UIC – Università Italiana Cracking

http://quequero.org

How to Hide a DLL

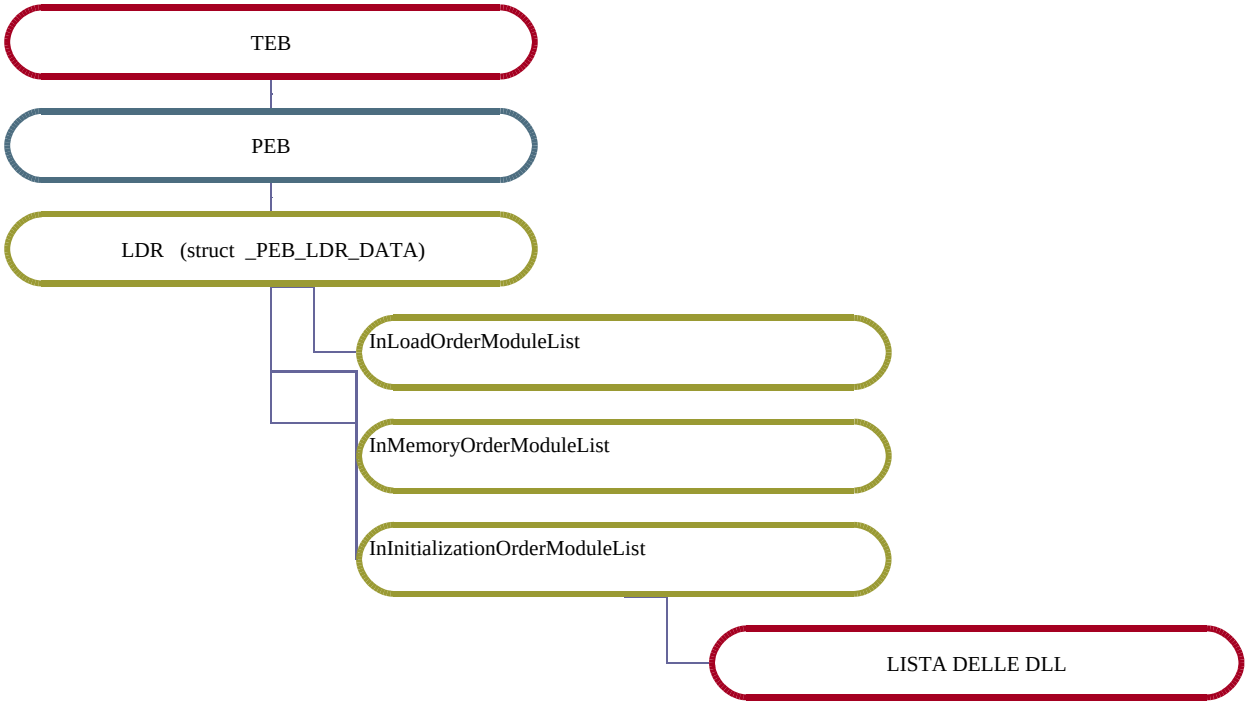
written by Pn[L]uck

UIC New Year Pack 2 – 01/Jan/2007

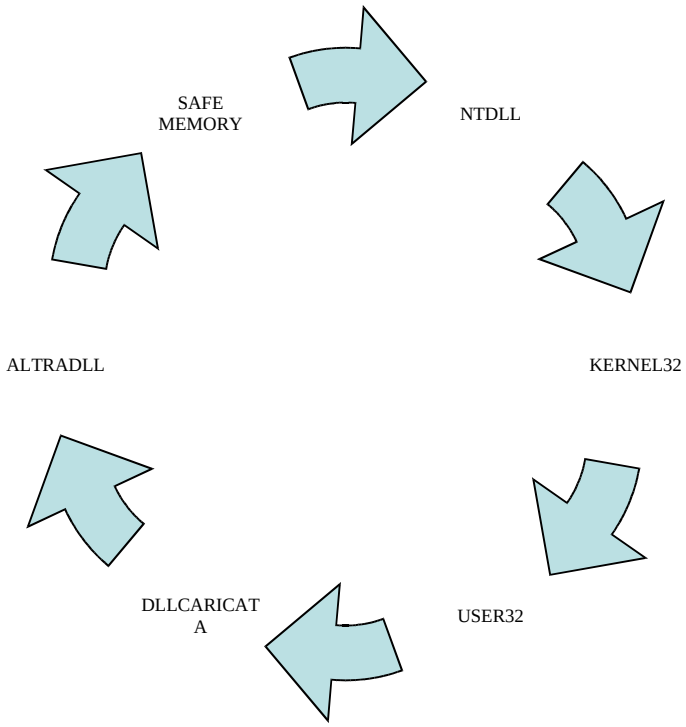
How to Hide a Dll

Abstract

Le informazioni riguardanti le Dll caricate in un processo, sono accessibili attraverso il PEB del processo stesso.
Il PEB, che è residente nello spazio di memoria del programma, e contiene tutte le informazioni del processo: dall'ImageBase al SizeOfImage, dal BeingDebugged alla lista delle Dll caricate nel processo, per l'appunto, nel seguente modo:



_PEB_LDR_DATA, che è un membro della struttura PEB,è una struttura che ha molti membri, ma a noi ne interessano solo tre membri, i quali puntano ad altre strutture (sempre le stesse e collegate tra loro), che contengono le informazioni riguardato le DLL caricate(Nome, ImageBase, etc..), e per ogni DLL caricata, viene creata una nuova struttura.



PS: La struttura, che io ho chiamato SAFE MEMORY, ha solo le tre strutture_LIST_ENTRY settate, mentre tutte gli altri membri sono NULL.

Quando si usano programmi quali Task Explorer o il vecchio LordPe, questi chiamano delle API quali EnumProcessModules o Module32First, che non fanno altre che leggere queste informazioni per ogni processo.

Theory

Come già detto in precedenza, ogni struttura che contiene le informazioni di una Dll (struct _LDR_MODULE) è collegata a quella successiva e precedente, con dei puntatori. Quindi in teoria cambiando i valori di questi puntatori, contenuti nelle strutture _LIST_ENTRY ,si può nascondere una Dll ai process viewer.

Per far questo, ci serve conoscere le diverse strutture, impegnate in questo processo.
Innanzi tutto quella del Peb:

```
#typedef struct _PEB {

  BOOLEAN                 InheritedAddressSpace;

  BOOLEAN                 ReadImageFileExecOptions;

  BOOLEAN                 BeingDebugged;

  BOOLEAN                 Spare;

  HANDLE                  Mutant;

  PVOID                   ImageBaseAddress;

  PEB_LDR_DATA           LoaderData;
```

```
//gli altri membri che non ci servono

}
```

Quindi PEB_LDR_DATA

```
typedef struct _PEB_LDR_DATA
{
         ULONG Length;
         UCHAR Initialized;
         PVOID SsHandle;
         LIST_ENTRY InLoadOrderModuleList;  //<-- Puntore al membro InLoad
         LIST_ENTRY InMemoryOrderModuleList; //<-- Puntore al membro InMem
         LIST_ENTRY InInitializationOrderModuleList; //<-Puntore al membro InIni
         PVOID EntryInProgress;

} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

Come detto precedentemente, i tre membri LIST_ENTRY puntano a delle strutture _LDR_MODULE che contengono tutte le informazioni delle dll caricate, solo che a membri diversi; infatti i puntatori della LIST_ENTRY InLoadOrderModuleList, punteranno al membro InLoadOrderModuleList della struttura precedente o successiva, e così via.

Poi al contrario di InLoadOrderModuleLis ed InMememoryOrderModuleList, che puntano alla struttura del Modulo principale, InInitializationOrderModuleList punta direttamente al membro InInitializationOrderModuleList della prima Dll caricata.

```
typedef struct _LDR_MODULE {
      LIST_ENTRY InLoadOrderModuleList;  //<-- InLoad punta qui
      LIST_ENTRY InMemoryOrderModuleList; //<-- PInMem punta qui
      LIST_ENTRY InInitializationOrderModuleList;  //<-- InInitia punta qui
      PVOID BaseAddress;
      PVOID EntryPoint;
      ULONG SizeOfImage;
      UNICODE_STRING FullDllName;
      UNICODE_STRING BaseDllName;
      ULONG Flags;
      SHORT LoadCount;
      SHORT TlsIndex;
      LIST_ENTRY HashTableEntry;
      ULONG TimeDateStamp;

} LDR_MODULE, *PLDR_MODULE;
```

L’Hook, come già accennato, si basa sulla modifica dei valori della struttura _LIST_ENTRY

```
0:000> dt _list_entry

ntdll!_LIST_ENTRY

   +0x000 Flink            : Ptr32 to _LDR_MODULE

   +0x004 Blink            : Ptr32 to _LDR_MODULE
```

Dove Flink contiene l’address alla struttura successiva e Blink a quella precedente. Quindi modificando le tre _LIST_ENTRY della struttura _LDR_MODULE precedente e successiva, a quella relativa della DLL desiderata, la si nasconde.

Practical

Ora che ho descritto un po' tutto il funzionamento teorico dell'hook e le varie strutture da utilizzare, posso passare alla parte pratica: un po' di codice

```
      ULONG_PTR ldr_addr, var, DLL = 0x76bb0000 ; //DLL = ImageBase della dll da
nascondere
      PEB_LDR_DATA* ldr_data;
      LDR_MODULE  *modulo, *prec, *next; //Modulo è il modulo che nasconderemo

 try
 {
      __asm mov eax, fs:[0x30]  //Ricaviamo l'indirizzo del PEB nel processo
      __asm add eax, 0xc        //PEB->LDR
      __asm mov eax,[eax]
      __asm mov ldr_addr, eax   //salvo l'indirizzo di LDR


       ldr_data = (PEB_LDR_DATA*)ldr_addr ;  //inizializzo la struttura.

       modulo = (LDR_MODULE*)ldr_data->InLoadOrderModuleList.Flink; //1° modulo.
```

Questo codice non fa altro che ricavare l'addr della struttura LDR_DATA, dalla quale ricava l'indirizzo della prima struttura LDR_MODULE, cioè quella riguardande il modulo caricato (l'EXE)

```
while(modulo->BaseAddress != 0) //Listo tutto i moduli fino a SafeMemory
      {
      if( (ULONG_PTR)modulo->BaseAddress == Dll)
            {

            //controllo se è stato passato l'addr del Modulo principale
            if(modulo->InInitializationOrderModuleList.Blink == NULL)
                  throw 0;

            //Ricavo il modulo caricato precedentemente e successivo a questo
            prec = (LDR_MODULE*) (ULONG_PTR) ((ULONG_PTR)modulo-
>InInitializationOrderModuleList.Blink – 16);
            next = (LDR_MODULE*) (ULONG_PTR) ((ULONG_PTR)modulo-
>InInitializationOrderModuleList.Flink – 16);

            //cambio i valori
            prec->InInitializationOrderModuleList.Flink =
                              modulo->InInitializationOrderModuleList.Flink;
                  next->InInitializationOrderModuleList.Blink =
                              modulo->InInitializationOrderModuleList.Blink;

                  //Ricavo le strutture del modulo precedente e successivo
                  prec = (LDR_MODULE*)modulo->InLoadOrderModuleList.Blink;
                  next = (LDR_MODULE*)modulo->InLoadOrderModuleList.Flink;

                  //cambio i valori nel modulo precedente
                  prec->InLoadOrderModuleList.Flink =
                              modulo->InLoadOrderModuleList.Flink;
                  prec->InMemoryOrderModuleList.Flink =
                              modulo->InMemoryOrderModuleList.Flink;

                  //cambio i valodri nel modulo successivo
                  next->InLoadOrderModuleList.Blink =
```

```
                              modulo->InLoadOrderModuleList.Blink;
                    next->InMemoryOrderModuleList.Blink =
                              modulo->InMemoryOrderModuleList.Blink;

                    break;
               }
               modulo = (LDR_MODULE*)modulo->InLoadOrderModuleList.Flink;
          }

 }
 catch(...)
 {
       _tprintf(TEXT("\nError\n"));

}
```

Ecco a voi una funzionce semplice semplice, che permette di nascondere un modulo qualsiasi.

Implementation

Ora vi mostro come implementare questa conoscenza(e che conoscenza hihi) in un crackme o perché no in un programma.

Implementiamo una protezione serial, la quale esegue il controllo in una Dll.

Iniziamo a scrivere un po' di codice, ecco il main:

```
void(WINAPI *CheckSerial)(TCHAR*,TCHAR*);

int main()
{
      TCHAR USER[20];
      TCHAR PSW[20];

      _tprintf(TEXT("\n\nInsert UserName: "));
      _tscanf_s(TEXT("%s"),USER,20);

      _tprintf(TEXT("\n\nInsert Password: "));
      _tscanf_s(TEXT("%s"),PSW,20);

      //carico la dll
      HINSTANCE hinstLib = LoadLibrary(TEXT("CoRegistration.dll"));
      CheckSerial(USER,PSW);

    user_wait();

return 0;

}
```

CheckSerial, è la funzione che si preoccupa di controllare il serial, come ben notate, al momento della compilazione, punta ad una DWORD nulla, infatti la mia intenzione è cambiare quel valore, nel DllMain della nostra dll, in questo modo:

```
BOOL APIENTRY DllMain( HMODULE hModule, DWORD  ul_reason_for_call, LPVOID
lpReserved )
{
```

```
      if(ul_reason_for_call == DLL_PROCESS_ATTACH)
      {
            //mi nascondo
            HideDll((ULONG_PTR)hModule);
            //modifico l'addr della funzione CheckSerial
            //l'addr l'ho ricavato tramite disasm
            FixExeValue(0x40faf8,(ULONG_PTR)&CheckSerial);
    }
    return TRUE;

}
```

Una volta caricata la Dll in memoria, questa si nasconde, e fixa l'address di CheckSerial del programma, con l'address di CheckSerial (&CheckSerial) presente nella Dll, cioè con la vera funzione che controlla la password.

Ecco a voi il Disasm, per rendervi conto della situazione:

```
00401051  PUSH test_hoo.0040D3D8              ; FileName = "CoRegistration.dll"

00401056  CALL DWORD PTR DS:[<&KERNEL32.LoadLibraryW>]

0040105C  LEA EAX,DWORD PTR SS:[EBP-2C]

0040105F  PUSH EAX

00401060  LEA EAX,DWORD PTR SS:[EBP-54]

00401063  PUSH EAX

00401064  CALL DWORD PTR DS:[40FAF8]           ;DS:[40FAF8] = 0
```

Come potete vedere, un reverser noterebbe subito che c'è qualcosa di strano, vista la semplicità del codice: la presenza di un LoadLibrary, vicina ad una Call che putta nel vuoto.

Per rendere l'analisi più ardua ad un Reverser, potremmo mettere il LoadLibrary all'inizio della WinMain, e la funzione di controllo(quella dove vi è la call che punta al vuoto), sparsa nel codice(sto parlando di un progetto ampio e complesso), oppure implementare una Dll-jungle, cioè mettere le finzioni HideDll e FixExeValue, all'interno di dll, che svolge il compito di disegnare l'interfaccia del programma, o con altri compiti. Oppure inserire un LoadLibrary ad una dll tipo CoRegistration, all'interndo di una di queste dll usate per il disegno dell'interfaccia, facendo puntare Winapi *CheckSerial, ad una fakefunction per il controllo del serial (quante belle idee).

Ringraziamenti

Ringrazio Que che mi ha inserito tra gli autori del NYP, Ntoskrnl e Quake per il loro tempo prezioso speso per me, EvilCry perché grazie a lui sto imparando la cryptografia, gli altri del NYP, Deroko che mi ha dato l'idea, Zairon, LordFelix, LonelyWolf, Locu, il mio compaesano Ermes, e tutta la gente di #crack-it e #pmode

Pn[L]uck