eac 反调试_自己动手制作一个过保护调试器

原创

weixin_39847034 于 2020-12-19 13:45:38 发布 653 收藏 1
文章标签： eac 反调试

原标题：自己动手制作一个过保护调试器

一、起因

本人是新手第一次接触驱动开发的小白，事情是这样的，一个星期前突发奇想想做一个调试器保护程序用于调试游戏，既然要调试驱动保护的程序，自然也要深入驱动底层。做调试器必须要hook api去隐藏调试器的参数，所以就有了今天这篇文章。

二、准备的东西

第一个版本的调试器保护程序就是今天要说的用到 ETW(Event Tracing for Windows) hook 去Hook api 实现隐藏参数的

ETW Hook(下文叫做 InfinityHook

原理简单点概括为 ETW在记录系统函数被调用事件的时候被调用的系统函数的指针还存放在栈上,能对应的指针从而实现Hook. 更多关于InfinityHook的可以百度一下 “InfinityHook” 看雪的大佬写的比我好,理解比我深入,就不卖丑了.

infinityhook的优点是简单好用,兼容性高,不像VT只能在intel cpu用而AMD必须另写SVM很麻烦,坏处是只能做r3层的api hook,R0的API是无法拦截的.github地址:https://github.com/everdox/InfinityHook

基本的思路:

1.设置CreateProcessNotify去标记调试器进程的PID

2. 利用ObRegisterCallbacks注册process和thread的回调,当调试器打开进程时被反作弊的回调降权后再升级回去

3.hook NtQuerySystemInformation、NtSetInformationThread、NtQueryObject、NtQueryInformationThread、NtGetContextThread、NtQueryInformationProcess、pfn_NtGetContextThread XXOO anti debug三、驱动编写

首先是抄了两个传家的ssdt函数获取的东西,通用的框架.不需要多言:

```
extern "C" NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING
RegistryPath) { DebugPrint("\n[DebugMessage] Driver Loading!\n"); RtlInitUnicodeString(&DeviceName,
L"\\Device\\HuojiDebuger"); RtlInitUnicodeString(&Win32Device, L"\\DosDevices\\HuojiDebuger");
PDEVICE_OBJECT DeviceObject = NULL; NTSTATUS status = IoCreateDevice(DriverObject, 0,
&DeviceName, FILE_DEVICE_UNKNOWN, FILE_DEVICE_SECURE_OPEN, FALSE, &DeviceObject); if
(!NT_SUCCESS(status)) { DebugPrint("[DeugMessage] IoCreateDevice Error...\r\n"); return status; } if
(!DeviceObject) { DebugPrint("[DeugMessage] Unexpected I/O Error...\r\n"); return
STATUS_UNEXPECTED_IO_ERROR; } DebugPrint("[DeugMessage] Device %.*ws created
successfully!\r\n", DeviceName.Length / sizeof(WCHAR), DeviceName.Buffer); DriverObject->DriverUnload =
DriverUnload; if (!NT_SUCCESS(NTDLL::Initialize)) { DebugPrint("[DebugMessage] Ntdll::Initialize
failed...\r\n"); return STATUS_UNSUCCESSFUL; } //获取SSDT函数 pfn_NtQueryInformationProcess =
(NTQUERYINFORMATIONPROCESS)SSDT::GetFunctionAddress("NtQueryInformationProcess");
DebugPrint("[DebugMessage] NtQueryInformationProcess: 0x%08X...\r\n", pfn_NtQueryInformationProcess);
pfn_NtGetContextThread = (NTGETCONTEXTHREAD)SSDT::GetFunctionAddress("NtGetContextThread");
DebugPrint("[DebugMessage] NtGetContextThread: 0x%08X...\r\n", pfn_NtGetContextThread);
pfn_NtQueryInformationThread =
(NTQUERYINFORMATIONTHREAD)SSDT::GetFunctionAddress("NtQueryInformationThread"); DebugPrint("
[DebugMessage] NtQueryInformationThread: 0x%08X...\r\n", pfn_NtQueryInformationThread);
pfn_NtQueryObject = (NTQUERYOBJECT)SSDT::GetFunctionAddress("NtQueryObject"); DebugPrint("
[DebugMessage] NtQueryObject: 0x%08X...\r\n", pfn_NtQueryObject); pfn_NtSetInformationThread =
(NTSETINFORMATIONTHREAD)SSDT::GetFunctionAddress("NtSetInformationThread"); DebugPrint("
[DebugMessage] NtSetInformationThread: 0x%08X...\r\n", pfn_NtSetInformationThread);
pfn_NtQuerySystemInformation =
(NTQUERYSYSTEMINFORMATION)SSDT::GetFunctionAddress("NtQuerySystemInformation"); DebugPrint("
[DebugMessage] NtQuerySystemInformation: 0x%08X...\r\n", pfn_NtQuerySystemInformation);
pfn_NtOpenProcess = (NTOPENPROCESS)SSDT::GetFunctionAddress("NtOpenProcess"); DebugPrint("
[DebugMessage] NtOpenProcess: 0x%08X...\r\n", pfn_NtOpenProcess); pfn_NtReadVirtualMemory =
(NTREADVIRTUALMEMORY)SSDT::GetFunctionAddress("NtReadVirtualMemory"); DebugPrint("
[DebugMessage] NtReadVirtualMemory: 0x%08X...\r\n", pfn_NtReadVirtualMemory);
pfn_NtWriteVirtualMemory =
(NTWRITEVIRTUALMEMORY)SSDT::GetFunctionAddress("NtWriteVirtualMemory"); DebugPrint("
[DebugMessage] NtWriteVirtualMemory: 0x%08X...\r\n", pfn_NtWriteVirtualMemory); pfn_NtCreateThread =
(NTCREATETHREAD)SSDT::GetFunctionAddress("NtCreateThread"); DebugPrint("[DebugMessage]
NtCreateThread: 0x%08X...\r\n", pfn_NtCreateThread); pfn_NtTerminateProcess =
(NTTERMINATEPROCESS)SSDT::GetFunctionAddress("NtTerminateProcess"); DebugPrint("
[DebugMessage] NtTerminateProcess: 0x%08X...\r\n", pfn_NtTerminateProcess); // 绕过
MmVerifyCallbackFunction。 PLDR_DATA_TABLE_ENTRY64 ldr =
(PLDR_DATA_TABLE_ENTRY64)DriverObject->DriverSection; ldr->Flags |= 0x20; //安装回调
InstallCallBacks; //初始化ETW HOOK if (!NT_SUCCESS(IfhInitialize(SyscallCallBack))) DebugPrint("
[DebugMessage] IfhInitialize failed...\r\n"); return STATUS_SUCCESS; }
```

其中

IfhInitialize(SyscallCallBack))

注册后在SyscallCallBack捕获我们的要hook api函数:

```
void __fastcall SyscallCallBack(_In_ unsigned int SystemCallIndex, _Inout_ void** SystemCallFunction) { if
(*SystemCallFunction == pfn_NtOpenProcess) *SystemCallFunction = HookNtOpenProcess; else if
(*SystemCallFunction == pfn_NtReadVirtualMemory) *SystemCallFunction = HookNtReadVirtualMemory; else
if (*SystemCallFunction == pfn_NtWriteVirtualMemory) *SystemCallFunction = HookNtWriteVirtualMemory;
else if (*SystemCallFunction == pfn_NtCreateThread) *SystemCallFunction = HookNtCreateThread; if
(*SystemCallFunction == pfn_NtQueryInformationProcess) *SystemCallFunction =
HookNtQueryInformationProcess; else if (*SystemCallFunction == pfn_NtGetContextThread)
*SystemCallFunction = HookNtGetContextThread; else if (*SystemCallFunction ==
pfn_NtQueryInformationThread) *SystemCallFunction = HookNtQueryInformationThread; else if
(*SystemCallFunction == pfn_NtQueryObject) *SystemCallFunction = HookNtQueryObject; else if
(*SystemCallFunction == pfn_NtSetInformationThread) *SystemCallFunction = HookNtSetInformationThread;
else if (*SystemCallFunction == pfn_NtQuerySystemInformation) *SystemCallFunction =
HookNtQuerySystemInformation; else if (*SystemCallFunction == pfn_NtTerminateProcess)
*SystemCallFunction = HookNtTerminateProcess; }
```

说一下主要的:

HookNtOpenProcess用于确定调试器打开的进程:

```
NTSTATUS NTAPI HookNtOpenProcess(PHANDLE ProcessHandle,ACCESS_MASK
DesiredAccess,POBJECT_ATTRIBUTES ObjectAttributes,PCLIENT_ID ClientId) { NTSTATUS ret =
STATUS_ACCESS_VIOLATION; if (g_startDebug) { ULONG pid = (ULONG)
(ULONG_PTR)PsGetCurrentProcessId; if (g_DbgPid == pid && DesiredAccess == 0x001F0FFF) { g_GamePid
= (ULONG)ClientId->UniqueProcess; DebugPrint("[DebugMessage] Lock Game Id %d ACCESS_MASK :
0x%08X \r\n", g_GamePid, DesiredAccess); } } return pfn_NtOpenProcess(ProcessHandle, DesiredAccess,
ObjectAttributes, ClientId); }
```

HookNtQueryInformationProcess、HookNtGetContextThread、HookNtQueryInformationThread、HookNtSetInformationThread分别用于XXOO调试器检测和硬断检测,检测原理和处理手段大概是这样的:

硬断会设置dr寄存器的值,比如dr0 = xxxx (我们要断下的函数地址,检测也很容易, if(dr0 != 0) banned 为了避免检测我们就需要hook HookNtQueryInformationThread、HookNtGetContextThread、HookNtSetInformationThread .

值得注意的是ThreadHideFromDebugger这个东西.微软对“ ThreadHideFromDebugger ”描述是：设置此参数将使这条线程对调试器“隐藏”。即调试器收不到调试信息。因此我们要处理掉这个flag

```
NTSTATUS NTAPI HookNtQueryInformationProcess( IN HANDLE ProcessHandle, IN PROCESSINFOCLASS
ProcessInformationClass, OUT PVOID ProcessInformation, IN ULONG ProcessInformationLength, OUT
PULONG ReturnLength) { if (g_startDebug) { ULONG dbg_pid =
GetProcessIDFromProcessHandle(ProcessHandle); ULONG pid = (ULONG)
(ULONG_PTR)PsGetCurrentProcessId; if (g_DbgPid == dbg_pid && g_GamePid != -1) { return
STATUS_ACCESS_VIOLATION; } if (g_GamePid == pid) { NTSTATUS ret =
pfn_NtQueryInformationProcess(ProcessHandle, ProcessInformationClass, ProcessInformation,
ProcessInformationLength, ReturnLength); if (ProcessInformationClass == ProcessDebugFlags) { DebugPrint("
[DebugMessage] ProcessDebugFlags by %d\r\n", pid); *(unsigned int*)ProcessInformation = 1; //若为0，则进
程处于被调试状态，若为1，则进程处于非调试状态。 } else if (ProcessInformationClass ==
ProcessDebugPort) { DebugPrint("[DebugMessage] ProcessDebugPort by %d\r\n", pid); *
(ULONG_PTR*)ProcessInformation = 0; } else if (ProcessInformationClass == ProcessDebugObjectHandle) {
DebugPrint("[DebugMessage] ProcessDebugObjectHandle by %d\r\n", pid); HANDLE CantTouchThis =
nullptr; __try { __try { CantTouchThis = *static_cast(ProcessInformation); } __except
(EXCEPTION_EXECUTE_HANDLER) { NOTHING; } *static_cast(ProcessInformation) = nullptr; ret =
STATUS_PORT_NOT_SET; } __except (EXCEPTION_EXECUTE_HANDLER) { ret = GetExceptionCode; } if
(CantTouchThis != nullptr) { BOOLEAN AuditOnClose; const NTSTATUS HandleStatus =
ObQueryObjectAuditingByHandle(CantTouchThis, &AuditOnClose); if (HandleStatus !=
STATUS_INVALID_HANDLE) ObCloseHandle(CantTouchThis, ExGetPreviousMode); } } return ret; } } return
pfn_NtQueryInformationProcess(ProcessHandle, ProcessInformationClass, ProcessInformation,
ProcessInformationLength, ReturnLength); } NTSTATUS NTAPI HookNtGetContextThread( IN HANDLE
ThreadHandle, IN OUT PCONTEXT Context) { NTSTATUS ret = pfn_NtGetContextThread(ThreadHandle,
Context); if (NT_SUCCESS(ret) && g_startDebug) { ULONG pid =
GetProcessIDFromThreadHandle(ThreadHandle); ULONG my_pid = (ULONG)
(ULONG_PTR)PsGetCurrentProcessId; if (g_GamePid == pid && g_DbgPid != my_pid) { Context->Dr0 =
NULL; Context->Dr1 = NULL; Context->Dr2 = NULL; Context->Dr3 = NULL; Context->Dr6 = NULL; Context-
>Dr7 = NULL; Context->LastBranchToRip = NULL; Context->LastBranchFromRip = NULL; Context-
>LastExceptionToRip = NULL; Context->LastExceptionFromRip = NULL; Context->EFlags = Context->EFlags
& ~0x10; } } return ret; } NTSTATUS NTAPI HookNtQueryInformationThread(HANDLE ThreadHandle,
THREADINFOCLASS ThreadInformationClass, PVOID ThreadInformation, ULONG ThreadInformationLength,
PULONG ReturnLength) { NTSTATUS ret = pfn_NtQueryInformationThread(ThreadHandle,
ThreadInformationClass, ThreadInformation, ThreadInformationLength, ReturnLength); if (NT_SUCCESS(ret)
&& g_startDebug && ThreadInformationClass == ThreadWow64Context) { ULONG pid =
GetProcessIDFromThreadHandle(ThreadHandle); ULONG my_pid = (ULONG)
(ULONG_PTR)PsGetCurrentProcessId; if (g_GamePid == pid && g_DbgPid != my_pid) {
PWOW64_CONTEXT contex = (PWOW64_CONTEXT)ThreadInformation; contex->Dr0 = NULL; contex->Dr1
= NULL; contex->Dr2 = NULL; contex->Dr3 = NULL; contex->Dr6 = NULL; contex->Dr7 = NULL; contex-
>EFlags = contex->EFlags & ~0x10; DebugPrint("[DebugMessage] Block ZwQueryInformationThread Meme!
\n"); } } return ret; } NTSTATUS NTAPI HookNtSetInformationThread(IN HANDLE ThreadHandle,IN
THREADINFOCLASS ThreadInformationClass,IN PVOID ThreadInformation,IN ULONG
ThreadInformationLength) { if (g_startDebug) { ULONG pid = (ULONG)(ULONG_PTR)PsGetCurrentProcessId;
ULONG my_pid = GetProcessIDFromThreadHandle(ThreadHandle); if (pid == g_GamePid || my_pid ==
g_GamePid) { if (pid == g_DbgPid) { return pfn_NtSetInformationThread(ThreadHandle,
ThreadInformationClass, ThreadInformation, ThreadInformationLength); } else { if (ThreadInformationClass ==
ThreadHideFromDebugger) { DebugPrint("[DebugMessage] ThreadHideFromDebugger by %d\r\n", pid);
PETHREAD Thread; NTSTATUS status = ObReferenceObjectByHandle(ThreadHandle,
THREAD_SET_INFORMATION, *PsThreadType, ExGetPreviousMode, (PVOID*)&Thread, NULL); if
(NT_SUCCESS(status)) ObDereferenceObject(Thread); return status; } else if (ThreadInformationClass ==
ThreadWow64Context) { PWOW64_CONTEXT Wow64Context = (PWOW64_CONTEXT)ThreadInformation;
ULONG OriginalContextFlags = 0; DebugPrint("[DebugMessage] HookNtSetInformationThread by %d\r\n",
pid); ProbeForWrite(&Wow64Context->ContextFlags, sizeof(ULONG), 1); OriginalContextFlags =
Wow64Context->ContextFlags; Wow64Context->ContextFlags = OriginalContextFlags & ~0x10;
//CONTEXT_DEBUG_REGISTERS ^ CONTEXT_AMD64/CONTEXT_i386 NTSTATUS Status =
pfn_NtSetInformationThread(ThreadHandle, ThreadInformationClass, ThreadInformation,
ThreadInformationLength); ProbeForWrite(&Wow64Context->ContextFlags, sizeof(ULONG), 1);
Wow64Context->ContextFlags = OriginalContextFlags; return Status; } } } } return
pfn_NtSetInformationThread(ThreadHandle, ThreadInformationClass, ThreadInformation,
ThreadInformationLength); }
```

HookNtQuerySystemInformation用于隐藏进程而隐藏系统调试器(虽然没什么用,后续会说为什么

```
NTSTATUS NTAPI HookNtQuerySystemInformation(IN SYSTEM_INFORMATION_CLASS
SystemInformationClass,OUT PVOID SystemInformation,IN ULONG SystemInformationLength,OUT PULONG
ReturnLength OPTIONAL) { NTSTATUS ret = pfn_NtQuerySystemInformation(SystemInformationClass,
SystemInformation, SystemInformationLength, ReturnLength); if (NT_SUCCESS(ret) && g_startDebug) { if
(SystemInformationClass == SystemKernelDebuggerInformation) { typedef struct
_SYSTEM_KERNEL_DEBUGGER_INFORMATION { BOOLEAN DebuggerEnabled; BOOLEAN
DebuggerNotPresent; } SYSTEM_KERNEL_DEBUGGER_INFORMATION, *
PSYSTEM_KERNEL_DEBUGGER_INFORMATION; SYSTEM_KERNEL_DEBUGGER_INFORMATION*
DebuggerInfo = (SYSTEM_KERNEL_DEBUGGER_INFORMATION*)SystemInformation; DebuggerInfo-
>DebuggerEnabled = false; DebuggerInfo->DebuggerNotPresent = true; } /* //EAC BE会通过遍历 threads去找
隐藏线程 反而增加flag
//https://github.com/huoji120/EACReversing/blob/master/EasyAntiCheat.sys/hiddenprocess.c else if
(SystemInformationClass == SystemProcessInformation) { PSYSTEM_PROCESS_INFORMATION
pPrevProcessInfo = NULL; PSYSTEM_PROCESS_INFORMATION pCurrProcessInfo =
(PSYSTEM_PROCESS_INFORMATION)SystemInformation; while (pCurrProcessInfo != NULL) { ULONG
uPID = (ULONG)pCurrProcessInfo->ProcessId; UNICODE_STRING strTmpProcessName = pCurrProcessInfo-
>ImageName; //获取当前遍历的 SYSTEM_PROCESS_INFORMATION 节点的进程名称和进程 ID if (uPID ==
g_DbgPid) { if (pPrevProcessInfo) { if (pCurrProcessInfo->NextEntryOffset) { //将当前这个进程(即要隐藏的进
程)从 SystemInformation 中摘除(更改链表偏移指针实现) pPrevProcessInfo->NextEntryOffset +=
pCurrProcessInfo->NextEntryOffset; } else { //说明当前要隐藏的这个进程是进程链表中的最后一个
pPrevProcessInfo->NextEntryOffset = 0; } } else { //第一个遍历到得进程就是需要隐藏的进程 if
(pCurrProcessInfo->NextEntryOffset) { *(PCHAR)SystemInformation += pCurrProcessInfo->NextEntryOffset; }
else { SystemInformation = NULL; } } } pPrevProcessInfo = pCurrProcessInfo; if (pCurrProcessInfo-
>NextEntryOffset) pCurrProcessInfo = (PSYSTEM_PROCESS_INFORMATION)(((PCHAR)pCurrProcessInfo)
+ pCurrProcessInfo->NextEntryOffset); else pCurrProcessInfo = NULL; } }*/ } return ret; }
```

之后设置的安装的ProcessHandleCallbacks、ThreadHandleCallbacks用于句柄提权和保护调试器不会被扫特征(当然这是没什么用的,至于为什么没用以后再说:

```
OB_PREOP_CALLBACK_STATUS ThreadHandleCallbacks(PVOID RegistrationContext,
POB_PRE_OPERATION_INFORMATION OperationInformation) { if (g_DbgPid == -1 || g_GamePid == -1)
return OB_PREOP_SUCCESS; if (OperationInformation->KernelHandle) return OB_PREOP_SUCCESS;
PEPROCESS OpenedProcess = (PEPROCESS)OperationInformation->Object, CurrentProcess =
PsGetCurrentProcess; ULONG ulProcessId = (ULONG)PsGetProcessId(OpenedProcess); ULONG
myProcessId = (ULONG)PsGetProcessId(CurrentProcess); if (myProcessId == g_DbgPid) { if
(OperationInformation->Operation == OB_OPERATION_HANDLE_CREATE || OperationInformation-
>Operation == OB_OPERATION_HANDLE_DUPLICATE) { OperationInformation->Parameters-
>CreateHandleInformation.DesiredAccess = 0x1fffff; OperationInformation->Parameters-
>CreateHandleInformation.OriginalDesiredAccess = 0x1fffff; } } else if (ulProcessId == g_DbgPid) { if
(OperationInformation->Operation == OB_OPERATION_HANDLE_CREATE) OperationInformation-
>Parameters->CreateHandleInformation.DesiredAccess = (SYNCHRONIZE |
THREAD_QUERY_LIMITED_INFORMATION); else OperationInformation->Parameters-
>DuplicateHandleInformation.DesiredAccess = (SYNCHRONIZE |
THREAD_QUERY_LIMITED_INFORMATION); } return OB_PREOP_SUCCESS; }
OB_PREOP_CALLBACK_STATUS ProcessHandleCallbacks(PVOID RegistrationContext,
POB_PRE_OPERATION_INFORMATION OperationInformation) {
UNREFERENCED_PARAMETER(RegistrationContext); if (g_DbgPid == -1 || g_GamePid == -1) return
OB_PREOP_SUCCESS; if (OperationInformation->KernelHandle) return OB_PREOP_SUCCESS;
PEPROCESS ProtectedProcessPEPROCESS; PEPROCESS ProtectedUserModeACPEPROCESS;
PEPROCESS OpenedProcess = (PEPROCESS)OperationInformation->Object, CurrentProcess =
PsGetCurrentProcess; ULONG ulProcessId = (ULONG)PsGetProcessId(OpenedProcess); ULONG
```

myProcessId = (ULONG)PsGetProcessId(CurrentProcess); if (myProcessId == g_DbgPid && ulProcessId == g_GamePid) { if (OperationInformation->Operation == OB_OPERATION_HANDLE_CREATE || OperationInformation->Operation == OB_OPERATION_HANDLE_DUPLICATE) { OperationInformation->Parameters->CreateHandleInformation.DesiredAccess = 0x1fffff; OperationInformation->Parameters->CreateHandleInformation.OriginalDesiredAccess = 0x1fffff; } } else { if (ulProcessId == g_DbgPid) //如果进程是调试器进程 { if (OperationInformation->Operation == OB_OPERATION_HANDLE_CREATE) // striping handle { if ((OperationInformation->Parameters->CreateHandleInformation.OriginalDesiredAccess & PROCESS_TERMINATE) == PROCESS_TERMINATE) { //Terminate the process, such as by calling the user-mode TerminateProcess routine.. OperationInformation->Parameters->CreateHandleInformation.DesiredAccess &= ~PROCESS_TERMINATE; } if ((OperationInformation->Parameters->CreateHandleInformation.OriginalDesiredAccess & PROCESS_VM_OPERATION) == PROCESS_VM_OPERATION) { //Modify the address space of the process, such as by calling the user-mode WriteProcessMemory and VirtualProtectEx routines. OperationInformation->Parameters->CreateHandleInformation.DesiredAccess &= ~PROCESS_VM_OPERATION; } if ((OperationInformation->Parameters->CreateHandleInformation.OriginalDesiredAccess & PROCESS_VM_READ) == PROCESS_VM_READ) { //Read to the address space of the process, such as by calling the user-mode ReadProcessMemory routine. OperationInformation->Parameters->CreateHandleInformation.DesiredAccess &= ~PROCESS_VM_READ; } if ((OperationInformation->Parameters->CreateHandleInformation.OriginalDesiredAccess & PROCESS_VM_WRITE) == PROCESS_VM_WRITE) { //Write to the address space of the process, such as by calling the user-mode WriteProcessMemory routine. OperationInformation->Parameters->CreateHandleInformation.DesiredAccess &= ~PROCESS_VM_WRITE; } } if (OperationInformation->Operation == OB_OPERATION_HANDLE_DUPLICATE) { if ((OperationInformation->Parameters->CreateHandleInformation.OriginalDesiredAccess & PROCESS_TERMINATE) == PROCESS_TERMINATE) { //Terminate the process, such as by calling the user-mode TerminateProcess routine.. OperationInformation->Parameters->CreateHandleInformation.DesiredAccess &= ~PROCESS_TERMINATE; } if ((OperationInformation->Parameters->CreateHandleInformation.OriginalDesiredAccess & PROCESS_VM_OPERATION) == PROCESS_VM_OPERATION) { //Modify the address space of the process, such as by calling the user-mode WriteProcessMemory and VirtualProtectEx routines. OperationInformation->Parameters->CreateHandleInformation.DesiredAccess &= ~PROCESS_VM_OPERATION; } if ((OperationInformation->Parameters->CreateHandleInformation.OriginalDesiredAccess & PROCESS_VM_READ) == PROCESS_VM_READ) { //Read to the address space of the process, such as by calling the user-mode ReadProcessMemory routine. OperationInformation->Parameters->CreateHandleInformation.DesiredAccess &= ~PROCESS_VM_READ; } if ((OperationInformation->Parameters->CreateHandleInformation.OriginalDesiredAccess & PROCESS_VM_WRITE) == PROCESS_VM_WRITE) { //Write to the address space of the process, such as by calling the user-mode WriteProcessMemory routine. OperationInformation->Parameters->CreateHandleInformation.DesiredAccess &= ~PROCESS_VM_WRITE; } } } } return OB_PREOP_SUCCESS; } 四、被EAC殴打

信心满满的打开调试器后附加到EAC保护的进程,结果游戏居然关闭了!关闭了!

这就很难受了,为什么呢?

查看eac的源码,发现EAC R3 会通过NtQuerySystemInformation去得到进程句柄数量,如果进程句柄数量 > 1 则banned 所以被殴打了,连附加都还没做!

五、尝试做meme

于是开始了meme做法,hook NtReadVirtualMemory、NtWriteVirtualMemory我自己的做法是当发现是自己调试器进程读取游戏进程内存时候,自己实现一个读写内存的,不用自带的,并且OpenTProcess那里不给返回句柄:

NTSTATUS NTAPI HookNtReadVirtualMemory(IN HANDLE ProcessHandle,IN PVOID BaseAddress,OUT PVOID Buffer,IN ULONG NumberOfBytesToRead, PULONG NumberOfBytesReaded) { if (g_startDebug) { ULONG pid = (ULONG)(ULONG_PTR)PsGetCurrentProcessId; if (g_DbgPid == pid) { if (GetProcessIDFromProcessHandle(ProcessHandle) == g_GamePid) { return MyReadProcessMemory(g_GameHandle, BaseAddress, Buffer, NumberOfBytesToRead, NumberOfBytesReaded); } } } return pfn_NtReadVirtualMemory(ProcessHandle, BaseAddress, Buffer, NumberOfBytesToRead, NumberOfBytesReaded); } NTSTATUS NTAPI HookNtWriteVirtualMemory(HANDLE ProcessHandle, PVOID BaseAddress, PVOID Buffer, ULONG NumberOfBytesToWrite, PULONG NumberOfBytesWritten) { if (g_startDebug) { ULONG pid = (ULONG)(ULONG_PTR)PsGetCurrentProcessId; if (g_DbgPid == pid) { if (GetProcessIDFromProcessHandle(ProcessHandle) == g_GamePid) { return MyWriteProcessMemory(g_GameHandle, BaseAddress, Buffer, NumberOfBytesToWrite, NumberOfBytesWritten); } } } return pfn_NtWriteVirtualMemory(ProcessHandle, BaseAddress, Buffer, NumberOfBytesToWrite, NumberOfBytesWritten); } NTSTATUS NTAPI HookNtCreateThread(PHANDLE ThreadHandle, ACCESS_MASK DesiredAccess, POBJECT_ATTRIBUTES ObjectAttributes, HANDLE ProcessHandle, PCLIENT_ID ClientId, PCONTEXT ThreadContext, PVOID InitialTeb, BOOLEAN CreateSuspended) { if (g_startDebug) { ULONG pid = (ULONG)(ULONG_PTR)PsGetCurrentProcessId; if (g_DbgPid == pid) { if (GetProcessIDFromProcessHandle(ProcessHandle) == g_GamePid) { DebugPrint("[DebugMessage] NtCreateThread by %d\r\n", pid); return MyNtCreatThread(ThreadHandle, DesiredAccess, ObjectAttributes, g_GameHandle, ClientId, ThreadContext, InitialTeb, CreateSuspended); } } } return pfn_NtCreateThread(ThreadHandle, DesiredAccess, ObjectAttributes, ProcessHandle, ClientId, ThreadContext,InitialTeb, CreateSuspended); }

然后发现走不通,MmCopyVirtualMemory不稳定,非常容易蓝屏,自己插入APC然后用rtlmemory去拷贝内存死活拷贝不了,可能跟代码垃圾有关系.

转念一想,修改openprocess返回一个假句柄,然后驱动自己打开一个句柄,然后其他读写操作使用驱动里面的句柄进行读写操作

NTSTATUS NTAPI HookNtOpenProcess(PHANDLE ProcessHandle,ACCESS_MASK DesiredAccess,POBJECT_ATTRIBUTES ObjectAttributes,PCLIENT_ID ClientId) { NTSTATUS ret = STATUS_ACCESS_VIOLATION; if (g_startDebug) { ULONG pid = (ULONG)(ULONG_PTR)PsGetCurrentProcessId; if (g_DbgPid == pid && DesiredAccess == 0x001F0FFF) { g_GamePid = (ULONG)ClientId->UniqueProcess; PEPROCESS Process; if (NT_SUCCESS(PsLookupProcessByProcessId((HANDLE)g_GamePid, &Process))) { //自己打开一个句柄 if (g_oldGameHandle) ObDereferenceObject(g_oldGameHandle), g_oldGameHandle = 0; g_GameHandle = CreateNewHandle(Process); g_oldGameHandle = g_GameHandle; } ObDereferenceObject(Process); //不走调试器的句柄 *ProcessHandle = 0x1337; DebugPrint("[DebugMessage] Lock Game Id %d ACCESS_MASK : 0x%08X \r\n", g_GamePid, DesiredAccess); return STATUS_SUCCESS; } } return pfn_NtOpenProcess(ProcessHandle, DesiredAccess, ObjectAttributes, ClientId); }

然而现实当头一棒,因为要处理并且替换调试器的句柄到驱动的API很多,而且不光是进程的,线程的句柄也要处理!太多了,太烦躁了.如果不完全处理就会出现如下情况:

六、分析EAC

所以我决定冷静一下分析EAC:

首先 这段对应了开头的伏笔,InfinityHook只能HOOK R3的程序的东西,不能HOOK R0的 可EAC却在

内核里面通过 NtQuerySystemInformation 扫描所有进程的句柄,然后就GG:

所以,我们R3的处理在内核里面完全透明! 这也是为什么我接下来放弃用ETW HOOK的原因,毫无作用!在内核里面完全透明

并且我们的EAC通过扫描线程方式扫描隐藏进程:

所以我们的隐藏进程是完全没用的.

但是好处是,其他的关于调试器和硬断的检测在内核里面并没有太多实现.应该还是在R3实现.

七、峰回路转

通过分析R3的eac代码(没有任何加密,easyanticheat_x64.dll) 发现事实上EAC犯了致命错误,前面说到的句柄计数这个问题EAC关闭游戏居然是通过R3的exit函数来关闭游戏!

因此:

NTSTATUS NTAPI HookNtTerminateProcess(HANDLE ProcessHandle,NTSTATUS ExitStatus) { if (g_startDebug) { ULONG pid = (ULONG)(ULONG_PTR)PsGetCurrentProcessId; ULONG my_pid = GetProcessIDFromProcessHandle(ProcessHandle); if (pid == g_GamePid && my_pid == g_GamePid) { //由游戏发起的关闭指令目标也是游戏 //EAC: pls fix this return STATUS_SUCCESS; } else if (g_DbgPid == my_pid) { return STATUS_SUCCESS; } } return pfn_NtTerminateProcess(ProcessHandle, ExitStatus); }

结合之前的反调试计数,居然成功了(这里用居然是因为我也没想到EAC在R3关闭进程

八、结论

1.EAC太弱,BE不会出现R3关闭进程的情况.

2.这个方法会被检测,因为没有在内核层处理NtQueryInformationProcess导致EAC的驱动依然可以获得句柄并且banned.

但是无所谓了.因为换VT部分的了,至于又遇到了什么坑这是另外一个故事了.

EAC的逆向代码fork自国外大神: https://github.com/huoji120/EACReversing九、后记

笔者再写完这个文章后又重新看了一下代码,审视了一下

“转念一想,修改openprocess返回一个假句柄,然后驱动自己打开一个句柄,然后其他读写操作使用驱动里面的句柄进行读写操作”

其实完全行得通,只不过要花点时间给NtOpenThreat做处理还需要hook NtClose 这样自己的R3程序就不会有句柄,EAC也不会report了(句柄来自于驱动

但依然会被扫描到调试器进程和特征,为了完全避免特征,只能进入虚拟机去Hook这些API

这是笔者花了10分钟改进的版本,没有阻止EAC关闭进程:

此方法也适用于BE: