System-wide anti-debug technique using NtQuerySystemInformation and DuplicateHandle

System-wide anti-debug technique using NtQuerySystemInformation and DuplicateHandle

Posted: 01/02/2022

I have developed an anti-debug technique that targets user-mode debuggers indiscriminately, rather than detecting if an individual process is being debugged.

In summary, this method works as follows:

1. Retrieve a list of all open handles in the system using NtQuerySystemInformation with SystemExtendedHandleInformation.
2. Check if any processes contain an active debug handle.
3. Terminate this handle in the remote process using DuplicateHandle with the DUPLICATE_CLOSE_SOURCE flag.
4. Loop back to step #1.

One thing that complicates the method above is in identifying debug handles during step #2. Each handle type is identified using the ObjectTypeIndex field returned by NtQuerySystemInformation, but this value is not consistent between different versions of Windows.

We could obviously hard-code the various possible values in a lookup table, but a generic solution is always preferable. In order to calculate the debug handle type index for the current OS, I took the following steps:

1. Retrieve a list of all open handles in the system using NtQuerySystemInformation with SystemExtendedHandleInformation.
2. Set a flag for each unique object type that was found in the current process (eg file handle, process handle, registry key, etc).
3. Call DebugActiveProcess with a PID of 0. DebugActiveProcess creates an internal debug handle in the current process even if the target PID is invalid. The debug handle remains in place even if the call fails.
3. Repeat step #1.
4. Loop through the updated list of handles for the current process and look for the entry with an object type that wasn't previously flagged in step #2. Assuming this is a single-threaded application, we know that this will be the debug handle previously created by

DebugActiveProcess. We can extract the debug handle type from the ObjectTypeIndex field of this entry.
5. Manually close this temporary debug handle using CloseHandle.

Full program code below:

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

#define SystemExtendedHandleInformation 64
#define STATUS_INFO_LENGTH_MISMATCH 0xC0000004

struct SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX
{
ULONG Object;
ULONG UniqueProcessId;
ULONG HandleValue;
ULONG GrantedAccess;
USHORT CreatorBackTraceIndex;
USHORT ObjectTypeIndex;
ULONG HandleAttributes;
ULONG Reserved;
};

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

SYSTEM_HANDLE_INFORMATION_EX *pGlobal_SystemHandleInfo = NULL;
DWORD dwGlobal_DebugObjectType = 0;

DWORD GetSystemHandleList()
{
DWORD (WINAPI *NtQuerySystemInformation)(DWORD SystemInformationClass,
PVOID SystemInformation, ULONG SystemInformationLength, PULONG ReturnLength);
DWORD dwAllocSize = 0;
DWORD dwStatus = 0;
DWORD dwLength = 0;
BYTE *pSystemHandleInfoBuffer = NULL;

// get NtQuerySystemInformation function ptr
```

```
NtQuerySystemInformation = (unsigned long (__stdcall *)(unsigned long,void *,unsigned
long,unsigned long *))GetProcAddress(GetModuleHandle("ntdll.dll"),
"NtQuerySystemInformation");
if(NtQuerySystemInformation == NULL)
{
return 1;
}

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

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

if(dwAllocSize != 0)
{
// allocate new buffer
pSystemHandleInfoBuffer = (BYTE*)malloc(dwAllocSize);
if(pSystemHandleInfoBuffer == NULL)
{
return 1;
}
}

// get system handle list
dwStatus = NtQuerySystemInformation(SystemExtendedHandleInformation,
(void*)pSystemHandleInfoBuffer, 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 handles created between checks)
dwAllocSize = (dwLength + 1024);
}
else
{
// other error
free(pSystemHandleInfoBuffer);
return 1;
}
}

// store handle info ptr
pGlobal_SystemHandleInfo = (SYSTEM_HANDLE_INFORMATION_EX*)pSystemHandleInfoBuffer;

return 0;
}

DWORD GetDebugHandleObjectType(DWORD *pdwDebugObjectType)
{
DWORD dwHandleTypeList[128];
DWORD dwHandleTypeCount = 0;
DWORD dwCurrentHandleTypeAlreadyExists = 0;
DWORD dwDebugObjectType = 0;
DWORD dwFoundDebugObjectType = 0;

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

// store a list of handle types for this process
for(DWORD i = 0; i < pGlobal_SystemHandleInfo->NumberOfHandles; i++)
{
// check if this handle is for the current process
if(pGlobal_SystemHandleInfo->HandleList[i].UniqueProcessId != GetCurrentProcessId())
{
continue;
}

// check if this handle type already exists in the list
dwCurrentHandleTypeAlreadyExists = 0;
for(DWORD ii = 0; ii < dwHandleTypeCount; ii++)
```

```
{
if(dwHandleTypeList[ii] == pGlobal_SystemHandleInfo->HandleList[i].ObjectTypeIndex)
{
dwCurrentHandleTypeAlreadyExists = 1;
break;
}
}

// ignore if this handle type already exists in the list
if(dwCurrentHandleTypeAlreadyExists != 0)
{
continue;
}

// add this handle type to the list
if(dwHandleTypeCount >= (sizeof(dwHandleTypeList) / sizeof(DWORD)))
{
// not enough space in the list
return 1;
}
dwHandleTypeList[dwHandleTypeCount] = pGlobal_SystemHandleInfo-
>HandleList[i].ObjectTypeIndex;
dwHandleTypeCount++;
}

// DebugActiveProcess will create a debug handle for this process, even if the pid is
invalid
DebugActiveProcess(0);

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

// compare against the old list to find the newly created debug handle type
for(i = 0; i < pGlobal_SystemHandleInfo->NumberOfHandles; i++)
{
// check if this handle is for the current process
if(pGlobal_SystemHandleInfo->HandleList[i].UniqueProcessId != GetCurrentProcessId())
{
continue;
}

// check if this handle type already existed before creating the debug handle
```

```
dwCurrentHandleTypeAlreadyExists = 0;
for(DWORD ii = 0; ii < dwHandleTypeCount; ii++)
{
if(dwHandleTypeList[ii] == pGlobal_SystemHandleInfo->HandleList[i].ObjectTypeIndex)
{
dwCurrentHandleTypeAlreadyExists = 1;
break;
}
}

if(dwCurrentHandleTypeAlreadyExists == 0)
{
// found the debug handle - store the object type
dwFoundDebugObjectType = 1;
dwDebugObjectType = pGlobal_SystemHandleInfo->HandleList[i].ObjectTypeIndex;

// close the debug handle
CloseHandle((HANDLE)pGlobal_SystemHandleInfo->HandleList[i].HandleValue);

break;
}
}

// ensure the debug handle type was found
if(dwFoundDebugObjectType == 0)
{
return 1;
}

// store debug object type
*pdwDebugObjectType = dwDebugObjectType;

return 0;
}

DWORD CheckForDebuggerProcess(DWORD *pdwFoundDebugger, DWORD
*pdwDebuggerPID, HANDLE *phRemoteDebugHandle)
{
DWORD dwFoundDebugger = 0;
DWORD dwDebuggerPID = 0;
HANDLE hRemoteDebugHandle = NULL;

// get system handle list
if(GetSystemHandleList() != 0)
{
```

```
return 1;
}

// check for debug handles
for(DWORD i = 0; i < pGlobal_SystemHandleInfo->NumberOfHandles; i++)
{
if(pGlobal_SystemHandleInfo->HandleList[i].ObjectTypeIndex ==
dwGlobal_DebugObjectType)
{
// found a debugger - store PID
dwFoundDebugger = 1;
dwDebuggerPID = pGlobal_SystemHandleInfo->HandleList[i].UniqueProcessId;
hRemoteDebugHandle = (HANDLE)pGlobal_SystemHandleInfo-
>HandleList[i].HandleValue;

break;
}
}

// store values
*pdwFoundDebugger = dwFoundDebugger;
*pdwDebuggerPID = dwDebuggerPID;
*phRemoteDebugHandle = hRemoteDebugHandle;

return 0;
}

int main()
{
HANDLE hDebuggerProcess = NULL;
HANDLE hRemoteDebugHandle = NULL;
HANDLE hClonedDebugHandle = NULL;
DWORD dwFoundDebugger = 0;
DWORD dwDebuggerPID = 0;

printf("SystemAntiDebug - www.x86matthew.com\n\n");

// find the debug handle object type
if(GetDebugHandleObjectType(&dwGlobal;_DebugObjectType) != 0)
{
return 1;
}

// wait for a debugger
for(;;)
```

```
{
// check if a debugger is currently running on the system
if(CheckForDebuggerProcess(&dwFoundDebugger;, &dwDebuggerPID;,
&hRemoteDebugHandle;) != 0)
{
return 1;
}

// check if a debugger was found
if(dwFoundDebugger != 0)
{
// found a debugger
printf("Found debugger - PID: %u\n", dwDebuggerPID);

// open debugger process
hDebuggerProcess = OpenProcess(PROCESS_DUP_HANDLE, 0, dwDebuggerPID);
if(hDebuggerProcess == NULL)
{
// failed to open process handle
printf("Failed to open debugger process - PID: %u\n", dwDebuggerPID);
}
else
{
// close debugger handle from remote process and terminate the original handle
if(DuplicateHandle(hDebuggerProcess, hRemoteDebugHandle, GetCurrentProcess(),
&hClonedDebugHandle;, 0, 0, DUPLICATE_SAME_ACCESS |
DUPLICATE_CLOSE_SOURCE) == 0)
{
// failed to duplicate handle
printf("Failed to kill debugger - PID: %u\n", dwDebuggerPID);
}
else
{
// closed the target handle in the remote process successfully
printf("Killed debugger successfully - PID: %u\n", dwDebuggerPID);

// close local (cloned) debug handle
CloseHandle(hClonedDebugHandle);
}

// close debugger process handle
CloseHandle(hDebuggerProcess);
}
}
```

```
// wait 500ms before searching again
Sleep(500);
}

return 0;
}
```

With the anti-debug program running in the background, a debugger will fail to attach to any other process: