局部钩子能防全局钩子吗_手动打造应用层钩子扫描

weixin_39979516 于 2020-12-01 09:21:10 发布 31 收藏
文章标签： 局部钩子能防全局钩子吗
本文为看雪论坛优秀文章看雪论坛作者ID：千音丶

前言

>>>>文章概述

有时候我们想看某个游戏或程序下了什么钩子（应用层），或者说，我们想看某个外挂对某个进程写入了一些什么数据，我们可以选择PCHunter这款工具里的进程钩子功能来看我们所需要的东西，但是现在很多游戏或者程序都对这款软件加入了检测，所以我就想自己来实现做一个钩子扫描的工具，也算是对PE知识的活学活用。

>>>>所需知识

PE、C语言、win32

>>>>环境

VS2017

思路

我不知道PCHunter是怎么实现这个功能的，可能比较高级，我是按照我自己的思路来的。

首先，要对某个程序下钩子，那肯定是要下在程序的代码段的，所以我们的关注点就是代码段（也就是可执行的区段），所以，我们先把该程序在硬盘中的代码段读出来，然后在读取内存中的代码段，然后逐字节进行对比，这样就可以把下钩子的地方找出来。大致的思路如下：

1、绑定进程

2、遍历该程序的所有模块

3、读取该模块在硬盘中的内容

4、确定该模块在硬盘中代码段的位置和大小

5、修复重定位表

6、读取该模块在内存中代码段内容

7、逐个指令进行比较

实现过程

>>>>1. 绑定进程

```
//通过进程名获取进程句柄 参数：进程名
HANDLE GetProcessHandle(PWCHAR name)
{

    //初始化进程快照
    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(PROCESSENTRY32);
    HANDLE hProcessSanp = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0); //获得快照句柄

    Process32First(hProcessSanp, &pe32); //获取第一个进程
    do
    {


        if (wcscmp(name, pe32.szExeFile) == 0)
        {
            CloseHandle(hProcessSanp);//关闭快照句柄，避免内存泄漏
            return OpenProcess(PROCESS_ALL_ACCESS, FALSE, pe32.th32ProcessID); //注意这里使用的权限

        }
    } while (Process32Next(hProcessSanp, &pe32));

    CloseHandle(hProcessSanp);//关闭快照句柄，避免内存泄漏
    return (HANDLE)NULL;


}
```

>>>>2. 映射PE文件数据到内存

```
//映射PE文件数据 参数：文件路径，用来存放数据的指针
DWORD ReadPEFile(IN LPSTR file_path, OUT LPVOID* pFileBuffer)
{

    FILE* fp = fopen(file_path, "rb");

    if (fp == NULL)
    {
        printf("打开文件失败n");
        return 0;
    }
    fseek(fp, 0, SEEK_END);//移动文件位置到末尾
    long long filesize = ftell(fp);//计算大小
    fseek(fp, 0, SEEK_SET); //恢复到文件开始的位置

    LPVOID pFileBuffer_temp = malloc(sizeof(char)*filesize); //开辟指定大小的内存
    if (pFileBuffer_temp == NULL)
    {
        printf("开辟空间失败n");
        fclose(fp);
        return 0;

    }

    size_t n = fread(pFileBuffer_temp, sizeof(char), filesize, fp); //将文件数据拷贝到缓冲区
    if (!n)
    {
        printf("读取数据失败n");
        free(pFileBuffer_temp);
        fclose(fp);
        return 0;
    }

    *pFileBuffer = pFileBuffer_temp;
    pFileBuffer_temp = NULL;
    fclose(fp);
    return filesize;
}
```

>>>>3. 读取各个节表的信息

首先先定义一下节表：

```
IMAGE_DOS_HEADER* peDosHeader; //dos头
IMAGE_NT_HEADERS* peNtHeader; //nt头
IMAGE_FILE_HEADER* peFileHeader; //标准pe头
IMAGE_OPTIONAL_HEADER32* peOptionalHeader; //可选pe头
IMAGE_SECTION_HEADER* peSectionHeader; //节表
```

```
//读取表的信息
DWORD PeHead_information(IN LPVOID pFileBuffer)
{

    if (pFileBuffer == NULL)
    {
        printf("缓冲区指针无效n");
        return 0;
    }

    /*判断是否是有效的MZ标记*/
    if (*((PWORD)pFileBuffer) != IMAGE_DOS_SIGNATURE) //PWORD就是word*类型，取头两个字节的内容， IMAGE_DOS_SI
    {
        printf("不是有效的MZ标记n");
        return 0;
    }

    peDosHeader = (IMAGE_DOS_HEADER*)pFileBuffer; //dos赋值，因为是结构体，所以只需要知道首地址就等于知道了dos


    /*判断是否是有效的pe标志*/
    if (*((PDWORD)((DWORD)pFileBuffer + peDosHeader->e_lfanew)) != IMAGE_NT_SIGNATURE) //这里为什么要转成DOW
    {
        printf("不是有效的pe标记n");
        return 0;
    }



    peNtHeader = (IMAGE_NT_HEADERS*)((DWORD)pFileBuffer + peDosHeader->e_lfanew); //NT头赋值

    peFileHeader = (IMAGE_FILE_HEADER*)((DWORD)peNtHeader + 4); //标准pe头赋值

    peOptionalHeader = (IMAGE_OPTIONAL_HEADER32*)((DWORD)peFileHeader + IMAGE_SIZEOF_FILE_HEADER);//可选pe头

    peSectionHeader = (IMAGE_SECTION_HEADER*)((DWORD)peOptionalHeader + peFileHeader->SizeOfOptionalHeader)

}
```

>>>>4.拉伸PE

```
//拉伸PE 参数：硬盘状态的PE数据指针，用来存放拉伸后的PE数据的指针
DWORD simulation_ImageBuffer(IN LPVOID pFileBuffer, OUT LPVOID* pImageBuffer)
{
    if (pFileBuffer == NULL)
    {
        //printf("缓冲区指针无效n");
        return 0;
    }

    LPVOID pImageBuffer_temp = malloc(sizeof(char)*peOptionalHeader->SizeOfImage);//申请ImageBuffer所需的内存
    if (pImageBuffer_temp == NULL)
    {
        //printf("开辟ImageBuffer空间失败n");
        return 0;
    }

    memset(pImageBuffer_temp, 0, peOptionalHeader->SizeOfImage); //将空间初始化为0
    memcpy(pImageBuffer_temp, pFileBuffer, peOptionalHeader->SizeOfHeaders); //把头+节表+对齐的内存复制过去

    /*复制节*/
    for (int i = 0; i < peFileHeader->NumberOfSections; i++)
    {

        pImageBuffer_temp = (LPVOID)((DWORD)pImageBuffer_temp + (peSectionHeader + i)->VirtualAddress);//定
        pFileBuffer = (LPVOID)((DWORD)pFileBuffer + (peSectionHeader + i)->PointerToRawData); //定位这个节在
        memcpy(pImageBuffer_temp, pFileBuffer, (peSectionHeader + i)->SizeOfRawData); //复制节在文件中所占的

        pImageBuffer_temp = (LPVOID)((DWORD)pImageBuffer_temp - (peSectionHeader + i)->VirtualAddress);
        pFileBuffer = (LPVOID)((DWORD)pFileBuffer - (peSectionHeader + i)->PointerToRawData); //恢复到起始位
    }


    *pImageBuffer = pImageBuffer_temp;
    pImageBuffer_temp = NULL;
    return peOptionalHeader->SizeOfImage;

}
```

>>>>5. 修复重定位表

这里说一下，为什么要修复重定位表呢？

假设一个程序用到了一个局部变量， 那么编译时生成的地址 = ImageBase + RVA， 这个地址在程序编译完成后，已经写入文件了。

那假设，程序在加载的时候，没有按照预定的基址载入到指定的位置呢？（1）也就是说，如果程序能够按照预定的ImageBase来加载的话，那么就不需要重定位表 ，这也是为什么exe很少有重定位表，而DLL大多都有重定位表的原因。（2）一旦某个模块没有按照ImageBase进行加载，那么所有类似上面中的地址就都需要修正，否则，引用的地址就是无效的。（3）一个EXE中，需要修正的地方会很多，那我们如何来记录都有哪些地方需要修正呢？

答案就是重定位表。

在本项目中，如果不修复重定位表的话，那么当我们读取到有局部变量的汇编的时候，那肯定就出问题了。

修复重定位表思路：1、首先定位重定位表2、然后需要修复的地址就是：具体项的后12位+每块的VirtualAddress+imagebuffer3、怎么修复？就是把需要修复的地址里的值在原来的基础上+（新的基址-旧的基址）4、通过循环修复完毕

```
//修复重定位表 参数：拉伸后的PE指针，该模块被加载时候的基址
VOID RepairRelocation(IN LPVOID pImageBuffer,DWORD NewImageBase)
{
    if (pImageBuffer == NULL)
    {
        printf("指针无效n");
        return;
    }

    IMAGE_DOS_HEADER* pImageBuffer_peDosHeader = (IMAGE_DOS_HEADER*)pImageBuffer;
    IMAGE_NT_HEADERS* pImageBuffer_peNtHeader = (IMAGE_NT_HEADERS*)((DWORD)pImageBuffer + pImageBuffer_peDo
    IMAGE_FILE_HEADER* pImageBuffer_peFileHeader = (IMAGE_FILE_HEADER*)((DWORD)pImageBuffer_peNtHeader + 4)
    IMAGE_OPTIONAL_HEADER32* pImageBuffer_peOptionalHeader = (IMAGE_OPTIONAL_HEADER32*)((DWORD)pImageBuffer
    IMAGE_SECTION_HEADER* pImageBuffer_peSectionHeader = (IMAGE_SECTION_HEADER*)((DWORD)pImageBuffer_peOpti

    /*定位重定位表地址*/
    IMAGE_DATA_DIRECTORY* pImageBuffer_Data = (IMAGE_DATA_DIRECTORY*)((DWORD)(&(pImageBuffer_peOptionalHead
    IMAGE_DATA_DIRECTORY* pImageBuffer_Relocation_temp = (IMAGE_DATA_DIRECTORY*)pImageBuffer_Data + 5;//数据
    IMAGE_BASE_RELOCATION* pImageBuffer_Relocation = (IMAGE_BASE_RELOCATION*)((DWORD)pImageBuffer + pImageB



    /*计算块的总数*/
    DWORD size_block = 0; //存放块的总数

    while ( pImageBuffer_Relocation->SizeOfBlock != 0 && pImageBuffer_Relocation->VirtualAddress != 0)
    {

        size_block++;
        pImageBuffer_Relocation = (IMAGE_BASE_RELOCATION*)( (DWORD)pImageBuffer_Relocation + pImageBuffer_R

    }
    pImageBuffer_Relocation = (IMAGE_BASE_RELOCATION*)((DWORD)pImageBuffer + pImageBuffer_Relocation_temp->

    /* 修复重定位表 */
    LPWORD temp = (LPWORD)((DWORD)(&(pImageBuffer_Relocation->SizeOfBlock)) + 4); //存放具体项
    DWORD size_temp = 0;//存放具体项的数目
    LPDWORD dest = NULL; //要修复的地址

    for (int i = 0; i < size_block; i++)
    {
        if (pImageBuffer_Relocation->SizeOfBlock <=8)
        {
            continue;
        }

        size_temp = (pImageBuffer_Relocation->SizeOfBlock - 8) / 2; //计算具体项的个数

        for (int j = 0; j < size_temp; j++)
        {

            dest = (LPDWORD)( (temp[j] & 0x0FFF) + pImageBuffer_Relocation->VirtualAddress + (DWORD)pImageB
            if (temp[j] >> 12 == 3) //只有当具体项的前四位是3的时候才需要修改
            {
                *dest = *dest + (NewImageBase - pImageBuffer_peOptionalHeader->ImageBase); //当前地址+(新的基
```

```
                *dest = *dest + (NewImageBase- pImageBuffer_peOptionalHeader->ImageBase); //当前地址+(新的基
            }
        }

        pImageBuffer_Relocation = (IMAGE_BASE_RELOCATION*)((DWORD)pImageBuffer_Relocation + pImageBuffer_Re
        temp = (LPWORD)((DWORD)(&(pImageBuffer_Relocation->SizeOfBlock)) + 4);

    }




}
```

>>>>6. 获取模块代码段的位置和大小

创建了一个结构体用来存放代码段信息。

```
//Test段的信息
struct TestInformation
{
    DWORD VirtualAddress; //节区在内存的偏移
    DWORD PointerToRawData; //节区在文件中的偏移
    DWORD SizeTest; //大小
};
```

之前我获取代码段是通过text这个名字去判断的，但是后面发现会有bug，因为有些程序的代码段不止一处，所以我们要用可执行的标志的去判断。

```
//获取本程序代码段的起始地址（在内存中的）和大小 返回值：返回代码段的数量
DWORD GetTestInformation(TestInformation* te)
{

    int len = 0;
    for (int i = 0; i < peFileHeader->NumberOfSections; i++)
    {

        if (((peSectionHeader + i)->Characteristics & 0x20000000) == 0x20000000) //判断是否是可执行的代码
        {
            (te+len)->VirtualAddress = (peSectionHeader + i)->VirtualAddress;
            (te + len)->PointerToRawData = (peSectionHeader + i)->PointerToRawData;
            (te + len)->SizeTest = (peSectionHeader + i)->SizeOfRawData;
            len++;
        }

    }

    return len;
}
```

>>>>7. 具体实现

首先自然是要遍历模块了，然后所有的工作都在遍历模块的这个循环中完成。

```
MODULEENTRY32 me32;
    me32.dwSize = sizeof(MODULEENTRY32); //在使用这个结构前，先设置它的大小
    HANDLE hModuleSnap = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, pid); //HANDLE也是属于句柄，是通用句柄 ，I

    if (hModuleSnap == INVALID_HANDLE_VALUE) //INVALID_HANDLE_VALUE表示无效的句柄
    {
        MessageBoxA(0, "模块读取失败", 0, 0);
        return;
    }

    BOOL bMore = Module32First(hModuleSnap, &me32); //获取第一个模块信息

    char* ModulePath = NULL; //模块路径
    while (bMore)
    {
        ........
       }
```

读取硬盘里的PE内容，并且拉伸。

```
/* 获取模块路径以及获取模块名字 */
USES_CONVERSION;
ModulePath = W2A(me32.szExePath); //获得模块路径
char ModulePath_big[MAX_PATH] = { 0 }; //存放大写的结果
Conversion_Big(ModulePath, ModulePath_big); //这里全部转换成了大写，防止不一致

ModuleBase = (DWORD)me32.modBaseAddr;

char* ModuleName = my_strstr(ModulePath, "");
while (my_strstr(ModuleName, "") != NULL)
{
    ModuleName += 1;
}

/* 定位PE的代码节和内存的代码节的地址及大小 */
ReadPEFile(ModulePath, &FileBuffer); //映射PE文件
PeHead_information(FileBuffer); //读取各个表的信息
simulation_ImageBuffer(FileBuffer, &ImageBuffer); //模拟imagbebuffer

TestInformation performCode[10];
memset(performCode, 0, sizeof(TestInformation)* 10);

CodeLen = GetTestInformation(performCode); //获取代码段的结构信息 ------------------
```

读取硬盘代码段内容：

```
char* PeText_temp[10] = { 0 }; 用来保存代码段的内容的，因为代码段可能不止一个，所以用了指针数组

    if ((performCode + 0)->SizeTest != 0 )
    {

        for (int i = 0; i < CodeLen; i++)
        {
            PeText_temp[i] = (char*)malloc((performCode + i)->SizeTest);
        }

        RepairRelocation(ImageBuffer, ModuleBase); //修复重定位表


        for (int i = 0; i < CodeLen; i++)
        {
            memcpy(PeText_temp[i], (char*)((DWORD)ImageBuffer + (performCode + i)->VirtualAddress), (perfor
        }


    }
```

读取内存中代码段内容：

```
char* MeText_temp[10] = { 0 };
if ((performCode + 0)->SizeTest != 0 )
{


    for (int i = 0; i < CodeLen; i++)
    {
        MeText_temp[i] = (char*)malloc((performCode + i)->SizeTest);
    }

    DWORD ProtectTemp = NULL;
    for (int i = 0; i < CodeLen; i++)
    {
        VirtualProtectEx(g_hProcess, (LPVOID)(ModuleBase + (performCode + i)->VirtualAddress), sizeof(char)
        ReadProcessMemory(g_hProcess, (void*)(ModuleBase + (performCode + i)->VirtualAddress), MeText_temp[
        VirtualProtectEx(g_hProcess, (LPVOID)(ModuleBase + (performCode + i)->VirtualAddress), sizeof(char)
    }

}
```

比较部分：

我用的是逐个指令比较的，那么就需要获取每条指令的长度，需要自己写个反汇编引擎，感觉麻烦，所以偷了个懒，直接用了https://bbs.pediy.com/thread-147401.htm这里的代码，大家也可以去参考一下。

```
/* 比较部分 */
if ((performCode + 0)->SizeTest != 0 )
{

    char OpCode[2] = { 0 }; //存放OpCode
    int indicators = 0; //指标
    int Asmlen = 0; //存放指令长度
```

```
char OriginalCode[20] = { 0 }; //存放原始字节
char NowCode[20] = { 0 }; //存放现在字节
char ff = 0xff; //过滤IAT的
int List_line = 0; //list的当前行数

for (int s = 0; s < CodeLen; s++)
{
    while (indicators < (performCode + s)->SizeTest)
    {


        OpCode[0] = *(LPBYTE)((DWORD)PeText_temp[s] + indicators);
        OpCode[1] = *(LPBYTE)((DWORD)PeText_temp[s] + indicators + 1); //读取opcode
        Asmlen = Decode_Size(OpCode); //解析指令长度

        memcpy(OriginalCode, PeText_temp[s] + indicators, Asmlen); //读取原始字节
        memcpy(NowCode, MeText_temp[s] + indicators, Asmlen); //读取当前字节


        for (int i = 0; i < Asmlen; i++)
        {
            if (OriginalCode[i] != NowCode[i] /*&& memcmp(&OpCode[0], &ff, 1) != 0*/) //判断字节是否不相等
            {
                char Addres[30] = { 0 };
                char arr1[30] = { 0 }; //原始字节字符串
                char arr2[30] = { 0 }; //现在字节字符串

                BytesToHexStr1((unsigned char*)OriginalCode, Asmlen, arr1);
                BytesToHexStr1((unsigned char*)NowCode, Asmlen, arr2);

                sprintf(Addres, "%s+%x", ModuleName, indicators + (performCode + s)->VirtualAddress);



                //判断是否需要过滤
                BOOL guolv = false;
                if (SizeFilter > 0)
                {
                    for (int j = 0; j < SizeFilter; j++)
                    {
                        if (strcmp(&Filter[j * 40], Addres) == 0)
                        {
                            guolv = true;
                            break;
                        }
                    }

                }

                if (guolv == true)
                {
                    break;
                }


                USES_CONVERSION;
                m_ListControl.InsertItem(List_line, A2W(Addres));
                m_ListControl.SetItemText(List_line, 1, A2W(arr2));
                m_ListControl.SetItemText(List_line, 2, A2W(arr1));
                List line++;
```

```
                    break;

                }
            }

            memset(OriginalCode, 20, 0);
            memset(NowCode, 20, 0);
            indicators += Asmlen;

        }

        indicators = 0;

    }

}
```

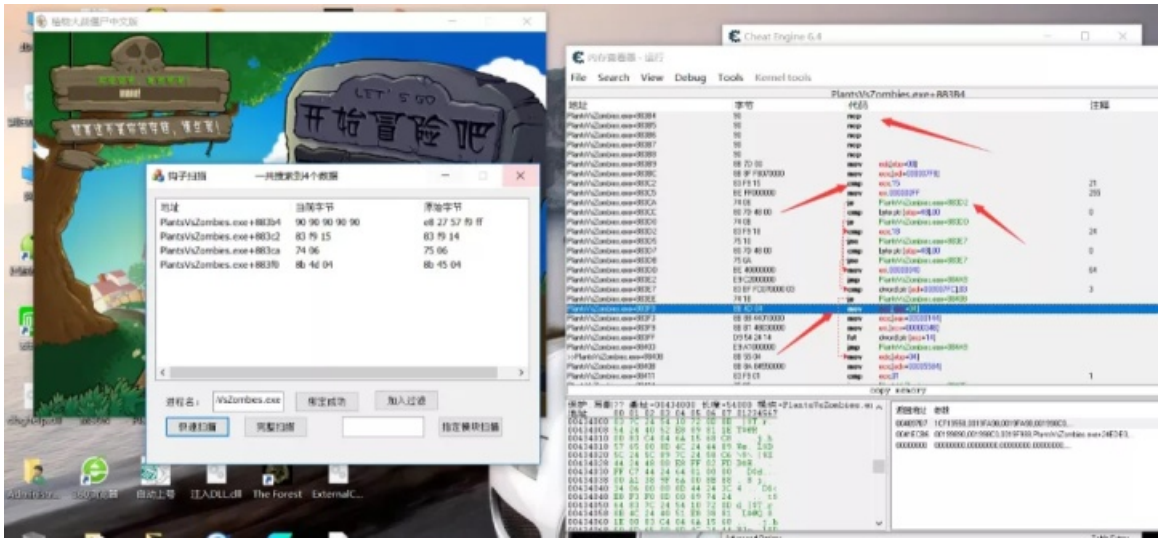
当然了，不能忘了清理工作。

```
/* 清理工作 */
    memset(ModulePath_big, 0, MAX_PATH);
    for (int i = 0; i < CodeLen; i++)
    {
        free(PeText_temp[i]);
        free(MeText_temp[i]);

    }


    free(FileBuffer);
    free(ImageBuffer);
    FileBuffer = NULL;
    ImageBuffer = NULL;
    bMore = Module32Next(hModuleSnap, &me32);

}

CloseHandle(hModuleSnap);
```

效果图

这里有快速扫描和完整扫描，快速扫描就是果过滤了系统dll。

弄了一个过滤的功能，方便自己分析。



总结

代码写得不是很好，请多包含，就看个思路，然后不足之处说一下。

（1）对有驱动保护的程序扫描不了，原因就不说了，解决办法有很多，可以过保护，也可以写成dll，然后劫持进去。

（2）有些模块代码段会被加密，运行的时候才解密代码，这时候扫描不正确了。

- End -

看雪ID：千音、

https://bbs.pediy.com/user-840606.htm

*本文由看雪论坛 千音、 原创，转载请注明来自看雪社区

推荐文章++++

* AFL afl_fuzz.c 详细分析

* 固件分析--工具、方法技巧浅析（上）

* 固件分析--工具、方法技巧浅析（下）

* 7种Android Native Anti Hook的实现思路

* 跨平台模拟执行 - AndroidNativeEmu实用手册