DodgeBox: A deep dive into the updated arsenal of APT41 | Part 1

Yin Hong Chang, Sudeep Singh ⋮⋮ 7/10/2024

Security Research

July 10, 2024 - 19 min read

Introduction

This is Part 1 of our two-part technical deep dive into APT41's new tooling, which includes DodgeBox and MoonWalk. For details about MoonWalk, go to Part 2.

In April 2024, Zscaler ThreatLabz uncovered a previously unknown loader called DodgeBox. Upon further analysis, striking similarities were found between DodgeBox and variants of StealthVector, a tool associated with the China-based advanced persistent threat (APT) actor APT41 / Earth Baku. DodgeBox is a loader that proceeds to load a new backdoor named MoonWalk. MoonWalk shares many evasion techniques implemented in DodgeBox and utilizes Google Drive for command-and-control (C2) communication.

This two-part blog series aims to provide detailed technical analysis of both the DodgeBox loader and the MoonWalk backdoor. The goal is to assist blue teams in comprehending this emerging threat and offer insights into our attribution of the threat. Part 1 will offer an in-depth examination of the DodgeBox loader, highlighting its distinct characteristics and resemblances to StealthVector while Part 2 will delve into the intricacies of the MoonWalk backdoor.

Key Takeaways

- APT41, a China-based nation state threat actor known for its campaigns in Southeast Asian countries, has recently been observed deploying an advanced and upgraded version of StealthVector. We have named this new variant DodgeBox.
- DodgeBox incorporates various evasive techniques such as call stack spoofing, DLL sideloading, DLL hollowing and environmental guardrails. These techniques work together to significantly decrease the chances of detection by security defenses.
- Upon analyzing DodgeBox, we discovered significant resemblances to the well-known StealthVector loader used by APT41. These similarities, combined with the distinct utilization of DLL side loading and the acquisition of telemetry data from targeted countries, have led us to attribute this new loader to APT41 / Earth Baku with a moderate level of confidence.

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.

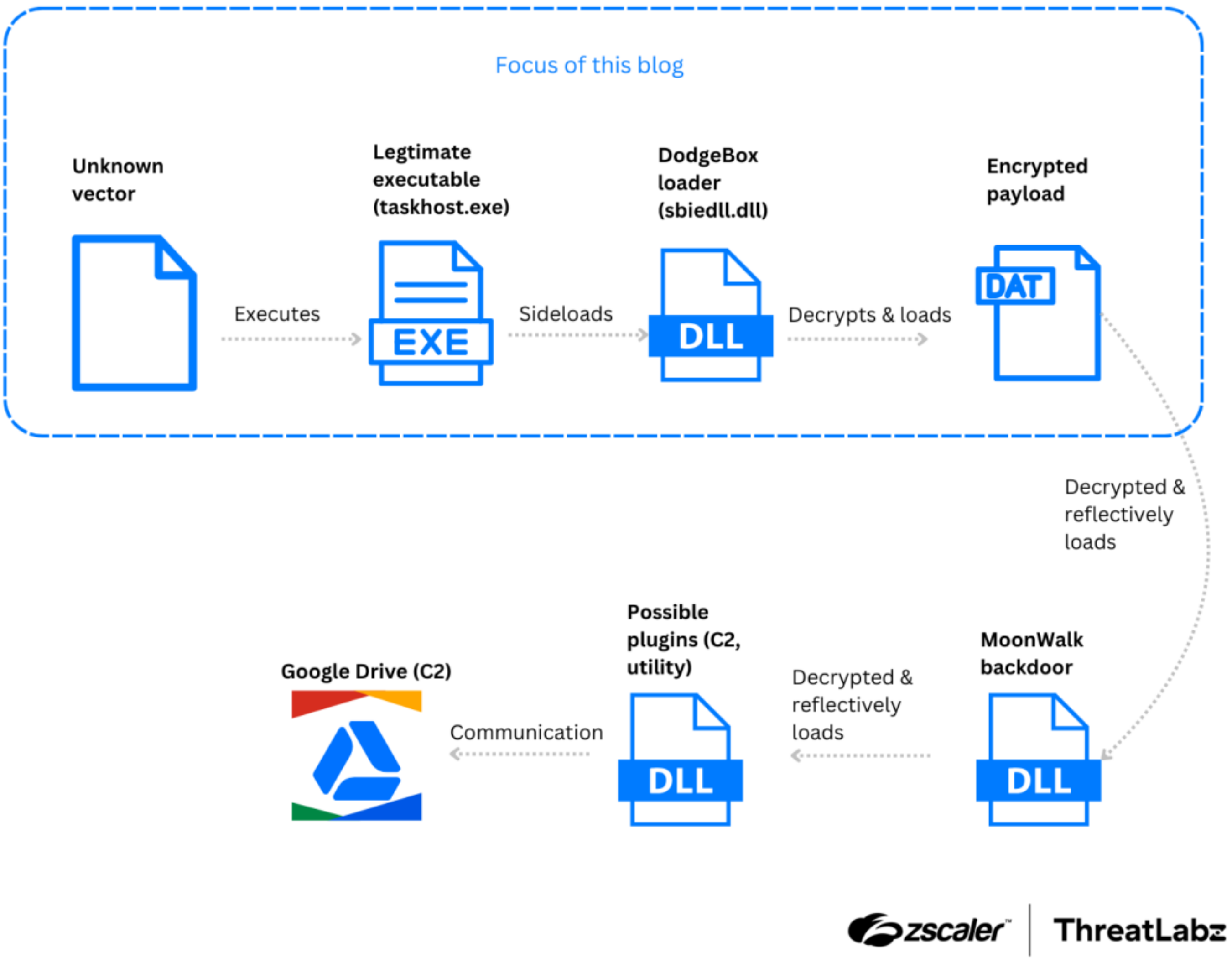


Figure 1: Attack chain used to deploy the DodgeBox loader and MoonWalk backdoor.

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 < pwszInputStringEnd )
{
 do  // Inlined FNV1a hash
 {
   chChar = *pwszInputString_Char;
   pwszInputString_Char = (pwszInputString_Char + 1);
   dwHash = 0x1000193 * (dwHash ^ chChar);
 }
 while ( pwszInputString_Char < pwszInputStringEnd );
}
v17 = &g_HashSaltPostfix;    // Salt value: CB 24 B4 BA
do  // Inlined FNV1a hash, use previous hash as seed
{
 v18 = *v17;
 v17 = (v17 + 1);
 dwHash = 0x1000193 * (dwHash ^ v18);
}
while ( v17 < g_HashSaltPostfix_End );
```

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<int>::~_DebugMallocator<int>`, 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__<random bytes from pcrt4!UuidCreate>\<name of chosen DLL>.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.

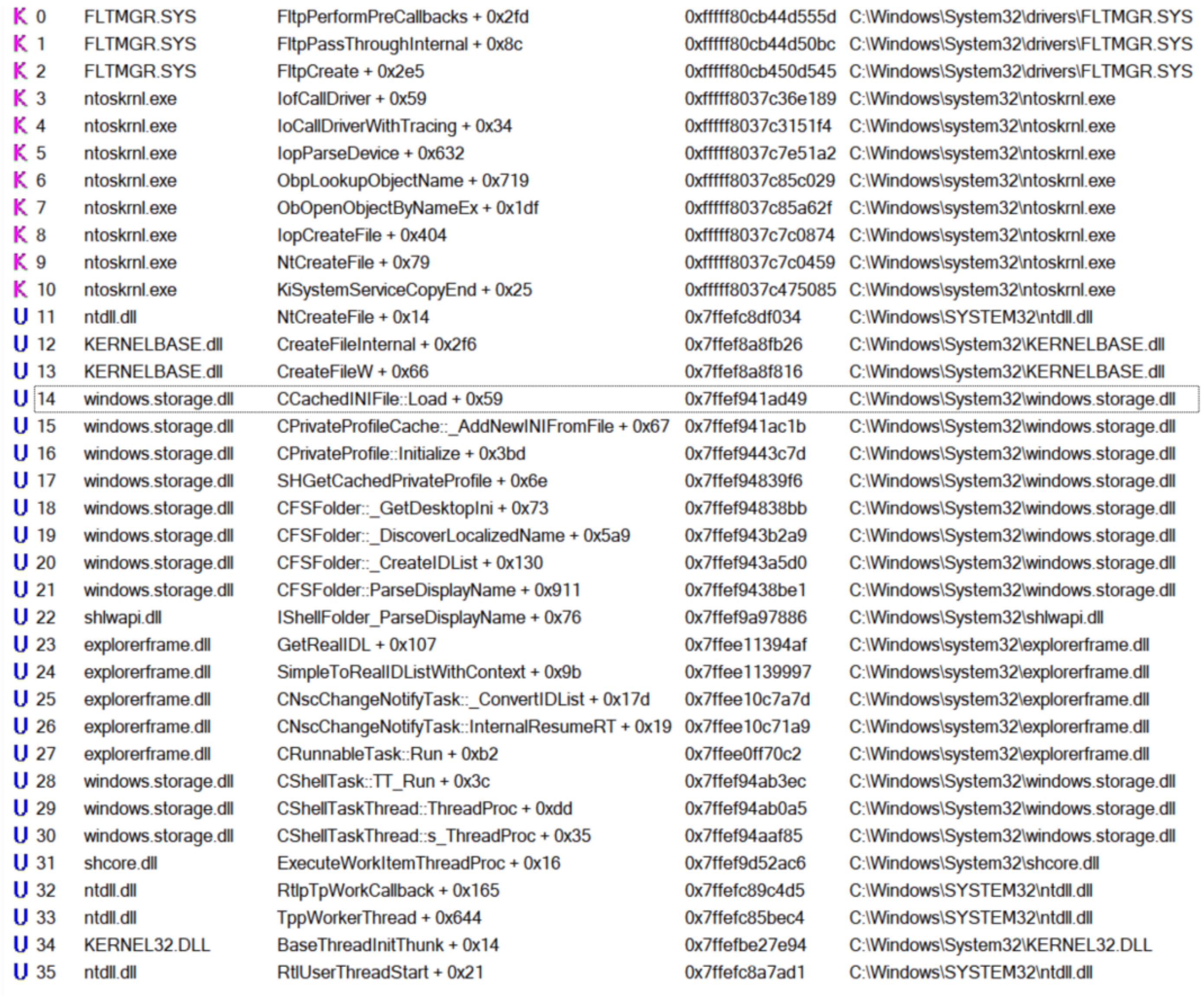
DodgeBox specifically utilizes call stack spoofing when invoking Windows APIs that are more likely to be monitored. As an example, it directly calls `RtlInitUnicodeString`, a Windows API that only performs string manipulation, instead of using stack spoofing.

```
(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.

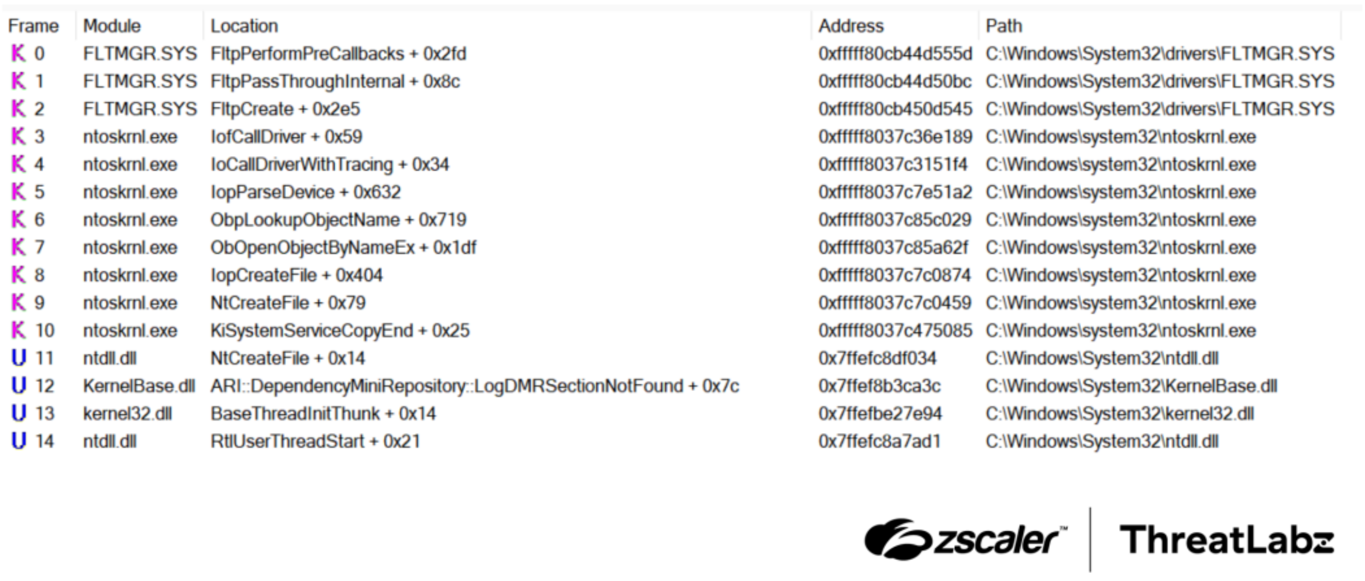


K	0	FLTMGR.SYS	FltpPerformPreCallbacks + 0x2fd	0xfffff80cb44d555d	C:\Windows\System32\drivers\FLTMGR.SYS
K	1	FLTMGR.SYS	FltpPassThroughInternal + 0x8c	0xfffff80cb44d50bc	C:\Windows\System32\drivers\FLTMGR.SYS
K	2	FLTMGR.SYS	FltpCreate + 0x2e5	0xfffff80cb450d545	C:\Windows\System32\drivers\FLTMGR.SYS
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

zscaler | ThreatLabz

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.

Threat Attribution

In this section, we outline the different tactics, techniques, and procedures (TTPs) that were utilized as indicators during our threat attribution process. Through the identification of these overlapping TTPs, we attribute this activity to a China-based threat actor known as APT41. Our confidence level in this attribution is medium.

Abuse of DLL sideloading

DLL sideloading is a technique commonly utilized by APT groups with links to China. Typically, this method involves three essential components: a legitimate executable (EXE) file that is signed, a malicious DLL file, and an encrypted data file. While the specific combination of the EXE and DLL files mentioned here has not been publicly documented as being associated with APT41, the presence of these three components could indicate the involvement of a group linked to China.

Targeted regions

Analysis of the telemetry available in VirusTotal reveals that DodgeBox samples have been submitted from both Thailand and Taiwan. This observation aligns with previous instances of APT41 employing StealthVector in campaigns primarily targeting users in the Southeast Asian (SEA) region.

Furthermore, during the monitoring of the attacker-controlled Google Drive account utilized for C2 communication, a spreadsheet containing the personal details of individuals from India was discovered. This spreadsheet is publicly available from other sources, suggesting that the threat actor may have leveraged it to identify potential additional targets.

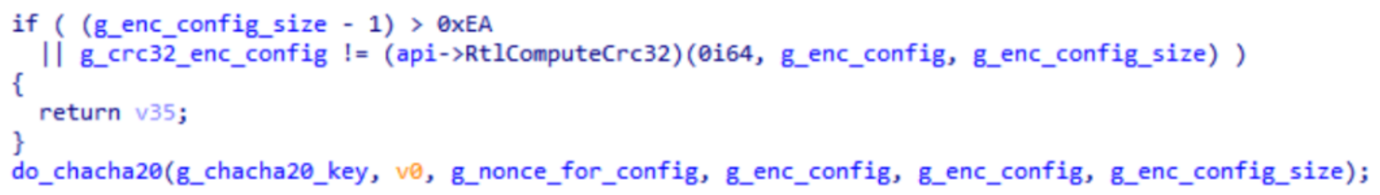
Similarities between DodgeBox and StealthVector

During our analysis of DodgeBox, we noted a number of similarities with StealthVector. In this section, we compare the code between variants of StealthVector uploaded to VirusTotal in 2021 and 2024, along with DodgeBox.

Similarities in checksum and configuration decryption

Both StealthVector and DodgeBox perform an integrity check on their encrypted configurations. This verification process consists of two essential steps. First, the hard-coded size of the configuration is validated, ensuring it matches the expected size. Second, the hash of the configuration is verified to ensure its integrity. Once these checks are successfully completed, the malware proceeds with decrypting the configuration.

StealthVector (2021)

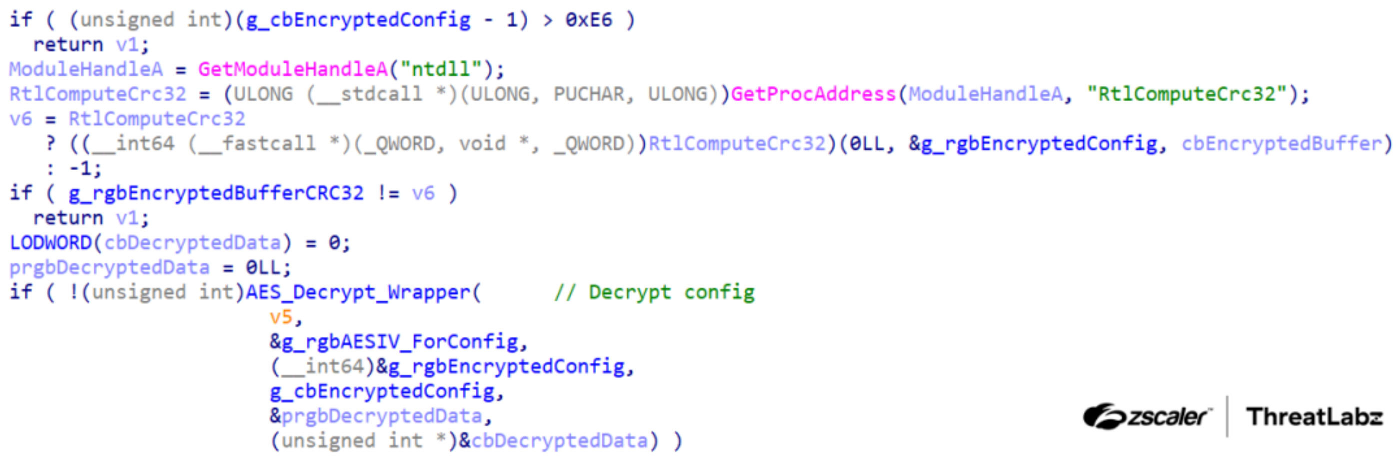


```
if ( (g_enc_config_size - 1) > 0xEA
  || g_crc32_enc_config != (api->RtlComputeCrc32)(0i64, g_enc_config, g_enc_config_size) )
{
  return v35;
}
do_chacha20(g_chacha20_key, v0, g_nonce_for_config, g_enc_config, g_enc_config, g_enc_config_size);
```

Figure 4: StealthVector uses the CRC32 hashing algorithm and the ChaCha20 algorithm for decryption (screenshot from TrendMicro).

Old variants of StealthVector use a CRC32 hashing algorithm for integrity checks and ChaCha20 for decryption of the configuration.

StealthVector (2024)

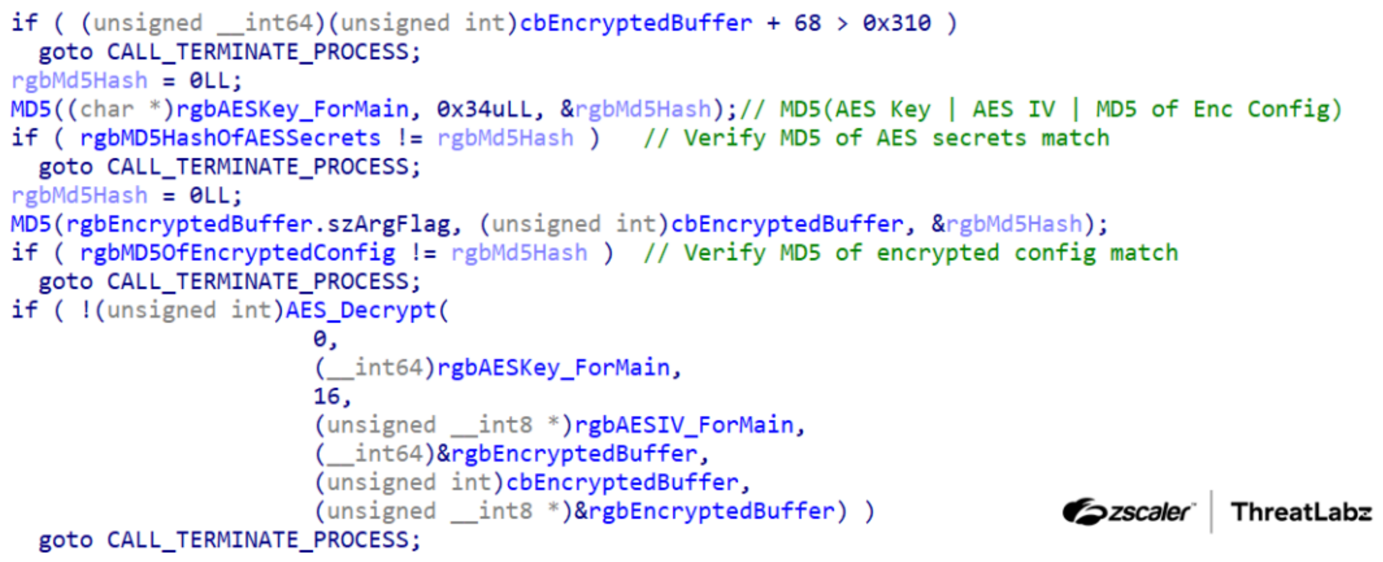


```
if ( (unsigned int)(g_cbEncryptedConfig - 1) > 0xE6 )
  return v1;
ModuleHandleA = GetModuleHandleA("ntdll");
RtlComputeCrc32 = (ULONG (__stdcall *)(ULONG, PUCHAR, ULONG))GetProcAddress(ModuleHandleA, "RtlComputeCrc32");
v6 = RtlComputeCrc32
   ? ((__int64 (__fastcall *)(_QWORD, void *, _QWORD))RtlComputeCrc32)(0LL, &g_rgbEncryptedConfig, cbEncryptedBuffer)
   : -1;
if ( g_rgbEncryptedBufferCRC32 != v6 )
  return v1;
LODWORD(cbDecryptedData) = 0;
prgbDecryptedData = 0LL;
if ( !(unsigned int)AES_Decrypt_Wrapper(      // Decrypt config
                      v5,
                      &g_rgbAESIV_ForConfig,
                      (__int64)&g_rgbEncryptedConfig,
                      g_cbEncryptedConfig,
                      &prgbDecryptedData,
                      (unsigned int *)&cbDecryptedData) )
```

Figure 5: StealthVector uses the CRC32 hashing algorithm and AES-CBC algorithm for decryption.

Newer variants of StealthVector use a CRC32 hashing algorithm, and AES-CBC for decryption.

DodgeBox



```
if ( (unsigned __int64)(unsigned int)cbEncryptedBuffer + 68 > 0x310 )
  goto CALL_TERMINATE_PROCESS;
rgbMd5Hash = 0LL;
MD5((char *)rgbAESKey_ForMain, 0x34uLL, &rgbMd5Hash);// MD5(AES Key | AES IV | MD5 of Enc Config)
if ( rgbMD5HashOfAESSecrets != rgbMd5Hash )   // Verify MD5 of AES secrets match
  goto CALL_TERMINATE_PROCESS;
rgbMd5Hash = 0LL;
MD5(rgbEncryptedBuffer.szArgFlag, (unsigned int)cbEncryptedBuffer, &rgbMd5Hash);
if ( rgbMD5OfEncryptedConfig != rgbMd5Hash )  // Verify MD5 of encrypted config match
  goto CALL_TERMINATE_PROCESS;
if ( !(unsigned int)AES_Decrypt(
                      0,
                      (__int64)rgbAESKey_ForMain,
                      16,
                      (unsigned __int8 *)rgbAESIV_ForMain,
                      (__int64)&rgbEncryptedBuffer,
                      (unsigned int)cbEncryptedBuffer,
                      (unsigned __int8 *)&rgbEncryptedBuffer) )
  goto CALL_TERMINATE_PROCESS;
```

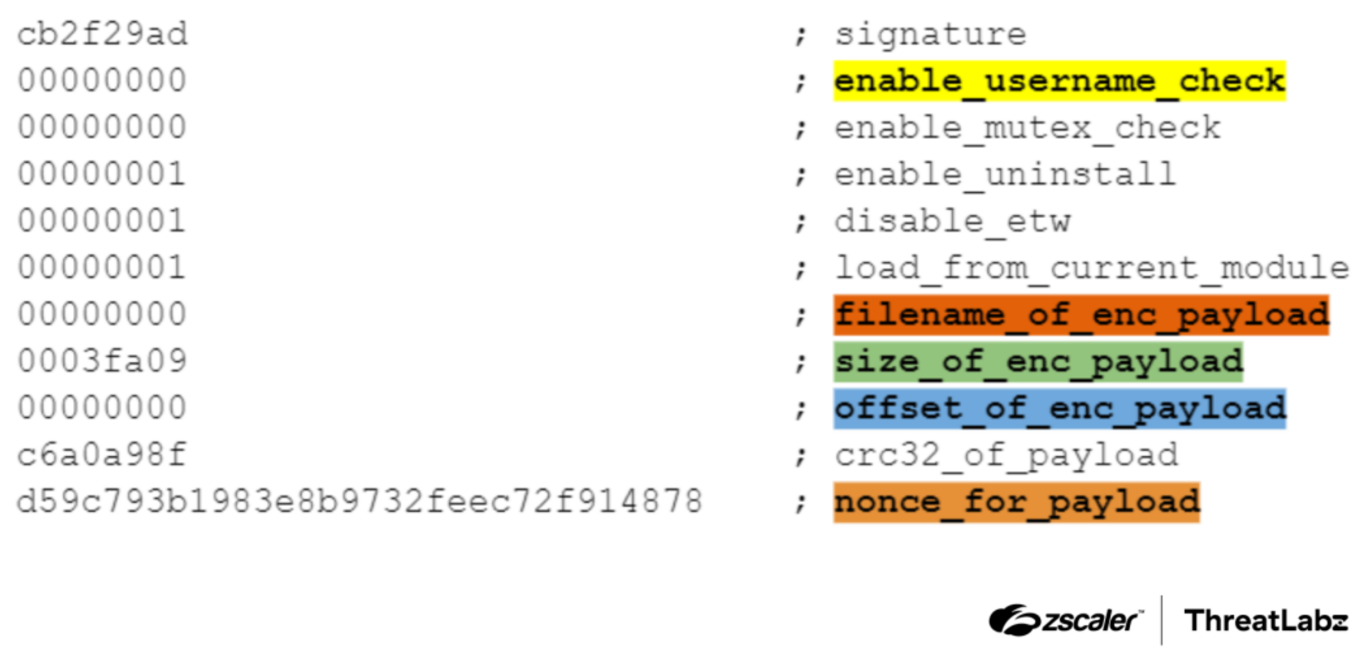
Figure 6: DodgeBox uses the MD5 hashing algorithm and AES-CFB algorithm for decryption.

DodgeBox uses an MD5 hashing algorithm for integrity checks, and AES-CFB for decryption of the configuration.

Similarities in decrypted configuration format

These similarities encompass various aspects such as guardrails, payload filenames, sizes and offsets, as well as cryptographic secrets. Both the original StealthVector and DodgeBox configurations also incorporate checksums for their encrypted payloads.

StealthVector (2021)

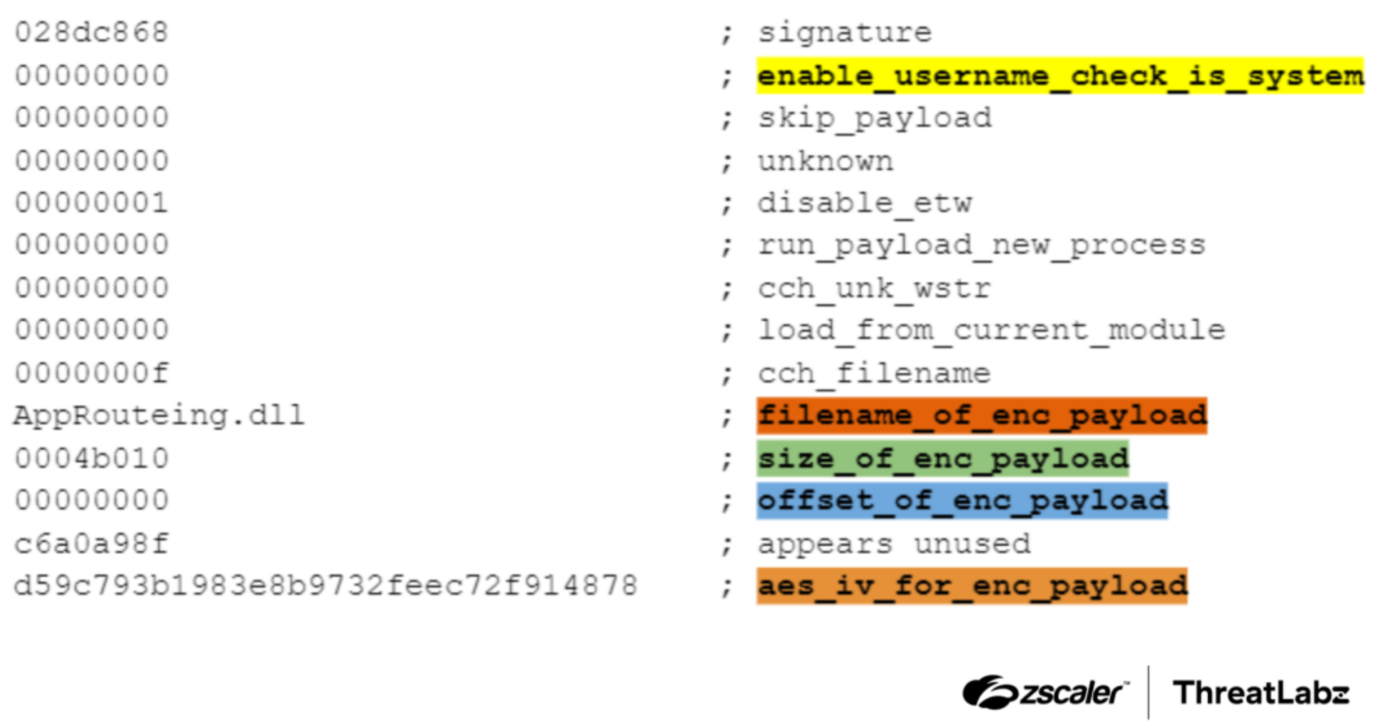


```
cb2f29ad                              ; signature
00000000                              ; enable_username_check
00000000                              ; enable_mutex_check
00000001                              ; enable_uninstall
00000001                              ; disable_etw
00000001                              ; load_from_current_module
00000000                              ; filename_of_enc_payload
0003fa09                              ; size_of_enc_payload
00000000                              ; offset_of_enc_payload
c6a0a98f                              ; crc32_of_payload
d59c793b1983e8b9732feec72f914878      ; nonce_for_payload
```

Figure 7: Configuration extracted from the 2021 variant of StealthVector.

The configuration extracted from the 2021 variant of StealthVector reveals several similarities with the 2024 variant of StealthVector and DodgeBox.

StealthVector (2024)

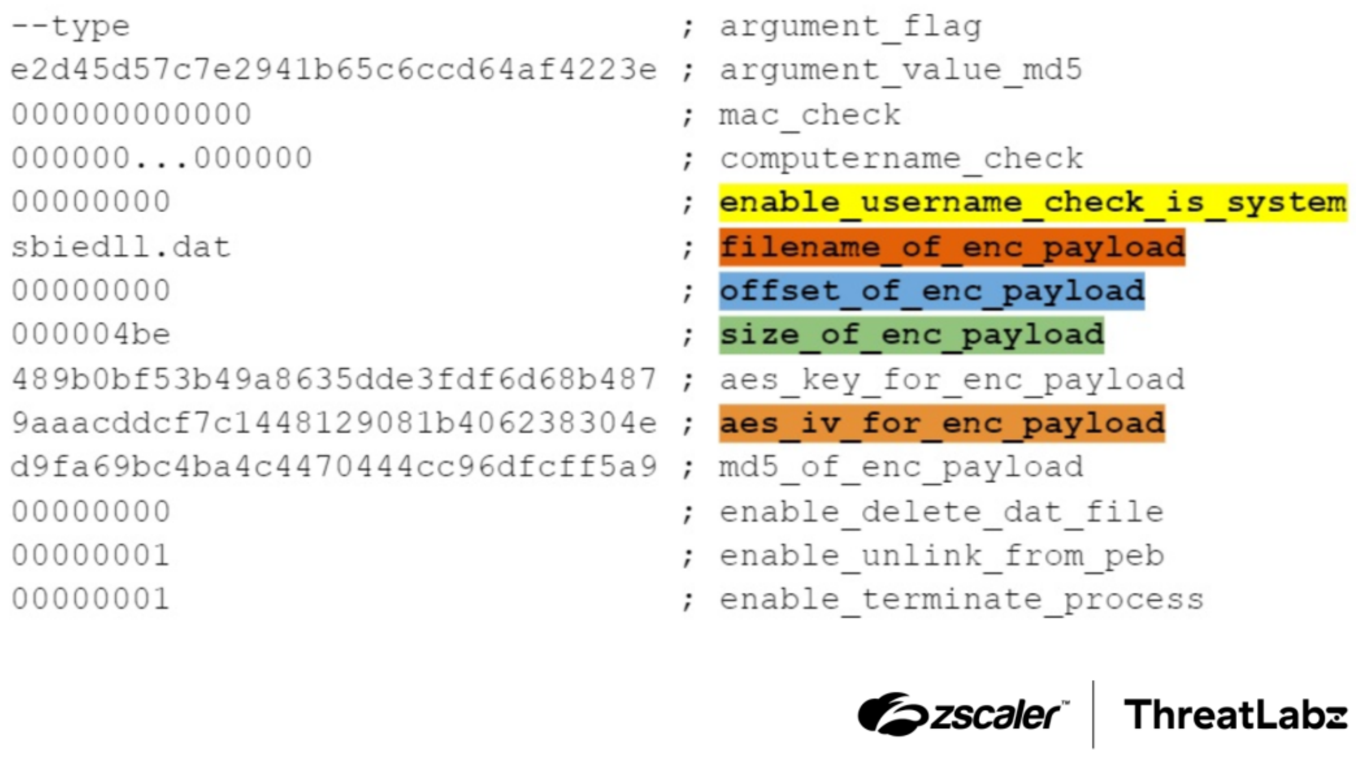


```
028dc868                              ; signature
00000000                              ; enable_username_check_is_system
00000000                              ; skip_payload
00000000                              ; unknown
00000001                              ; disable_etw
00000000                              ; run_payload_new_process
00000000                              ; cch_unk_wstr
00000000                              ; load_from_current_module
0000000f                              ; cch_filename
AppRouteing.dll                       ; filename_of_enc_payload
0004b010                              ; size_of_enc_payload
00000000                              ; offset_of_enc_payload
c6a0a98f                              ; appears unused
d59c793b1983e8b9732feec72f914878      ; aes_iv_for_enc_payload
```

Figure 8: Configuration extracted from the 2024 variant of StealthVector.

The configuration extracted from the 2024 variant of StealthVector reveals several similarities with the 2021 variant of StealthVector and DodgeBox.

DodgeBox



```
--type                            ; argument_flag
e2d45d57c7e2941b65c6ccd64af4223e ; argument_value_md5
000000000000                      ; mac_check
000000...000000                   ; computername_check
00000000                          ; enable_username_check_is_system
sbiedll.dat                       ; filename_of_enc_payload
00000000                          ; offset_of_enc_payload
000004be                          ; size_of_enc_payload
489b0bf53b49a8635dde3fdf6d68b487 ; aes_key_for_enc_payload
9aaacddcf7c1448129081b406238304e ; aes_iv_for_enc_payload
d9fa69bc4ba4c4470444cc96dfcff5a9 ; md5_of_enc_payload
00000000                          ; enable_delete_dat_file
00000001                          ; enable_unlink_from_peb
00000001                          ; enable_terminate_process
```

Figure 9: Configuration extracted from DodgeBox.

The configuration extracted from DodgeBox reveals several similarities with the 2024 and 2021 variant of StealthVector.

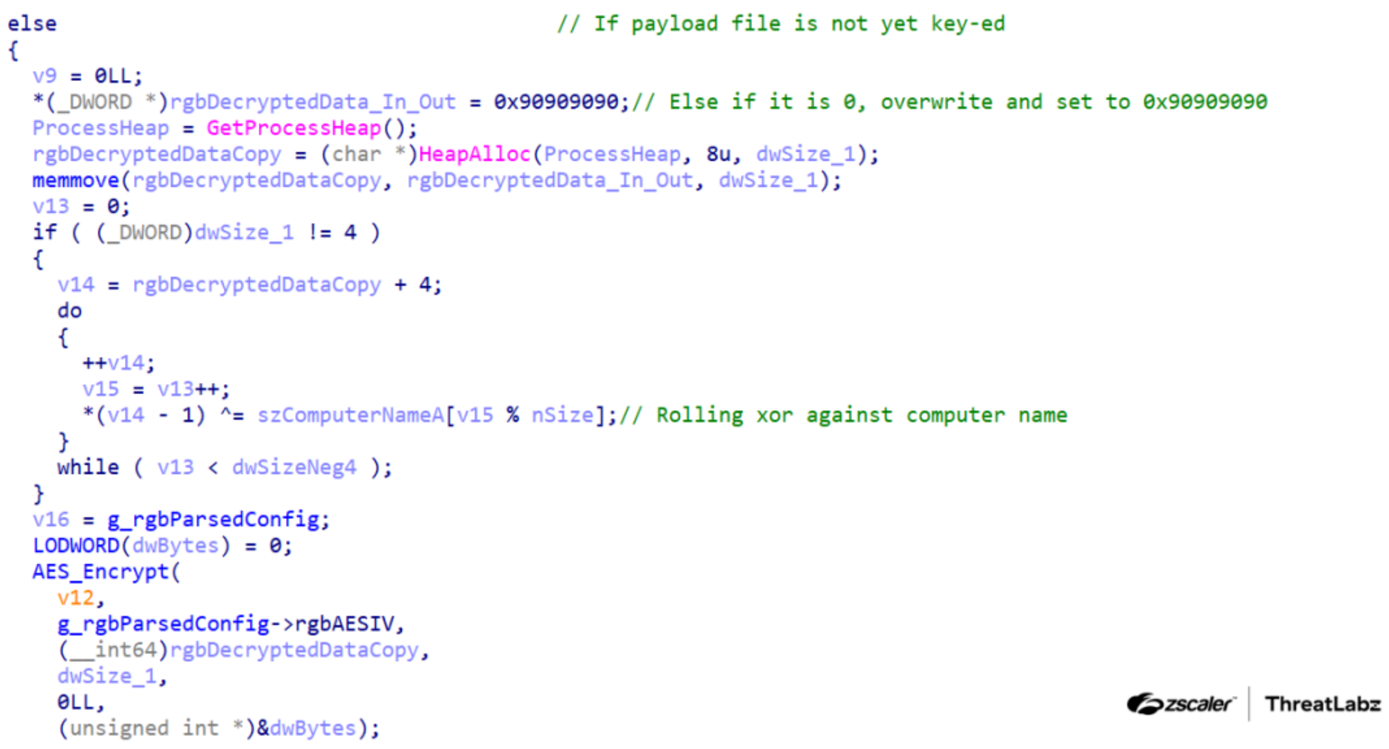
Similarities in environment keying

Both StealthVector and DodgeBox perform environment keying by decrypting then re-encrypting the bundled payload.

StealthVector (2021)

TrendMicro’s report did not document StealthVector utilizing environment keying.

StealthVector (2024)



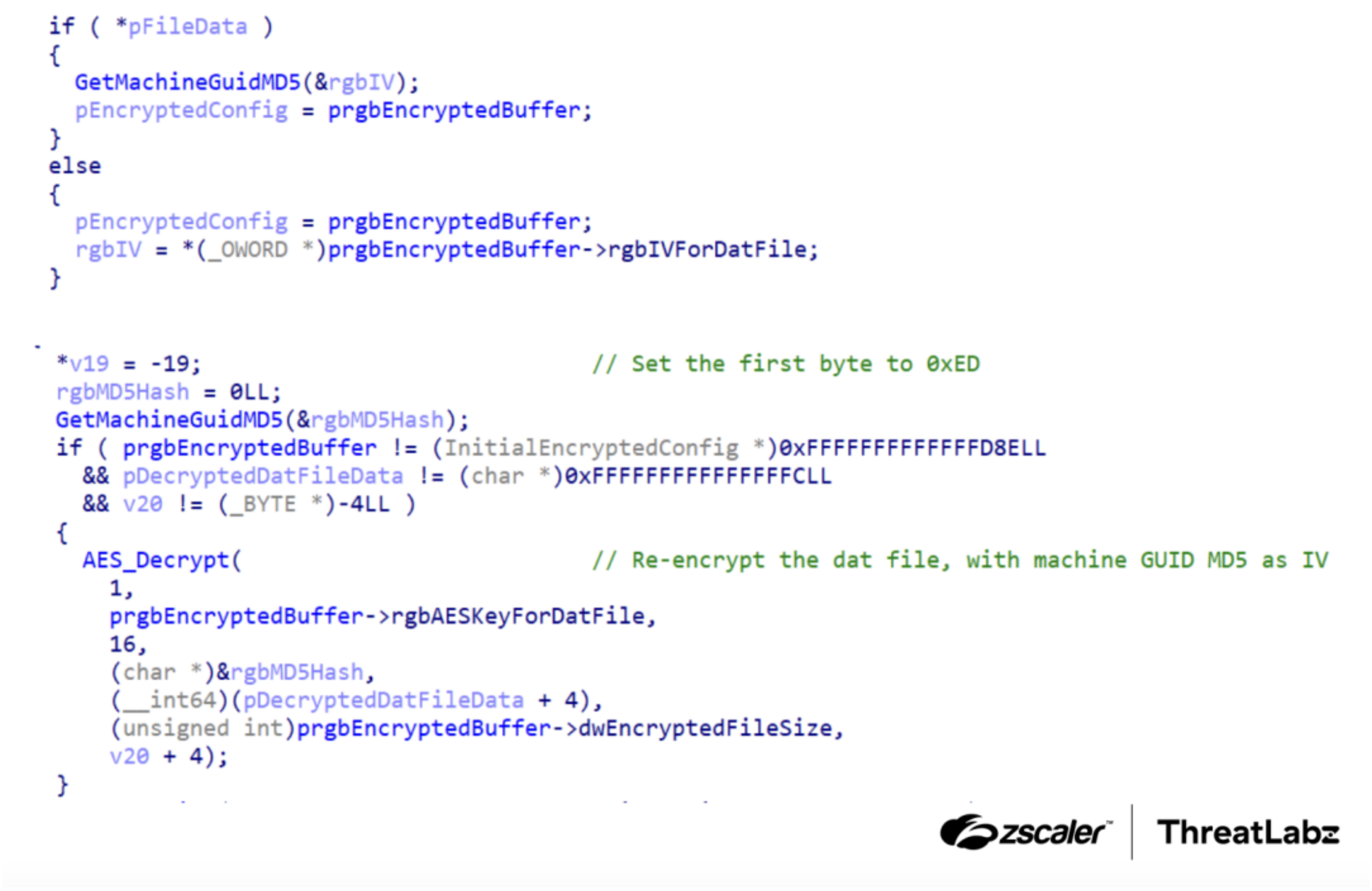
```
else                                        // If payload file is not yet key-ed
{
  v9 = 0LL;
  *(_DWORD *)rgbDecryptedData_In_Out = 0x90909090;// Else if it is 0, overwrite and set to 0x90909090
  ProcessHeap = GetProcessHeap();
  rgbDecryptedDataCopy = (char *)HeapAlloc(ProcessHeap, 8u, dwSize_1);
  memmove(rgbDecryptedDataCopy, rgbDecryptedData_In_Out, dwSize_1);
  v13 = 0;
  if ( (_DWORD)dwSize_1 != 4 )
  {
    v14 = rgbDecryptedDataCopy + 4;
    do
    {
      ++v14;
      v15 = v13++;
      *(v14 - 1) ^= szComputerNameA[v15 % nSize];// Rolling xor against computer name
    }
    while ( v13 < dwSizeNeg4 );
  }
  v16 = g_rgbParsedConfig;
  LODWORD(dwBytes) = 0;
  AES_Encrypt(
    v12,
    g_rgbParsedConfig->rgbAESIV,
    (__int64)rgbDecryptedDataCopy,
    dwSize_1,
    0LL,
    (unsigned int *)&dwBytes);
```

Figure 10: 2024 variant of StealthVector performing environment keying, using a rolling XOR against the computer name.

The updated version of StealthVector employs the first four bytes of the payload (`rgbDecryptedData_In_Out`) to check whether the payload has been keyed. If the payload has not been previously keyed, StealthVector proceeds to key it using the computer name of the target machine.

This keying process involves a rolling XOR operation to encode the payload, followed by re-encryption using AES. In the analyzed sample, StealthVector sets the first four bytes of the payload to `0x90909090`, serving as an indicator that the payload has been successfully keyed.

DodgeBox



```
if ( *pFileData )
{
  GetMachineGuidMD5(&rgbIV);
  pEncryptedConfig = prgbEncryptedBuffer;
}
else
{
  pEncryptedConfig = prgbEncryptedBuffer;
  rgbIV = *(_OWORD *)prgbEncryptedBuffer->rgbIVForDatFile;
}


 *v19 = -19;                          // Set the first byte to 0xED
 rgbMD5Hash = 0LL;
 GetMachineGuidMD5(&rgbMD5Hash);
 if ( prgbEncryptedBuffer != (InitialEncryptedConfig *)0xFFFFFFFFFFFFFD8ELL
   && pDecryptedDatFileData != (char *)0xFFFFFFFFFFFFFFFCLL
   && v20 != (_BYTE *)-4LL )
 {
   AES_Decrypt(                       // Re-encrypt the dat file, with machine GUID MD5 as IV
     1,
     prgbEncryptedBuffer->rgbAESKeyForDatFile,
     16,
     (char *)&rgbMD5Hash,
     (__int64)(pDecryptedDatFileData + 4),
     (unsigned int)prgbEncryptedBuffer->dwEncryptedFileSize,
     v20 + 4);
 }
```

Figure 11: DodgeBox uses a technique called environment keying, where it uses the hash of the machine's GUID as the AES Initialization Vector (IV).

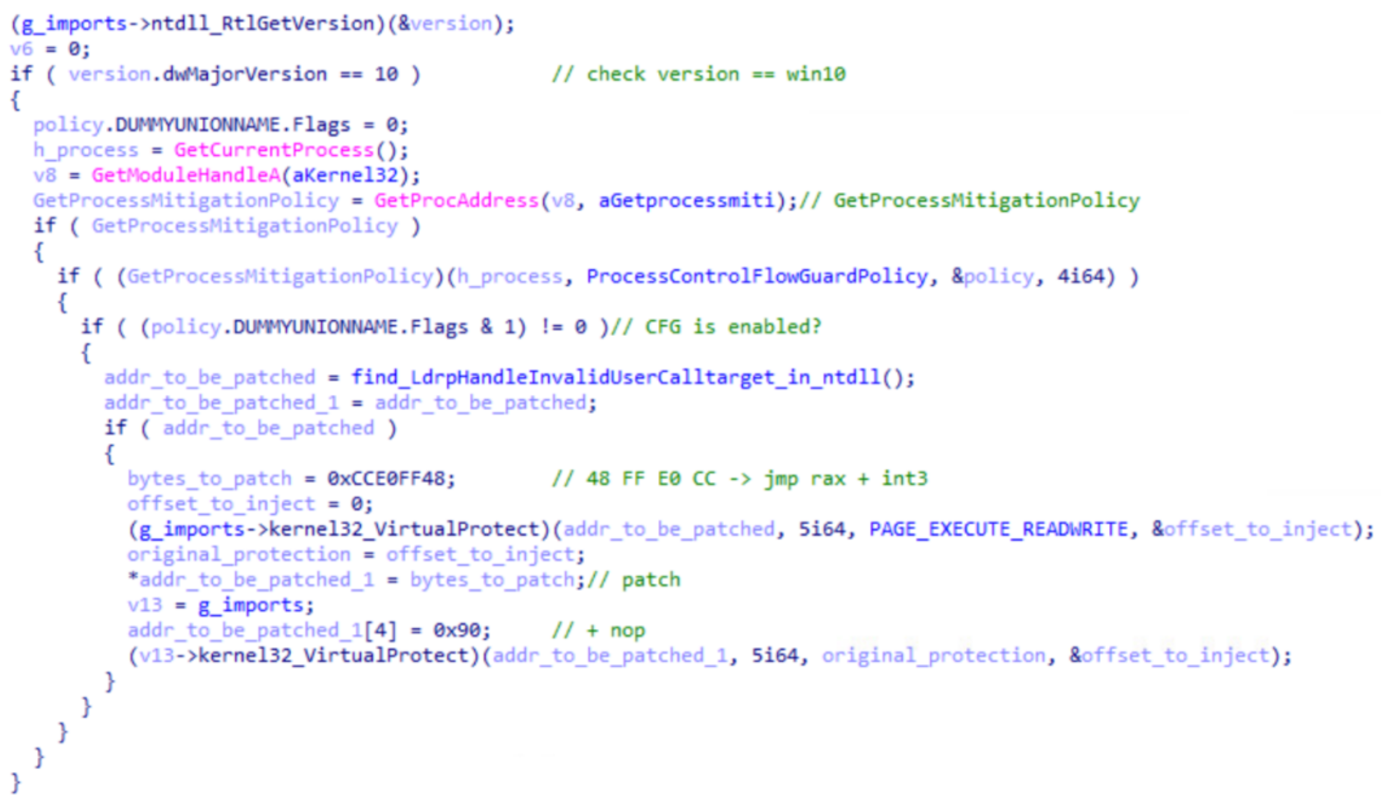
DodgeBox employs the first four bytes of the payload (`pFileData`) to determine whether it has been keyed. If the payload has not been previously keyed, DodgeBox decrypts the payload using the AES IV from its configuration. DodgeBox then proceeds to re-encrypt it using the MD5 hash of the target machine's MachineGUID as the new AES IV.

In the given sample, DodgeBox sets the first four bytes of the payload to `0x000000ED`. This non-zero value serves as an indicator that the payload has indeed been keyed and should be decrypted with the new AES IV.

Similarities in disabling CFG

All three samples exhibit remarkably similar logic in their approach to patching CFG. This similarity extends to the use of identical byte patterns for locating the LdrpHandleInvalidUserCallTarget function, as well as applying the same patch in this function.

StealthVector (2021)

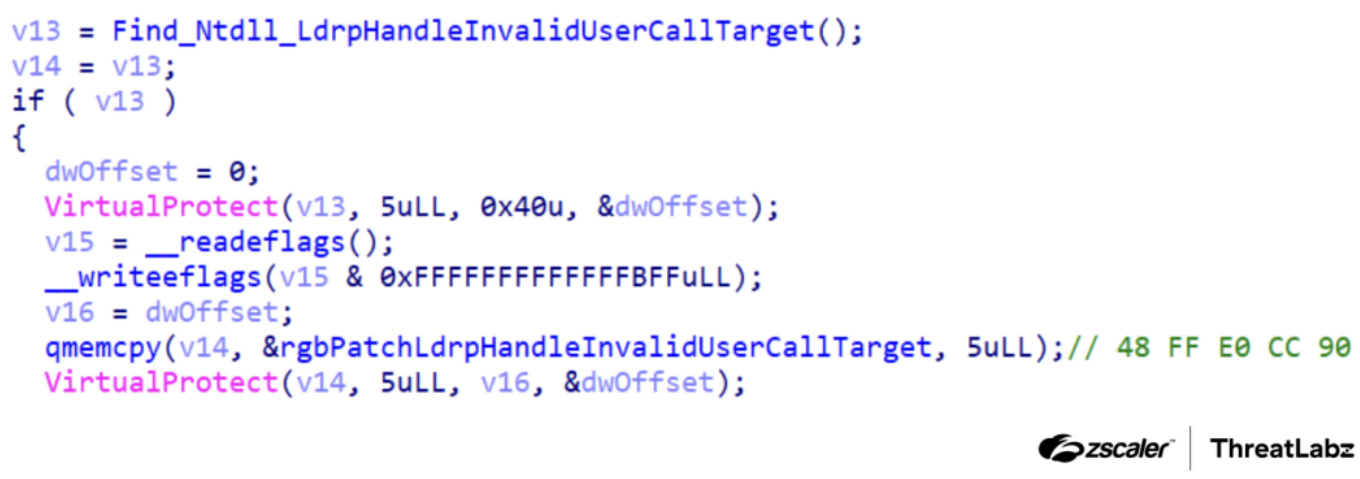


```
(g_imports->ntdll_RtlGetVersion)(&version);
v6 = 0;
if ( version.dwMajorVersion == 10 )           // check version == win10
{
  policy.DUMMYUNIONNAME.Flags = 0;
  h_process = GetCurrentProcess();
  v8 = GetModuleHandleA(aKernel32);
  GetProcessMitigationPolicy = GetProcAddress(v8, aGetprocessmiti);// GetProcessMitigationPolicy
  if ( GetProcessMitigationPolicy )
  {
    if ( (GetProcessMitigationPolicy)(h_process, ProcessControlFlowGuardPolicy, &policy, 4i64) )
    {
      if ( (policy.DUMMYUNIONNAME.Flags & 1) != 0 )// CFG is enabled?
      {
        addr_to_be_patched = find_LdrpHandleInvalidUserCalltarget_in_ntdll();
        addr_to_be_patched_1 = addr_to_be_patched;
        if ( addr_to_be_patched )
        {
          bytes_to_patch = 0xCCE0FF48;         // 48 FF E0 CC -> jmp rax + int3
          offset_to_inject = 0;
          (g_imports->kernel32_VirtualProtect)(addr_to_be_patched, 5i64, PAGE_EXECUTE_READWRITE, &offset_to_inject);
          original_protection = offset_to_inject;
          *addr_to_be_patched_1 = bytes_to_patch;// patch
          v13 = g_imports;
          addr_to_be_patched_1[4] = 0x90;      // + nop
          (v13->kernel32_VirtualProtect)(addr_to_be_patched_1, 5i64, original_protection, &offset_to_inject);
        }
      }
    }
  }
}
```

Figure 12: Code from the 2021 variant of StealthVector disabling CFG (screenshot from TrendMicro).

The code extracted from the 2021 variant of StealthVector showcases the disabling of CFG with striking similarity to all three samples.

StealthVector (2024)

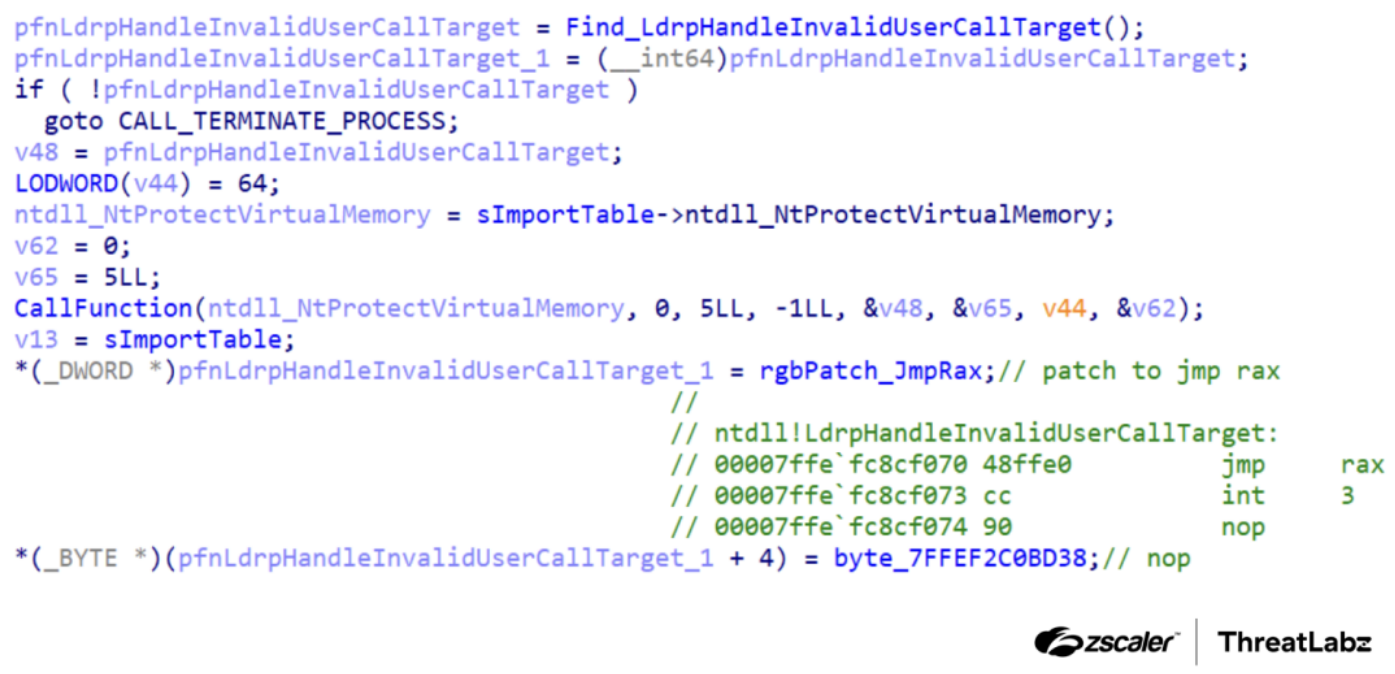


```
v13 = Find_Ntdll_LdrpHandleInvalidUserCallTarget();
v14 = v13;
if ( v13 )
{
  dwOffset = 0;
  VirtualProtect(v13, 5uLL, 0x40u, &dwOffset);
  v15 = __readeflags();
  __writeeflags(v15 & 0xFFFFFFFFFFFFFBFFuLL);
  v16 = dwOffset;
  qmemcpy(v14, &rgbPatchLdrpHandleInvalidUserCallTarget, 5uLL);// 48 FF E0 CC 90
  VirtualProtect(v14, 5uLL, v16, &dwOffset);
```

Figure 13: Code from the 2024 variant of StealthVector, disabling CFG.

The code extracted from the 2024 variant of StealthVector showcases the disabling of CFG with striking similarity to all three samples.

DodgeBox



```
pfnLdrpHandleInvalidUserCallTarget = Find_LdrpHandleInvalidUserCallTarget();
pfnLdrpHandleInvalidUserCallTarget_1 = (__int64)pfnLdrpHandleInvalidUserCallTarget;
if ( !pfnLdrpHandleInvalidUserCallTarget )
  goto CALL_TERMINATE_PROCESS;
v48 = pfnLdrpHandleInvalidUserCallTarget;
LODWORD(v44) = 64;
ntdll_NtProtectVirtualMemory = sImportTable->ntdll_NtProtectVirtualMemory;
v62 = 0;
v65 = 5LL;
CallFunction(ntdll_NtProtectVirtualMemory, 0, 5LL, -1LL, &v48, &v65, v44, &v62);
v13 = sImportTable;
*(_DWORD *)pfnLdrpHandleInvalidUserCallTarget_1 = rgbPatch_JmpRax;// patch to jmp rax
                                                  //
                                                  // ntdll!LdrpHandleInvalidUserCallTarget:
                                                  // 00007ffe`fc8cf070 48ffe0          jmp     rax
                                                  // 00007ffe`fc8cf073 cc              int     3
                                                  // 00007ffe`fc8cf074 90              nop
*(_BYTE *)(pfnLdrpHandleInvalidUserCallTarget_1 + 4) = byte_7FFEF2C0BD38;// nop
```

Figure 14: Code from DodgeBox disabling CFG.

The code extracted from DodgeBox showcases the disabling of CFG with striking similarity to all three samples.

Similarities in the use of DLL Hollowing

All three samples exhibit the capability to load bundled payloads through DLL hollowing. Notably, the 2024 version of StealthVector shares an identical list of blocklisted DLLs with DodgeBox.

To Be Continued

DodgeBox is a newly identified malware loader that employs multiple techniques to evade both static and behavioral detection. Based on a combination of known TTPs, potential countries targeted, and similarities with StealthVector, we have attributed this activity to the China-based nation state threat actor APT41 with moderate confidence. In our journey through Part 1 of this series, we analyzed the technical details surrounding DodgeBox, and its similarities with StealthVector. In Part 2, we will analyze the MoonWalk backdoor - which is dropped by DodgeBox.

Indicators Of Compromise (IOCs)

MD5	Filename	Description
0d068b6d0523f069d1ada59c12891c4a	`Music.zip`	ZIP archive containing DodgeBox samples.
b3067f382d70705d4c8f6977a7d7bee4	`taskhost.exe`	Original Sandboxie signed binary.
d72f202c1d684c9a19f075290a60920f	`Sbiedll.dll`	DodgeBox DLL sideloaded by taskhost.exe.
294cc02db5a122e3a1bc4f07997956da	`Sbiedll.dat`	Encrypted payload DLL that decrypts to the MoonWalk backdoor.
393065ef9754e3f39b24b2d1051eab61	`Atstrust.dll`	DodgeBox DLL which is sideloaded by an undetermined AhnLab executable.
bcac2cbda36019776d7861f12d9b59c4	`Atstrust.dat`	Encrypted payload DLL that decrypts the MoonWalk backdoor.
f062183da590aba5e911d2392bc29181	`AppRouted.dll`	2024 StealthVector loader.

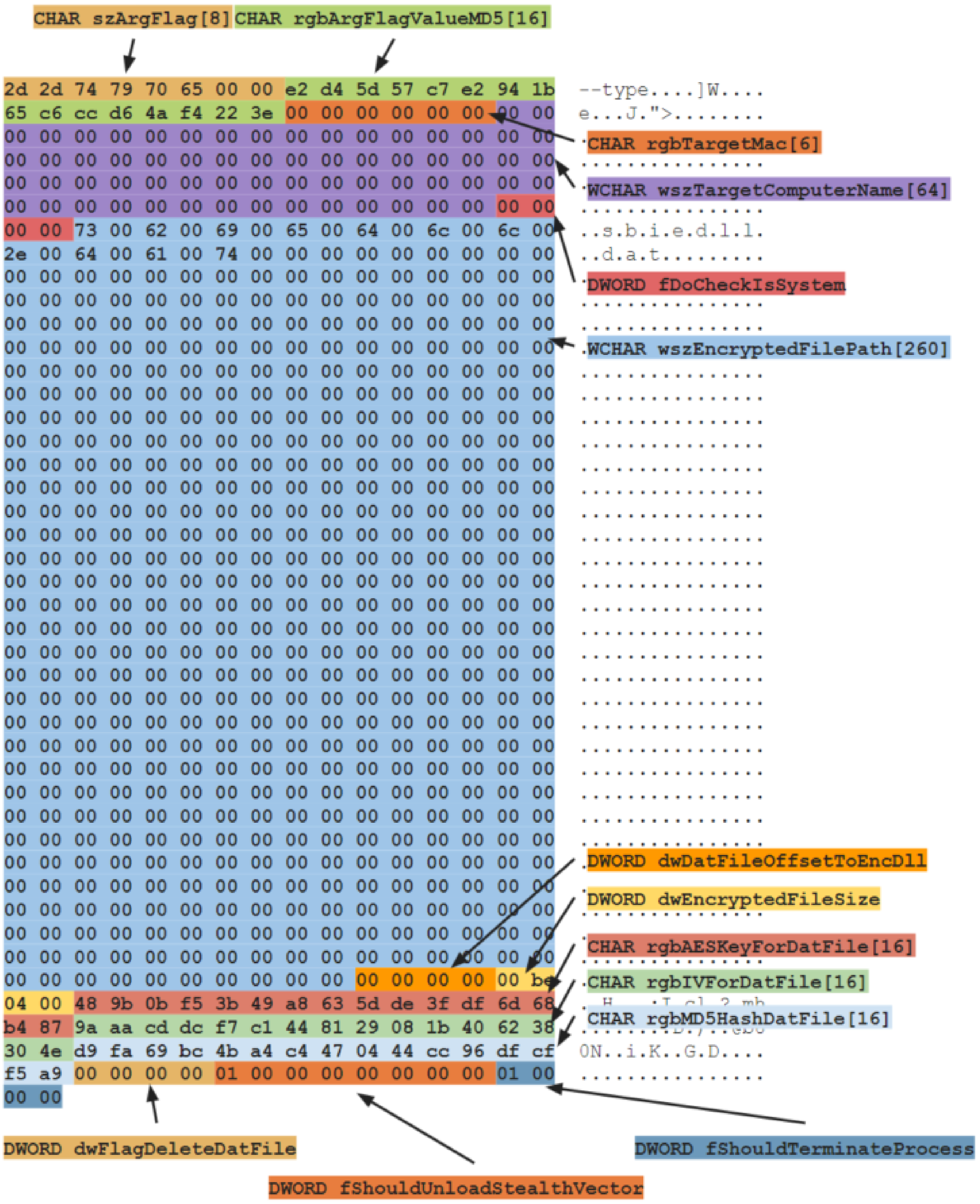
MD5	Filename	Description
4141c4b827ff67c180096ff5f2cc1474	`AppRouteing.dll`	Encrypted shellcode and payload DLL that decrypts to CobaltStrike.
bc85062de0f70afd44bb072b0b71a8cc	`N/A`	2024 StealthVector loader
72070b165d1f11bd4d009a81bf28a3e5	`mscms.dll`	2024 StealthVector loader
f0953ed4a679b987a2da955788737602	`roboform-x64.dll`	2024 StealthVector loader

MITRE ATT&CK Framework

Tactic	ID	Technique	Description
Defense Evasion	T1574.002	Hijack Execution Flow: DLL Side-Loading	DodgeBox samples are designed to be executed by DLL sideloading.
Defense Evasion	T1480	Execution Guardrails	DodgeBox terminates execution if specific arguments are not provided. DodgeBox contains capabilities to restrict execution to machines with specific MAC addresses, computer names, and user names.
Defense Evasion	T1480.001	Execution Guardrails: Environmental Keying	DodgeBox keys the encrypted payload to a machine, using a machine's GUID.
Defense Evasion	T1027	Obfuscated Files or Information	DodgeBox uses AES-CFB to encrypt strings, configurations, and bundled payloads.
Defense Evasion	T1027.007	Obfuscated Files or Information: Dynamic API Resolution	DodgeBox uses salted FNV1a hashes to dynamically resolve APIs.
Defense Evasion	T1620	Reflective Code Loading	DodgeBox reflectively loads payload DLLs, utilizing DLL hollowing.
Defense Evasion	T1106	Native API	DodgeBox uses Windows Native APIs like `NtCreateFile`, `LdrLoadDll`, and `NtAllocateVirtualMemory`, as opposed to their Win32 counterparts.
Defense Evasion	T1562.001	Impair Defenses: Disable or Modify Tools	DodgeBox utilizes stack spoofing when calling APIs to evade security software monitoring. DodgeBox performs a scan within its own address space to detect any alterations, such as hooks or debugger breakpoints. If it identifies any signs of modification, DodgeBox takes action to restore the original code from disk, effectively undoing any unauthorized changes made to its code.

Appendix

An example decrypted configuration of DodgeBox is shown in the figure below.


CHAR szArgFlag[8]
CHAR rgbArgFlagValueMD5[16]
CHAR rgbTargetMac[6]
WCHAR wszTargetComputerName[64]
DWORD fDoCheckIsSystem
WCHAR wszEncryptedFilePath[260]
DWORD dwDatFileOffsetToEncDll
DWORD dwEncryptedFileSize
CHAR rgbAESKeyForDatFile[16]
CHAR rgbIVForDatFile[16]
CHAR rgbMD5HashDatFile[16]
DWORD dwFlagDeleteDatFile
DWORD fShouldTerminateProcess
DWORD fShouldUnloadStealthVector

```
2d 2d 74 79 70 65 00 00 e2 d4 5d 57 c7 e2 94 1b   --type....]W....
65 c6 cc d6 4a f4 22 3e 00 00 00 00 00 00 00 00   e...J.">........
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 73 00 62 00 69 00 65 00 64 00 6c 00 6c 00   ..s.b.i.e.d.l.l.
2e 00 64 00 61 00 74 00 00 00 00 00 00 00 00 00   ..d.a.t.........
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 be   ................
04 00 48 9b 0b f5 3b 49 a8 63 5d de 3f df 6d 68   ..H...;I.c].?.mh
b4 87 9a aa cd dc f7 c1 44 81 29 08 1b 40 62 38   ........D.)..@b8
30 4e d9 fa 69 bc 4b a4 c4 47 04 44 cc 96 df cf   0N..i.K..G.D....
f5 a9 00 00 00 00 01 00 00 00 00 00 00 00 01 00   ................
00 00
```

The Python implementation of DodgeBox's salted FNV1a hash is shown below.

```
def fnv1a_salted(data, salt, seed_value=0x811c9dc5):
   _data = data + salt
   _hash = seed_value
   prime = 0x01000193
   for byte in _data:
       _hash ^= byte
       _hash *= prime
       _hash &= 0xFFFFFFFF
   return _hash
# ntdll in utf-16
ntdll = b"n\x00t\x00d\x00l\x00l\x00"
salt = b"\xba\xb4\x24\xcb"
print(hex(fnv1a_salted(ntdll, salt)))  # 0xfe0b07b0
ldrloaddll = b"LdrLoadDll"
print(hex(fnv1a_salted(ldrloaddll, salt)))  # 0xca7bb6ac
```

DodgeBox's list of blocklisted DLLs is shown below.

- `advapi32.dll`
- `bcrypt.dll`
- `bcryptprimitives.dll`
- `cfgmgr32.dll`
- `combase.dll`
- `cryptbase.dll`
- `cryptsp.dll`
- `dhcpcsvc.dll`
- `dhcpcsvc6.dll`
- `dnsapi.dll`
- `FWPUCLNT.DLL`
- `gdi32.dll`
- `gdi32full.dll`
- `iertutil.dll`
- `imm32.dll`
- `IPHLPAPI.DLL`
- `kernel.appcore.dll`
- `kernel32.dll`
- `KernelBase.dll`
- `locale.nls`
- `msvcp_win.dll`
- `msvcrt.dll`
- `mswsock.dll`
- `NapiNSP.dll`
- `nlaapi.dll`
- `nsi.dll`
- `ntdll.dll`

- ntmarta.dll
- oleaut32.dll
- OnDemandConnRouteHelper.dll
- pnrpnsp.dll
- powrprof.dll
- apphelp.dll
- profapi.dll
- rasadhlp.dll
- rpcrt4.dll
- rsaenh.dll
- sechost.dll
- SHCore.dll
- shell32.dll
- shlwapi.dll
- sspicli.dll
- ucrtbase.dll
- urlmon.dll
- user32.dll
- userenv.dll
- webio.dll
- win32u.dll
- windows.storage.dll
- winhttp.dll
- wininet.dll
- winnlsres.dll
- winnsi.dll
- winrnr.dll
- winsta.dll
- ws2_32.dll
- wshbth.dll
- wtsapi32.dll