Bumblebee Loader

research.openanalysis.net/bumblebee/malware/loader/unpacking/2022/05/12/bumblebee_loader.html

OALABS Research May 12, 2022

Overview

According to Google's Threat Analysis Group...

> The loader can be recognized by its use of a unique user-agent “bumblebee” which both variants share. The malware, hence dubbed BUMBLEBEE.

This loader has been observed downloading payloads such as cobalt strike and is often delivered itself via an ISO file. The sample we are strating with today is an ISO.

References

Sample

0d740a348362171814cb314a48d763e336407904a36fa278eaf390c5743ec33b

Triage

The ISO contains two files `desk.dll` and `New Folder.Lnk`. We can right click `properties` on the lnk file to take a look at its command. The lnk file is used to launch the dll with the following command.

```
C:\Windows\System32\rundll32.exe desk.dll,aCmHmjrptS
```

Unpacking

- load rundll32.exe in x64dbg and change the command line to pass `desk.dll,#1`
- enable break on dll load
- once `desk.dll` is loaded locate export we want to debug (`aCmHmjrptS` ord 1) and add a hardware breakpoint
- remove the break on dll load and run until the export is bp is hit
- we initially tried watching for allocated memory via `VirutalAllocEx` but didn't see anything interesting
- instead we eneabled break on exit and just ran the dll
- when the break on exit was hit we searched memory for the PE header DOS string and located a mapped PE
- we unmapped the PE to reveal the payload

Payload

Unpacked and unmapped payload `abaa83ab368cbd3bbdaf7dd844251da61a571974de9fd27f5dbaed945b7c38f6` available on malshare.

Build Artifacts

There is a build artifact that may be useful for hunting other samples.

```
Z:\hooker2\Common\md5.cpp
```

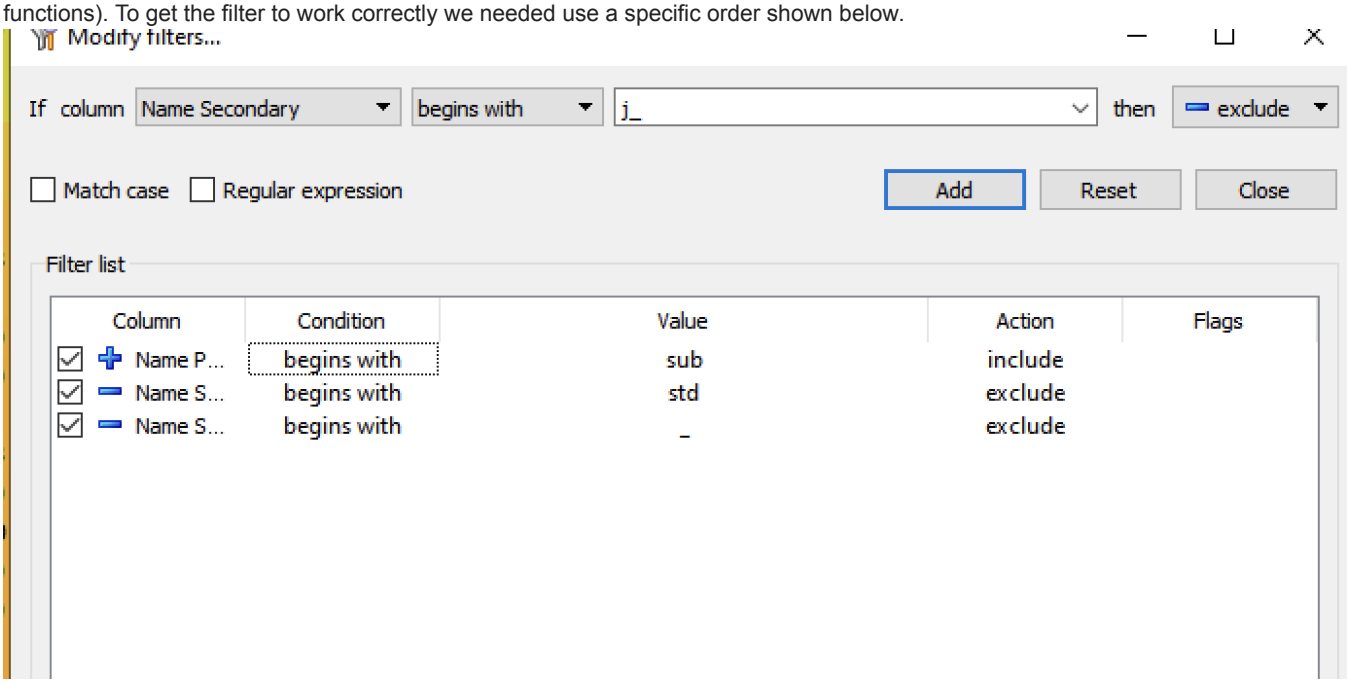
We searched for this on VirusTotal using the search term `https://www.virustotal.com/gui/search/content%253A%257B5a003a005c0068006f006f006b006500720032005c00%257D/files` and found other sample but nothing too interesting.

Anti-Analysis

There are many anti-analysis checks some of which have been directly copied from the open source project al-khaser. To get some free work we compiled al-khaser and created and IDB using a build version with symbols. We when used bindiff to match the al-khaser IDB with the payload. This allowed us to import all of the symbols from al-khaser.

IDA Filtering

While using BinDiff we ran into some issues with the IDA filter not working correcte (we were trying to filter out std and internal functions). To get the filter to work correctly we needed use a specific order shown below.



Config

Instead of a config the payload contains a series of encrypted strings in the `.data` section. These strings include the campaign name and a C2 list. The encryption is **RC4** and the key is a hard-coded plaintext string (also in the `.data` section). In our sample the key was `BLACK`.

Decrypted Config String

```
def unhex(hex_string):
    import binascii
    if type(hex_string) == str:
        return binascii.unhexlify(hex_string.encode('utf-8'))
    else:
        return binascii.unhexlify(hex_string)


def tohex(data):
    import binascii
    if type(data) == str:
        return binascii.hexlify(data.encode('utf-8'))
    else:
        return binascii.hexlify(data)

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)
```

```
data =
b'\x47\xCB\xD6\x45\x96\xAD\x39\x36\x82\x64\xA3\x68\xBB\x80\x5C\x8F\x4F\x86\x35\x73\xFD\xE9\x2E\x6D\x8C\x70\xB2\xE5\xEE\xD3\


key = 'BLACK'

out = rc4crypt(data, key)
print(out)

b'142.11.222.79:443,23.254.224.200:443,103.175.16.52:443,199.195.252.30:443\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0


data1 =
'47CED45EC69C1704B0568D5F82BA68BB7CAA0740D3DB1B59A24280D1C0E1F6215A962659A8F26249124408134E69C4616B46849D9BDDA8F6BC6D3D52F8


out = rc4crypt(unhex(data1), key)
print(out)

b'1105a\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0


data1 =
'42CBD06BA79C1704B0568D5F82BA68BB7CAA0740D3DB1B59A24280D1C0E1F6215A962659A8F26249124408134E69C4616B46849D9BDDA8F6BC6D3D52F8

out = rc4crypt(unhex(data1), key)
print(out)

b'444\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
```