Windows 平台反调试相关的技术方法总结—part 2

惊鸿一瞥最是珍贵 / 2019-06-05 09:14:00 / 浏览数 5075 安全技术 二进制安全 顶(0) 踩(0)

陷阱标识检查

陷阱标志(TF)位于EFLAGS寄存器内。如果TF设置为1，CPU将在每个指令执行后产生INT 01h或'单步'异常。以下反调试示例基于TF设置和异常调用检查：

```
BOOL isDebugged = TRUE;
__try
{
   __asm
   {
       pushfd
       or dword ptr[esp], 0x100 // set the Trap Flag
       popfd                    // Load the value into EFLAGS register
       nop
   }
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
   // If an exception has been raised – debugger is not present
   isDebugged = FALSE;
}
if (isDebugged)
{
   std::cout << "Stop debugging program!" << std::endl;
   exit(-1);
}
```

在这里，tf被故意设置为生成异常。如果正在调试进程，则异常将被调试器捕获。

如何绕过TF检查

要在调试期间绕过TF标志检查，请不要单步执行pushfd指令，而要跳过它，在它后面放置断点并继续执行程序。在它后面放置断点并继续执行程序。在断点之后，可以继续

CheckRemoteDebuggerPresent和NtQueryInformationProcess

与IsDebuggerPresent函数不同，CheckRemoteDebuggerPresent.aspx
"CheckRemoteDebuggerPresent")检查一个进程是否正在被另一个并行进程调试。以下是基于CheckRemoteDebuggerPresent的反调试技术示例：

```
int main(int argc, char *argv[])
{
   BOOL isDebuggerPresent = FALSE;
   if (CheckRemoteDebuggerPresent(GetCurrentProcess(), &isDebuggerPresent ))
   {
       if (isDebuggerPresent )
       {
           std::cout << "Stop debugging program!" << std::endl;
           exit(-1);
       }
   }
   return 0;
}
```

在CheckRemoteDebuggerPresent内部，调用NtQueryInformationProcess函数：

```
0:000> uf kernelbase!CheckRemotedebuggerPresent
KERNELBASE!CheckRemoteDebuggerPresent:
...
75207a24 6a00            push    0
75207a26 6a04            push    4
75207a28 8d45fc          lea     eax,[ebp-4]
75207a2b 50              push    eax
75207a2c 6a07            push    7
75207a2e ff7508          push    dword ptr [ebp+8]
```

```
75207a31 ff151c602775    call    dword ptr [KERNELBASE!_imp__NtQueryInformationProcess (7527601c)]
75207a37 85c0            test    eax,eax
75207a39 0f88607e0100    js      KERNELBASE!CheckRemoteDebuggerPresent+0x2b (7521f89f)
...
```

如果我们看一下NtQueryInformationProcess文档，这个汇编程序列表将向我们显示CheckRemoteDebuggerPresent函数被分配了DebugPort值，因为ProcessInformati

```
typedef NTSTATUS(NTAPI *pfnNtQueryInformationProcess)(
   _In_      HANDLE           ProcessHandle,
   _In_      UINT             ProcessInformationClass,
   _Out_     PVOID            ProcessInformation,
   _In_      ULONG            ProcessInformationLength,
   _Out_opt_ PULONG           ReturnLength
   );
const UINT ProcessDebugPort = 7;
int main(int argc, char *argv[])
{
   pfnNtQueryInformationProcess NtQueryInformationProcess = NULL;
   NTSTATUS status;
   DWORD isDebuggerPresent = 0;
   HMODULE hNtDll = LoadLibrary(TEXT("ntdll.dll"));

   if (NULL != hNtDll)
   {
       NtQueryInformationProcess = (pfnNtQueryInformationProcess)GetProcAddress(hNtDll, "NtQueryInformationProcess");
       if (NULL != NtQueryInformationProcess)
       {
           status = NtQueryInformationProcess(
               GetCurrentProcess(),
               ProcessDebugPort,
               &isDebuggerPresent,
               sizeof(DWORD),
               NULL);
           if (status == 0x00000000 && isDebuggerPresent != 0)
           {
               std::cout << "Stop debugging program!" << std::endl;
               exit(-1);
           }
       }
   }
   return 0;
}
```

如何绕过CheckRemoteDebuggerPresent和NtQueryInformationProcess

若要绕过CheckRemoteDebuggerPresent和NTQueryInformationProcess，需要替换NtQueryInformationProcess函数返回的值，您可以使用mhook来完成此操作。若

```
#include <Windows.h>
#include "mhook.h"
typedef NTSTATUS(NTAPI *pfnNtQueryInformationProcess)(
   _In_      HANDLE           ProcessHandle,
   _In_      UINT             ProcessInformationClass,
   _Out_     PVOID            ProcessInformation,
   _In_      ULONG            ProcessInformationLength,
   _Out_opt_ PULONG           ReturnLength
   );
const UINT ProcessDebugPort = 7;
pfnNtQueryInformationProcess g_origNtQueryInformationProcess = NULL;
NTSTATUS NTAPI HookNtQueryInformationProcess(
   _In_      HANDLE           ProcessHandle,
   _In_      UINT             ProcessInformationClass,
   _Out_     PVOID            ProcessInformation,
   _In_      ULONG            ProcessInformationLength,
   _Out_opt_ PULONG           ReturnLength
   )
{
   NTSTATUS status = g_origNtQueryInformationProcess(
       ProcessHandle,
       ProcessInformationClass,
```

```
        ProcessInformation,
        ProcessInformationLength,
        ReturnLength);
    if (status == 0x00000000 && ProcessInformationClass == ProcessDebugPort)
    {
        *((PDWORD_PTR)ProcessInformation) = 0;
    }
    return status;
}
DWORD SetupHook(PVOID pvContext)
{
    HMODULE hNtDll = LoadLibrary(TEXT("ntdll.dll"));
    if (NULL != hNtDll)
    {
        g_origNtQueryInformationProcess = (pfnNtQueryInformationProcess)GetProcAddress(hNtDll, "NtQueryInformationProcess");
        if (NULL != g_origNtQueryInformationProcess)
        {
            Mhook_SetHook((PVOID*)&g_origNtQueryInformationProcess, HookNtQueryInformationProcess);
        }
    }
    return 0;
}
BOOL WINAPI DllMain(HINSTANCE hInstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
    switch (fdwReason)
    {
    case DLL_PROCESS_ATTACH:
        DisableThreadLibraryCalls(hInstDLL);
        CreateThread(NULL, NULL, (LPTHREAD_START_ROUTINE)SetupHook, NULL, NULL, NULL);
        Sleep(20);
    case DLL_PROCESS_DETACH:
        if (NULL != g_origNtQueryInformationProcess)
        {
            Mhook_Unhook((PVOID*)&g_origNtQueryInformationProcess);
        }
        break;
    }
    return TRUE;
}
```

基于NtQueryInformationProcess的其它反调试保护技术

从NtQueryInformationProcess函数中提供的信息，我们可以知道有很多调试器检测技术：

1.ProcessDebugPort 0x07[redacted]

2.ProcessDebugObjectHandle 0x1E

3.ProcessDebugFlags 0x1F

4.ProcessBasicInformation 0x00

ProcessDebugObjectHandle

我们将详细考虑第2条和第4条

ProcessDebugObjectHandle

从WindowsXP开始，将为调试的进程创建一个"调试对象"。以下就是检查当前进程调试对象的例子：

```
status = NtQueryInformationProcess(
            GetCurrentProcess(),
            ProcessDebugObjectHandle,
            &hProcessDebugObject,
            sizeof(HANDLE),
            NULL);
if (0x00000000 == status && NULL != hProcessDebugObject)
{
    std::cout << "Stop debugging program!" << std::endl;
```

```
    exit(-1);
}
```

如果存在调试对象，则正在调试该进程。

ProcessDebugFlags

当检查该标识时，它会返回到EPROCESS内核结构的NoDebugInherit位的反转值。如果NtQueryInformationProcess函数的返回值为0，则正在调试该进程。以下是此类反

```
status = NtQueryInformationProcess(
    GetCurrentProcess(),
    ProcessDebugObjectHandle,
    &debugFlags,
    sizeof(ULONG),
    NULL);
if (0x00000000 == status && NULL != debugFlags)
{
    std::cout << "Stop debugging program!" << std::endl;
    exit(-1);
}
```

ProcessBasicInformation

当使用ProcessBasicInformation标志调用NtQueryInformationProcess函数时，将返回PROCESS_BASIC_INGISION结构：

```
typedef struct _PROCESS_BASIC_INFORMATION {
    NTSTATUS ExitStatus;
    PVOID PebBaseAddress;
    ULONG_PTR AffinityMask;
    KPRIORITY BasePriority;
    HANDLE UniqueProcessId;
    HANDLE InheritedFromUniqueProcessId;
} PROCESS_BASIC_INFORMATION, *PPROCESS_BASIC_INFORMATION;
```

该结构中最有趣的是InheritedFromUniqueProcessId字段。在这里，我们需要获取父进程的名称并将其与流行调试器的名称进行比较，下是这种反调试检查的示例：

```
std::wstring GetProcessNameById(DWORD pid)
{
    HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (hProcessSnap == INVALID_HANDLE_VALUE)
    {
        return 0;
    }
    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(PROCESSENTRY32);
    std::wstring processName = L"";
    if (!Process32First(hProcessSnap, &pe32))
    {
        CloseHandle(hProcessSnap);
        return processName;
    }
    do
    {
        if (pe32.th32ProcessID == pid)
        {
            processName = pe32.szExeFile;
            break;
        }
    } while (Process32Next(hProcessSnap, &pe32));

    CloseHandle(hProcessSnap);
    return processName;
}
status = NtQueryInformationProcess(
    GetCurrentProcess(),
    ProcessBasicInformation,
    &processBasicInformation,
    sizeof(PROCESS_BASIC_INFORMATION),
    NULL);
```

```
std::wstring parentProcessName = GetProcessNameById((DWORD)processBasicInformation.InheritedFromUniqueProcessId);
if (L"devenv.exe" == parentProcessName)
{
   std::cout << "Stop debugging program!" << std::endl;
   exit(-1);
}
```

如何绕过NtQueryInformationProcess检查

绕过NtQueryInformation进程检查非常简单。NtQueryInformationProcess函数返回的值应更改为不表示存在调试器的值：

1.■ProcessDebugObjectHandle■■■0

2.■ProcessDebugFlags■■■1

3.■■ProcessBasicInformation■■InheritedFromUniqueProcessId■■■■■■■■■ID■

断点

断点是调试器提供的主要功能。断点允许您在指定的位置中断程序执行。有两种类型的断点：
软件断点
硬件断点
在没有断点的情况下对软件进行逆向工程是非常困难的。目前流行的反逆向工程策略都是以检测断点为基础，提供了一系列相应的反调试方法。

软件断点

在IA-32架构中，有一个特定的指令 - 带有0xCC操作码的int 3h -
用于调用调试句柄。当CPU执行此指令时，会产生中断并将控制权转移到调试器。为了获得控制，调试器必须将int3h指令注入到代码中。要检测断点，我们可以计算函数的

```
DWORD CalcFuncCrc(PUCHAR funcBegin, PUCHAR funcEnd)
{
   DWORD crc = 0;
   for (; funcBegin < funcEnd; ++funcBegin)
   {
       crc += *funcBegin;
   }
   return crc;
}
#pragma auto_inline(off)
VOID DebuggeeFunction()
{
   int calc = 0;
   calc += 2;
   calc <<= 8;
   calc -= 3;
}
VOID DebuggeeFunctionEnd()
{
};
#pragma auto_inline(on)
DWORD g_origCrc = 0x2bd0;
int main()
{
   DWORD crc = CalcFuncCrc((PUCHAR)DebuggeeFunction, (PUCHAR)DebuggeeFunctionEnd);
   if (g_origCrc != crc)
   {
       std::cout << "Stop debugging program!" << std::endl;
       exit(-1);
   }
   return 0;
}
```

值得一提的是，这只在/INCREMENTAL:NO链接器选项设置的情况下才起作用，否则，在获取函数地址以计算校验和时，我们将获得相对跳转地址：

```
DebuggeeFunction:
013C16DB  jmp         DebuggeeFunction (013C4950h)
```

g_origCrc全局变量包含已经由CalcFuncCrc函数计算的CRC。为了终止检测函数，我们使用了存根函数的技巧。由于函数代码是按顺序放置的，所以DebuggeeFunction
auto_inline■off■指令来防止编译器的嵌入函数。

如何绕过软件断点检查

没有一种通用的方法可以绕过软件断点检查。要绕过这种保护，您应该找到计算校验和的代码，并用常量替换返回值，以及存储函数校验和的所有变量的值。

硬件断点

在x86体系结构中，有一组调试寄存器供开发人员在检查和调试代码时使用。这些寄存器允许您在访问内存进行读取或写入时中断程序执行并将控制转移到调试器。调试寄存
= 0的实模式或安全模式下由程序使用。8字节的调试寄存器DR0-DR7有：

```
1.DR0-DR3 -■■■■■
2.DR4■DR5 -■■
3.DR6 -■■■■
4.DR7 – ■■■■
```

DR0-DR3包含断点的线性地址。在物理地址转换之前对这些地址进行比较。在DR7寄存器中分别描述这些断点中的每个断点。DR6寄存器指示哪个断点被激活。DR7通过访

```
CONTEXT ctx = {};
ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
if (GetThreadContext(GetCurrentThread(), &ctx))
{
   if (ctx.Dr0 != 0 || ctx.Dr1 != 0 || ctx.Dr2 != 0 || ctx.Dr3 != 0)
   {
       std::cout << "Stop debugging program!" << std::endl;
       exit(-1);
   }
}
```

也可以通过SetThreadContext函数重置硬件断点。以下是硬件断点重置的示例：

```
CONTEXT ctx = {};
ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
SetThreadContext(GetCurrentThread(), &ctx);
```

我们可以看到，所有DRx寄存器都设置为0。

如何绕过硬件断点检查和重置

如果我们查看GetThreadContext函数内部，就会发现它调用了NtGetContextThread函数：

```
0:000> u KERNELBASE!GetThreadContext L6
KERNELBASE!GetThreadContext:
7538d580 8bff            mov     edi,edi
7538d582 55              push    ebp
7538d583 8bec            mov     ebp,esp
7538d585 ff750c          push    dword ptr [ebp+0Ch]
7538d588 ff7508          push    dword ptr [ebp+8]
7538d58b ff1504683975    call    dword ptr [KERNELBASE!_imp__NtGetContextThread (75396804)]
```

若反调试保护在Dr0-DR7中接收到零值，请重置上下文结构的ContextFlages字段中的CONTEXT_DEBUG_RESTRIGS标志，然后在原始的NtGetContextThread函数调用之

```
typedef NTSTATUS(NTAPI *pfnNtGetContextThread)(
   _In_  HANDLE             ThreadHandle,
   _Out_ PCONTEXT           pContext
   );
typedef NTSTATUS(NTAPI *pfnNtSetContextThread)(
   _In_ HANDLE              ThreadHandle,
   _In_ PCONTEXT            pContext
   );
pfnNtGetContextThread g_origNtGetContextThread = NULL;
pfnNtSetContextThread g_origNtSetContextThread = NULL;
NTSTATUS NTAPI HookNtGetContextThread(
   _In_  HANDLE              ThreadHandle,
   _Out_ PCONTEXT            pContext)
{
   DWORD backupContextFlags = pContext->ContextFlags;
   pContext->ContextFlags &= ~CONTEXT_DEBUG_REGISTERS;
```

```
   NTSTATUS status = g_origNtGetContextThread(ThreadHandle, pContext);
   pContext->ContextFlags = backupContextFlags;
   return status;
}
NTSTATUS NTAPI HookNtSetContextThread(
   _In_ HANDLE              ThreadHandle,
   _In_ PCONTEXT            pContext)
{
   DWORD backupContextFlags = pContext->ContextFlags;
   pContext->ContextFlags &= ~CONTEXT_DEBUG_REGISTERS;
   NTSTATUS status = g_origNtSetContextThread(ThreadHandle, pContext);
   pContext->ContextFlags = backupContextFlags;
   return status;
}
void HookThreadContext()
{
 HMODULE hNtDll = LoadLibrary(TEXT("ntdll.dll"));
 g_origNtGetContextThread = (pfnNtGetContextThread)GetProcAddress(hNtDll, "NtGetContextThread");
 g_origNtSetContextThread = (pfnNtSetContextThread)GetProcAddress(hNtDll, "NtSetContextThread");
 Mhook_SetHook((PVOID*)&g_origNtGetContextThread, HookNtGetContextThread);
 Mhook_SetHook((PVOID*)&g_origNtSetContextThread, HookNtSetContextThread);
}
```

SEH（结构化异常处理）

结构化异常处理是操作系统向应用程序提供的一种机制，允许应用程序接收有关异常情况的通知，如除数是零、引用不存在的指针或执行受限指令。此机制允许您在不涉及技

```
0:000> dt ntdll!_EXCEPTION_REGISTRATION_RECORD
  +0x000 Next             : Ptr32 _EXCEPTION_REGISTRATION_RECORD
  +0x004 Handler          : Ptr32 _EXCEPTION_DISPOSITION
```

启动异常时，控制权将转移到当前SEH处理程序。根据具体情况，此SEH处理程序应返回_EXCEPTION_DANDITY的一个值：

```
typedef enum _EXCEPTION_DISPOSITION {
   ExceptionContinueExecution,
   ExceptionContinueSearch,
   ExceptionNestedException,
   ExceptionCollidedUnwind
} EXCEPTION_DISPOSITION;
```

如果处理程序返回ExceptionContinueSearch，系统将继续从触发异常的指令执行。如果处理程序不知道如何处理异常，则返回ExceptionContinueSearch，然后系统移动

```
0:000> !exchain
00a5f3bc: AntiDebug!_except_handler4+0 (008b7530)
 CRT scope  0, filter: AntiDebug!SehInternals+67 (00883d67)
              func:   AntiDebug!SehInternals+6d (00883d6d)
00a5f814: AntiDebug!__scrt_stub_for_is_c_termination_complete+164b (008bc16b)
00a5f87c: AntiDebug!_except_handler4+0 (008b7530)
 CRT scope  0, filter: AntiDebug!__scrt_common_main_seh+1b0 (008b7c60)
              func:   AntiDebug!__scrt_common_main_seh+1cb (008b7c7b)
00a5f8e8: ntdll!_except_handler4+0 (775674a0)
 CRT scope  0, filter: ntdll!__RtlUserThreadStart+54386 (7757f076)
              func:   ntdll!__RtlUserThreadStart+543cd (7757f0bd)
00a5f900: ntdll!FinalExceptionHandlerPad4+0 (77510213)
```

链中的最后一个是系统分配的默认处理程序。如果以前的处理程序都无法处理异常，则系统处理程序将转到注册表以获取

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\AeDebug
```

根据AeDebug键值，要么终止应用程序，要么将控制转移到调试器。调试器路径应在调试器REG_SZ中指示。
创建新流程时，系统会将主SEH框架添加到其中。主SEH框架的处理程序也由系统定义。主SEH框架本身几乎位于为进程分配的内存堆栈的最开始处。SEH处理程序函数签名

```
typedef EXCEPTION_DISPOSITION (*PEXCEPTION_ROUTINE) (
   __in struct _EXCEPTION_RECORD *ExceptionRecord,
   __in PVOID EstablisherFrame,
   __inout struct _CONTEXT *ContextRecord,
   __inout PVOID DispatcherContext
   );
```

如果正在调试应用程序，则在生成int 3h中断后，控制将被调试器截取。否则，控制权将转移到SEH处理程序。以下代码示例显示基于SEH框架的反调试保护：

```
BOOL g_isDebuggerPresent = TRUE;
EXCEPTION_DISPOSITION ExceptionRoutine(
   PEXCEPTION_RECORD ExceptionRecord,
   PVOID             EstablisherFrame,
   PCONTEXT          ContextRecord,
   PVOID             DispatcherContext)
{
   g_isDebuggerPresent = FALSE;
   ContextRecord->Eip += 1;
   return ExceptionContinueExecution;
}
int main()
{
   __asm
   {
       // set SEH handler
       push ExceptionRoutine
       push dword ptr fs:[0]
       mov  dword ptr fs:[0], esp
       // generate interrupt
       int  3h
       // return original SEH handler
       mov  eax, [esp]
       mov  dword ptr fs:[0], eax
       add  esp, 8
   }
   if (g_isDebuggerPresent)
   {
       std::cout << "Stop debugging program!" << std::endl;
       exit(-1);
   }
   return 0
}
```

在本例中，设置了SEH处理程序。指向此处理程序的指针放在处理程序链的开头。然后生成int3h中断。如果未调试应用程序，则控制权将转移到SEH处理程序，并且g_isDe
ContextRecord-> Eip + = 1行更改执行流程中下一条指令的地址，这将导致执行int
3h后的指令。然后，代码返回原始SEH处理程序，清除堆栈，并检查是否存在调试器。

未完待续.

上一篇：CVE-2019-0697：通过D... 下一篇：bugbounty：利用文件上传 ...