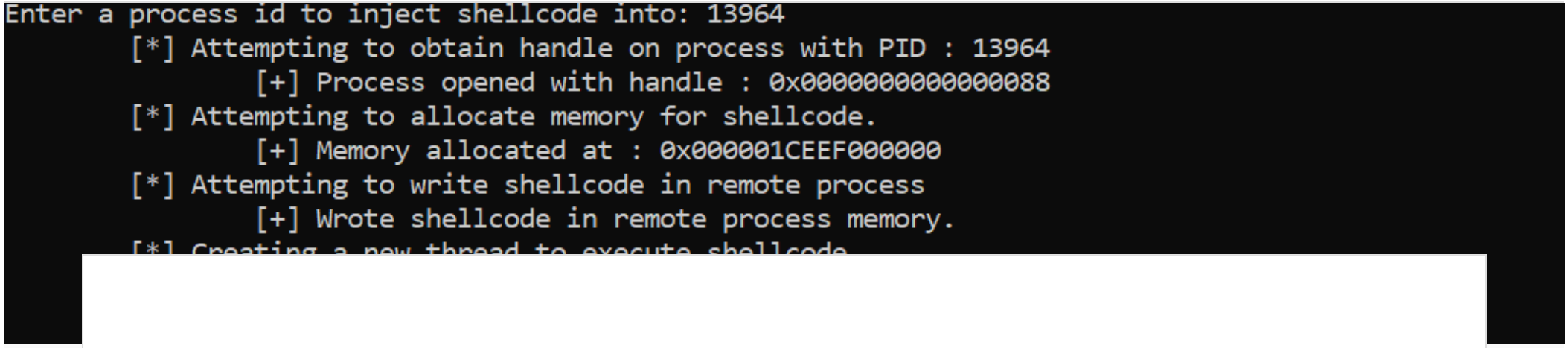

Demystifying Code Injection Techniques: Part 1 – Shellcode Injection

BY HIMANSHU KHOKHAR / ON JUNE 13, 2019

/ IN CODE INJECTION, MALWARE ANALYSIS, REVERSE ENGINEERING

Introduction

Code injection refers to the act of injecting arbitrary external code in an application. There are two types of code injection:

1. Injection into vulnerable programs.
2. Injection into non-vulnerable programs.

If code injection is done in vulnerable applications, it is done via exploitation of a bug which occurs when processing invalid data. In this case, the extent of code injection is dependent on the bug in the application, which we also refer to as "vulnerability". The problem with this scenario is that the application should have a bug that can be leveraged to gain code execution.

This series targets the second scenario, where the application is not expected/required to have vulnerability. This, as can be expected, gives a very wide area of application as well as opportunity of abusing this functionality provided by the operating system.

Malwares typically use this method as it does not require any conditions to be met before performing code injection as well as it is fully supported by the operating system itself. All major operating systems provide ABIs and APIs to interact with other process, control them, read and write memory in the process space of other programs.

Code Injection via CreateRemoteThread API

CreateRemoteThread is a function provided by Win32 API for creating threads in another processes. There are two conditions that must be met before creating the thread in another application, and they are:

1. The process attempting to create a thread in another process must have permissions to create thread. In simpler terms, this translates to having the same or higher privileges than the victim process (*target* refers to the process in which we want to create a thread).
2. Both the processes (target and attacker processes) must reside in the same session. If the session identifiers do not match, thread will not be created.

In case either of the above-mentioned conditions are not met, the process of code injection will be denied by the operating system itself.

This is *not* a security hole in the architecture of the Windows OS, rather it is *provided functionality* by the operating system. Since we cannot modify the processes having higher privileges than we are having, no security boundary is crossed.

Algorithm

In order to achieve code injection via *CreateRemoteThread* API, we follow this algorithm:

1. Select a process as the victim process.
2. Obtain access to process using *OpenProcess* function with parameter *PROCESS_ALL_ACCESS* to be able to perform required operations.
3. If *OpenProcess* fails, exit.
4. Allocate memory in the process space of victim process by using *VirtualAllocEx* function.
5. If *VirtualAllocEx* fails, exit.

6. Write the shellcode to the memory location allocated by *VirtualAllocEx*.
7. If shellcode writing fails or shellcode written partially, exit.
8. After the shellcode has been written, call *CreateRemoteThread* with necessary parameters, pointing the address of shellcode as the *LPTHREAD_START_ROUTINE*.

Environment

For the purpose of this series, I have the following environment setup:

- Visual Studio 2019
- Windows 10 RS6 (x64)

Note: The code can be accessed from my github repository – here.

In-depth working of code injection

Accessing Remote Process

To perform memory manipulation on any process, we must have access to it. It can be obtained by using the function *OpenProcess* which is prototyped as:

```
C++

HANDLE OpenProcess(
  DWORD dwDesiredAccess,
  BOOL  bInheritHandle,
  DWORD dwProcessId
);
```

It takes three parameters:

1. *dwDesiredAccess*: The requested access to the process object. It is checked against the security token of the victim process. There are handful of desired access values that one can specify but specifying *PROCESS_ALL_ACCESS* encompasses all the possible access rights for the process.
2. *bInheritHandle*: It is a Boolean value which dictates whether the process created by this process will inherit this handle or not.
3. *dwProcessId*: This is the process identifier of the victim process.

This process returns a *HANDLE* to other process (on success) which can be used by other API functions to manipulate the memory of the victim process. On failure, it returns NULL.

Allocating space for shellcode

Once we obtain the handle to the victim process, we move on to allocating space for our shellcode in the victim process memory. This is done by using *VirtualAllocEx* API call.

VirtualAllocEx is prototyped as:

```
C++

LPVOID VirtualAllocEx(
  HANDLE hProcess,
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD  flAllocationType,
  DWORD  flProtect
);
```

It takes five parameters:

1. *hProcess*: Handle to the process inside which we want to allocate memory.
2. *lpAddress*: A pointer to the specified address in the victim process memory. If the parameter is specified as NULL, then the function automatically selects a memory page to allocate to.
3. *dwSize*: The size of the memory region to be allocated. It is specified in bytes.
4. *flAllocationType*: It specifies the type of memory to be allocated. The parameter must contain one of the three values: MEM_COMMIT, MEM_RESERVE, MEM_RESET, MEM_RESET_UNDO.
5. *flProtect*: It specifies the memory protection of the allocation. For our purposes, since it will contain our code to be executed, and we want it readable as well writable, we will set this to PAGE_EXECUTE_READWRITE.

The function returns the base address of allocation on success, while on failure, it returns NULL.

At this point, we have successfully managed to allocate executable memory in the victim process.

Writing shellcode in remote process

Now, we need to write our shellcode in the allocated region. To achieve this, we have a function called *WriteProcessMemory*.

WriteProcessMemory is a function that writes data to an area of memory of the specified process by the caller. A caveat to this is that the entire memory area must be writable else the function fails, and that is why we allocated the memory as writable, in conjunction with readable and executable.

WriteProcessMemory is prototyped as:

```
C++

BOOL WriteProcessMemory(
  HANDLE  hProcess,
  LPVOID  lpBaseAddress,
  LPCVOID lpBuffer,
  SIZE_T  nSize,
  SIZE_T  *lpNumberOfBytesWritten
);
```

WriteProcessMemory takes five parameters:

1. *hProcess*: Handle to the process inside which we want to write data to.
2. *lpBaseAddress*: The address (in form of pointer) where we want to write our data at.
3. *lpBuffer*: Pointer to data that must be written. The pointer must be a *const* pointer.
4. *nSize*: The amount of data that must be written (in bytes).
5. **lpNumberOfBytesWritten*: A pointer to a SIZE_T which will store the number of bytes written in that target.

It will return false if the function fails due to some reason and will return a true if it succeeds.

At this point, stage is all set and all that is required is to create a thread in the remote process and run it.

Executing our shellcode

For creating a thread in a remote process, we use the function *CreateRemoteThread* provided by Win32 API.

CreateRemoteThread is a function that is used to create a thread that runs in the virtual space of another process. CreateRemoteThread provides limited functionality and to gain access to extended attributes which can be specified to threads, *CreateRemoteThreadEx* can be used but in our case, the former will suffice.

CreateRemoteThread is prototyped as:

```
C++

HANDLE CreateRemoteThread(
  HANDLE                 hProcess,
  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  SIZE_T                 dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID                 lpParameter,
  DWORD                  dwCreationFlags,
  LPDWORD                lpThreadId
);
```

CreateRemoteThread takes seven parameters out of which only three are of interest to us. The rest can have default values though by tweaking them, more control can be taken on the newly created thread.

The parameters of our interest are:

1. *hProcess*: The handle to the victim process inside which we want to create the thread.
2. *lpStartAddress*: It is a pointer to THREAD_START_ROUTINE which is the location from which the thread will start executing code once created.
3. *lpParameters*: A pointer to the parameters expected by *LPTHREAD_START_ROUTINE*. Since in this case, it is a plain shellcode, it does not expect any arguments and therefore, we will keep it NULL. This parameter will be of value in DLL Injection.

Demo

For the demo, I am using a shellcode to launch *calc.exe* (generated using *msfvenom*), on Windows 10 RS6 (x64).

Advantages

Code injection using CreateRemoteThread has several advantages over other code injection techniques. Some of them are:

1. Easiest to implement.
2. Does not require address resolution of library functions.
3. Does not wait for events.
4. No need of manipulating file structure in memory.

Disadvantages

Along with advantages, there are several disadvantages which does not make it a go-to code injection technique. Some of them are:

1. Only binary code can be injected.
2. Requires a lot of changes when the shellcode is changed.
3. There is a limit on how much you can achieve using a shellcode.

I want to thank my friends Karsten Hahn and Ravi Kiran for reviewing this post.

CODE INJECTION SHELLCODE INJECTION

PREVIOUS

Windows Kernel Exploitation Part 3: Integer Overflow

NEXT

Exploiting CVE-2019-1132: Another NULL Pointer Dereference in Windows Kernel

Leave a Reply

COMMENT

NAME *

EMAIL *

WEBSITE

Save my name, email, and website in this browser for the next time I comment.

Notify me of follow-up comments by email.

Notify me of new posts by email.

POST COMMENT