minhook源码阅读分析

wonderkun / 2019-03-23 09:01:00 / 浏览数 4505 安全工具 工具 顶(0) 踩(0)

---

minhook源码阅读分析

minhook是一个inline
Hook的库，同时支持x32和x64系统，并且是开源的，地址在这里https://www.codeproject.com/Articles/44326/MinHook-The-Minimalistic-x-x-API-Hooking-Libra。

0x1 调用实例

首先看一下官网上给出的c的调用的例子：

```
#include <Windows.h>
#include "./include/MinHook.h"


typedef int (WINAPI *MESSAGEBOXW)(HWND, LPCWSTR, LPCWSTR, UINT);

// Pointer for calling original MessageBoxW.
MESSAGEBOXW fpMessageBoxW = NULL;

// Detour function which overrides MessageBoxW.
int WINAPI DetourMessageBoxW(HWND hWnd, LPCWSTR lpText, LPCWSTR lpCaption, UINT uType)
{
   return fpMessageBoxW(hWnd, L"Hooked!", lpCaption, uType);
}

int main()
{
   // Initialize MinHook.
   if (MH_Initialize() != MH_OK)
   {
       return 1;
   }
   // Create a hook for MessageBoxW, in disabled state.
   if (MH_CreateHook(&MessageBoxW, &DetourMessageBoxW,
       reinterpret_cast<LPVOID*>(&fpMessageBoxW)) != MH_OK)
   {
       return 1;
   }
   // or you can use the new helper function like this.
   //if (MH_CreateHookApiEx(
   //    L"user32", "MessageBoxW", &DetourMessageBoxW, &fpMessageBoxW) != MH_OK)
   //{
   //    return 1;
   //}
   // Enable the hook for MessageBoxW.
   if (MH_EnableHook(&MessageBoxW) != MH_OK)
   {
       return 1;
   }

   // Expected to tell "Hooked!".
   MessageBoxW(NULL, L"Not hooked...", L"MinHook Sample", MB_OK);

   // Disable the hook for MessageBoxW.
   if (MH_DisableHook(&MessageBoxW) != MH_OK)
   {
       return 1;
   }
   // Expected to tell "Not hooked...".
   MessageBoxW(NULL, L"Not hooked...", L"MinHook Sample", MB_OK);

   // Uninitialize MinHook.
   if (MH_Uninitialize() != MH_OK)
   {
```

```
        return 1;
    }
    return 0;
}
```

0x2 初始化钩子的过程

根据这个调用流程跟踪一下源代码，首先看MH_Initialize函数，此函数就干了一件事情，初始化了一个大小自增长的堆，并将堆的句柄存储在全局变量g_hHeap中。

```
g_hHeap = HeapCreate(0, 0, 0);
```

接下来就是创建hook的过程了，这里需要注意几个结构体：

```
struct
{
    PHOOK_ENTRY pItems;     // Data heap
    UINT        capacity;   // Size of allocated data heap, items
    UINT        size;       // Actual number of data items
} g_hooks;
```

g_hooks是一个全局变量，此结构体存储了当前创建的所有钩子，每个钩子的信息都存在了pItems这个指针里。PHOOK_ENTRY结构体的定义如下：

```
typedef struct _HOOK_ENTRY
{
    LPVOID pTarget;             // Address of the target function.
    LPVOID pDetour;             // Address of the detour or relay function.
    LPVOID pTrampoline;         // Address of the trampoline function.
    UINT8  backup[8];           // Original prologue of the target function.

    UINT8  patchAbove  : 1;     // Uses the hot patch area.
    UINT8  isEnabled   : 1;     // Enabled.
    UINT8  queueEnable : 1;     // Queued for enabling/disabling when != isEnabled.

    UINT   nIP : 4;             // Count of the instruction boundaries.
    UINT8  oldIPs[8];           // Instruction boundaries of the target function.
    UINT8  newIPs[8];           // Instruction boundaries of the trampoline function.
} HOOK_ENTRY, *PHOOK_ENTRY;
```

pTarget存储了被hook的函数的地址，pDetour是你写的假的函数的地址，pTrampoline是一个中间的跳转函数，一会再细说。backup[8]是对被Hook函数的前五字节的

接下来调用MH_CreateHook函数，在这个函数里面，首先调用FindHookEntry查找g_hooks中是否已经存放了被hook的目标，如果不存在，就进入创建一个_HOOK_ENT

```
static UINT FindHookEntry(LPVOID pTarget)
{
    UINT i;
    for (i = 0; i < g_hooks.size; ++i)
    {
        if ((ULONG_PTR)pTarget == (ULONG_PTR)g_hooks.pItems[i].pTarget)
            return i;
    }
    return INVALID_HOOK_POS;
}
```

但是在初始化_HOOK_ENTRY之前先要初始化一个_TRAMPOLINE，这部分是minHook的关键,结构体定义如下：

```
typedef struct _TRAMPOLINE
{
    LPVOID pTarget;         // [In] Address of the target function.
    LPVOID pDetour;         // [In] Address of the detour function.
    LPVOID pTrampoline;     // [In] Buffer address for the trampoline and relay function.

#if defined(_M_X64) || defined(__x86_64__)
    LPVOID pRelay;          // [Out] Address of the relay function.
#endif
    BOOL   patchAbove;      // [Out] Should use the hot patch area?
    UINT   nIP;             // [Out] Number of the instruction boundaries.
    UINT8  oldIPs[8];       // [Out] Instruction boundaries of the target function.
    UINT8  newIPs[8];       // [Out] Instruction boundaries of the trampoline function.
} TRAMPOLINE, *PTRAMPOLINE;
```

这个结构体其他部分的定义跟_HOOK_ENTRY结构体一毛一样，但是这里有一个初始化pTrampoline指针的函数AllocateBuffer，此函数中核心逻辑在GetMemoryBloc

```
while ((ULONG_PTR)pAlloc >= minAddr)
       {
           pAlloc = FindPrevFreeRegion(pAlloc, (LPVOID)minAddr, si.dwAllocationGranularity); // ■■■■■■■■■■■■■■free■■■■■■
           if (pAlloc == NULL)
               break;

           pBlock = (PMEMORY_BLOCK)VirtualAlloc(
               pAlloc, MEMORY_BLOCK_SIZE, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
           if (pBlock != NULL)
               break;
       }
   }

   // Alloc a new block below if not found.
   if (pBlock == NULL)
   {
       LPVOID pAlloc = pOrigin;
       while ((ULONG_PTR)pAlloc <= maxAddr)
       {
           pAlloc = FindNextFreeRegion(pAlloc, (LPVOID)maxAddr, si.dwAllocationGranularity);
           if (pAlloc == NULL)
               break;

           pBlock = (PMEMORY_BLOCK)VirtualAlloc(
               pAlloc, MEMORY_BLOCK_SIZE, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
           if (pBlock != NULL)
               break;
       }
   }
```

大意是在被Hook函数的左右512M空间找找到处于空闲状态的内存空间，并返回其地址。接下来就是初始化TRAMPOLINE结构体的函数CreateTrampolineFunction，此

1. 将被Hook的函数的前五个字节放置在pTrampoline指向的buffer中，创建中间函数。也就是我们自己定义函数指针fpMessageBoxW的函数体。但是在拷贝的时候，比

```
do
   {
       HDE       hs;
       UINT      copySize;
       LPVOID    pCopySrc;
       ULONG_PTR pOldInst = (ULONG_PTR)ct->pTarget     + oldPos;
       ULONG_PTR pNewInst = (ULONG_PTR)ct->pTrampoline + newPos;

       copySize = HDE_DISASM((LPVOID)pOldInst, &hs); //■■■■■■■■■■
       if (hs.flags & F_ERROR)
           return FALSE;

       pCopySrc = (LPVOID)pOldInst;
       if (oldPos >= sizeof(JMP_REL))
       {
           // The trampoline function is long enough.
           // Complete the function with the jump to the target function.
#if defined(_M_X64) || defined(__x86_64__)
           jmp.address = pOldInst; // x64■■■■ 0xFF25 disp64■■jmp
#else
           jmp.operand = (UINT32)(pOldInst - (pNewInst + sizeof(jmp)));
#endif
           pCopySrc = &jmp;
           copySize = sizeof(jmp);

           finished = TRUE;
       }
#if defined(_M_X64) || defined(__x86_64__)
       else if ((hs.modrm & 0xC7) == 0x05) // ■x64■■■■■■ [rip+disp32] ■■■■■■■
       {
           // Instructions using RIP relative addressing. (ModR/M = 00???101B)
           // ■■RIP■■■■■■
           // Modify the RIP relative address.
           PUINT32 pRelAddr;

           // Avoid using memcpy to reduce the footprint.
```

```
#ifndef _MSC_VER
            memcpy(instBuf, (LPBYTE)pOldInst, copySize);
#else
            __movsb(instBuf, (LPBYTE)pOldInst, copySize);
#endif
            pCopySrc = instBuf;
            // Relative address is stored at (instruction length - immediate value length - 4).
            pRelAddr = (PUINT32)(instBuf + hs.len - ((hs.flags & 0x3C) >> 2) - 4);
            *pRelAddr
                = (UINT32)((pOldInst + hs.len + (INT32)hs.disp.disp32) - (pNewInst + hs.len));

            // ■■■■■■■■
            // Complete the function if JMP (FF /4).
            if (hs.opcode == 0xFF && hs.modrm_reg == 4)
                finished = TRUE;
        }
#endif
        else if (hs.opcode == 0xE8) // ■■■call■■
        {
            // Direct relative CALL
            ULONG_PTR dest = pOldInst + hs.len + (INT32)hs.imm.imm32; //call ■■■■■■■■■
#if defined(_M_X64) || defined(__x86_64__)
            call.address = dest;
#else
            call.operand = (UINT32)(dest - (pNewInst + sizeof(call))); // ■■call■■■■■■■■■
#endif
            pCopySrc = &call;
            copySize = sizeof(call);
        }
        else if ((hs.opcode & 0xFD) == 0xE9) // ■■■jmp
        {
            // Direct relative JMP (EB or E9)
            ULONG_PTR dest = pOldInst + hs.len;

            if (hs.opcode == 0xEB) // isShort jmp
                dest += (INT8)hs.imm.imm8;
            else
                dest += (INT32)hs.imm.imm32;

            // Simply copy an internal jump.
            if ((ULONG_PTR)ct->pTarget <= dest
                && dest < ((ULONG_PTR)ct->pTarget + sizeof(JMP_REL)))
            {
                if (jmpDest < dest)
                    jmpDest = dest;
            }
            else
            {
#if defined(_M_X64) || defined(__x86_64__)
                jmp.address = dest;
#else
                jmp.operand = (UINT32)(dest - (pNewInst + sizeof(jmp)));
#endif
                pCopySrc = &jmp;
                copySize = sizeof(jmp);

                // Exit the function If it is not in the branch
                finished = (pOldInst >= jmpDest);
            }
        }
        else if ((hs.opcode & 0xF0) == 0x70
            || (hs.opcode & 0xFC) == 0xE0
            || (hs.opcode2 & 0xF0) == 0x80)
        {
            // Direct relative Jcc
            ULONG_PTR dest = pOldInst + hs.len;

            if ((hs.opcode & 0xF0) == 0x70      // Jcc
                || (hs.opcode & 0xFC) == 0xE0)  // LOOPNZ/LOOPZ/LOOP/JECXZ
```

```
                dest += (INT8)hs.imm.imm8;
            else
                dest += (INT32)hs.imm.imm32;

            // Simply copy an internal jump.
            if ((ULONG_PTR)ct->pTarget <= dest
                && dest < ((ULONG_PTR)ct->pTarget + sizeof(JMP_REL)))
            {
                if (jmpDest < dest)
                    jmpDest = dest;
            }
            else if ((hs.opcode & 0xFC) == 0xE0)
            {
                // LOOPNZ/LOOPZ/LOOP/JCXZ/JECXZ to the outside are not supported.
                return FALSE;
            }
            else
            {
                UINT8 cond = ((hs.opcode != 0x0F ? hs.opcode : hs.opcode2) & 0x0F);
#if defined(_M_X64) || defined(__x86_64__)
                // Invert the condition in x64 mode to simplify the conditional jump logic.
                jcc.opcode  = 0x71 ^ cond;
                jcc.address = dest;
#else
                jcc.opcode1 = 0x80 | cond;
                jcc.operand = (UINT32)(dest - (pNewInst + sizeof(jcc)));
#endif
                pCopySrc = &jcc;
                copySize = sizeof(jcc);
            }
        }
        else if ((hs.opcode & 0xFE) == 0xC2)
        {
            // RET (C2 or C3)

            // Complete the function if not in a branch.
            finished = (pOldInst >= jmpDest);
        }

        // Can't alter the instruction length in a branch.
        if (pOldInst < jmpDest && copySize != hs.len)
            return FALSE;

        // Trampoline function is too large.
        if ((newPos + copySize) > TRAMPOLINE_MAX_SIZE)
            return FALSE;

        // Trampoline function has too many instructions.
        if (ct->nIP >= ARRAYSIZE(ct->oldIPs))
            return FALSE;

        ct->oldIPs[ct->nIP] = oldPos;
        ct->newIPs[ct->nIP] = newPos;
        ct->nIP++;

        // Avoid using memcpy to reduce the footprint.
#ifndef _MSC_VER
        memcpy((LPBYTE)ct->pTrampoline + newPos, pCopySrc, copySize);
#else
        __movsb((LPBYTE)ct->pTrampoline + newPos,(LPBYTE)pCopySrc, copySize);
#endif
        newPos += copySize;
        oldPos += hs.len;
    }
    while (!finished);
```

1. 接下来就是还需要在pTrampoline的末尾写上一个长跳转指令，跳转到被Hook函数的指定位置开始执行(注意不是被Hook函数的开始，因为被Hook函数的开始部分已经

```
JMP_ABS jmp = {
       0xFF, 0x25, 0x00000000, // FF25 00000000: JMP [RIP+6]
       0x0000000000000000ULL   // Absolute destination address
   };
   // 0xff25■■■■■■

if (oldPos >= sizeof(JMP_REL))
       {
           // The trampoline function is long enough.
           // Complete the function with the jump to the target function.
#if defined(_M_X64) || defined(__x86_64__)
           jmp.address = pOldInst; // x64■■■■ 0xFF25 disp64■■jmp
#else
           jmp.operand = (UINT32)(pOldInst - (pNewInst + sizeof(jmp)));
#endif
           pCopySrc = &jmp;
           copySize = sizeof(jmp);
           finished = TRUE;
}
```

0x3 安装钩子

钩子函数已经初始化成功了，接下来就需要开始安装了，调用MH_EnableHook函数。核心操作在函数EnableHookLL中：

```
static MH_STATUS EnableHookLL(UINT pos, BOOL enable)
{
   PHOOK_ENTRY pHook = &g_hooks.pItems[pos];
   DWORD  oldProtect;
   SIZE_T patchSize    = sizeof(JMP_REL);
   LPBYTE pPatchTarget = (LPBYTE)pHook->pTarget;

   if (pHook->patchAbove)
   {
       pPatchTarget -= sizeof(JMP_REL);
       patchSize    += sizeof(JMP_REL_SHORT);
   }

   if (!VirtualProtect(pPatchTarget, patchSize, PAGE_EXECUTE_READWRITE, &oldProtect))
       return MH_ERROR_MEMORY_PROTECT;

   if (enable)
   {
       PJMP_REL pJmp = (PJMP_REL)pPatchTarget;
       pJmp->opcode = 0xE9;
       pJmp->operand = (UINT32)((LPBYTE)pHook->pDetour - (pPatchTarget + sizeof(JMP_REL)));

       if (pHook->patchAbove)
       {
           PJMP_REL_SHORT pShortJmp = (PJMP_REL_SHORT)pHook->pTarget;
           pShortJmp->opcode = 0xEB;
           pShortJmp->operand = (UINT8)(0 - (sizeof(JMP_REL_SHORT) + sizeof(JMP_REL)));
       }
   }
   else
   {
       if (pHook->patchAbove)
           memcpy(pPatchTarget, pHook->backup, sizeof(JMP_REL) + sizeof(JMP_REL_SHORT));
       else
           memcpy(pPatchTarget, pHook->backup, sizeof(JMP_REL));
   }

   VirtualProtect(pPatchTarget, patchSize, oldProtect, &oldProtect);

   // Just-in-case measure.
   FlushInstructionCache(GetCurrentProcess(), pPatchTarget, patchSize);

   pHook->isEnabled   = enable;
   pHook->queueEnable = enable;
```

```
   return MH_OK;
}
```

核心代码就下面三行：

```
PJMP_REL pJmp = (PJMP_REL)pPatchTarget;
       pJmp->opcode = 0xE9;
       pJmp->operand = (UINT32)((LPBYTE)pHook->pDetour - (pPatchTarget + sizeof(JMP_REL)));
```

在被Hook的函数的前五个字节写上0xe9+■■■■,跳转到我们创建假的函数地址的位置。

但是再执行EnableHookLL还要执行一个操作，就是先暂停本进程出去本线程之外的所有线程,调用freeze函数实现操作：

```
static VOID Freeze(PFROZEN_THREADS pThreads, UINT pos, UINT action)
{
   pThreads->pItems   = NULL;
   pThreads->capacity = 0;
   pThreads->size     = 0;
   EnumerateThreads(pThreads);

   if (pThreads->pItems != NULL)
   {
       UINT i;
       for (i = 0; i < pThreads->size; ++i)
       {
           HANDLE hThread = OpenThread(THREAD_ACCESS, FALSE, pThreads->pItems[i]);
           if (hThread != NULL)
           {
               SuspendThread(hThread);
               ProcessThreadIPs(hThread, pos, action);
               CloseHandle(hThread);
           }
       }
   }
}
```

跟踪一下ProcessThreadIPs函数的操作：

```
static void ProcessThreadIPs(HANDLE hThread, UINT pos, UINT action)
{
   // If the thread suspended in the overwritten area,
   // move IP to the proper address.

   CONTEXT c;
#if defined(_M_X64) || defined(__x86_64__)
   DWORD64 *pIP = &c.Rip;
#else
   DWORD   *pIP = &c.Eip;
#endif
   UINT count;

   c.ContextFlags = CONTEXT_CONTROL;
   if (!GetThreadContext(hThread, &c))
       return;

   if (pos == ALL_HOOKS_POS)
   {
       pos = 0;
       count = g_hooks.size;
   }
   else
   {
       count = pos + 1;
   }

   for (; pos < count; ++pos)
   {
       PHOOK_ENTRY pHook = &g_hooks.pItems[pos];
       BOOL        enable;
       DWORD_PTR   ip;
```

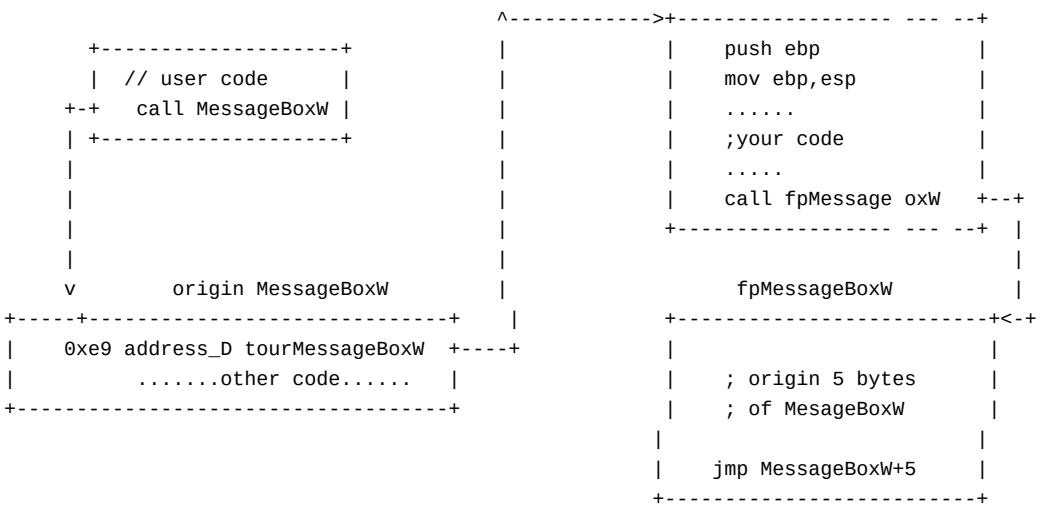
```
        switch (action)
        {
        case ACTION_DISABLE:
            enable = FALSE;
            break;

        case ACTION_ENABLE:
            enable = TRUE;
            break;

        default: // ACTION_APPLY_QUEUED
            enable = pHook->queueEnable;
            break;
        }
        if (pHook->isEnabled == enable)
            continue;

        if (enable)
            ip = FindNewIP(pHook, *pIP);
        else
            ip = FindOldIP(pHook, *pIP);
        if (ip != 0)
        {
            *pIP = ip;
            SetThreadContext(hThread, &c);
        }
    }
}
```

emm，这里直接修改了其他线程的Eip，操作有点秀啊。。。。。

接下来就是恢复线程的操作了,不在细说。

0x4 Hook之后的调用过程

就以实例代码中的Hook`MessageBoxW`的调用过程为例，以下图展示：



0x5 需要改进的地方

因为想做不被执行程序感知的Hook，这里明显的问题是，被Hook的系统API的第一条指令都是`0xe9...`很容易被发现。另外一个问题是这里没有对栈做处理，导致也可以通`stack`技巧轻易发现API被Hook过。

所以接下来的工作就是修改这个两个地方。

上一篇：How To Bypass AMS... 下一篇：跨域方式及其产生的安全问题
1. 0 条回复

- 动动手指，沙发就是你的了！

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板