Deep Dive into a Dumped Malware without a PE Header

fortinet.com/blog/threat-research/deep-dive-into-a-dumped-malware-without-a-pe-header

May 29, 2025

Article Contents

By Xiaopeng Zhang and John Simmons | May 29, 2025

Background

This analysis is part of an incident investigation led by the FortiGuard Incident Response Team.

We discovered malware that had been running on a compromised machine for several weeks. The threat actor had executed a batch of scripts and PowerShell to run the malware in a Windows process. Although obtaining the original malware executable was difficult, a memory dump of the running malware process and a full memory dump of the compromised machine (the “fullout” file, size 33GB) were successfully acquired.

Figure 1: Full memory dump file of the compromised machine.
Figure 1 provides detailed file information about the dumped memory file, “fullout,” which we scanned to build a local test environment for analyzing the malware.

[2025 Global Threat Landscape Report]

[Use this report to understand the latest attacker tactics, assess your exposure, and prioritize action before the next exploit hits your environment.]

The Dumped Malware File

The malware was running within a dllhost.exe process with PID 8200. The dumped file is named pid.8200.vad.0x1c3eefb0000-0x1c3ef029fff.dmp. The file name reveals that the malware was loaded and deployed in memory at address range 0x1c3eefb0000 to 0x1c3ef029fff.

Figure 2: View of the dumped malware file.

The dumped file is a deployed 64-bit PE (Portable Executable) file. During execution, the Windows Loader reads and parses its DOS and PE headers to load and deploy the PE file. Once deployed, these headers are no longer needed. To evade dumping the malware into a file for analysis by researchers, some malware often corrupts these header regions by overwriting them with zeros (like this one) or random data. As shown in Figure 2, both the DOS and PE headers are corrupted, making it difficult to reconstruct the entire executable from memory.

Deploying the Dumped Malware Locally

To dynamically analyze the malware, we needed to replicate the compromised system's environment locally. This required launching the dllhost.exe process in a debugger to serve as a target process for deploying the dumped malware. This would allow us to analyze the malware within a local analysis environment.

Preparing the malware to execute properly in this controlled setting involves several complicated steps.

Locating the Entry Point

The first step is to locate the entry point function (the start function), which is the initial code executed when the malware is loaded into memory by the Windows Loader.

While the offset of the entry point function is typically stored in the PE header, this was not the case. Instead, we had to manually locate the entry point (the start function). Based on our experience, the first instruction of the entry function is typically compiled as "sub rsp, 28h," but other functions may also contain this instruction. However, by dumping the malware in IDA Pro, we were able to search for all occurrences of this instruction in the IDA Pro database.

Fortunately, only eight instances of the instruction appeared in the malware (Figure 3). After analysis, we confirmed that the fourth function (at 0x1C3EEFEE0A8) is the entry point.

Figure 3: Locating the entry point function.

Allocating the Major Memory

In a newly launched dllhost.exe process, we manually executed some instructions to allocate memory for deploying the dumped malware, as seen in Figure 4. It calls a relevant VirtualAlloc() API with the same base address—0x1C3EEB70000—as seen in the compromised system.

Figure 4: Newly allocated memory in a dllhost.exe.

Once allocated, the dumped malware was copied into the newly created memory.

Resolving the Import Table

A PE file's Import table lists the Windows APIs it depends on. These API loading addresses differ on different Windows systems. To run and analyze the dumped malware in the local system, these addresses needed to be relocated to the ones loaded in the local system.

Figure 5: Partial view of the malware's Import Table.
Figure 5 shows part of the Windows API addresses from the Import Table. Based on our analysis, the final API address can be calculated from this information.

For instance, the API address at 0x1C3EF0240D0 is 0x1C3EEEE1CE0, as shown in Figure 5. It calculates the API address as 0x7FFD74224630 by executing the following ASM code at address 0x1C3EEEE1CE0h:

```
001C3EEEE1CE0   mov  r10, 0E528F49552F112B4h

001C3EEEE1CEA    mov r11, 0E5288B6826D35484h

001C3EEEE1CF4    xor r11, r10

001C3EEEE1CF7    jmp r11    ; 0x7FFD74224630
```

Using the Volatility tool, we listed the loaded modules in the dllhost.exe process (PID 8200) from the "fullout" file. As you can see in Figure 6, the API at 0x7FFD74224630 is exported from module GDI32.dll.

Figure 6: Loaded module list in dllhost.exe.
By dumping the GDI32.dll from the "fullout" file and analyzing it, we determined that the API at address 0x7FFD74224630 corresponds to GetObjectW() in the compromised system.

In our local test environment, this same API resides at address 0x07FFFF77CB870. For this API, its original address was replaced with the local address.

This malware has 257 Windows APIs requiring relocation across 16 modules, including:

- kernel32.dll
- ws2_32.dll
- ntdll.dll
- gdi32.dll
- shlwapi.dll
- sspicli.dll

- user32.dll
- shell32.dll
- msvcrt.dll
- advapi32.dll,
- comctl32.dll
- crypt32.dll
- gdiplus.dll
- ole32.dll
- rpcrt4.dll
- userenv.dll

For each API in the Import Table, its original address must be replaced with its corresponding local address using the same method used for API GetObjectW().

> In addition, we needed to load all the required modules that were not automatically loaded by dllhost.exe. To do this, the API LoadLibraryA() or LoadLibraryW() with the module name must be called to load them into the malware's memory.
>
> Figure 7 shows the malware running in a debugger. The RIP register points to the entry point's address at 0x1C3EEFEE0A8. The debugger also shows some of the relocated Windows API functions at the bottom.

Figure 7: Break at the entry point function with the fixed API table.

Allocating More Memory

Based on our analysis, the malware also required some global variable data located at address 0x1C3EEB7000 with a size of 0x5A000 bytes.

We extracted the required global data from the "fullout" file using the Volatility tool and the dd command. We again called the VirtualAlloc() API to allocate a new memory region within the dllhost.exe process at the desired address and size. After successful allocation, the extracted data was copied into the newly allocated memory space, as shown in Figure 8.

Figure 8: Copied global variable data starting from the address 0x1C3EEB7000.

Fixing the Parameters to the Entry Point and Stack

Static analysis of the malware's entry point function reveals that the function requires three parameters.

1. The first parameter (RCX) is the base address of the loaded malware, which in this case is 0x1C3EEFB0000.

2. The value of the second parameter (RDX) is 0x1.

3. The third parameter (R8) is a pointer to a 0x30-byte buffer, which can also be extracted from the “fullout” file.

We prepared all three parameters accordingly and allocated additional memory to store the 30H data for the third parameter, as illustrated in Figure 9.

Figure 9: Prepared all three parameters and the adjusted RSP register.
The last critical item is the correct alignment of the RSP register. When breaking at the entry point, the lowest four bits of the RSP value must be 0x8, as shown in Figure 9. Failing to align the RSP properly can trigger an exception of EXCEPTION_ACCESS_VIOLATION (code 0xC0000005) when the malware starts.

This is due to a misalignment error, particularly when executing instructions like “movdqa,” which requires 16-byte alignment. In 64-bit code mode, before calling the entry point function, the RSP’s value is 16-byte aligned (the lowest four bits are 0x0). It pushes the return address onto the stack, and the RSP value is minus eight. To correct this, we adjusted the value from 0x09CAD11D850 to 0x09CAD11D848.

Analyzing the Malware

After multiple trials, errors, and repeated fixes, we finally managed to run the malware in the local environment.

Upon execution, the malware calls a function to decrypt its C2 server domain information stored in memory. As shown in Figure 10, the decryption function is displayed along with the newly decrypted domain details, including the domain ("rushpapers.com") and port number ("443").

Figure 10: Just decrypted C2 server information.
The malware then establishes communication with its C2 server by creating a thread. As shown in Figure 11, it prepares to invoke the CreateThread() API with the thread function at 0x1C3EEFDE300.

Figure 11: A thread about to be created for C2 communication.
The newly created thread is responsible for handling communication with its C2 server. Part of the thread’s code is shown on the right side of Figure 11. After launching the thread, the main thread enters a sleep state until the communication thread completes its execution.

Communicating with the C2 Server

As the decrypted domain port “443” indicates, the malware communicates with the C2 server over the TLS protocol. It uses the getaddrinfo() API to obtain the domain “rushpapers.com” IP address via a DNS query.

Figure 12 is a Wireshark capture of the network traffic generated by the malware as it communicates with its C2 server.

Figure 12: The network packets exchanged between the malware and the C2 server.
Since the TLS packet is encrypted, we need to inspect the data before encryption or after decryption to view the plaintext content. This can be done by setting breakpoints in a debugger on both encryption and decryption routines used by this malware.

The malware leverages two API functions, SealMessage() and DecryptMessage(), to encrypt and decrypt the data for the TLS traffic. As you can see in Figure 13, the malware is preparing to encrypt an HTTP GET request using the SealMessage() API.

Figure 13: Preparing to encrypt the GET request using SealMessage().
Below are two examples of plaintext packets--one sent and one received--:

Request packet (before TLS encryption):

GET /ws/ HTTP/1.1

Host: rushpapers[.]com

Connection: Upgrade

Upgrade: websocket

Sec-WebSocket-Version: 13

Sec-WebSocket-Key: OCnq155rYct3ykkkdLrjvQ==

Response packet (after TLS decryption):

HTTP/1.1 101 Switching Protocols

Server: nginx/1.18.0

Date: Fri, 28 Mar 2025 06:13:24 GMT

Connection: upgrade

Upgrade: websocket

Sec-WebSocket-Accept: Bzr0K1o6RJ4bYvvm4AM5AAG172Y=

These two plaintext packets are used to complete a handshake-like process. Afterward, it switches
to a custom encryption algorithm to encrypt the packet data before applying TLS encryption.

Below is an example of the data that has been encrypted using the custom algorithm.

00000000 82 A6 16 98 5C 75 59 CB 66 55 41 F1 32 11 79 EF ,¦ ˜\uYËfUAñ2 yï

00000010 2F 55 27 A8 7C 5A 36 AE 68 58 74 F1 28 55 3E A9 /U'¨|Z6®hXtñ(U>©

00000020 6C 5B 26 B6 6D 4C 26 AC 69 5C 1B 92 l[&¶mL&¬i\ ’

A clearer breakdown of the data is in the table below:

Offset	Length	Description
00	01	Magic tag. 0x82
01	Variable	Variable-extended-length. 0xA6 for this case.
02	04	Encryption key. 16 98 5C 75
06	26H	Encrypted data. 59 CB 66 … 1B 92

According to the variable-extended-length rule, the data size is calculated as 0xA6-0x80=0x26.

The encryption key (such as 0x755C9816) is a randomly generated number. The custom encryption's algorithm performs a repeated XOR operation between each byte of the key and the encrypted data bytes.

As a result, the decrypted data is as follows:

```
00000000   4F 53 3A 20 57 69 6E 64  6F 77 73 20 31 30 20 2F   OS: Windows 10 /
00000010   20 36 34 2D 62 69 74 20  28 31 30 2E 30 2E 31 39   64-bit (10.0.19
00000020   30 34 35 29 0D 0A                                  045)
```

The decrypted data clearly reveals the system information from our local test environment: "OS: Windows 10 / 64-bit (10.0.19045)\r\n". This information is collected and sent to the C2 server when requested by the C2 server.

Below is a code snippet demonstrating the custom algorithm used to encrypt and decrypt the data.

```
[…]
001C3EF00CED3 loc_1C3EF00CED3:    ; CODE XREF: sub_1C3EF00CE08+FD↓j
001C3EF00CED3                mov    eax, r9d
001C3EF00CED6                and    eax, 80000003h
001C3EF00CEDB                jge    short loc_1C3EF00CEE4
001C3EF00CEDD                dec    eax
001C3EF00CEDF                or     eax, 0FFFFFFFCh
001C3EF00CEE2                inc    eax
001C3EF00CEE4
001C3EF00CEE4 loc_1C3EF00CEE4:  ; CODE XREF: sub_1C3EF00CE08+D3↑j
001C3EF00CEE4                mov    ecx, [rbx+30h]
```

```
001C3EF00CEE7              add     ecx, r9d

001C3EF00CEEA              cdqe

001C3EF00CEEC              inc     r9d

001C3EF00CEEF              mov     r8b, byte ptr [rsp+rax+28h+arg_0] ;; the random key

001C3EF00CEF4xorr8b, [r10]  ;;;;;;; encrypt/decrypt the data with random key.

001C3EF00CEF7              inc     r10

001C3EF00CEFA              mov     rax, [rbx+20h]

001C3EF00CEFE              mov     [rcx+rax], r8b

001C3EF00CF02              cmp     r9d, edi        ; edi is data size

001C3EF00CF05              jl      short loc_1C3EF00CED3

001C3EF00CF07

001C3EF00CF07 loc_1C3EF00CF07:  ; CODE XREF: sub_1C3EF00CE08+C6↑j

001C3EF00CF07              add     [rbx+30h], edi

001C3EF00CF0A              jmp     loc_1C3EF00CE55

[…]
```

Feature Analysis

Through a comprehensive analysis of its API calls and execution flow, we have confirmed this malware to be a RAT (Remote Access Trojan). This section details the malware's capabilities for controlling the compromised system.

Screenshot capture

The malware has a feature that captures the victim's screen as JPEG images and exfiltrates them to its C2 server. It also collects the title of the current active (topmost) program to provide context about what the user is doing at the time of capture.

To do this, it calls a sequence of APIs, including CreateStreamOnHGlobal(), GdiplusStartup(), GetSystemMetrics(), CreateCompatibleDC(), CreateCompatibleBitmap(), BitBlt(), GdipCreateBitmapFromHBITMAP(), GdipSaveImageToStream(), and

GdipDisposeImage().

Figure 14 shows how the malware captures the screenshot by calling these APIs.

Figure 14: The pseudocode of the function capturing a screenshot.

Acting as a server

The malware includes a thread function designed to act as a server, listening on a TCP port specified by the C2 server. Once activated, this function allows the malware to await incoming connections from the attacker.

It implements a multi-threaded socket architecture: each time a new client (attacker) connects, the malware spawns a new thread to handle the communication. This design enables concurrent sessions and supports more complex interactions.

By operating in this mode, the malware effectively turns the compromised system into a remote-access platform, allowing the attacker to launch further attacks or perform various actions on behalf of the victim.

Control system services

The malware can enumerate and manipulate the system services on the infected machine. It achieves this by leveraging several Windows Service Control Manager (SCM) APIs, including OpenSCManagerW(), EnumServicesStatusExW(), ControlService(), and more.

Conclusion

This analysis successfully demonstrated the deployment and dynamic analysis of malware with corrupted DOS and PE headers in a controlled local environment.

The detailed process—from preparing the malware for execution, including memory allocation and API resolution, to correcting execution parameters—ensured accurate emulation of the malware's behavior.

Our investigation into the payload revealed its sophisticated communication with the C2 server, including secure encryption and decryption mechanisms using the SealMessage() and DecryptMessage() APIs.

Finally, we confirmed the malware's significant capabilities on the compromised system, such as screen capture, remote server functionality, and manipulation of system services via Service Control Manager APIs.

Fortinet protects against these attacks, and the FortiGuard IR team is available to assist you whenever needed.

Fortinet Protections

Fortinet customers are already protected from this malware with FortiGuard's AntiVirus, Web Filtering, and Anti-Botnet services as follows:

- The FortiGuard Anti-Botnet Service blocks the DNS requests for the C2 server domain.
- FortiGate blocks the malicious TLS certificate used by the C2 server.

The relevant C2 server URL is rated as "**Malicious Websites**" by the FortiGuard Web Filtering service.

FortiGate, FortiMail, FortiClient, and FortiEDR support the FortiGuard AntiVirus service. The FortiGuard AntiVirus engine is part of each solution.

As a result, customers who have these products with up-to-date protections are already protected.

You can sign up for future alerts and stay informed of new and emerging threats.

We also suggest our readers go through the free NSE training: NSE 1 – Information Security Awareness, a module on Internet threats designed to help end users learn how to identify and protect themselves from phishing attacks.

If you believe this or any other cybersecurity threat has impacted your organization, please contact our Global FortiGuard Incident Response Team.

IOCs

URLs

hxxps[:]//rushpapers[.]com/ws/

Sha256

F3EB67B8DDAC2732BB8DCC07C0B7BC307F618A0A684520A04CFC817D8D0947B9