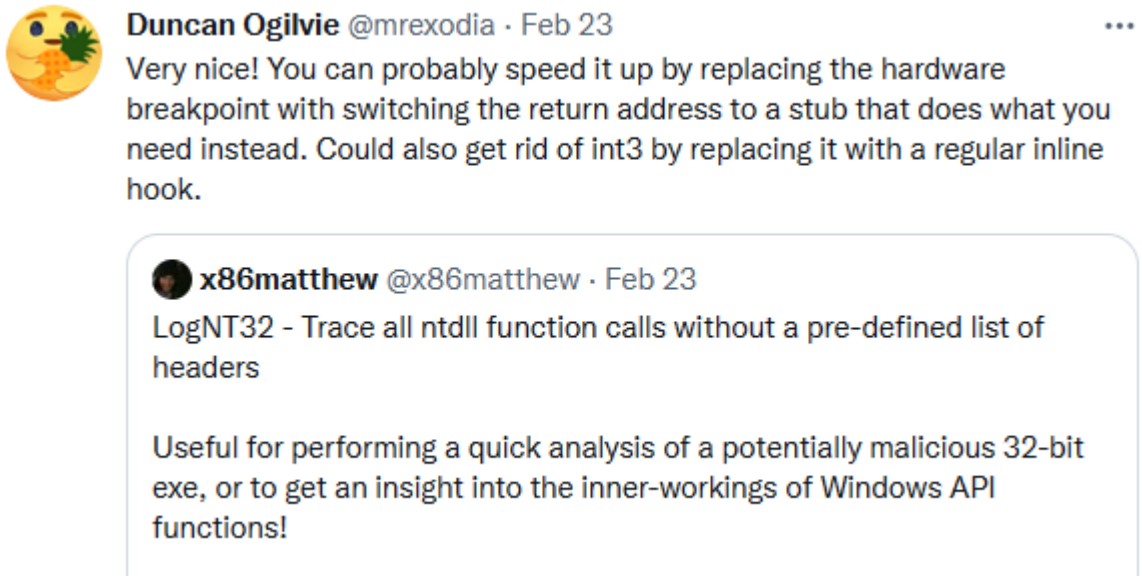
LogNT32 - Part 2 - Return-address hijacking implemented to improve efficiency

Author: x86matthew | **Twitter**: @x86matthew | **E-Mail**: x86matthew@gmail.com

Posted: 25/02/2022

Link: https://www.x86matthew.com/view_post?id=lognt32_p2

This a short follow-up to my previous post - LogNT32. Shortly after the first version was released, @mrexodia (author of x64dbg) made the following suggestions:



This was an interesting idea, so I decided to implement these changes into the original code. The entire exception handler framework has been removed from the module - this simplifies the code and improves overall efficiency.

Firstly, I created two custom function stubs. One handles the entry point of each export call, and one handles the return point:

```
void __declspec(naked) StartCallStub()
{
    _asm
    {
        // preserve eax/ebx/ecx/edx values
        push eax
        push ebx
        push ecx
        push edx

        // get original stack ptr (ignore preserved eax/ebx/ecx/edx values)
        mov ecx, esp
        add ecx, 16

        // log new function call - pass the stack pointer
        push ecx
        call LogNewCall
```

```
        // temporarily store LogNewCall return value in unused stack space
        // (LogNewCall returns the cloned address of the function call)
        mov dword ptr [esp], eax

        // clean up after LogNewCall
        add esp, 4

        // restore register values
        pop edx
        pop ecx
        pop ebx
        pop eax

        // ignore temporary return address
        add esp, 4

        // jump to cloned function
        jmp dword ptr [esp - 24]
    }
}
```

“‘void __declspec(naked) EndCallStub() { _asm {

```
    // preserve eax/ebx/ecx/edx values
    push eax
    push ebx
    push ecx
    push edx

    // get original stack ptr (ignore preserved eax/ebx/ecx/edx values)
    mov ecx, esp
    add ecx, 16

    // log the end of this function call - pass the stack pointer and the
    function return value (eax)
    push eax
    push ecx
    call LogEndCall

    // temporarily store LogEndCall return value in unused stack space
    // (LogEndCall returns the original return address of the function call)
    mov dword ptr [esp], eax

    // clean up after LogEndCall
    add esp, 8

    // restore register values
    pop edx
    pop ecx
    pop ebx
    pop eax

    // continue program execution
    jmp dword ptr [esp - 24]
}
```

}

The code changes are described below.

The following initialisation steps are taken before the main program execution begins:

1. Create a clone of all executable sections within ```ntdll.dll```. Copying only the .text section is r
2. Read the export address table in ```ntdll.dll``` - add a relative ```CALL``` instruction (```0xE8 0x
3. Add a ```JMP``` instruction to the KiUserExceptionDispatcher function - this should redirect straigh
4. Process user-specified filters - set flags to include or exclude specific exports.
5. Main program execution begins.

When an function within ```ntdll.dll``` is called by the target process, the following sequence of even

1. The hooked function calls ```StartCallStub``` as described above.
2. Because the ```StartCallStub``` function was called via a ```CALL``` instruction rather than a ```JM
3. Now that we have the original function address, we can look up the export name. The ```StartCallStub
4. Overwrite the original return address (of the hooked function) with the address of the ```EndCallStu
5. Jump to the cloned version of the original function and continue execution.
6. After the target function has completed, it will return to our ```EndCallStub``` function due to the
7. The ```EndCallStub``` function calls ```LogEndCall```, which calculates the number of parameters pas
8. Jump to the original return address and continue execution.

Note: Register values must be preserved in the custom stubs, because some ```ntdll.dll``` functions suc

In testing, this method does appear to have some performance benefits, although the main bottleneck see

New binaries:
See LogNT32_V2.zip

Full code below:

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

#define MAX_EXECUTABLE_SECTIONS 32 #define CONFIG_FILE_TIMEOUT_SECONDS 10 #define MAX_FILTER_LENGTH (16 * 1024) #define MAX_CALL_DEPTH 32 #define MAX_NTDLL_EXPORT_LIST_COUNT 4096 #define KERNEL_CALLBACK_CALL_STACK_LIST_COUNT 32 #define SCAN_STRING_PARAM_MAX_LENGTH 1024 #define MAX_THREAD_COUNT 1024 #define MAX_PARAM_COUNT 32 #define MAX_LOG_ENTRY_LENGTH (16 * 1024)

struct ExecutableSectionInfoStruct { DWORD dwStartAddr; DWORD dwLength; DWORD dwOriginalProtect; };

struct ANSI_STRING { USHORT Length; USHORT MaximumLength; PCHAR Buffer; };

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 CLIENT_ID { HANDLE UniqueProcess; HANDLE UniqueThread; };

struct CfgDataStruct { char szLogFilePath[512]; char szFilter[MAX_FILTER_LENGTH]; DWORD dwStringOnly; };

struct ThreadCallEntryStruct { char szExportName[64]; DWORD dwStackPtr; DWORD dwReturnAddress; DWORD dwParamList[MAX_PARAM_COUNT]; };

struct CallStackStruct { ThreadCallEntryStruct ThreadCalls[MAX_CALL_DEPTH]; DWORD dwCallDepth; DWORD dwLoggedKernelCallbackStart; };

struct ProcessThreadDataStruct { DWORD dwInUse; DWORD dwThreadID; HANDLE hThread; CallStackStruct CallStack; CallStackStruct KernelCallbackCallStackList[KERNEL_CALLBACK_CALL_STACK_LIST_COUNT]; DWORD dwKernelCallbackDepth; };

struct ExportEntryStruct { char szExportName[64]; DWORD dwAddr; DWORD dwRedirectExport; DWORD dwLogExport; };

struct ClonedExportPtrEntryStruct { char *pExportName; void **pFunctionPtr; };

// ntdll ptrs DWORD (WINAPI *RtlEnterCriticalSection)(CRITICAL_SECTION* pCriticalSection); DWORD (WINAPI *RtlLeaveCriticalSection)(CRITICAL_SECTION* pCriticalSection); DWORD (WINAPI *NtOpenThread)(HANDLE* ThreadHandle, DWORD DesiredAccess, OBJECT_ATTRIBUTES *ObjectAttributes, CLIENT_ID* ClientId);

// cloned export ptr list ClonedExportPtrEntryStruct Global_ClonedExportPtrList[] = { { "RtlEnterCriticalSection", (void)**&RtlEnterCriticalSection }, { "RtlLeaveCriticalSection", (void)**&RtlLeaveCriticalSection }, { "NtOpenThread", (void**)&NtOpenThread }, };

char *pGlobal_DefaultIgnoreExportList[] = { "RtlTryEnterCriticalSection", "RtlEnterCriticalSection", "RtlLeaveCriticalSection", "RtlInitializeCriticalSection", "RtlInitializeCriticalSectionEx", "RtlInitializeCriticalSectionAndSpinCount", "RtlDeleteCriticalSection", "RtlAllocateHeap", "RtlFreeHeap", "RtlLockHeap", "RtlUnlockHeap", "RtlRestoreLastWin32Error", "RtlUpcaseUnicodeChar", "RtlDowncaseUnicodeChar", "RtlAnsiCharToUnicodeChar", };

// global vars ExecutableSectionInfoStruct Global_NtdllExecutableSectionList[MAX_EXECUTABLE_SECTIONS]; DWORD dwGlobal_NtdllExecutableSectionListCount = 0; BYTE *pGlobal_NtdllBase = NULL; BYTE* pGlobal_NtdllCodeStart = NULL; DWORD dwGlobal_NtdllCodeLength = 0; BYTE *pGlobal_NtdllClone = NULL; CRITICAL_SECTION Global_LogCriticalSection; HANDLE hGlobal_LogFile = NULL; char szGlobal_BaseDirectory[512]; CfgDataStruct Global_CfgData; DWORD dwGlobal_Wow64 = 0; BYTE* pGlobal_32BitUser32CallbackReturnClone = NULL; DWORD dwGlobal_32BitUser32CallbackReturnHooked = 0; DWORD dwGlobal_Orig32BitUser32CallbackReturnAddr = 0; ExportEntryStruct Global_NtdllExportList[MAX_NTDL DWORD dwGlobal_NtdllExportListCount = 0; ProcessThreadDataStruct Global_ProcessThreadData[MAX_THREAD_COU DWORD dwGlobal_InitialThreadID = 0; HANDLE hGlobal_InitialThread = NULL; char *pGlobal_IgnoreListFileEntries = NULL; char szGlobal_LogEntryBuffer[MAX_LOG_ENTRY_LENGTH]; DWORD dwGlobal_StartCallStubAddr = 0; DWORD dwGlobal_EndCallStubAddr = 0;

// User32CallbackReturn function: // mov eax,dword ptr ss:[esp+4], int 0x2B, retn 4 BYTE Global_32BitUser32CallbackReturnCode[] = { 0x8B, 0x44, 0x24, 0x04, 0xCD, 0x2B, 0xC2, 0x04, 0x00 };

IMAGE_NT_HEADERS *GetModuleNtHeader(DWORD dwModuleBase) { IMAGE_DOS_HEADER* pDosHeader = NULL; IMAGE_NT_HEADERS *pNtHeader = NULL;

```
// get dos header ptr (start of module)
pDosHeader = (IMAGE_DOS_HEADER*)dwModuleBase;
if(pDosHeader->e_magic != IMAGE_DOS_SIGNATURE)
{
    return NULL;
}

// get nt header ptr
pNtHeader = (IMAGE_NT_HEADERS*)((BYTE*)pDosHeader + pDosHeader->e_lfanew);
if(pNtHeader->Signature != IMAGE_NT_SIGNATURE)
{
    return NULL;
}
```

```
return pNtHeader;
}
```

IMAGE_EXPORT_DIRECTORY *GetModuleExportDirectory(DWORD dwModuleBase) { IMAGE_NT_HEADERS* pNtHeader = NULL; IMAGE_EXPORT_DIRECTORY *pExportDirectory = NULL;

```
// get nt header ptr
pNtHeader = GetModuleNtHeader(dwModuleBase);
if(pNtHeader == NULL)
{
    return NULL;
}

// ensure at least one data directory exists
if(pNtHeader->OptionalHeader.NumberOfRvaAndSizes == 0)
{
    return NULL;
}

// get export directory ptr
pExportDirectory = (IMAGE_EXPORT_DIRECTORY*)(dwModuleBase +
pNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);

return pExportDirectory;
}
```

DWORD EnumModuleExportNames(DWORD dwModuleBase, DWORD (*pCallback)(char* pExportName, DWORD dwExportAddr, BYTE *pParam), BYTE* pParam) { IMAGE_EXPORT_DIRECTORY *pImageExports = NULL; DWORD* pdwFunctionList = NULL; DWORD *pdwNameList = NULL; WORD* pwOrdinalList = NULL; char *pCurrExportName = NULL; DWORD dwCurrExportAddr = 0;

```
// get exports
pImageExports = GetModuleExportDirectory(dwModuleBase);
if(pImageExports == NULL)
{
    return 1;
}


// ensure at least 1 function is exported by name
if(pImageExports->NumberOfNames != 0)
{
    // get ptrs
    pdwFunctionList = (DWORD*)(dwModuleBase + pImageExports->AddressOfFunctions);
    pdwNameList = (DWORD*)(dwModuleBase + pImageExports->AddressOfNames);
    pwOrdinalList = (WORD*)(dwModuleBase + pImageExports->AddressOfNameOrdinals);

    // loop through all export names
    for(DWORD i = 0; i < pImageExports->NumberOfNames; i++)
    {
        // get ptrs for current export
        pCurrExportName = (char*)(dwModuleBase + pdwNameList[i]);
        dwCurrExportAddr = (DWORD)(dwModuleBase + pdwFunctionList[pwOrdinalList[i]]);

        // callback
```

```
        if(pCallback(pCurrExportName, dwCurrExportAddr, pParam) != 0)
        {
            // error
            return 1;
        }
    }
}
return 0;
```

}

DWORD GetModuleCodeSection(DWORD dwModuleBase, DWORD *pdwCodeSectionStart, DWORD* pdwCodeSectionLength) { IMAGE_NT_HEADERS *pNtHeader = NULL; IMAGE_SECTION_HEADER* pCurrSectionHeader = NULL; char szCurrSectionName[16]; DWORD dwFound = 0; DWORD dwCodeSectionStart = 0; DWORD dwCodeSectionLength = 0;

```
// get nt header ptr
pNtHeader = GetModuleNtHeader(dwModuleBase);
if(pNtHeader == NULL)
{
    return 1;
}

// loop through all sections
for(DWORD i = 0; i < pNtHeader->FileHeader.NumberOfSections; i++)
{
    // get current section header
    pCurrSectionHeader = (IMAGE_SECTION_HEADER*)((BYTE*)pNtHeader +
    sizeof(IMAGE_NT_HEADERS) + (i * sizeof(IMAGE_SECTION_HEADER)));

    // pCurrSectionHeader->Name is not null terminated if all 8 characters
    are used - copy it to a larger local buffer
    memset(szCurrSectionName, 0, sizeof(szCurrSectionName));
    memcpy(szCurrSectionName, pCurrSectionHeader->Name, sizeof(pCurrSectionHeader->Name));

    // check if this is the main code section
    if(strcmp(szCurrSectionName, ".text") == 0)
    {
        // found code section
        dwFound = 1;
        dwCodeSectionStart = dwModuleBase + pCurrSectionHeader->VirtualAddress;
        dwCodeSectionLength = pCurrSectionHeader->SizeOfRawData;
        break;
    }
}

// ensure the code section was found
if(dwFound == 0)
{
    return 1;
}

// store values
*pdwCodeSectionStart = dwCodeSectionStart;
*pdwCodeSectionLength = dwCodeSectionLength;
```

```
return 0;
```

}

ExportEntryStruct *GetExportEntryByAddr(DWORD dwExportAddr) {

```
// find export name by address in Global_NtdllExportList
for(DWORD i = 0; i < dwGlobal_NtdllExportListCount; i++)
{
    if(Global_NtdllExportList[i].dwAddr == dwExportAddr)
    {
        // return export name
        return &Global_NtdllExportList[i];
    }
}

// not found
return NULL;
```

}

ExportEntryStruct *GetExportEntryByName(char* pExportName) { DWORD dwFound = 0; DWORD dwExportAddr = 0;

```
// find export address by name in Global_NtdllExportList
for(DWORD i = 0; i < dwGlobal_NtdllExportListCount; i++)
{
    if(strcmp(Global_NtdllExportList[i].szExportName, pExportName) == 0)
    {
        // found export
        return &Global_NtdllExportList[i];
    }
}

// not found
return NULL;
```

}

DWORD GetClonedAddr(DWORD dwInputAddr, DWORD *pdwOutputAddr) { DWORD dwDelta = 0;

```
// get ptr delta
dwDelta = (dwInputAddr - (DWORD)pGlobal_NtdllCodeStart);
if(dwDelta > dwGlobal_NtdllCodeLength)
{
    return 1;
}

// store output addr
*pdwOutputAddr = (DWORD)pGlobal_NtdllClone + dwDelta;
return 0;
```

}

DWORD GetClonedAddrByExportName(char *pExportName, DWORD* pdwOutputAddr) { ExportEntryStruct *pExportEntry = NULL;

```
// get export addr by name
pExportEntry = GetExportEntryByName(pExportName);
```

```
if(pExportEntry == NULL)
{
    return 1;
}

// get cloned addr
if(GetClonedAddr(pExportEntry->dwAddr, pdwOutputAddr) != 0)
{
    return 1;
}

return 0;
```

}

DWORD Log_WriteRawData(char *pData) { DWORD dwBytesWritten = 0;

```
// write data to output file
WriteFile(hGlobal_LogFile, (void*)pData, strlen(pData), &dwBytesWritten, NULL);
return 0;
```

}

DWORD Log_EndLogEntry() { DWORD dwIgnoreEntry = 0;

```
// check if "string only" mode is enabled
if(Global_CfgData.dwStringOnly != 0)
{
    // only display log entries that contain string parameters
    dwIgnoreEntry = 1;

    // check if this is a 'CALL_END' log entry
    if(strstr(szGlobal_LogEntryBuffer, "[CALL_END]") != NULL)
    {
        // check if this line contains a string value
        if(strstr(szGlobal_LogEntryBuffer, "<") != NULL)
        {
            // display line
            dwIgnoreEntry = 0;
        }
    }
}



if(dwIgnoreEntry == 0)
{
    // write line to file
    Log_WriteRawData(szGlobal_LogEntryBuffer);
    Log_WriteRawData("\r\n");
}

// reset log entry buffer
szGlobal_LogEntryBuffer[0] = '\0';

return 0;
```

}

DWORD Log_AppendLogEntry(char *pLogText) { DWORD dwOrigLength = 0; DWORD dwAppendLength = 0;

```
// calculate original length
dwOrigLength = strlen(szGlobal_LogEntryBuffer);

// calculate length of text to appen
dwAppendLength = strlen(pLogText);

// validate total length
if((dwOrigLength + dwAppendLength) >= sizeof(szGlobal_LogEntryBuffer))
{
    return 1;
}


// append text to buffer (including null terminator)
memcpy((void*)&szGlobal_LogEntryBuffer[dwOrigLength], (void*)pLogText,
dwAppendLength + 1);
return 0;
```

}

DWORD Log_NewLogEntry(DWORD dwThreadID, char *pLogText) { char szOutput[64]; SYSTEMTIME LocalTime;

```
// reset log entry buffer
szGlobal_LogEntryBuffer[0] = '\0';

// get local time
GetLocalTime(&LocalTime);

// write line to output file
memset(szOutput, 0, sizeof(szOutput));
_snprintf(szOutput, sizeof(szOutput) - 1, "[%02u/%02u/%04u %02u:%02u:%02u]
[TID:%u] ", LocalTime.wDay, LocalTime.wMonth, LocalTime.wYear, LocalTime.wHour,
LocalTime.wMinute, LocalTime.wSecond, dwThreadID);

// write timestamp and thread ID
Log_AppendLogEntry(szOutput);

// write log text
Log_AppendLogEntry(pLogText);
return 0;
```

}

HANDLE OpenThreadSyncHandle(DWORD dwThreadID) { HANDLE hThread = NULL; OBJECT_ATTRIBUTES ObjectAttributes; CLIENT_ID ClientId;

```
// open thread with SYNCHRONIZE access
memset((void*)&ObjectAttributes, 0, sizeof(ObjectAttributes));
ObjectAttributes.Length = sizeof(ObjectAttributes);
ClientId.UniqueProcess = 0;
ClientId.UniqueThread = (HANDLE)dwThreadID;
if(NtOpenThread(&hThread, SYNCHRONIZE, &ObjectAttributes, &ClientId) != 0)
```

```
{
    return NULL;
}

return hThread;
}
```

ProcessThreadDataStruct *GetProcessThreadData(DWORD dwThreadID) { // find stored details for this thread ID for(DWORD i = 0; i < sizeof(Global_ProcessThreadData) / sizeof(Global_ProcessThreadData[0]); i++) { if(Global_ProcessThreadData[i].dwInUse == 0) { continue; }

```
    if(Global_ProcessThreadData[i].dwThreadID == dwThreadID)
    {
        // found thread entry
        return &Global_ProcessThreadData[i];
    }
}

return NULL;
}
```

ProcessThreadDataStruct *AddProcessThreadData(DWORD dwThreadID) { HANDLE hThread = NULL; DWORD dwStatus = 0;

```
// check for threads that have exited
for(DWORD i = 0; i < sizeof(Global_ProcessThreadData) /
sizeof(Global_ProcessThreadData[0]); i++)
{
    if(Global_ProcessThreadData[i].dwInUse == 0)
    {
        continue;
    }

    // check if this thread has exited
    dwStatus = WaitForSingleObject(Global_ProcessThreadData[i].hThread, 0);
    if(dwStatus == 0)
    {
        // thread exited - no longer in use
        CloseHandle(Global_ProcessThreadData[i].hThread);
        Global_ProcessThreadData[i].dwInUse = 0;
    }
}

// get a handle to the current thread
hThread = OpenThreadSyncHandle(dwThreadID);
if(hThread == NULL)
{
    return NULL;
}

// add current thread details - find a free entry in the list
for(i = 0; i < sizeof(Global_ProcessThreadData) /
sizeof(Global_ProcessThreadData[0]); i++)
{
    if(Global_ProcessThreadData[i].dwInUse != 0)
```

```
    {
        continue;
    }

    // found an unused entry - store details
    memset((void*)&Global_ProcessThreadData[i], 0,
    sizeof(ProcessThreadDataStruct));
    Global_ProcessThreadData[i].dwInUse = 1;
    Global_ProcessThreadData[i].dwThreadID = dwThreadID;
    Global_ProcessThreadData[i].hThread = hThread;

    // return ptr to new entry
    return &Global_ProcessThreadData[i];
}

// failed to add thread details - close handle
CloseHandle(hThread);
// error
return NULL;
```

}

DWORD Add32BitUser32CallbackReturnHook() { DWORD dwInstructionCount = 0; DWORD dwCurrSearchPos = 0; DWORD dwBytesRemaining = 0; DWORD dwFoundAddr = 0; DWORD dwCodeSectionStart = 0; DWORD dwCodeSectionLength = 0; BYTE *pUser32Module = NULL; DWORD dwOriginalProtection = 0; DWORD dwTempProtection = 0; DWORD dwRelativeAddr = 0;

```
// get user32.dll base address
pUser32Module = (BYTE*)GetModuleHandle("user32.dll");
if(pUser32Module == NULL)
{
    return 1;
}

// find user32 code section range
if(GetModuleCodeSection((DWORD)pUser32Module, &dwCodeSectionStart,
&dwCodeSectionLength) != 0)
{
    return 1;
}

// find this instruction in the ntdll code section
dwCurrSearchPos = dwCodeSectionStart;
dwBytesRemaining = dwCodeSectionLength;
dwFoundAddr = 0;
for(;;)
{
    // check if the end of the code section has been reached
    if(dwBytesRemaining < sizeof(Global_32BitUser32CallbackReturnCode))
    {
        break;
    }

    // check if the instruction exists here
    if(memcmp((void*)dwCurrSearchPos, (void*)Global_32BitUser32CallbackReturnCode,
```

```
    sizeof(Global_32BitUser32CallbackReturnCode)) == 0)
    {
        dwFoundAddr = dwCurrSearchPos;
        break;
    }

    // update search indexes
    dwCurrSearchPos++;
    dwBytesRemaining--;
}

// ensure the opcode was found
if(dwFoundAddr == 0)
{
    return 1;
}

// store original address
dwGlobal_Orig32BitUser32CallbackReturnAddr = dwFoundAddr;


// temporarily set memory to writable
if(VirtualProtect((void*)dwGlobal_Orig32BitUser32CallbackReturnAddr, 1, PAGE_EXECUTE_READWRITE, &dwOrig
{
    return 1;
}

// hook User32CallbackReturn - call StartCallStub function
*(BYTE*)dwGlobal_Orig32BitUser32CallbackReturnAddr = 0xE8;
dwRelativeAddr = dwGlobal_StartCallStubAddr - (dwGlobal_Orig32BitUser32CallbackReturnAddr + 5);
*(DWORD*)(dwGlobal_Orig32BitUser32CallbackReturnAddr + 1) = dwRelativeAddr;

// restore original memory protection
if(VirtualProtect((void*)dwGlobal_Orig32BitUser32CallbackReturnAddr, 1, dwOriginalProtection, &dwTempPr
{
    return 1;
}
return 0;
```

}

DWORD NewCall(DWORD dwThreadID, char *pExportName, DWORD dwStackPtr) { ProcessThreadDataStruct* pProcessThreadData = NULL; DWORD dwReturnAddress = 0; char szOutput[2048]; ThreadCallEntryStruct *pThreadCallEntry = NULL; char szCallDepthStr[MAX_CALL_DEPTH + 4]; CallStackStruct* pCurrCallStack = NULL; DWORD dwNewFunctionCall = 0;

```
// check if the current thread already exists in the data list
pProcessThreadData = GetProcessThreadData(dwThreadID);
if(pProcessThreadData == NULL)
{
    // thread doesn't exist - add a new entry
    pProcessThreadData = AddProcessThreadData(dwThreadID);
    if(pProcessThreadData == NULL)
    {
        return 1;
```

```
    }
}


// get call return address
dwReturnAddress = *(DWORD*)dwStackPtr;

// check if this is a kernel callback (no return address)
if(strcmp(pExportName, "KiUserCallbackDispatcher") == 0)
{
    // kernel callback started - entered user-mode callback from kernel-mode
    if(pProcessThreadData->dwKernelCallbackDepth >= KERNEL_CALLBACK_CALL_STACK_LIST_COUNT)
    {
        // kernel callback list is at maximum length
        return 1;
    }


    // create new kernel callback call stack
    pCurrCallStack = &pProcessThreadData->
    KernelCallbackCallStackList[pProcessThreadData->dwKernelCallbackDepth];
    pCurrCallStack->dwCallDepth = 0;
    pProcessThreadData->dwKernelCallbackDepth++;

    // don't write a log entry until a function is called within this callback
    pCurrCallStack->dwLoggedKernelCallbackStart = 0;

    // check if this is a 32-bit OS
    if(dwGlobal_Wow64 == 0)
    {
        if(dwGlobal_32BitUser32CallbackReturnHooked == 0)
        {
            // add a hook to the User32CallbackReturn function within user32.dll
            (32-bit only - 64-bit always uses NtCallbackReturn)
            if(Add32BitUser32CallbackReturnHook() != 0)
            {
                return 1;
            }

            // set flag
            dwGlobal_32BitUser32CallbackReturnHooked = 1;
        }
    }
}

else if(strcmp(pExportName, "RtlRaiseException") == 0)
{
    // user exception raised - add entry to log but don't add a new call
     to the stack (RtlRaiseException never returns)
    Log_NewLogEntry(dwThreadID, "[EXCEPTION_RAISED]");
    Log_EndLogEntry();
}
else if(strcmp(pExportName, "NtCallbackReturn") == 0)
{
```

```
    // kernel callback end - return to kernel mode
    if(pProcessThreadData->dwKernelCallbackDepth == 0)
    {
        // not currently inside a kernel callback - error
        return 1;
    }

    // check if the entry to this kernel callback was logged - don't log the exit if not
    if(pProcessThreadData->
    KernelCallbackCallStackList[pProcessThreadData->dwKernelCallbackDepth
    - 1].dwLoggedKernelCallbackStart != 0)
    {
        // write log entry
        memset(szOutput, 0, sizeof(szOutput));
        _snprintf(szOutput, sizeof(szOutput) - 1, "[KERNEL_CALLBACK_END:%u]",
        pProcessThreadData->dwKernelCallbackDepth);
        Log_NewLogEntry(dwThreadID, szOutput);
        Log_EndLogEntry();
    }

    // revert to previous call stack
    pProcessThreadData->dwKernelCallbackDepth--;

    // set a hardware breakpoint on the last call in the previous stack chain
    if(pProcessThreadData->dwKernelCallbackDepth == 0)
    {
        // use standard call stack - not currently within a kernel callback
        pCurrCallStack = &pProcessThreadData->CallStack;
    }
    else
    {
        // currently within a kernel callback - get the latest call stack in the chain
        pCurrCallStack = &pProcessThreadData->
        KernelCallbackCallStackList[pProcessThreadData->dwKernelCallbackDepth - 1];
    }
}
else if(dwReturnAddress == 0)
{
    // current function call has no return address - don't wait for it to return
    memset(szOutput, 0, sizeof(szOutput));
    _snprintf(szOutput, sizeof(szOutput) - 1, "[CALL_NO_RETN] %s", pExportName);
    Log_NewLogEntry(dwThreadID, szOutput);
    Log_EndLogEntry();
}
else
{
    // new function call - get current call-stack
    if(pProcessThreadData->dwKernelCallbackDepth == 0)
    {
        // use standard call stack - not currently within a kernel callback
        pCurrCallStack = &pProcessThreadData->CallStack;
    }
    else
    {
```

```
    // currently within a kernel callback - get the latest call stack in the chain
    pCurrCallStack = &pProcessThreadData->
    KernelCallbackCallStackList[pProcessThreadData->dwKernelCallbackDepth - 1];
}

// this is a new function call - add it to the call stack
if(pCurrCallStack->dwCallDepth >= MAX_CALL_DEPTH)
{
    // current call stack list is at maximum length
    return 1;
}

// check if this is the first function call within any kernel callbacks
in the chain - write to log if necessary
for(DWORD i = 0; i < pProcessThreadData->dwKernelCallbackDepth; i++)
{
    // check if this kernel callback entry has already been logged
    if(pProcessThreadData->KernelCallbackCallStackList[i].dwLoggedKernelCallbackStart == 0)
    {
        // kernel callback entry hasn't been logged yet - add a log entry
        memset(szOutput, 0, sizeof(szOutput));
        _snprintf(szOutput, sizeof(szOutput) - 1, "[KERNEL_CALLBACK_START:%u]", i + 1);
        Log_NewLogEntry(dwThreadID, szOutput);
        Log_EndLogEntry();

        // set flag
        pProcessThreadData->KernelCallbackCallStackList[i].dwLoggedKernelCallbackStart = 1;
    }
}


// generate call stack depth string - use '-' characters to display the
depth of the current function
memset(szCallDepthStr, 0, sizeof(szCallDepthStr));
memset(szCallDepthStr, '-', pCurrCallStack->dwCallDepth);
if(pCurrCallStack->dwCallDepth != 0)
{
    // add a space character after the dashes
    szCallDepthStr[pCurrCallStack->dwCallDepth] = ' ';
}

// get next entry in call stack
pThreadCallEntry = &pCurrCallStack->
ThreadCalls[pCurrCallStack->dwCallDepth];

// add current function details to call stack
memset((void*)pThreadCallEntry, 0, sizeof(ThreadCallEntryStruct));
strncpy(pThreadCallEntry->szExportName, pExportName,
sizeof(pThreadCallEntry->szExportName) - 1);
pThreadCallEntry->dwStackPtr = dwStackPtr;
pThreadCallEntry->dwReturnAddress = dwReturnAddress;

// store param list
for(i = 0; i < MAX_PARAM_COUNT; i++)
```

```
    {
        pThreadCallEntry->dwParamList[i] = *(DWORD*)(dwStackPtr +
        ((i + 1) * sizeof(DWORD)));
    }

    // validate call stack
    if(pCurrCallStack->dwCallDepth != 0)
    {
        if(dwStackPtr > pCurrCallStack->
        ThreadCalls[pCurrCallStack->dwCallDepth - 1].dwStackPtr)
        {
            // invalid call stack
            Log_NewLogEntry(dwThreadID, "[INVALID_CALL_STACK]");
            Log_EndLogEntry();
            return 1;
        }
    }

    // increase call stack depth
    pCurrCallStack->dwCallDepth++;

    // write log entry
    memset(szOutput, 0, sizeof(szOutput));
    _snprintf(szOutput, sizeof(szOutput) - 1,
    "[CALL_BEGIN]   %s%s [RETN_ADDR:0x%08X]", szCallDepthStr, pExportName, dwReturnAddress);
    Log_NewLogEntry(dwThreadID, szOutput);
    Log_EndLogEntry();
}
return 0;
```

}

DWORD ReadLocalMemory(DWORD dwAddr, BYTE *pOutput, DWORD dwLength) { // read memory from the local process // this is safer than reading directly because it avoids exceptions if the memory is free'd after validating the read protection if(ReadProcessMemory(GetCurrentProcess(), (void*)dwAddr, pOutput, dwLength, NULL) == 0) { return 1; } return 0; }

DWORD CheckAnsiString(DWORD dwCurrPtr, char *pParamString, DWORD dwMaxLength) { ANSI_STRING AnsiString; char szAnsiString[SCAN_STRING_PARAM_MAX_LENGTH]; DWORD dwAnsiStringLength = 0;

```
// attempt to read ANSI_STRING structure
if(ReadLocalMemory(dwCurrPtr, (BYTE*)&AnsiString, sizeof(AnsiString)) != 0)
{
    return 1;
}

// validate length
if(AnsiString.Length == 0)
{
    return 1;
}

// validate length
if(AnsiString.Length > (SCAN_STRING_PARAM_MAX_LENGTH - 1))
{
```

```
    return 1;
}

// read string value
memset(szAnsiString, 0, sizeof(szAnsiString));
if(ReadLocalMemory((DWORD)AnsiString.Buffer, (BYTE*)szAnsiString,
AnsiString.Length) != 0)
{
    return 1;
}

// validate length matches expected value
dwAnsiStringLength = strlen(szAnsiString);
if(dwAnsiStringLength != AnsiString.Length)
{
    return 1;
}

// validate string characters
for(DWORD i = 0; i < dwAnsiStringLength; i++)
{
    if(szAnsiString[i] > 0x7F)
    {
        return 1;
    }
    else if(szAnsiString[i] < 0x20)
    {
        if(szAnsiString[i] == '\r' || szAnsiString[i] == '\n')
        {
            szAnsiString[i] = '.';
        }
        else
        {
            return 1;
        }
    }
}

// store string value
strncpy(pParamString, szAnsiString, dwMaxLength);
return 0;
```

}

DWORD CheckUnicodeString(DWORD dwCurrPtr, char *pParamString, DWORD dwMaxLength) { UNICODE_STRING UnicodeString; wchar_t wszString[SCAN_STRING_PARAM_MAX_LENGTH]; char szAnsiString[SCAN_STRING_PARAM_MAX_LENGTH]; DWORD dwAnsiStringLength = 0;

```
// attempt to read UNICODE_STRING structure
if(ReadLocalMemory(dwCurrPtr, (BYTE*)&UnicodeString, sizeof(UnicodeString)) != 0)
{
    return 1;
}

// validate length
```

```
if(UnicodeString.Length == 0)
{
    return 1;
}

// validate length
if(UnicodeString.Length % 2 != 0)
{
    return 1;
}

// validate length
if(UnicodeString.Length > ((SCAN_STRING_PARAM_MAX_LENGTH - 1) * sizeof(wchar_t)))
{
    return 1;
}

// read string value
memset(wszString, 0, sizeof(wszString));
if(ReadLocalMemory((DWORD)UnicodeString.Buffer, (BYTE*)wszString,
UnicodeString.Length) != 0)
{
    return 1;
}

// convert widechar string to ansi string
memset(szAnsiString, 0, sizeof(szAnsiString));
wcstombs(szAnsiString, wszString, sizeof(szAnsiString) - 1);

// validate length matches expected value
dwAnsiStringLength = strlen(szAnsiString);
if(dwAnsiStringLength != UnicodeString.Length / sizeof(wchar_t))
{
    return 1;
}

// validate string characters
for(DWORD i = 0; i < dwAnsiStringLength; i++)
{
    if(szAnsiString[i] > 0x7F)
    {
        return 1;
    }
    else if(szAnsiString[i] < 0x20)
    {
        if(szAnsiString[i] == '\r' || szAnsiString[i] == '\n')
        {
            szAnsiString[i] = '.';
        }
        else
        {
            return 1;
        }
    }
```

```
}

// store string value
strncpy(pParamString, szAnsiString, dwMaxLength);
return 0;
}
```

DWORD CheckObjectAttributes(DWORD dwCurrPtr, char *pParamString, DWORD dwMaxLength) { OBJECT_ATTRIBUTES ObjectAttributes;

```
// attempt to read OBJECT_ATTRIBUTES structure
if(ReadLocalMemory(dwCurrPtr, (BYTE*)&ObjectAttributes,
sizeof(ObjectAttributes)) != 0)
{
    return 1;
}

// validate length
if(ObjectAttributes.Length < sizeof(OBJECT_ATTRIBUTES))
{
    return 1;
}

// attempt to read ObjectName (UNICODE_STRING) value
if(CheckUnicodeString((DWORD)ObjectAttributes.ObjectName,
pParamString, dwMaxLength) != 0)
{
    return 1;
}
return 0;
}
```

DWORD CallComplete(DWORD dwThreadID, DWORD dwStackPtr, DWORD dwReturnValue, DWORD *pdwOrigReturnAddress) { ProcessThreadDataStruct* pProcessThreadData = NULL; DWORD dwReturnAddress = 0; char szOutput[2048]; ThreadCallEntryStruct *pThreadCallEntry = NULL; char szCallDepthStr[MAX_CALL_DEPTH + 4]; char szCurrParam[32]; DWORD dwStringFound = 0; DWORD dwCurrParam = 0; char szCurrParamStringValue[SCAN_STRING_PARAM_MAX_LENGTH]; DWORD dwParamCount = 0; CallStackStruct* pCurrCallStack = NULL; DWORD dwOrigReturnAddress = 0;

```
// find the current thread in the data list
pProcessThreadData = GetProcessThreadData(dwThreadID);
if(pProcessThreadData == NULL)
{
    return 1;
}

// get current call-stack
if(pProcessThreadData->dwKernelCallbackDepth == 0)
{
    // use standard call stack - not currently within a kernel callback
    pCurrCallStack = &pProcessThreadData->CallStack;
}
else
{
    // currently within a kernel callback - get the latest call stack in the chain
```

```
    pCurrCallStack = &pProcessThreadData->
    KernelCallbackCallStackList[pProcessThreadData->dwKernelCallbackDepth - 1];
}

// ensure a call has already started
if(pCurrCallStack->dwCallDepth == 0)
{
    return 1;
}

// get last call in the call stack
pThreadCallEntry = &pCurrCallStack->ThreadCalls[pCurrCallStack->dwCallDepth - 1];

// store original return address
dwOrigReturnAddress = pThreadCallEntry->dwReturnAddress;

// previous call has returned - reduce call stack depth
pCurrCallStack->dwCallDepth--;

// generate call stack depth string - use '-' characters to display the depth
of the current function
memset(szCallDepthStr, 0, sizeof(szCallDepthStr));
memset(szCallDepthStr, '-', pCurrCallStack->dwCallDepth);
if(pCurrCallStack->dwCallDepth != 0)
{
    szCallDepthStr[pCurrCallStack->dwCallDepth] = ' ';
}

// calculate the number of parameters for the previous function
dwParamCount = ((dwStackPtr - pThreadCallEntry->dwStackPtr) -
sizeof(DWORD)) / sizeof(DWORD);
if(dwParamCount > MAX_PARAM_COUNT)
{
    return 1;
}

// write log entry
memset(szOutput, 0, sizeof(szOutput));
_snprintf(szOutput, sizeof(szOutput) - 1, "[CALL_END]     %s%s(",
szCallDepthStr, pThreadCallEntry->szExportName);
Log_NewLogEntry(dwThreadID, szOutput);

// get param values from stack
for(DWORD i = 0; i < dwParamCount; i++)
{
    // add comma between parameter values
    if(i != 0)
    {
        Log_AppendLogEntry(", ");
    }

    // get the current param value from the stack
    dwCurrParam = pThreadCallEntry->dwParamList[i];
```

```
    // write the current param value to the log
    memset(szCurrParam, 0, sizeof(szCurrParam));
    _snprintf(szCurrParam, sizeof(szCurrParam) - 1, "0x%08X", dwCurrParam);
    Log_AppendLogEntry(szCurrParam);

    // check if this parameter contains a string value
    dwStringFound = 0;
    memset(szCurrParamStringValue, 0, sizeof(szCurrParamStringValue));
    if(CheckObjectAttributes(dwCurrParam, szCurrParamStringValue,
    sizeof(szCurrParamStringValue) - 1) == 0)
    {
        // found OBJECT_ATTRIBUTES parameter
        dwStringFound = 1;
    }
    else if(CheckAnsiString(dwCurrParam, szCurrParamStringValue,
    sizeof(szCurrParamStringValue) - 1) == 0)
    {
        // found ANSI_STRING parameter
        dwStringFound = 1;
    }
    else if(CheckUnicodeString(dwCurrParam, szCurrParamStringValue,
    sizeof(szCurrParamStringValue) - 1) == 0)
    {
        // found UNICODE_STRING parameter
        dwStringFound = 1;
    }

    // check if a string value was found
    if(dwStringFound != 0)
    {
        // write string value to log entry
        Log_AppendLogEntry(" <");
        Log_AppendLogEntry(szCurrParamStringValue);
        Log_AppendLogEntry(">");
    }
}

// write the function return value to the log entry
memset(szOutput, 0, sizeof(szOutput));
_snprintf(szOutput, sizeof(szOutput) - 1, ") [RETN_VALUE:0x%08X]", dwReturnValue);
Log_AppendLogEntry(szOutput);
Log_EndLogEntry();

// store original return address
*pdwOrigReturnAddress = dwOrigReturnAddress;
return 0;
```

}

DWORD GetCurrentThreadID_TEB() { DWORD dwCurrentThreadID = 0;

```
// get current thread ID (without calling API functions)
_asm push eax
_asm mov eax, fs:[0x18]
_asm push [eax + 0x24]
```

```
_asm pop dwCurrentThreadID
_asm pop eax
return dwCurrentThreadID;
```

}

DWORD SetExportHookFilters_CheckUserFilters() { DWORD dwPositiveFilter = 0; char *pCurrFilterListPtr = NULL; char* pFilterListNextEntry = NULL; DWORD dwTempFunctionNameLength = 0; char szTempFunctionName[128]; DWORD dwLastFilterListEntry = 0;

```
// check if user filters are set
if(Global_CfgData.szFilter[0] != '\0')
{
    // check if a positive filter was specified (inclusions only)
    if(Global_CfgData.szFilter[0] == '+')
    {
        // disable all existing hooks
        for(DWORD i = 0; i < dwGlobal_NtdllExportListCount; i++)
        {
            // clear hook flag
            Global_NtdllExportList[i].dwLogExport = 0;
        }

        // positive filter
        dwPositiveFilter = 1;
    }
    else if(Global_CfgData.szFilter[0] == '-')
    {
        // negative filter
        dwPositiveFilter = 0;
    }
    else
    {
        // invalid filter type
        return 1;
    }

    // check if this function exists in the ignore list file
    pCurrFilterListPtr = &Global_CfgData.szFilter[1];
    for(;;)
    {
        // find next name in list
        pFilterListNextEntry = strstr(pCurrFilterListPtr, ",");
        if(pFilterListNextEntry == NULL)
        {
            // last entry in list
            dwLastFilterListEntry = 1;

            // calculate length
            dwTempFunctionNameLength = strlen(pCurrFilterListPtr);
        }
        else
        {
            // calculate length
            dwTempFunctionNameLength = (DWORD)(pFilterListNextEntry -
```

```
    pCurrFilterListPtr);
}

if(dwTempFunctionNameLength >= sizeof(szTempFunctionName))
{
    // function name too long
    return 1;
}

// extract function name from ignore list
memset(szTempFunctionName, 0, sizeof(szTempFunctionName));
memcpy(szTempFunctionName, pCurrFilterListPtr, dwTempFunctionNameLength);

if(strlen(szTempFunctionName) != 0)
{
    // find this function in the export list
    for(DWORD i = 0; i < dwGlobal_NtdllExportListCount; i++)
    {
        if(strcmp(Global_NtdllExportList[i].szExportName,
        szTempFunctionName) != 0)
        {
            continue;
        }

        // found function - set the hook flag
        if(dwPositiveFilter != 0)
        {
            // positive filter - set hook flag
            Global_NtdllExportList[i].dwLogExport = 1;
        }
        else
        {
            // negative filter - clear hook flag
            Global_NtdllExportList[i].dwLogExport = 0;
        }

        // also unhook any duplicate exports
        for(DWORD ii = 0; ii < dwGlobal_NtdllExportListCount; ii++)
        {
            if(ii == i)
            {
                continue;
            }

            if(Global_NtdllExportList[ii].dwAddr == Global_NtdllExportList[i].dwAddr)
            {
                // found duplicate - copy hook flag value
                Global_NtdllExportList[ii].dwLogExport =
                Global_NtdllExportList[i].dwLogExport;
            }
        }
    }
}
```

```
        // check if this is the last entry
        if(dwLastFilterListEntry != 0)
        {
            break;
        }

        // update current ptr
        pCurrFilterListPtr = pFilterListNextEntry;
        pCurrFilterListPtr++;
    }
}
return 0;
}
```

DWORD SetExportHookFilters_CheckIgnoreList() { char *pCurrIgnoreListPtr = NULL; char* pIgnoreListNextEntry = NULL; char *pCarriageReturn = NULL; DWORD dwTempFunctionNameLength = 0; char szTempFunctionName[128]; DWORD dwLastIgnoreListEntry = 0;

```
// check if this function exists in the ignore list file
pCurrIgnoreListPtr = pGlobal_IgnoreListFileEntries;
for(;;)
{
    // find next name in list
    pIgnoreListNextEntry = strstr(pCurrIgnoreListPtr, "\n");
    if(pIgnoreListNextEntry == NULL)
    {
        // last entry in list
        dwLastIgnoreListEntry = 1;

        // calculate length
        dwTempFunctionNameLength = strlen(pCurrIgnoreListPtr);
    }
    else
    {
        // calculate length
        dwTempFunctionNameLength = (DWORD)(pIgnoreListNextEntry -
        pCurrIgnoreListPtr);
    }

    if(dwTempFunctionNameLength >= sizeof(szTempFunctionName))
    {
        // function name too long
        return 1;
    }

    // extract function name from ignore list
    memset(szTempFunctionName, 0, sizeof(szTempFunctionName));
    memcpy(szTempFunctionName, pCurrIgnoreListPtr, dwTempFunctionNameLength);

    // remove carriage return character if it exists
    pCarriageReturn = strstr(szTempFunctionName, "\r");
    if(pCarriageReturn != NULL)
    {
        *pCarriageReturn = '\0';
```

```
    }

    if(strlen(szTempFunctionName) != 0)
    {
        // unhook this function
        for(DWORD i = 0; i < dwGlobal_NtdllExportListCount; i++)
        {
            if(strcmp(Global_NtdllExportList[i].szExportName, szTempFunctionName) != 0)
            {
                continue;
            }

            // found function - remove the hook flag
            Global_NtdllExportList[i].dwLogExport = 0;

            // also unhook any duplicate exports
            for(DWORD ii = 0; ii < dwGlobal_NtdllExportListCount; ii++)
            {
                if(ii == i)
                {
                    continue;
                }

                if(Global_NtdllExportList[ii].dwAddr == Global_NtdllExportList[i].dwAddr)
                {
                    // found duplicate - unhook
                    Global_NtdllExportList[ii].dwLogExport = 0;
                }
            }
        }
    }

    // check if this is the last entry
    if(dwLastIgnoreListEntry != 0)
    {
        break;
    }

    // update current ptr
    pCurrIgnoreListPtr = pIgnoreListNextEntry;
    pCurrIgnoreListPtr++;
}
return 0;
}
```

DWORD SetExportHookFilters_CheckMandatoryList() { for(DWORD i = 0; i < dwGlobal_NtdllExportListCount; i++) { // ensure kernel callback functions are hooked regardless of user filters if(strcmp(Global_NtdllExportList[i].szExportNam "KiUserCallbackDispatcher") == 0) { Global_NtdllExportList[i].dwLogExport = 1; } else if(strcmp(Global_NtdllExportList[i].s "NtCallbackReturn") == 0) { Global_NtdllExportList[i].dwLogExport = 1; }

```
    // ensure kernel syscall functions are not logged
    if(strcmp(Global_NtdllExportList[i].szExportName,
    "KiFastSystemCall") == 0)
    {
```

```
        Global_NtdllExportList[i].dwLogExport = 0;
    }
    else if(strcmp(Global_NtdllExportList[i].szExportName,
    "KiFastSystemCallRet") == 0)
    {
        Global_NtdllExportList[i].dwLogExport = 0;
    }
    else if(strcmp(Global_NtdllExportList[i].szExportName,
    "KiIntSystemCall") == 0)
    {
        Global_NtdllExportList[i].dwLogExport = 0;
    }
}
return 0;
```

}

DWORD SetExportHookFilters() { // set initial hook flags for(DWORD i = 0; i < dwGlobal_NtdllExportListCount; i++) { if(Global_NtdllExportList[i].dwRedirectExport != 0) { // set log flag Global_NtdllExportList[i].dwLogExport = 1; } }

```
// check user command-line filters
if(SetExportHookFilters_CheckUserFilters() != 0)
{
    return 1;
}

// check ignore list entries
if(SetExportHookFilters_CheckIgnoreList() != 0)
{
    return 1;
}

// set mandatory values
if(SetExportHookFilters_CheckMandatoryList() != 0)
{
    return 1;
}
return 0;
```

}

DWORD LogEndCall(DWORD dwStackPtr, DWORD dwRetnValue) { DWORD dwCurrentThreadID = 0; DWORD dwOrigReturnAddress = 0;

```
dwCurrentThreadID = GetCurrentThreadID_TEB();



RtlEnterCriticalSection(&Global_LogCriticalSection);



// a previous call has returned

if(CallComplete(dwCurrentThreadID, dwStackPtr, dwRetnValue,
&dwOrigReturnAddress) != 0)
```

```
{
    // failed to process completed call
    Log_NewLogEntry(dwCurrentThreadID, "[ERROR_PROCESSING_END_OF_CALL]");
    Log_EndLogEntry();
    RtlLeaveCriticalSection(&Global_LogCriticalSection);

    // error
    return 0;
}

RtlLeaveCriticalSection(&Global_LogCriticalSection);
return dwOrigReturnAddress;
```

}

void ___declspec(naked) EndCallStub() { _asm { // preserve eax/ebx/ecx/edx values push eax push ebx push ecx push edx

```
    // get original stack ptr (ignore preserved eax/ebx/ecx/edx values)
    mov ecx, esp
    add ecx, 16

    // log the end of this function call - pass the stack pointer and the function return value (eax)
    push eax
    push ecx
    call LogEndCall

    // store LogEndCall return value in unused stack space
    mov dword ptr [esp], eax

    // clean up after LogEndCall
    add esp, 8

    // restore register values
    pop edx
    pop ecx
    pop ebx
    pop eax

    // continue program execution
    jmp dword ptr [esp - 24]
}
```

}

DWORD LogNewCall(DWORD dwStackPtr) { DWORD dwExportAddr = 0; ExportEntryStruct *pExportEntry = NULL; DWORD dwClonedAddress = 0; DWORD dwCurrentThreadID = 0; DWORD dwIgnoreLog = 0;

```
dwCurrentThreadID = GetCurrentThreadID_TEB();

// check if logging should be ignored for this thread
if(dwCurrentThreadID == dwGlobal_InitialThreadID)
{
    // this is the initial thread (LogNT32 setup) - don't log
    dwIgnoreLog = 1;
}
```

```
// get the address of the original function call (subtract 5 from the return
address: 0xE8,0x??,0x??,0x??,0x??)
dwExportAddr = *(DWORD*)dwStackPtr;
dwExportAddr -= 5;

if(dwGlobal_Wow64 == 0 && dwExportAddr ==
dwGlobal_Orig32BitUser32CallbackReturnAddr)
{
    // User32CallbackReturn function

    RtlEnterCriticalSection(&Global_LogCriticalSection);

    // check if we are already logging a function call
    if(Global_LogCriticalSection.RecursionCount > 1)
    {
        // ignore - this is an internal call from within LogNT32
        dwIgnoreLog = 1;
    }

    // get User32CallbackReturn cloned address
    dwClonedAddress = (DWORD)pGlobal_32BitUser32CallbackReturnClone;

    // check if logging should be ignored for this thread
    if(dwIgnoreLog == 0)
    {
        // 32-bit OS - return from user32 callback (simulate a NtCallbackReturn call
        if(NewCall(dwCurrentThreadID, "NtCallbackReturn", dwStackPtr +
        sizeof(DWORD)) != 0)
        {
            // failed to process new call
            Log_NewLogEntry(dwCurrentThreadID, "[ERROR_PROCESSING_USER32_CALLBACK]");
            Log_EndLogEntry();

            RtlLeaveCriticalSection(&Global_LogCriticalSection);

            // error - return the cloned function address without logging this call
            return dwClonedAddress;
        }
    }

    RtlLeaveCriticalSection(&Global_LogCriticalSection);
}
else
{
    pExportEntry = GetExportEntryByAddr(dwExportAddr);
    if(pExportEntry == NULL)
    {
        // error - return null ptr
        return 0;
    }

    // get cloned addr
    if(GetClonedAddr(pExportEntry->dwAddr, &dwClonedAddress) != 0)
```

```
    {
        // error - return null ptr
        return 0;
    }

    // check if this export should be logged
    if(pExportEntry->dwLogExport == 0)
    {
        // don't log this export
        return dwClonedAddress;
    }

    RtlEnterCriticalSection(&Global_LogCriticalSection);

    // check if we are already logging a function call
    if(Global_LogCriticalSection.RecursionCount > 1)
    {
        // ignore - this is an internal call from within LogNT32
        dwIgnoreLog = 1;
    }

    // check if logging should be ignored for this thread
    if(dwIgnoreLog == 0)
    {
        // new function call - log this entry
        if(NewCall(dwCurrentThreadID, pExportEntry->szExportName, dwStackPtr + sizeof(DWORD)) != 0)
        {
            // failed to process new call
            Log_NewLogEntry(dwCurrentThreadID, "[ERROR_PROCESSING_NEW_CALL]");
            Log_EndLogEntry();

            RtlLeaveCriticalSection(&Global_LogCriticalSection);

            // error - return the cloned function address without logging this call
            return dwClonedAddress;
        }

        // update the return address to EndCallStub
        *(DWORD*)(dwStackPtr + sizeof(DWORD)) = dwGlobal_EndCallStubAddr;
    }

    RtlLeaveCriticalSection(&Global_LogCriticalSection);
}

// return the cloned function address to continue execution
return dwClonedAddress;

}
void __declspec(naked) StartCallStub() { _asm {
    // preserve eax/ebx/ecx/edx values
    push eax
    push ebx
    push ecx
```

```
    push edx

    // get original stack ptr (ignore preserved eax/ebx/ecx/edx values)
    mov ecx, esp
    add ecx, 16

    // log new function call - pass the stack pointer
    push ecx
    call LogNewCall

    // store LogNewCall return value in unused stack space
    mov dword ptr [esp], eax

    // clean up after LogNewCall
    add esp, 4

    // restore register values
    pop edx
    pop ecx
    pop ebx
    pop eax

    // ignore temporary return address
    add esp, 4

    // jump to cloned function
    jmp dword ptr [esp - 24]
}
```

}

DWORD InstallExportHooks() { DWORD dwBypassHook = 0; DWORD dwClonedAddress = 0; DWORD dwRelativeAddr = 0;

```
for(DWORD i = 0; i < dwGlobal_NtdllExportListCount; i++)
{
    if(Global_NtdllExportList[i].dwRedirectExport != 0)
    {
        dwBypassHook = 0;
        if(strcmp(Global_NtdllExportList[i].szExportName,
        "KiUserExceptionDispatcher") == 0)
        {
            // don't hook KiUserExceptionDispatcher - jump straight to
            the cloned version
            dwBypassHook = 1;
        }

        // check if this export should bypass the hook
        if(dwBypassHook != 0)
        {

            // get cloned function addr
            if(GetClonedAddr(Global_NtdllExportList[i].dwAddr,
            &dwClonedAddress) != 0)
            {
```

```
                return 1;
            }

            // jmp to cloned function
            *(BYTE*)Global_NtdllExportList[i].dwAddr = 0xE9;
            dwRelativeAddr = dwClonedAddress -
            (Global_NtdllExportList[i].dwAddr + 5);
            *(DWORD*)(Global_NtdllExportList[i].dwAddr + 1) = dwRelativeAddr;
        }
        else
        {
            // call StartCallStub
            *(BYTE*)Global_NtdllExportList[i].dwAddr = 0xE8;
            dwRelativeAddr = dwGlobal_StartCallStubAddr -
            (Global_NtdllExportList[i].dwAddr + 5);
            *(DWORD*)(Global_NtdllExportList[i].dwAddr + 1) = dwRelativeAddr;
        }
    }
}
return 0;
}
```

DWORD CloneNtdllExecutableSections() { MEMORY_BASIC_INFORMATION MemoryInformation; IMAGE_NT_HEADERS *pNtHeader = NULL; DWORD dwFirstExecSectionIndex = 0; DWORD dwLastExecSectionIndex = 0; BYTE* pCurrSectionStart = NULL; DWORD dwTempProtect = 0; IMAGE_SECTION_HEADER *pCurrSectionHeader = NULL;

```
// get nt header ptr
pNtHeader = GetModuleNtHeader((DWORD)pGlobal_NtdllBase);
if(pNtHeader == NULL)
{
    return 1;
}

// loop through all sections
for(DWORD i = 0; i < pNtHeader->FileHeader.NumberOfSections; i++)
{
    // get current section header
    pCurrSectionHeader = (IMAGE_SECTION_HEADER*)((BYTE*)pNtHeader +
    sizeof(IMAGE_NT_HEADERS) + (i * sizeof(IMAGE_SECTION_HEADER)));

    // get current section start ptr
    pCurrSectionStart = (BYTE*)pGlobal_NtdllBase + pCurrSectionHeader->VirtualAddress;

    // get memory information for current section
    memset((void*)&MemoryInformation, 0, sizeof(MemoryInformation));
    if(VirtualQuery(pCurrSectionStart, &MemoryInformation, sizeof(MemoryInformation)) == 0)
    {
        return 1;
    }

    // check if this section is executable
    if(MemoryInformation.Protect & PAGE_EXECUTE || MemoryInformation.Protect &
    PAGE_EXECUTE_READ || MemoryInformation.Protect & PAGE_EXECUTE_READWRITE ||
```

```
    MemoryInformation.Protect & PAGE_EXECUTE_WRITECOPY)
    {
        // executable section
        if(dwGlobal_NtdllExecutableSectionListCount >= MAX_EXECUTABLE_SECTIONS)
        {
            // too many code sections
            return 1;
        }

        // store section in list
        Global_NtdllExecutableSectionList[dwGlobal_NtdllExecutableSectionListCount].dwStartAddr
        = (DWORD)pCurrSectionStart;

        Global_NtdllExecutableSectionList[dwGlobal_NtdllExecutableSectionListCount].dwLength =
        pCurrSectionHeader->SizeOfRawData;

        Global_NtdllExecutableSectionList[dwGlobal_NtdllExecutableSectionListCount].dwOriginalProtect =
        MemoryInformation.Protect;

        // increase count
        dwGlobal_NtdllExecutableSectionListCount++;
    }
}

// ensure at least one code section was found
if(dwGlobal_NtdllExecutableSectionListCount == 0)

{
    return 1;
}

// find first code section
for(i = 0; i < dwGlobal_NtdllExecutableSectionListCount; i++)
{
    if(i == 0)
    {
        // store initial value
        dwFirstExecSectionIndex = i;
        continue;
    }

    // check if the current section is lower than the previous
    if(Global_NtdllExecutableSectionList[i].dwStartAddr < Global_NtdllExecutableSectionList[dwFirstExec
    {
        // update value
        dwFirstExecSectionIndex = i;
    }
}

// find last code section
for(i = 0; i < dwGlobal_NtdllExecutableSectionListCount; i++)
{
    if(i == 0)
    {
```

```
        // store initial value
        dwLastExecSectionIndex = i;
        continue;
    }

    // check if the current section is higher than the previous
    if(Global_NtdllExecutableSectionList[i].dwStartAddr > Global_NtdllExecutableSectionList[dwLastExecS
    {
        // update value
        dwLastExecSectionIndex = i;
    }
}

// calculate total code length
dwGlobal_NtdllCodeLength =
(Global_NtdllExecutableSectionList[dwLastExecSectionIndex].dwStartAddr +
Global_NtdllExecutableSectionList[dwLastExecSectionIndex].dwLength) -
Global_NtdllExecutableSectionList[dwFirstExecSectionIndex].dwStartAddr;

// store code start addr
pGlobal_NtdllCodeStart = (BYTE*)Global_NtdllExecutableSectionList[dwFirstExecSectionIndex].dwStartAddr;

// allocate data
pGlobal_NtdllClone = (BYTE*)VirtualAlloc(NULL, dwGlobal_NtdllCodeLength, MEM_COMMIT, PAGE_EXECUTE_READW
if(pGlobal_NtdllClone == NULL)
{
    return 1;
}

// clone code section data
for(i = 0; i < dwGlobal_NtdllExecutableSectionListCount; i++)
{
    // clone current section
    memcpy((BYTE*)(pGlobal_NtdllClone + (Global_NtdllExecutableSectionList[i].dwStartAddr -
    Global_NtdllExecutableSectionList[dwFirstExecSectionIndex].dwStartAddr)),
    (BYTE*)Global_NtdllExecutableSectionList[i].dwStartAddr,
    Global_NtdllExecutableSectionList[i].dwLength);

    // make section writable (for setting breakpoints)
    if(VirtualProtect((void*)Global_NtdllExecutableSectionList[i].dwStartAddr,
    Global_NtdllExecutableSectionList[i].dwLength, PAGE_EXECUTE_READWRITE,
     &dwTempProtect) == 0)
    {
        return 1;
    }
}
return 0;
```

}

DWORD SetupLogger() { DWORD dwClonedExportPtrListCount = 0;

```
// initialise log critical section
InitializeCriticalSection(&Global_LogCriticalSection);
```

```
// store a list of non-hooked ntdll function ptrs for use within this module
dwClonedExportPtrListCount = sizeof(Global_ClonedExportPtrList) /
sizeof(Global_ClonedExportPtrList[0]);
for(DWORD i = 0; i < dwClonedExportPtrListCount; i++)
{
    // store the cloned (non-hooked) ntdll ptr for the current function
    if(GetClonedAddrByExportName(Global_ClonedExportPtrList[i].pExportName,
    (DWORD*)Global_ClonedExportPtrList[i].pFunctionPtr) != 0)
    {
        return 1;
    }
}

// store the current thread ID
dwGlobal_InitialThreadID = GetCurrentThreadID_TEB();



// open a handle to the initial thread - this prevents the thread ID from being re-used
hGlobal_InitialThread = OpenThreadSyncHandle(dwGlobal_InitialThreadID);
if(hGlobal_InitialThread == NULL)
{
    return 1;
}

// create output file
hGlobal_LogFile = CreateFile(Global_CfgData.szLogFilePath, GENERIC_WRITE,
FILE_SHARE_READ, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

if(hGlobal_LogFile == INVALID_HANDLE_VALUE)
{
    return 1;
}
return 0;
```

}

DWORD RestoreMemoryProtection() { DWORD dwTempProtect = 0;

```
// restore original protection value for all sections
for(DWORD i = 0; i < dwGlobal_NtdllExecutableSectionListCount; i++)
{
    // restore protection
    if(VirtualProtect((void*)Global_NtdllExecutableSectionList[i].dwStartAddr,
     Global_NtdllExecutableSectionList[i].dwLength,
     Global_NtdllExecutableSectionList[i].dwOriginalProtect, &dwTempProtect) == 0)
    {
        return 1;
    }
}
return 0;
```

}

DWORD StartLog(char *pFileName) { char szLogOutput[512];

```
RtlEnterCriticalSection(&Global_LogCriticalSection);
```

```
// add startup logging
Log_WriteRawData("LogNT32 - x86matthew\r\n");
Log_WriteRawData("http://www.x86matthew.com/\r\n");
Log_WriteRawData("\r\n");
memset(szLogOutput, 0, sizeof(szLogOutput));
_snprintf(szLogOutput, sizeof(szLogOutput) - 1, "Logging executable: %s\r\n", pFileName);
Log_WriteRawData(szLogOutput);
Log_WriteRawData("\r\n");

RtlLeaveCriticalSection(&Global_LogCriticalSection);
return 0;
```

}

DWORD PopulateNtdllExportList_Callback(char *pExportName, DWORD dwExportAddr, BYTE* pParam) {
DWORD dwRedirectExport = 0;

```
if(dwGlobal_NtdllExportListCount >= MAX_NTDLL_EXPORT_LIST_COUNT)
{
    // error
    return 1;
}

// check if the export addr is within an executable section
if(dwExportAddr >= (DWORD)pGlobal_NtdllCodeStart && dwExportAddr <=
((DWORD)pGlobal_NtdllCodeStart + dwGlobal_NtdllCodeLength))
{
    // check if the first character of the export is upper-case (eg NtXXX, ignore memcpy etc)
    if(*pExportName >= 'A' && *pExportName <= 'Z')
    {
        dwRedirectExport = 1;
    }
}

// store export details
strncpy(Global_NtdllExportList[dwGlobal_NtdllExportListCount].szExportName,
pExportName, sizeof(Global_NtdllExportList[dwGlobal_NtdllExportListCount].szExportName) - 1);
Global_NtdllExportList[dwGlobal_NtdllExportListCount].dwAddr = dwExportAddr;
Global_NtdllExportList[dwGlobal_NtdllExportListCount].dwRedirectExport =
dwRedirectExport;
Global_NtdllExportList[dwGlobal_NtdllExportListCount].dwLogExport = 0;

// increase export count
dwGlobal_NtdllExportListCount++;
return 0;
```

}

DWORD InstallNtdllHook() { char szFileName[512];

```
// get exe path
memset(szFileName, 0, sizeof(szFileName));
GetModuleFileName(NULL, szFileName, sizeof(szFileName) - 1);

// get ntdll base address
pGlobal_NtdllBase = (BYTE*)GetModuleHandle("ntdll.dll");
```

```
if(pGlobal_NtdllBase == NULL)
{
    return 1;
}

// create clone of ntdll executable sections
if(CloneNtdllExecutableSections() != 0)
{
    return 1;
}

// populate ntdll export list
memset((void*)Global_NtdllExportList, 0, sizeof(Global_NtdllExportList));
if(EnumModuleExportNames((DWORD)pGlobal_NtdllBase, PopulateNtdllExportList_Callback,
NULL) != 0)
{
    return 1;
}

// setup logger
if(SetupLogger() != 0)
{
    return 1;
}

// set export hook list
if(SetExportHookFilters() != 0)
{
    return 1;
}

// set breakpoints
if(InstallExportHooks() != 0)
{
    return 1;
}

// restore original memory protection in ntdll sections
if(RestoreMemoryProtection() != 0)
{
    return 1;
}

// start logging
if(StartLog(szFileName) != 0)
{
    return 1;
}
return 0;
```

}

DWORD GetConfig() { DWORD dwStartTime = 0; HANDLE hConfigFile = NULL; char szFilePath[512]; DWORD dwBytesRead = 0;

```
// set cfg file path
memset(szFilePath, 0, sizeof(szFilePath));
_snprintf(szFilePath, sizeof(szFilePath) - 1, "%s\\LogNT32_%u.cfg",
szGlobal_BaseDirectory, GetCurrentProcessId());

// set initial start time
dwStartTime = GetTickCount();

for(;;)
{
    // check timeout
    if(GetTickCount() - dwStartTime >= (CONFIG_FILE_TIMEOUT_SECONDS * 1000))
    {
        // timeout
        return 1;
    }

    // open configuration file
    hConfigFile = CreateFile(szFilePath, GENERIC_READ, 0, NULL, OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL, NULL);
    if(hConfigFile == INVALID_HANDLE_VALUE)
    {
        // failed to open cfg file - try again in 100ms
        Sleep(100);
        continue;
    }
    break;
}

// read cfg data
if(ReadFile(hConfigFile, (void*)&Global_CfgData, sizeof(Global_CfgData),
&dwBytesRead, NULL) == 0)
{
    // error
    CloseHandle(hConfigFile);
    return 1;
}

// close file handle
CloseHandle(hConfigFile);

// delete configuration file
DeleteFile(szFilePath);
return 0;
```

}

DWORD CreateDefaultIgnoreList(char *pFilePath, HANDLE* phFile) { HANDLE hIgnoreListFile = NULL; DWORD dwCount = 0; DWORD dwBytesWritten = 0;

```
// open configuration file
hIgnoreListFile = CreateFile(pFilePath, GENERIC_READ | GENERIC_WRITE, 0, NULL,
CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
if(hIgnoreListFile == INVALID_HANDLE_VALUE)
{
```

```
    return 1;
}

// write default entries to file
dwCount = sizeof(pGlobal_DefaultIgnoreExportList) / sizeof(pGlobal_DefaultIgnoreExportList[0]);
for(DWORD i = 0; i < dwCount; i++)
{
    // write current entry
    WriteFile(hIgnoreListFile, pGlobal_DefaultIgnoreExportList[i],
    strlen(pGlobal_DefaultIgnoreExportList[i]), &dwBytesWritten, NULL);
    WriteFile(hIgnoreListFile, "\r\n", strlen("\r\n"), &dwBytesWritten, NULL);
}

// return to start of file
SetFilePointer(hIgnoreListFile, 0, NULL, FILE_BEGIN);

// store file handle
*phFile = hIgnoreListFile;
return 0;
```

}

DWORD GetIgnoreList() { char szFilePath[512]; DWORD dwFileSize = 0; HANDLE hIgnoreListFile = NULL; DWORD dwBytesRead = 0; BYTE *pFileContent = NULL;

```
// set cfg file path
memset(szFilePath, 0, sizeof(szFilePath));
_snprintf(szFilePath, sizeof(szFilePath) - 1, "%s\\LogNT32_IgnoreList.txt",
szGlobal_BaseDirectory);

// open configuration file
hIgnoreListFile = CreateFile(szFilePath, GENERIC_READ, 0, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);

if(hIgnoreListFile == INVALID_HANDLE_VALUE)
{
    // ignore list doesn't exist - create default file
    if(CreateDefaultIgnoreList(szFilePath, &hIgnoreListFile) != 0)
    {
        return 1;
    }
}

// get file size
dwFileSize = GetFileSize(hIgnoreListFile, NULL);
if(dwFileSize == INVALID_FILE_SIZE)
{
    // error
    CloseHandle(hIgnoreListFile);
    return 1;
}

// allocate space for file contents
pFileContent = (BYTE*)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT, dwFileSize + 1);
if(pFileContent == NULL)
```

```
{
    // error
    CloseHandle(hIgnoreListFile);
    return 1;
}

// read file data
if(ReadFile(hIgnoreListFile, (void*)pFileContent, dwFileSize, &dwBytesRead, NULL) == 0)
{
    // error
    GlobalFree(pFileContent);
    CloseHandle(hIgnoreListFile);
    return 1;
}

// close file handle
CloseHandle(hIgnoreListFile);

// store ignore list
pGlobal_IgnoreListFileEntries = (char*)pFileContent;
return 0;
```

}

DWORD GetModuleDirectory(HINSTANCE hModule, char *pDirectoryPath, DWORD dwMaxLength) { char szFileName[512]; char* pCurrChar = NULL; char *pLastSlashCharacter = NULL; DWORD dwFileNameLength = 0;

```
// get module file path
memset(szFileName, 0, sizeof(szFileName));
if(GetModuleFileName(hModule, szFileName, sizeof(szFileName) - 1) == 0)
{
    return 1;
}

// calculate file path length
dwFileNameLength = (DWORD)strlen(szFileName);

// find the last slash in the path
pCurrChar = szFileName;
for(DWORD i = 0; i < dwFileNameLength; i++)
{
    // check if this character is a slash
    if(*pCurrChar == '\\')
    {
        // store ptr (don't break - there may be more slashes)
        pLastSlashCharacter = pCurrChar;
    }

    // increase ptr
    pCurrChar++;
}

// ensure the path contains slashes
if(pLastSlashCharacter == NULL)
```

```
{
    return 1;
}

// terminate the string at the last slash
*pLastSlashCharacter = '\0';

// store the directory path
strncpy(pDirectoryPath, szFileName, dwMaxLength);
return 0;
```

}

DWORD CheckWow64() { BOOL (WINAPI *IsWow64Process)(HANDLE hProcess, DWORD* Wow64Process); DWORD dwWow64 = 0;

```
// reset flag
dwGlobal_Wow64 = 0;

// check if this is a 64bit OS
IsWow64Process = (int (__stdcall *)(void *,unsigned long *))
GetProcAddress(GetModuleHandle("kernel32.dll"), "IsWow64Process");

if(IsWow64Process != NULL)
{
    if(IsWow64Process(GetCurrentProcess(), &dwWow64) != 0)
    {
        if(dwWow64 != 0)
        {
            // 64-bit OS
            dwGlobal_Wow64 = 1;
        }
    }
}
return 0;
```

}

BOOL WINAPI DllMain(HINSTANCE hDllHandle, DWORD dwReason, LPVOID Reserved) { if(dwReason == DLL_PROCESS_ATTACH) { // save stub ptrs dwGlobal_StartCallStubAddr = (DWORD)StartCallStub; dwGlobal_EndCallStubAddr = (DWORD)EndCallStub;

```
    // check if this process is running on a 64bit OS
    if(CheckWow64() != 0)
    {
        // error
        return 0;
    }

    if(dwGlobal_Wow64 == 0)
    {
        // 32bit OS - allocate executable data for the User32CallbackReturn function clone
        pGlobal_32BitUser32CallbackReturnClone = (BYTE*)VirtualAlloc(NULL,
        sizeof(Global_32BitUser32CallbackReturnCode), MEM_COMMIT, PAGE_EXECUTE_READWRITE);

        if(pGlobal_32BitUser32CallbackReturnClone == NULL)
        {
```

```
            // error
            return 0;
        }

        // write User32CallbackReturn code
        memcpy(pGlobal_32BitUser32CallbackReturnClone, Global_32BitUser32CallbackReturnCode,
        sizeof(Global_32BitUser32CallbackReturnCode));
    }

    // get base directory
    memset(szGlobal_BaseDirectory, 0, sizeof(szGlobal_BaseDirectory));
    if(GetModuleDirectory(hDllHandle, szGlobal_BaseDirectory, sizeof(szGlobal_BaseDirectory) - 1) != 0)
    {
        // error
        return 0;
    }

    // get configuation settings
    if(GetConfig() != 0)
    {
        // error
        return 0;
    }

    // get export ignore list
    if(GetIgnoreList() != 0)
    {
        // error
        return 0;
    }

    // install ntdll hooks
    if(InstallNtdllHook() != 0)
    {
        // error
        return 0;
    }
}

// success
return 1;
} "'
```