

From XLoader to Phantom Stealer: A DHL-Themed multi-stage Infection Chain

 medium.com/@mk7912/from-xloader-to-phantom-stealer-a-dhl-themed-multi-stage-infection-chain-5d552eee828d

Manoj Kshirsagar

January 27, 2026

Executive Summary

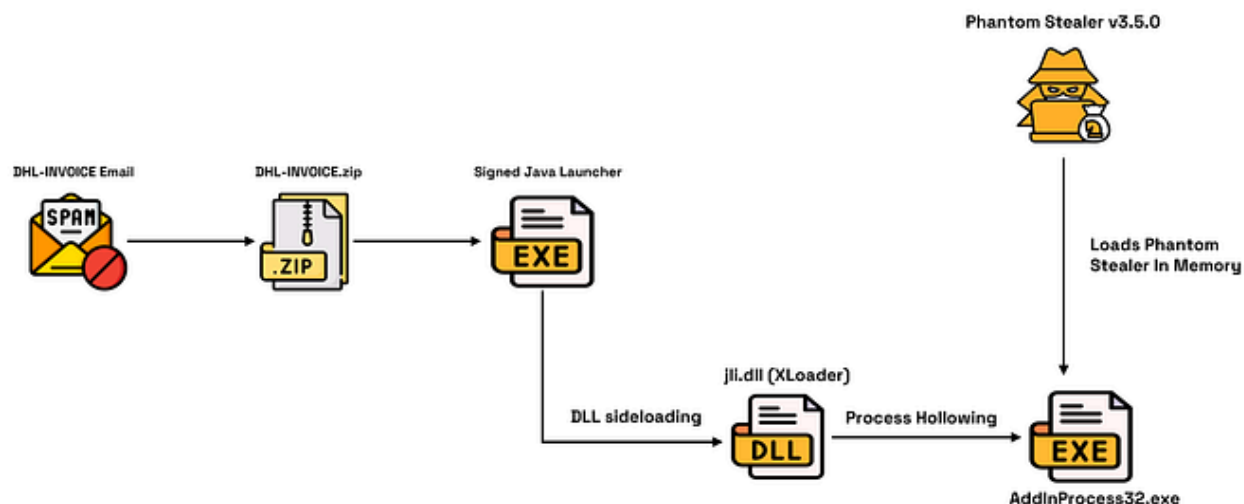
I have identified a multi-stage malware campaign delivering **Phantom Stealer v3.5.0** via a DHL-themed spam email. Initial access occurs through a ZIP attachment masquerading as an invoice and requiring user execution.

The archive drops a signed Java utility (**jdeps.exe**, renamed as *DHL-INVOICE.exe*) alongside a malicious *jli.dll*, identified as **XLoader**. Execution is achieved through **DLL sideloading**, allowing the trusted Java launcher to load the malicious DLL and transfer execution to the XLoader.

XLoader employs obfuscated, state-driven logic to parse embedded configuration, manually map an encrypted PE payload into memory, and avoid standard Windows loading APIs. The payload is injected via **process hollowing** into *AddInProcess32.exe*, enabling execution under a legitimate Microsoft process.

The final stage is confirmed as **Phantom Stealer v3.5.0**, a modular .NET-based information stealer supporting credential theft and multi-channel data exfiltration, demonstrating a mature and stealth-focused delivery chain.

Attack Chain



Initial Vector: Spam Email

It's been observed that the infection starts with a spam email that pretends to be sent by **DHL Express** as shown in **Figure 2**. The subject of the email is “**DHL INVOICE**”, which is commonly used in real shipment notifications and does not immediately raise suspicion. The email claims that a package has arrived and will be shipped within the next **48 hours**. The recipient is asked to open the attached file to confirm the consignee name and delivery address. The message is short, generic, and does not include any specific tracking number or personal details, which indicates that the campaign is sent in bulk.

The sender address appears as ‘, and the email body uses simple DHL-style wording. This is enough for many users to trust the message, especially in corporate environments where DHL shipments are common. The email contains a ZIP attachment named **DHL-INVOICE.zip**, shown as an invoice document. This helps the attachment bypass basic email filtering and appear harmless to the victim.

The malware execution only happens if the user downloads and opens the ZIP file. After extraction, the next stage of the Phantom Stealer infection begins with the execution of the embedded XLoader.

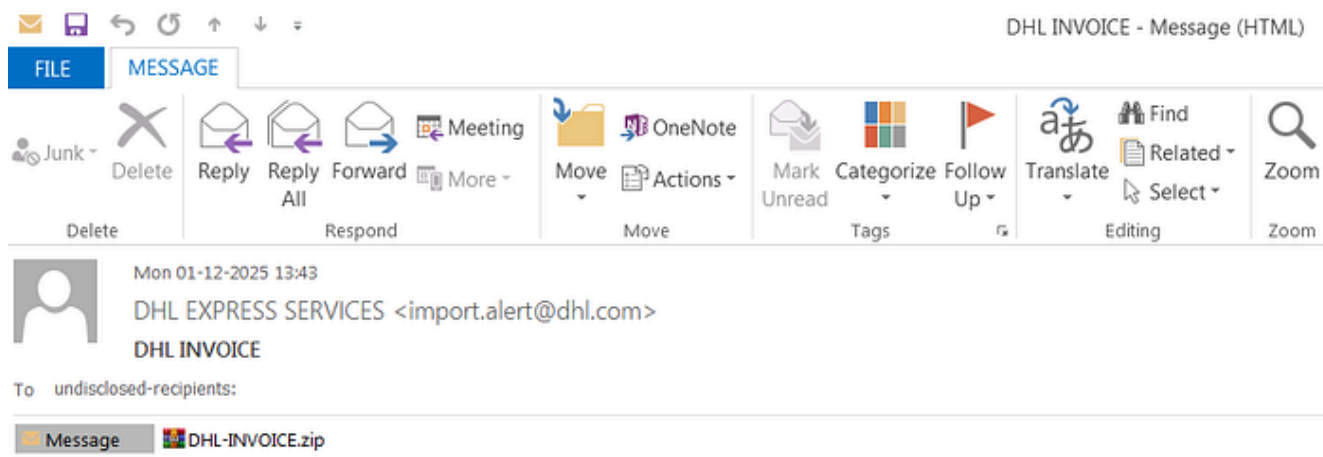


Figure 2 DHL-themed spam email with a malicious ZIP attachment used for initial access.

Technical Analysis

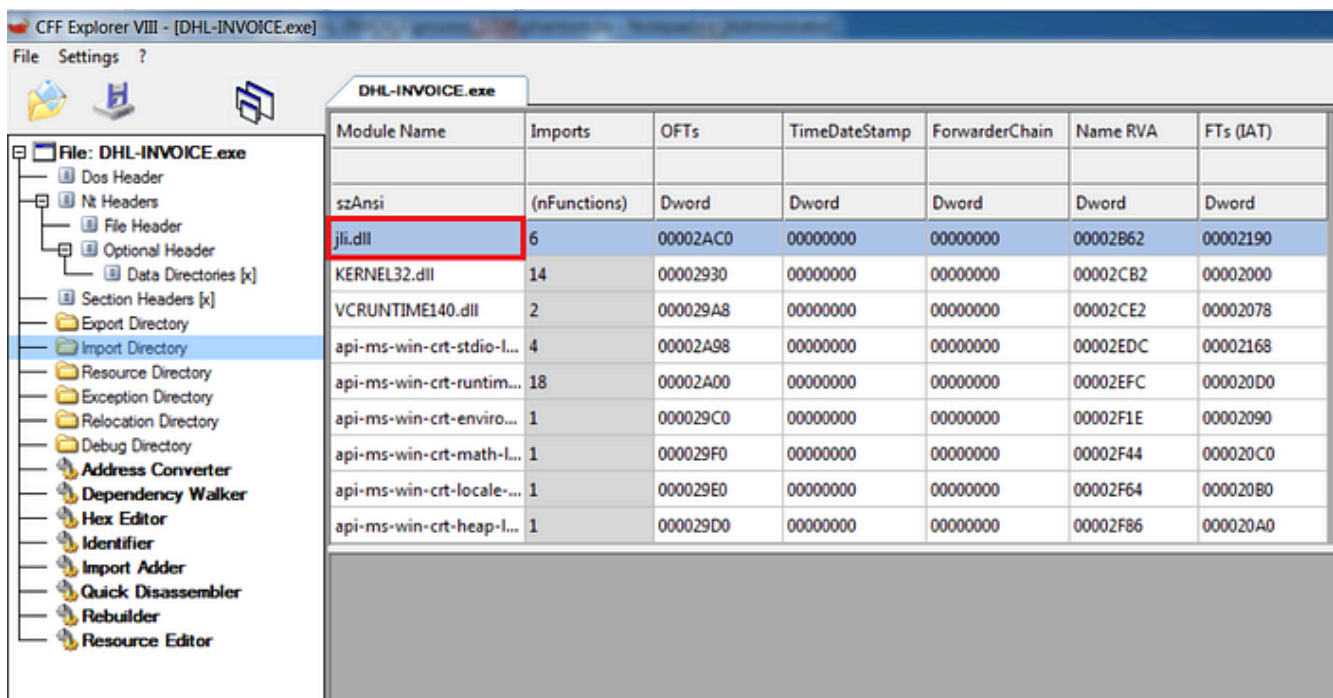
Once the ZIP file is extracted, it reveals an executable named **DHL-INVOICE.exe** along with multiple runtime DLLs required for execution as shown in **Figure 3**. DHL-INVOICE.exe initially appeared legitimate. The file is digitally signed by **Azul Systems, Inc.**, a vendor known for distributing **Zulu OpenJDK**. This setup helps the malware run reliably on the victim's system.

api-ms-win-crt-convert-l1-1-0.dll	14-10-2023 17:08	Application extens...	25 KB
api-ms-win-crt-environment-l1-1-0.dll	14-10-2023 17:08	Application extens...	21 KB
api-ms-win-crt-filesystem-l1-1-0.dll	14-10-2023 17:08	Application extens...	21 KB
api-ms-win-crt-heap-l1-1-0.dll	14-10-2023 17:08	Application extens...	21 KB
api-ms-win-crt-locale-l1-1-0.dll	14-10-2023 17:08	Application extens...	21 KB
api-ms-win-crt-math-l1-1-0.dll	14-10-2023 17:08	Application extens...	29 KB
api-ms-win-crt-runtime-l1-1-0.dll	14-10-2023 17:08	Application extens...	25 KB
api-ms-win-crt-stdio-l1-1-0.dll	14-10-2023 17:08	Application extens...	25 KB
api-ms-win-crt-string-l1-1-0.dll	14-10-2023 17:08	Application extens...	25 KB
concr140e.dll	14-10-2023 17:08	Application extens...	735 KB
DHL-INVOICE.exe	14-10-2023 17:08	Application	23 KB
jli.dll	14-10-2023 17:08	Application extens...	2,482 KB
VCRUNTIME140.dll	14-10-2023 17:08	Application extens...	84 KB

Figure 3 Contents of the malicious DHL-INVOICE ZIP archive.

Once the user clicks on DHL-INVOICE.exe, the executable starts by loading **jli.dll**, which is clearly visible both in the extracted file set (Figure 2) and in the binary's import table (Figure 3).

Inspection of the import table using CFF Explorer confirms that **jli.dll** is explicitly listed as a dependency (see **Figure 4**). The executable does not specify an absolute path while loading this DLL, causing Windows to load the copy present in the same directory as the executable.



CFF Explorer VIII - [DHL-INVOICE.exe]

File Settings ?

DHL-INVOICE.exe

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
jli.dll	6	00002AC0	00000000	00000000	00002B62	00002190
KERNEL32.dll	14	00002930	00000000	00000000	00002CB2	00002000
VCRUNTIME140.dll	2	000029A8	00000000	00000000	00002CE2	00002078
api-ms-win-crt-stdio-l...	4	00002A98	00000000	00000000	00002EDC	00002168
api-ms-win-crt-runtim...	18	00002A00	00000000	00000000	00002EFC	000020D0
api-ms-win-crt-enviro...	1	000029C0	00000000	00000000	00002F1E	00002090
api-ms-win-crt-math-l...	1	000029F0	00000000	00000000	00002F44	000020C0
api-ms-win-crt-locale-...	1	000029E0	00000000	00000000	00002F64	000020B0
api-ms-win-crt-heap-l...	1	000029D0	00000000	00000000	00002F86	000020A0

File: DHL-INVOICE.exe

- Dos Header
- NT Headers
 - File Header
 - Optional Header
 - Data Directories [x]
 - Section Headers [x]
- Export Directory
- Import Directory
- Resource Directory
- Exception Directory
- Relocation Directory
- Debug Directory
- Address Converter
- Dependency Walker
- Hex Editor
- Identifier
- Import Adder
- Quick Disassembler
- Rebuilder
- Resource Editor

Figure 4 Import table of DHL-INVOICE.exe highlighting jli.dll.

This behavior demonstrates **DLL sideloading**. The executable itself is a signed Java launcher and does not contain any malicious code. Instead, the malicious activity is hidden inside the attacker-provided **jli.dll**, which is placed in the same directory. When the executable runs, it loads this DLL and hands over execution to it, abusing the trust of a signed and legitimate-looking application. Let's see detailed analysis in the next sections.

Stage 1: Analysis of DHL-INVOICE.exe

The original file name of DHL-INVOICE.exe is jdeps.exe, which is a legitimate Java tool that normally comes with the JDK. The execution starts from the `main( )` function. From `main()`, the program calls several functions named `JLI_InitArgProcessing`, `JLI_CmdToArgs`, `JLI_GetStdArgc`, `JLI_MemAlloc`, `JLI_GetStdArgs`, and `JLI_Launch`. These function names match exports normally found in `jli.dll`. However, in this sample, all of these functions execute the same malicious code. To avoid repetition, I highlighted and analyzed only one of these functions, **JLI_InitArgProcessing** as shown **Figure 5**. This shows how the malware executes its malicious code through `jli.dll` using DLL sideloading technique. In the next section, I have analyzed `jli.dll` in detail.

```

int64_t main()
{
    int64_t rdx;
    (uint8_t)rdx = 1;
    int64_t rcx;
    (uint8_t)rcx = 1;
    JLI_InitArgProcessing(rcx, rdx);
    char* rax;
    int64_t rdx_1;
    rax = getenv("_JAVA_LAUNCHER_DEBUG");

    if (rax)
    {
        MW_DebugPrintf("Windows original main args:\n", rdx_1);
        int32_t i = 0;

        if (*(uint32_t*)__p___argc() > 0)
        {
            int64_t rdi_1 = 0;
            int32_t* rax_3;

            do
            {
                *(uint64_t*)(*(uint64_t*)__p___argv() + rdi_1);
                MW_DebugPrintf("wwwd_args[%d] = %s\n", (uint64_t)i);
                i += 1;
                rdi_1 += 8;
                rax_3 = __p___argc();
            } while (i < *(uint32_t*)rax_3);
        }
    }
}

```

Figure 5 main() invoking a JLI_InitArgProcessing routine.

Stage 2: Analysis of jli.dll (XLoader)

The function **JLI_InitArgProcessing** is exported from jli.dll, which has been identified as **XLoader**, and it does not perform any legitimate Java-related functionality. Instead, this XLoader function initializes a local structure, sets up thread-local storage (TLS), and directly calls **MW_MainDispatcher**, which contains the core **XLoader** malicious logic (see **Figure 6**).

After execution, the function stores the TLS value and returns. This behavior confirms that **JLI_InitArgProcessing** serves only as a **wrapper function**, whose sole purpose is to transfer execution to the core XLoader dispatcher routine.

```

void* JLI_InitArgProcessing()
{
    void var_10;
    MW_TLSInitializeContext(&var_10);
    MW_MainDispatcher();
    return MW_TLSStoreValue(&var_10);
}

```

Figure 6 JLI_InitArgProcessing function invoking the XLoader core dispatcher routine.

The **XLoader MW_MainDispatcher** function controls the entire execution flow using a state-based loop rather than a linear execution model. It relies on constant (magic) values combined with XOR and modulo operations (see **Figure 7**), selects execution paths through switch statements, and continuously mutates its internal state to determine the next execution step. This design is characteristic of **XLoader's control-flow obfuscation**.

```

MW_MainDispatcher()
{
    (uint32_t)var_28 = 8;

    while (true)
    {
        int32_t rcx_3 = 0x282cd3e1;

        while (true)
        {
            label_180007fe3:
            int32_t rcx_4 = rcx_3 ^ 0x24e003c5;
            int32_t rbx_1;

            switch (rcx_4 % 6)
            {
                case 0:
                {
                    break;
                    break;
                }

                case 1:
                {
                    label_180008078:
                    rbx_1 = rbp[-4];
                    goto label_180008082;
                }

                case 3:
                {
                    label_180008336:

```

Figure 7 Control-flow logic inside XLoader MW_MainDispatcher.

Configuration Parsing

Inside the obfuscated XLoader control flow, the dispatcher contains logic that handles configuration retrieval. One of the dispatcher cases calls **MW_GetConfigurationData** to fetch an embedded configuration blob.

If no configuration data is found, the pointers and size values are set to zero. If the data exists, XLoader extracts a pointer and size from the structure and passes it to

MW_ParseAndCacheConfiguration for processing. After this step, execution moves to the next state in the dispatcher (see **Figure 8**).

Join Medium for free to get updates from this writer.

This configuration contains the PE payload to be loaded by XLoader. Once parsed, the configuration is cached in memory and reused throughout execution. Although the parser itself is heavily obfuscated, its role is clear that it provides the runtime parameters that control how the XLoader operates.

```
case 0x180008476:
{
    void* rax_30 = MW_GetConfigurationData();
    int32_t rax_31;
    void* rcx_49;

    if (!rax_30)
    {
        rcx_49 = nullptr;
        rax_31 = 0;
    }
    else
    {
        rcx_49 = (char*)rax_30 + 0x10;
        rax_31 = *(uint32_t*)((char*)rax_30 + 8);
    }

    rbp[-0xa] = rcx_49;
    rbp[-9] = rax_31;
    rbp[-0xb] = MW_ParseAndCacheConfiguration(&rbp[-0xa]);
    rbx_1 = 4;
}
```

Figure 8 XLoader Configuration retrieval inside the dispatcher.

Image Loading Decision Point

Another critical case in the dispatcher initiates the actual payload loading. This is where the XLoader transitions from configuration to action. The **MW_LoadImageFromFilePath** function is called with a path identifier (PE extracted from the configuration) and a set of loader flags (see

Figure 9). This function triggers the complete PE loading pipeline.

```
case 0x180008188:
{
    void* loaderFlags = sub_18001a0e0();
    MW_LoadImageFromFilePath(rbp[-0xc], loaderFlags);
    rdx_1 = 0xa;
    goto label_180008082;
}
```

Figure 9 Payload loading triggered inside MW_MainDispatcher.

PE File Validation

Before attempting to load the PE file, the XLoader performs validation. This is implemented in **MW_ValidatePEBinaryFormat**, an obfuscated function that checks whether the provided data represents a valid PE file.

DOS Signature Validation

The validation begins with the DOS signature check. Every PE file starts with the DOS header, which begins with the two-byte signature 0x4D5A (magic number). XLoader performs checks for valid DOS signature header of 0x4D5A i.e ('MZ') as shown in **Figure 10**.

```

if (*(uint8_t*)((char*)arg1 + 0x10) != 0x4d)
|   return 4;

    rax_6 = 0x1f;
    goto label_180008b90;
}
case 0x1a:
{
    if (*(uint32_t*)((char*)arg1 + 8) < r14 + 2) {...}

    if (data_1802882d8) {...}

    rax_6 = 0x15;

    if (*(uint32_t*)(*(uint64_t*)(data_1802555e0 + 8)
        + 0x174)) {...}

    rax_6 = 3;
    goto label_180008b90;
}
case 0x1e: {...}
case 0x1f:
{
    if (*(uint32_t*)((char*)arg1 + 8) <= 1)
    |   goto label_1800090ad;

    if (*(uint8_t*)((char*)arg1 + 0x11) != 0x5a)
    {
    |   rax_6 = 0x11;
    |   goto label_180008b90;
    }
}

```

Figure 10 Magic Number Validation check before loading the payload.

PE Signature Validation

After confirming the DOS signature, XLoader reads the **e_lfanew** field from the DOS header. This field, located at offset 0x3C which points to the PE header as shown in **Figure 11**.

```

// Validate e_lfanew field
// exists
if (*(uint32_t*)((char*)arg1 + 8) <= 0x3c)
|   goto label_1800090ad;

if (*(uint32_t*)((char*)arg1 + 8) <= 0x3d)
|   goto label_1800090ad;

if (*(uint32_t*)((char*)arg1 + 8) <= 0x3e)
|   goto label_1800090ad;

if (*(uint32_t*)((char*)arg1 + 8) <= 0x3f)
|   goto label_1800090ad;

// Read e_lfanew (PE header
// offset)
r13 = (uint32_t)*(uint8_t*)((char*)arg1 + 0x4c)
|   (uint32_t)*(uint8_t*)((char*)arg1 + 0x4d)
<< 8 | (uint32_t)*(uint8_t*)((char*)arg1 + 0x4e)
<< 0x10
|   (uint32_t)*(uint8_t*)((char*)arg1 + 0x4f)
<< 0x18;

```

Figure 11 Validation of the e_lfanew PE header field.

Architecture Detection

The validation function also determines whether the PE is 32-bit or 64-bit by examining the Optional Header Magic field as shown in **Figure 12**. This architecture detection is crucial because 32-bit and 64-bit PE files have different structure layouts, particularly in the Optional Header.

```

// Reading Optional Header
// Magic (indicates 32-bit or
// 64-bit)
(uint8_t)rbp = ((uint32_t)
    *(uint8_t*)((char*)arg1 + (uint64_t)r14 + 0x10) | (
    uint32_t)(uint16_t)((uint32_t)
    *(uint8_t*)((char*)arg1 + rcx_15 + 0x10) << 8))
    == 0x20b;
// 0x10B = PE32 (32-bit)
// 0x20B = PE32+ (64-bit)
rbp = (uint32_t)(uint8_t)rbp;
r12 = 0xe;
continue;

```

Figure 12 Parsing the Optional Header magic to determine 32-bit or 64-bit PE.

The Loading Pipeline: From File to Memory

After validation, the XLoader starts its manual PE loading process. It resolves the payload path from configuration data and normalizes it using **MW_ConvertPathWithFlags**. An image loader context is then allocated from thread-local storage and initialized via **MW_InitializeImageLoader** (see **Figure 13**). Finally, **MW_LoadPEImageIntoMemory** maps the PE file into memory without using standard Windows loader APIs.

```
int64_t MW_LoadImageFromFilePath(int64_t* pathIdentifier, int64_t loaderFlags)
{
    int64_t var_18 = 0;
    int64_t* rax = MW_LookupAndGetImagePath(pathIdentifier);
    int64_t var_20 = 0;
    MW_ConvertPathWithFlags(&var_20, rax, 3);
    var_18 = var_20;
    void** imageLoaderCtx = MW_TLSAllocateObject(&data_18028c3f8);
    MW_InitializeImageLoader(imageLoaderCtx, MW_GetStringDataPointer(&var_18),
        loaderFlags);
    MW_LoadPEImageIntoMemory(imageLoaderCtx);
    return 1;
}
```

Figure 13 Manual PE loading routine implemented by MW_LoadImageFromFilePath.

Memory Allocation and Section Copying

MW_LoadPEImageIntoMemory is the main function responsible for loading the payload. It handles the steps needed to turn a PE file on disk into a runnable image in memory. Below is the first important step in this process (see **Figure 14**).

```
int64_t MW_LoadPEImageIntoMemory(void* imageLoaderCtx)
{
    int128_t var_50;
    __builtin_memset(&var_50, 0, 0x18);
    MW_AllocateImageMemoryAndCopySections(imageLoaderCtx);
    char* var_40;
    MW_AllocateImageBuffers(*(uint64_t*)((char*)imageLoaderCtx + 0x18),
        (uint128_t*)((char*)&var_50 + 8), &var_40);
    int64_t rsi = *(uint64_t*)((char*)&var_50 + 8);
}
```

Figure 14 Manual memory allocation and section copying during PE loading.

The **MW_AllocateImageMemoryAndCopySections** function is where things get really interesting and suspicious. This is where I see the first major red flag, the allocation of memory with **PAGE_EXECUTE_READWRITE** permissions.

XLoader allocates a memory region large enough to hold the entire image. The allocation is done using **MEM_COMMIT | MEM_RESERVE (0x3000)** with **PAGE_EXECUTE_READWRITE (0x40)** permissions. This creates a memory region that can be written to and executed. It uses indirect calls instead of standard APIs as shown in **Figure 15**.

```
int64_t MW_AllocateImageMemoryAndCopySections(void* imageLoaderCtx)
{
    void* rcx = *(uint64_t*)((char*)imageLoaderCtx + 8);
    *(uint32_t*)rcx;
    int32_t rax = MW_GetImageSize(rcx);

    if (data_1802882e0)
        MW_GetDebugLock();

    if (data_1802882e0)
        MW_GetGlobalDataWithLock();

    void* rax_1 = *(uint64_t*)(data_1802555e8 + 0x40);
    *(uint64_t*)((char*)imageLoaderCtx + 0x28) = (*(uint64_t*)((char*)rax_1 + 0x20))(
        *(uint64_t*)((char*)rax_1 + 8), 0, (uint64_t)rax, 0x3000, 0x40);
    char* i_3 = *(uint64_t*)((char*)imageLoaderCtx + 0x28);

    if (!i_3)
    {
        void** rax_11 = MW_TLSAllocateObject(&data_18028fa28);
        MW_InitializeExceptionInfo(rax_11, &data_180271a48);
        uint64_t ssp;
        MW_ThrowException(rax_11, ssp);
        /* no return */
    }
}
```

Figure 15 Payload memory allocated with PAGE_EXECUTE_READWRITE permissions.

Copying PE Headers and Sections

This part of the code handles copying the **PE headers and section data** into memory.

First, XLoader checks whether the payload is a 32-bit or 64-bit executable and reads the **SizeOfHeaders** value from the appropriate NT header. This value determines how much header data needs to be copied.

Next, the PE headers including the DOS header, PE signature, COFF header, optional header, and section headers are copied from the file buffer into the allocated memory region.

After the headers are in place, the code reads the total number of sections from the file header and loops through each one. For every section, it retrieves the section header, calculates the correct source and destination addresses, and copies the section data into memory (see **Figure 16**).

This logic closely follows the Windows loader's process for mapping a PE file, but it is implemented manually to bypass standard loading APIs and evade API-based hooking by EDR solutions.

```
if (*(uint8_t*)((char*)rcx_5 + 0x30))
    // SizeOfHeaders
    i_5 = *(uint32_t*)(MW_GetNTHeaders64(rcx_5) + 0x54);
else
    i_5 = *(uint32_t*)(MW_GetNTHeaders32(rcx_5) + 0x54);

// Copy PE headers (DOS header, PE signature, COFF header,
// Optional header, Section headers)
char* rcx_6 = *(uint64_t*)((char*)imageLoaderCtx + 0x20);
char* rax_5 = *(uint64_t*)((char*)imageLoaderCtx + 0x28);
int32_t rdx_1 = 0;
uint64_t i_1 = (uint64_t)i_5;

if (i_1 > 0)
{
    do
    {
        int64_t r10_1 = (int64_t)rdx_1;
        rax_5[r10_1] = rcx_6[r10_1];
        rdx_1 += 1;
    } while ((int64_t)rdx_1 < i_1);
}

int32_t sectionIndex = 0;
void* rcx_7 = *(uint64_t*)((char*)imageLoaderCtx + 8);
*(uint8_t*)rcx_7;

if (*(uint16_t*)(MW_GetFileHeaderPtr rcx_7) + 2) > 0)
{
    uint32_t rcx_10;

    do
    {
        void* rcx_8 = *(uint64_t*)((char*)imageLoaderCtx + 8);
        *(uint32_t*)rcx_8;
        void* rax_7 = MW_GetSectionHeaderByIndex rcx_8, sectionIndex);
        int32_t i_4 = *(uint32_t*)((char*)rax_7 + 0x10);
```

Figure 16 Manual copying of PE headers and sections during payload loading.

Creating the PE Parser

After copying all the data, XLoader creates a PE parser object. This PE parser object gives XLoader an easy way to access different parts of the PE file, such as headers, sections, imports, and exports, without recalculating offsets each time (see **Figure 17**). The use of **MW_StorePointerWithGCTracking** indicates that XLoader has implemented some form of garbage collection or reference counting for its internal objects.

```
void** rax_10 = MW_TLSAllocateObject(&data_18028c448);
int64_t result =
    MW_InitializePEParser(rax_10, *(uint64_t*)((char*)imageLoaderCtx + 0x28));
    MW_StorePointerWithGCTracking((char*)imageLoaderCtx + 0x10, rax_10);
return result;
```

Figure 17 Initializing the PE parser and storing it with GC tracking.

Process Hollowing

After loading the payload into memory, XLoader injects it into a legitimate .NET related processes using process hollowing. It creates **AddInProcess32.exe** in a suspended state using **CreateProcessW** API (see **Figure 18**). It then queries the process information to locate the Process Environment Block and unmaps the original executable image from memory using **ZwUnmapViewOfSection**, leaving the process hollow while keeping all loaded DLLs intact. Next, the loader allocates memory inside the target process with **VirtualAllocEx** using **PAGE_EXECUTE_READWRITE** permissions and writes the payload's PE headers and sections into this memory using **WriteProcessMemory**. If required, relocations are applied to ensure the payload runs correctly at the new base address. Because AddInProcess32.exe is a 32-bit process, the XLoader uses WOW64 specific APIs to read and modify the thread context, redirecting execution to the payload's entry point. Finally, the suspended thread is resumed, causing the payload to execute under the identity of a trusted Microsoft process, making the malicious activity blend in with normal system behavior and harder to detect.

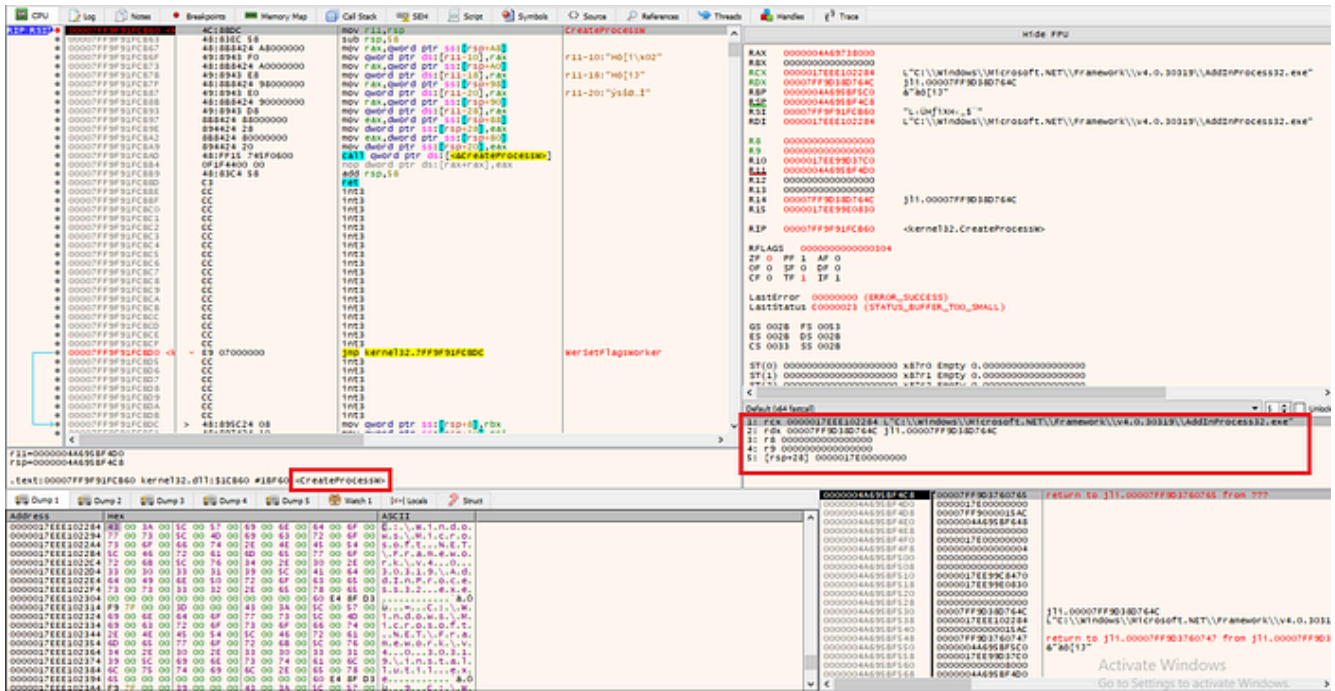


Figure 18 CreateProcessW called by the loader to spawn AddInProcess32.exe prior to process hollowing.

By placing a breakpoint on **WriteProcessMemory**, I have identified the payload injection stage and successfully dumped the final payload from the memory of AddInProcess32.exe to disk for further analysis.

By performing static analysis on dumped file, I confirmed that the final payload is a .NET based Phantom Stealer.

Stage 3: Phantom Stealer v3.5.0

After dumping and decompiling the payload, the class structure confirms that it is a **.NET-based Phantom Stealer v3.5.0** with a modular design (see **Figure 19**). The code includes modules for browser and email credential theft, cryptocurrency wallets, keylogging, screenshots, clipboard monitoring, and file grabbing. Multiple exfiltration channels such as FTP, SMTP, Discord, and Telegram are also supported. Additional components related to anti-analysis, native API interaction, and persistence indicate a mature and well-developed malware.

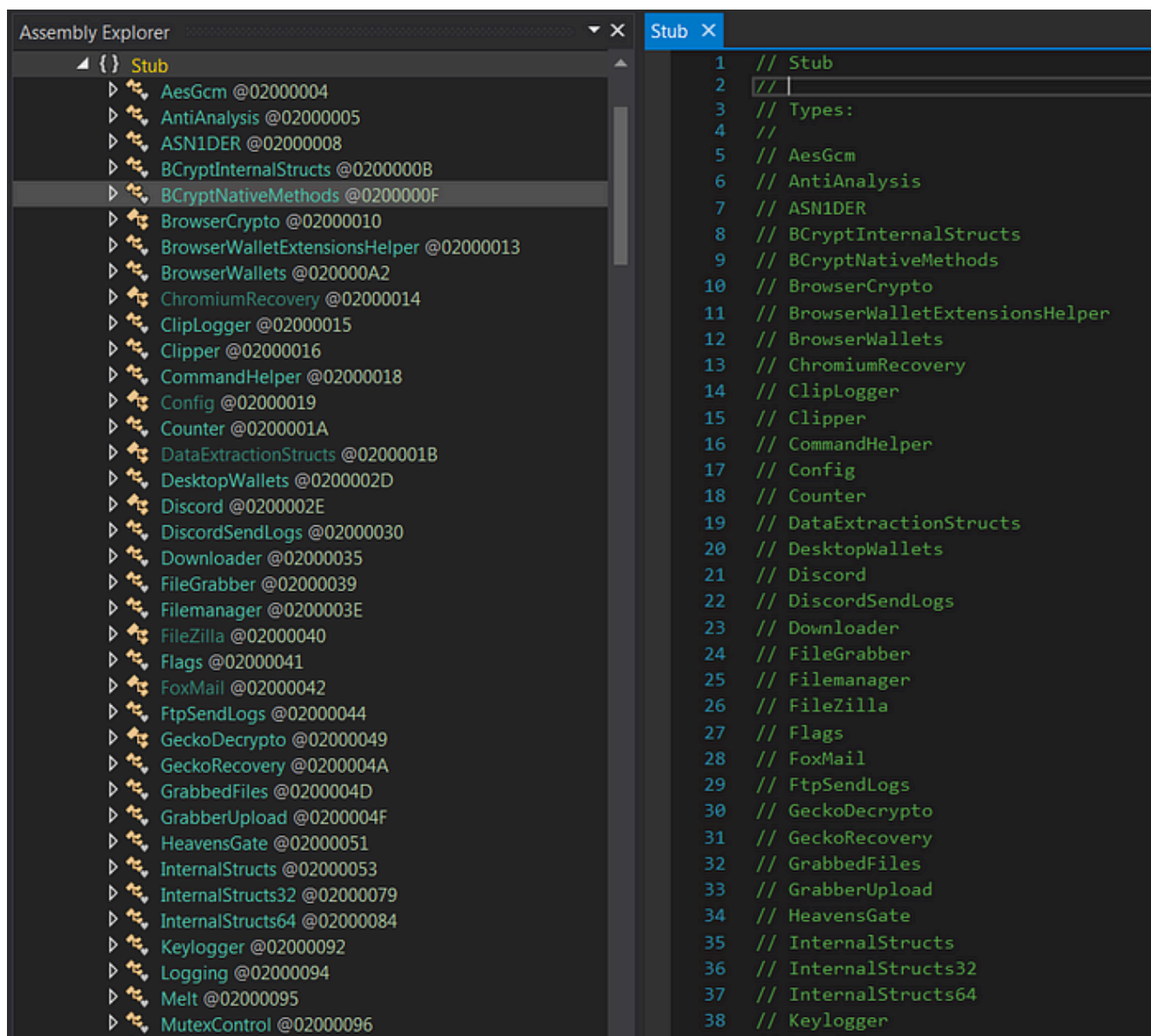


Figure 19 Decompiled .NET payload showing modular stealer components.

Since the technical analysis of Phantom Stealer has been widely documented, I just took an interest in the config part of phantom stealer found during this analysis. The **Config class** contains the **core configuration and control logic** for the payload. All sensitive values such as Telegram API keys, Discord webhooks, SMTP credentials, FTP details, and grabber settings are stored in **encrypted form** and decrypted at runtime using *StringsCrypt.DecryptConfig*. **Figure 20** contains the config part of phantom stealer which is encrypted.

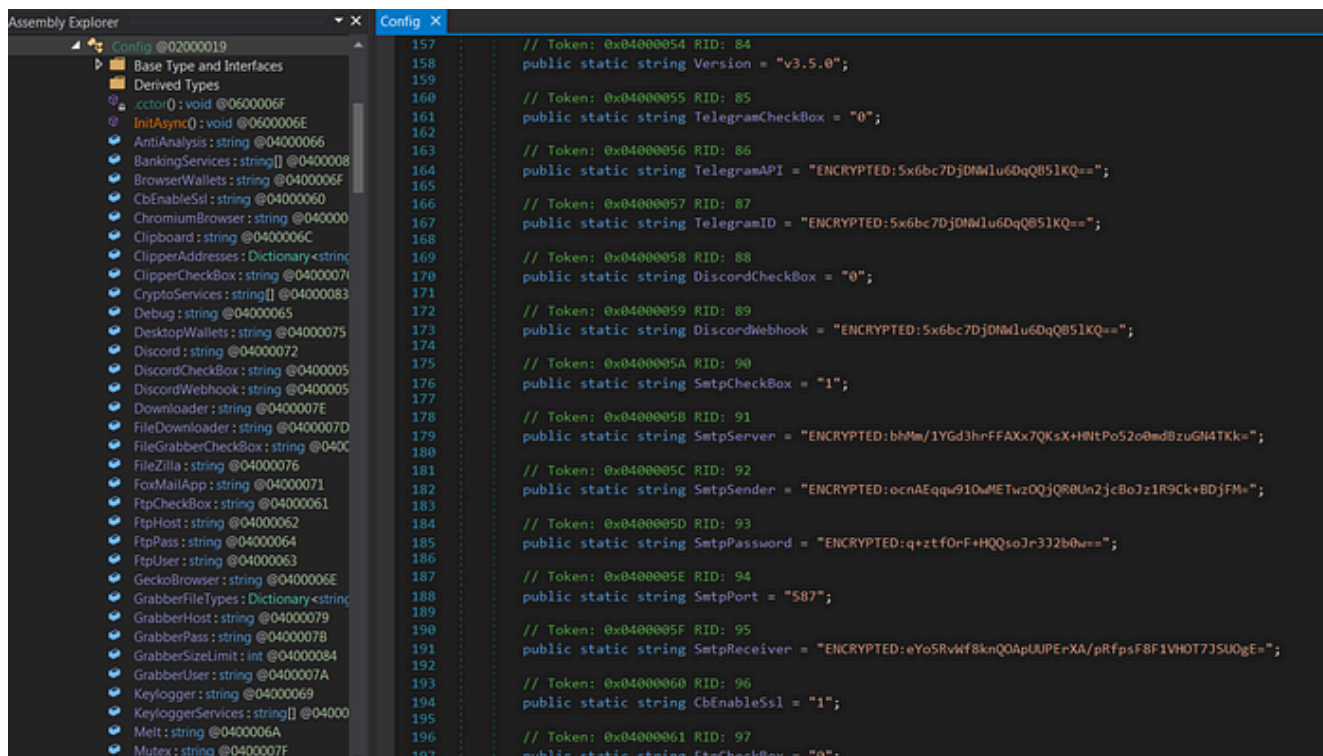


Figure 20 Encrypted configuration values and feature flags in the payload's Config class.

Decrypting the Encrypted Configuration

To decrypt the encrypted configuration values, I analyzed the *StringsCrypt* class used by the payload. The configuration entry starting with the prefix **ENCRYPTED:** is first **Base64-decoded** and then decrypted using **AES-256 in CBC** mode. The AES key and IV are derived using **PBKDF2 (Rfc2898DeriveBytes)** with a hardcoded encryption key (**CryptKey**), a hardcoded salt (**SaltBytes**), and 1000 iterations.

I reproduced the payload's configuration decryption routine by porting the AES-256-CBC and PBKDF2 logic to Python, allowing me to decrypt all embedded configuration values offline and confirm the active exfiltration channels as shown in Table 1.

SmtptServer: mail.polywin.co.inSmtptSender: it@polywin.co.inSmtptPassword: p@wordUsdSmtptReceiver: office1one@infroyale.com

Conclusion

This case shows how a simple “DHL invoice” email can lead to a full malware infection. The ZIP drops a **legit signed Java tool** renamed as **DHL-INVOICE.exe**, but it loads a **malicious jli.dll (XLoader)** using DLL sideloading.

After that, XLoader **loads the next payload directly in memory** and then **hollows AddInProcess32.exe** so the final malware runs inside a trusted Windows process. The final payload was confirmed as **Phantom Stealer v3.5.0**.

The main takeaway: these attacks don't rely on one trick. They mix **trusted files + stealth loading + process hollowing**, so defenders should watch for **DLL sideloading, suspicious memory allocation, and hollowing behavior**, not just file signatures.

IOCs:

4dd0f2124b49724aff35d279e68f4102 DHL-INVOICE.msg52d4f47f57cd8bae509405b8b9158c37 DHL-INVOICE.zip8de05cd12b96755cde2e259fee0c6efc jli.dll561313b3caf1ee0697055742fd267b86 DHL-INVOICE.exef2526cd1839ebc2d32e9e2a9c0fe9f5e stub.exe (Phantom Stealer)