Fuel Pumps II – PoSlurp.B

norfolkinfosec.com/fuel-pumps-ii-poslurp-b/

norfolk December 31, 2019

In a previous post, this blog examined malware used in a financially-motivated incident at a fuel dispensing company, as disclosed in a security bulletin by VISA. The bulletin detailed a second incident that is likely attributable to an additional threat actor. Specifically, VISA identified C2 infrastructure, a filename, and additional TTPs that allegedly align with FIN8 activity, as described in public Gigamon and Root9b reporting. These TTPs suggest that the threat actors used a memory scraper referred to as ***PoSlurp.B*** in public reporting to scrape customer credit card data from a targeted device.

This post examines a PoSlurp.B file identified (through its shellcode loader) by Twitter user @just_windex to provide additional details regarding the malware's functionality that were not previously disclosed in open source. This analysis focuses on the final payload of the shellcode loader, although additional information and advice for bringing this file into a debuggable state is available at the end of the post.

Unlike the previously analyzed file (FrameworkPoS/GratefulPOS), which indiscriminately scraped all processes on a device, PoSlurp.B is designed to scrape the memory of an attacker-specified process.

Analysis

Shellcode Hash:
MD5: b54283d17b7c13329943168b898ff07e
SHA1: 67a06663b0c8a885d444b8bedb8261b28f050a39
SHA256: e78d9a6cd94bd8ec3095a0ecbbc9c4add78d3281d2bf46497164d0406c346395

Dumped PoSlurp.B Payload (Uploaded to VT for this blog, not from ITW)
MD5: 3d5ae56c6746e0b3ed5b15124264a0d2
SHA1: f92c886f85928041148d0dcd7c4fb9623b157f94
SHA256: d9e442cd69d1f656a3e8cfd0792333a8f0108193e052a4ee2d7f9138a4b253b2

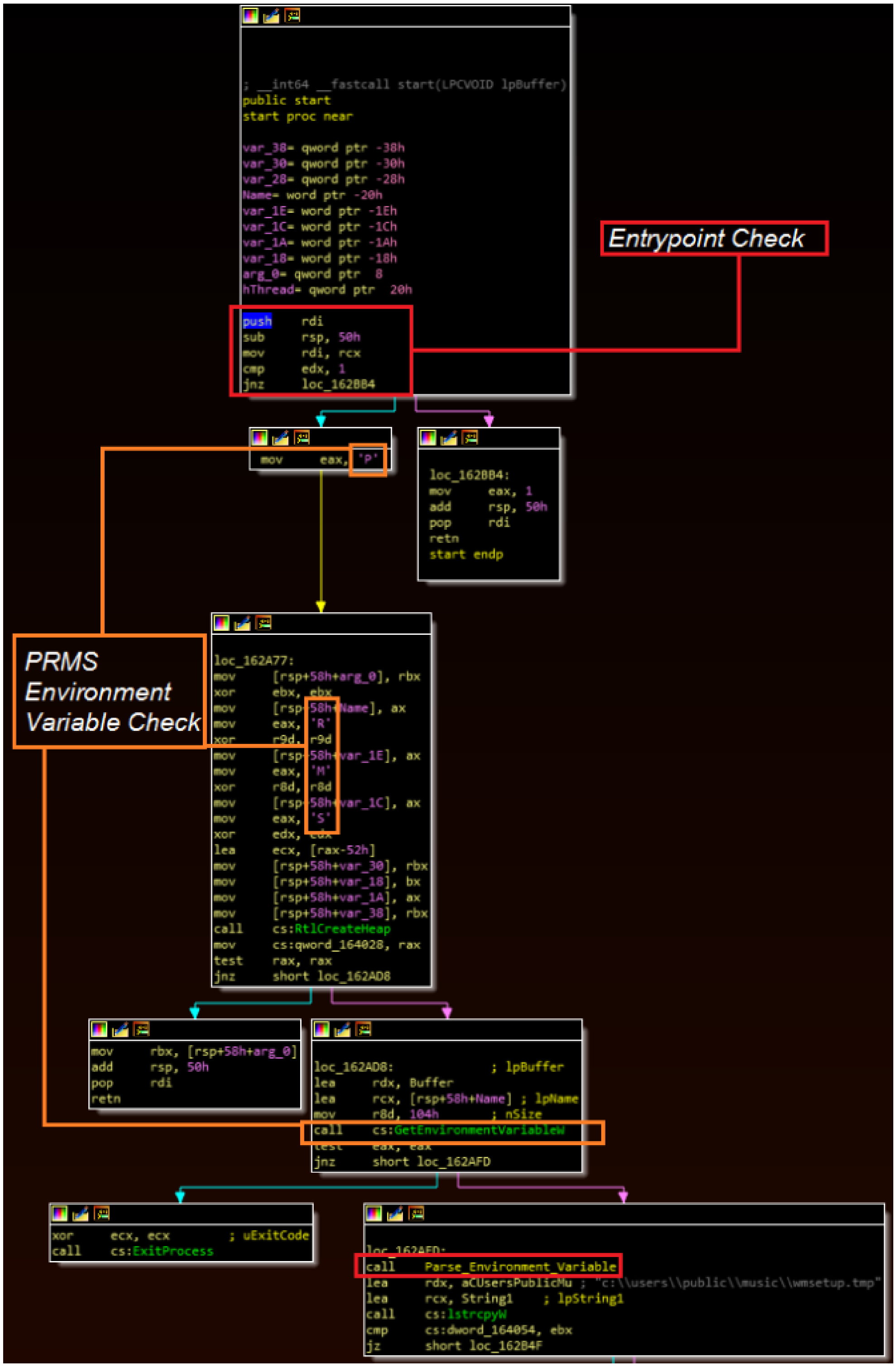
Initial Checks and Exit Conditions

PoSlurp.B is a 64-bit executable that is expected to be run in memory. When executed, the malware performs two conditional checks:

– The malware must have been loaded into memory
– In this analysis, the check appears to be conducted by examining the the entry point
– The malware *must* identify an environment variable – "PRMS" – that contains data to direct

the workflow
– Setting this in the system settings did not appear to work. Setting this in a PowerShell injector script did.

A Gigamon report previously described the need for this environment variable and its presence in a PowerShell loader. While this loader is not currently available on VirusTotal, information regarding reconstructing one is available at the end of this blog. The malware uses a stack string to assemble this environment variable name, likely to limit static detection of the string. These first two checks can be seen below. Following these checks, the malware moves to a validation and parsing function (boxed in red in the bottom right of this image) to extract information from this environment variable.



```
; __int64 __fastcall start(LPCVOID lpBuffer)
public start
start proc near

var_38= qword ptr -38h
var_30= qword ptr -30h
var_28= qword ptr -28h
Name= word ptr -20h
var_1E= word ptr -1Eh
var_1C= word ptr -1Ch
var_1A= word ptr -1Ah
var_18= word ptr -18h
arg_0= qword ptr  8
hThread= qword ptr  20h

push    rdi
sub     rsp, 50h
mov     rdi, rcx
cmp     edx, 1
jnz     loc_162BB4
```

```
mov     eax, 'P'
```

```
loc_162BB4:
mov     eax, 1
add     rsp, 50h
pop     rdi
retn
start endp
```

```
loc_162A77:
mov     [rsp+58h+arg_0], rbx
xor     ebx, ebx
mov     [rsp+58h+Name], ax
mov     eax, 'R'
xor     r9d, r9d
mov     [rsp+58h+var_1E], ax
mov     eax, 'M'
xor     r8d, r8d
mov     [rsp+58h+var_1C], ax
mov     eax, 'S'
xor     edx, edx
lea     ecx, [rax-52h]
mov     [rsp+58h+var_30], rbx
mov     [rsp+58h+var_18], bx
mov     [rsp+58h+var_1A], ax
mov     [rsp+58h+var_38], rbx
call    cs:RtlCreateHeap
mov     cs:qword_164028, rax
test    rax, rax
jnz     short loc_162AD8
```

```
mov     rbx, [rsp+58h+arg_0]
add     rsp, 50h
pop     rdi
retn
```

```
loc_162AD8:             ; lpBuffer
lea     rdx, Buffer
lea     rcx, [rsp+58h+Name] ; lpName
mov     r8d, 104h       ; nSize
call    cs:GetEnvironmentVariableW
test    eax, eax
jnz     short loc_162AFD
```

```
xor     ecx, ecx        ; uExitCode
call    cs:ExitProcess
```

```
loc_162AFD:
call    Parse_Environment_Variable
lea     rdx, aCUsersPublicMu ; "c:\\users\\public\\music\\wmsetup.tmp"
lea     rcx, String1    ; lpString1
call    cs:lstrcpyW
cmp     cs:dword_164054, ebx
jz      short loc_162B4F
```

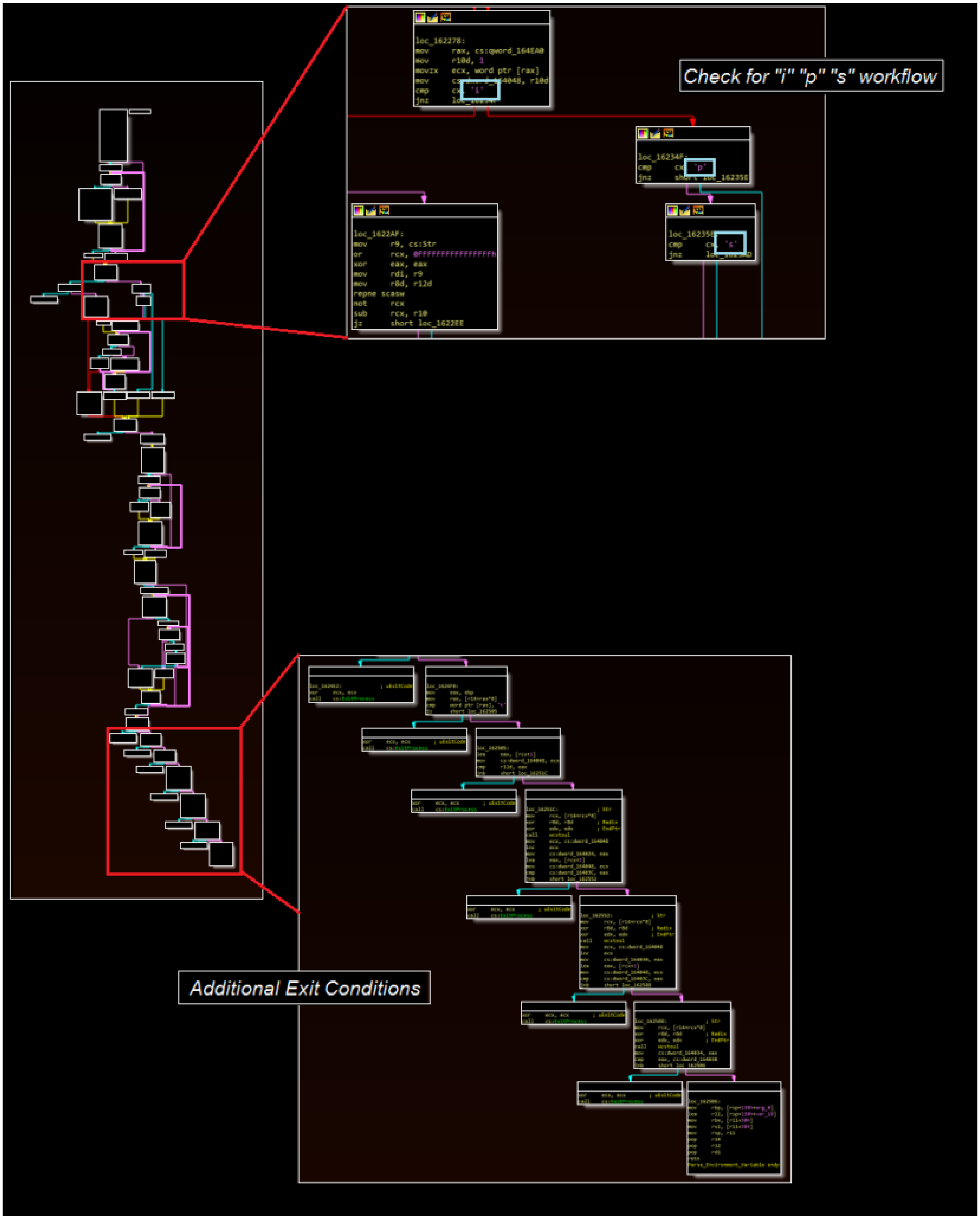
PRMS Environment Variable Check

The parsing function is designed to extract the contents of the environment variable. The function contains nine different references to the ExitProcess Windows API call. Combined with the previous function, the following exit conditions for the malware have been identified:

– The malware determines it wasn’t injected or started properly
– The malware can’t locate the “PRMS” environment variable
– The environment variable doesn’t contain “t” as the first letter of a value in a workflow-

specific position
– When run in injection mode, the malware is unable to identify a process specified for injection
– An invalid value is in the workflow parameter location (i.e. not “i” “s” or “p”)
– An incorrect number of arguments have been specified
– The malware runs successfully

While some of these appear to be anti-analysis checks, this blog assesses that others *may* be for workflow validation and to prevent errors, crashing, or unexpected events. In particular, there are multiple checks regarding the correct number of parameters being passed to the malware that eventually become redundant, as a final check requires a larger number of parameters than an initial check. There are additional exit conditions that are not yet fully understood.



```
loc_162278:
mov     rax, cs:qword_164EA0
mov     r10d, 1
movzx   ecx, word ptr [rax]
mov     cs:dword_164048, r10d
cmp     cx, 'i'
jnz     loc_16234F
```

```
loc_16234F:
cmp     cx, 'p'
jnz     short loc_16235E
```

```
loc_16235E:
cmp     cx, 's'
jnz     loc_16245D
```

```
loc_1622AF:
mov     r9, cs:Str
or      rcx, 0FFFFFFFFFFFFFFFFh
xor     eax, eax
mov     rdi, r9
mov     r8d, r12d
repne scasw
not     rcx
sub     rcx, r10
jz      short loc_1622EE
```

Check for "i" "p" "s" workflow
Additional Exit Conditions
Parse_Environment_Variable endp

Validation and Parsing Function

Environment Variable and Three Workflows

The environment variable is expected to contain multiple values, delimited by a "|" character. The first character specifies which workflow to take, and can be the letter p, s, or i.

– “p” scrapes a specified process for credit card data
– “i” injects the malware into a process and creates a thread at the scraping function used by p
– “s” injects the malware into a suspended svchost process and creates a thread at the same scraping function

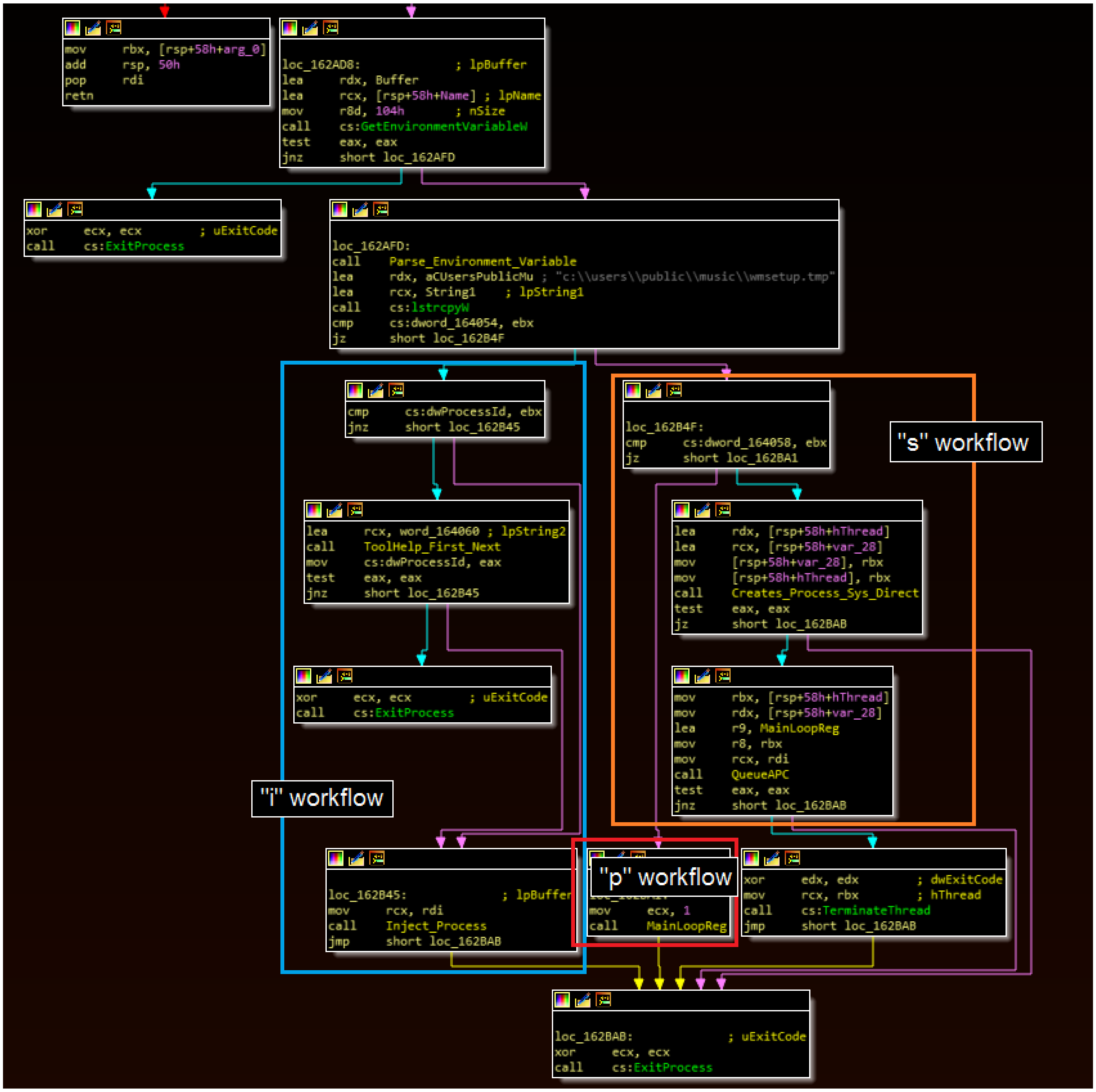
The malware ultimately *appears* to expect more arguments than are necessary in certain cases. For example, if the environment variable were set to:

p|notepad.exe|t|[value]|[value]

The first three values would be sufficient to validate many of the checks and scrape the “notepad.exe” process, although *something* would need to fill the remaining values to successfully run. It is possible that these additional values may perform further validation checks, which were bypassed for the purpose of this analysis (and which would need to be bypassed if using the environment variables exactly as written in this blog).

The malware also treats these arguments differently depending on the mode selected. For example, in “p” and “s” mode the first argument specified after “p” is the process to be scraped. In “i” mode, the first argument after “i” is the process to be injected, whereas the next argument is the process to be scraped. Thus, using “i” mode would require a value such as:

i|injection_target.exe|process_to_be_scraped.exe|t|[unknown]|[unknown]



```
mov     rbx, [rsp+58h+arg_0]
add     rsp, 50h
pop     rdi
retn

loc_162AD8:                 ; lpBuffer
lea     rdx, Buffer
lea     rcx, [rsp+58h+Name] ; lpName
mov     r8d, 104h       ; nSize
call    cs:GetEnvironmentVariableW
test    eax, eax
jnz     short loc_162AFD

xor     ecx, ecx        ; uExitCode
call    cs:ExitProcess

loc_162AFD:
call    Parse_Environment_Variable
lea     rdx, aCUsersPublicMu ; "c:\\users\\public\\music\\wmsetup.tmp"
lea     rcx, String1    ; lpString1
call    cs:lstrcpyW
cmp     cs:dword_164054, ebx
jz      short loc_162B4F

cmp     cs:dwProcessId, ebx
jnz     short loc_162B45

lea     rcx, word_164060 ; lpString2
call    ToolHelp_First_Next
mov     cs:dwProcessId, eax
test    eax, eax
jnz     short loc_162B45

xor     ecx, ecx        ; uExitCode
call    cs:ExitProcess

loc_162B45:                 ; lpBuffer
mov     rcx, rdi
call    Inject_Process
jmp     short loc_162BAB

loc_162B4F:
cmp     cs:dword_164058, ebx
jz      short loc_162BA1

lea     rdx, [rsp+58h+hThread]
lea     rcx, [rsp+58h+var_28]
mov     [rsp+58h+var_28], rbx
mov     [rsp+58h+hThread], rbx
call    Creates_Process_Sys_Direct
test    eax, eax
jz      short loc_162BAB

mov     rbx, [rsp+58h+hThread]
mov     rdx, [rsp+58h+var_28]
lea     r9, MainLoopReg
mov     r8, rbx
mov     rcx, rdi
call    QueueAPC
test    eax, eax
jnz     short loc_162BAB

mov     ecx, 1
call    MainLoopReg

xor     edx, edx        ; dwExitCode
mov     rcx, rbx        ; hThread
call    cs:TerminateThread
jmp     short loc_162BAB

loc_162BAB:                 ; uExitCode
xor     ecx, ecx
call    cs:ExitProcess
```

Malware Workflow Options

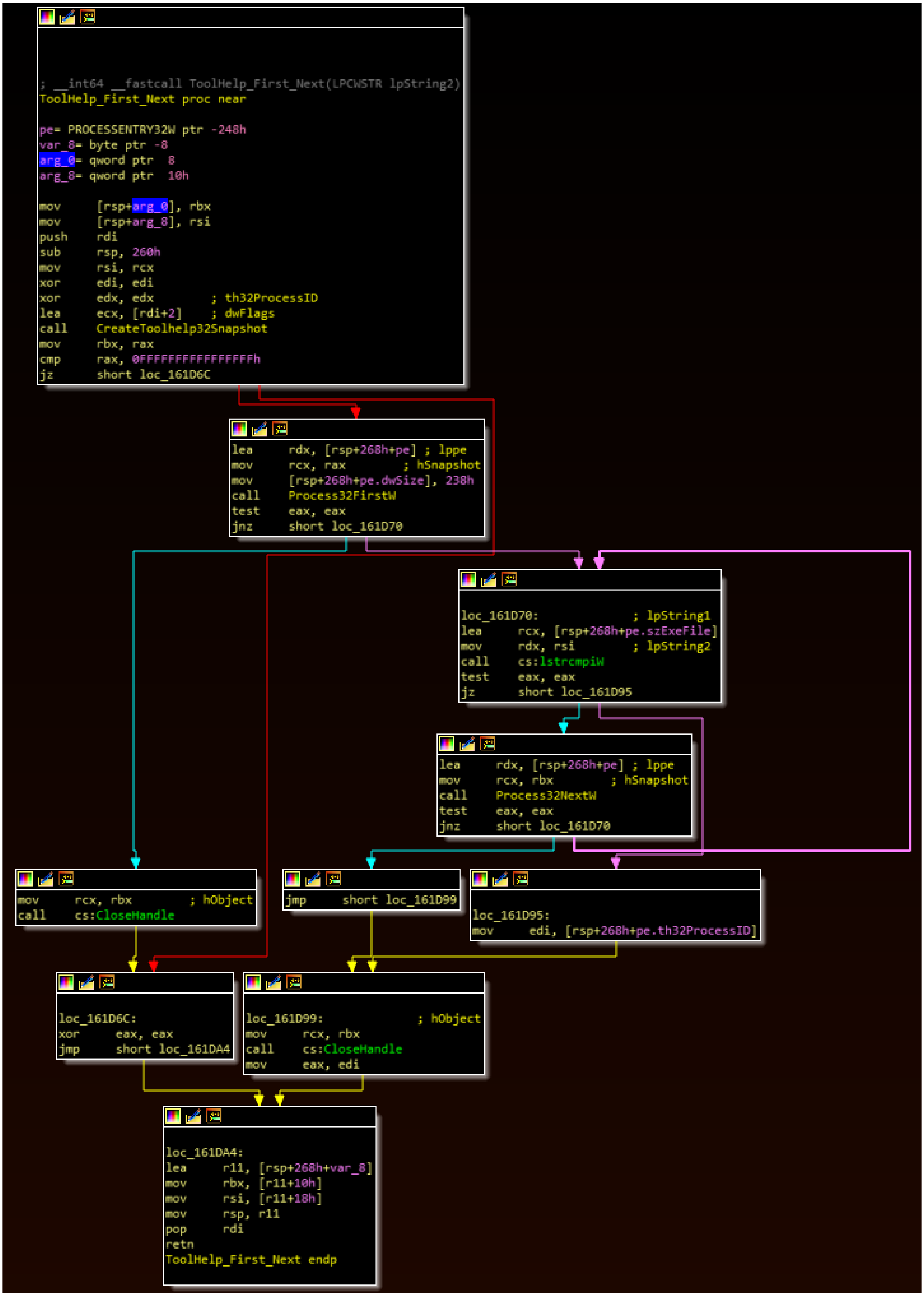
Injection Workflow ("i")

The injection workflow contains two relatively simple functions.

Function One
– The malware uses the CreateToolhelp32Snapshot and Process32First/Next APIs to list running processes
– The malware compares each process name to the first process argument in the environment variable
– If no match is found, the malware returns and exits

Function Two
– The malware opens a handle to the targeted process

– The malware uses the VirtualAllocEx and WriteProcessMemory to write itself to the targeted process
– The malware creates a thread at the location of the main scraping loop within this injected process



```
; __int64 __fastcall ToolHelp_First_Next(LPCWSTR lpString2)
ToolHelp_First_Next proc near

pe= PROCESSENTRY32W ptr -248h
var_8= byte ptr -8
arg_0= qword ptr  8
arg_8= qword ptr  10h

mov     [rsp+arg_0], rbx
mov     [rsp+arg_8], rsi
push    rdi
sub     rsp, 260h
mov     rsi, rcx
xor     edi, edi
xor     edx, edx        ; th32ProcessID
lea     ecx, [rdi+2]    ; dwFlags
call    CreateToolhelp32Snapshot
mov     rbx, rax
cmp     rax, 0FFFFFFFFFFFFFFFFh
jz      short loc_161D6C
```

```
lea     rdx, [rsp+268h+pe] ; lppe
mov     rcx, rax           ; hSnapshot
mov     [rsp+268h+pe.dwSize], 238h
call    Process32FirstW
test    eax, eax
jnz     short loc_161D70
```

```
loc_161D70:                 ; lpString1
lea     rcx, [rsp+268h+pe.szExeFile]
mov     rdx, rsi            ; lpString2
call    cs:lstrcmpiW
test    eax, eax
jz      short loc_161D95
```

```
lea     rdx, [rsp+268h+pe] ; lppe
mov     rcx, rbx           ; hSnapshot
call    Process32NextW
test    eax, eax
jnz     short loc_161D70
```

```
mov     rcx, rbx        ; hObject
call    cs:CloseHandle
```

```
jmp     short loc_161D99
```

```
loc_161D95:
mov     edi, [rsp+268h+pe.th32ProcessID]
```

```
loc_161D6C:
xor     eax, eax
jmp     short loc_161DA4
```

```
loc_161D99:             ; hObject
mov     rcx, rbx
call    cs:CloseHandle
mov     eax, edi
```

```
loc_161DA4:
lea     r11, [rsp+268h+var_8]
mov     rbx, [r11+10h]
mov     rsi, [r11+18h]
mov     rsp, r11
pop     rdi
retn
ToolHelp_First_Next endp
```

First Function (Process Identification) in Injection Workflow

Svchost Workflow (s)

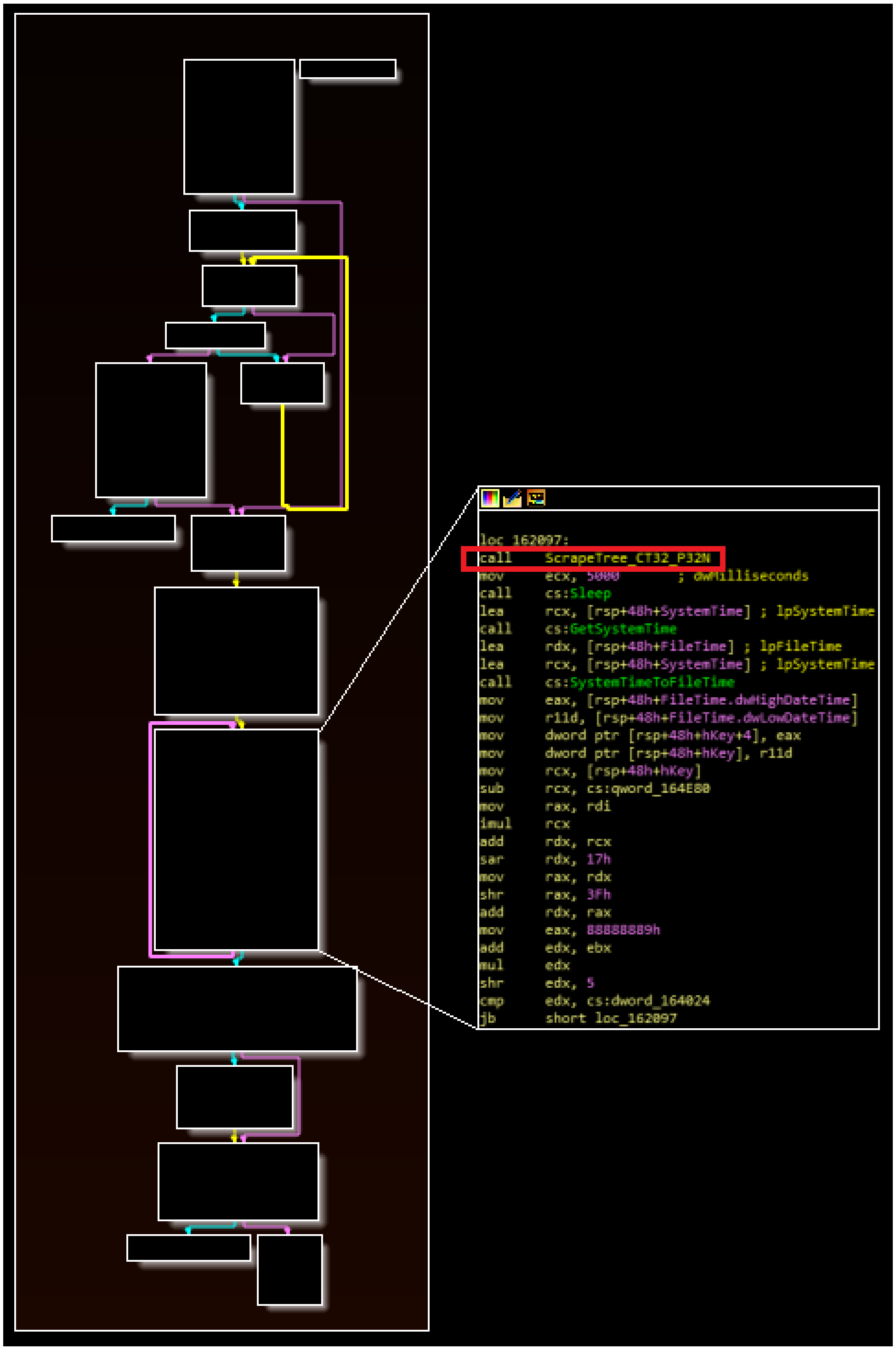
The svchost workflow also contains two functions.

First, the malware uses stack strings to assemble “svchost.exe” (similar to the “PRMS” string creation), likely to avoid static detection of this value. The malware then identifies the system directory via API call and concatenates the svchost.exe process name to this string and spawns this process in a suspended state.

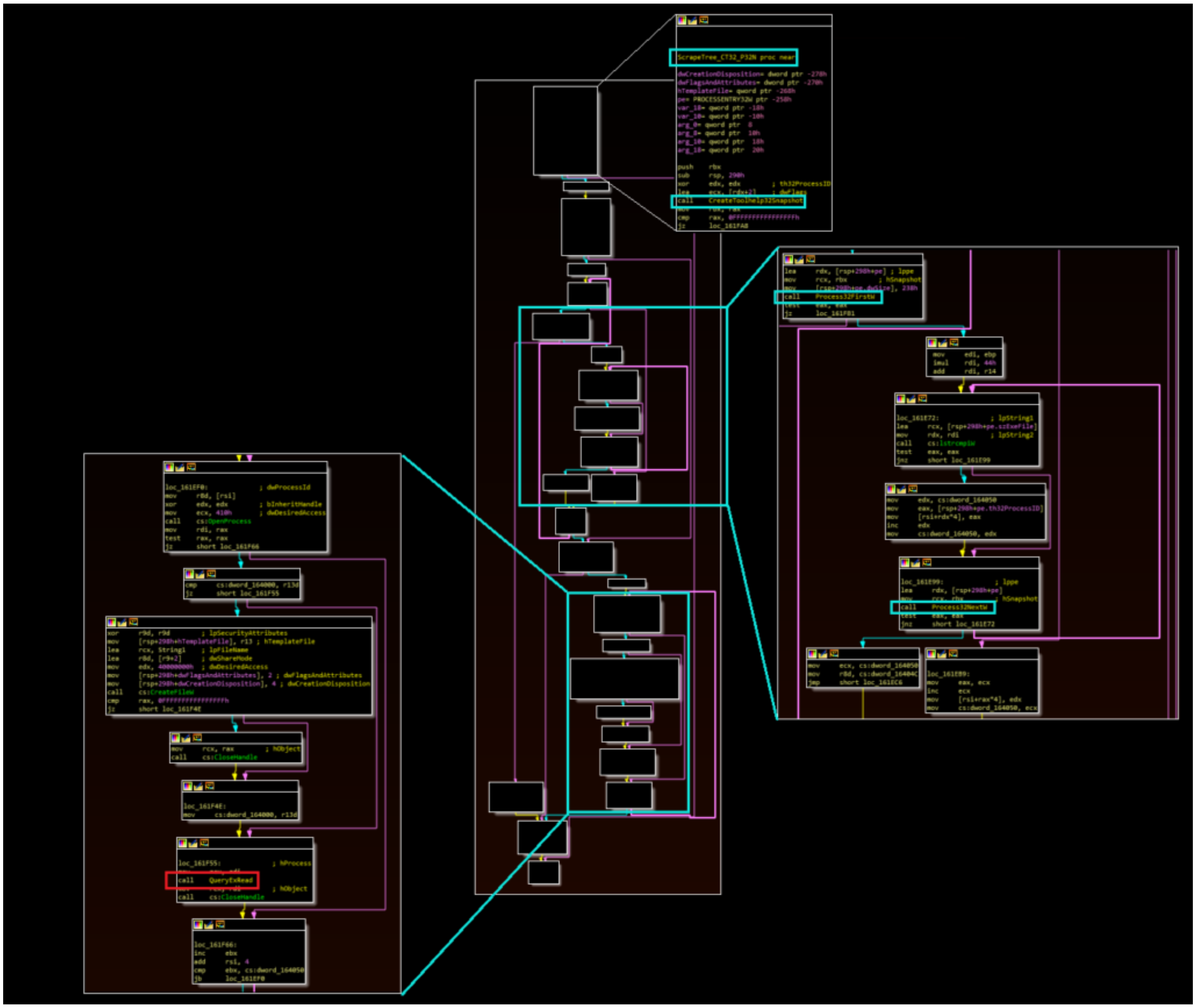
Second, the malware uses a form of process injection similar to a method described in open source reporting as the “Zberp” method. The malware uses CreateFileMappingA, MapViewOfFile, and NtMapViewOfSection to inject itself into the suspended svchost process. Finally, the malware uses NtQueApcThread and ResumeThread to run the main scraping loop.

Main Scraping Loop

The main scraping loop, which is either called directly through the “p” workflow or invoked through the other workflows as a created thread, represents the core of the malware’s functionality. Similar to the “i” routine, the main scraping loop calls a function that enumerates running processes (via CreateToolhelp32Snapshot, Process32First, and Process32Next) to identify a match with a specified target process (right click on the images below and open in a new tab to expand).



```
loc_162097:
call    ScrapeTree_CT32_P32N
mov     ecx, 5000       ; dwMilliseconds
call    cs:Sleep
lea     rcx, [rsp+48h+SystemTime] ; lpSystemTime
call    cs:GetSystemTime
lea     rdx, [rsp+48h+FileTime] ; lpFileTime
lea     rcx, [rsp+48h+SystemTime] ; lpSystemTime
call    cs:SystemTimeToFileTime
mov     eax, [rsp+48h+FileTime.dwHighDateTime]
mov     r11d, [rsp+48h+FileTime.dwLowDateTime]
mov     dword ptr [rsp+48h+hKey+4], eax
mov     dword ptr [rsp+48h+hKey], r11d
mov     rcx, [rsp+48h+hKey]
sub     rcx, cs:qword_164E80
mov     rax, rdi
imul    rcx
add     rdx, rcx
sar     rdx, 17h
mov     rax, rdx
shr     rax, 3Fh
add     rdx, rax
mov     eax, 88888889h
add     edx, ebx
mul     edx
shr     edx, 5
cmp     edx, cs:dword_164024
jb      short loc_162097
```



```
ScrapeTree_CT32_P32N proc near

dwCreationDisposition= dword ptr -278h
dwFlagsAndAttributes= dword ptr -270h
hTemplateFile= qword ptr -268h
pe= PROCESSENTRY32W ptr -258h
var_18= qword ptr -18h
var_10= qword ptr -10h
arg_0= qword ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h
arg_18= qword ptr  20h

push    rbx
sub     rsp, 290h
xor     edx, edx        ; th32ProcessID
lea     ecx, [rdx+2]    ; dwFlags
call    CreateToolhelp32Snapshot
mov     rbx, rax
cmp     rax, 0FFFFFFFFFFFFFFFFh
jz      loc_161FA8
```

```
lea     rdx, [rsp+298h+pe] ; lppe
mov     rcx, rbx        ; hSnapshot
mov     [rsp+298h+pe.dwSize], 238h
call    Process32FirstW
test    eax, eax
jz      loc_161F81
```

```
mov     edi, ebp
imul    rdi, 44h
add     rdi, r14
```

```
loc_161E72:             ; lpString1
lea     rcx, [rsp+298h+pe.szExeFile]
mov     rdx, rdi        ; lpString2
call    cs:lstrcmpiW
test    eax, eax
jnz     short loc_161E99
```

```
mov     edx, cs:dword_164050
mov     eax, [rsp+298h+pe.th32ProcessID]
mov     [rsi+rdx*4], eax
inc     edx
mov     cs:dword_164050, edx
```

```
loc_161E99:             ; lppe
lea     rdx, [rsp+298h+pe]
mov     rcx, rbx        ; hSnapshot
call    Process32NextW
test    eax, eax
jnz     short loc_161E72
```

```
mov     ecx, cs:dword_164050
mov     r8d, cs:dword_16404C
jmp     short loc_161EC6
```

```
loc_161E89:
mov     eax, ecx
inc     ecx
mov     [rsi+rax*4], edx
mov     cs:dword_164050, ecx
```

```
loc_161EF0:             ; dwProcessId
mov     r8d, [rsi]
xor     edx, edx        ; bInheritHandle
mov     ecx, 410h       ; dwDesiredAccess
call    cs:OpenProcess
mov     rdi, rax
test    rax, rax
jz      short loc_161F66
```

```
cmp     cs:dword_164000, r13d
jz      short loc_161F55
```

```
xor     r9d, r9d        ; lpSecurityAttributes
mov     [rsp+298h+hTemplateFile], r13 ; hTemplateFile
lea     rcx, String1    ; lpFileName
lea     r8d, [r9+2]     ; dwShareMode
mov     edx, 40000000h  ; dwDesiredAccess
mov     [rsp+298h+dwFlagsAndAttributes], 2 ; dwFlagsAndAttributes
mov     [rsp+298h+dwCreationDisposition], 4 ; dwCreationDisposition
call    cs:CreateFileW
cmp     rax, 0FFFFFFFFFFFFFFFFh
jz      short loc_161F4E
```

```
mov     rcx, rax        ; hObject
call    cs:CloseHandle
```

```
loc_161F4E:
mov     cs:dword_164000, r13d
```

```
loc_161F55:             ; hProcess
mov     rcx, rdi
call    QueryExRead
mov     rcx, rdi        ; hObject
call    cs:CloseHandle
```

```
loc_161F66:
inc     ebx
add     rsi, 4
cmp     ebx, cs:dword_164050
jb      loc_161EF0
```

If a process name is found that matches the target name, the malware calls the function boxed in red in the image above. The malware uses the VirtualQueryEx and ReadProcessMemory APIs to read the process, and then subsequently calls the actual data parsing routine. The malware looks for data formatted similarly to magnetic strip information. If found, the malware calls an additional function (referenced in five locations) to encode and write this data to a file located at “c:\users\public\music\wmsetup.tmp” and then repeats the loop.

Once the scraping is completed (or if the scraping fails), the malware can perform two additional cleanup functions before exiting. First, the malware deletes a registry entry located at Software\Microsoft\CurrentVersion\Run named PSMon. The malware can also delete a key named ODBC2 under Software*.

The purpose of these two keys is currently unknown. This blog speculates that both may be used as components of persistence mechanisms (perhaps with the former pointing to a script and the latter pointing to second-stage data stored in the registry). If this is the case, they may be named to mimic legitimate processes expected on these devices, such as the Unix Process Monitor tool and a SQL Database component (ODBC). It is also possible – but less likely – that this activity is designed to terminate these legitimate processes.

Additional Thoughts

While there are still some information gaps (particularly regarding the installer for this malware), this point of sale scraper represents a very different approach from the previously examined incident. Whereas that file scraped the memory of every process on a system, PoSlurp.B is designed for a more targeted approach. This suggests that the attackers conducted sufficient reconnaissance within the environment to determine where credit card data was likely to be held (or knew this information prior to the intrusion).

Analysis Tips

Analyzing this file proved particularly challenging, given the high number of conditional exits and the need for the malware to successfully parse an environment variable. Ultimately, I can recommend the following approach:

The hash 82953a819daff3a81e678c75ce7736b3 contains a PowerShell byte array loader that I found during a search for other FIN8 malware (whether or not it is actually affiliated with this group, I have not checked).

– Take the shellcode, open it in a hex editor (e.g. HxD), and copy the hex into a text editor (Notepad++)
– Replace the spaces from the hex bytes with a ",0x"
– Add a leading "0x" to the first bytes
– Add an additional two bytes, 0xEB and 0xFE, to the start of the file. This is an infinite loop.
– Replace the payload bytes in the hash above with these bytes
– Add the environment variable
– Run the PowerShell file
– In x64dbg, attach to the PowerShell file
– Resume the program
– Look in the memory map for the executable section of memory
– Set a breakpoint at this section
– NOP the infinite jump instruction
– Begin debugging

The idea here is to get PowerShell to load the shellcode, but to do so in a way in which is doesn't execute. EB FE is a shorthand for an infinite loop in which the malware jumps to the jumping instruction. The malware will run this indefinitely, until you manually place a breakpoint there. Programs such as jmp2it will do this automatically, but I ran into issues attaching to it in a 64-bit debugger. A few other creative approaches (side-loading in place of Chinese APT shellcode, injecting it into other processes) came up short. They also didn't allow the malware to recognize an environment variable.

For simply statically analyzing the shellcode and its subsequent payload, I'd recommend Adam's approach. It looks like a lot of steps, but it only takes a few minutes, and you can build a 64-bit executable that's pretty easy to directly debug (and subsequently dump a payload from).