By Philip C. Okoh

Edited By Elizabeth Laub

Background

CryptBot is an Info-Stealer malware that has been making its rounds this year, 2022, distributed via cracked software and sketchy websites. Its capabilities range from stealing browser credentials, cookies, and cryptocurrency wallets, and enumerating system information sent to its C2 Server.

As reported by Mandiant, the group UNC3512 is currently using the bot to steal sensitive information with capabilities of taking screenshots, cookie and password stealing, and crypto wallet money stealing. Mandiant also claims that this bot has also been tied to installing additional botnets; **DANABOT** Mandiant Trending Threats .

This analysis aims to confirm previous findings, report tactics and techniques that were formerly undocumented and contribute this new information to the cybersecurity industry.

Stage 1 – af0403b7c12d7b7fa9c487eb4a6e68705e9247abf7bc542f77168bd4ed3408fb

Anti-Debugging

```
int __cdecl sub_401225(int a1)
{
  int v1; // edi
  int v2; // esi

  v1 = a1;
  if ( a1 <= dword_66D000 || NtCurrentPeb()->BeingDebugged == 1 )
  {
    MEMORY[0] = 0;
    return 0;
  }
  else
  {
    v2 = dword_66D000;
    if ( *(unsigned __int8 *)(mw_api_resolution(
                                NtCurrentPeb()->Ldr->InMemoryOrderModuleList.Flink->Flink->Flink[2].Fl
                                -1676131285 - dword_66D000)
                            + 6)
       + v2
       - 1 == v2 + 116 )
      return sub_4010D9(a1);
    return v1;
  }
```

Figure 1 - Anti Debugging PEB

Using the PEB structure to determine if the sample is being debugged. It causes an exception if it is. This can be bypassed by using the x64dbg feature **Hide Debugger**.

API Hashing

Obfuscation through API hashing. From my analysis, I found that the hashing algorithm was just an offshoot of an already discovered hashing algorithm. It differs by adding 10 to the end to make automatic hashing lookup slower. It's possible that as CryptBot evolves, it will produce different algorithms by just adding additional values to the hash result.

```
int v14; // [esp+10h] [ebp-Ch]
DWORD NumberOfNames; // [esp+14h] [ebp-8h]
char *AddressOfNames_cpy; // [esp+18h] [ebp-4h]

v4 = 0;
v5 = (_IMAGE_EXPORT_DIRECTORY *)((char *)a1 + *(_DWORD *)((char *)&a1[1].e_res2[8] + a1->e_lfanew));
AddressOfNames = (char *)a1 + v5->AddressOfNames;
AddressOfFunctions = (char *)a1 + v5->AddressOfFunctions;
AddressOfNameOrdinals = (char *)a1 + v5->AddressOfNameOrdinals;
AddressOfNames_cpy = AddressOfNames;
NumberOfNames = v5->NumberOfNames;
if ( !NumberOfNames )
  return 0;
v14 = a2 + dword_66D000;
while ( 1 )                                     // HASH_GENERATION
{
  v9 = (char *)a1 + *(_DWORD *)&AddressOfNames[4 * v4];
  v10 = 0;
  while ( 1 )
  {
    LOBYTE(v12) = *v9;
    if ( !*v9 )
      break;
    v11 = __ROR4__(v10, 13);
    v12 = (char)v12;
    if ( (char)v12 >= 97 )
      v12 = (char)v12 - 32;
    v10 = v12 + v11;
    ++v9;
  }
  if ( v10 == v14 )
    break;
  AddressOfNames = AddressOfNames_cpy;
  if ( ++v4 >= NumberOfNames )
    return 0;
}
return (int)a1 + *(_DWORD *)&AddressOfFunctions[4 * *(unsigned __int16 *)&AddressOfNameOrdinals[2 * v4]];
}
```

Figure 2 - API Hashing Algorithm

```python
rol = lambda val, r_bits, max_bits: \
    (val << r_bits%max_bits) & (2**max_bits-1) | \
    ((val & (2**max_bits-1)) >> (max_bits-(r_bits%max_bits)))

ror = lambda val, r_bits, max_bits: \
    ((val & (2**max_bits-1)) >> r_bits%max_bits) | \
    (val << (max_bits-(r_bits%max_bits)) & (2**max_bits-1))


'''
ORIGINAL REVERSING
'''
def hash(function_name):

    GENERATED_HASH = 0
    for character in function_name:

        segment_1 = ror(GENERATED_HASH, 13, 32)
        segment_2 = ord(character)
        if segment_2 >= 97:
            segment_2 -= 32
        GENERATED_HASH = segment_1 + segment_2
    return GENERATED_HASH
```

Figure 3 - Reverse Python API Hashing Algorithm

VirtualAlloc, resolved from API hashing, allocated space for the encrypted data, all of which is XOR encrypted. The two buffers produced consist of stage 2 shellcode and the final stage CryptBot executable. For the rest of the analysis, I will be focusing on the final stage executable.

Final Stage Executable –
f158d132733c7bc9f696a2b626f1324e868030513c12dae8d1273c7c841ad899

API Hashing
This stage relies heavily on API hashing to resolve most of its essential APIs, such as InternetOpenA, RegQueryValueA, and RegOpenKeyA. The hashing algorithm was not included in HashDB and is a new incorporation that I have named cryptbot_ror13_add_10.

```
1  rol = lambda val, r_bits, max_bits: \
 2     (val << r_bits%max_bits) & (2**max_bits-1) | \
 3     ((val & (2**max_bits-1)) >> (max_bits-(r_bits%max_bits)))
 4
 5 ror = lambda val, r_bits, max_bits: \
 6     ((val & (2**max_bits-1)) >> r_bits%max_bits) | \
 7     (val << (max_bits-(r_bits%max_bits)) & (2**max_bits-1))
 8
 9 def hash(data):
10
11
12     GENERATED_HASH = 0
13     for character in data:
14
15         segment_1 = ror(GENERATED_HASH, 13, 32) & 0xffffffff
16         segment_2 = character
17         if segment_2 >= 97:
18             segment_2 -= 32
19         GENERATED_HASH = segment_1 + segment_2
20     return GENERATED_HASH + 10
```

String Encryption
In the .rdata section there isa long list of encrypted strings, in the following format:

- First Byte - Data Length
- Next 5 Bytes - XOR Key
- Rest of Bytes - Encrypted Data

The contents of these strings are path strings for browsers, wallets, crypto wallet browser extensions and IDs. It's tedious to decrypt these manually, so I created an IDA Script to handle the decryption.

```
#IDA String Decryption Algorithm

#MAIN FUNCTION TO BE USED
def string_decryption_ida_v2(start, end):


    while start < end:

        encrypted_string_size = get_item_size(start)
        string_decryption_ida(start)
        start += encrypted_string_size


#HELPER FUNCTION
def replace_data(ea, new_string, offset):
    ea_start = ea
    for s in new_string:
        patch_byte(ea, ord(s))
        ea+=1
    while ea < ea_start+offset:
        patch_byte(ea, 0)
        ea+=1
    create_strlit(ea_start, idc.BADADDR)

## HELP FUNCTION
def string_decryption_ida(ea):

    item_size = get_item_size(ea)
    original_ea = ea
    data_size = get_bytes(ea, 1)
    data_size = int.from_bytes(data_size, byteorder='big')

    ea = ea+1
    xor_key = get_bytes(ea, 5)
    ea = ea+5

    output = []

    counter = 0

    while counter < data_size:

        encrypted_byte = get_bytes(ea, 1)
        output.append(chr(encrypted_byte[0] ^ xor_key[counter % 5]))
        counter+=1
```

Figure 4 - IDA String Decryption Algorithm

Figure 5 - Decrypted SQL Queries

Decrypting the strings also revealed SQL queries for credentials and cookie stealing.

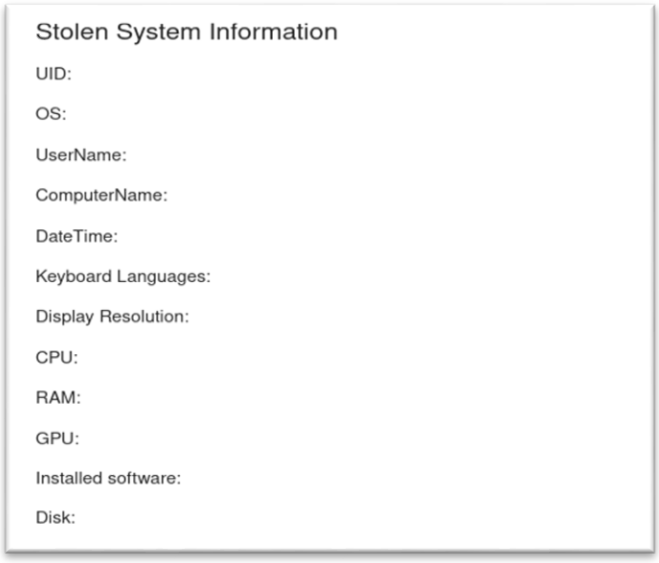

Figure 6 - Stolen Statistics

They also showed capabilities to steal system statistics and capabilities

Targeted Crypto Wallets, Password Managers and Authentication :

NAME	ID
GUARDA	hpglfhgfnhbgpjdenjgmdgoeiappafln
COIN98	aeachknmefphepccionboohckonoeemg
MATH	afbcbjpbpfadlkmhmclhkeeodmamcflc
TRONLINK	ibnejdfjmmkpcnlpebklmnkoeoihofec
KEPLR	dmkamcknogkgcdfhhbddcghachkejeap
TEZOS	ookjlbkiijinhpmnjffcofjonbfbgaoc
ATOMIC	fhilaheimglignddkjgofkcbgekhenbh
YOROI	ffnbelfdoeiohenkjibnmadjiehjhajb
JAXX	cjelfplplebdjjenllpjcblmjkfcffne
EOS_AUTHENTICATOR	oeljdldpnmdbchonielidgobddfffflal
GAUTH_AUTHENTICATOR	ilgcnhelpchnceeipipijaljkblbcobl
TREZOR_PASSWORD_MANAGER	imloifkgjagghnncjkhggdhalmcnfklk
TRUST	egjidjbpglichdcondbcbdnbeeppgdp
METAMASK	ejbalbakoplchlghecdalmeeeajnimhm
MATH WALLET	dfeccadlilpndjjohbjdblepmjeahlmm
YOROI	akoiaibnepcedcplijmiamnaigbepmcb
JAXX WALLET	dmdimapfghaakeibppbfeokhgoikeoci

Contacted URLs – As the creation of this report both URLs are dead

- dixuip12.top
- luedil01.top

Conclusion

My work confirms previous tactics and techniques documented about the CryptBot malware. It also provides insight into their hashing algorithms and how their implementation slows automatic static analysis. CryptBot, while a standard bot stealer, appears to be very effective because of its prevalence.