For obvious reasons, it is not possible for a process to debug itself. To prevent this from occurring inadvertently, the NtDebugActiveProcess function checks if the target process has the same PID as the current process. If this is the case, the function returns the error code 0xC0000022 (STATUS_ACCESS_DENIED):

```
NtDebugActiveProcess proc near
mov r11, rsp
mov [r11+8], rbx
mov [r11+10h], rbp
push rsi
push rdi
push r14
sub rsp, 50h
mov rax, gs:188h
mov r14, rdx
and qword ptr [r11-38h], 0
mov edx, 800h ; DesiredAccess
mov r8, cs:PsProcessType ; ObjectType
and qword ptr [r11+18h], 0
mov bpl, [rax+232h]
lea rax, [r11+18h]
and qword ptr [r11-28h], 0
```

```
mov r9b, bpl ; AccessMode
mov [r11-40h], rax
mov [rsp+68h+Tag], 4F676244h ; Tag
call ObReferenceObjectByHandleWithTag
test eax, eax
js loc_140886246
mov rax, gs:188h ; KeGetCurrentThread
mov rdi, [rsp+68h+arg_10] ; get target PID
mov rsi, [rax+0B8h] ; get current PID
cmp rdi, rsi ; if(TargetPID == CurrentPID)
jz error_C0000022
cmp rdi, cs:PsInitialSystemProcess ; if(TargetPID == PsInitialSystemProcess)
jz error_C0000022
mov r8, rdi
```

Despite the checks mentioned previously, it is possible to force the target process into a state which prevents a real debugger from attaching. This technique does require a temporary child process to be launched momentarily, but this does not need to remain running for the duration of the parent process. The method works as follows:

1. The main process should relaunch itself as a child process, passing a parameter to differentiate this instance from the target process.
2. The child process calls DebugActiveProcess to debug the parent process.
3. The child process retrieves the debug object handle from the PEB and calls DuplicateHandle to duplicate it into the parent process.
4. The child process calls WaitForDebugEvent/ContinueDebugEvent to clear all debug events.
5. The child process calls CloseHandle to force the local debug object to close.
6. The child process terminates.
7. The main process resumes normal execution now that the child process has exited.

Under normal circumstances, a debugger would stop debugging a process by calling DebugActiveProcessStop (NtRemoveProcessDebug) - however, in this case, the debug object is being closed "uncleanly" using CloseHandle. Prior to closing the debug object, it is duplicated into the parent process, therefore increasing its reference count and keeping it "alive". This means that the parent process (which is being debugged) will continue running after the child process exits, and the original process is now in a state where it is essentially debugging itself.

Of course, this technique has a major drawback - the process will enter a suspended state when an exception occurs. As the suspended process will not be able to handle its own debug messages, the process will freeze indefinitely. Despite this limitation, it is still possible to create a simple program that uses this technique with some success.

One issue that we may encounter relates to threads - a debug message is generated whenever a thread starts or ends within the target process. As mentioned earlier, any new debug messages will freeze the process. To create a thread without generating a debug message, we can first create the new thread in a suspended state, set the ThreadHideFromDebugger flag using NtSetInformationThread to suppress debug messages for this thread, and finally resume the thread:

```
HANDLE CreateHiddenThread(LPTHREAD_START_ROUTINE lpStartAddress, LPVOID
lpParameter)
{
DWORD dwThreadID = 0;
HANDLE hThread = NULL;

// create thread (suspended initially)
hThread = CreateThread(NULL, 0, lpStartAddress, lpParameter, CREATE_SUSPENDED,
&dwThreadID);
if(hThread == NULL)
{
// error
return NULL;
}

// hide thread from debugger before resuming
if(pNtSetInformationThread(hThread, ThreadHideFromDebugger, NULL, 0) != 0)
{
// error
TerminateThread(hThread, 0);
CloseHandle(hThread);
return NULL;
}

// resume thread
ResumeThread(hThread);

return hThread;
}
```

Windows 10 introduced a feature called parallel loading, which uses a thread pool to load initial dependencies in the background for performance reasons. This means that even a single-threaded program will also contain additional threads running in the background. After 60 seconds, these background threads will exit and cause the program to freeze due to the debug messages that are generated. For the purpose of this proof-of-concept, a quick workaround for this issue is to terminate the background threads and forcibly close the TpWorkerFactory objects to prevent any new background threads from starting. These changes will allow the sample program to run indefinitely.

This technique can be bypassed by closing the debug handle within the target process before attaching a real debugger.

Full code below:

```
#include <stdio.h>
#include <windows.h>

#define ObjectTypeInformation 2
#define SystemExtendedHandleInformation 64
#define ThreadHideFromDebugger 17
#define ThreadBasicInformation 0
#define SystemProcessInformation 5

#define STATUS_INFO_LENGTH_MISMATCH 0xC0000004

struct CLIENT_ID
{
HANDLE UniqueProcess;
HANDLE UniqueThread;
};

struct UNICODE_STRING
{
USHORT Length;
USHORT MaximumLength;
PWSTR Buffer;
};

struct OBJECT_ATTRIBUTES
{
ULONG Length;
HANDLE RootDirectory;
UNICODE_STRING *ObjectName;
ULONG Attributes;
PVOID SecurityDescriptor;
PVOID SecurityQualityOfService;
};

struct SYSTEM_PROCESS_INFORMATION
{
ULONG NextEntryOffset;
ULONG NumberOfThreads;
BYTE Reserved1[48];
UNICODE_STRING ImageName;
DWORD BasePriority;
HANDLE UniqueProcessId;
PVOID Reserved2;
ULONG HandleCount;
ULONG SessionId;
PVOID Reserved3;
SIZE_T PeakVirtualSize;
SIZE_T VirtualSize;
ULONG Reserved4;
SIZE_T PeakWorkingSetSize;
SIZE_T WorkingSetSize;
PVOID Reserved5;
SIZE_T QuotaPagedPoolUsage;
PVOID Reserved6;
SIZE_T QuotaNonPagedPoolUsage;
SIZE_T PagefileUsage;
SIZE_T PeakPagefileUsage;
SIZE_T PrivatePageCount;
LARGE_INTEGER Reserved7[6];
};

struct SYSTEM_THREAD_INFORMATION
{
LARGE_INTEGER Reserved1[3];
ULONG Reserved2;
PVOID StartAddress;
CLIENT_ID ClientId;
DWORD Priority;
LONG BasePriority;
ULONG Reserved3;
ULONG ThreadState;
ULONG WaitReason;
};

struct SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX
{
PVOID Object;
PVOID UniqueProcessId;
PVOID HandleValue;
ULONG GrantedAccess;
USHORT CreatorBackTraceIndex;
USHORT ObjectTypeIndex;
```

```
ULONG HandleAttributes;
ULONG Reserved;
};

struct SYSTEM_HANDLE_INFORMATION_EX
{
PVOID NumberOfHandles;
PVOID Reserved;
SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX HandleList[1];
};

DWORD(WINAPI *pNtQueryObject)(HANDLE Handle, DWORD ObjectInformationClass,
PVOID ObjectInformation, DWORD ObjectInformationLength, DWORD *ReturnLength) =
NULL;
DWORD(WINAPI *pNtQuerySystemInformation)(DWORD SystemInformationClass,
PVOID SystemInformation, ULONG SystemInformationLength, PULONG ReturnLength)
= NULL;
DWORD(WINAPI *pNtOpenThread)(HANDLE *ThreadHandle, DWORD DesiredAccess,
OBJECT_ATTRIBUTES *ObjectAttributes, CLIENT_ID *ClientId) = NULL;

SYSTEM_PROCESS_INFORMATION *pGlobal_SystemProcessInfo = NULL;
SYSTEM_HANDLE_INFORMATION_EX *pGlobal_SystemHandleInfo = NULL;

DWORD GetNtdllFunctions()
{
HMODULE hNtdll = NULL;

// get ntdll base
hNtdll = GetModuleHandleA("ntdll.dll");
if(hNtdll == NULL)
{
return 1;
}

// get NtQuerySystemInformation ptr
pNtQuerySystemInformation = (DWORD(WINAPI*)(DWORD, PVOID, ULONG,
PULONG))GetProcAddress(hNtdll, "NtQuerySystemInformation");
if(pNtQuerySystemInformation == NULL)
{
return 1;
}

// get NtOpenThread ptr
pNtOpenThread = (DWORD(WINAPI*)(HANDLE *, DWORD, OBJECT_ATTRIBUTES *,
CLIENT_ID *))GetProcAddress(hNtdll, "NtOpenThread");
if(pNtOpenThread == NULL)
```

```
{
return 1;
}

// get NtQueryObject ptr
pNtQueryObject = (DWORD(WINAPI*)(HANDLE, DWORD, PVOID, DWORD, DWORD
*))GetProcAddress(hNtdll, "NtQueryObject");
if(pNtQueryObject == NULL)
{
return 1;
}

return 0;
}

DWORD WaitForDebuggerAttach()
{
BYTE *pPEB = NULL;

// get PEB ptr
#if _WIN64
pPEB = (BYTE *)__readgsqword(0x60);
#else
pPEB = (BYTE *)__readfsdword(0x30);
#endif

for(;;)
{
// check BeingDebugged flag
if(*(BYTE *)(pPEB + 0x2) != 0)
{
// debugger found
break;
}

// wait for 100ms before retrying
Sleep(100);
}

return 0;
}

HANDLE GetDebugObject()
{
BYTE *pTEB = NULL;
HANDLE hDebugObject = NULL;
```

```
// get DbgSsReserved[1] from TEB
#if _WIN64
pTEB = (BYTE *)__readgsqword(0x30);
hDebugObject = *(HANDLE *)(pTEB + 0x16A8);
#else
pTEB = (BYTE *)__readfsdword(0x18);
hDebugObject = *(HANDLE *)(pTEB + 0xF24);
#endif

return hDebugObject;
}

BYTE *GetSystemInformationBlock(DWORD dwSystemInformationClass)
{
DWORD dwAllocSize = 0;
DWORD dwStatus = 0;
DWORD dwLength = 0;
BYTE *pSystemInfoBuffer = NULL;

// get system process list
dwAllocSize = 0;
for(;;)
{
if(pSystemInfoBuffer != NULL)
{
// free previous inadequately sized buffer
free(pSystemInfoBuffer);
pSystemInfoBuffer = NULL;
}

if(dwAllocSize != 0)
{
// allocate new buffer
pSystemInfoBuffer = (BYTE *)malloc(dwAllocSize);
if(pSystemInfoBuffer == NULL)
{
return NULL;
}
}

// query system info block
dwStatus = pNtQuerySystemInformation(dwSystemInformationClass, (void
*)pSystemInfoBuffer, dwAllocSize, &dwLength);
if(dwStatus == 0)
{
// success
break;
}
else if(dwStatus == STATUS_INFO_LENGTH_MISMATCH)
{
// not enough space - allocate a larger buffer and try again (also add an extra 1kb to allow
for additional data between checks)
dwAllocSize = (dwLength + 1024);
}
else
{
// other error
free(pSystemInfoBuffer);
return NULL;
}
}

return pSystemInfoBuffer;
}

DWORD GetSystemProcessInformation()
{
// free previous process info list
if(pGlobal_SystemProcessInfo != NULL)
{
free(pGlobal_SystemProcessInfo);
}

// get system process list
pGlobal_SystemProcessInfo = (SYSTEM_PROCESS_INFORMATION
*)GetSystemInformationBlock(SystemProcessInformation);
if(pGlobal_SystemProcessInfo == NULL)
{
return 1;
}

return 0;
}

DWORD GetSystemHandleList()
{
// free previous handle info list
if(pGlobal_SystemHandleInfo != NULL)
{
free(pGlobal_SystemHandleInfo);
}
```

```
// get system handle list
pGlobal_SystemHandleInfo = (SYSTEM_HANDLE_INFORMATION_EX
*)GetSystemInformationBlock(SystemExtendedHandleInformation);
if(pGlobal_SystemHandleInfo == NULL)
{
return 1;
}

return 0;
}

DWORD TerminateAllThreadpoolThreads()
{
HANDLE hThread = NULL;
SYSTEM_PROCESS_INFORMATION *pCurrProcessInfo = NULL;
SYSTEM_PROCESS_INFORMATION *pNextProcessInfo = NULL;
SYSTEM_PROCESS_INFORMATION *pTargetProcessInfo = NULL;
SYSTEM_THREAD_INFORMATION *pCurrThreadInfo = NULL;
OBJECT_ATTRIBUTES ObjectAttributes;
DWORD dwStatus = 0;

// get snapshot of processes/threads
if(GetSystemProcessInformation() != 0)
{
return 1;
}

// find the target process in the list
pCurrProcessInfo = pGlobal_SystemProcessInfo;
for(;;)
{
// check if this is the target PID
if((SIZE_T)pCurrProcessInfo->UniqueProcessId == (SIZE_T)GetCurrentProcessId())
{
// found target process
pTargetProcessInfo = pCurrProcessInfo;
break;
}

// check if this is the end of the list
if(pCurrProcessInfo->NextEntryOffset == 0)
{
// end of list
break;
}
else
{
// get next process ptr
pNextProcessInfo = (SYSTEM_PROCESS_INFORMATION *)((BYTE *)pCurrProcessInfo
+ pCurrProcessInfo->NextEntryOffset);
}

// go to next process
pCurrProcessInfo = pNextProcessInfo;
}

// ensure the target process was found in the list
if(pTargetProcessInfo == NULL)
{
return 1;
}

// loop through all threads within the target process
pCurrThreadInfo = (SYSTEM_THREAD_INFORMATION *)((BYTE *)pTargetProcessInfo
+ sizeof(SYSTEM_PROCESS_INFORMATION));
for(DWORD i = 0; i < pTargetProcessInfo->NumberOfThreads; i++)
{
// ignore current (main) thread
if((SIZE_T)pCurrThreadInfo->ClientId.UniqueThread != (SIZE_T)GetCurrentThreadId())
{
// open thread
memset((void *)&ObjectAttributes, 0, sizeof(ObjectAttributes));
ObjectAttributes.Length = sizeof(ObjectAttributes);
dwStatus = pNtOpenThread(&hThread, THREAD_ALL_ACCESS, &ObjectAttributes,
&pCurrThreadInfo->ClientId);
if(dwStatus == 0)
{
// terminate thread
TerminateThread(hThread, 0);

// close handle
CloseHandle(hThread);
}
}

// move to the next thread
pCurrThreadInfo++;
}

return 0;
}
```

```
DWORD CloseAllThreadpoolHandles()
{
BYTE bObjectTypeBuffer[4096];
DWORD dwObjectTypeLength = 0;
DWORD dwStatus = 0;
UNICODE_STRING *pObjectNameUnicodeString = NULL;

// get system handle list
if(GetSystemHandleList() != 0)
{
return 1;
}

// find threadpool handles
for(SIZE_T i = 0; i < (SIZE_T)pGlobal_SystemHandleInfo->NumberOfHandles; i++)
{
// check if this handle exists within the current process
if((SIZE_T)pGlobal_SystemHandleInfo->HandleList[i].UniqueProcessId !=
(SIZE_T)GetCurrentProcessId())
{
continue;
}

// get object type
memset(bObjectTypeBuffer, 0, sizeof(bObjectTypeBuffer));
dwStatus = pNtQueryObject(pGlobal_SystemHandleInfo->HandleList[i].HandleValue,
ObjectTypeInformation, bObjectTypeBuffer, sizeof(bObjectTypeBuffer),
&dwObjectTypeLength);
if(dwStatus != 0)
{
continue;
}

// check if this is a threadpool object
pObjectNameUnicodeString = (UNICODE_STRING *)bObjectTypeBuffer;
if(wcsncmp(pObjectNameUnicodeString->Buffer, L"TpWorkerFactory",
pObjectNameUnicodeString->Length / 2) == 0)
{
// close threadpool handle
CloseHandle(pGlobal_SystemHandleInfo->HandleList[i].HandleValue);
}
}

return 0;
}
```

```
DWORD InitialiseTemporaryChildProcess(DWORD dwParentPID)
{
HANDLE hProcess = NULL;
HANDLE hDebug = NULL;
HANDLE hClonedDebugHandle = NULL;
DEBUG_EVENT DBEvent;
DWORD dwTimeoutCount = 0;

// open parent process
hProcess = OpenProcess(PROCESS_ALL_ACCESS, 0, dwParentPID);
if(hProcess == NULL)
{
return 1;
}

// debug parent process
if(DebugActiveProcess(dwParentPID) == 0)
{
CloseHandle(hProcess);
return 1;
}

// get the local debug object from the TEB
hDebug = GetDebugObject();
if(hDebug == NULL)
{
CloseHandle(hProcess);
return 1;
}

// duplicate the debug object from the temporary child process (debugger) into the parent
process (debugee)
if(DuplicateHandle(GetCurrentProcess(), hDebug, hProcess, &hClonedDebugHandle, 0,
0, DUPLICATE_SAME_ACCESS) == 0)
{
CloseHandle(hDebug);
CloseHandle(hProcess);
return 1;
}

// flush the debug message queue
dwTimeoutCount = 0;
for(;;)
{
// check timeout count
if(dwTimeoutCount >= 10)
{
// no debug events for 1 second
break;
}

// wait for event
if(WaitForDebugEvent(&DBEvent, 100) == 0)
{
dwTimeoutCount++;
continue;
}

// reset counter
dwTimeoutCount = 0;

// continue
ContinueDebugEvent(DBEvent.dwProcessId, DBEvent.dwThreadId, DBG_CONTINUE);
}

// manually close the debug handle in the temporary child process
CloseHandle(hDebug);

return 0;
}

DWORD InitialiseMainProcess()
{
STARTUPINFOA StartupInfo;
PROCESS_INFORMATION ProcessInfo;
char szCurrentPath[512];
char szChildProcessCommandLine[512];
DWORD dwChildProcessExitCode = 0;

// get current exe path
memset(szCurrentPath, 0, sizeof(szCurrentPath));
GetModuleFileNameA(NULL, szCurrentPath, sizeof(szCurrentPath) - 1);

// set child process command line
memset(szChildProcessCommandLine, 0, sizeof(szChildProcessCommandLine));
_snprintf_s(szChildProcessCommandLine, sizeof(szChildProcessCommandLine) - 1,
"\"%s\" %u", szCurrentPath, GetCurrentProcessId());

// launch temporary child process
memset(&StartupInfo, 0, sizeof(StartupInfo));
StartupInfo.cb = sizeof(StartupInfo);
if(CreateProcessA(NULL, szChildProcessCommandLine, NULL, NULL, 0, 0, NULL,
NULL, &StartupInfo, &ProcessInfo) == 0)
```

```
{
return 1;
}

// close all threadpool handles in the main process
CloseAllThreadpoolHandles();

// close all threadpool threads
TerminateAllThreadpoolThreads();

// free process info list
if(pGlobal_SystemProcessInfo != NULL)
{
free(pGlobal_SystemProcessInfo);
}

// free handle info list
if(pGlobal_SystemHandleInfo != NULL)
{
free(pGlobal_SystemHandleInfo);
}

// wait for debugger to attach
WaitForDebuggerAttach();

// wait for child process to end, get exit code
WaitForSingleObject(ProcessInfo.hProcess, INFINITE);
GetExitCodeProcess(ProcessInfo.hProcess, &dwChildProcessExitCode);

// close handles
CloseHandle(ProcessInfo.hProcess);
CloseHandle(ProcessInfo.hThread);

// ensure child process returned success code
if(dwChildProcessExitCode != 0)
{
return 1;
}

return 0;
}

DWORD ExecuteMainCode()
{
DWORD dwCount = 0;
```

```
for(;;)
{
printf("Main thread running (%u)\n", dwCount);
dwCount++;
Sleep(1000);
}

return 0;
}

int main(int argc, char *argv[])
{
DWORD dwParentPID = 0;

// get ntdll function ptrs
if(GetNtdllFunctions() != 0)
{
return 1;
}

// check if this is the parent/child process
if(argc == 2)
{
// this is the temporary child process
dwParentPID = atoi(argv[1]);
if(InitialiseTemporaryChildProcess(dwParentPID) != 0)
{
return 1;
}
}
else
{
// this is the main process
if(InitialiseMainProcess() != 0)
{
return 1;
}

// anti-debug ready - execute target code
ExecuteMainCode();
}

return 0;
}
```