Windows 平台反调试相关的技术方法总结—part 3

惊鸿一瞥最是珍贵 / 2019-06-18 06:02:00 / 浏览数 5501 安全技术 二进制安全 顶(0) 踩(0)

如何绕过SEH检查

没有通用的方法来绕过SEH检查，但还是有一些小技巧节省逆向工程人员的工作量，让我们看看导致SEH处理程序调用的调用堆栈：

```
0:000> kn
# ChildEBP RetAddr
00 0059f06c 775100b1 AntiDebug!ExceptionRoutine
01 0059f090 77510083 ntdll!ExecuteHandler2+0x26
02 0059f158 775107ff ntdll!ExecuteHandler+0x24
03 0059f158 003b11a5 ntdll!KiUserExceptionDispatcher+0xf
04 0059fa90 003d7f4e AntiDebug!main+0xb5
05 0059faa4 003d7d9a AntiDebug!invoke_main+0x1e
06 0059fafc 003d7c2d AntiDebug!__scrt_common_main_seh+0x15a
07 0059fb04 003d7f68 AntiDebug!__scrt_common_main+0xd
08 0059fb0c 753e7c04 AntiDebug!mainCRTStartup+0x8
09 0059fb20 7752ad1f KERNEL32!BaseThreadInitThunk+0x24
0a 0059fb68 7752acea ntdll!__RtlUserThreadStart+0x2f
0b 0059fb78 00000000 ntdll!_RtlUserThreadStart+0x1b
```

可以看到该调用来自于ntdll!ExecuteHandler2。此函数是调用任何SEH处理程序的起点。可以在调用指令中设置断点：

```
0:000> u ntdll!ExecuteHandler2+24 L3
ntdll!ExecuteHandler2+0x24:
775100af ffd1            call    ecx
775100b1 648b2500000000  mov     esp,dword ptr fs:[0]
775100b8 648f0500000000  pop     dword ptr fs:[0]
0:000> bp 775100af
```

设置断点后，应该分析每个调用SEH处理程序的代码。如果涉及到对SEH处理函数的多次调用，则很难进行下一步反调试工作了。

VEH（向量化异常处理）

Veh是在WindowsXP中引入的，是SEH的变体。Veh和SEH不是相互依赖的，他们两个是可以同时工作的。添加新的VEH处理程序时，SEH链不会受到影响，因为VEH处理程

```
PVOID WINAPI AddVectoredExceptionHandler(
   ULONG                       FirstHandler,
   PVECTORED_EXCEPTION_HANDLER VectoredHandler
);
ULONG WINAPI RemoveVectoredExceptionHandler(
   PVOID Handler
);
LONG CALLBACK VectoredHandler(
   PEXCEPTION_POINTERS ExceptionInfo
);
The _EXCEPTION_POINTERS structure looks like this:
typedef struct _EXCEPTION_POINTERS {
 PEXCEPTION_RECORD ExceptionRecord;
 PCONTEXT          ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

在处理程序中接收控制权之后，系统收集当前进程上下文并通过ContextRecord参数传递。下面是使用向量异常处理的反调试保护代码示例：

```
LONG CALLBACK ExceptionHandler(PEXCEPTION_POINTERS ExceptionInfo)
{
   PCONTEXT ctx = ExceptionInfo->ContextRecord;
   if (ctx->Dr0 != 0 || ctx->Dr1 != 0 || ctx->Dr2 != 0 || ctx->Dr3 != 0)
   {
       std::cout << "Stop debugging program!" << std::endl;
       exit(-1);
   }
   ctx->Eip += 2;
   return EXCEPTION_CONTINUE_EXECUTION;
}
```

```
int main()
{
   AddVectoredExceptionHandler(0, ExceptionHandler);
   __asm int 1h;
   return 0;
}
```

在这里，我们设置了一个VEH处理程序，并生成了一个中断(不需要int1h)。当产生中断时，将出现异常并将控制权转移到VEH处理程序。如果设置了硬件断点，则程序执行将

如何绕过硬件断点检查和VEH

让我们看看导致VEH处理程序的调用堆栈：

```
0:000> kn
# ChildEBP RetAddr
00 001cf21c 774d6822 AntiDebug!ExceptionHandler
01 001cf26c 7753d151 ntdll!RtlpCallVectoredHandlers+0xba
02 001cf304 775107ff ntdll!RtlDispatchException+0x72
03 001cf304 00bf4a69 ntdll!KiUserExceptionDispatcher+0xf
04 001cfc1c 00c2680e AntiDebug!main+0x59
05 001cfc30 00c2665a AntiDebug!invoke_main+0x1e
06 001cfc88 00c264ed AntiDebug!__scrt_common_main_seh+0x15a
07 001cfc90 00c26828 AntiDebug!__scrt_common_main+0xd
08 001cfc98 753e7c04 AntiDebug!mainCRTStartup+0x8
09 001cfcac 7752ad1f KERNEL32!BaseThreadInitThunk+0x24
0a 001cfcf4 7752acea ntdll!__RtlUserThreadStart+0x2f
0b 001cfd04 00000000 ntdll!_RtlUserThreadStart+0x1b
```

控制权已从main+0x59转移到ntdll!KiUserExceptionDispatcher。接下来看一下main+0x59中负责该操作的具体指令：

```
0:000> u main+59 L1
AntiDebug!main+0x59
00bf4a69 cd02            int     1
```

KiUserExceptionDispatcher函数是系统从内核模式调用到用户模式的回调方法之一。以下是它的签名：

```
VOID NTAPI KiUserExceptionDispatcher(
   PEXCEPTION_RECORD pExcptRec,
   PCONTEXT ContextFrame
);
```

下一个代码示例演示如何通过KiUserExceptionDispatcher函数钩子来绕过硬件断点检查：

```
typedef  VOID (NTAPI *pfnKiUserExceptionDispatcher)(
   PEXCEPTION_RECORD pExcptRec,
   PCONTEXT ContextFrame
   );
pfnKiUserExceptionDispatcher g_origKiUserExceptionDispatcher = NULL;
VOID NTAPI HandleKiUserExceptionDispatcher(PEXCEPTION_RECORD pExcptRec, PCONTEXT ContextFrame)
{
   if (ContextFrame && (CONTEXT_DEBUG_REGISTERS & ContextFrame->ContextFlags))
   {
       ContextFrame->Dr0 = 0;
       ContextFrame->Dr1 = 0;
       ContextFrame->Dr2 = 0;
       ContextFrame->Dr3 = 0;
       ContextFrame->Dr6 = 0;
       ContextFrame->Dr7 = 0;
       ContextFrame->ContextFlags &= ~CONTEXT_DEBUG_REGISTERS;
   }
}
__declspec(naked) VOID NTAPI HookKiUserExceptionDispatcher()
// Params: PEXCEPTION_RECORD pExcptRec, PCONTEXT ContextFrame
{
   __asm
   {
       mov eax, [esp + 4]
       mov ecx, [esp]
       push eax
       push ecx
```

```
        call HandleKiUserExceptionDispatcher
        jmp g_origKiUserExceptionDispatcher
    }
}
int main()
{
    HMODULE hNtDll = LoadLibrary(TEXT("ntdll.dll"));
    g_origKiUserExceptionDispatcher = (pfnKiUserExceptionDispatcher)GetProcAddress(hNtDll, "KiUserExceptionDispatcher");
    Mhook_SetHook((PVOID*)&g_origKiUserExceptionDispatcher, HookKiUserExceptionDispatcher);
    return 0;
}
```

在本例中，DRx寄存器的值在HookKiUserExceptionDispatcher函数中重置，

NtSetInformationThread-从调试器中隐藏线程

在Windows2000中，出现了传递到NtSetInformationThread函数的新线程信息类-ThreadHideFromDebugger。它是Windows提供的第一个反调试技术之一，功能非常

```
1: kd> dt _ETHREAD HideFromDebugger 86bfada8
ntdll!_ETHREAD
  +0x248 HideFromDebugger : 0y1
```

下面是如何从调试器设置ThreadHideFromDebugger的示例：

```
typedef NTSTATUS (NTAPI *pfnNtSetInformationThread)(
    _In_ HANDLE ThreadHandle,
    _In_ ULONG  ThreadInformationClass,
    _In_ PVOID  ThreadInformation,
    _In_ ULONG  ThreadInformationLength
    );
const ULONG ThreadHideFromDebugger = 0x11;
void HideFromDebugger()
{
    HMODULE hNtDll = LoadLibrary(TEXT("ntdll.dll"));
    pfnNtSetInformationThread NtSetInformationThread = (pfnNtSetInformationThread)
        GetProcAddress(hNtDll, "NtSetInformationThread");
    NTSTATUS status = NtSetInformationThread(GetCurrentThread(),
        ThreadHideFromDebugger, NULL, 0);
}
```

如何绕过从调试器隐藏线程

为了防止应用程序对调试器隐藏线程，需要钩住NtSetInformationThread函数调用。

```
pfnNtSetInformationThread g_origNtSetInformationThread = NULL;
NTSTATUS NTAPI HookNtSetInformationThread(
    _In_ HANDLE ThreadHandle,
    _In_ ULONG  ThreadInformationClass,
    _In_ PVOID  ThreadInformation,
    _In_ ULONG  ThreadInformationLength
    )
{
    if (ThreadInformationClass == ThreadHideFromDebugger &&
        ThreadInformation == 0 && ThreadInformationLength == 0)
    {
        return STATUS_SUCCESS;
    }
    return g_origNtSetInformationThread(ThreadHandle,
        ThreadInformationClass, ThreadInformation, ThreadInformationLength
}

void SetHook()
{
    HMODULE hNtDll = LoadLibrary(TEXT("ntdll.dll"));
    if (NULL != hNtDll)
    {
        g_origNtSetInformationThread = (pfnNtSetInformationThread)GetProcAddress(hNtDll, "NtSetInformationThread");
        if (NULL != g_origNtSetInformationThread)
        {
```

```
            Mhook_SetHook((PVOID*)&g_origNtSetInformationThread, HookNtSetInformationThread);
        }
    }
}
```

在钩子函数中，当正确调用时，会返回STATUS_SUCCESS，而不会将控制权转移到原始的NtSetInformationThread函数。

NtCreateThreadEx

Windows Vista引入了NtCreateThreadEx函数，其签名如下：

```
NTSTATUS NTAPI NtCreateThreadEx (
   _Out_    PHANDLE              ThreadHandle,
   _In_     ACCESS_MASK          DesiredAccess,
   _In_opt_ POBJECT_ATTRIBUTES   ObjectAttributes,
   _In_     HANDLE               ProcessHandle,
   _In_     PVOID                StartRoutine,
   _In_opt_ PVOID                Argument,
   _In_     ULONG                CreateFlags,
   _In_opt_ ULONG_PTR            ZeroBits,
   _In_opt_ SIZE_T               StackSize,
   _In_opt_ SIZE_T               MaximumStackSize,
   _In_opt_ PVOID                AttributeList
);
```

最有趣的参数是CreateFlgs。此参数获取如下标志：

```
#define THREAD_CREATE_FLAGS_CREATE_SUSPENDED 0x00000001
#define THREAD_CREATE_FLAGS_SKIP_THREAD_ATTACH 0x00000002
#define THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER 0x00000004
#define THREAD_CREATE_FLAGS_HAS_SECURITY_DESCRIPTOR 0x00000010
#define THREAD_CREATE_FLAGS_ACCESS_CHECK_IN_TARGET 0x00000020
#define THREAD_CREATE_FLAGS_INITIAL_THREAD 0x00000080
```

如果新线程获得THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER标志，则在创建时将对调试器隐藏该线程。这与NtSetInformationThread函数设置的ThreadHideFro

如何绕过NtCreateThreadEx

可以通过钩住NtCreateThreadEx函数绕过此技术，在NtCreateThreadEx函数中，其中`THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER`将被重置。

句柄跟踪

从WindowsXP开始，Windows系统已经有了跟踪内核对象句柄的机制。当跟踪模式处于启用状态时，所有具有处理程序的操作都将保存到循环缓冲区中，同时，当尝试使用

```
EXCEPTION_DISPOSITION ExceptionRoutine(
   PEXCEPTION_RECORD ExceptionRecord,
   PVOID             EstablisherFrame,
   PCONTEXT          ContextRecord,
   PVOID             DispatcherContext)
{
   if (EXCEPTION_INVALID_HANDLE == ExceptionRecord->ExceptionCode)
   {
       std::cout << "Stop debugging program!" << std::endl;
       exit(-1);
   }
   return ExceptionContinueExecution;
}
int main()
{
   __asm
   {
       // set SEH handler
       push ExceptionRoutine
       push dword ptr fs : [0]
       mov  dword ptr fs : [0], esp
   }
   CloseHandle((HANDLE)0xBAAD);
   __asm
```

```
   {
       // return original SEH handler
       mov  eax, [esp]
       mov  dword ptr fs : [0], eax
       add  esp, 8
   }
   return 0
}
```

堆栈段操作

在操作ss堆栈段寄存器时，调试器跳过指令跟踪。在下一个示例中，调试器将立即移到 xor edx, edx 指令，同时执行上一个指令：

```
__asm
{
   push ss
   pop  ss
   mov  eax, 0xC000C1EE // This line will be traced over by debugger
   xor  edx, edx        // Debugger will step to this line
}
```

调试信息

自Windows10以来，OutputDebugString函数的实现已更改为带有特定参数的简单RaiseException调用。因此，调试输出异常现在必须由调试器处理。
有两种异常类型：`DBG_PRINTEXCEPTION_C(0x40010006)`和`DBG_PRINTEXCEPTION_W(0x4001000A)`，可用于检测调试器是否存在

```
#define DBG_PRINTEXCEPTION_WIDE_C 0x4001000A
WCHAR * outputString = L"Any text";
ULONG_PTR args[4] = {0};
args[0] = (ULONG_PTR)wcslen(outputString) + 1;
args[1] = (ULONG_PTR)outputString;
__try
{
   RaiseException(DBG_PRINTEXCEPTION_WIDE_C, 0, 4, args);
   printf("Debugger detected");
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
   printf("Debugger NOT detected");
}
```

因此，如果异常未处理，则意味着没有附加调试器。
`DBG_PRINTEXCEPTION_W`用于宽字符输出，`DBG_PRINTEXCEPTION_C`用于ansi字符。这表示在使用`DBG_PRINTEXCEPTION_C`的情况下，arg[0]会保存strlen()的结果，而
*■。

总结

本文从最简单处入手，描述了一系列反向工程技术，特别是反调试方法，并描述了绕过它们的方法。但技术总是层出不穷，还有很多技术本文并没有提及，比如：
1.自调试过程；

2.使用FindWindow函数进行调试器检测；

3.时间计算方法；

4.NtQueryObject；

5.BlockInput；

6.NtSetDebugFilterState；

7.自修改代码；
虽然我们关注的是反调试保护方法，但也有其他反逆向工程方法，包括反转储和混淆技术。
我们要再次强调，即使是最好的反逆向工程技术也不能完全保护软件不被逆向破解。反逆向工程技术主要目的是加大逆向工程的难度。

References

https://msdn.microsoft.com/library
http://www.infosecinstitute.com/

http://pferrie.tripod.com/
http://www.openrce.org/articles/
http://www.nynaeve.net/
http://stackoverflow.com/
http://x86.renejeschke.de/

点击收藏 | 1 关注 | 2

上一篇：bug bounty实例：框架注入... 下一篇：谈js静态文件在漏洞挖掘中的利用

1. 0 条回复

- 动动手指，沙发就是你的了！