

Forum:[Hack Forums](#)

Thread: [\[Tutorial\] \[C#\] Inject PE From Resources](#)

Tsukuyomi - July 11, 2012, 5:59 am

[VB.Net Version of this tutorial](#)

Hello.

In this tutorial we will...Create a simple PE file (C++)

Save that file as a managed resource (C#)

Inject that resource into notepad, forcing it to execute our code.

For this tutorial you need...A C++ Compiler (I will use VS2008)

A C# Compiler (I will use VS2008)

The C++ Part (Creating the PE DLL)

If you for some reason want to skip this(C++) part, here's a precompiled version of the DLL we will be creating:

[ThePE.dll \(Direct Link \)](#)

It's a 28,5kb native DLL named "ThePE.dll" that pops up a MessageBox "Injected!" on attach.

Even if you decided not to compile the DLL yourself, I suggest you still read this part so you get a basic idea of what's going on.

Fire up your C++ IDE and create a new Windows DLL project.

(How to? VS 2008)

Spoiler ([Click to View](#))

[img]http://i.minus.com/1342005374/YOqYePTPe8vedqyD7sYc2Q/i8iHtVuYfGz3L.png[/img]

[img]http://i.minus.com/1342005521/7A3PgMypriNNsnf6UbzP3g/inVcHiutwQoma.png[/img]

[img]http://i.minus.com/1342005627/Hf3i6lGNj2E6XR9mr_VZ0g/iN1NkkBFeleMe.png[/img]

Regardless of your compiler of choice, you should end up with compiler generated DLL entry point, which looks somewhat like this:

```
Code BOOL APIENTRY DllMain( HMODULE hModule,  
    DWORD ul_reason_for_call,
```

```

LPVOID lpReserved
    )
{
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}

```

When we get to the part of injecting the DLL into another process, this is the function that gets called, and the "case DLL_PROCESS_ATTACH:" part of the switch statement will be executed. This is where we would normally create a thread/threads which our server, hack, extension, whatever would run on.

For the sake of simplicity however, we are just going to display a messagebox. Add this code after DLL_PROCESS_ATTACH:

Code `MessageBox(NULL, L"Injected!", L"Success", MB_OK);`
 Your code should now look more or less like this:

Code `#include "stdafx.h"`

```

BOOL APIENTRY DllMain( HMODULE hModule,
    DWORD ul_reason_for_call,
    LPVOID lpReserved
    )
{
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
            MessageBox(NULL, L"Injected!", L"Success", MB_OK);
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}

```

Compile the DLL(or download from the link above). And rename it to PE.Dat:

[img]<http://i.minus.com/1342008029/mlIRlw4DFdA9Rd3W8IM3Q/ibnH0Lj5uPI9CR.png>[/img]

You should now have a compiled DLL file renamed to PE.DAT.

The C# Part (Injector)

Create a new C# Console Application.

Select Properties and open the Resources tab:

[img]<http://i.minus.com/1342009422/Pwo0sfTmHDrs-jqdFyGa9A/ibcS99tIzzIChN.png>[/img]

Select Add Resource -> Add Existing file, find and select your PE.Dat.

[img]<http://i.minus.com/1342009518/ITbXMZ2O-4MdwXhN-1O7yA/ibkRShwGU9wQl0.png>[/img]

Now tap open program.cs. To keep things as simple as possible, we'll do everything from the main function.

Start by adding these two imports in the top of your code:

Code using System.Diagnostics;
using System.Runtime.InteropServices;
Then we need some declarations for calling windows API.

Copy these flag structures right after Code class Program {:

Code [Flags]

```
enum ProcessAccessFlags : uint
{
    All = 0x001F0FFF,
    Terminate = 0x001,
    CreateThread = 0x002,
    VMOperation = 0x008,
    VMRead = 0x0010,
    VMWrite = 0x0020,
    DupHandle = 0x0040,
    SetInformation = 0x00200,
    QueryInformation = 0x00400,
    Synchronize = 0x00100
}
```

[Flags()]

```
enum AllocationType : uint
{
    COMMIT = 0x1000,
    RESERVE = 0x2000,
    RESET = 0x80000,
    LARGE_PAGES = 0x200,
    PHYSICAL = 0x400,
    TOP_DOWN = 0x100,
    WRITE_WATCH = 0x200
}
```

[Flags()]

```
enum MemoryProtection : uint
{
    EXECUTE = 0x10,
    EXECUTE_READ = 0x20,
    EXECUTE_READWRITE = 0x40,
    EXECUTE_WRITECOPY = 0x80,
    NOACCESS = 0x01,
```

```

    READONLY = 0x02,
    READWRITE = 0x04,
    WRITECOPY = 0x08,
    GUARD_Modifierflag = 0x100,
    NOCACHE_Modifierflag = 0x200,
    WRITECOMBINE_Modifierflag = 0x400
}

```

And right after them, put these API call declarations:

```

Code [DllImport("kernel32.dll")]
static extern IntPtr CreateRemoteThread(IntPtr hProcess,
    IntPtr lpThreadAttributes, uint dwStackSize, IntPtr
    lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr lpThreadId);
[DllImport("kernel32.dll")]
static extern IntPtr OpenProcess(ProcessAccessFlags dwDesiredAccess,
[MarshalAs(UnmanagedType.Bool)] bool bInheritHandle, int dwProcessId);
[DllImport("kernel32", CharSet = CharSet.Ansi, ExactSpelling = true, SetLastError = true)]
static extern IntPtr GetProcAddress(IntPtr hModule, string procName);
[DllImport("kernel32.dll", CharSet = CharSet.Auto)]
public static extern IntPtr GetModuleHandle(string lpModuleName);
[DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress,
    uint dwSize, AllocationType flAllocationType, MemoryProtection flProtect);
[DllImport("kernel32.dll", SetLastError = true)]
static extern bool WriteProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress, byte[] lpBuffer, uint
nSize, out int lpNumberOfBytesWritten);
[DllImport("kernel32.dll", SetLastError = true)]
static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMilliseconds);

```

Right. Now our program is capable enough of calling windows API functions for our needs, so we can start writing the main function.

What we're going to do(in a nutshell) is something like this:

Pseudo code:

```

Code Find Notepad Window < Exit If Not Exists >
Extract PE From Resources And Save To File
Use Windows API -> CreateRemoteThread To Force Notepad To Call LoadLibrary On Saved PE
Wait For Remote Thread To Complete
Return

```

First thing to do is finding the notepad window. For this we'll use the managed System.Diagnostics.Process class to enumerate through processes named "notepad", check if the enumeration returned more than 0 results and if so, save the process ID of the "first notepad" in variable pidNotepad and proceed, otherwise exit("Couldn't find notepad window").

```

Code Process[] p = Process.GetProcessesByName("notepad");
if (p.Count() == 0)
{
    Console.WriteLine("Couldn't find notepad window...");
    Console.ReadKey();
    return;
}
int pidNotepad = p[0].Id;
Console.WriteLine("Notepad found [ PID: " + pidNotepad.ToString() + " ]");

```

Then we will use the OpenProcess API to return a handle with full read/write/execute access to the notepad process, and save it to variable hProc. If opening process fails(OpenProcess returns 0), exit("Couldn't open process").

```
Code IntPtr hProc = OpenProcess(ProcessAccessFlags.All, false, pidNotepad);
```

```
if (hProc == IntPtr.Zero)
{
    Console.WriteLine("Couldn't open process.");
    Console.ReadKey();
    return;
}
```

Now we're going to extract our resource PE.Dat and save it to our application data folder with the name "TempPE.dat":

```
Code string pathTempPE =
Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\TempPE.dat";
```

```
byte[] PE_Bytes = Properties.Resources.PE;
```

```
System.IO.FileStream fTempPE = new System.IO.FileStream(pathTempPE,
System.IO.FileMode.Create, System.IO.FileAccess.Write);
fTempPE.Write(PE_Bytes, 0, PE_Bytes.Length);
fTempPE.Close();
```

Okay. Now we need to allocate some memory inside notepad process and get a handle to where that memory begins, so we know where to create our thread. Sounds scary right? Luckily there's an API for that too, called VirtualAllocEx which makes it very easy for us.

We'll just call VirtualAllocEx(Opened Process, 0, our dll path's length, Commit memory, Feel free to wreak havoc) and save the return value into ptrMem, which is a pointer to the start of our brand new allocated memory. If something fails, the return value will be 0 in which case we'll exit("Couldn't allocate memory").

```
Code IntPtr ptrMem = VirtualAllocEx(hProc, (IntPtr)0, (uint)pathTempPE.Length,
AllocationType.COMMIT, MemoryProtection.EXECUTE_READ);
```

```
if (ptrMem == IntPtr.Zero)
{
    Console.WriteLine("Couldn't allocate memory.");
    Console.ReadKey();
    return;
}
```

Almost there. A bit later we will force notepad to call LoadLibrary with path to our temp PE as it's parameter, but first we have to store that path somewhere into notepad's own memory, and for that we will use the WriteProcessMemory API. WriteProcessMemory takes the following parameters:

```
Process ( our hProc )
Address to write ( our ptrMem we just allocated )
Bytes to write ( path to our temp PE converted to bytes )
Bytes count ( count of bytes in ^ )
[out] Bytes written ( returns the bytes written into this variable, our ipTmp )
```

So we create a managed ASCIIEncoding object and convert our pathTempPE into bytes and save

those bytes into variable pathTempPEBytes, then call WriteProcessMemory and check its return value. If it returns true, proceed. If false, exit("Couldn't write memory").

```
Code ASCIIEncoding ae = new ASCIIEncoding();
```

```
byte[] pathTempPEBytes = ae.GetBytes(pathTempPE);
```

```
int ipTmp = 0;
```

```
if (!WriteProcessMemory(hProc, ptrMem, pathTempPEBytes, (uint)pathTempPEBytes.Length, out ipTmp))
```

```
{
    Console.WriteLine("Couldn't write memory.");
    Console.ReadKey();
    return;
}
```

The final part! To call CreateRemoteThread with LoadLibrary as it's parameter we will first need to know the address of the LoadLibrary function. So we will first call the GetProcAddress API with "KERNEL32.DLL" and "LoadLibraryA" as it's parameters. This means we are trying to find the entry point "LoadLibraryA" in "KERNEL32.DLL" and save it's address to variable ptrLoadLib.

Then(finally!) we call the main function, CreateRemoteThread, which takes the following parameters:

Process (our hProc)

SecurityAttributes(don't mind this for now, we'll just pass 0 here)

Stack Size (don't mind this either, we'll pass 0 again)

Start Address (where to start the thread. we want the thread to call LoadLibrary, so we pass ptrLoadLib here.)

Parameter (pointer to the parameter we created, path to temp PE. we'll pass ptrMem here.

Creation flags (don't mind, pass 0)

[out] Thread ID (if you want to track the thread or something, you can give a variable here and the thread ID will be saved to it. I will however pass 0.) We save the thread handle instead, by using IntPtr hThread = CreateRemoteThread...

After which, we will use the WaitForSingleObject API to wait for the thread to complete, AKA the DLL to be injected.

Finally, report("Injection complete").

```
Code IntPtr ptrLoadLib = GetProcAddress(GetModuleHandle("KERNEL32.DLL"), "LoadLibraryA");
```

```
IntPtr hThread = CreateRemoteThread(hProc, IntPtr.Zero, 0, ptrLoadLib, ptrMem, 0, IntPtr.Zero);
```

```
WaitForSingleObject(hThread, unchecked((uint)-1));
```

```
Console.WriteLine("Injection complete.");
```

```
Console.ReadKey();
```

It's ready, open notepad and run the program. You should see notepad display a MessageBox "Injected!", "Success".

[img]<http://i.minus.com/1342021561/aM3w2OM-gn4ewqikpYxsXg/iCdIz8TV3c2Rw.png>[img]

In case of compile errors / laziness, here's the whole project for download:

[Download Project Files \(Direct \)](#)

NOTE: It is very much recommended to actually read through the tutorial and try to understand the essence of Win API, instead of just copy & pasting the code.

Conclusion:

You can accomplish almost anything on windows by using the windows API, and you can use it in almost any programming language. In C++, however, you can just include "windows.h" and you're good to go.

If you're on a x64 system, you will have to do slight modifications, but I will not go there.

Note that once injected, the DLL file will become unavailable to any other process (you can't delete it, write to it, etc) until notepad decides to unload it. In other words, you will have to write your own unload routines, or just close notepad and start over.

You can use this method to create a crypter, for instance.

Guess that's it. Any questions about anything I'm more than happy to answer them.

Penfold - July 11, 2012, 6:58 am

Detailed tutorial, thanks very much.

Bookmarked for future reference.

hackpet - July 11, 2012, 9:37 am

Very nice tutorial, really THX! :) 35 char -.-

?lassy - July 11, 2012, 4:11 pm

Pictures are broken, but this seems like a detailed HQ tutorial. :)

I don't know shit about injecting in C#, but I hope other people learn from this. :D

JeromeMarshall - July 12, 2012, 2:15 am

I was going to write a tutorial the other day about injecting a managed dll (with a c++ wrapper) into an unmanaged process, but I noticed there's a few tutorials out there already. This is a little bit different, but a way more details tut than I would have wrote. Nice one pal, HF needs more of these posts, it will really help them understand the deep inner workings of the framework.

Also for anyone trying to expand on OP's idea, just make sure if you're doing something other than msgbox, that you make sure to create a separate thread, and don't use the dllmain. Lots and lots of room for a deadlock if you don't do that.

Tsukuyomi - July 12, 2012, 7:42 am

(Jul 12, 2012 02:15 AM)JeromeMarshall Wrote: [\[CITE\]](#) I was going to write a tutorial the other day about injecting a managed dll (with a c++ wrapper) into an unmanaged process, but I noticed there's a few tutorials out there already. This is a little bit different, but a way more details tut than I would have wrote. Nice one pal, HF needs more of these posts, it will really help them understand the deep inner workings of the framework.

Also for anyone trying to expand on OP's idea, just make sure if you're doing something other than

msgbox, that you make sure to create a separate thread, and don't use the dllmain. Lots and lots of room for a deadlock if you don't do that.

About managed DLL injection, you can actually start the CLR runtime in a process after which you can inject .Net 4.0 modules into that process. Lots of fun, you can even do hooks with managed code. Quite useless though. If you're already injecting a native DLL, why not just use unmanaged code for whatever it is you're doing.

I wanted this tutorial to be detailed, as the purpose was not just to show people how to inject a DLL, but to demonstrate the capabilities of the windows API altogether.

(Jul 11, 2012 04:11 PM)BlankHat Wrote: [\[CITE\]](#) Pictures are broken, but this seems like a detailed HQ tutorial. :)

I don't know shit about injecting in C#, but I hope other people learn from this. :D

The pictures are back. I have no idea why they were off, maybe i.minus was down for a while.

unnamednoob - July 12, 2012, 11:37 am

great tutorial i've been looking for stuff like this for a while in the hacking section and never found something good. Thanks a lot, hope you don't mind if i PM you with some questions when i try to apply this tutorial and extend it.

Washi - July 12, 2012, 11:44 am

Nice tutorial. I'd suggest to create a new instance of a Win32Exception with the Error Code you can get at Marshal.GetLastWin32Error() to get the last native error with an error message and stuff like that when an IntPtr.Zero is returned. Otherwise you'll have a message saying Couldn't open process and you've no idea why it failed.

Tsukuyomi - July 12, 2012, 12:19 pm

(Jul 12, 2012 11:44 AM)TheUnknownProgrammer Wrote: [\[CITE\]](#) Nice tutorial. I'd suggest to create a new instance of a Win32Exception with the Error Code you can get at Marshal.GetLastWin32Error() to get the last native error with an error message and stuff like that when an IntPtr.Zero is returned. Otherwise you'll have a message saying Couldn't open process and you've no idea why it failed.

There was a plenty of things left out of this tutorial for the sake of simplicity, advanced error handling being one of them.

But you're right though, using native errors is generally good practice when calling Win API.