Analyzing Cobalt Strike for Fun and Profit

randhome.io/blog/2020/12/20/analyzing-cobalt-strike-for-fun-and-profit/

20 Dec 2020 · 10 minutes read

I am not sure what happened this year but it seems that Cobalt Strike is now the most used malware around the world, from APT41 to APT32, even the last SolarWinds supply chain attack involved Cobalt Strike. Without relaunching the heated debate on publishing offensive tools, this blog post intends to summarize what an analyst needs to know about Cobalt Strike to quickly identify and analyze it during incidents.

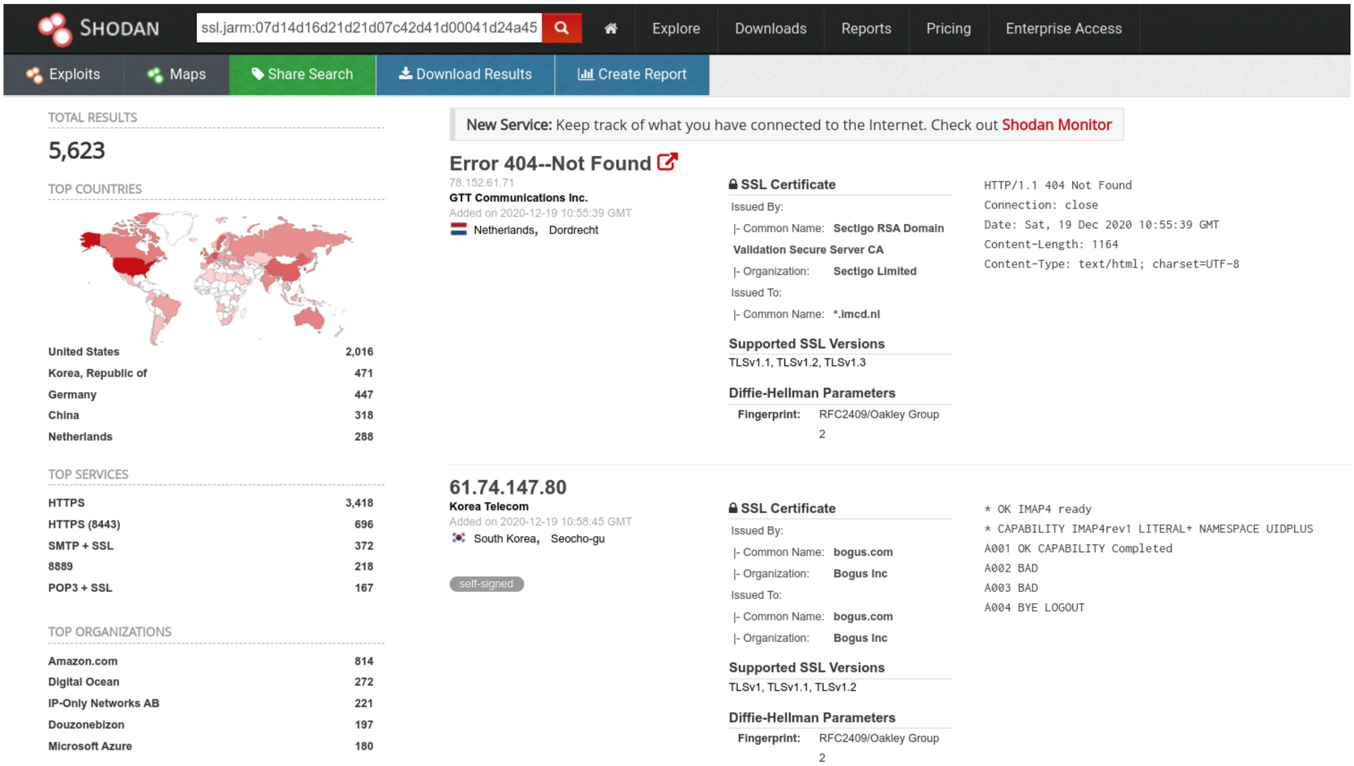
Finding Cobalt Strike Servers#

A few months ago, the Salesforce security team published a new active fingerprint tool called JARM. It is the active equivalent to JA3 they published last year. It generates a fingerprint based on the TLS configuration of a remote server, such as the TLS version or the TLS extensions, without considering the certificate. It is especially useful to identify custom web servers used by some tools, and Cobalt Strike is one of them.

Here is the Cobalt Strike JARM signature : `07d14d16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1`

JARM has already been added to Shodan, BinaryEdge, and SecurityTrails, Shodan recently added indexing on `ssl.jarm`, so it is easy to find Cobalt Strike servers in the wild.

Let's check Shodan with `ssl.jarm:07d14d16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1` :



Shodan has identified 5623 IP with this JARM fingerprint Cobalt Strike servers, mostly on Amazon and Digital Ocean. If we limit to port 443, we get 3423 IPs.

We can easily confirm that Cobalt Strike is still running on port 443 of the first IP using JARM:

```
$ python jarm.py 78.152.61.71
Domain: 78.152.61.71
Resolved IP: 78.152.61.71
JARM: 07d14d16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1
```

(This JARM signature is actually a signature of the Java Web server and specific to the JAVA 11 stack, so it includes other tools not related to Cobalt Strike (like Burp Suite) and does not include Cobalt Strike using different Java versions. Cobalt Strike recently wrote a blog post about this question.)

Getting a Cobalt Strike Payload#

Cobalt Strike uses a checksum of the url using an algorithm called checksum8 to serve the 32b or 64b version of the payload (in the same way as the metasploit server). The decompiled code of Cobalt Strike has been published several times on GitHub or elsewhere, it provides information on this checksum:

```
public static long checksum8(String text) {
    if (text.length() < 4) {
        return 0L;
    }
    text = text.replace("/", "");
    long sum = 0L;
    for (int x = 0; x < text.length(); x++) {
        sum += text.charAt(x);
    }

    return sum % 256L;
}

public static boolean isStager(String uri) {
    return (checksum8(uri) == 92L);
}

public static boolean isStagerX64(String uri) {
    return (checksum8(uri) == 93L && uri.matches("/[A-Za-z0-9]{4}"));
}
```

We can easily bruteforce the algorithm to find urls that match it in python:

```
from itertools import product
import string

def checksum8(strr):
    j = 0
    if len(strr) < 4:
        return 0
    strr = strr.replace("/", "")
    for c in strr:
        j += ord(c)
    return j % 256

chars = string.ascii_letters + string.digits
to_attempt = product(chars, repeat=4)
for attempt in to_attempt:
    word = ''.join(attempt)
    r = checksum8(word)
    if r == 92:
        print("{:30} - 32b checksum".format(word))
    elif r == 93:
        print("{:30} - 64b checksum".format(word))

$ python bf_checksum8.py
aaa9                           - 32b checksum
aab8                           - 32b checksum
aab9                           - 64b checksum
aac7                           - 32b checksum
aac8                           - 64b checksum
aad6                           - 32b checksum
aad7                           - 64b checksum
[...]
```

So `/aaa9` should return the 32 bits beacon (if available) and `/aab9` should return the 64 bits beacon (if available). Let’s test that on one of the Cobalt Strike servers from the Shodan list, `103.39.18.184` (AS136800 - ICIDC NETWORK - China) (one thing to know is that the Cobalt Strike server blocks unusual user agents).

```
$ wget --no-check-certificate https://103.39.18.184/aaa9
--2020-12-19 17:44:32--  https://103.39.18.184/aaa9
Connecting to 103.39.18.184:443... connected.
WARNING: cannot verify 103.39.18.184's certificate, issued by ‘CN=gmail.com,OU=Google Mail,O=Google GMail,L=Mountain
View,ST=CA,C=US’:
  Self-signed certificate encountered.
    WARNING: certificate common name ‘gmail.com’ doesn't match requested host name ‘103.39.18.184’.
HTTP request sent, awaiting response... 404 Not Found
2020-12-19 17:44:33 ERROR 404: Not Found.


$ wget --no-check-certificate --user-agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/70.0.3538.77 Safari/537.36" https://103.39.18.184/aaa9
--2020-12-19 17:44:52--  https://103.39.18.184/aaa9
Connecting to 103.39.18.184:443... connected.
WARNING: cannot verify 103.39.18.184's certificate, issued by ‘CN=gmail.com,OU=Google Mail,O=Google GMail,L=Mountain
View,ST=CA,C=US’:
  Self-signed certificate encountered.
    WARNING: certificate common name ‘gmail.com’ doesn't match requested host name ‘103.39.18.184’.
HTTP request sent, awaiting response... 200 OK
Length: 208980 (204K) [application/octet-stream]
Saving to: ‘aaa9’

aaa9                   100%[===========================>] 204.08K   655KB/s    in 0.3s

2020-12-19 17:44:53 (655 KB/s) - ‘aaa9’ saved [208980/208980]


$ wget --no-check-certificate --user-agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/70.0.3538.77 Safari/537.36" https://103.39.18.184/aab9
--2020-12-19 17:44:58--  https://103.39.18.184/aab9
Connecting to 103.39.18.184:443... connected.
WARNING: cannot verify 103.39.18.184's certificate, issued by ‘CN=gmail.com,OU=Google Mail,O=Google GMail,L=Mountain
View,ST=CA,C=US’:
  Self-signed certificate encountered.
    WARNING: certificate common name ‘gmail.com’ doesn't match requested host name ‘103.39.18.184’.
HTTP request sent, awaiting response... 200 OK
Length: 260679 (255K) [application/octet-stream]
Saving to: ‘aab9’

aab9                   100%[===========================>] 254.57K   872KB/s    in 0.3s

2020-12-19 17:44:59 (872 KB/s) - ‘aab9’ saved [260679/260679]
```

So this IP `103.39.18.184` gives us two Cobalt Strike beacons:

- 742a06efbebca717271b6beda1ff4a22f6f0be6acda9590ab32b38e1d5721140 aaa9 (32b)
- 04bf2657dedfc99235220f59d3e7284d9e2ef0a183cd90ee3514137481a27d6c aab9 (64b)

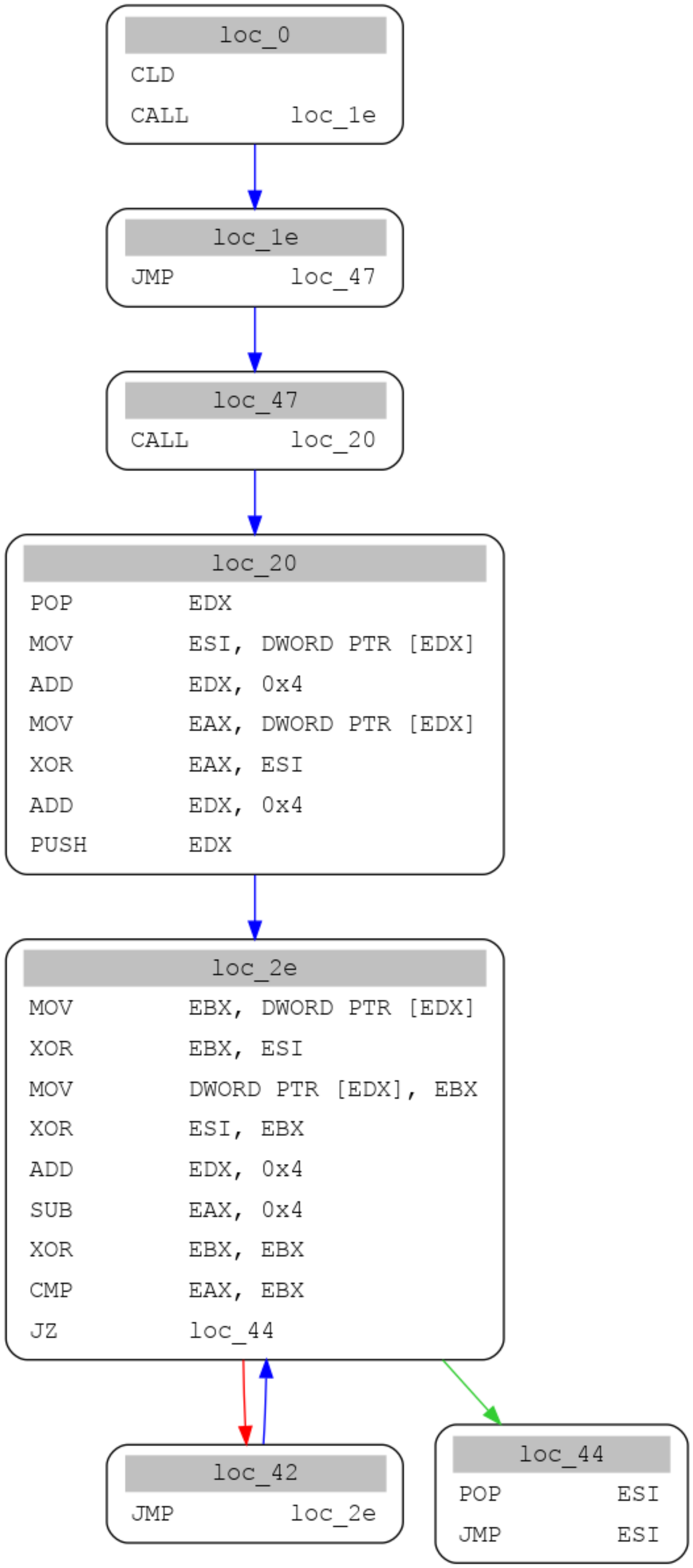
Decrypting Cobalt Strike Beacons#

The files returned by the server are actually not PE file:

```
$ file *
aaa9:     data
aab9:     data
```

During the execution of a Cobalt Strike exploit, it downloads this beacon and run it directly in memory. So this file is a blob of data containing a shellcode at the beginning that decodes and executes the beacon.

We can easily graph this shellcode with miasm:



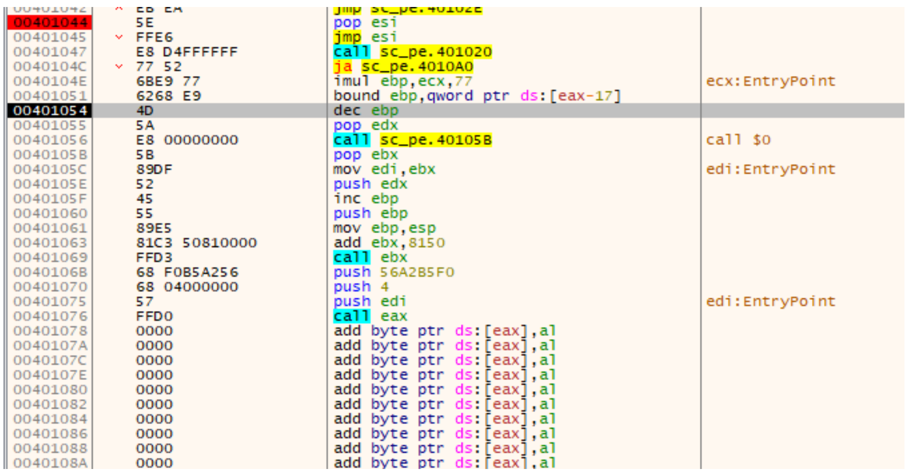
The first JMP goes to a call right before the encryption key and encrypted beacon. The call goes back to the shellcode and the next `POP EDX` gets the address of the key. The code in `loc_f` gets the key in EBX, the length of the payload in EAX, and store on the address of the final beacon on the stack. The loop in `loc_1d` goes through the beacon and xor it with the key.

We can easily reproduce it in python, the challenge is to find the base address. Here the `loc_47` is the address of the call just before the key, length, and encrypted payload, so the base address is 0x47 + 5 (the length of the call instruction). The base address changes with different payloads but it is possible to find it easily by searching the last call instruction.

```
import struct

def xor(a, b):
    return bytearray([a[0]^b[0], a[1]^b[1], a[2]^b[2], a[3]^b[3]])

with open("aaa9", "rb") as f:
    data = f.read()

ba = 0x4c
key = data[ba:ba+4]
print("Key : {}".format(key))
size = struct.unpack("I", xor(key, data[ba+4:ba+8]))[0]
print("Size : {}".format(size))

res = bytearray()
i = ba+8
while i < (len(data) - ba - 8):
    d = data[i:i+4]
    res += xor(d, key)
    key = d
    i += 4

with open("a.out", "wb+") as f:
    f.write(res)
```

We get a PE file : 3c9a06b2477694919b1c77d3288984cb793a47dd328ef39e15132cd0cfb593ab.

It is surprising to see a PE file directly there because it is directly executed by the last call. The trick is that the PE file is actually modified to be at the same time a valid PE file and executable directly (something Cobalt Strike calls raw stageless payload artifact). We see here the MZ header being executed:



The DOS header is modified to include valid instructions that jump to the address 0x8157 in the binary. This address is the address of the exported `_ReflectiveLoader@4` function, a function based on the ReflectiveDLLInjection software that is in charge of reproducing a simple PE loader to load and map import functions before calling the entry point.

(Note that this is only optional in Cobalt Strike, many Cobalt Strike payloads do not have a PE file format but the payload directly in a shellcode-like format).

Extracting the Configuration#

The Cobalt Strike configuration is encrypted within the payload, with a different key depending on the Cobalt Strike version, either 0x2E or 0x69. Once decoded, the configuration is stored in the format type-length-value :

- One short that represent the key of the data (the list can be found in the Cobalt Strike source code)
- Two short bytes representing the type of data (Int, Short, String, etc.) and its length
- The data itself

To find the start address of the configuration, we can look for the encoded value of the first key, which is 1 (DNS/SSL), 1 (Short), 2 (2 bytes), which is encoded to `ihihik` with the key 0x69 or `./././,` with the key 0x2E.

We can then decode the configuration of the beacon:

```
dns                           False
ssl                           True
port                          443
.sleeptime                    60000
.http-get.server.output
00000004000000010000017700000001000000fa0000000200000004000000020000001c00000002000000240000000200000012000000020000000400000002000

.jitter                       15
.maxdns                       255
publickey
30819f300d06092a864886f70d010101050003818d0030818902818100aef69a6fb8f21092c01a95cbdcac0f03f79738adecda36cffc6c5cf607943e72663865f8f6

.http-get.uri                 156.226.191.234,/_/scs/mail-static/_/js/,djiqowenlsakdj.com,/_/scs/mail-static/_/js/
.user-agent                   Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0; MALCJS)
.http-post.uri                /mail/u/0/
.http-get.client              OSID=Cookie
GAccept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
ui=d3244c4707ient
                hop=6928632     start=0
=Content-Type: application/x-www-form-urlencoded;charset=utf-8OSID=Cookie
.spawto
.post-ex.spawnto_x86          %windir%\syswow64\notepad.exe
.post-ex.spawnto_x64          %windir%\sysnative\notepad.exe
.pipename
.cryptoscheme                 0
.dns_idle                     134743044
.dns_sleep                    0
.http-get.verb                GET
.http-post.verb               POST
shouldChunkPosts              0
.watermark                    305419896
.stage.cleanup                0
CFGCaution                    0
host_header
cookieBeacon                  1
.proxy_type                   2
funk                          0
killdate                      0
text_section                  0
process-inject-start-rwx      64
process-inject-use-rwx        64
process-inject-min_alloc      0
process-inject-transform-x86
process-inject-transform-x64
process-inject-stub           a56c813864af878a4c10083ca1578e0a
process-inject-execute
process-inject-allocation-method 0
```

Putting Everything Together#

Now that we have decoded everything, it is quite easy to do the request, extract the beacon and the configuration directly. I have put all this in a script available on this github repository:

```
$ python scan.py https://103.39.18.184/
Checking https://103.39.18.184/
Unknown config command 55
Configuration of the x86 payload
dns                           False
ssl                           True
port                          443
.sleeptime                    60000
.http-get.server.output
00000004000000010000017700000001000000fa0000000200000004000000020000001c00000002000000240000000200000012000000020000000400000002000

.jitter                       15
.maxdns                       255
[SNIP]

x86_64: Payload not found
```

It is common to find the same configuration for many different samples because CS uses what they call Malleable C2 Profiles which are actually configurations for the CS beacons that can be shared easily through configuration files. For instance, Ocean Lotus uses a public profile mimicking Google Safebrowsing urls. This repository list profiles used by different APT or cybercrime groups.

One interesting value in the configuration is the watermark, which is a number generated from the license file. As it is unique to a customer, it can be used to pivot and link multiple CobaltStrike instances together (as it was done for Trickbot). As such, many cracked versions of CobaltStrike disable this watermark. This watermark is technically associated with the Cobalt Strike Customer id, so it should be possible to report this id to Cobalt Strike and identify the customer for people using paid licenses, but I have never heard anyone doing that (I guess few APT groups have a valid CS license).

Extracting Configuration of 1000 Cobalt Strike Servers#

Based on the same code, I have scanned the 3424 servers identified with JARM in Shodan, I have scanned them all using this script and found 520 serving Cobalt Strike beacons.

I have uploaded on GitHub a csv listing the IPs and configuration of these beacons, here is a short extract:

Host	GET URI
54.66.253.144	`54.66.253.144,/s/ref=nb_sb_noss_1/167-3294888-0262949/field-keywords=books`
103.243.183.250	`103.243.183.250,/search.js`
185.82.126.47	`185.82.126.47,/pixel`
94.156.174.121	`94.156.174.121,/watch`
194.36.191.118	`194.36.191.118,/visit.js`
23.106.160.198	`repshd.com,/us/ky/louisville/312-s-fourth-st.html,pinglis.com,/us/ky/louisville/312-s-fourth-st.html,stargut.com,/us/ky/louisville/312-s-fourth-st.html`
23.81.246.46	`contmetric.com,/s/ref=nb_sb_noss_1/167-3294888-0262949/field-keywords=books`
108.174.193.11	`qw.removerchangefile.monster,/media.html,as.removerchangefile.monster,/media.html,zx.removerchangefile`
213.217.0.218	`213.217.0.218,/visit.js`
213.252.247.31	`1nubjgrcfjhjhkjftdd.com,/s/ref=nb_sb_noss_1/167-3294888-0262949/field-keywords=books`

165 of them over 523 have no watermark, another watermark (305419896) is used by 160 IPs, so it is likely a default value. Then a few watermarks have more than 5 servers, such as 1580103814, 1873433027 or 16777216.

That’s All Folks#

I have uploaded all these scripts and yara rules for beacons on Github, feel free to DM me on Twitter or send me an email if you have any question.

Here are some other interesting resources on Cobalt Strike:

This blog post was mostly written while listening to Susumu Yokota.

Edit 1: adding link to Cobalt Strike JARM analysis (thanks @AZobec)

- malware
- threatintel