DodgeBox：深入探究 APT41 的最新武器库 | 第 1 部分

admin：

介绍

这是我们对 APT41 的新工具（包括 DodgeBox 和 MoonWalk）进行两部分技术深入研究的第一部分。

2024 年 4 月，Zscaler ThreatLabz 发现了一种以前未知的加载程序，名为 DodgeBox。经过进一步分析，发现 DodgeBox 与 StealthVector 变体之间存在惊人的相似之处，StealthVector 是一种与某国高级持续威胁 (APT) 行为者 APT41 / Earth Baku 相关的工具。DodgeBox 是一种加载程序，它会继续加载名为 MoonWalk 的新后门。MoonWalk 共享 DodgeBox 中实现的许多规避技术，并利用 Google Drive 进行命令和控制 (C2) 通信。

本系列博文分为两部分，旨在对 DodgeBox 加载程序和 MoonWalk 后门进行详细的技术分析。目的是帮助蓝队理解这一新兴威胁，并深入了解我们对该威胁的归因。第 1 部分将深入研究 DodgeBox 加载程序，重点介绍其独特特征和与 StealthVector 的相似之处，而第 2 部分将深入探讨 MoonWalk 后门的复杂性。

关键要点

- APT41 是一个总部位于某国的民族国家威胁行为者，因其在东南亚国家的活动而闻名，最近有人发现它部署了 StealthVector 的高级升级版本。我们将这个新变种命名为 DodgeBox。
- DodgeBox 采用了多种规避技术，例如调用堆栈欺骗、DLL 侧载、DLL 挖空和环境护栏。这些技术协同工作，可显著降低被安全防御检测到的可能性。
- 在分析 DodgeBox 时，我们发现它与 APT41 使用的著名 StealthVector 加载程序有显著相似之处。这些相似之处，加上对 DLL 侧载的独特利用以及从目标国家获取遥测数据，使我们有信心将此新加载程序归因于 APT41 / Earth Baku。

技术分析

攻击链

APT41 使用 DLL 侧载作为执行 DodgeBox 的手段。他们利用由 Sandboxie 签名的合法可执行文件 (taskhost.exe) 侧载恶意 DLL (sbiedll.dll)。这个恶意 DLL DodgeBox 充当加载器，负责从加密的 DAT 文件 (sbiedll.dat) 解密第二阶段有效负载。解密的有效负载 MoonWalk 充当后门，滥用 Google Drive 进行命令和控制 (C2) 通信。下图从高层次说明了攻击链。

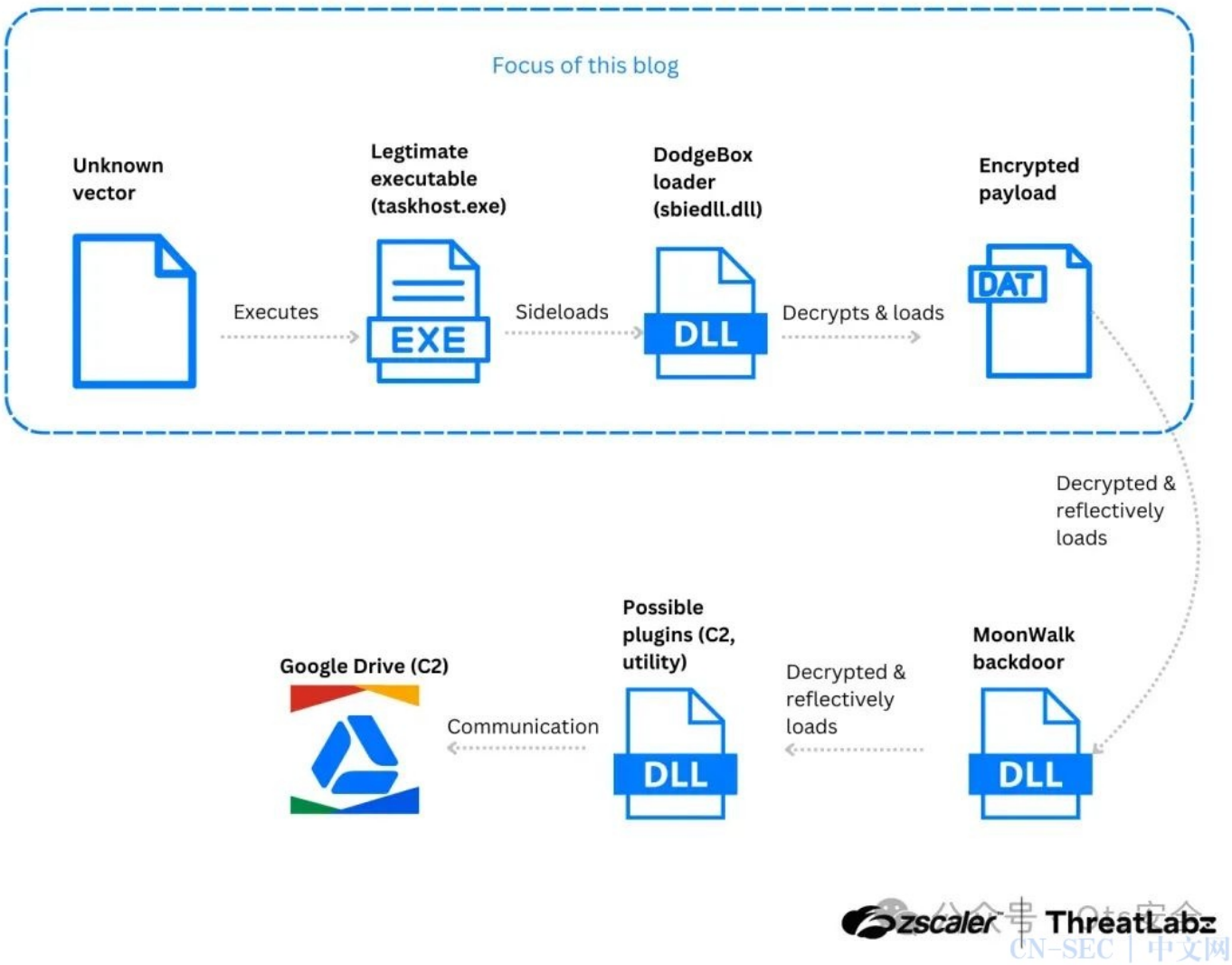

图 1：用于部署 DodgeBox 加载程序和 MoonWalk 后门的攻击链。

DodgeBox 分析

DodgeBox 是一款用 C 语言编写的反射式 DLL 加载程序，其概念与 StealthVector 相似，但在实现方面进行了重大改进。它提供多种功能，包括解密和加载嵌入式 DLL、进行环境检查和绑定以及执行清理程序。DodgeBox 与其他恶意软件的不同之处在于其独特的算法和技术。

在我们的威胁搜寻活动中，我们发现了两个 DodgeBox 样本，它们被设计为通过签名的合法可执行文件进行侧载。其中一个可执行文件由 Sandboxie（SandboxieWUAU.exe）开发，另一个由 AhnLab 开发。DLL 中的所有导出都指向一个主要调用恶意软件主函数的函数，如下所示：

```
1.  void SbieDll_Hook()
    {
     if ( dwExportCalled )
     {
       Sleep(0xFFFFFFFF);
     }
```

```
 else
 {
   hSbieDll_ = hSbieDll;
   dwExportCalled = 1;
   MalwareMain();
 }
}
```

MalwareMain实现 DodgeBox 的主要功能，可分为三个主要阶段：

1.解密DodgeBox的配置

DodgeBox 采用 AES 密码反馈 (AES-CFB) 模式来加密其配置。AES-CFB 将 AES 从分组密码转换为流密码，允许加密不同长度的数据而无需填充。加密配置嵌入在 二进制部分中。为了确保配置的完整性，DodgeBox 使用硬编码的 MD5 哈希来验证嵌入的 AES 密钥和加密配置。作为参考，可以在本博客的附录部分找到 DodgeBox 解密配置的示例。我们将在以下部分中.data使用变量引用此示例配置 。Config

2. 执行护栏和环境设置

解密其配置后，DodgeBox 会执行几项环境检查以确保它在预定目标上运行。

执行护栏：参数检查

DodgeBox 首先验证进程是否使用正确的参数启动。它会扫描参数argv中定义的特定字符串Config.szArgFlag。接下来，它会计算后续参数的 MD5 哈希值，并将其与中指定的哈希值进行比较Config.rgbArgFlagValueMD5。在本例中，DodgeBox 期望参数包含--type driver。如果此验证检查失败，则终止进程。

环境设置：API解析

之后，DodgeBox 继续解析用于额外环境检查和设置的多个 API。值得注意的是，DodgeBox 对 DLL 和函数名称使用加盐 FNV1a 哈希。这种加盐哈希机制可帮助 DodgeBox 逃避通常搜索 DLL 或函数名称哈希的静态检测。此外，它还允许不同的 DodgeBox 样本对相同的 DLL 和函数使用不同的值，同时保持核心哈希算法的完整性。

下面的代码显示 DodgeBox 调用其 ResolveImport函数来解析的地址 LdrLoadDll，并填充其导入表。

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

在ResolveImport函数内部，DodgeBox 使用 FNV1a 哈希函数进行两步处理。首先，它对输入字符串（代表 DLL 或函数名称）进行哈希处理。然后，它单独对盐值进行哈希处理。此两步哈希处理过程相当于对输入字符串和盐的连接进行哈希处理。以下伪代码表示加盐哈希的实现：

```
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 );
```

附录中包含了生成加盐哈希的 Python 脚本。

除了加盐哈希实现之外，DodgeBox 还在其 ResolveImport 函数中加入了另一个值得注意的功能。此函数既接受 DLL 名称作为字符串，也接受其哈希值作为参数。这种冗余似乎是为了提供灵活性而设计的，允许 DodgeBox 处理尚未加载目标 DLL 的情况。在这种情况下，DodgeBox 使用 LoadLibraryW提供的字符串调用该函数来动态加载 DLL。

此外，DodgeBox 可以有效处理转发导出和按序数导出。它会在必要时使用 `ntdll!LdrLoadDll`和 ntdll!LdrGetProcedureAddressEx来解析导出函数的地址。这种方法确保 DodgeBox 可以成功解析和利用所需的函数，无论使用哪种导出方法。

环境设置：DLL 解除挂钩

一旦 DodgeBox 解析了必要的函数，它就会继续扫描并解除从 System32 目录加载的 DLL。此过程包括遍历 .pdata每个 DLL 的部分，检索每个函数的起始和结束地址，并计算每个函数字节的 FNV1a 哈希值。然后，DodgeBox 计算存储在磁盘上的相同函数字节的相应哈希值。如果两个哈希值不同，则可以检测到潜在的篡改，DodgeBox 将用磁盘中的原始版本替换内存中的函数。

对于每个已成功扫描的 DLL，DodgeBox LDR_DATA_TABLE_ENTRY通过清除 ReservedFlags6字段并将高位设置为 1 来标记相应的 DLL。此标记使 DodgeBox 避免两次扫描同一个 DLL。

环境设置：禁用 CFG

随后，DodgeBox 检查操作系统是否为 Windows 8 或更新版本。如果是，代码将通过使用参数进行调用来验证控制流保护 (CFG) 是否已启用 GetProcessMitigationPolicy。ProcessControlFlowGuardPolicy如果 CFG 处于活动状态，恶意软件将尝试将其禁用。

为了实现这一点，DodgeBox 通过搜索特定的字节序列来 定位LdrpHandleInvalidUserCallTarget函数 。一旦找到，恶意软件就会用一条简单的指令修补此函数：`ntdll.dlljmprax`

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

CFG 验证间接调用目标的有效性。当 CFG 检查失败时， LdrpHandleInvalidUserCallTarget 将被调用，通常会引发中断。此时，rax 寄存器包含无效的目标地址。补丁修改了此行为，直接调用目标而不是引发中断，从而 绕过 CFG 保护。

此外，DodgeBox msvcrt!_guard_check_icall_fptr用 替换了 msvcrt!_DebugMallocator<int>::~_DebugMallocator<int>，该函数返回 0 来禁用 执行的 CFG 检查 msvcrt。

执行护栏：MAC、计算机名称和用户名检查

最后，DodgeBox 执行一系列检查以验证其是否配置为在当前计算机上运行。恶意软件将计算机的 MAC 地址与 进行比较 Config.rgbTargetMac，并将计算机名称与 进行比较 Config.wszTargetComputerName。根据 Config.fDoCheckIsSystem标志，DodgeBox 检查其是否以特权运行 SYSTEM。如果任何这些检查失败，恶意软件将终止执行。

3. 有效载荷解密和环境密钥

有效载荷解密

在最后阶段，DodgeBox 开始对 MoonWalk 有效载荷 DAT 文件进行解密。代码首先检查文件的前四个字节。如果这些字节非零，则表示 DAT 文件已与特定机器绑定（如下所述）。但是，如果 DAT 文件不是特定于机器的，DodgeBox 将继续使用 AES-CFB 加密解密文件，利用存储在配置文件中的密钥参数。在 ThreatLabz 分析的样本中，这个解密的 DAT 文件对应于一个 DLL，即 MoonWalk 后门。

有效载荷的环境键控

解密过程结束后，DodgeBox 采取了额外的步骤，将有效载荷锁定到当前机器。它通过使用 Config.rgbAESKeyForDatFile 密钥重新加密有效载荷来实现这一点。但是，在这种特定情况下，该过程偏离了配置文件的 IV（初始化向量）。相反，它利用当前机器 GUID 的 MD5 哈希作为 AES IV。这种方法保证了解密的 DAT 文件无法在任何其他机器上解密，从而增强了有效载荷的安全性。

使用 DLL Hollowing 加载有效载荷

接下来，DodgeBox 使用 DLL 挖空技术反射性地加载有效载荷。从高层次上讲，该过程从 System32目录中随机选择一个主机 DLL 开始，确保它不在阻止列表中（附录部分中提供了 DLL 阻止列表）并且具有足够大的 .text 部分。然后在 处创建此 DLL 的副本 。DodgeBox 通过禁用 NX 标志、删除和 部分并使用简单的 修补其入口点来 C:WindowsMicrosoft.NETassemblyGAC_MSILSystem.Data.Tracev4.0_4.0.0.0__<random bytes from pcrt4!UuidCreate><name of chosen DLL>.dll修改此副本 。`relocTLSreturn 1`

在准备好要注入的主机 DLL 之后，DodgeBox 会将 PE 标头以及与有效载荷 DLL 的 、 和 目录相对应的结构清零 。然后 IMAGE_DATA_DIRECTORY将 修改后的有效载荷 DLL 插入到先前选择的主机 DLL 中。使用和 API将修改后的主机 DLL 的结果副本加载到内存中 。 `importrelocdebugNtCreateSectionNtMapViewOfSection`

成功加载 DLL 后，DodgeBox 会更新进程环境块 (PEB) 中的相关条目，以反映新加载的 DLL。为了进一步隐藏其活动，DodgeBox 会用其原始内容覆盖主机 DLL 的修改副本，使其看起来像磁盘上合法的、已签名的 DLL。最后，恶意软件调用有效载荷 DLL 的入口点。

有趣的是，如果负责 DLL 挖空的函数无法加载有效载荷 DLL，DodgeBox 会采用回退机制。此回退函数使用 NtAllocateVirtualMemory 和 实现传统形式的反射 DLL 加载 NtProtectVirtualMemory。

此时，有效载荷DLL已成功加载，并通过调用第一个导出函数将控制权转移到有效载荷DLL。

调用堆栈欺骗

DodgeBox 在上述三个阶段中还采用了最后一种技术： **调用堆栈欺骗**。调用堆栈欺骗用于掩盖 API 调用的来源，使 EDR 和防病毒系统更难检测到恶意活动。通过操纵调用堆栈，DodgeBox 使 API 调用看起来好像来自受信任的二进制文件而不是恶意软件本身。这可以防止安全解决方案获取有关 API 调用真实来源的上下文信息。

DodgeBox 在调用更可能被监视的 Windows API 时特别利用了调用堆栈欺骗。例如，它直接调用 RtlInitUnicodeString 仅执行字符串操作的 Windows API，而不是使用堆栈欺骗。

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

然而，在调用已知被恶意软件滥用的 API时，会使用调用堆栈欺骗，如下所示：NtAllocateVirtualMemory,

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

上述技术可以在下图中观察到。在第一张图中，我们可以看到 explorer.exe 调用 CreateFileW 函数时的典型调用堆栈。系统监视工具 SysMon 有效地遍历了调用堆栈，使我们能够了解此 API 调用背后的目的并检查该过程中涉及的模块和函数。

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

explorer.exe图 2：调用 的堆栈跟踪的正常示例 CreateFileW。

相比之下，下图显示了当 DodgeBox 使用堆栈欺骗调用 CreateFileW 函数时 SysMon 记录的调用堆栈。请注意，没有迹象表明触发 API 调用的 DodgeBox 模块。相反，所涉及的模块似乎都是合法的 Windows 模块。

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

图 3：使用堆栈欺骗技术调用的DodgeBox 堆栈跟踪。CreateFileW

关于这项技术有一篇非常出色的 文章，因此我们仅重点介绍 DodgeBox 特有的一些实现细节：

- 当 CallFunction调用时，DodgeBox 使用 `jmp qword ptr [rbp+48h]`位于 .textK 部分内的随机小工具ernelBase。
- DodgeBox 分析该部分内的展开代码 .pdata来提取包含所选小工具的函数的展开大小。
- RtlUserThreadStart + 0x21 DodgeBox 获取和 的地址 BaseThreadInitThunk + 0x14，以及它们各自的展开大小。
- RtlUserThreadStart + 0x21DodgeBox 通过在正确的位置插入、 的地址和小工具的地址来设置堆栈，并利用检索到的展开大小。BaseThreadInitThunk + 0x14
- 随后，DodgeBox 继续插入适当的返回地址，[rbp+48h]并准备寄存器和堆栈，其中包含要传递给 API 的必要参数值。此准备可确保正确调用 API 并使用预期的参数。
- 最后，DodgeBox 执行一条 jmp 指令将控制流重定向到目标 API。

威胁归因

在本节中，我们概述了威胁归因过程中用作指标的不同策略、技术和程序 (TTP)。通过识别这些重叠的 TTP，我们将此活动归因于一个名为 APT41 的某国威胁行为者。我们对此归因的信心水平为中等。

滥用 DLL 侧载

DLL 侧载是与某国有关联的 APT 组织常用的一种技术。通常，这种方法涉及三个基本组件：经过签名的合法可执行 (EXE) 文件、恶意 DLL 文件和加密数据文件。虽然这里提到的 EXE 和 DLL 文件的具体组合尚未公开记录为与 APT41 有关，但这三个组件的存在可能表明与某国有关联的组织参与其中。

目标区域

对 VirusTotal 中提供的遥测数据进行分析后发现，泰国和台湾均已提交 DodgeBox 样本。这一观察结果与 APT41 之前在主要针对东南亚 (SEA) 地区用户的活动中使用 StealthVector 的情况相符。

此外，在监控用于 C2 通信的攻击者控制的 Google Drive 帐户时，发现了一份包含印度个人个人信息的电子表格。该电子表格可从其他来源公开获取，这表明威胁行为者可能利用它来识别潜在的其他目标。

DodgeBox 和 StealthVector 之间的相似之处

在分析 DodgeBox 的过程中，我们注意到它与 StealthVector 有许多相似之处。在本节中，我们将 2021 年和 2024 年上传到 VirusTotal 的 StealthVector 变种的代码与 DodgeBox 的代码进行比较。

校验和与配置解密的相似之处

StealthVector 和 DodgeBox 都会对其加密配置执行完整性检查。此验证过程包括两个基本步骤。首先，验证配置的硬编码大小，确保其与预期大小相匹配。其次，验证配置的哈希值以确保其完整性。成功完成这些检查后，恶意软件将继续解密配置。

StealthVector（2021年）

```
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);
```


图 4：StealthVector 使用 CRC32 哈希算法和 ChaCha20 算法进行解密（截图来自 TrendMicro）。

StealthVector 的旧变体使用 CRC32 哈希算法进行完整性检查，并使用 ChaCha20 解密配置。

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) )
```


图 5：StealthVector 使用 CRC32 哈希算法和 AES-CBC 算法进行解密。

StealthVector 的较新变体使用 CRC32 哈希算法和 AES-CBC 进行解密。

躲避箱

```
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;
```


图6：DodgeBox使用MD5哈希算法和AES-CFB算法进行解密。

DodgeBox 使用 MD5 哈希算法进行完整性检查，并使用 AES-CFB 进行配置解密。

解密配置格式的相似之处

这些相似之处涵盖了各个方面，例如防护栏、有效载荷文件名、大小和偏移量以及加密秘密。原始 StealthVector 和 DodgeBox 配置也都包含其加密有效载荷的校验和。

StealthVector（2021年）

```
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
```


图 7：从 StealthVector 2021 变体中提取的配置。

从 StealthVector 2021 变体中提取的配置与 StealthVector 2024 变体和 DodgeBox 有几个相似之处。

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
```


图 8：从 StealthVector 2024 变体中提取的配置。

从 StealthVector 2024 变体中提取的配置与 StealthVector 2021 变体和 DodgeBox 有几个相似之处。

躲避箱

```
--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
```


图 9：从 DodgeBox 提取的配置。

从 DodgeBox 中提取的配置与 StealthVector 的 2024 和 2021 变体有几个相似之处。

环境键控的相似之处

StealthVector 和 DodgeBox 都通过解密然后重新加密捆绑的有效载荷来执行环境密钥控制。

StealthVector（2021年）

TrendMicro 的报告并未记录 StealthVector 使用环境密钥。

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);
```


图 10：StealthVector 的 2024 变体执行环境密钥，使用滚动 XOR 对计算机名称进行操作。

更新后的 StealthVector 版本使用有效载荷的前四个字节 (rgbDecryptedData_In_Out) 来检查有效载荷是否已输入密钥。如果有效载荷之前未输入密钥，StealthVector 将继续使用目标计算机的计算机名称对其进行输入密钥。

此密钥过程涉及滚动 XOR 运算以对有效载荷进行编码，然后使用 AES 重新加密。在所分析的样本中，StealthVector 将有效载荷的前四个字节设置为 0x90909090，以指示有效载荷已成功密钥化。

躲避箱

```
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 *)0xFFFFFFFFFFFFD8ELL
  && 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);
}
```


图 11：DodgeBox 使用一种称为环境密钥的技术，其中它使用机器 GUID 的哈希值作为 AES 初始化向量 (IV)。DodgeBox

使用有效载荷的前四个字节 (pFileData) 来确定它是否已被密钥化。如果有效载荷之前没有被密钥化，DodgeBox 将使用其配置中的 AES IV 解密有效载荷。然后，DodgeBox 继续使用目标机器的 MachineGUID 的 MD5 哈希值作为新的 AES IV 对其进行重新加密。

在给定的示例中，DodgeBox 将有效载荷的前四个字节设置为 0x000000ED。此非零值表示有效载荷确实已被密钥化，并且应使用新的 AES IV 进行解密。

禁用 CFG 的相似之处

这三个样本在修补 CFG 的方法上表现出了非常相似的逻辑。这种相似性延伸到使用相同的字节模式来定位 LdrpHandleInvalidUserCallTarget 函数，以及在此函数中应用相同的补丁。

StealthVector（2021年）

```
(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);
        }
      }
    }
  }
}
```


图 12：StealthVector 2021 变体禁用 CFG 的代码（TrendMicro 的屏幕截图）。

从 StealthVector 2021 变体中提取的代码展示了 CFG 的禁用，与这三个样本有着惊人的相似性。

StealthVector（2024）

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


图 13：StealthVector 2024 变体的代码，禁用 CFG。

从 StealthVector 2024 变体中提取的代码展示了 CFG 的禁用，与这三个样本有着惊人的相似性。

躲避箱

```
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
```


图 14：来自 DodgeBox 的禁用 CFG 的代码。

从 DodgeBox 中提取的代码展示了 CFG 的禁用，与这三个样本有着惊人的相似性。

DLL Hollowing 使用的相似之处

所有三个样本都展示了通过 DLL 挖空加载捆绑有效载荷的能力。值得注意的是，2024 版 StealthVector 与 DodgeBox 共享相同的黑名单 DLL 列表。

待续

DodgeBox 是一种新发现的恶意软件加载程序，它采用多种技术来逃避静态和行为检测。根据已知的 TTP、潜在目标国家以及与 StealthVector 的相似性，我们相当有信心地将此活动归因于某国国家威胁行为者 APT41。在本系列第 1 部分中，我们分析了 DodgeBox 的技术细节及其与 StealthVector 的相似性。在第 2 部分中，我们将分析由 DodgeBox 投放的 MoonWalk 后门。

Zscaler 覆盖范围

Zscaler 的多层云安全平台检测到与 DodgeBox 相关的指标，这些指标具有以下威胁名称

- Win64.Payload.DodgeBox
- Win64.Backdoor.Moonwalk
- Win32.Backdoor.APT41
- Win64.Backdoor.APT41

攻击指标 (IOC)

MD5	Filename	Description
0d068b6d0523f069d1ada59c12891c4a	`Music.zip`	ZIP archive containing DodgeBox samples.

MD5	Filename	Description
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.
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

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

原文始发于微信公众号（Ots安全）：DodgeBox：深入探究 APT41 的最新武器库 | 第 1 部分