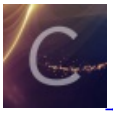
调试器攻击技术 - ThreadHideFromDebugger

转载

ReversalC 于 2015-03-19 22:31:20 发布 3476 收藏 3
分类专栏： 反调试技术


反调试技术 专栏收录该内容
5 篇文章 0 订阅
订阅专栏

3、 ThreadHideFromDebugger

这项技术用到了常常被用来设置线程优先级的API ntdll!NtSetInformationThread()，不过这个API也能够用来防止调试事件被发往调试器。

NtSetInformationThread()的参数列表如下。要实现这一功能，ThreadHideFromDebugger(0x11)被当作ThreadInformationClass参数传递，ThreadHandle通常设为当前线程的句柄(0xFFFFFFFE)：

```
NTSTATUS NTAPI NtSetInformationThread(
HANDLE                    ThreadHandle,
THREAD_INFORMATION_CLASS    ThreadInformaitonClass,
PVOID                     ThreadInformation,
ULONG                     ThreadInformationLength
);
```

ThreadHideFromDebugger内部设置内核结构ETHREAD[16]的HideThreadFromDebugger成员。一旦这个成员设置以后，主要用来向调试器发送事件的内核函数_DbgkpSendApiMessage()将不再被调用。

示例

调用NtSetInformationThread()的一个典型示例：

```
push      0                       ;InformationLength
push      NULL                    ;ThreadInformation
push      ThreadHideFromDebugger      ;0x11
push      0xfffffffe                  ;GetCurrentThread()
call      [NtSetInformationThread]
```

对策

可以在ntdll!NtSetInformationThread()里下断，断下来后，逆向分析人员可以操纵EIP防止API调用到达内核，这些都可以通过ollyscript来自动完成。另外，Olly Advanced插件也有补这个API的选项。补过之后一旦ThreadInformaitonClass参数为HideThreadFromDebugger，API将不再深入内核仅仅执行一个简单的返回。

【原创】inline hook SSDT 躲避 Themida 的ThreadHideFromDebugger （学习笔记2）

1 个附件

Themida保护的程序会采用ThreadHideFromDebugger来反调试，这个一旦设置，就一直起效，所以对于需要附加的程序就不太好办了，或者有的程序直接调用 int 2e来反调试, 虽然有插件比如 invisible可以躲过，但是需要附加进行调试的程序就不起作用了。

比如我以前写的一个反调试函数

```
void anti()
{

/
/
//NtSetInformationThread 功能设置调试端口为0
//
//
//

::GetCurrentThread();
_asm // xp系统
{
push 0 //InformationLength
push 0 //ThreadInformation
push 0x11 //11 就是ThreadHideFromDebugger
push eax //当前线程句柄
mov eax, 0xe5 //EAX NtSetInformationThread调用号
mov edx, esp //EDX 当前堆栈放
int 0x2e
add esp, 0x10
}


::GetCurrentThread();
_asm // 2000系统
{
push 0
push 0
push 0x11
push eax
mov eax, 0xc6
mov edx, esp
int 0x2e
add esp, 0x10
}
}
```

为了躲避上面的ANTI inline hook是个不错的方法

inline hook 分5步

第一步 申请一个全局变量 用来存放被破坏的指令
第二步 声明一个类型 用来强制转换
第三步 自己的函数
第四步 安装HOOK
第五部 卸载HOOK
其中起关键作用的是下面的代理函数

```
NTSTATUS __stdcall //第三步 自己的函数
MyNtSetInformationThread(
IN HANDLE ThreadHandle,
IN THREADINFOCLASS ThreadInformationClass,
IN PVOID ThreadInformation,
IN ULONG ThreadInformationLength
)
{
if(ThreadInformationClass == 0x11)
{
DbgPrint("发现 ANTI DEBUG 行为");

DbgPrint("NtSetInformationThread 参数 %8x %8x %8x %8x ", ThreadHandle, ThreadInformationClass, ThreadInformationClass,
ThreadInformationLength); // 打印出参数
ThreadInformationClass = (THREADINFOCLASS)0xff;

DbgPrint("参数被我改过后是 %8x %8x %8x %8x ", ThreadHandle, ThreadInformationClass, ThreadInformationClass,
ThreadInformationLength);

return STATUS_SUCCESS; //直接返回成功 这样ANTI就失效了
}

//强制转换成函数的形式
return ((SYSNTSETINFORMATIIONTHREAD)OldNtSetInformationThread)( // 放行
ThreadHandle,
ThreadInformationClass,
ThreadInformation,
ThreadInformationLength
);
}
```

下面是全部的代码 附件中也包含完成的代码
SYS加载有 你只要调用下 anti这个函数 代理函数就会输出参数 并且 改变参数

```
void * OldNtSetInformationThread;; //第一步 申请一个全局变量 用来存放被破坏的指令


//NtSetInformationThread
typedef NTSTATUS (__stdcall *SYSNTSETINFORMATIIONTHREAD) //第二步 声明一个类型 用来强制转换
(
IN HANDLE ThreadHandle,
IN THREADINFOCLASS ThreadInformationClass,
IN PVOID ThreadInformation,
IN ULONG ThreadInformationLength
);
```

```
NTSTATUS __stdcall //第三步 自己的函数
MyNtSetInformationThread(
IN HANDLE ThreadHandle,
IN THREADINFOCLASS ThreadInformationClass,
IN PVOID ThreadInformation,
IN ULONG ThreadInformationLength
)
{
if(ThreadInformationClass == 0x11)
{
DbgPrint("发现 ANTI DEBUG 行为");

DbgPrint("NtSetInformationThread 参数 %8x %8x %8x %8x ", ThreadHandle, ThreadInformationClass, ThreadInformationClass,
ThreadInformationLength); //证明自己存在

ThreadInformationClass = (THREADINFOCLASS)0xff;

DbgPrint("参数被我改过后是 %8x %8x %8x %8x ", ThreadHandle, ThreadInformationClass, ThreadInformationClass,
ThreadInformationLength); //证明自己存在

return STATUS_SUCCESS; //直接返回成功
}

//强制转换成函数的形式
return ((SYSNTSETINFORMATIIONTHREAD)OldNtSetInformationThread)( // 放行
ThreadHandle,
ThreadInformationClass,
ThreadInformation,
ThreadInformationLength
);
}


ULONG AddrNtSetInformationThread;
VOID OnUnload(IN PDRIVER_OBJECT DriverObject);
// 驱动程序加载时调用DriverEntry例程
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObj, PUNICODE_STRING pRegistryString)
{

DbgPrint("Hook NtSetInformationThread DriverEntry");
pDriverObj->DriverUnload = OnUnload;

AddrNtSetInformationThread = *PULONG((ULONG)KeServiceDescriptorTable->ServiceTable + 0xe5 * 4);
AfxHookCode((void*)AddrNtSetInformationThread, (void*)MyNtSetInformationThread, (void**)&OldNtSetInformationThread, 10); //第四步
AFXHOOK


// if(驱动加载失败)
// 一定要 AfxUnHookCode 不然驱动加载失败 自己分配的内存代码也就失效了 系统经过这个SSDT也就蓝了
return STATUS_SUCCESS;
}
```

```
//
VOID OnUnload(IN PDRIVER_OBJECT DriverObject)
{
AfxUnHookCode((void*)AddrNtSetInformationThread, OldNtSetInformationThread, 10); //第五部 卸载 驱动卸载一定要还原HOOK 因为驱动卸载了，自己分配的内存也就失效了
DbgPrint("Unload Hook NtSetInformationThread");
}
```

下面是RING 0 HOOK 模块

```
#ifndef __AFXHOOKCODE_H__
#define __AFXHOOKCODE_H__

bool AfxHookCode(void* TargetProc, void* NewProc,void ** l_OldProc, int bytescopy = 5)
{
*l_OldProc = ExAllocatePool(NonPagedPool,(bytescopy+5)); // 执行被覆盖的指令 再跳到原来的代码上运行


memcpy(*l_OldProc, TargetProc, bytescopy); // 事先保存被破坏的指令

__asm{
cli
mov eax,cr0
and eax,not 10000h
mov cr0,eax
}

// 我的内存的代码执行完 跳到原来的 代码+破坏的代码的长度 上去
*((unsigned char*)(*l_OldProc) + bytescopy) = 0xe9;

// 我内存代码跳到原来代码上的偏移
*(unsigned int *)((unsigned char*)(*l_OldProc) +bytescopy + 1)=(unsigned int)(TargetProc) + bytescopy - ( (unsigned int)((*l_OldProc)) + 5 + bytescopy ) ;

//被HOOK的函数头改为jmp
*(unsigned char*)TargetProc =(unsigned char)0xe9;

//被HOOK的地方跳到我的新过程 接受过滤
*(unsigned int*)((unsigned int)TargetProc +1) = (unsigned int)NewProc - ( (unsigned int)TargetProc + 5);

__asm{
mov eax,cr0
or eax,10000h
mov cr0,eax
sti
}
return true;
}
```

```
bool AfxUnHookCode(void* TargetAddress, void * l_SavedCode, unsigned int len)
{

__asm{
cli
mov eax,cr0
and eax,not 10000h
mov cr0,eax
}

// 恢复HOOK
memcpy(TargetAddress, l_SavedCode, len);

__asm{
mov eax,cr0
or eax,10000h
mov cr0,eax
sti
}

return true;
}

#endif
```