

Forum:[Hack Forums](#)

Thread: [\[Tutorial\] \[VB.Net\] Inject PE From Resources](#)

Tsukuyomi - July 11, 2012, 7:08 am

[C# Version of this tutorial](#)

Hello.

In this tutorial we will...Create a simple PE file (C++) (Don't worry, you can skip the C++ part if you like and just download a precompiled version.)

Save that file as a managed resource (VB.Net)

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

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

A VB.Net 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/1342005521/7A3PgMyprINNsnf6UzbP3g/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/mIIRlVw4DFdA9Rd3W8IM3Q/ibnH0Lj5uPI9CR.png>[/img]

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

The VB.Net Part (Injector)

Create a new VB.Net Console Application.

Select Properties and open the Resources tab:

[img]http://i.minus.com/1342009422/Pwo0sfTmHDrs-jqdFyGa9A/ibcS99tlZZIChN.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 Module1.vb. 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 Module1:

```
Code <Flags()> _  
Private Enum ProcessAccessFlags As UInteger  
All = &H1F0FFF  
Terminate = &H1  
CreateThread = &H2  
VMOperation = &H8  
VMRead = &H10  
VMWrite = &H20  
DupHandle = &H40  
SetInformation = &H200  
QueryInformation = &H400  
Synchronize = &H100  
End Enum
```

```
<Flags()> _  
Private Enum AllocationType As UInteger  
COMMIT = &H1000  
RESERVE = &H2000  
RESET = &H80000  
LARGE_PAGES = &H200  
PHYSICAL = &H400  
TOP_DOWN = &H100  
WRITE_WATCH = &H200  
End Enum
```

```
<Flags()> _  
Private Enum MemoryProtection As UInteger  
EXECUTE = &H10  
EXECUTE_READ = &H20  
EXECUTE_READWRITE = &H40  
EXECUTE_WRITECOPY = &H80  
NOACCESS = &H1  
[READONLY] = &H2  
READWRITE = &H4
```

```

WRITECOPY = &H8
GUARD_Modifierflag = &H100
NOCACHE_Modifierflag = &H200
WRITECOMBINE_Modifierflag = &H400
End Enum

```

And right after them, put these API declarations:

```

Code <DllImport("kernel32.dll")> _
Private Function CreateRemoteThread(ByVal hProcess As IntPtr, ByVal lpThreadAttributes As
IntPtr, ByVal dwStackSize As UInteger, ByVal lpStartAddress As IntPtr, ByVal lpParameter As
IntPtr, ByVal dwCreationFlags As UInteger, _
ByVal lpThreadId As IntPtr) As IntPtr
End Function
<DllImport("kernel32.dll")> _
Private Function OpenProcess(ByVal dwDesiredAccess As ProcessAccessFlags,
<MarshalAs(UnmanagedType.Bool)> ByVal bInheritHandle As Boolean, ByVal dwProcessId As
Integer) As IntPtr
End Function
Private Declare Ansi Function GetProcAddress Lib "kernel32" (ByVal hModule As IntPtr, ByVal
procName As String) As IntPtr
<DllImport("kernel32.dll", CharSet:=CharSet.Auto)> _
Public Function GetModuleHandle(ByVal lpModuleName As String) As IntPtr
End Function
Private Declare Auto Function VirtualAllocEx Lib "kernel32.dll" (ByVal hProcess As IntPtr, ByVal
lpAddress As IntPtr, ByVal dwSize As UInteger, ByVal flAllocationType As AllocationType, ByVal
flProtect As MemoryProtection) As IntPtr
<DllImport("kernel32.dll", SetLastError:=True)> _
Private Function WriteProcessMemory(ByVal hProcess As IntPtr, ByVal lpBaseAddress As
IntPtr, ByVal lpBuffer As Byte(), ByVal nSize As UInteger, ByVal lpNumberOfBytesWritten As
Integer) As Boolean
End Function
<DllImport("kernel32.dll", SetLastError:=True)> _
Private Function WaitForSingleObject(ByVal hHandle As IntPtr, ByVal dwMilliseconds As UInt32)
As UInt32
End Function

```

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 Dim p As Process() = Process.GetProcessesByName("notepad")
If p.Count() = 0 Then

```

```
Console.WriteLine("Couldn't find notepad window...")
Console.ReadKey()
Return
```

```
End If
```

```
Dim pidNotepad As Integer = 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 Dim hProc As IntPtr = OpenProcess(ProcessAccessFlags.All, False, pidNotepad)
```

```
If hProc = IntPtr.Zero Then
```

```
Console.WriteLine("Couldn't open process.")
```

```
Console.ReadKey()
```

```
Return
```

```
End If
```

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

```
Code Dim pathTempPE As String =
```

```
Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\TempPE.dat"
```

```
Dim PE_Bytes As Byte() = My.Resources.PE
```

```
Dim fTempPE As 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 Dim ptrMem As IntPtr = VirtualAllocEx(hProc, IntPtr.Zero, CUInt(pathTempPE.Length),  
AllocationType.COMMIT, MemoryProtection.EXECUTE_READ)
```

```
If ptrMem = IntPtr.Zero Then
```

```
Console.WriteLine("Couldn't allocate memory.")
```

```
Console.ReadKey()
```

```
Return
```

```
End If
```

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 Dim ae As New ASCIIEncoding()
```

```
Dim pathTempPEBytes As Byte() = ae.GetBytes(pathTempPE)
```

```
Dim ipTmp As Integer = 0
```

```
If Not WriteProcessMemory(hProc, ptrMem, pathTempPEBytes,  
CUInt(pathTempPEBytes.Length), ipTmp) Then  
    Console.WriteLine("Couldn't write memory.")  
    Console.ReadKey()  
    Return  
End If
```

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 Dim ptrLoadLib As IntPtr = GetProcAddress(GetModuleHandle("KERNEL32.DLL"),  
"LoadLibraryA")
```

```
Dim hThread As IntPtr = CreateRemoteThread(hProc, IntPtr.Zero, 0, ptrLoadLib, ptrMem, 0, _  
IntPtr.Zero)
```

```
WaitForSingleObject(hThread, 0)
```

```
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/iCdlz8TV3c2Rw.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.

ubbelol - July 11, 2012, 7:54 am

The reason crypters wouldn't use this is because it requires you to write the file to disk non-encrypted and vulnerable before loading it into the target process. Good tut anyway. :)

aTastyUnicorn - July 11, 2012, 8:57 am

Really nice Tut & Explanations. I like the C++ Part.

One advice: Upload the Pictures to another website, pic's are already down.

ubanooba - July 11, 2012, 9:21 am

Manually map the file straight from resources into the remote process without having to write it to the hard-disk and this would be more impressive. Still, nice tut.

h4kr! - July 11, 2012, 11:29 am

good tut.

the "Imports" at the top were in c# format,however. VB.NET is slightly different as:

Code Imports System.Diagnostics

Imports System.Runtime.InteropServices

????? - July 11, 2012, 11:35 am

There are so many different things you could do using native code, why implement this with a crypter?

Good tut nonetheless.

Tsukuyomi - July 12, 2012, 7:51 am

(Jul 11, 2012 11:35 AM)????? Wrote: [\[CITE\]](#) There are so many different things you could do using native code, why implement this with a crypter?

Good tut nonetheless.

Well the point of this tutorial was to get people excited about the windows API and understand it's capabilities and simplicity.

There's a zillion .Net developers around here who think c++ looks like this Code

('&%;9]!~}|z2Vxwv-,POqponl\$Hjig%eB@@>}=<M:9wv6WsU2T|nm-,jcL(I&%\$##"

`CB]V?Tx<uVtT`Rpo3NIF.Jh++FdbCBA@?]|~|4XzyTT43Qsqq(Lnmkj"Fhg\${z@> and don't know that anything but the managed functions exist. If you're a windows developer, you should know windows API.. :)

I wasn't seriously suggesting writing a crypter with exactly this technique, but the concept is similar and it's something to get people going on.

Sketchy - July 12, 2012, 1:23 pm

Remember you are pausing the main thread.... >_> which can be fatal also your stack/eax will be fucked up after O:<

ubanooba - July 12, 2012, 5:17 pm

Manual Map the PE into memory, way more fun :3

jessman420 - July 12, 2012, 5:29 pm

This is a great tut, very useful information and well put together, many thanks.

Sketchy - July 12, 2012, 6:50 pm

(Jul 12, 2012 05:17 PM)ubanooba Wrote: [\[CITE\]](#) Manual Map the PE into memory, way more fun :3

Ah, I find no point in manual mapping if the host process is not scanning/successful injection. you will just blow up the payload and possibly increase more problems such as memory leaks.

SIM-99212 - July 13, 2012, 12:38 am

Is there any reason why the resource DLL bytes need to be written to the disk?

DaRkReD - July 13, 2012, 2:20 am

(Jul 13, 2012 12:38 AM)Binary Blue Wrote: [\[CITE\]](#) Is there any reason why the resource DLL bytes need to be written to the disk?

Because the API's target things on the file system rather than in memory. You can get around it by using VirtualAllocEx and ZwUnmapViewOfSection API's

uspoer321 - July 29, 2018, 1:06 pm

good tutorial fo begginers