Hunting the LockBit Gang's Exfiltration Infrastructures

yoroi.company/research/hunting-the-lockbit-gangs-exfiltration-infrastructures/

September 24, 2021

09/24/2021

Introduction

Nowadays ransomware operators have consolidated the double extortion practice by mastering data exfiltration techniques. From time to time, we observed many threat actors approach the data theft in diverse ways, some prefeed to rely on legit services and tools such as RClone, FTP sites, and some through VPN channels, but others also with customized tools.

Also, during the last months the LockBit gang (TH-276) decided to develop and evolve a custom tool specialized in data exfiltration and used as a peculiar element to distinguish their criminal brand. In fact, the StealBit 2.0 tool is part of the toolset the gang offers to their crooks to overcome the difficulties of massive data theft: an out-of-the-box tool ready to be used against the target company next to the LockBit 2.0 encryption tool.

From an intelligence perspective, understand the mechanisms and the infrastructure behind this tool is particularly valuable, especialy to early detect animminent ransomware impact. For this reason, Yoroi Malware ZLAB dissected a recent version of StealBit, tracking down the infrastructures abused by the infamous tool, configured there by the cyber criminals (Stealbit-Configuration-Decryptor available).

Technical Analysis

The initial sample we have chosen to start our investigation has the following static information:

Hash	3407f26b3d69f1dfce76782fee1256274cf92f744c65aa1ff2d3eaaaf61b0b1d
Threat	StealBit
Brief Description	Exfiltration utility adopted by lockbit gang during their cyber intrusions
Filesize	52.7 KB
Ssdeep	768:FXPkQ2CsnwhxvfhkO88yb6cvXbhb7vJawOuArU1o/xnmGP:YLqvZko9ybpvrtvJa/uArU+5nNP

Table 1: Static information about the sample

Analyzing the malicious component, we immediately noticed the lack of metadata in the PE fields. In fact, we obtained few data: the bitness, entry point, the compiler timestamp, and not much more than the DOS header. Something huge is missing.

md5	9B905A490A98CD8EDF2E4B09AC8676AB
sha1	63632224F977AAAA1C7D88BE65CF16878B4BEF56
sha256	3407F26B3D69F1DFCE76782FEE1256274CF92F744C65AA1FF2D3EAAAF61B0B1D
md5-without-overlay	n/a
sha1-without-overlay	n/a
sha256-without-overlay	n/a
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
first-bytes-text	M Z .. .. @ .. ..
file-size	53760 (bytes)
size-without-overlay	n/a
entropy	6.638
imphash	n/a
signature	n/a
entry-point	55 8B EC 83 E4 F8 64 A1 30 00 00 00 81 EC DC 00 00 00 F6 40 68 70 56 74 02 EB FE 51 8D 4C 24 48 E8
file-version	n/a
description	n/a
file-type	**executable**
cpu	**32-bit**
subsystem	GUI
compiler-stamp	0x6104F747 (Sat Jul 31 09:09:59 2021)

Figure 1: Static information about the sample

In fact, the "imphash" section is not available in the sample. Surprisingly, this is not an error of the tool. The import table of the sample is completely void, empty, no Windows API listed. At this point, we decided to deep inside the code to understand the internals of the sample.

Anti-Debug Techniques

Anyway, the lack of system API does not prevent malware developers from protecting their code. So, one of the first things the StealBit sample does just after the entry point is implementing a low-level anti-analysis technique.

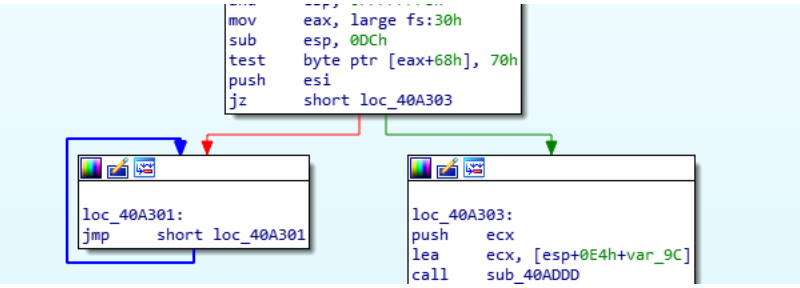

Figure 2: Simple Anti-Debug Routine

It is an anti-debug technique documented in many open source resources. The technique is based on the checking of specific values in Process Environment Block (PEB), a data structure in the Windows NT operating systems used to contain information about the execution of a specific process. One of the flags contained inside the PEB is "NtGlobalFlags": this value is accessed through the following opcodes.

```
mov eax, fs:[30h] ; Load the PEB data structure

mov eax, eax+68h ; Load the value of the “NtGlobalFlags” flag
```

Code snippet 1

If the value in the indicated flag is 0x70, it means that the process is debugged. In this case, the malware loops at the same instruction, otherwise it goes with its malicious activities.

The Runtime Loading of APIs and Libraries

As previously stated, the malware has an empty import address table, so it needs to load the required libraries to perform its malicious activities. Even when no IAT entry is present the operating system loads the three basic DLLs:

757E0000	kernelbase.dll	Sistema
75BC0000	kernel32.dll	Sistema
779A0000	ntdll.dll	Sistema

Figure 3: Automatic import of the base Windows libraries

To load all the rest of the system API needed to exfiltrate data, StealBit hides the native DLL names to import into stack strings. This means the name of the DLL to load is pushed into the running thread stack a char at a time, and then popped out to reconstruct the desired string, just like in the following piece of code:

```
push    0
push    'w'
mov     dword ptr [ecx], 7Eh
call    load_char_from_stack
push    1
push    's'
mov     [ecx+4], al
call    load_char_from_stack
push    2
push    '2'
mov     [ecx+5], al
call    load_char_from_stack
push    3
push    '_'
mov     [ecx+6], al
call    load_char_from_stack
push    4
push    '3'
mov     [ecx+7], al
call    load_char_from_stack
push    5
push    '2'
mov     [ecx+8], al
call    load_char_from_stack
push    6
push    '.'
mov     [ecx+9], al
call    load_char_from_stack
push    7
push    'd'
mov     [ecx+10], al
call    load_char_from_stack
push    8
push    'l'
mov     [ecx+11], al
call    load_char_from_stack
push    9
push    'l'
mov     [ecx+12], al
call    load_char_from_stack
mov     [ecx+13], al
```

Figure 4: Example of stack-strings loading

In this case, the reconstructed string is "ws2_32.dll", a native library for internet communication. Instead, the stack-strings of the other libraries loaded by StealBit are the following:

```
0018FEBC 00 4F 6C 65 33 32 2E 64 6C 6C 00 75 73 65 72 33 .Ole32.dll.user3
0018FECC 32 2E 64 6C 6C 00 63 6F 6D 63 74 6C 33 32 2E 64 2.dll.comctl32.d
0018FEDC 6C 6C 00 43 6F 6D 64 6C 67 33 32 2E 64 6C 6C 00 ll.Comdlg32.dll.
0018FEEC 7E 00 00 00 77 73 32 5F 33 32 2E 64 6C 6C 00 00 ~...ws2_32.dll..
0018FEFC 08 00 00 00 73 68 6C 77 61 70 69 2E 64 6C 6C 00 ....shlwapi.dll.
0018FF0C 18 00 00 00 73 68 65 6C 6C 33 32 2E 64 6C 6C 00 ....shell32.dll.
0018FF1C 42 00 00 00 67 64 69 70 6C 75 73 2E 64 6C 6C 00 B...gdiplus.dll.
0018FF2C 59 00 00 00 63 6F 6D 62 61 73 65 2E 64 6C 6C 00 Y...combase.dll.
0018FF3C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Figure 5: DLLs to load

Stack string obfuscation was extensively used across the sample, so we automatized the extraction process, and the results are reported in Appendix 1.

Data Exfiltration

When the Command and Control correctly responds to the malware, it starts its exfiltration routine, performed by using the HTTP method PUT and the implemented method is designed to be as fast as possible:

http.request.method==PUT

No.	Time	Source	Destination	Protocol	Length	Info	
2676	61.415423000	10	93.190.143.101	HTTP	587	PUT /E3	HTTP/1.1
2774	61.421820000	10	93.190.143.101	HTTP	813	PUT /B9	HTTP/1.1
2835	61.425149000	10	93.190.143.101	HTTP	813	PUT /B9	HTTP/1.1
2898	61.429015000	10	93.190.143.101	HTTP	813	PUT /B9	HTTP/1.1
2970	61.433631000	10	93.190.143.101	HTTP	813	PUT /B9	HTTP/1.1
3043	61.438491000	10	93.190.143.101	HTTP	813	PUT /B9	HTTP/1.1
3135	61.444403000	10	93.190.143.101	HTTP	813	PUT /B9	HTTP/1.1
3516	64.809675000	10	93.190.143.101	HTTP	813	PUT /B9	HTTP/1.1
3566	64.812468000	10	93.190.143.101	HTTP	724	PUT /16	HTTP/1.1
3726	64.821791000	10	93.190.143.101	HTTP	813	PUT /B9	HTTP/1.1
3760	64.823819000	10	93.190.143.101	HTTP	724	PUT /16	HTTP/1.1
3928	64.833791000	10	93.190.143.101	HTTP	813	PUT /B9	HTTP/1.1
4000	64.838281000	10	93.190.143.101	HTTP	724	PUT /16	HTTP/1.1

Figure 6: Piece of the Exfiltration C2 Communication

So, we decided to deepen the communication routine and we isolated all the fields of the request. The principal fields of the request are the following:

- PUT: HTTP PUT Method

- File Hash: indicates the file to put on the server
- HTTP classic headers
- DAV2 Constant Header: The body of the request starts with the DAV2 key
- The Config ID: (which we'll explain in the next paragraph)
- The complete file name of the exfiltrated file
- The content of the file in cleartext

An example of the construction of the malicious request is the following:

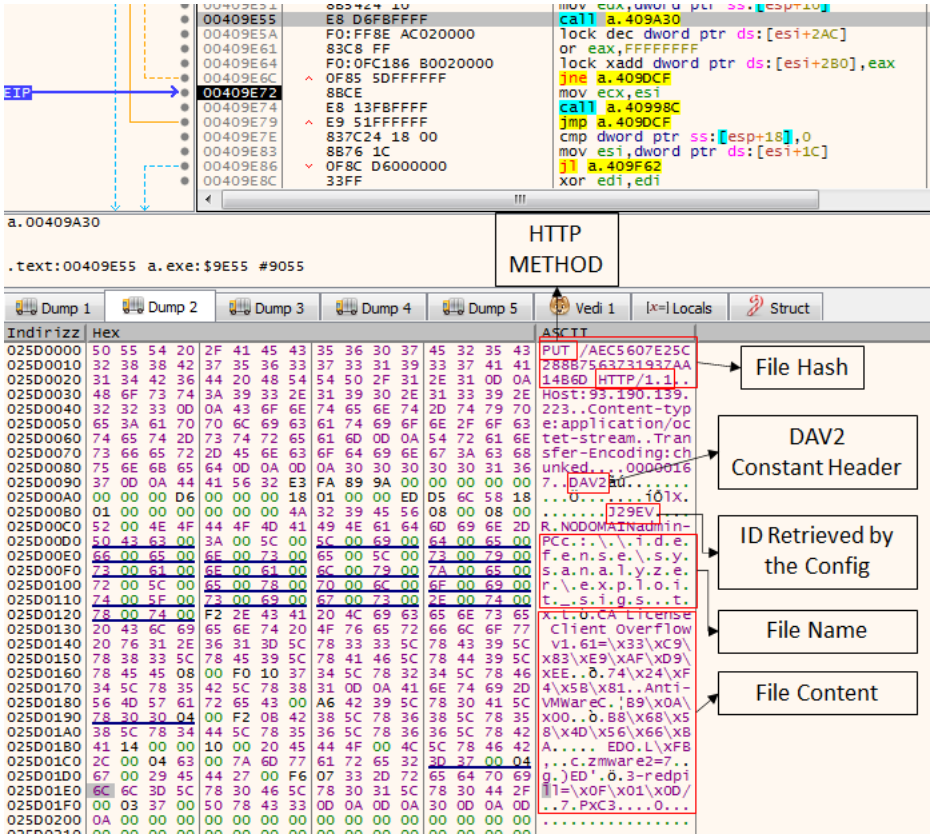

Figure 7: HTTP PUT request construction

Despite what LockBit gang advertises, their StealBit does not actually compress the file extracted by the system. In fact, the malware selectively uploads all the files reachable on the target machine except system files, registry hives, scripts and files matching specific extensions such as .cmd, .msi, .ocx, .cpl , .hta, .lnk, .exe , .dll, etc. .

The full list of file exclusions is available on Appendix 1.

Configuration Extraction

One of the most interesting points of malware was the static configuration protection mechanism in place. During the analysis we isolated the piece of code containing the routine adopted by the malicious developers to decrypt the StealBit configuration.

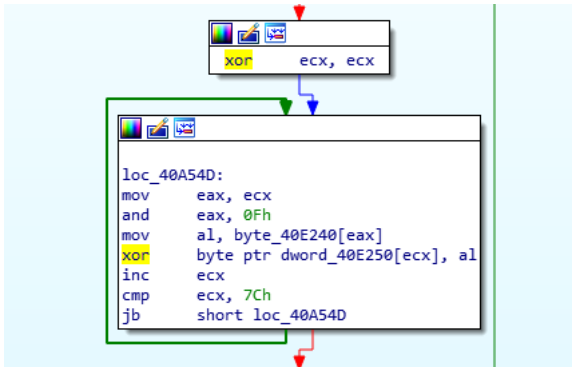

Figure 8: Configuration decoding routine

This piece of code contains a neat algorithm to decrypt the configuration of the StealBit sample. It reads a small 8-byte key to decode the byte-chuck starting from the offset 0x40E250 (see above). The loop ends when all 124 bytes are decoded. In the following picture we can see the before and the after of the configuration:

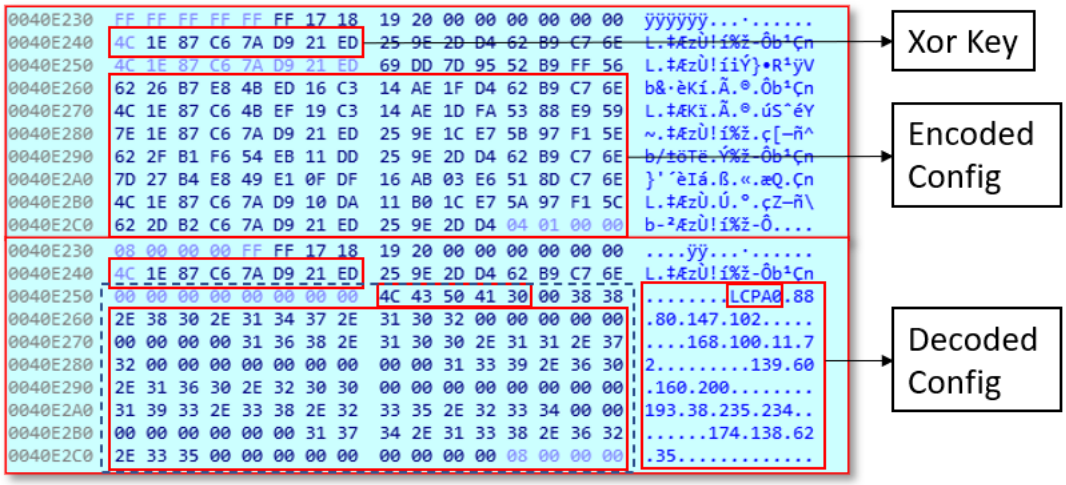

Figure 9: Before and after of the decoding process

The configuration chunk is composed of two parts: the first one is a 5-characters ID, probably identifying the victim or the current campaign, and the other chunk is a series of IP addresses to be contacted by the exfiltration tool. These remote IPs are the addresses of the infrastructure used by the threat actor to exfiltrate the data from the targeted companies.

Hunting the Samples

At this point, we created a Yara rule (see "Yara Rules" section) matching the configuration decrypting routine and automated the decoding of the static configurations of the StealBit samples in the wild using the Stealbit-Configuration-Decryptor. At the time of writing, we were spotted these samples:

Retrieved Hashes
2f18e61e3d9189f6ff5cc95252396bebaefe0d76596cc51cf0ade6a5156c6f66
4db7eeed852946803c16373a085c1bb5f79b60d2122d6fc9a2703714cdd9dac0
07a3dcb8d9b062fb480692fa33d12da05c21f544492cbaf9207956ac647ba9ae
3407f26b3d69f1dfce76782fee1256274cf92f744c65aa1ff2d3eaaaf61b0b1d
bd14872dd9fdead89fc074fdc5832caea4ceac02983ec41f814278130b3f943e
ced3de74196b2fac18e010d2e575335e2af320110d3fdaff09a33165edb43ca2
107d9fce05ff8296d0417a5a830d180cd46aa120ced8360df3ebfd15cb550636

Table 2: Retrieved hashes from Yara Hunting

These samples have a perfect code similarity with the original one and the only difference is properly the configuration chuck.

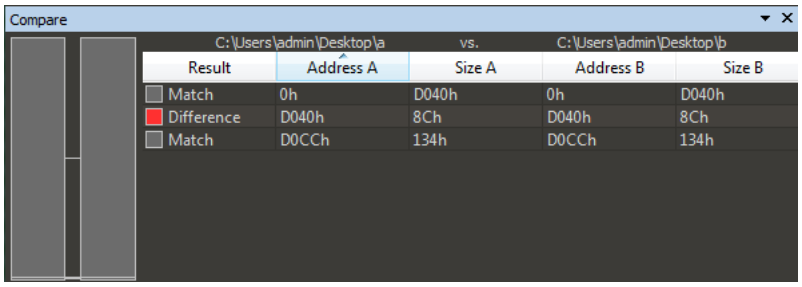

Figure 10: Binary Diff analysis of two samples

The result of the static configuration extraction from this first in-the-wild StealBit sample set is reported in the following table.

Hash	Compilation Time	ID	IPs

07a3dcb8d9b062fb480692fa33d12da05c21f544492cbaf9207956ac647ba9ae	2021-07-12 04:58:17	84AFC	93[.]190[.]143[[.]101 139[.]60[.]
107d9fce05ff8296d0417a5a830d180cd46aa120ced8360df3ebfd15cb550636	2021-07-31 07:09:59	J29EV	93[.]190[.]139[.]223 **168[.]100[.**
2f18e61e3d9189f6ff5cc95252396bebaefe0d76596cc51cf0ade6a5156c6f66	2021-07-31 07:09:59	D26VN	174[.]138[.]62[.]35 93[.]190[.]14
3407f26b3d69f1dfce76782fee1256274cf92f744c65aa1ff2d3eaaaf61b0b1d	2021-07-31 07:09:59	LCPA0	88[.]80[.]147[.]102 **168[.]100[.]**
4db7eeed852946803c16373a085c1bb5f79b60d2122d6fc9a2703714cdd9dac0	2021-07-12 04:58:17	4ATGY	139[.]60[.]160[.]200 193[.]38[.]2
bd14872dd9fdead89fc074fdc5832caea4ceac02983ec41f814278130b3f943e	2021-07-31 07:09:59	D26VN	174[.]138[.]62[.]35 93[.]190[.]14
ced3de74196b2fac18e010d2e575335e2af320110d3fdaff09a33165edb43ca2	2021-07-12 04:58:17	84AFC	93[.]190[.]143[.]101 139[.]60[.]1

Table 3: Automatic configuration extraction from the hunted samples

The Exfiltration Infrastructure

Once extracted the remote IP address hard-coded into the static configurations of the StealBit samples, we analyzed the exfiltration infrastructure from a threat intelligence point of view, tracking down past malicious activities related to those IPs. We noticed that some of them have been used in the past operation for other malicious purposes such as the distribution of mobile malware, or phishing attempts to banks etc., by actors unrelated to the LockBit gang and ransomware practice in general.

The connection between these different operations is still unclear and weak, in fact, different criminal organizations could have been accidentally chosen the same providers due to their potential lack of collaboration with western authorities, but also - at least in the 168.100.11[.72 case - the same remote address was used to conduct phishing operations in Italy and ransomware data exfiltration in adjacent same time spans.

IP	Count	Whois (NetName and Country)	Findings
139.60.160[.200	7	HOSTKEY-USA US	
168.100.11[.72	2	BLNETWORKS-01 US	Phishing to Italian banks between 12 – 24 Aug 2021
174.138.62[.35	4	DIGITALOCEAN-174-138-0-0 US	
185.215.113[.39	1	SC-ELITETEAM-20201113 SC	Distrubution of mobile banking malware in Feb21
193.162.143[.218	5	FirstByte RU	
193.38.235[.234	7	VDSINA-NET RU	RDP with machine name WIN-R84DEUE96RB and before WIN-5ODCFIGQRP3 in Aug21
45.227.255[.190	3	Okpay Investment Company PA-OICO-LACNIC	MongoDB scanning and exploitation in APR20
88.80.147[.102	1	BelCloud-net BG	
93.190.143[.101	4	WORLDSTREAM NL	Reported as Spam vector in 2020
93.190.139[.223	1	WORLDSTREAM NL	

Table 4: Information about the infrastructure

Conclusion

Data exfiltration tools are getting more popular in the cyber-criminal ecosystem. LockBit gang leveraged this kind of tools to distinguish from other ransomware operators and attract malicious affiliates in their criminal business, and today LockBit is one of the most active and violent threat groups operating the double extortion practice. Securing company data is nowadays a huge challenge and the proliferation of massive data theft tools like StealBit are an emergent threat.

Tracking down the adversary infrastructure is a relevant effort, by we believe it is necessary to help the security community to fight and pursue such criminals and protect the Yoroi's customers from data extortion threats.

Indicators of Compromise

- Hash:
  - 07a3dcb8d9b062fb480692fa33d12da05c21f544492cbaf9207956ac647ba9ae
  - 2f18e61e3d9189f6ff5cc95252396bebaefe0d76596cc51cf0ade6a5156c6f66
  - 3407f26b3d69f1dfce76782fee1256274cf92f744c65aa1ff2d3eaaaf61b0b1d
  - 4db7eeed852946803c16373a085c1bb5f79b60d2122d6fc9a2703714cdd9dac0
  - bd14872dd9fdead89fc074fdc5832caea4ceac02983ec41f814278130b3f943e
  - ced3de74196b2fac18e010d2e575335e2af320110d3fdaff09a33165edb43ca2
  - 107d9fce05ff8296d0417a5a830d180cd46aa120ced8360df3ebfd15cb550636
- Exfiltration:
  - 139.60.160[.200
  - 168.100.11[.72
  - 174.138.62[.35
  - 185.215.113[.39
  - 193.162.143[.218
  - 193.38.235[.234
  - 45.227.255[.190
  - 88.80.147[.102
  - 93.190.143[.101
  - 93.190.139[.223

Yara Rule

```
rule stealbit_decode {


meta:

                description = "Yara Rule for StealBit Configuration decryption"

                author = "Yoroi Malware Zlab"

                last_updated = "2021_09_01"

                tlp = "white"

                category = "informational"


strings:

$Offset = { ff 17 18 19 20 00 00 00 00 00 00 }

$decode_Conf = { 8b c1 83 e0 0f 8a 8? ?? ?? ?? ?? 30 8? ?? ?? ?? ?? 41 83 f9 7c  }


condition:

all of them

}
```

Suricata Rule

TLP:AMBER - accessible by Trusted Introducer CSIRT Network Members

Appendix 1 - Extraction of Stack Strings

Function
0x4036C0 0x4036C0 0x4036C0 0x4036C0 0x4036C0 0x4036C0 0x4036C0 0x4036C0 0x4036C0 0x4036C0 0x4036C0 0x4036C0 0x4

Appendix 2 - Decoder

Stealbit-Configuration-Decryptor available on Yoroi Malware ZLAB public GitHub repository.

This blop post was authored by Luigi Martire and Luca Mella of Yoroi Malware ZLAB