SIG^2 Secure Code Study Project Proof-Of-Concept   
  
Dynamic Forking of Win32 EXE  
by Tan Chew Keong  
7 April 2004   
Download   
  
Introduction  
  
This Proof-Of-Concept (POC) code demonstrates the dynamic loading of a Win32 EXE into the memory space of a process that was created using the CreateProcess API with the CREATE\_SUSPENDED parameter. This code also shows how to perform manual relocation of a Win32 EXE and how to unmap the original image of an EXE from its process space.   
  
Description of Technique  
  
Under Windows, a process can be created in suspend mode using the CreateProcess API with the CREATE\_SUSPENDED parameter. The EXE image will be loaded into memory by Windows but execution will not begin until the ResumeThread API is used. Before calling ResumeThread, it is possible to read and write this process&#39;s memory space using APIs like ReadProcessMemory and WriteProcessMemory. This makes it possible to overwrite the image of the original EXE with the image of another EXE, thus enabling the execution of the second EXE within the memory space of the first EXE. This can be achieved with the following sequence of steps.   
  
Use the CreateProcess API with the CREATE\_SUSPENDED parameter to create a suspended process from any EXE file. (Call this the first EXE).  
  
Call GetThreadContext API to obtain the register values (thread context) of the suspended process. The EBX register of the suspended process points to the process&#39;s PEB. The EAX register contains the entry point of the process (first EXE).  
  
Obtain the base-address of the suspended process from its PEB, i.e. at [EBX+8]  
  
Load the second EXE into memory (using ReadFile) and perform the neccessary alignment manually. This is required if the file alignment is different from the memory alignment  
  
If the second EXE has the same base-address as the suspended process and its image-size is <= to the image-size of the suspended process, simply use the WriteProcessMemory function to write the image of the second EXE into the memory space of the suspended process, starting at the base-address.  
  
Otherwise, unmap the image of the first EXE using ZwUnmapViewOfSection (exported by ntdll.dll) and use VirtualAllocEx to allocate enough memory for the second EXE within the memory space of the suspended process. The VirtualAllocEx API must be supplied with the base-address of the second EXE to ensure that Windows will give us memory in the required region. Next, copy the image of the second EXE into the memory space of the suspended process starting at the allocated address (using WriteProcessMemory).  
  
If the unmap operation failed but the second EXE is relocatable (i.e. has a relocation table), then allocate enough memory for the second EXE within the suspended process at any location. Perform manual relocation of the second EXE based on the allocated memory address. Next, copy the relocated EXE into the memory space of the suspended process starting at the allocated address (using WriteProcessMemory).  
  
Patch the base-address of the second EXE into the suspended process&#39;s PEB at [EBX+8].  
  
Set EAX of the thread context to the entry point of the second EXE.  
  
Use the SetThreadContext API to modify the thread context of the suspended process.  
  
Use the ResumeThread API to resume execute of the suspended process.  
  
  
  
Techniques Demonstrated by POC Code  
  
  
  
Manual relocation of an EXE using its Relocation Table.   
Unmapping the image of the original EXE using ZwUnmapViewOfSection.   
Reading and Writing to a process&#39;s memory space using ReadProcessMemory and WriteProcessMemory.   
Changing the base-address of a process by modifying its value in the process&#39;s PEB.   
  
  
Usage  
  
loadEXE.exe <EXE filename>   
  
This POC code will use the CreateProcess API to create a process in suspend mode from calc.exe. It would then load and align the EXE file given by the "EXE filename" commandline parameter. Following this, it would copy the aligned EXE image into calc.exe&#39;s memory space and resume execution.   
  
  
  
  
  
Contacts  
  
For further enquries or to submit malicious code for our analysis, email them to the following.   
Overall-in-charge: Tan Chew Keong

补充：

复制内容到剪贴板

**代码:**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
// loadEXE.cpp : Defines the entry point for the console application.  
//  
// Proof-Of-Concept Code  
// Copyright (c) 2004  
// All rights reserved.  
//  
// Permission is hereby granted, free of charge, to any person obtaining a  
// copy of this software and associated documentation files (the  
// "Software"), to deal in the Software without restriction, including  
// without limitation the rights to use, copy, modify, merge, publish,  
// distribute, and/or sell copies of the Software, and to permit persons  
// to whom the Software is furnished to do so, provided that the above  
// copyright notice(s) and this permission notice appear in all copies of  
// the Software and that both the above copyright notice(s) and this  
// permission notice appear in supporting documentation.  
//  
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS  
// OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF  
// MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT  
// OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR  
// HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL  
// INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING  
// FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,  
// NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION  
// WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.  
//  
// Usage:  
// loadEXE <EXE filename>  
//  
// This will execute calc.exe in suspended mode and replace its image with  
// the new EXE&#39;s image.  The thread is then resumed, thus causing the new EXE to  
// execute within the process space of svchost.exe.  
//  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#include <stdio.h>  
#include <windows.h>  
#include <tlhelp32.h>  
#include <psapi.h>  
  
struct PE\_Header   
{  
  unsigned long signature;  
  unsigned short machine;  
  unsigned short numSections;  
  unsigned long timeDateStamp;  
  unsigned long pointerToSymbolTable;  
  unsigned long numOfSymbols;  
  unsigned short sizeOfOptionHeader;  
  unsigned short characteristics;  
};  
  
struct PE\_ExtHeader  
{  
  unsigned short magic;  
  unsigned char majorLinkerVersion;  
  unsigned char minorLinkerVersion;  
  unsigned long sizeOfCode;  
  unsigned long sizeOfInitializedData;  
  unsigned long sizeOfUninitializedData;  
  unsigned long addressOfEntryPoint;  
  unsigned long baseOfCode;  
  unsigned long baseOfData;  
  unsigned long imageBase;  
  unsigned long sectionAlignment;  
  unsigned long fileAlignment;  
  unsigned short majorOSVersion;  
  unsigned short minorOSVersion;  
  unsigned short majorImageVersion;  
  unsigned short minorImageVersion;  
  unsigned short majorSubsystemVersion;  
  unsigned short minorSubsystemVersion;  
  unsigned long reserved1;  
  unsigned long sizeOfImage;  
  unsigned long sizeOfHeaders;  
  unsigned long checksum;  
  unsigned short subsystem;  
  unsigned short DLLCharacteristics;  
  unsigned long sizeOfStackReserve;  
  unsigned long sizeOfStackCommit;  
  unsigned long sizeOfHeapReserve;  
  unsigned long sizeOfHeapCommit;  
  unsigned long loaderFlags;  
  unsigned long numberOfRVAAndSizes;  
  unsigned long exportTableAddress;  
  unsigned long exportTableSize;  
  unsigned long importTableAddress;  
  unsigned long importTableSize;  
  unsigned long resourceTableAddress;  
  unsigned long resourceTableSize;  
  unsigned long exceptionTableAddress;  
  unsigned long exceptionTableSize;  
  unsigned long certFilePointer;  
  unsigned long certTableSize;  
  unsigned long relocationTableAddress;  
  unsigned long relocationTableSize;  
  unsigned long debugDataAddress;  
  unsigned long debugDataSize;  
  unsigned long archDataAddress;  
  unsigned long archDataSize;  
  unsigned long globalPtrAddress;  
  unsigned long globalPtrSize;  
  unsigned long TLSTableAddress;  
  unsigned long TLSTableSize;  
  unsigned long loadConfigTableAddress;  
  unsigned long loadConfigTableSize;  
  unsigned long boundImportTableAddress;  
  unsigned long boundImportTableSize;  
  unsigned long importAddressTableAddress;  
  unsigned long importAddressTableSize;  
  unsigned long delayImportDescAddress;  
  unsigned long delayImportDescSize;  
  unsigned long COMHeaderAddress;  
  unsigned long COMHeaderSize;  
  unsigned long reserved2;  
  unsigned long reserved3;  
};  
  
  
struct SectionHeader  
{  
  unsigned char sectionName[8];  
  unsigned long virtualSize;  
  unsigned long virtualAddress;  
  unsigned long sizeOfRawData;  
  unsigned long pointerToRawData;  
  unsigned long pointerToRelocations;  
  unsigned long pointerToLineNumbers;  
  unsigned short numberOfRelocations;  
  unsigned short numberOfLineNumbers;  
  unsigned long characteristics;  
};  
  
struct MZHeader  
{  
  unsigned short signature;  
  unsigned short partPag;  
  unsigned short pageCnt;  
  unsigned short reloCnt;  
  unsigned short hdrSize;  
  unsigned short minMem;  
  unsigned short maxMem;  
  unsigned short reloSS;  
  unsigned short exeSP;  
  unsigned short chksum;  
  unsigned short exeIP;  
  unsigned short reloCS;  
  unsigned short tablOff;  
  unsigned short overlay;  
  unsigned char reserved[32];  
  unsigned long offsetToPE;  
};  
  
  
struct ImportDirEntry  
{  
  DWORD importLookupTable;  
  DWORD timeDateStamp;  
  DWORD fowarderChain;  
  DWORD nameRVA;  
  DWORD importAddressTable;  
};  
  
  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// This function reads the MZ, PE, PE extended and Section Headers from an EXE file.  
//  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
bool readPEInfo(FILE \*fp, MZHeader \*outMZ, PE\_Header \*outPE, PE\_ExtHeader \*outpeXH,  
        SectionHeader \*\*outSecHdr)  
{  
  fseek(fp, 0, SEEK\_END);  
  long fileSize = ftell(fp);  
  fseek(fp, 0, SEEK\_SET);  
  
  if(fileSize < sizeof(MZHeader))  
  {  
    printf("File size too small\n");      
    return false;  
  }  
  
  // read MZ Header  
  MZHeader mzH;  
  fread(&mzH, sizeof(MZHeader), 1, fp);  
  
  if(mzH.signature != 0x5a4d)    // MZ  
  {  
    printf("File does not have MZ header\n");  
    return false;  
  }  
  
  //printf("Offset to PE Header = %X\n", mzH.offsetToPE);  
  
  if((unsigned long)fileSize < mzH.offsetToPE + sizeof(PE\_Header))  
  {  
    printf("File size too small\n");      
    return false;  
  }  
  
  // read PE Header  
  fseek(fp, mzH.offsetToPE, SEEK\_SET);  
  PE\_Header peH;  
  fread(&peH, sizeof(PE\_Header), 1, fp);  
  
  //printf("Size of option header = %d\n", peH.sizeOfOptionHeader);  
  //printf("Number of sections = %d\n", peH.numSections);  
  
  if(peH.sizeOfOptionHeader != sizeof(PE\_ExtHeader))  
  {  
    printf("Unexpected option header size.\n");  
      
    return false;  
  }  
  
  // read PE Ext Header  
  PE\_ExtHeader peXH;  
  
  fread(&peXH, sizeof(PE\_ExtHeader), 1, fp);  
  
  //printf("Import table address = %X\n", peXH.importTableAddress);  
  //printf("Import table size = %X\n", peXH.importTableSize);  
  //printf("Import address table address = %X\n", peXH.importAddressTableAddress);  
  //printf("Import address table size = %X\n", peXH.importAddressTableSize);  
  
  
  // read the sections  
  SectionHeader \*secHdr = new SectionHeader[peH.numSections];  
  
  fread(secHdr, sizeof(SectionHeader) \* peH.numSections, 1, fp);  
  
  \*outMZ = mzH;  
  \*outPE = peH;  
  \*outpeXH = peXH;  
  \*outSecHdr = secHdr;  
  
  return true;  
}  
  
  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// This function calculates the size required to load an EXE into memory with proper alignment.  
//  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
int calcTotalImageSize(MZHeader \*inMZ, PE\_Header \*inPE, PE\_ExtHeader \*inpeXH,  
             SectionHeader \*inSecHdr)  
{  
  int result = 0;  
  int alignment = inpeXH->sectionAlignment;  
  
  if(inpeXH->sizeOfHeaders % alignment == 0)  
    result += inpeXH->sizeOfHeaders;  
  else  
  {  
    int val = inpeXH->sizeOfHeaders / alignment;  
    val++;  
    result += (val \* alignment);  
  }  
  
  
  for(int i = 0; i < inPE->numSections; i++)  
  {  
    if(inSecHdr[i].virtualSize)  
    {  
      if(inSecHdr[i].virtualSize % alignment == 0)  
        result += inSecHdr[i].virtualSize;  
      else  
      {  
        int val = inSecHdr[i].virtualSize / alignment;  
        val++;  
        result += (val \* alignment);  
      }  
    }  
  }  
  
  return result;  
}  
  
  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// This function calculates the aligned size of a section  
//  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
unsigned long getAlignedSize(unsigned long curSize, unsigned long alignment)  
{    
  if(curSize % alignment == 0)  
    return curSize;  
  else  
  {  
    int val = curSize / alignment;  
    val++;  
    return (val \* alignment);  
  }  
}  
  
  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// This function loads a PE file into memory with proper alignment.  
// Enough memory must be allocated at ptrLoc.  
//  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
bool loadPE(FILE \*fp, MZHeader \*inMZ, PE\_Header \*inPE, PE\_ExtHeader \*inpeXH,  
      SectionHeader \*inSecHdr, LPVOID ptrLoc)  
{  
  char \*outPtr = (char \*)ptrLoc;  
  
  fseek(fp, 0, SEEK\_SET);  
  unsigned long headerSize = inpeXH->sizeOfHeaders;  
  
  // certain PE files have sectionHeaderSize value > size of PE file itself.    
  // this loop handles this situation by find the section that is nearest to the  
  // PE header.  
  
  for(int i = 0; i < inPE->numSections; i++)  
  {  
    if(inSecHdr[i].pointerToRawData < headerSize)  
      headerSize = inSecHdr[i].pointerToRawData;  
  }  
  
  // read the PE header  
  unsigned long readSize = fread(outPtr, 1, headerSize, fp);  
  //printf("HeaderSize = %d\n", headerSize);  
  if(readSize != headerSize)  
  {  
    printf("Error reading headers (%d %d)\n", readSize, headerSize);  
    return false;      
  }  
  
  outPtr += getAlignedSize(inpeXH->sizeOfHeaders, inpeXH->sectionAlignment);  
  
  // read the sections  
  for(i = 0; i < inPE->numSections; i++)  
  {  
    if(inSecHdr[i].sizeOfRawData > 0)  
    {  
      unsigned long toRead = inSecHdr[i].sizeOfRawData;  
      if(toRead > inSecHdr[i].virtualSize)  
        toRead = inSecHdr[i].virtualSize;  
  
      fseek(fp, inSecHdr[i].pointerToRawData, SEEK\_SET);  
      readSize = fread(outPtr, 1, toRead, fp);  
  
      if(readSize != toRead)  
      {  
        printf("Error reading section %d\n", i);  
        return false;  
      }  
      outPtr += getAlignedSize(inSecHdr[i].virtualSize, inpeXH->sectionAlignment);  
    }  
    else  
    {  
      // this handles the case where the PE file has an empty section. E.g. UPX0 section  
      // in UPXed files.  
  
      if(inSecHdr[i].virtualSize)  
        outPtr += getAlignedSize(inSecHdr[i].virtualSize, inpeXH->sectionAlignment);  
    }  
  }  
  
  return true;  
}  
  
  
struct FixupBlock  
{  
  unsigned long pageRVA;  
  unsigned long blockSize;  
};  
  
  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// This function loads a PE file into memory with proper alignment.  
// Enough memory must be allocated at ptrLoc.  
//  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
void doRelocation(MZHeader \*inMZ, PE\_Header \*inPE, PE\_ExtHeader \*inpeXH,  
          SectionHeader \*inSecHdr, LPVOID ptrLoc, DWORD newBase)  
{  
  if(inpeXH->relocationTableAddress && inpeXH->relocationTableSize)  
  {  
    FixupBlock \*fixBlk = (FixupBlock \*)((char \*)ptrLoc + inpeXH->relocationTableAddress);  
    long delta = newBase - inpeXH->imageBase;  
  
    while(fixBlk->blockSize)  
    {  
      //printf("Addr = %X\n", fixBlk->pageRVA);  
      //printf("Size = %X\n", fixBlk->blockSize);  
  
      int numEntries = (fixBlk->blockSize - sizeof(FixupBlock)) >> 1;  
      //printf("Num Entries = %d\n", numEntries);  
  
      unsigned short \*offsetPtr = (unsigned short \*)(fixBlk + 1);  
  
      for(int i = 0; i < numEntries; i++)  
      {  
        DWORD \*codeLoc = (DWORD \*)((char \*)ptrLoc + fixBlk->pageRVA + (\*offsetPtr & 0x0FFF));  
          
        int relocType = (\*offsetPtr & 0xF000) >> 12;  
          
        //printf("Val = %X\n", \*offsetPtr);  
        //printf("Type = %X\n", relocType);  
  
        if(relocType == 3)  
          \*codeLoc = ((DWORD)\*codeLoc) + delta;  
        else  
        {  
          printf("Unknown relocation type = %d\n", relocType);  
        }  
        offsetPtr++;  
      }  
  
      fixBlk = (FixupBlock \*)offsetPtr;  
    }  
  }    
}  
  
  
#define TARGETPROC "calc.exe"  
  
typedef struct \_PROCINFO  
{  
  DWORD baseAddr;  
  DWORD imageSize;  
} PROCINFO;  
  
  
  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// Creates the original EXE in suspended mode and returns its info in the PROCINFO structure.  
//  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
  
BOOL createChild(PPROCESS\_INFORMATION pi, PCONTEXT ctx, PROCINFO \*outChildProcInfo)  
{  
  STARTUPINFO si = {0};  
  
  if(CreateProcess(NULL, TARGETPROC,  
             NULL, NULL, 0, CREATE\_SUSPENDED, NULL, NULL, &si, pi))      
  {  
    ctx->ContextFlags=CONTEXT\_FULL;  
    GetThreadContext(pi->hThread, ctx);  
  
    DWORD \*pebInfo = (DWORD \*)ctx->Ebx;  
    DWORD read;  
    ReadProcessMemory(pi->hProcess, &pebInfo[2], (LPVOID)&(outChildProcInfo->baseAddr), sizeof(DWORD), &read);  
    
    DWORD curAddr = outChildProcInfo->baseAddr;  
    MEMORY\_BASIC\_INFORMATION memInfo;  
    while(VirtualQueryEx(pi->hProcess, (LPVOID)curAddr, &memInfo, sizeof(memInfo)))  
    {  
      if(memInfo.State == MEM\_FREE)  
        break;  
      curAddr += memInfo.RegionSize;  
    }  
    outChildProcInfo->imageSize = (DWORD)curAddr - (DWORD)outChildProcInfo->baseAddr;  
  
    return TRUE;  
  }  
  return FALSE;  
}  
  
  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// Returns true if the PE file has a relocation table  
//  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
BOOL hasRelocationTable(PE\_ExtHeader \*inpeXH)  
{  
  if(inpeXH->relocationTableAddress && inpeXH->relocationTableSize)  
  {  
    return TRUE;  
  }  
  return FALSE;  
}  
  
  
typedef DWORD (WINAPI \*PTRZwUnmapViewOfSection)(IN HANDLE ProcessHandle, IN PVOID BaseAddress);  
  
  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
//  
// To replace the original EXE with another one we do the following.  
// 1) Create the original EXE process in suspended mode.  
// 2) Unmap the image of the original EXE.  
// 3) Allocate memory at the baseaddress of the new EXE.  
// 4) Load the new EXE image into the allocated memory.    
// 5) Windows will do the necessary imports and load the required DLLs for us when we resume the suspended   
//   thread.  
//  
// When the original EXE process is created in suspend mode, GetThreadContext returns these useful  
// register values.  
// EAX - process entry point  
// EBX - points to PEB  
//  
// So before resuming the suspended thread, we need to set EAX of the context to the entry point of the  
// new EXE.  
//  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
void doFork(MZHeader \*inMZ, PE\_Header \*inPE, PE\_ExtHeader \*inpeXH,  
      SectionHeader \*inSecHdr, LPVOID ptrLoc, DWORD imageSize)  
{  
  STARTUPINFO si = {0};  
  PROCESS\_INFORMATION pi;  
  CONTEXT ctx;  
  PROCINFO childInfo;  
    
  if(createChild(&pi, &ctx, &childInfo))   
  {      
    printf("Original EXE loaded (PID = %d).\n", pi.dwProcessId);  
    printf("Original Base Addr = %X, Size = %X\n", childInfo.baseAddr, childInfo.imageSize);  
      
    LPVOID v = (LPVOID)NULL;  
      
    if(inpeXH->imageBase == childInfo.baseAddr && imageSize <= childInfo.imageSize)  
    {  
      // if new EXE has same baseaddr and is its size is <= to the original EXE, just  
      // overwrite it in memory  
      v = (LPVOID)childInfo.baseAddr;  
      DWORD oldProtect;  
      VirtualProtectEx(pi.hProcess, (LPVOID)childInfo.baseAddr, childInfo.imageSize, PAGE\_EXECUTE\_READWRITE, &oldProtect);        
        
      printf("Using Existing Mem for New EXE at %X\n", (unsigned long)v);  
    }  
    else  
    {  
      // get address of ZwUnmapViewOfSection  
      PTRZwUnmapViewOfSection pZwUnmapViewOfSection = (PTRZwUnmapViewOfSection)GetProcAddress(GetModuleHandle("ntdll.dll"), "ZwUnmapViewOfSection");  
  
      // try to unmap the original EXE image  
      if(pZwUnmapViewOfSection(pi.hProcess, (LPVOID)childInfo.baseAddr) == 0)  
      {  
        // allocate memory for the new EXE image at the prefered imagebase.  
        v = VirtualAllocEx(pi.hProcess, (LPVOID)inpeXH->imageBase, imageSize, MEM\_RESERVE | MEM\_COMMIT, PAGE\_EXECUTE\_READWRITE);  
        if(v)  
          printf("Unmapped and Allocated Mem for New EXE at %X\n", (unsigned long)v);  
      }  
    }  
  
    if(!v && hasRelocationTable(inpeXH))  
    {  
      // if unmap failed but EXE is relocatable, then we try to load the EXE at another  
      // location  
      v = VirtualAllocEx(pi.hProcess, (void \*)NULL, imageSize, MEM\_RESERVE | MEM\_COMMIT, PAGE\_EXECUTE\_READWRITE);  
      if(v)  
      {  
        printf("Allocated Mem for New EXE at %X. EXE will be relocated.\n", (unsigned long)v);  
  
        // we&#39;ve got to do the relocation ourself if we load the image at another  
        // memory location          
        doRelocation(inMZ, inPE, inpeXH, inSecHdr, ptrLoc, (DWORD)v);  
      }  
    }  
  
    printf("EIP = %X\n", ctx.Eip);  
    printf("EAX = %X\n", ctx.Eax);  
    printf("EBX = %X\n", ctx.Ebx);    // EBX points to PEB  
    printf("ECX = %X\n", ctx.Ecx);  
    printf("EDX = %X\n", ctx.Edx);  
      
    if(v)  
    {        
      printf("New EXE Image Size = %X\n", imageSize);  
        
      // patch the EXE base addr in PEB (PEB + 8 holds process base addr)  
      DWORD \*pebInfo = (DWORD \*)ctx.Ebx;  
      DWORD wrote;              
      WriteProcessMemory(pi.hProcess, &pebInfo[2], &v, sizeof(DWORD), &wrote);  
  
      // patch the base addr in the PE header of the EXE that we load ourselves  
      PE\_ExtHeader \*peXH = (PE\_ExtHeader \*)((DWORD)inMZ->offsetToPE + sizeof(PE\_Header) + (DWORD)ptrLoc);  
      peXH->imageBase = (DWORD)v;  
        
      if(WriteProcessMemory(pi.hProcess, v, ptrLoc, imageSize, NULL))  
      {    
        printf("New EXE image injected into process.\n");  
  
        ctx.ContextFlags=CONTEXT\_FULL;          
        //ctx.Eip = (DWORD)v + ((DWORD)dllLoaderWritePtr - (DWORD)ptrLoc);  
          
        if((DWORD)v == childInfo.baseAddr)  
        {  
          ctx.Eax = (DWORD)inpeXH->imageBase + inpeXH->addressOfEntryPoint;    // eax holds new entry point  
        }  
        else  
        {  
          // in this case, the DLL was not loaded at the baseaddr, i.e. manual relocation was  
          // performed.  
          ctx.Eax = (DWORD)v + inpeXH->addressOfEntryPoint;    // eax holds new entry point  
        }  
  
        printf("\*\*\*\*\*\*\*\*> EIP = %X\n", ctx.Eip);  
        printf("\*\*\*\*\*\*\*\*> EAX = %X\n", ctx.Eax);  
  
        SetThreadContext(pi.hThread,&ctx);  
  
        ResumeThread(pi.hThread);  
        printf("Process resumed (PID = %d).\n", pi.dwProcessId);  
      }  
      else  
      {  
        printf("WriteProcessMemory failed\n");  
        TerminateProcess(pi.hProcess, 0);  
      }  
    }  
    else  
    {  
      printf("Load failed.  Consider making this EXE relocatable.\n");  
      TerminateProcess(pi.hProcess, 0);  
    }  
  }  
  else  
  {  
    printf("Cannot load %s\n", TARGETPROC);  
  }  
}  
  
  
  
  
int main(int argc, char\* argv[])  
{  
  if(argc != 2)  
  {  
    printf("\nUsage: %s <EXE filename>\n", argv[0]);  
    return 1;  
  }  
  
  FILE \*fp = fopen(argv[1], "rb");  
  if(fp)  
  {  
    MZHeader mzH;  
    PE\_Header peH;  
    PE\_ExtHeader peXH;  
    SectionHeader \*secHdr;  
  
    if(readPEInfo(fp, &mzH, &peH, &peXH, &secHdr))  
    {  
      int imageSize = calcTotalImageSize(&mzH, &peH, &peXH, secHdr);  
      //printf("Image Size = %X\n", imageSize);  
  
      LPVOID ptrLoc = VirtualAlloc(NULL, imageSize, MEM\_COMMIT, PAGE\_EXECUTE\_READWRITE);  
      if(ptrLoc)  
      {  
        //printf("Memory allocated at %X\n", ptrLoc);  
        loadPE(fp, &mzH, &peH, &peXH, secHdr, ptrLoc);                          
          
        doFork(&mzH, &peH, &peXH, secHdr, ptrLoc, imageSize);                  
      }  
      else  
        printf("Allocation failed\n");  
    }  
  
    fclose(fp);  
  }  
  else  
    printf("\nCannot open the EXE file!\n");  
  
  return 0;  
}

人情如冰六月