VEH+硬件断点实现无痕HOOK

鬼手56 于 2021-09-24 16:24:20 发布 1948 收藏 20
分类专栏： 软件逆向 文章标签： windows hook
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/qq_38474570/article/details/120457798
版权

软件逆向 专栏收录该内容
17 篇文章 109 订阅
订阅专栏

文章目录

- hook的分类
- 硬件断点hook原理
  - 设置硬件断点
  - 注册VEH
- 代码实现VEH无痕Hook
- 说说一路踩过的坑
- 实际效果
- 小结
- 关于veh hook的对抗
- 参考文章

hook的分类

hook方式有多种，这里做了一个系统性的总结对比，如下：

	初(low)级(bi)hook	普通hook	高级hook	神级hook
方式	对hook点用call或jmp替换	对hook点用0xCC替换	对hook点下硬件断点，不改变原有的硬编码	VT读写分离
原理	找准hook点，更改原机器码为jmp或call，跳转到自己的处理逻辑。然后修复被破坏的机器码，接着跳回原hook点继续执行	找准hook点，更改为0xCC，原程序执行到这里时产生异常。这里可以自定义异常处理函数(通过AddVectoredExceptionHandler添加)来实现自定义的逻辑，然后跳转回原hook点执行	找准hook点，通过设置DR0-DR3、DR7下硬件断点，程序执行到这里后产生异常，由我们自定义的异常处理函数(通过AddVectoredExceptionHandler添加)接管，来实现自定义的处理逻辑	找准hook点，更改原机器码为jmp或call，跳转到自己的处理逻辑。然后修复被破坏的机器码，接着跳回原hook点继续执行
优点	1、原理简单粗暴，容易理解，适合新手入门	1、原理和OD、x32dbg等调试器的断点一样；这里只需要破坏一个字节。 2、并不直接从hook点跳转到自定义的处理函数，不容易被其他外挂/逆向人员找到	1、对原机器码没有任何改动，hook后完全不用修复原机器码 2、可以绕过普通的CRC32检测 3、并不直接从hook点跳转到自定义的处理函数，不容易被其他外挂/逆向人员找到	1、通过页的读写分离，吊打所有的CRC32检测和dll模块检测
缺点	1、无法规避CRC32检测，很容易被检测到 2、jmp或hook至少破坏5个byte，有时为了防止前后的机器码粘连，还需要额外使用NOP断开，造成更多的破坏，动静太大 3、如果hook点遇到了jmp或call，修复时还要重新计算hook点原来的目的地到现在自定义逻辑处的偏移，然后从新的目的地跳转过去，逻辑不难，但是有点复杂，容易算错 4、从hook点跳转到自己的处理逻辑太明显，很容易被其他外挂/逆向找到，然后被“黑吃黑”	1、还是会被破坏1个字节，还是无法通过CRC32检测	1、硬件断点只有4个(当然可以通过VT实现无限硬件断点) 2、原程序一旦设置DR0-DR4为0、或DR7的硬件断点开关为0，硬件断点立即失效（PG为了做CRC32检测，是提前对DR寄存器清零了的）	1、要开启VT：由于物理CPU要执行host和guest的代码，虚拟机内存还要转成物理机内存(也即是EPT)，性能有损耗 2、做成外挂后普通用户使用也需要开启VT，很麻烦 3、VT是可以嵌套的。如果其他程序先一步开启了VT，自己程序的VT还是在别人VT的监控之下，一举一动还是在被监控之下

CSDN @鬼手56

实际上第二种和第三种属于同一类型的hook，都是利用异常处理函数来处理，只是触发异常的方式不同

硬件断点hook原理

想要实现硬件断点hook，需要实现下面几个步骤

1. 设置硬件断点
2. 注册veh异常处理函数
3. 编写异常处理函数，实现自己的hook代码

设置硬件断点

首先来说一下关于硬件断点hook的原理，在Windows API中存在一个重要的结构体**PCONTEXT**

```
typedef struct DECLSPEC_NOINITALL _CONTEXT {

    //
    // The flags values within this flag control the contents of
    // a CONTEXT record.
    //
    // If the context record is used as an input parameter, then
    // for each portion of the context record controlled by a flag
    // whose value is set, it is assumed that that portion of the
    // context record contains valid context. If the context record
    // is being used to modify a threads context, then only that
    // portion of the threads context will be modified.
    //
    // If the context record is used as an IN OUT parameter to capture
    // the context of a thread, then only those portions of the thread's
    // context corresponding to set flags will be returned.
    //
    // The context record is never used as an OUT only parameter.
    //

    DWORD ContextFlags;
```

```
//
// This section is specified/returned if CONTEXT_DEBUG_REGISTERS is
// set in ContextFlags.  Note that CONTEXT_DEBUG_REGISTERS is NOT
// included in CONTEXT_FULL.
//

DWORD   Dr0;
DWORD   Dr1;
DWORD   Dr2;
DWORD   Dr3;
DWORD   Dr6;
DWORD   Dr7;

//
// This section is specified/returned if the
// ContextFlags word contians the flag CONTEXT_FLOATING_POINT.
//

FLOATING_SAVE_AREA FloatSave;

//
// This section is specified/returned if the
// ContextFlags word contians the flag CONTEXT_SEGMENTS.
//

DWORD   SegGs;
DWORD   SegFs;
DWORD   SegEs;
DWORD   SegDs;

//
// This section is specified/returned if the
// ContextFlags word contians the flag CONTEXT_INTEGER.
//

DWORD   Edi;
DWORD   Esi;
DWORD   Ebx;
DWORD   Edx;
DWORD   Ecx;
DWORD   Eax;

//
// This section is specified/returned if the
// ContextFlags word contians the flag CONTEXT_CONTROL.
//

DWORD   Ebp;
DWORD   Eip;
DWORD   SegCs;              // MUST BE SANITIZED
DWORD   EFlags;             // MUST BE SANITIZED
DWORD   Esp;
DWORD   SegSs;

//
// This section is specified/returned if the ContextFlags word
// contains the flag CONTEXT_EXTENDED_REGISTERS.
// The format and contexts are processor specific
//
```

```
    BYTE    ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];

} CONTEXT;

typedef CONTEXT *PCONTEXT;
```

这个结构体里保存着所有的寄存器信息，其中和硬件断点相关的字段有下面几个

```
DWORD   Dr0;
DWORD   Dr1;
DWORD   Dr2;
DWORD   Dr3;
DWORD   Dr6;
DWORD   Dr7;
```

和硬件调试相关的寄存器一共有6个。如果我们能获取到线程的Context环境，并且修改dr寄存器，就能给需要hook的地址下硬件断点。

注册VEH

接下来就进入到第二步，注册异常处理函数。当程序执行到hook点时，触发硬件断点，从而触发`EXCEPTION_SINGLE_STEP`异常，那么我们就可以自己注册一个VEH异常处理函数来处理这个异常，那么就可以在处理完成之后编写自己的hook代码

windwos操作系统专门针对异常的处理有一整套完整的机制，这里为了理解，简单介绍一下：windwos下3环进程运行时，如果遇到异常，大致的处理顺序如下：

1. 先看看有没有调试器（通过编译器运行exe也算），如果有，就发消息给调试器让其处理；
2. 如果没有调试器，或则调试器没处理，进入进程自己的VEH继续处理。VEH本质是个双向链表，存储了异常的handler代码，此时windwos会挨个遍历这个链表执行这些handler（感觉原理和vmp很像，估计vmp借鉴了这里的思路）
3. 如果VEH还没处理好，接着由线程继续处理。线程同样有个异常接管的链表，叫SEH；windows同样会遍历SEH来处理异常
4. 如果SEH还没处理好，继续给线程的UEH传递，UEH只有一个处理函数了
5. 如果UEH还没处理好，就回到进程的VCH处理

基于windwos开发的应用数以万计，微软绝对不可能出厂时就考虑到所有的异常，其各种handler不太可能处理所有的异常，所以微软又开放了接口，让开发人员自定义异常的handler；对于开发人员来说，肯定是越靠前越好，所以这里选择VEH来添加自定义的handler（调试器是最先收到异常通知的，但外挂在正常使用时不太可能有调试的功能，除非开发人员自己单独开发调试器的功能，这样成本太高了）。windwos开放了一个API，叫AddVectoredExceptionHandler，可以给VEH添加用户自定义的异常处理handler，如下

```
AddVectoredExceptionHandler(1, PvectoredExceptionHandler)
```

函数有两个参数：第一个参数如果不是0，那么自定义的handler最先执行；如果是0，那么自定义的handler最后执行。这里我们当然希望自己的handler最先执行了，所以设置成1；另一个参数就是自定义的回调函数。

代码实现VEH无痕Hook

首先我们需要获取到当前的线程环境结构体

```
void SetSehHook()
{

 //遍历线程 通过openthread获取到线程环境后设置硬件断点
 HANDLE hTool32 = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
 if (hTool32!= INVALID_HANDLE_VALUE)
 {
  //线程环境结构体
  THREADENTRY32 thread_entry32;

  thread_entry32.dwSize = sizeof(THREADENTRY32);

  HANDLE hHookThread = NULL;

  //遍历线程
  if (Thread32First(hTool32,&thread_entry32))
  {
   do
   {
    //如果线程父进程ID为当前进程ID
    if (thread_entry32.th32OwnerProcessID==GetCurrentProcessId())
    {
     hHookThread = OpenThread(THREAD_SET_CONTEXT | THREAD_GET_CONTEXT | THREAD_QUERY_INFORMATION, FALSE, thread_
entry32.th32ThreadID);


     SuspendThread(hHookThread);//暂停线程

     //设置硬件断点
     CONTEXT thread_context = { CONTEXT_DEBUG_REGISTERS };
     thread_context.Dr0 = g_HookAddr;
     thread_context.Dr7 = 0x405;

     //设置线程环境 这里抛异常了

     DWORD oldprotect;
     VirtualProtect((LPVOID)g_HookAddr, 5, PAGE_EXECUTE_READWRITE, &oldprotect);//修改PTE p=1 r/w1=0

     SetThreadContext(hHookThread, &thread_context);

     ResumeThread(hHookThread);//线程跑起来吧~~~


     CloseHandle(hHookThread);
    }

   } while (Thread32Next(hTool32, &thread_entry32));



  }
  CloseHandle(hTool32);

 }

}
```

这里通过遍历线程，设置线程环境的方式来给所有的线程设置硬件断点。接着注册VEH异常处理函数

```
AddVectoredExceptionHandler(1, (PVECTORED_EXCEPTION_HANDLER)ExceptionFilter);//添加VEH异常处理
```

接着编写回调函数

```
LONG NTAPI  ExceptionFilter(PEXCEPTION_POINTERS ExceptionInfo)
{

 //判断当前异常码是否为硬件断点异常
 if (ExceptionInfo->ExceptionRecord->ExceptionCode== EXCEPTION_SINGLE_STEP)
 {

  //判断发生异常的地址是否和hook的地址一致
  if ((DWORD)ExceptionInfo->ExceptionRecord->ExceptionAddress == g_HookAddr)
  {

   //获取当前线程上下文
   PCONTEXT pcontext = ExceptionInfo->ContextRecord;

   //获取聊天记录
   RecvMsg(pcontext);

   //修复EIP
   pcontext->Eip=(DWORD)&OriginalFunc;


   //异常处理完成 让程序继续执行
   return EXCEPTION_CONTINUE_EXECUTION;

  }
 }
 return EXCEPTION_CONTINUE_SEARCH;
}
```

最后需要修复EIP，让程序正常运行，我这里时让EIP指向一个逻函数，再通过裸函数跳转到目标返回地址

```

void __declspec(naked) OriginalFunc(void)
{
 __asm
 {
  //调用被覆盖的call
  call OverReciveMsgCallAddr;
  //跳转到返回地址
  jmp RetkReciveMsgAddr;

 }
}
```

说说一路踩过的坑

整个过程从原理看上去很简单，但实际操作起来会遇到几个坑。

[外链图片转存失败,源站可能有防盗链机制,建议将图片保存下来直接上传(img-wMQoCzM3-1632471774460)(VEH+硬件断点实现无痕HOOK.assets/image-20210924154835965.png)]

遇到的第一个问题，当我直接注入dll到目标进程时，`AddVectoredExceptionHandler`这个函数直接崩溃，但是我用VS调试dll的时候，就不报错了，并且回调函数正常执行。

实际上这个问题的原因是由于调用了`OutputDebugStringA`造成的。

```
在内部，调试字符串作为异常处理。OutputDebugString 使用 DBG_PRINTEXCEPTION_C（定义为 0x40010006）和字符串地址和大小作为异常参数调用 RaiseException。
```

OutputDebugString是通过抛异常的方式来实现的，如果在VEH相关的地方调用这个函数就会导致无限递归，然后导致堆栈溢出

第二个问题是硬件断点无法设置成功，后来在看雪的评论区找到了答案。

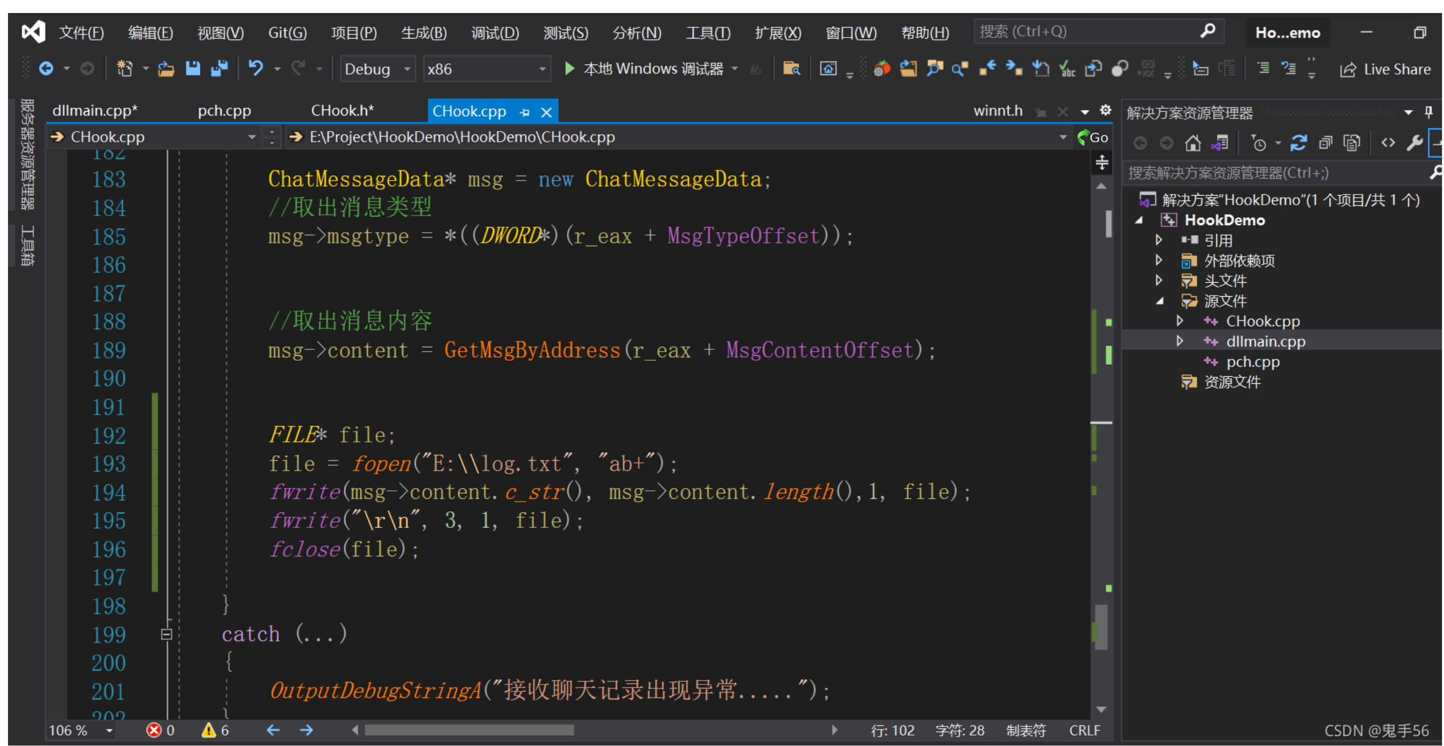
```
SetThreadContext:
Do not try to set the context for a running thread; the results are unpredictable.

这是CSDN  上的对SetThreadContext 解释。
尤其是对自身的上下文进行设置，这很可能使得自身线程崩溃或者得不到想要的结果

历遍所有线程，SuspendThread，然后SetThreadContext ，再 ResumeThread ，同时排除自身线程。
```

需要同时遍历所有的线程，并且排除自身进程设置硬件断点，才能达到全局硬断的效果

实际效果



我这里是hook了微信的接收消息，但是由于不能使用`OutputDebugStringA`函数，所以用读写文本的方式来输出

log.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)


```
11<?xml version="1.0"?>
<msg>
	<img aeskey="ef0a2dcebd0675985c008d5093b63d54" encryver="0" cdr
N□sYupS 1\11<msg><emoji fromusername="wxid_178580785821 1" tousername=
lvzPFFDXeG57GPbu0Fq06D92/0" designerid="" t賬ag;<?xml version="1.0"?>
<msg>
	<img aeskey="e303c7617f0acf5d783e85a32e6325b1" encryver="0" cdn
a:0_0" md5="30fa661f5d8a26cf117ebe43c6362052" len = "829293" productid
304283f7d64968320c5eef" len = "367503" productid="" androidmd5="314f04
03063323730363632376630646536323334623566303930303030313036&amp
<msg>
	<appmsg appid="wx9365521986f876dd" sdkver="0">
		<title>.^□b箕 N□N□ 66CQhQ齎骕胈迦/ 骕胈XN□ 蕲湇詮:W□ </title>
		<des>66CQhQ齎骕胈8n□ 厤□Y弝)R蕲湇l{`Oeg鑅□ ^ </des>
		<action />
		<type>36</type>
		<showtype>0</showtype>
		<soundtype>0</soundtype>
		<mediatagname />
```


第 1 行，第 1 列 100% Unix (LF) ANSI

CSDN @鬼手56

小结

利用这种方式可以实现在不破坏代码的前提下进行hook，可以完美的避开crc检测，不容易被分析人员发现。由于是在异常处理函数中实现的hook逻辑，还能顺手给微信加一个反调试，防止别人逆向自己写的程序。

附上完整工程链接：
https://download.csdn.net/download/qq_38474570/24415530?spm=1001.2014.3001.5503

关于veh hook的对抗

既然这种方式那么隐蔽，那么假如我们调试的程序采用了类似的hook或者反调试手段，应该怎么处理呢？实际上处理的方式有两种

1. 再编写一个VEH异常处理函数。veh是异常处理链，系统每次都先调用最顶层的那个，再根据最顶层那个的返回值来决定是否调下一个。我们只要再注册一个异常处理函数，返回处理成功不调用下一个，就能把之前的veh顶下去
2. OD设置系统断点断下，再下断 `AddVectoredExceptionHandler`

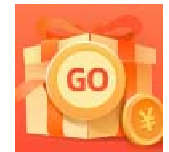
参考文章

xxxx(六)：吊打CRC32检测的无痕hook：https://www.cnblogs.com/theseventhson/p/14399097.html

SEH + 硬件断点HOOK：https://blog.csdn.net/weixin_42052102/article/details/83719791

关于VEH+硬件断点打内存补丁：https://bbs.pediy.com/thread-154035.htm

https://stackoverflow.com/questions/25634376/why-does-addvectoredexceptionhandler-crash-my-dll



创作打卡挑战赛

赢取流量/现金/CSDN周边激励大奖