x86matthew - StealthHook - A method for hooking a function without modifying memory protection

x86matthew.com/view_post

StealthHook - A method for hooking a function without modifying memory protection

10/12/2022

Common user-mode function interception methods include inline code hooks, IAT hooks, and hardware breakpoint hooks. These methods are effective, but they require modifications to the .text section, memory protection changes using NtProtectVirtualMemory, or custom exception-handlers, all of which can be "noisy". This post describes a method for stealthily hooking a function without modifying memory protection. By overwriting a global pointer or virtual table entry that is nested within the target function, it is possible to hook the function without raising suspicion because many of these memory regions already have write permissions enabled. This is a method that I have used for many years with good results.

If we are able to intercept a sub-function that is nested within the target function, we can manipulate the execution flow and return value of the original function. By walking the stack from the nested function, we can locate the return address of the original target function. We can then overwrite the original return address with a new value, forcing the program to execute further instructions before continuing with the original code. This method of function hooking does not require any modifications to the executable code, and does not require any memory protection changes, making it difficult to detect using traditional techniques.

The concept behind this method is relatively simple, but manually finding a suitable hooking point can be difficult and time consuming, especially within a large function. To simplify this process, I have developed a tool that automates this process and quickly finds suitable hooking points, if any exist, for any given function. This makes it easy to implement this method in practice, and supports both 32-bit and 64-bit code.

The program builds a list of references by scanning for writable pointers that point to the instructions in the target function. It does this by installing a custom exception handler and single-stepping through the function, one instruction at a time. For each instruction, the program scans each loaded module to see if any writable pointers exist that point to the current instruction. The exception handler is only used by the developer to initially discover potential hooking points, the final hooking code does not require a custom exception handler to be installed.

When running within a WoW64 process, we are unable to single-step into native 64-bit code - in this case, the program catches the transition from 32-bit to 64-bit mode in the Wow64Transition function and sets a hardware breakpoint at the return address. Single-

stepping is temporarily disabled until the target function returns back to 32-bit mode, at which point it will resume and continue monitoring the function.

After creating a list of references, the program will overwrite each pointer in the list one at a time and then run the target function again. If the overwritten pointer is executed during the run, this indicates that it is a successful hooking point, and the program will mark it as such. This process is repeated for all the pointers in the list until all potential hooking points have been tested.

The program also calculates and displays the stack delta, which is the number of bytes that need to be moved down the stack in order to overwrite the original return address, for each hooking point that it identifies.

It is important to consider the possibility that the pointers to sub-functions that are being hooked may be called by other functions in the program. In this case, care must be taken to ensure that the caller is the target function, and not some other function that may be using the same sub-function.

One potential disadvantage of using this method is that the code flow within the target function may vary across different versions of the target module - this can make it more difficult to reliably intercept function calls. To improve the reliability of a hook using this method, I would recommend enumerating the stack frames properly instead of using a fixed stack delta. Modern control-flow enforcement methods may be effective at preventing these types of hooks in future.

For demonstration, we will use the tool to search for hooking points in the CreateFileA function:

```
StealthHook - x86matthew
www.x86matthew.com

Searching for hooking points...

Instruction 0x777B3440 referenced at KERNELBASE.dll!0x7785FA7C (sect: .data, virt_addr: 0x1DFA7C, stack delta: 0x30)
Instruction 0x77783AB0 referenced at KERNELBASE.dll!0x7785F650 (sect: .data, virt_addr: 0x1DF650, stack delta: 0x100)

Found 2 potential hooking points, testing...

Overwriting reference: 0x7785FA7C...
Calling target function...
Hook caught successfully!

Overwriting reference: 0x7785F650...
```

Calling target function...
Hook caught successfully!

Finished - found 2 successful hooking points

As we can see above, the tool discovered two writable global pointers that are suitable for hooking CreateFileA. From looking at the debug symbols, we can see that these two functions are _pfnEightBitStringToUnicodeString and feclient_EfsClientFreeProtectorList respectively. As the feclient_EfsClientFreeProtectorList function is not commonly used (unlike _pfnEightBitStringToUnicodeString), this is the best choice to use in this case.

The following code shows how we can use the information discovered above to hook and manipulate the return value of CreateFileA calls in a 32-bit test program:

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

DWORD dwGlobal_OrigCreateFileReturnAddr = 0;
DWORD dwGlobal_OrigReferenceAddr = 0;

void __declspec(naked) ModifyReturnValue()
{
// the original return address for the CreateFile call redirects to here
_asm
{
// CreateFile complete - overwrite return value
mov eax, 0x12345678

// continue original execution flow (ecx is safe to overwrite at this point)
mov ecx, dwGlobal_OrigCreateFileReturnAddr
jmp ecx
}
}

void __declspec(naked) HookStub()
{
// the hooked global pointer nested within CreateFile redirects to here
_asm
{
// store original CreateFile return address
mov eax, dword ptr [esp + 0x100]
mov dwGlobal_OrigCreateFileReturnAddr, eax

// overwrite the CreateFile return address
```

```
lea eax, ModifyReturnValue
mov dword ptr [esp + 0x100], eax

// continue original execution flow
mov eax, dwGlobal_OrigReferenceAddr
jmp eax
}
}

DWORD InstallHook()
{
BYTE *pModuleBase = NULL;
BYTE *pHookAddr = NULL;

// get base address of kernelbase.dll
pModuleBase = (BYTE*)GetModuleHandle("kernelbase.dll");
if(pModuleBase == NULL)
{
return 1;
}

// get ptr to function reference
pHookAddr = pModuleBase + 0x1DF650;

// store original value
dwGlobal_OrigReferenceAddr = *(DWORD*)pHookAddr;

// overwrite ptr to call HookStub
*(DWORD*)pHookAddr = (DWORD)HookStub;

return 0;
}

int main()
{
HANDLE hFile = NULL;

// create temporary file (without hook)
printf("Creating file #1...\n");
hFile = CreateFile("temp_file_1.txt", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);
printf("hFile: 0x%X\n\n", hFile);

// install hook
printf("Installing hook...\n\n");
```

```
if(InstallHook() != 0)
{
return 1;
}

// create temporary file (with hook)
printf("Creating file #2...\n");
hFile = CreateFile("temp_file_2.txt", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);
printf("hFile: 0x%X\n\n", hFile);

return 0;
}
```

The results of this test program can be seen below:

```
Creating file #1...
hFile: 0xDC

Installing hook...

Creating file #2...
hFile: 0x12345678
```

The output shows that the first CreateFileA call completed as normal. The second call, after the hook was installed, returned our hooked value of 0x12345678 instead.

The full source-code for the tool is below:

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

#define DEBUG_REGISTER_EXEC_DR0 0x1
#define DEBUG_REGISTER_EXEC_DR1 0x4
#define DEBUG_REGISTER_EXEC_DR2 0x10
#define DEBUG_REGISTER_EXEC_DR3 0x40

#define SINGLE_STEP_FLAG 0x100

#define MAXIMUM_STORED_ADDRESS_COUNT 1024

#define OVERWRITE_REFERENCE_ADDRESS_VALUE 1
```

```
#if _WIN64
#define NATIVE_VALUE ULONGLONG
#define CURRENT_EXCEPTION_STACK_PTR ExceptionInfo->ContextRecord->Rsp
#define CURRENT_EXCEPTION_INSTRUCTION_PTR ExceptionInfo->ContextRecord-
>Rip
#else
#define NATIVE_VALUE DWORD
#define CURRENT_EXCEPTION_STACK_PTR ExceptionInfo->ContextRecord->Esp
#define CURRENT_EXCEPTION_INSTRUCTION_PTR ExceptionInfo->ContextRecord-
>Eip
#endif

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

struct PEB_LDR_DATA
{
DWORD Length;
DWORD Initialized;
PVOID SsHandle;
LIST_ENTRY InLoadOrderModuleList;
LIST_ENTRY InMemoryOrderModuleList;
LIST_ENTRY InInitializationOrderModuleList;
};

struct LDR_DATA_TABLE_ENTRY
{
LIST_ENTRY InLoadOrderLinks;
LIST_ENTRY InMemoryOrderLinks;
LIST_ENTRY InInitializationOrderLinks;
PVOID DllBase;
PVOID EntryPoint;
PVOID SizeOfImage;
UNICODE_STRING FullDllName;
UNICODE_STRING BaseDllName;
PVOID Reserved5[3];
PVOID Reserved6;
ULONG TimeDateStamp;
};
```

```
struct PEB
{
BYTE Reserved1[2];
BYTE BeingDebugged;
BYTE Reserved2[1];
PVOID Reserved3[1];
PVOID ImageBaseAddress;
PEB_LDR_DATA *Ldr;
// .....
};

DWORD dwGlobal_TraceStarted = 0;
DWORD dwGlobal_AddressCount = 0;
DWORD dwGlobal_SuccessfulHookCount = 0;
DWORD dwGlobal_CurrHookExecuted = 0;
NATIVE_VALUE dwGlobal_Wow64TransitionStub = 0;
NATIVE_VALUE dwGlobal_InitialStackPtr = 0;
NATIVE_VALUE dwGlobal_OriginalReferenceValue = 0;
NATIVE_VALUE dwGlobal_AddressList[MAXIMUM_STORED_ADDRESS_COUNT];
BYTE *pGlobal_ExeBase = NULL;

DWORD ExecuteTargetFunction();

DWORD ScanModuleForAddress(BYTE *pModuleBase, char *pModuleName,
NATIVE_VALUE dwAddr, NATIVE_VALUE dwStackPtr)
{
IMAGE_DOS_HEADER *pImageDosHeader = NULL;
IMAGE_NT_HEADERS *pImageNtHeader = NULL;
IMAGE_SECTION_HEADER *pCurrSectionHeader = NULL;
DWORD dwReadOffset = 0;
BYTE *pCurrPtr = NULL;
MEMORY_BASIC_INFORMATION MemoryBasicInfo;
DWORD dwStackDelta = 0;

// get dos header
pImageDosHeader = (IMAGE_DOS_HEADER *)pModuleBase;
if(pImageDosHeader->e_magic != 0x5A4D)
{
return 1;
}

// get nt header
pImageNtHeader = (IMAGE_NT_HEADERS *)(pModuleBase + pImageDosHeader-
>e_lfanew);
if(pImageNtHeader->Signature != IMAGE_NT_SIGNATURE)
```

```
{
return 1;
}

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

// ignore executable sections
if(pCurrSectionHeader->Characteristics & IMAGE_SCN_MEM_EXECUTE)
{
continue;
}

// scan current section for the target address
dwReadOffset = pCurrSectionHeader->VirtualAddress;
for(DWORD ii = 0; ii < pCurrSectionHeader->Misc.VirtualSize / sizeof(NATIVE_VALUE);
ii++)
{
// check if the current value contains the target address
pCurrPtr = pModuleBase + dwReadOffset;
if(*(NATIVE_VALUE *)pCurrPtr == dwAddr)
{
// found target address - check memory protection
memset((void *)&MemoryBasicInfo, 0, sizeof(MemoryBasicInfo));
if(VirtualQuery(pCurrPtr, &MemoryBasicInfo, sizeof(MemoryBasicInfo)) != 0)
{
// check if the current region is writable
if(MemoryBasicInfo.Protect == PAGE_EXECUTE_READWRITE ||
MemoryBasicInfo.Protect == PAGE_READWRITE)
{
// ensure the address list is not full
if(dwGlobal_AddressCount >= MAXIMUM_STORED_ADDRESS_COUNT)
{
printf("Error: Address list is full\n");
return 1;
}

// store current address in list
dwGlobal_AddressList[dwGlobal_AddressCount] = (NATIVE_VALUE)pCurrPtr;
dwGlobal_AddressCount++;
```

```
// calculate stack delta
dwStackDelta = (DWORD)(dwGlobal_InitialStackPtr - dwStackPtr);

printf("Instruction 0x%p referenced at %s!0x%p (sect: %s, virt_addr: 0x%X, stack delta:
0x%X)\n", (void*)dwAddr, pModuleName, (void*)pCurrPtr, pCurrSectionHeader->Name,
dwReadOffset, dwStackDelta);
}
}
}

// increase read offset
dwReadOffset += sizeof(NATIVE_VALUE);
}
}

return 0;
}

DWORD ScanAllModulesForAddress(NATIVE_VALUE dwAddr, NATIVE_VALUE
dwStackPtr)
{
DWORD dwPEB = 0;
PEB *pPEB = NULL;
LDR_DATA_TABLE_ENTRY *pCurrEntry = NULL;
LIST_ENTRY *pCurrListEntry = NULL;
DWORD dwEntryOffset = 0;
char szModuleName[512];
DWORD dwStringLength = 0;

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

// get InMemoryOrderLinks offset in structure
dwEntryOffset = (DWORD)((BYTE *)&pCurrEntry->InLoadOrderLinks - (BYTE
*)pCurrEntry);

// get first link
pCurrListEntry = pPEB->Ldr->InLoadOrderModuleList.Flink;

// enumerate all modules
for(;;)
{
```

```
// get ptr to current module entry
pCurrEntry = (LDR_DATA_TABLE_ENTRY *)((BYTE *)pCurrListEntry - dwEntryOffset);

// check if this is the final entry
if(pCurrEntry->DllBase == 0)
{
// end of module list
break;
}

// ignore main exe module
if(pCurrEntry->DllBase != pGlobal_ExeBase)
{
// convert module name to ansi
dwStringLength = pCurrEntry->BaseDllName.Length / sizeof(wchar_t);
if(dwStringLength > sizeof(szModuleName) - 1)
{
dwStringLength = sizeof(szModuleName) - 1;
}
memset(szModuleName, 0, sizeof(szModuleName));
wcstombs(szModuleName, pCurrEntry->BaseDllName.Buffer, dwStringLength);

// scan current module
ScanModuleForAddress((BYTE *)pCurrEntry->DllBase, szModuleName, dwAddr,
dwStackPtr);
}

// get next module entry in list
pCurrListEntry = pCurrListEntry->Flink;
}

return 0;
}

LONG WINAPI ExceptionHandler(EXCEPTION_POINTERS *ExceptionInfo)
{
NATIVE_VALUE dwReturnAddress = 0;

// check exception code
if(ExceptionInfo->ExceptionRecord->ExceptionCode == EXCEPTION_SINGLE_STEP)
{
if(dwGlobal_TraceStarted == 0)
{
// trace not started - ensure the current eip is the target function
if(CURRENT_EXCEPTION_INSTRUCTION_PTR != ExceptionInfo->ContextRecord-
>Dr0)
```

```
{
return EXCEPTION_CONTINUE_SEARCH;
}

// store original stack pointer
dwGlobal_InitialStackPtr = CURRENT_EXCEPTION_STACK_PTR;

// set hardware breakpoint on the original return address
ExceptionInfo->ContextRecord->Dr1 = *(NATIVE_VALUE *)dwGlobal_InitialStackPtr;

// initial trace started
dwGlobal_TraceStarted = 1;
}

// set debug control field
ExceptionInfo->ContextRecord->Dr7 = DEBUG_REGISTER_EXEC_DR1;

// check current instruction pointer
if(CURRENT_EXCEPTION_INSTRUCTION_PTR == dwGlobal_Wow64TransitionStub)
{
// we have hit the wow64 transition stub - don't single step here, set a breakpoint on the
current return address instead
dwReturnAddress = *(NATIVE_VALUE *)CURRENT_EXCEPTION_STACK_PTR;
ExceptionInfo->ContextRecord->Dr0 = dwReturnAddress;
ExceptionInfo->ContextRecord->Dr7 |= DEBUG_REGISTER_EXEC_DR0;
}
else if(CURRENT_EXCEPTION_INSTRUCTION_PTR == ExceptionInfo-
>ContextRecord->Dr1)
{
// we have reached the original return address - remove all breakpoints
ExceptionInfo->ContextRecord->Dr7 = 0;
}
else
{
// scan all modules for the current instruction pointer
ScanAllModulesForAddress(CURRENT_EXCEPTION_INSTRUCTION_PTR,
CURRENT_EXCEPTION_STACK_PTR);

// single step
ExceptionInfo->ContextRecord->EFlags |= SINGLE_STEP_FLAG;
}

// continue execution
return EXCEPTION_CONTINUE_EXECUTION;
}
else if(ExceptionInfo->ExceptionRecord->ExceptionCode ==
EXCEPTION_ACCESS_VIOLATION)
```

```
{
// access violation - check if the eip matches the expected value
if(CURRENT_EXCEPTION_INSTRUCTION_PTR !=
OVERWRITE_REFERENCE_ADDRESS_VALUE)
{
return EXCEPTION_CONTINUE_SEARCH;
}

// caught current hook successfully
dwGlobal_CurrHookExecuted = 1;

// restore correct instruction pointer
CURRENT_EXCEPTION_INSTRUCTION_PTR = dwGlobal_OriginalReferenceValue;

// continue execution
return EXCEPTION_CONTINUE_EXECUTION;
}

return EXCEPTION_CONTINUE_SEARCH;
}

DWORD InitialiseTracer()
{
NATIVE_VALUE dwWow64Transition = 0;
PVOID(WINAPI * RtlAddVectoredExceptionHandler)(DWORD dwFirstHandler, void
*pExceptionHandler) = NULL;
HMODULE hNtdllBase = NULL;

// store exe base
pGlobal_ExeBase = (BYTE *)GetModuleHandleA(NULL);
if(pGlobal_ExeBase == NULL)
{
return 1;
}

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

// get RtlAddVectoredExceptionHandler function ptr
RtlAddVectoredExceptionHandler = (void *(WINAPI*)(unsigned long, void
*))GetProcAddress(hNtdllBase, "RtlAddVectoredExceptionHandler");
if(RtlAddVectoredExceptionHandler == NULL)
```

```
{
return 1;
}

// add exception handler
if(RtlAddVectoredExceptionHandler(1, (void *)ExceptionHandler) == NULL)
{
return 1;
}

// find Wow64Transition export
dwWow64Transition = (NATIVE_VALUE)GetProcAddress(hNtdllBase,
"Wow64Transition");
if(dwWow64Transition != 0)
{
// get Wow64Transition stub address
dwGlobal_Wow64TransitionStub = *(NATIVE_VALUE *)dwWow64Transition;
}

return 0;
}

DWORD BeginTrace(BYTE *pTargetFunction)
{
CONTEXT DebugThreadContext;

// reset values
dwGlobal_TraceStarted = 0;
dwGlobal_SuccessfulHookCount = 0;
dwGlobal_AddressCount = 0;

// set initial debug context - hardware breakpoint on target function
memset((void *)&DebugThreadContext, 0, sizeof(DebugThreadContext));
DebugThreadContext.ContextFlags = CONTEXT_DEBUG_REGISTERS;
DebugThreadContext.Dr0 = (NATIVE_VALUE)pTargetFunction;
DebugThreadContext.Dr7 = DEBUG_REGISTER_EXEC_DR0;
if(SetThreadContext(GetCurrentThread(), &DebugThreadContext) == 0)
{
return 1;
}

// execute the target function
ExecuteTargetFunction();

return 0;
}
```

```
DWORD TestHooks()
{
// attempt to hook the target function at all referenced instructions found earlier
for(DWORD i = 0; i < dwGlobal_AddressCount; i++)
{
printf("\nOverwriting reference: 0x%p...\n", (void*)dwGlobal_AddressList[i]);

// reset flag
dwGlobal_CurrHookExecuted = 0;

// store original value
dwGlobal_OriginalReferenceValue = *(NATIVE_VALUE *)dwGlobal_AddressList[i];

// overwrite referenced value with placeholder value
*(NATIVE_VALUE *)dwGlobal_AddressList[i] =
OVERWRITE_REFERENCE_ADDRESS_VALUE;

printf("Calling target function...\n");

// execute target function
ExecuteTargetFunction();

// restore original value
*(NATIVE_VALUE *)dwGlobal_AddressList[i] = dwGlobal_OriginalReferenceValue;

// check if the hook was executed
if(dwGlobal_CurrHookExecuted == 0)
{
// hook wasn't executed - ignore
printf("Failed to catch hook\n");
}
else
{
// hook was executed - this address can be used to hook the target function
printf("Hook caught successfully!\n");
dwGlobal_SuccessfulHookCount++;
}
}

return 0;
}

DWORD ExecuteTargetFunction()
{
// call the target function
CreateFileA("temp_file.txt", GENERIC_WRITE, FILE_SHARE_READ |
FILE_SHARE_WRITE, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
```

```
return 0;
}

// www.x86matthew.com
int main()
{
printf("StealthHook - x86matthew\n");
printf("www.x86matthew.com\n\n");

// initialise tracer
if(InitialiseTracer() != 0)
{
return 1;
}

printf("Searching for hooking points...\n\n");

// begin trace
if(BeginTrace((BYTE *)CreateFileA) != 0)
{
return 1;
}

// check if any referenced addresses were found
if(dwGlobal_AddressCount == 0)
{
// none found
printf("No potential hooking points found\n");
}
else
{
printf("\nFound %u potential hooking points, testing...\n", dwGlobal_AddressCount);

// test all of the potential hooks
if(TestHooks() != 0)
{
return 1;
}
}

// finished
printf("\nFinished - found %u successful hooking points\n\n",
dwGlobal_SuccessfulHookCount);

return 0;
}
```