DodgeBox | ThreatLabz

By Yin Hong Chang, Sudeep Singh

Published: 2024-07-10 · Archived: 2026-04-05 14:40:16 UTC

Technical Analysis

Attack chain

APT41 employs DLL sideloading as a means of executing DodgeBox. They utilize a legitimate executable (taskhost.exe), signed by Sandboxie, to sideload a malicious DLL (sbiedll.dll). This malicious DLL, DodgeBox, serves as a loader and is responsible for decrypting a second stage payload from an encrypted DAT file (sbiedll.dat). The decrypted payload, MoonWalk functions as a backdoor that abuses Google Drive for command-and-control (C2) communication. The figure below illustrates the attack chain at a high level.

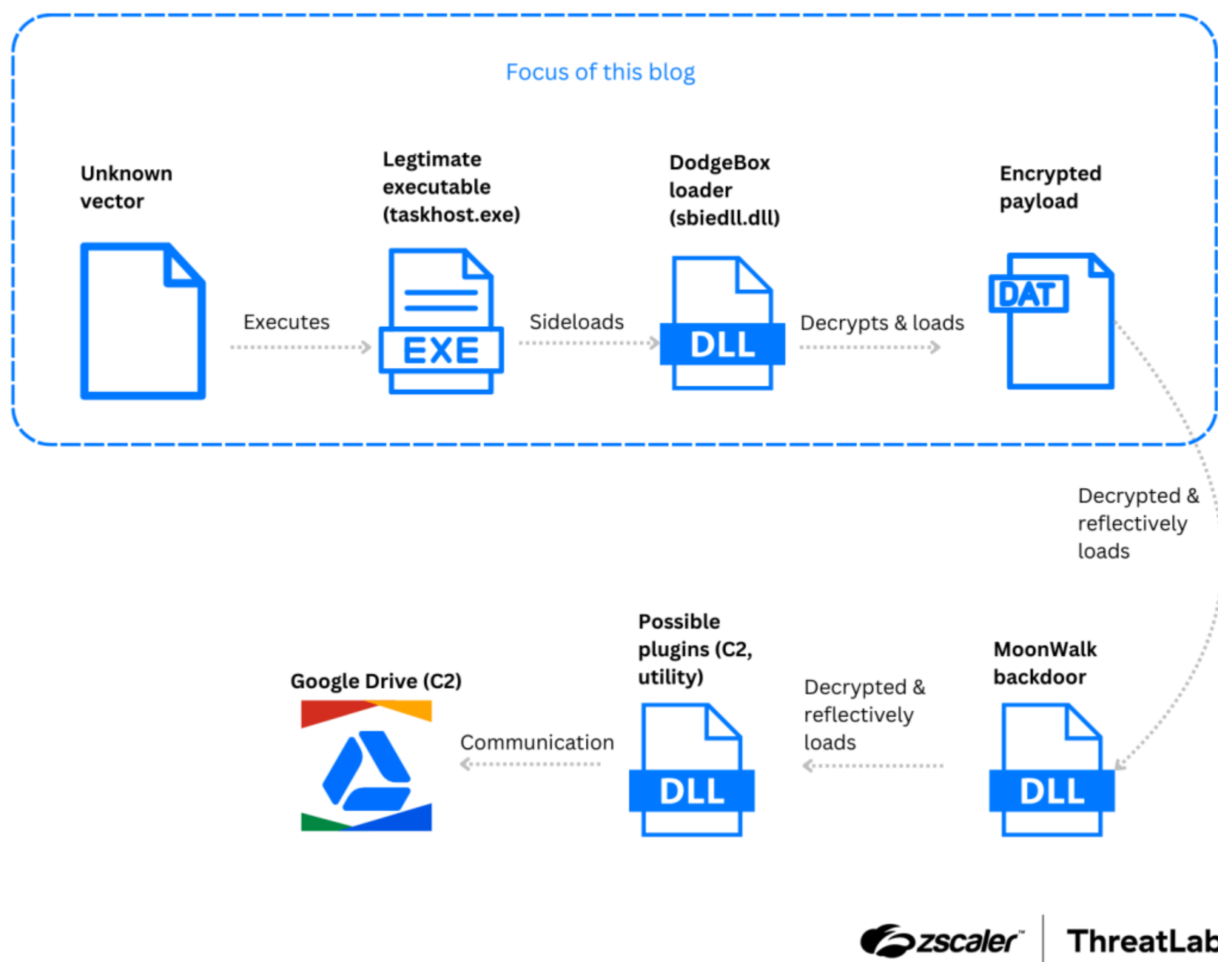

Figure 1: Attack chain used to deploy the DodgeBox loader and MoonWalk backdoor.

DodgeBox analysis

DodgeBox, a reflective DLL loader written in C, showcases similarities to StealthVector in terms of concept but incorporates significant improvements in its implementation. It offers various capabilities, including decrypting and loading embedded DLLs, conducting environment checks and bindings, and executing cleanup procedures. What sets DodgeBox apart from other malware is its unique algorithms and techniques.

During our threat hunting activities, we came across two DodgeBox samples that were designed to be sideloaded by signed legitimate executables. One of these executables was developed by Sandboxie (`SandboxieWUAU.exe`), while the other was developed by AhnLab. All exports within the DLL point to a single function that primarily invokes the main function of the malware, as illustrated below:

```
 void SbieDll_Hook()
{
 if ( dwExportCalled )
 {
   Sleep(0xFFFFFFFF);
 }
 else
 {
   hSbieDll_ = hSbieDll;
   dwExportCalled = 1;
   MalwareMain();
 }
}
```

`MalwareMain` implements the main functionality of DodgeBox, and can be broken down into three main phases:

1. Decryption of DodgeBox's configuration

DodgeBox employs AES Cipher Feedback (AES-CFB) mode for encrypting its configuration. AES-CFB transforms AES from a block cipher into a stream cipher, allowing for the encryption of data with different lengths without requiring padding. The encrypted configuration is embedded within the `.data` section of the binary. To ensure the integrity of the configuration, DodgeBox utilizes hard-coded MD5 hashes to validate both the embedded AES keys and the encrypted configuration. For reference, a sample of DodgeBox's decrypted configuration can be found in the Appendix section of this blog. We will reference this sample configuration using the variable `Config` in the following sections.

2. Execution guardrails and environment setup

After decrypting its configuration, DodgeBox performs several environment checks to ensure it is running on its intended target.

Execution guardrail: Argument check

DodgeBox starts by verifying that the process was launched with the correct arguments. It scans the `argv` parameter for a specific string defined in `Config.szArgFlag`. Next, it calculates the MD5 hash of the subsequent

argument and compares it to the hash specified in `Config.rgbArgFlagValueMD5`. In this case, DodgeBox expects the arguments to include `--type driver`. If this verification check fails, the process is terminated.

Environment setup: API Resolution

Afterwards, DodgeBox proceeds to resolve multiple APIs that are utilized for additional environment checks and setup. Notably, DodgeBox employs a salted FNV1a hash for DLL and function names. This salted hash mechanism aids DodgeBox in evading static detections that typically search for hashes of DLL or function names. Additionally, it enables different samples of DodgeBox to use distinct values for the same DLL and function, all while preserving the integrity of the core hashing algorithm.

The following code shows DodgeBox calling its `ResolveImport` function to resolve the address of `LdrLoadDll`, and populating its import table.

```
 // ResolveImport takes in (wszDllName, dwDllNameHash, dwFuncNameHash)
sImportTable->ntdll_LdrLoadDll = ResolveImport(L"ntdll", 0xFE0B07B0, 0xCA7BB6AC);
```

Inside the `ResolveImport` function, DodgeBox utilizes the FNV1a hashing function in a two-step process. First, it hashes the input string, which represents a DLL or function name. Then, it hashes a salt value separately. This two-step hashing procedure is equivalent to hashing the concatenation of the input string and salt. The following pseudo-code represents the implementation of the salted hash:

```
 dwHash = 0x811C9DC5;     // Standard initial seed for FNV1a
pwszInputString_Char = pwszInputString;
cchInputString = -1LL;
do
 ++cchInputString;
while ( pwszInputString[cchInputString] );
pwszInputStringEnd = (pwszInputString + 2 * cchInputString);
if ( pwszInputString
```

A Python script to generate the salted hashes is included in the Appendix.

In addition to the salted hash implementation, DodgeBox incorporates another noteworthy feature in its `ResolveImport` function. This function accepts both the DLL name as a string and its hash value as arguments. This redundancy appears to be designed to provide flexibility, allowing DodgeBox to handle scenarios where the target DLL has not yet been loaded. In such cases, DodgeBox invokes the `LoadLibraryW` function with the provided string to load the DLL dynamically.

Furthermore, DodgeBox effectively handles forwarded exports and exports by ordinals. It utilizes `ntdll!LdrLoadDll` and `ntdll!LdrGetProcedureAddressEx` when necessary to resolve the address of the exported function. This approach ensures that DodgeBox can successfully resolve and utilize the desired functions, regardless of the export method used.

Environment setup: DLL unhooking

Once DodgeBox has resolved the necessary functions, it proceeds to scan and unhook DLLs that are loaded from the System32 directory. This process involves iterating through the `.pdata` section of each DLL, retrieving each function's start and end addresses, and calculating an FNV1a hash for the bytes of each function. DodgeBox then computes a corresponding hash for the same function's bytes as stored on disk. If the two hashes differ, potential tampering can be detected, and DodgeBox will replace the in-memory function with the original version from the disk.

For each DLL that has been successfully scanned, DodgeBox marks the corresponding `LDR_DATA_TABLE_ENTRY` by clearing the `ReservedFlags6` field and setting the upper bit to 1. This marking allows DodgeBox to avoid scanning the same DLL twice.

Environment setup: Disabling CFG

Following that, DodgeBox checks if the operating system is Windows 8 or newer. If so, the code verifies whether Control Flow Guard (CFG) is enabled by calling `GetProcessMitigationPolicy` with the `ProcessControlFlowGuardPolicy` parameter. If CFG is active, the malware attempts to disable it.

To achieve this, DodgeBox locates the `LdrpHandleInvalidUserCallTarget` function within `ntdll.dll` by searching for a specific byte sequence. Once found, the malware patches this function with a simple `jmp` `rax` instruction:

```
 ntdll!LdrpHandleInvalidUserCallTarget:
00007ffe`fc8cf070 48ffe0          jmp     rax
00007ffe`fc8cf073 cc              int     3
00007ffe`fc8cf074 90              nop
```

CFG verifies the validity of indirect call targets. When a CFG check fails, `LdrpHandleInvalidUserCallTarget` is invoked, typically raising an interrupt. At this point, the rax register contains the invalid target address. The patch modifies this behavior, calling the target directly instead of raising an interrupt, thus bypassing CFG protection.

In addition, DodgeBox replaces `msvcrt!_guard_check_icall_fptr` with `msvcrt!_DebugMallocator::~_DebugMallocator`, a function that returns 0 to disable the CFG check performed by `msvcrt`.

Execution guardrail: MAC, computer name, and user name checks

Finally, DodgeBox performs a series of checks to verify if it is configured to run on the current machine. The malware compares the machine's MAC address against `Config.rgbTargetMac`, and compares the computer name against `Config.wszTargetComputerName`. Depending on the `Config.fDoCheckIsSystem` flag, DodgeBox checks whether it is running with `SYSTEM` privileges. If any of these checks fail, the malware terminates execution.

3. Payload decryption and environment keying

Payload decryption

In the final phase, DodgeBox commences the decryption process for the MoonWalk payload DAT file. The code starts by inspecting the first four bytes of the file. If these bytes are non-zero, it signifies that the DAT file has been tied to a particular machine, (which is described below). However, if the DAT file is not machine-specific, DodgeBox proceeds to decrypt the file using AES-CFB encryption, utilizing the key parameters stored in the configuration file. In the samples analyzed by ThreatLabz, this decrypted DAT file corresponds to a DLL, which is the MoonWalk backdoor.

Environment keying of the payload

After the decryption process, DodgeBox takes the additional step of keying the payload to the current machine. It accomplishes this by re-encrypting the payload using the Config.rgbAESKeyForDatFile key. However, in this specific scenario, the process deviates from the configuration file's IV (Initialization Vector). Instead, it utilizes the MD5 hash of the current machine's GUID as the AES IV. This approach guarantees that the decrypted DAT file cannot be decrypted on any other machine, thus enhancing the payload's security.

Loading the payload using DLL hollowing

Next, DodgeBox reflectively loads the payload using a DLL hollowing technique. At a high level, the process begins with the random selection of a host DLL from the `System32` directory, ensuring it is not on a blocklist (DLL blocklist available in the Appendix section) and has a sufficiently large .text section. A copy of this DLL is then created at `C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System.Data.Trace\v4.0_4.0.0.0__\.dll`. DodgeBox modifies this copy by disabling the NX flag, removing the `reloc` and `TLS` sections, and patching its entry point with a simple `return 1`.

Following the preparation of the host DLL for injection, DodgeBox proceeds by zeroing the PE headers, and the `IMAGE_DATA_DIRECTORY` structures corresponding to the `import`, `reloc`, and `debug` directories of the payload DLL. This modified payload DLL is then inserted into the previously selected host DLL. The resulting copy of the modified host DLL is loaded into memory using the `NtCreateSection` and `NtMapViewOfSection` APIs.

Once the DLL is successfully loaded, DodgeBox updates the relevant entries in the Process Environment Block (PEB) to reflect the newly loaded DLL. To further conceal its activities, DodgeBox overwrites the modified copy of the host DLL with its original contents, making it appear as a legitimate, signed DLL on disk. Finally, the malware calls the entrypoint of the payload DLL.

Interestingly, if the function responsible for DLL hollowing fails to load the payload DLL, DodgeBox employs a fallback mechanism. This fallback function implements a traditional form of reflective DLL loading using `NtAllocateVirtualMemory` and `NtProtectVirtualMemory`.

At this stage, the payload DLL has been successfully loaded, and control is transferred to the payload DLL by invoking the first exported function.

Call stack spoofing

There is one last technique employed by DodgeBox throughout all three phases discussed above: **call stack spoofing**. Call stack spoofing is employed to obscure the origins of API calls, making it more challenging for

EDRs and antivirus systems to detect malicious activity. By manipulating the call stack, DodgeBox makes API calls appear as if they originate from trusted binaries rather than the malware itself. This prevents security solutions from gaining contextual information about the true source of the API calls.

DodgeBox specifically utilizes call stack spoofing when invoking Windows APIs that are more likely to be monitored. As an example, it directly calls `RtlInitUnicodeString`, a Windows API that only performs string manipulation, instead of using stack spoofing.

```
(sImportTable->ntdll_RtlInitUnicodeString)(v25, v26);
```

However, call stack spoofing is used when calling `NtAllocateVirtualMemory,` an API known to be abused by malware, as shown below:

```
CallFunction(
  sImportTable->ntdll_NtAllocateVirtualMemory,  // API to call
  0,    // Unused
  6LL,  // Number of parameters
  // Parameters to the API
  -1LL, &pAllocBase, 0LL, &dwSizeOfImage, 0x3000, PAGE_READWRITE)
```

The technique mentioned above can be observed in the figures below. In the first figure, we can see a typical call stack when explorer.exe invokes the CreateFileW function. The system monitoring tool, SysMon, effectively walks the call stack, enabling us to understand the purpose behind this API call and examine the modules and functions involved in the process.

K 0	FLTMGR.SYS	FltpPerformPreCallbacks + 0x2fd	0xfffff80cb44d555d	C:\Windows\System32\drivers\FLTMGR.SYS
K 1	FLTMGR.SYS	FltpPassThroughInternal + 0x8c	0xfffff80cb44d50bc	C:\Windows\System32\drivers\FLTMGR.SYS
K 2	FLTMGR.SYS	FltpCreate + 0x2e5	0xfffff80cb450d545	C:\Windows\System32\drivers\FLTMGR.SYS
K 3	ntoskrnl.exe	IofCallDriver + 0x59	0xfffff8037c36e189	C:\Windows\system32\ntoskrnl.exe
K 4	ntoskrnl.exe	IoCallDriverWithTracing + 0x34	0xfffff8037c3151f4	C:\Windows\system32\ntoskrnl.exe
K 5	ntoskrnl.exe	IopParseDevice + 0x632	0xfffff8037c7e51a2	C:\Windows\system32\ntoskrnl.exe
K 6	ntoskrnl.exe	ObpLookupObjectName + 0x719	0xfffff8037c85c029	C:\Windows\system32\ntoskrnl.exe
K 7	ntoskrnl.exe	ObOpenObjectByNameEx + 0x1df	0xfffff8037c85a62f	C:\Windows\system32\ntoskrnl.exe
K 8	ntoskrnl.exe	IopCreateFile + 0x404	0xfffff8037c7c0874	C:\Windows\system32\ntoskrnl.exe
K 9	ntoskrnl.exe	NtCreateFile + 0x79	0xfffff8037c7c0459	C:\Windows\system32\ntoskrnl.exe
K 10	ntoskrnl.exe	KiSystemServiceCopyEnd + 0x25	0xfffff8037c475085	C:\Windows\system32\ntoskrnl.exe
U 11	ntdll.dll	NtCreateFile + 0x14	0x7ffefc8df034	C:\Windows\SYSTEM32\ntdll.dll
U 12	KERNELBASE.dll	CreateFileInternal + 0x2f6	0x7ffef8a8fb26	C:\Windows\System32\KERNELBASE.dll
U 13	KERNELBASE.dll	CreateFileW + 0x66	0x7ffef8a8f816	C:\Windows\System32\KERNELBASE.dll
U 14	windows.storage.dll	CCachedINIFile::Load + 0x59	0x7ffef941ad49	C:\Windows\System32\windows.storage.dll
U 15	windows.storage.dll	CPrivateProfileCache::_AddNewINIFromFile + 0x67	0x7ffef941ac1b	C:\Windows\System32\windows.storage.dll
U 16	windows.storage.dll	CPrivateProfile::Initialize + 0x3bd	0x7ffef9443c7d	C:\Windows\System32\windows.storage.dll
U 17	windows.storage.dll	SHGetCachedPrivateProfile + 0x6e	0x7ffef94839f6	C:\Windows\System32\windows.storage.dll
U 18	windows.storage.dll	CFSFolder::_GetDesktopIni + 0x73	0x7ffef94838bb	C:\Windows\System32\windows.storage.dll
U 19	windows.storage.dll	CFSFolder::_DiscoverLocalizedName + 0x5a9	0x7ffef943b2a9	C:\Windows\System32\windows.storage.dll
U 20	windows.storage.dll	CFSFolder::_CreateIDList + 0x130	0x7ffef943a5d0	C:\Windows\System32\windows.storage.dll
U 21	windows.storage.dll	CFSFolder::ParseDisplayName + 0x911	0x7ffef9438be1	C:\Windows\System32\windows.storage.dll
U 22	shlwapi.dll	IShellFolder_ParseDisplayName + 0x76	0x7ffef9a97886	C:\Windows\System32\shlwapi.dll
U 23	explorerframe.dll	GetRealIDL + 0x107	0x7ffee11394af	C:\Windows\system32\explorerframe.dll
U 24	explorerframe.dll	SimpleToRealIDListWithContext + 0x9b	0x7ffee1139997	C:\Windows\system32\explorerframe.dll
U 25	explorerframe.dll	CNscChangeNotifyTask::_ConvertIDList + 0x17d	0x7ffee10c7a7d	C:\Windows\system32\explorerframe.dll
U 26	explorerframe.dll	CNscChangeNotifyTask::InternalResumeRT + 0x19	0x7ffee10c71a9	C:\Windows\system32\explorerframe.dll
U 27	explorerframe.dll	CRunnableTask::Run + 0xb2	0x7ffee0ff70c2	C:\Windows\system32\explorerframe.dll
U 28	windows.storage.dll	CShellTask::TT_Run + 0x3c	0x7ffef94ab3ec	C:\Windows\System32\windows.storage.dll
U 29	windows.storage.dll	CShellTaskThread::ThreadProc + 0xdd	0x7ffef94ab0a5	C:\Windows\System32\windows.storage.dll
U 30	windows.storage.dll	CShellTaskThread::s_ThreadProc + 0x35	0x7ffef94aaf85	C:\Windows\System32\windows.storage.dll
U 31	shcore.dll	ExecuteWorkItemThreadProc + 0x16	0x7ffef9d52ac6	C:\Windows\System32\shcore.dll
U 32	ntdll.dll	RtlpTpWorkCallback + 0x165	0x7ffefc89c4d5	C:\Windows\SYSTEM32\ntdll.dll
U 33	ntdll.dll	TppWorkerThread + 0x644	0x7ffefc85bec4	C:\Windows\SYSTEM32\ntdll.dll
U 34	KERNEL32.DLL	BaseThreadInitThunk + 0x14	0x7ffefbe27e94	C:\Windows\System32\KERNEL32.DLL
U 35	ntdll.dll	RtlUserThreadStart + 0x21	0x7ffefc8a7ad1	C:\Windows\SYSTEM32\ntdll.dll

Figure 2: Normal example of stack trace from `explorer.exe` calling `CreateFileW`.

In contrast, the next figure shows the call stack recorded by SysMon when DodgeBox uses stack spoofing to call the CreateFileW function. Notice that there is no indication of DodgeBox's modules that triggered the API call. Instead, the modules involved all appear to be legitimate Windows modules.

Frame	Module	Location	Address	Path
K 0	FLTMGR.SYS	FltpPerformPreCallbacks + 0x2fd	0xfffff80cb44d555d	C:\Windows\System32\drivers\FLTMGR.SYS
K 1	FLTMGR.SYS	FltpPassThroughInternal + 0x8c	0xfffff80cb44d50bc	C:\Windows\System32\drivers\FLTMGR.SYS
K 2	FLTMGR.SYS	FltpCreate + 0x2e5	0xfffff80cb450d545	C:\Windows\System32\drivers\FLTMGR.SYS
K 3	ntoskrnl.exe	IofCallDriver + 0x59	0xfffff8037c36e189	C:\Windows\system32\ntoskrnl.exe
K 4	ntoskrnl.exe	IoCallDriverWithTracing + 0x34	0xfffff8037c3151f4	C:\Windows\system32\ntoskrnl.exe
K 5	ntoskrnl.exe	IopParseDevice + 0x632	0xfffff8037c7e51a2	C:\Windows\system32\ntoskrnl.exe
K 6	ntoskrnl.exe	ObpLookupObjectName + 0x719	0xfffff8037c85c029	C:\Windows\system32\ntoskrnl.exe
K 7	ntoskrnl.exe	ObOpenObjectByNameEx + 0x1df	0xfffff8037c85a62f	C:\Windows\system32\ntoskrnl.exe
K 8	ntoskrnl.exe	IopCreateFile + 0x404	0xfffff8037c7c0874	C:\Windows\system32\ntoskrnl.exe
K 9	ntoskrnl.exe	NtCreateFile + 0x79	0xfffff8037c7c0459	C:\Windows\system32\ntoskrnl.exe
K 10	ntoskrnl.exe	KiSystemServiceCopyEnd + 0x25	0xfffff8037c475085	C:\Windows\system32\ntoskrnl.exe
U 11	ntdll.dll	NtCreateFile + 0x14	0x7ffefc8df034	C:\Windows\System32\ntdll.dll
U 12	KernelBase.dll	ARI::DependencyMiniRepository::LogDMRSectionNotFound + 0x7c	0x7ffef8b3ca3c	C:\Windows\System32\KernelBase.dll
U 13	kernel32.dll	BaseThreadInitThunk + 0x14	0x7ffefbe27e94	C:\Windows\System32\kernel32.dll
U 14	ntdll.dll	RtlUserThreadStart + 0x21	0x7ffefc8a7ad1	C:\Windows\System32\ntdll.dll

Figure 3: Stack trace of DodgeBox calling `CreateFileW` using the stack spoofing technique.

There is an excellent writeup of this technique, so we will only highlight some implementation details specific to DodgeBox:

- When the `CallFunction` is invoked, DodgeBox uses a random `jmp qword ptr [rbp+48h]` gadget residing within the `.text` section of K `ernelBase`.
- DodgeBox analyzes the unwind codes within the `.pdata` section to extract the unwind size for the function that includes the selected gadget.
- DodgeBox obtains the addresses of `RtlUserThreadStart + 0x21` and `BaseThreadInitThunk + 0x14`, along with their respective unwind sizes.
- DodgeBox sets up the stack by inserting the addresses of `RtlUserThreadStart + 0x21`, `BaseThreadInitThunk + 0x14`, and the address of the gadget at the right positions, utilizing the unwind sizes retrieved.
- Following that, DodgeBox proceeds to insert the appropriate return address at `[rbp+48h]` and prepares the registers and stack with the necessary argument values to be passed to the API. This preparation ensures that the API is called correctly and with the intended parameters.
- Finally, DodgeBox executes a `jmp` instruction to redirect the control flow to the targeted API.

Explore more Zscaler blogs

Source: https://www.zscaler.com/blogs/security-research/dodgebox-deep-dive-updated-arsenal-apt41-part-1