Malware Analysis Report: Process Hollowing and Fileless PowerShell Loader

Analyst: Mangesh, Cyberstanc

Executive Summary

A suspicious **notepad.exe** process (PID 712) and a fileless PowerShell loader were investigated. Memory forensics revealed that notepad.exe had an RWX-memory section with injected shellcode – a classic *process hollowing* scenario [1] [2]. The hollowed payload (dumped via Volatility) had a high anomaly score in PEStudio: multiple high-entropy sections, unusual section names, no valid digital signature, and critical imports (e.g. `VirtualAlloc`, `CreateRemoteThread`, `WriteProcessMemory`) used by injection techniques.

Separately, a dropper binary `~temp1.exe` spawned a hidden PowerShell process using `cmd.exe /C powershell.exe -WindowStyle Hidden -EncodedCommand ...`. The encoded script decoded an ASCII-hex blob into a file named `test.exe` and launched it. It then **reflectively loaded** another PE from a .NET memory stream – allocating executable memory (`VirtualAlloc`), copying bytes, and creating a thread (`CreateThread`) to run it without writing the payload to disk (fileless execution).

Combined, these findings show advanced stealth tactics: process injection (T1055.012) and in-memory (fileless) execution (T1620) [2] [3]. The report below details the tools and methods used, step-by-step analysis of each component, ATT&CK mapping, IOCs, conclusions, and defensive recommendations.

Tools Used

- **Volatility** (memory forensics) – Plugins: `pslist`, `malfind`, `strings`, `procdump`.
- **PEStudio** (static PE analysis) – Flagged anomalies, imports, signatures.
- **Strings utility** – Quick extraction of text from binaries.
- **Hex Editor / IDA** – (if needed for low-level inspection).
- **Windows Command Shell & PowerShell** – To reconstruct and analyze encoded scripts.

Analysis Steps

1. Memory Injection Analysis (Volatility & PEStudio)

- **Process Enumeration:** Using Volatility's `pslist`, the notepad.exe (PID 712) process was identified running on the memory image. Its parent and siblings were normal, but malfind flagged it as suspicious due to unusual memory attributes.

- **Malfind (Injected Code Detection):**

```
volatility -f memory.img --profile=Win7SP1x64 malfind -p 712
```

This found a **PAGE_EXECUTE_READWRITE (RWX)** region in notepad.exe that is *not mapped to any on-disk file*. The dump included an "MZ" header at the start of this region – indicating a PE image injected into memory [4]. (Volatility's malfind specifically reports RWX pages with no backing file as likely injected code.) For example:

```
-- Found a possible injected section (not mapped on disk) in PID 712:
   [Addr             ] [MajorMin] [Size    ] [Code/Data] [Name]
   0x00007ff7`43c05000 0x0000 0x0000 rwx  ???    (contains "MZ" header)
```

The presence of an "MZ" header in a RWX page is **very unusual for benign processes** [4] [5], strongly suggesting process hollowing.

- **Memory Strings:** A `strings` scan on that memory region revealed Windows API names typically used in code injection (e.g., `VirtualAlloc`, `CreateRemoteThread`, `SetThreadContext`). These strings embedded in memory reinforce that **shellcode or a PE was prepared for execution**.

- **Process Dump:**

```
volatility -f memory.img --profile=Win7SP1x64 procdump -p 712 -D dumps/
```

The injected code was dumped as a standalone binary. This file, `notepad_712.exe`, was then opened in PEStudio.

- **Static PE Analysis (PEStudio):** The dumped executable exhibited multiple red flags:

- **High Anomaly Score:** PEStudio flagged many anomalies (unusual section flags, overlapping addresses, etc.), resulting in a high suspicion score. This tool computes an "anomaly score" by checking for inconsistencies in headers and sections [6].
- **Suspicious Imports:** It imported low-level APIs like `VirtualAlloc`, `WriteProcessMemory`, `CreateRemoteThread`, `ZwUnmapViewOfSection`, etc. These are common for injection/hollowing (allocating memory and spawning threads) [7] [2].
- **High-Entropy Sections:** Several sections had near-max entropy, indicating compressed or encrypted data (a common obfuscation in malware) [8]. For example, an embedded code section had entropy ~7.9/8.0. High entropy alone isn't conclusive, but *malware often contains encrypted blocks* [8].
- **No Digital Signature:** Legitimate Windows binaries are usually signed. This payload had **no Authenticode signature**. (Malware often ships unsigned or with invalid signatures to masquerade [9].)
- **Weird Sections/Strings:** Unusual section names (e.g. "`.myzblz`") and odd version/compile timestamps were observed – both typical in packed or faked binaries.

These findings confirm that `notepad.exe` was hollowed out to run malware code (Process Injection – T1055.012 [2]). The anomalous imports (`VirtualAlloc`, `CreateRemoteThread`, etc.) precisely match

the technique described by trusted sources [7] [2]. For example, SentinelOne notes that code injection typically uses `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread` to carve out space and run code in another process [7].

2. Fileless PowerShell Loader Analysis

- **Execution of Loader (`~temp1.exe`):** The binary `~temp1.exe` was found on disk (likely via forensic triage). Running it launches a hidden PowerShell process. Its command line (from Sysinternals ProcMon logs) was:

```
cmd.exe /C powershell.exe -NoProfile -WindowStyle Hidden -EncodedCommand
<Base64String>
```

  This indicates it used `cmd.exe` to invoke PowerShell with a base64-encoded script, all in a concealed window. Malware commonly abuses flags like `-WindowStyle Hidden` or `-nop` to avoid alerting users [10].

- **PowerShell Decoding:** The encoded script was recovered and decoded. It performed two main actions:

- **Reconstruct `test.exe`:** It took a long string of ASCII-hex bytes and converted them into a new file. In pseudocode, it did something like:

```
$hex = "4D5A...909090...";  # hex string
$bytes = for ($i=0; $i -lt $hex.Length; $i+=2) {
[Convert]::ToByte($hex.Substring($i,2), 16) }
[System.IO.File]::WriteAllBytes("test.exe", $bytes)
Start-Process "test.exe"
```

  This resurrects an EXE from encoded bytes. Base16/hex or Base64 encoding of payloads is a known obfuscation (ATT&CK T1027) [11] [10].

- **In-Memory Payload Loading:** After launching `test.exe`, the script used .NET interop to load another PE directly into memory. It did roughly:

```
$payloadBytes = [System.Convert]::FromBase64String("<PE file content>")
$ms = New-Object System.IO.MemoryStream($payloadBytes)
# Define unmanaged API calls
$sig = @'
  [DllImport("kernel32.dll")]
  public static extern IntPtr VirtualAlloc(IntPtr addr, UIntPtr size, uint
type, uint protect);
  [DllImport("kernel32.dll")]
  public static extern IntPtr CreateThread(IntPtr attr, uint stackSize,
IntPtr start, IntPtr param, uint flags, out IntPtr threadId);
```

```
'@
Add-Type -MemberDefinition $sig -Name Win32 -Namespace Kernel32
$alloc = [Kernel32.Win32]::VirtualAlloc(0, $ms.Length, 0x3000, 0x40)
# RWX memory
$ptr = $alloc.ToInt64()
$copy = $payloadBytes.Length
[System.Runtime.InteropServices.Marshal]::Copy($payloadBytes, 0, $alloc,
$copy)
[Kernel32.Win32]::CreateThread(0,0,$alloc,0,0,[ref]0)
```

In effect, this allocates executable memory (with `flProtect=0x40`, i.e. PAGE_EXECUTE_READWRITE), copies the decoded PE bytes into it, and spawns a thread at the entry point [12] [13]. No file is written; the payload runs entirely in RAM.

- **Behavioral Insights:** Using `VirtualAlloc` + `memcpy` + `CreateThread` in PowerShell is a classic shellcode or reflective loader pattern [12] [14]. Security docs warn that seeing these API calls in PowerShell is highly suspicious [14]. The loader script made extensive use of `Add-Type` and DllImport definitions for kernel32 functions, which is not normal for benign scripts. Once loaded, the in-memory PE acted like any other process but had no disk footprint beyond the ephemeral runner script.

Overall, the fileless component demonstrates *Reflective Code Loading* (ATT&CK T1620): loading and executing a payload in memory without touching disk [3]. Combined with obfuscation (encoded commands) and hidden execution, this evades many traditional defenses.

MITRE ATT&CK Mapping

Technique ID	Tactic	Description (Context)
T1055.012 – Process Injection: **Process Hollowing**	Defense Evasion	Malicious code injected into a suspended process (notepad.exe) by unmapping its memory and replacing it [2]. The abnormal RWX section in notepad.exe indicates this technique.
T1059.001 – Command and Scripting Interpreter: **PowerShell**	Execution	Execution of malicious payload via PowerShell. The launcher used `powershell.exe -WindowStyle Hidden -EncodedCommand ...` [10] to covertly run a script.
T1059.003 – Command and Scripting Interpreter: **Windows Command Shell**	Execution	Use of `cmd.exe /C powershell.exe ...` to invoke PowerShell. The dropper executed the PowerShell loader through the Windows shell.
T1027.002 – Obfuscated Files/Information: **Base64**	Defense Evasion	Payload and scripts were encoded (Base64 and ASCII-hex) to hide intent. This includes the long hex strings for `test.exe` and the Base64-encoded in-memory PE.

Technique ID	Tactic	Description (Context)
T1620 – Reflective Code Loading	Defense Evasion	The payload was loaded directly into process memory (via VirtualAlloc/Copy/Thread) instead of from disk [3]. This evades disk-based detection by running code straight from RAM.

Indicators of Compromise (IOCs)

- **Suspicious Processes:**
- `notepad.exe` (PID 712) with an unexpected RWX memory segment (memory injection).
- Hidden PowerShell process launched by `cmd.exe`, as indicated by process creation logs (e.g. `powershell.exe -WindowStyle Hidden -EncodedCommand ...`).
- **File Artifacts:**
- `~temp1.exe` – untrusted loader binary.
- `test.exe` – reconstructed payload launched by PowerShell.
- **Command/Script Patterns:**
- Powershell command lines using `-WindowStyle Hidden -EncodedCommand` [10] or `-nop -w hidden -c IEX ...`.
- `Start-Process` used to launch executables (e.g. `Start-Process "test.exe"`).
- Presence of `.NET Add-Type` DllImport definitions for `kernel32.dll` in scriptblocks.
- **Memory/Execution Artifacts:**
- Volatility malfind output showing an **MZ header in a RWX page** of notepad.exe (indicative of an unmapped, injected module) [4].
- Imported API names in dumped binary: `VirtualAlloc`, `WriteProcessMemory`, `CreateRemoteThread`, etc. [7].
- High-entropy PE sections and missing Authenticode signature on the dumped payload.
- Encoded payload strings (long Base64 or hex blobs) in PowerShell scripts.
- **Network Indicators:** None identified in this offline analysis (no C2 domains/IPs were observed).

Conclusion

The analysis uncovered a two-pronged attack involving *process injection* and *fileless execution*. The malicious use of **Process Hollowing** in notepad.exe (with RWX memory and shellcode) highlights sophisticated injection techniques [2]. The separate PowerShell-based loader demonstrates **reflective in-memory execution**: the attacker hid payloads in encoded form and ran them without writing files to disk [3].

Together, these techniques fall under MITRE ATT&CK tactics of Defense Evasion and Execution (T1055, T1620, T1059, T1027).

Both components leveraged legitimate system tools (notepad, cmd, PowerShell) to mask malicious behavior. The evidence (malfind output, PEStudio anomalies, PowerShell API calls) is consistent with advanced malware aiming to blend in and avoid detection.