Windows 平台反调试相关的技术方法总结

惊鸿一瞥最是珍贵 / 2019-06-02 07:54:00 / 浏览数 4349 安全技术 二进制安全 顶(0) 踩(0)

---

■■■https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software

## 前言

在软件方面，逆向工程是研究程序以获得有关其工作原理和使用的算法的封闭信息的过程。虽然逆向工程可以用于合法目的，特别是恶意软件分析或无证系统研究，但提起它

## 反调试方法简介

这里有几种分析软件的方法：
1.使用数据包嗅探器分析通过网络交换的数据。
2.软件二进制代码反汇编
3.二进制或字节码的反编译，以高级编程语言重新创建源代码。
本文考虑了当下流行的防破解和反逆向工程保护技术，即Windows中的反调试方法。要完全保护软件免受逆向工程的影响是不可能的。各种反逆向工程技术的主要目标就是
要想防御，得先知道攻击来自哪里。本文介绍了流行的反调试技术，循序渐进，并且详细阐明如何绕过它们。我们不会考虑不同的软件保护理论，只考虑实际例子。

## IsDebuggerPresent

也许最简单的反调试方法是调用`IsDebuggerPresent`函数。此函数检测用户模式调试器是否正在调试调用进程。最简单的一个例子：

```
int main()
{
   if (IsDebuggerPresent())
   {
       std::cout << "Stop debugging program!" << std::endl;
       exit(-1);
   }
   return 0;
}
```

如果我们查看IsDebuggerPresent函数内部，会发现以下代码：

```
0:000< u kernelbase!IsDebuggerPresent L3
KERNELBASE!IsDebuggerPresent:
751ca8d0 64a130000000    mov     eax,dword ptr fs:[00000030h]
751ca8d6 0fb64002        movzx   eax,byte ptr [eax+2]
751ca8da c3              ret
```

对于64位进程

```
0:000< u kernelbase!IsDebuggerPresent L3
KERNELBASE!IsDebuggerPresent:
00007ffc`ab6c1aa0 65488b042560000000 mov   rax,qword ptr gs:[60h]
00007ffc`ab6c1aa9 0fb64002           movzx eax,byte ptr [rax+2]
00007ffc`ab6c1aad c3                 ret
```

我们通过相对于fs段的30h偏移量(x64系统相对于gs段的60h偏移量)来查看PEB(进程环境块)结构。如果我们查看PEB中的2个偏移量，就会发现BeingDebugged字段：

```
0:000< dt _PEB
ntdll!_PEB
  +0x000 InheritedAddressSpace : UChar
  +0x001 ReadImageFileExecOptions : UChar
  +0x002 BeingDebugged    : UChar
```

换句话说，IsDebuggerPresent函数读取BeingDebugged字段的值。如果正在调试进程，则值为1，否则为0。

## PEB (进程环境块)

PEB是Windows操作系统中使用的一个封闭结构。根据环境的不同，您需要以不同的方式获得PEB结构指针。
下面是如何为x32和x64系统获取PEB指针的例子:

```
// Current PEB for 64bit and 32bit processes accordingly
PVOID GetPEB()
```

```
{
#ifdef _WIN64
   return (PVOID)__readgsqword(0x0C * sizeof(PVOID));
#else
   return (PVOID)__readfsdword(0x0C * sizeof(PVOID));
#endif
}
```

WOW64机制用于在x64系统上启动x32进程，并创建另一个PEB结构。以下是如何在WOW64环境中获取PEB结构指针的示例：

```
// Get PEB for WOW64 Process
PVOID GetPEB64()
{
   PVOID pPeb = 0;
#ifndef _WIN64
   // 1. There are two copies of PEB - PEB64 and PEB32 in WOW64 process
   // 2. PEB64 follows after PEB32
   // 3. This is true for versions lower than Windows 8, else __readfsdword returns address of real PEB64
   if (IsWin8OrHigher())
   {
       BOOL isWow64 = FALSE;
       typedef BOOL(WINAPI *pfnIsWow64Process)(HANDLE hProcess, PBOOL isWow64);
       pfnIsWow64Process fnIsWow64Process = (pfnIsWow64Process)
           GetProcAddress(GetModuleHandleA("Kernel32.dll"), "IsWow64Process");
       if (fnIsWow64Process(GetCurrentProcess(), &isWow64))
       {
           if (isWow64)
           {
               pPeb = (PVOID)__readfsdword(0x0C * sizeof(PVOID));
               pPeb = (PVOID)((PBYTE)pPeb + 0x1000);
           }
       }
   }
#endif
   return pPeb;
}
```

用于检查操作系统版本的函数的代码如下：

```
WORD GetVersionWord()
{
   OSVERSIONINFO verInfo = { sizeof(OSVERSIONINFO) };
   GetVersionEx(&verInfo);
   return MAKEWORD(verInfo.dwMinorVersion, verInfo.dwMajorVersion);
}
BOOL IsWin8OrHigher() { return GetVersionWord() >= _WIN32_WINNT_WIN8; }
BOOL IsVistaOrHigher() { return GetVersionWord() >= _WIN32_WINNT_VISTA; }
```

## 如何绕过IsDebuggerPresent检查

若要绕过IsDebuggerPresent检查，请在执行检查代码之前将BeingDebugged设置为0。DLL注入可以用来做到这一点:

```
mov eax, dword ptr fs:[0x30]
mov byte ptr ds:[eax+2], 0
```

对于x64进程:

```
DWORD64 dwpeb = __readgsqword(0x60);
*((PBYTE)(dwpeb + 2)) = 0;
```

## TLS回调

检查主函数中是否存在调试器并不是最好的主意，因为在查看反汇编程序列表时，逆向人员首先会查看这个位置。
在主函数中实现的检查可以通过NOP指令删除，从而解除保护。如果使用CRT库，主线程在将控制转移到主函数之前将已经有一个特定的调用堆栈。因此，执行调试器状态检

```
#pragma section(".CRT$XLY", long, read)
__declspec(thread) int var = 0xDEADBEEF;
VOID NTAnopPI TlsCallback(PVOID DllHandle, DWORD Reason, VOID Reserved)
{
```

```
   var = 0xB15BADB0; // Required for TLS Callback call
   if (IsDebuggerPresent())
   {
       MessageBoxA(NULL, "Stop debugging program!", "Error", MB_OK | MB_ICONERROR);
       TerminateProcess(GetCurrentProcess(), 0xBABEFACE);
   }
}
__declspec(allocate(".CRT$XLY"))PIMAGE_TLS_CALLBACK g_tlsCallback = TlsCallback;
```

## NtGlobalFlag

在WindowsNT中，有一组标志存储在全局变量NtGlobalFlag中，这在整个系统中是常见的。启动时，将使用系统注册表项中的值初始化NtGlobalFlag全局系统变量：

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\GlobalFlag]
```

此变量值用于系统跟踪、调试和控制。变量标志未被文档化，但是SDK包含gflags实用程序，它允许您编辑全局标志值。PEB结构还包括NtGlobalFlag字段，它的位结构不对

```
FLG_HEAP_ENABLE_TAIL_CHECK (0x10)
FLG_HEAP_ENABLE_FREE_CHECK (0x20)
FLG_HEAP_VALIDATE_PARAMETERS (0x40)
```

若要检查是否已使用调试器启动进程，请检查PEB结构中NtGlobalFlag字段的值。在x32和x64系统中,该字段位于PEB结构的开始处的0x068和0x0bc偏移处。

```
0:000> dt _PEB NtGlobalFlag @$peb
ntdll!_PEB
  +0x068 NtGlobalFlag : 0x70
```

对于64位进程：

```
0:000> dt _PEB NtGlobalFlag @$peb
ntdll!_PEB
  +0x0bc NtGlobalFlag : 0x70
```

以下代码片段是基于NtGlobalFlag标志检查的反调试保护示例：

```
#define FLG_HEAP_ENABLE_TAIL_CHECK   0x10
#define FLG_HEAP_ENABLE_FREE_CHECK   0x20
#define FLG_HEAP_VALIDATE_PARAMETERS 0x40
#define NT_GLOBAL_FLAG_DEBUGGED (FLG_HEAP_ENABLE_TAIL_CHECK | FLG_HEAP_ENABLE_FREE_CHECK | FLG_HEAP_VALIDATE_PARAMETERS)
void CheckNtGlobalFlag()
{
   PVOID pPeb = GetPEB();
   PVOID pPeb64 = GetPEB64();
   DWORD offsetNtGlobalFlag = 0;
#ifdef _WIN64
   offsetNtGlobalFlag = 0xBC;
#else
   offsetNtGlobalFlag = 0x68;
#endif
   DWORD NtGlobalFlag = *(PDWORD)((PBYTE)pPeb + offsetNtGlobalFlag);
   if (NtGlobalFlag & NT_GLOBAL_FLAG_DEBUGGED)
   {
       std::cout << "Stop debugging program!" << std::endl;
       exit(-1);
   }
   if (pPeb64)
   {
       DWORD NtGlobalFlagWow64 = *(PDWORD)((PBYTE)pPeb64 + 0xBC);
       if (NtGlobalFlagWow64 & NT_GLOBAL_FLAG_DEBUGGED)
       {
           std::cout << "Stop debugging program!" << std::endl;
           exit(-1);
       }
   }
}
```

## 如何绕过NtGlobalFlag检查

若要绕过NtGlobalFlag检查，只需执行逆向检查之前执行的操作；换句话说，在通过反调试保护检查此值之前，将已调试进程的PEB结构的NtGlobalFlag字段设置为0。

# NtGlobalFlag和IMAGE_LOAD_CONFIG_DIRECTORY

可执行文件包含IMAGE_LOAD_DB2IGDIRECTY结构，该结构包含系统加载程序的其他配置参数。默认情况下，此结构不会内置到可执行文件中，但可以使用修补程序将其

```
PIMAGE_NT_HEADERS GetImageNtHeaders(PBYTE pImageBase)
{
   PIMAGE_DOS_HEADER pImageDosHeader = (PIMAGE_DOS_HEADER)pImageBase;
   return (PIMAGE_NT_HEADERS)(pImageBase + pImageDosHeader->e_lfanew);
}
PIMAGE_SECTION_HEADER FindRDataSection(PBYTE pImageBase)
{
   static const std::string rdata = ".rdata";
   PIMAGE_NT_HEADERS pImageNtHeaders = GetImageNtHeaders(pImageBase);
   PIMAGE_SECTION_HEADER pImageSectionHeader = IMAGE_FIRST_SECTION(pImageNtHeaders);
   int n = 0;
   for (; n < pImageNtHeaders->FileHeader.NumberOfSections; ++n)
   {
       if (rdata == (char*)pImageSectionHeader[n].Name)
       {
           break;
       }
   }
   return &pImageSectionHeader[n];
}
void CheckGlobalFlagsClearInProcess()
{
   PBYTE pImageBase = (PBYTE)GetModuleHandle(NULL);
   PIMAGE_NT_HEADERS pImageNtHeaders = GetImageNtHeaders(pImageBase);
   PIMAGE_LOAD_CONFIG_DIRECTORY pImageLoadConfigDirectory = (PIMAGE_LOAD_CONFIG_DIRECTORY)(pImageBase
       + pImageNtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG].VirtualAddress);
   if (pImageLoadConfigDirectory->GlobalFlagsClear != 0)
   {
       std::cout << "Stop debugging program!" << std::endl;
       exit(-1);
   }
}
void CheckGlobalFlagsClearInFile()
{
   HANDLE hExecutable = INVALID_HANDLE_VALUE;
   HANDLE hExecutableMapping = NULL;
   PBYTE pMappedImageBase = NULL;
   __try
   {
       PBYTE pImageBase = (PBYTE)GetModuleHandle(NULL);
       PIMAGE_SECTION_HEADER pImageSectionHeader = FindRDataSection(pImageBase);
       TCHAR pszExecutablePath[MAX_PATH];
       DWORD dwPathLength = GetModuleFileName(NULL, pszExecutablePath, MAX_PATH);
       if (0 == dwPathLength) __leave;
       hExecutable = CreateFile(pszExecutablePath, GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
       if (INVALID_HANDLE_VALUE == hExecutable) __leave;
       hExecutableMapping = CreateFileMapping(hExecutable, NULL, PAGE_READONLY, 0, 0, NULL);
       if (NULL == hExecutableMapping) __leave;
       pMappedImageBase = (PBYTE)MapViewOfFile(hExecutableMapping, FILE_MAP_READ, 0, 0,
           pImageSectionHeader->PointerToRawData + pImageSectionHeader->SizeOfRawData);
       if (NULL == pMappedImageBase) __leave;
       PIMAGE_NT_HEADERS pImageNtHeaders = GetImageNtHeaders(pMappedImageBase);
       PIMAGE_LOAD_CONFIG_DIRECTORY pImageLoadConfigDirectory = (PIMAGE_LOAD_CONFIG_DIRECTORY)(pMappedImageBase
           + (pImageSectionHeader->PointerToRawData
               + (pImageNtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG].VirtualAddress - pImageSect
       if (pImageLoadConfigDirectory->GlobalFlagsClear != 0)
       {
           std::cout << "Stop debugging program!" << std::endl;
           exit(-1);
       }
   }
   __finally
   {
       if (NULL != pMappedImageBase)
           UnmapViewOfFile(pMappedImageBase);
```

```
        if (NULL != hExecutableMapping)
            CloseHandle(hExecutableMapping);
        if (INVALID_HANDLE_VALUE != hExecutable)
            CloseHandle(hExecutable);
    }
}
```

在此代码示例中，CheckGlobalFlagsClearInProcess函数根据当前运行的进程的加载地址查找PIMAGE_LOAD_DB2IGDIRECTORY结构，并检查GlobalFlagsClear字段的值
CheckGlobalFlagsClearInFile函数对磁盘上的可执行文件执行相同的检查。

## Heap Flags and ForceFlags

PEB结构包含指向进程堆(_heap结构)的指针：

```
0:000> dt _PEB ProcessHeap @$peb
ntdll!_PEB
  +0x018 ProcessHeap : 0x00440000 Void
0:000> dt _HEAP Flags ForceFlags 00440000
ntdll!_HEAP
  +0x040 Flags      : 0x40000062
  +0x044 ForceFlags : 0x40000060
```

对于64位进程

```
0:000> dt _PEB ProcessHeap @$peb
ntdll!_PEB
  +0x030 ProcessHeap : 0x0000009d`94b60000 Void
0:000> dt _HEAP Flags ForceFlags 0000009d`94b60000
ntdll!_HEAP
  +0x070 Flags      : 0x40000062
  +0x074 ForceFlags : 0x40000060
```

如果正在调试进程，则两个字段Flags和ForceFlags都具有特定的调试值：
1.如果Flags字段没有设置HEAP_GROWABLE（0x00000002）标识，则正在调试进程。

2.如果ForceFlags！= 0，则正在调试进程。

不过要注意的是，_HEAP结构并未记录，并且Flags和ForceFlags字段的偏移值可能因操作系统版本而异。以下代码就是基于HeapFlag检查的反调试保护：

```
int GetHeapFlagsOffset(bool x64)
{
    return x64 ?
        IsVistaOrHigher() ? 0x70 : 0x14: //x64 offsets
        IsVistaOrHigher() ? 0x40 : 0x0C; //x86 offsets
}
int GetHeapForceFlagsOffset(bool x64)
{
    return x64 ?
        IsVistaOrHigher() ? 0x74 : 0x18: //x64 offsets
        IsVistaOrHigher() ? 0x44 : 0x10; //x86 offsets
}
void CheckHeap()
{
    PVOID pPeb = GetPEB();
    PVOID pPeb64 = GetPEB64();
    PVOID heap = 0;
    DWORD offsetProcessHeap = 0;
    PDWORD heapFlagsPtr = 0, heapForceFlagsPtr = 0;
    BOOL x64 = FALSE;
#ifdef _WIN64
    x64 = TRUE;
    offsetProcessHeap = 0x30;
#else
    offsetProcessHeap = 0x18;
#endif
    heap = (PVOID)*(PDWORD_PTR)((PBYTE)pPeb + offsetProcessHeap);
    heapFlagsPtr = (PDWORD)((PBYTE)heap + GetHeapFlagsOffset(x64));
    heapForceFlagsPtr = (PDWORD)((PBYTE)heap + GetHeapForceFlagsOffset(x64));
    if (*heapFlagsPtr & ~HEAP_GROWABLE || *heapForceFlagsPtr != 0)
    {
```

```
        std::cout << "Stop debugging program!" << std::endl;
        exit(-1);
    }
    if (pPeb64)
    {
        heap = (PVOID)*(PDWORD_PTR)((PBYTE)pPeb64 + 0x30);
        heapFlagsPtr = (PDWORD)((PBYTE)heap + GetHeapFlagsOffset(true));
        heapForceFlagsPtr = (PDWORD)((PBYTE)heap + GetHeapForceFlagsOffset(true));
        if (*heapFlagsPtr & ~HEAP_GROWABLE || *heapForceFlagsPtr != 0)
        {
            std::cout << "Stop debugging program!" << std::endl;
            exit(-1);
        }
    }
}
```

## 如何绕过Heap Flags和ForceFlags检查

为了避开基于HeapFlag检查的反调试保护，应该为Flags字段设置HEAP_GROWABLE标识，然后将ForceFlags的值设置为0。值得一提的是，字段值的重新定义应该在Heap

未完待续

```

点击收藏 | 1 关注 | 1

上一篇：CVE-2017-11176: 一... 下一篇：weblogic_2019_272...

1. 0 条回复
   - 动动手指，沙发就是你的了！
```