北京郵電大學
Beijing University of Posts and Telecommunications

Merak
hacking securily

汇编语言与逆向工程
Assembly Language and Reverse Engineering

北京邮电大学
付俊松

2024年6月

第十一章 反调试技术

- 一. 反调试技术简述
- 二. 代码调试状态检测常用方法
- 三. 代码扰乱调试器常用方法

第十一章 反调试技术

（1）反调试技术简述

- 反调试技术，即软件作者防止其他人对软件进行调试的技术，是一种软件安全保护技术
  - 代码编写者发现分析人员经常使用调试器来观察代码的内部结构和工作流程，反调试技术可以尽可能地延长代码的分析时间
  - 当代码意识到自己被调试时，可以通过改变自己的正常执行路径、修改自身导致崩溃等操作，增加调试时间和复杂度

（1）反调试技术简述

- 反反调试技术
  - 在代码运行期间动态修改代码，使其不能调用反调试检测功能
  - 修改调试检测函数的返回值，确保代码执行合适的路径

第十一章 反调试技术

- 一. 反调试技术简述
- 二. 代码调试状态检测常用方法
- 三. 代码扰乱调试器常用方法

- 代码调试状态检测常用方法
  - 标志位检测
  - 进程检测
  - 窗口检测
  - 调试器断点检测
  - 代码校验和
  - 时钟检测
  - 双进程反调试

（1）标志位检测

- 相关标志位提取
  - PEB
    - PEB（Process Environment Block，进程环境块）存放进程信息，每个进程都有自己的PEB信息，位于用户地址空间
  - NtQueryInformationProcess
    - 该函数能够获得各种与进程相关的信息，它可以将指定类型的进程信息拷贝到某个缓冲区中

（1）标志位检测

- 可检测标志位
  - PEB：
    - BeingDebugged
    - Flags
    - Force Flags
    - NtGlobalFlag
  - ProcessInformation：
    - ProcessDebugPort
    - ProcessDebugObjectHandle
    - ProcessDebugFlags

（1）标志位检测

- 如何获取PEB
  - PEB的地址可由TEB结构的偏移0x30处获取，FS段寄存器指向当前的TEB结构，故由FS:[0x30]可以获取到PEB的地址

（1）标志位检测

- PEB结构

```
typedef struct _PEB {
    BOOLEAN                 InheritedAddressSpace;
    BOOLEAN                 ReadImageFileExecOptions;
    BOOLEAN                 BeingDebugged;              //+0x002
    BOOLEAN                 Spare;
    HANDLE                  Mutant;
    PVOID                   ImageBaseAddress;
    PPEB_LDR_DATA           LoaderData;
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
    PVOID                   SubSystemData;
    PVOID                   ProcessHeap;                //+0x018
    PVOID                   FastPebLock;
    ......
    ULONG                   NumberOfProcessors;
    ULONG                   NtGlobalFlag;               //+0x068
    BYTE                    Spare2[0x4];
    ......
} PEB, *PPEB;
```

（1）标志位检测

- BeingDebugged
  - PEB.BeingDebugged成员用来判断进程是否正在调试，若是则为1，反之为0
    - Windows提供了API：IsDebuggerPresent() 获得BeingDebugged成员的状态，根据其值可用于判断进程是否正在被调试
    - BeingDebugged在PEB中的偏移是0x2，通过获取PEB的地址，再获得其偏移为0x2的成员变量的值即可判断进程是否正在调试

（1）标志位检测

- BeingDebugged

```
/*BeingDebugged*/
BOOL (*IsDebuggerPresent)();

void CheckDebug1()
{
    printf("CheckDebug1:\n");

    HMODULE hModule = GetModuleHandle("kernel32.dll");
    //获得IsDebuggerPresent函数的地址
    IsDebuggerPresent = (BOOL(*)())GetProcAddress(hModule,"
        IsDebuggerPresent");
    BOOL BeingDebugged = FALSE;

    BeingDebugged = IsDebuggerPresent();

    printf("\tIsDebuggerPresent() = %d\n", BeingDebugged);
    printf("\t%s\n",BeingDebugged == TRUE ? "Debug" : "Not debug");
}
```

（1）标志位检测

- BeingDebugged

```
/*asm_BeingDebugged*/
void CheckDebug2()
{
    printf("CheckDebug2:\n");

    char result = 0;

    __asm
    {
        mov eax, fs:[30h]
        mov al, BYTE PTR [eax + 2]
        mov result, al
    }

    printf("\t%s\n",result == 1 ? "Debug" : "Not debug");
}
```

（1）标志位检测

- ProcessHeap
  - PEB.ProcessHeap是指向Heap结构体的指针，Heap结构体中Flags与Force Flags成员在被调试时的值与正常运行时不同，可以据此判断该程序是否正在被调试
    - ProcessHeap结构体由PEB偏移0x18得到
    - Flags在ProcessHeap结构体中偏移为0x0c，在非调试状态下值为0x2
    - Force Flags在ProcessHeap结构体中偏移为0x10，在非调试状态下值为0x0

（1）标志位检测

- ProcessHeap

```
/*ProcessHeap*/
void CheckDebug3()
{
    printf("CheckDebug3:\n");

    LPBYTE pTEB = NULL;
    LPBYTE pPEB = NULL;
    FARPROC pProc = NULL;

    pProc = GetProcAddress(GetModuleHandle("ntdll.dll"), "NtCurrentTeb");
    pTEB = (LPBYTE)(*pProc)();                  //address of TEB
    pPEB = (LPBYTE)*(LPDWORD)(pTEB + 0x30);     //address of PEB

    //address of ProcessHeap
    LPBYTE pHeap = (LPBYTE)*(LPDWORD)(pPEB + 0x18);

    DWORD dwFlags = *(LPDWORD)(pHeap + 0xC);       //ProcessHeap_Flags
    DWORD dwForceFlags = *(LPDWORD)(pHeap + 0x10);//ProcessHeap_ForceFlags

    printf("\tProcessHeap_Flags = 0x%x\n", dwFlags);
    printf("\tProcessHeap_ForceFlags = 0x%x\n", dwForceFlags);
    printf("\t%s\n",(dwFlags != 0x2 || dwForceFlags != 0x0)
                        ? "Debug" : "Not debug");
}
```

（1）标志位检测

NtGlobalFlag

- NtGlobalFlag位于PEB偏移0x68处，在调试状态下值为0x70

```
void CheckDebug4()
{
    printf("CheckDebug4:\n");

    LPBYTE pTEB = NULL;
    LPBYTE pPEB = NULL;
    FARPROC pProc = NULL;

    pProc = GetProcAddress(GetModuleHandle("ntdll.dll"), "NtCurrentTeb");
    pTEB = (LPBYTE)(*pProc)();                    //address of TEB
    pPEB = (LPBYTE)*(LPDWORD)(pTEB + 0x30);       //address of PEB

    //address of NtGlobalFlag
    DWORD dwNtGlobalFlag = *(LPDWORD)(pPEB+0x68);

    printf("\tNtGlobalFlag = 0x%x\n", dwNtGlobalFlag);
    printf("\t%s\n",dwNtGlobalFlag == 0x70 ? "Debug" : "Not debug");
}
```

（1）标志位检测

- ProcessInformation
  - NtQueryInformationProcess函数能够获得各种与进程相关的信息，它可以将指定类型的进程信息拷贝到某个缓冲区中，其函数原型如下：

```
NTSTATUS WINAPI NtQueryInformationProcess(
  _In_      HANDLE           ProcessHandle,
  _In_      PROCESSINFOCLASS ProcessInformationClass,
  _Out_     PVOID            ProcessInformation,
  _In_      ULONG            ProcessInformationLength,
  _Out_opt_ PULONG           ReturnLength
);
```

（1）标志位检测

- ProcessInformation
  - NtQueryInformationProcess参数中的ProcessInformationClass为所需要获取的值，其为枚举类型

```
enum PROCESSINFOCLASS
{
    .....
    ProcessDebugPort = 7,
    .....
    ProcessDebugObjectHandle = 30,
    ProcessDebugFlags = 31,
    .....
};
```

（1）标志位检测

- ProcessInformation
  - 通过调用NtQueryInformationProcess获取ProcessDebugPort，ProcessDebugObjectHandle以及ProcessDebugFlags，并检查他们的值可以判断进程是否处于调试状态
    - ProcessDebugPort，程序在未被调试状态下值为0x0，被调试时值为0xFFFFFFFF
    - ProcessDebugObjectHandle，程序在未被调试状态下句柄为NULL
    - ProcessDebugFlags，程序在被调试状态下为0

（1）标志位检测

- ProcessInformation

```
void CheckDebug1()
{
    printf("CheckDebug1:\n");

    NTQUERYINFORMATIONPROCESS pNtQueryInformationProcess = NULL;

    pNtQueryInformationProcess = (NTQUERYINFORMATIONPROCESS)
                                 GetProcAddress(GetModuleHandle("ntdll.dll"
                                     ),
                                                "NtQueryInformationProcess"
                                                    );

    // ProcessDebugPort (0x7)
    DWORD DebugPort = 0;
    pNtQueryInformationProcess(GetCurrentProcess(),
                               ProcessDebugPort,
                               &DebugPort,
                               sizeof(DebugPort),
                               NULL);

    printf("\tProcessDebugPort = 0x%X\n", DebugPort);
    printf("\t%s\n",DebugPort != 0x0 ? "Debug" : "Not debug");
}
```

（1）标志位检测

- 破解方法
  - 修改相应标志位的值，将其改为未被调试时应有的值
  - Hook相应函数，使其返回的值为未被调试时应有的值

(2) 进程检测

- 检测方法
  - 检测调试器进程
  - 检测父进程

(2) 进程检测

- 检测调试器进程
  - 通过CreateToolhelp32Snapshot创建进程快照，通过遍历进程快照，若找到调试器进程就跳出循环
  - 该方法通常比较死板，需要指定进程名称，如常用的OllyDBG.EXE、OllyICE.exe、idaq.exe以及ImmunityDebugger.exe，当进程名称匹配时才会判定为找到调试器

（2）进程检测

- 检测调试器进程

```
//创建进程快照
HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
if(hProcessSnap == INVALID_HANDLE_VALUE)
{
    return ;
}

//遍历进程快照
BOOL More = Process32First(hProcessSnap, &pe32);
while(More)
{
    if(stricmp(pe32.szExeFile, "OllyDBG.EXE") == 0
    || stricmp(pe32.szExeFile, "OllyICE.exe") == 0
    || stricmp(pe32.szExeFile, "x64_dbg.exe") == 0
    || stricmp(pe32.szExeFile, "windbg.exe") == 0
    || stricmp(pe32.szExeFile, "ImmunityDebugger.exe") == 0)
    {
        ret = TRUE;
        break;
    }
    More = Process32Next(hProcessSnap, &pe32);
}
CloseHandle(hProcessSnap);
```

(2) 进程检测

- 检测父进程
  - 正常通过双击执行的进程的父进程是explorer.exe，若不是则可以认为程序被调试
  - 通过NtQueryInformationProcess，在第二个参数设置为ProcessBasicInformation时，ProcessBasicInformation的InheritedFromUniqueProcessId成员为父进程PID，通过枚举进程快照，通过PID得到进程名称，即可对比父进程名称是否为explorer.exe

（2）进程检测

- 检测父进程

```
pNtQueryInformationProcess = (NTQUERYINFORMATIONPROCESS)
                             GetProcAddress(GetModuleHandle("ntdll.dll"),
                                            "NtQueryInformationProcess");
pNtQueryInformationProcess(hProcess,
                           ProcessBasicInformation,
                           (PVOID)&pbi,
                           sizeof(PROCESS_BASIC_INFORMATION),
                           NULL);

PROCESSENTRY32 pe32;
pe32.dwSize = sizeof(pe32);

//创建进程快照
HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
if(hProcessSnap == INVALID_HANDLE_VALUE)
{
    return ;
}

// 遍历进程快照
BOOL More = Process32First(hProcessSnap, &pe32);
while(More)
{
    if (pbi.InheritedFromUniqueProcessId == pe32.th32ProcessID)
    {
        CloseHandle(hProcessSnap);
        printf("\t%s\n", stricmp(pe32.szExeFile, "explorer.exe") != 0 ? "Debug" : "Not debug");
    }
    More = Process32Next(hProcessSnap, &pe32);
}
CloseHandle(hProcessSnap);
```

(2) 进程检测

- 破解方法
  - Hook进程检测函数，使该种检测方法失效

(3) 窗口检测

- 检测方法
  - 检测窗口类名
  - 检测窗口标题

(3) 窗口检测

- 检测窗口类名
  - FindWindow()函数通过创建窗口时的类名和窗口名查找窗口并返回窗口的句柄，函数不会搜索子窗口，通过查找与调试器相关的类名就可以判断程序是否正在被调试

(3) 窗口检测

- 检测窗口类名

```
void CheckDebug1()
{
    printf("CheckDebug1:\n");

    if(FindWindowA("OLLYDBG", NULL) != NULL
    || FindWindowA("WinDbgFrameClass", NULL) != NULL
    || FindWindowA("QWidget", NULL) != NULL)
    {
        printf("\tDebug\n");
    }
    else
    {
        printf("\tNot debug\n");
    }
}
```

(3) 窗口检测

- 检测窗口标题
  - 通过EnumWindows或者GetForegroundWindow获取前台窗口句柄，由句柄通过GetWindowText得到窗口的标题文本，对其字符串进行匹配来确定其是否为调试器

(3) 窗口检测

- 检测窗口标题

```
BOOL CALLBACK EnumWndProc(HWND hwnd, LPARAM lParam)
{
    char cur_window[1024];
    GetWindowTextA(hwnd, cur_window, 1023);

    if(strstr(cur_window, "WinDbg") != NULL
    || strstr(cur_window, "x64_dbg") != NULL
    || strstr(cur_window, "OllyICE") != NULL
    || strstr(cur_window, "OllyDBG") != NULL
    || strstr(cur_window, "Immunity") != NULL)
    {
        *((BOOL*)lParam) = TRUE;
    }
    return true;
}

void CheckDebug2()
{
    printf("CheckDebug2:\n");
    BOOL ret = FALSE;

    EnumWindows(EnumWndProc, (LPARAM)&ret);

    printf("\t%s\n", ret ? "Debug" : "Not debug");
}
```

（3）窗口检测

- 检测窗口标题

```
void CheckDebug3()
{
    printf("CheckDebug3:\n");
    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)
    {
        printf("\tDebug\n");
    }
    else
    {
        printf("\tNot debug\n");
    }

}
```

（3）窗口检测

- 破解方法
  - Hook窗口检测函数，使该种检测方法失效

（4）调试器断点检测

- 检测原理
  - 在软件逆向工程中，为了帮助代码分析人员进行软件分析，可以使用调试器设置一个断点，或是单步执行一个进程
  - 代码常使用几种反调试技术，探测软件/硬件断点以及单步执行等行为

（4）调试器断点检测

- 检测方法
  - 检测软件断点
  - 检测硬件断点
  - 检测单步执行

（4）调试器断点检测

- 调试器断点
  - 软件断点
  - 硬件断点
  - 内存断点
- 单步执行
  - 单步步入
  - 单步步过

（4）调试器断点检测

调试器断点

- 软件断点
  - 又称int 3断点，这是一种基于软中断机制断点，3为中断号，使用软件断点时会将下断处的代码修改为0xCC，因此又称0xCC断点，当程序执行到此条指令时异常由调试器捕获并让程序暂停下来，从而实现了软件断点
- 硬件断点
  - 由硬件实现，其只用两位记录断点长度，所以只支持4个硬件断点
- 内存断点
  - 其原理主要基于内存属性，当下读写断点时，调试器会修改断点处读写属性，如果程序对此数据读写的话，会产生读写异常，调试器捕捉此异常并分析，其可以知道运行到何处，对代码段也可以下此断点

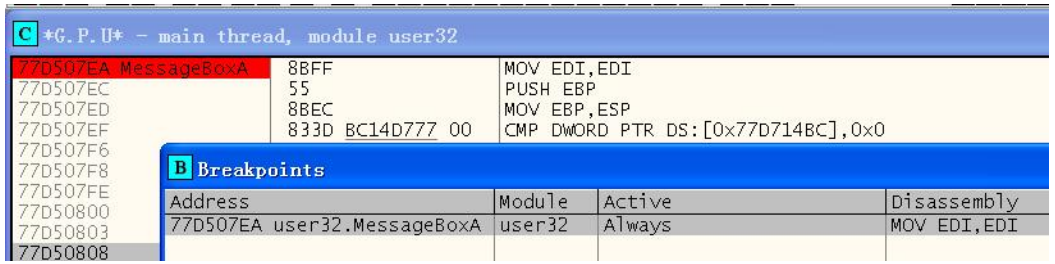
（4）调试器断点检测

- 软件断点
  - 在进行程序调试时，难免会设置一些软件断点，软件断点对应十六进制为0xCC，通过对代码段中的数据进行检测，若能检测到该指令，即可判断程序是否处于调试状态
  - 但是在检测时不能暴力的去扫描函数代码中是否存在0xCC，因为0xCC不仅仅会被用做操作码，也同时会被用作操作，因此，暴力的扫描并不可靠

（4）调试器断点检测

软件断点

- 而在实际调试的过程中经常会对API下断，通过对常用API的首字节检测0xCC，可以较为可靠的判断程序是否处于调试状态




```
unsigned char* func = (unsigned char*)GetProcAddress(LoadLibrary("
    user32.dll"),"MessageBoxA");
if(func[0]==0xcc)
    MessageBox(0,"Warnning","Title",0);
return 0;
```

（4）调试器断点检测

- 硬件断点
  - Dr0~Dr7是调试寄存器组，Dr0~Dr3用于设置硬件断点，由于只有四个寄存器，因此只能设置四个硬件断点，由硬件断点产生的异常是STATUS_SINGLE_STEP(单步异常)。Dr4和Dr5是由系统保留的。Dr7是一些控制位,用于控制断点的方式,Dr6是用于显示哪个硬件调试寄存器引发的断点,如果是Dr0 ~ Dr3的话,相应位会被置1
  - 通过GetThreadContext，可以获取线程的寄存器信息。通过对Dr0~Dr3的检测，可以检测出程序是否存在硬件断点。从而检测出程序是否被调试

（4）调试器断点检测

- 硬件断点

```
CONTEXT lpContext;
GetThreadContext(GetCurrentThread(),&lpContext);
printf("%08X %08X %08X %08X\n",lpContext.Dr0,lpContext.Dr1,lpContext.Dr2
    ,lpContext.Dr3);
if(lpContext.Dr0 || lpContext.Dr1 || lpContext.Dr2 || lpContext.Dr3)
{
    MessageBoxA(0,"Warnning","Title",0);
}
return 0;
```

（4）调试器断点检测

- 单步执行
  - 单步步入
    - 1.通过调试符号获取当前指令对应的行信息，并保存该行的信息
    - 2.设置TF位(Trap Flag，单步调试标记位)，开始CPU的单步执行
    - 3.在处理单步执行异常时，获取当前指令对应的行信息，与之前保存的行信息进行比较。如果相同，表示仍然在同一行上，重新设置TF位执行；如果不相同，表示已到了不同的行，结束单步步入

（4）调试器断点检测

- 单步执行
  - 单步步过
    - 1. 通过调试符号获取当前指令对应的行信息，并保存该行的信息
    - 2. 检查当前指令是否CALL指令。如果是，则在下一条指令设置一个断点，然后让被调试进程继续运行；如果不是，则设置TF位，开始CPU的单步执行，处理单步异常
    - 3. 处理断点异常时，恢复断点所在指令第一个字节的内容。然后获取当前指令对应的行信息，与之前所保存的行信息进行比较，如果相同，重复执行2；不同则结束单步步过
    - 5. 处理单步异常时，获取当前指令对应的行信息，与之前保存的行信息进行比较。如果相同，表示仍然在同一行上，重新设置TF位执行；如果不相同，表示已到了不同的行，结束单步步过

（4）调试器断点检测

- 单步执行
  - 根据前面对调试器实现单步步入以及单步步过的说明，可以得知调试器主要依赖于TF标志位来进行单步执行操作。因此结合SEH与TF标志位可以用于检测调试器
  - 代码中通过SEH捕获由TF标志位产生的EXCEPTION_SINGLE_STEP异常，如果异常被程序正常捕获说明程序未被调试，否则说明程序处于调试状态

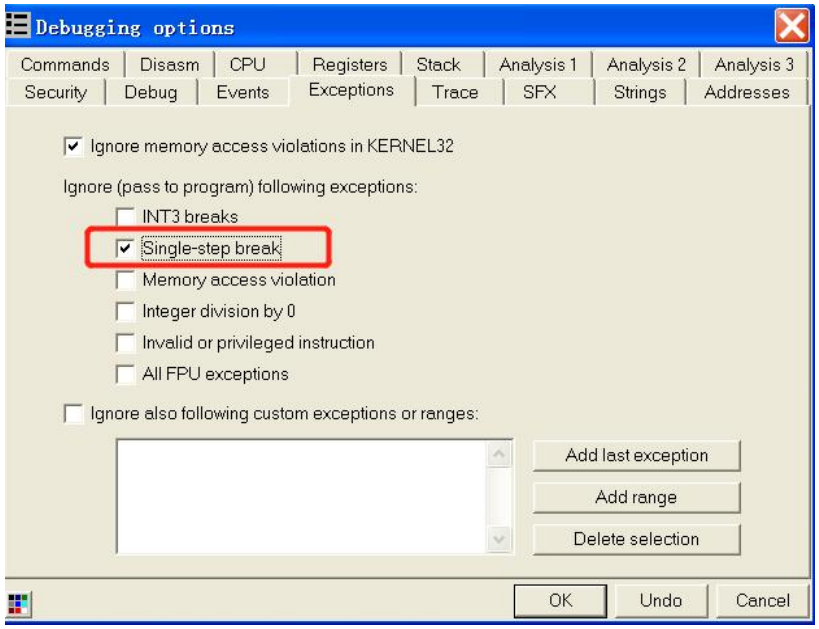
（4）调试器断点检测

- 单步执行

```
int main()
{
    int flag = 1;
    try
    {
        __asm
        {
            pushfd
            or DWORD PTR SS:[ESP],0x100
            popfd
            nop
        }
    }
    catch(...)
    {
        flag = 0;
    }
    if(flag)
    {
        MessageBox(0,"Warnning","Title",0);
    }
    return 0;
}
```

(4) 调试器断点检测

- 破解方法
  - 为了应对该种方法，首先应该在OllyDbg的Option->Debugging Option->Exceptions中勾选Single-step break，即忽略单步异常



（5）代码校验和

- 检测原理
  - 调试器在设置断点或是单步执行一个进程的时候，它们会修改进程中的代码
  - 可以对代码完整性校验，判断是否处于被调试状态

（5）代码校验和

- 代码校验和
  - 代码校验和通常用来校验代码完整性，防止破解人员修改关键函数代码。求代码校验和的方法有很多种，既可以自行实现，也可以使用现成的算法，只要验证校验值是否与预期一致即可。实际中可以运用CRC32算法或各种哈希算法进行验证
  - 由于软件断点的特性，代码校验和也可以轻松的检测出软件断点的存在。如果在校验的同时对硬件断点寄存器进行了检测，也可检测出硬件断点的存在状况。代码校验和是一种较为通用的检测方法

（5）代码校验和

- 代码校验和

```
void Checksum()
{
    BOOL bDebugging = FALSE;
    DWORD dwSize = 0, i, check = 0;
    DWORD* buf = (DWORD*)Checksum;
    dwSize =  (DWORD)main -(DWORD)Checksum;

    for(i=0;i<dwSize/4;i++)
    {
        check ^= buf[i];
    }

    bDebugging = check == OrgChecksum;

    if( bDebugging )  MessageBox(0,"Not debugging...","Caption",0);
    else              MessageBox(0,"Debugging!!!","Caption",0);
}
```

（6）时钟检测

- 时钟检测原理
  - 代码被调试时，进程的运行速度大大降低
    - 单步调试等功能的使用将大幅降低代码的运行速度
    - 时钟检测是代码探测调试器存在的最常用方式之一

（6）时钟检测

- 时钟检测的两种方法
  - 记录一段操作前后的时间戳，然后比较这两个时间戳，如果存在滞后，则可以认为存在调试器
  - 记录触发一个异常前后的时间戳
    - 如果不调试进程，可以很快处理完异常
    - 如果正在调试进程，异常处理存在时延
      - 调试器处理异常的速度非常慢，因为默认情况下，调试器处理异常时需要人为干预，这导致大量延迟
      - 虽然很多调试器允许我们忽略异常，将异常直接返回程序，但这样操作仍然存在不小的延迟

（7）双进程反调试

- 双进程反调试
  - 通常为一个进程以调试器的方式再次运行自身，父进程作为调试器，子进程运行真正的逻辑。当攻击者调试程序时只能调试父进程，无法执行到真正的程序逻辑中去，当攻击者使用调试器attach子进程时，由于子进程已经处于被调试状态，所以会弹出无法attach的提示
  - 同时因为父进程作为调试器，子进程中可以设置各种障碍，故意触发异常，由父进程调过异常代码，或修复错误代码再继续执行

（7）双进程反调试

- 破解方法
  - 修改分支语句强行进入子进程中可执行正确逻辑
  - 需要熟悉父进程对子进程的操作并在执行子进程过程中进行相同的操作，才能保证程序的正常运行

第十一章 反调试技术

- 一. 反调试技术简述
- 二. 代码调试状态检测常用方法
- 三. 代码扰乱调试器常用方法

- 代码扰乱调试器常用方法
  - 利用中断
  - 利用异常处理机制
  - 利用调试器漏洞

（1）利用中断

- 扰乱原理
  - 调试器通常使用INT 3来设置软件断点，一种反调试技术就是在合法代码段中插入0xCC(INT 3)欺骗调试器，使其认为这些0xCC机器码是自己设置的断点

（1）利用中断

- 实现方法
  - 当程序处于调试状态，程序返回FALSE；否则，返回TURE

```
1   BOOL CheckDebug()
2   {
3           __try
4           {
5                   __asm int 3
6           }
7           __except(1)
8           {
9                   return FALSE;
10          }
11          return TRUE;
12  }
```

(2) 利用异常处理机制

- 扰乱原理及实现方法
  - RaiseException函数产生的若干不同类型的异常可以被调试器捕获

```
1   BOOL TestExceptionCode(DWORD dwCode)
2   {
3         __try
4         {
5               RaiseException(dwCode, 0, 0, 0);
6         }
7         __except(1)
8         {
9               return FALSE;
10        }
11        return TRUE;
12  }
13
14  BOOL CheckDebug()
15  {
16        return TestExceptionCode(DBG_RIPEXCEPTION);
17  }
```

(2) 利用异常处理机制

- 扰乱原理及实现方法
  - SetUnhandledExceptionFilter函数在调试状态下不起作用，所以不会调用自定义的异常处理函数，而且也不会在SEH链中看到自定义的异常处理函数

（3）利用调试器漏洞

- 扰乱原理
  - 与所有软件一样，调试器也存在漏洞，有时恶意代码编写者为了防止被调试，会攻击这些漏洞
    - PE头漏洞
    - OutputDebugString漏洞

谢　谢！