南京郵電大學
Nanjing University of Posts and Telecommunications

反调试技术简介

本次课程支撑的毕业要求指标点

- 毕业要求4-3：

针对设计或开发的解决方案，能够基于信息安全领域科学原理对其进行研究，并能够通过理论证明、实验仿真或者系统实现等多种科学方案说明其有效性、合理性，并对解决方案的实施质量进行分析，通过信息综合得到合理有效的结论。

反调试简介 - 综述

反调试分类：

1. 调试器检测：各种方法查看调试器是否存在
2. 识别调试器：识别是否在调试中
3. 干扰调试器：令调试失败

调试器检测 -- Windows API

- **1.1 IsDebuggerPresent:**

IsDebuggerPresent查询进程环境块(PEB)中的IsDebugged标志。如果进程没有运行在调试器环境中，函数返回0；如果调试附加了进程，函数返回一个非零值。

```
1  BOOL CheckDebug()
2  {
3      return IsDebuggerPresent();
4  }
```

调试器检测 -- Windows API

- **1.2 CheckRemoteDebuggerPresent**

  CheckRemoteDebuggerPresent同IsDebuggerPresent几乎一致。它不仅可以探测系统其他进程是否被调试，通过传递自身进程句柄还可以探测自身是否被调试。

```
1  BOOL CheckDebug()
2  {
3      BOOL ret;
4      CheckRemoteDebuggerPresent(GetCurrentProcess(), &ret);
5      return ret;
6  }
```

调试器检测 -- Windows API

- **1.3 NtQueryInformationProcess**

  第二个参数是一个枚举类型，其中与反调试有关的成员有：

  ProcessDebugPort(0x7)、ProcessDebugObjectHandle(0x1E)

  和ProcessDebugFlags(0x1F)。若将该参数置为ProcessDebugPort，如果进程正在被调试，则返回调试端口，否则返回0

  注意，这里的NtQueryInformationProcess是通过函数名引出，不能直接调用。

```
BOOL CheckDebug()
{
    int debugPort = 0;
    HMODULE hModule = LoadLibrary("Ntdll.dll");
    NtQueryInformationProcessPtr NtQueryInformationProcess = (NtQueryInformationProcessPtr)GetProcAddress(hModule, "NtQueryInformationProc
ess");
    NtQueryInformationProcess(GetCurrentProcess(), 0x7, &debugPort, sizeof(debugPort), NULL);
    return debugPort != 0;
}
```

调试器检测 – 手动检测数据结构

- **2.1 BeingDebugged属性**

Windows操作系统维护着每个正在运行的进程的PEB结构，它包含与这个进程相关的所有用户态参数。这些参数包括进程环境数据，环境数据包括环境变量、加载的模块列表、内存地址，以及调试器状态

FS指向当前的TEB结构，偏移0x30处是ProcessEnvironmentBlock域，指向PEB结构体

```
typedef struct _PEB {
  BYTE Reserved1[2];
  BYTE BeingDebugged;
  BYTE Reserved2[1];
  PVOID Reserved3[2];
  PPEB_LDR_DATA Ldr;
```

```
1  BOOL CheckDebug()
2  {
3      int result = 0;
4      __asm
5      {
6          mov eax, fs:[30h]
7          mov al, BYTE PTR [eax + 2]
8          mov result, al
9      }
10     return result != 0;
11 }
```

调试器检测 – 手动检测数据结构

- **2.2 ProcessHeap属性**

ProcessHeap位于PEB结构的0x18处。第一个堆头部有一个属性字段，它告诉内核这个堆是否在调试器中创建。这些属性叫作ForceFlags。在Windows 7或更高版本的系统中，对于32位的应用程序来说ForceFlags属性位于堆头部偏移量0x44处。低版本的系统中，偏移量为0x10。

```
BOOL CheckDebug()
{
    int result = 0;
    DWORD dwVersion = GetVersion();
    DWORD dwWindowsMajorVersion = (DWORD)(LOBYTE(LOWORD(dwVersion)));
    //for xp
    if (dwWindowsMajorVersion == 5)
    {
        __asm
        {
            mov eax, fs:[30h]
            mov eax, [eax + 18h]
            mov eax, [eax + 10h]
            mov result, eax
        }
    }
    else
    {
        __asm
        {
            mov eax, fs:[30h]
            mov eax, [eax + 18h]
            mov eax, [eax + 44h]
            mov result, eax
        }
    }
    return result != 0;
```

调试器检测 – 手动检测数据结构

- **2.3 NTGlobalFlag**

由于调试器中启动进程与正常模式下启动进程有些不同，所以它们创建内存堆的方式也不同。系统使用PEB结构偏移量0x68处的一个未公开位置，来决定如何创建堆结构。如果这个位置的值为0x70，我们就知道进程正运行在调试器中。

```
BOOL CheckDebug()
{
    int result = 0;
    __asm
    {
        mov eax, fs:[30h]
        mov eax, [eax + 68h]
        and eax, 0x70
        mov result, eax
    }
    return result != 0;
}
```

调试器检测 – 系统痕迹检测

- 3.1 查找调试器引用的注册表项

SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug(32位系统)
SOFTWARE\Wow6432Node\Microsoft\WindowsNT\CurrentVersion\AeDebug(64位系统)

该注册表项指定当应用程序发生错误时，触发**哪一个调试器**。默认情况下，它被设置为Dr.Watson。如果该这册表的键值被修改为OllyDbg（**或其他非默认值**），则恶意代码可确定它**正在被调试**。

```
    }
    char tmp[256];
    DWORD len = 256;
    DWORD type;
    ret = RegQueryValueExA(hkey, key, NULL, &type, (LPBYTE)tmp, &len);
    if (strstr(tmp, "OllyIce")!=NULL || strstr(tmp, "OllyDBG")!=NULL || strstr(tmp, "WinDbg")!=NULL || strstr(tmp, "x64dbg")!=NULL || strs
tr(tmp, "Immunity")!=NULL)
    {
        return TRUE;
    }
    else
    {
        return FALSE;
```

调试器检测 – 系统痕迹检测

- 3.2 查找窗体信息

FindWindow函数检索处理顶级窗口的类名和窗口名称匹配指定的字符串

```
BOOL CheckDebug()
{
    if (FindWindowA("OLLYDBG", NULL)!=NULL || FindWindowA("WinDbgFrameClass", NULL)!=NULL || FindWindowA("QWidget", NULL)!=NULL)
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}
```

```
BOOL CheckDebug()
{
    char fore_window[1024];
    GetWindowTextA(GetForegroundWindow(), fore_window, 1023);
    if (strstr(fore_window, "WinDbg")!=NULL || strstr(fore_window, "x64_dbg")!=NULL || strstr(fore_window, "OllyICE")!=NULL || strstr(fore
_window, "OllyDBG")!=NULL || strstr(fore_window, "Immunity")!=NULL)
    {
        return TRUE;
```

识别调试器 – 检测断点

- **4.1 检测软件断点**
  - 调试器设置断点的基本机制是用软件中断指令INT 3临时替换运行程序中的一条指令，然后当程序运行到这条指令时，调用调试异常处理例程。
  - INT 3指令的机器码是0xCC，因此无论何时，使用调试器设置一个断点，它都会插入一个0xCC来修改代码。
  - 常用的一种反调试技术是在它的代码中查找机器码0xCC，来扫描调试器对它代码的INT 3修改。

识别调试器 – 检测断点

```
BOOL CheckDebug()
{
    PIMAGE_DOS_HEADER pDosHeader;
    PIMAGE_NT_HEADERS32 pNtHeaders;
    PIMAGE_SECTION_HEADER pSectionHeader;
    DWORD dwBaseImage = (DWORD)GetModuleHandle(NULL);
    pDosHeader = (PIMAGE_DOS_HEADER)dwBaseImage;
    pNtHeaders = (PIMAGE_NT_HEADERS32)((DWORD)pDosHeader + pDosHeader->e_lfanew);
    pSectionHeader = (PIMAGE_SECTION_HEADER)((DWORD)pNtHeaders + sizeof(pNtHeaders->Signature) + sizeof(IMAGE_FILE_HEADER) +
                     (WORD)pNtHeaders->FileHeader.SizeOfOptionalHeader);
    DWORD dwAddr = pSectionHeader->VirtualAddress + dwBaseImage;
    DWORD dwCodeSize = pSectionHeader->SizeOfRawData;
    BOOL Found = FALSE;
    __asm
    {
        cld
        mov     edi,dwAddr
        mov     ecx,dwCodeSize
        mov     al,0CCH
        repne   scasb
        jnz     NotFound
        mov Found,1
NotFound:
    }
    return Found;
}
```

识别调试器－检测断点

- **4.2 检测硬件断点**

OllyDbg的寄存器窗口按下右键，点击View debug registers可以看到DR0、DR1、DR2、DR3、DR6和DR7这几个寄存器。DR0、DR1、DR2、DR3用于设置硬件断点，由于只有4个硬件断点寄存器，所以同时最多只能设置4个硬件断点。DR4、DR5由系统保留。如果没有硬件断点，那么DR0、DR1、DR2、DR3这4个寄存器的值都为0。·

```
BOOL CheckDebug()
{
    CONTEXT context;
    HANDLE hThread = GetCurrentThread();
    context.ContextFlags = CONTEXT_DEBUG_REGISTERS;
    GetThreadContext(hThread, &context);
    if (context.Dr0 != 0 || context.Dr1 != 0 || context.Dr2 != 0 || context.Dr3!=0)
    {
        return TRUE;
    }
    return FALSE;
}
```

识别调试器 – 时钟检测

- **4.3 时钟检测**
  - 被调试时，进程的运行速度大大降低。例如，单步调试会大幅降低程序的运行速度，所以时钟检测是检测调试器存在的最常用方式之一。
  - 原理：记录一段操作前后的时间戳，然后比较两个时间戳，如果存在滞后，则可以认为存在调试器。
  - 常用的时钟检测方法是利用rdtsc指令(操作码0x0F31)，它返回一个系统重新启动以来的时钟数，并且将其作为一个64位的值存入EAX中。运行两次rdtsc指令，然后比较两次读取之间的差值 ·

```
BOOL CheckDebug()
{
    DWORD time1, time2;
    __asm
    {
        rdtsc
        mov time1, eax
        rdtsc
        mov time2, eax
    }
    if (time2 - time1 < 0xff)
    {
        return FALSE;
    }
    else
    {
        return TRUE;
    }
}
```

识别调试器 – 父进程判断

- **4.4 父进程判定**

一般双击运行的进程的父进程都是**explorer.exe**，但是如果进程被调试，父进程则是调试器进程。也就是说如果父进程不是explorer.exe则可以认为程序正在被调试。explorer.exe创建进程的时候会把**STARTUPINFO**结构中的值设为0，而非explorer.exe创建进程的时候会忽略这个结构中的值，也就是结构中的值不为0。所以可以利用STARTUPINFO来判断程序是否在被调试。

```
BOOL CheckDebug()
{
    STARTUPINFO si;
    GetStartupInfo(&si);
    if (si.dwX!=0 || si.dwY!=0 || si.dwFillAttribute!=0 || si.dwXSize!=0 || si.dwYSize!=0 || si.dwXCountChars!=0 || si.dwYCountChars!=0)
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}
```

干扰调试器 – TLS回调

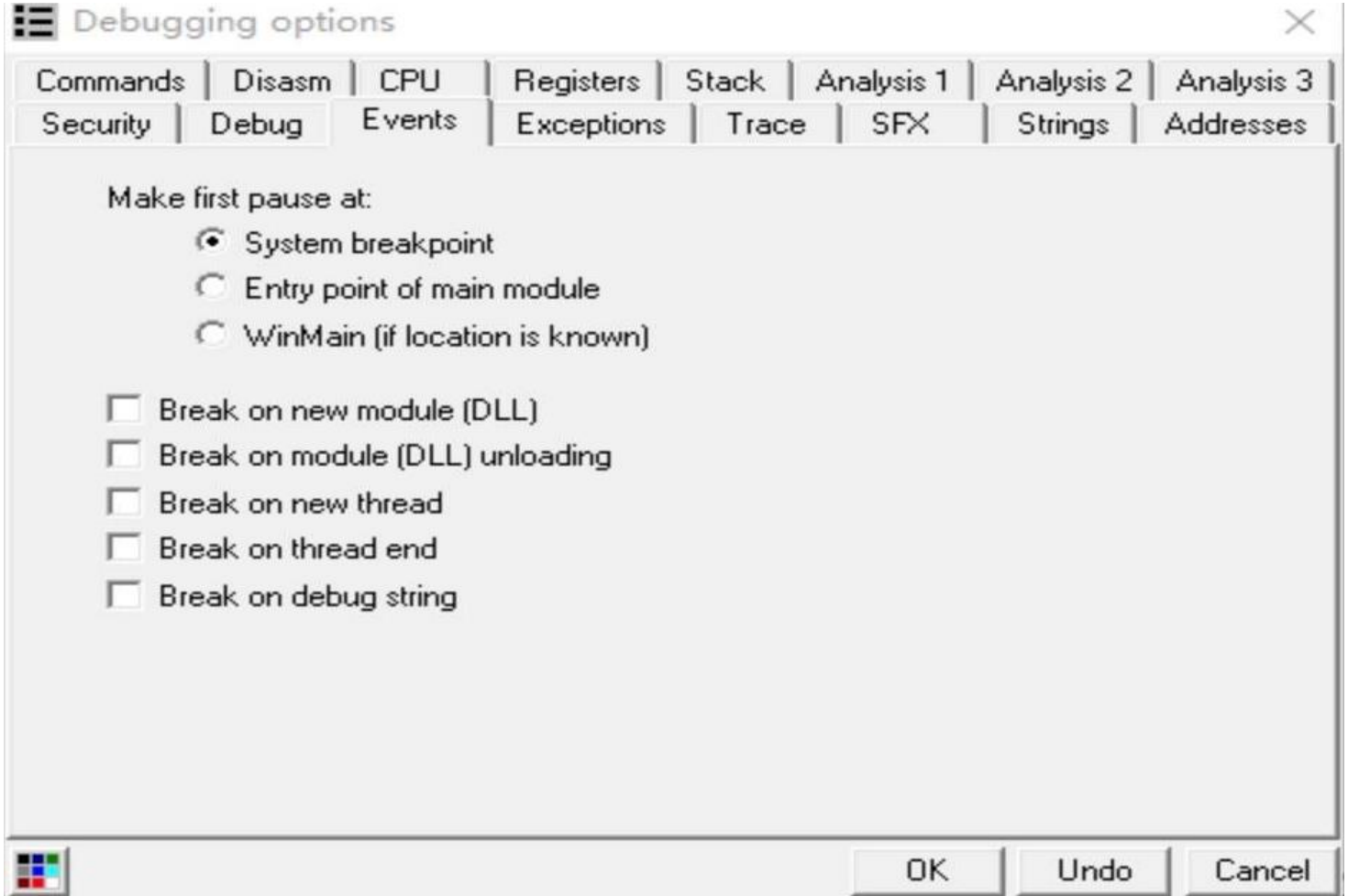
- **5.1 TLS回调**

  Thread Local Storage(TLS)，即线程本地存储，是Windows为解决一个进程中多个线程同时访问全局变量而提供的机制。TLS回调函数在进程创建时被调用，会早于进程的Main入口函数，利用这一特性进行反调试。

```
void NTAPI __stdcall TLS_CALLBACK1(PVOID DllHandle, DWORD Reason, PVOID Reserved)
{
    if (IsDebuggerPresent())
    {
        printf("TLS_CALLBACK: Debugger Detected!\n");
    }
    else
    {
        printf("TLS_CALLBACK: No Debugger Present!\n");
    }
}

int main(int argc, char* argv[])
{
    printf("233\n");
    return 0;
}
```

干扰调试器 – TLS回调


Debugging options
Commands
Disasm
CPU
Registers
Stack
Analysis 1
Analysis 2
Analysis 3
Security
Debug
Events
Exceptions
Trace
SFX
Strings
Addresses
Make first pause at:
System breakpoint
Entry point of main module
WinMain (if location is known)
Break on new module (DLL)
Break on module (DLL) unloading
Break on new thread
Break on thread end
Break on debug string
OK
Undo
Cancel

干扰调试器 – 利用中断

- **5.2 利用中断**
  - 双字节操作码0xCD03也可以产生INT 3中断，这是干扰WinDbg调试器的有效方法。在调试器外，0xCD03指令产生一个STATUS_BREAKPOINT异常。
  - 然而在WinDbg调试器内，由于断点通常是单字节机器码0xCC，因此WinDbg会捕获这个断点然后将EIP加1字节。这可能导致程序在被正常运行的WinDbg调试时，执行不同的指令集。

```
BOOL CheckDebug()
{
    __try
    {
        __asm
        {
            __emit 0xCD
            __emit 0x03
        }
    }
    __except(1)
    {
        return FALSE;
    }
    return TRUE;
}
```

干扰调试器 – 陷阱标志位

- **5.3 陷阱标志位**

EFLAGS寄存器的第八个比特位是陷阱标志位。如果设置了，就会产生一个单步异常。

.

```
BOOL CheckDebug()
{
    __try
    {
        __asm
        {
            pushfd
            or word ptr[esp], 0x100
            popfd
            nop
        }
    }
    __except(1)
    {
        return FALSE;
    }
    return TRUE;
}
```

干扰调试器 – 陷阱标志位

- **5.4 RaiseException函数**

  RaiseException函数产生的若干不同类型的异常可以被调试器捕获。只有当没有附加调试器的时候，异常处理程序才会执行。

```
BOOL TestExceptionCode(DWORD dwCode)
{
    __try
    {
        RaiseException(dwCode, 0, 0, 0);
    }
    __except(1)
    {
        return FALSE;
    }
    return TRUE;
}

BOOL CheckDebug()
{
    return TestExceptionCode(DBG_RIPEXCEPTION);
}
```

感谢大家！

南京郵電大學
Nanjing University of Posts and Telecommunications