

Cobalt Strike: Decrypting Obfuscated Traffic – Part 4

By Didier Stevens

Published: 2021-11-17 · Archived: 2026-04-05 21:14:10 UTC

Encrypted Cobalt Strike C2 traffic can be obfuscated with malleable C2 data transforms. We show how to deobfuscate such traffic.

This series of blog posts describes different methods to decrypt Cobalt Strike traffic. In [part 1 of this series](#), we revealed private encryption keys found in rogue Cobalt Strike packages. In [part 2](#), we decrypted Cobalt Strike traffic starting with a private RSA key. And in [part 3](#), we explain how to decrypt Cobalt Strike traffic if you don't know the private RSA key but do have a process memory dump.

In the first 3 parts of this series, we have always looked at traffic that contains the unaltered, encrypted data: the data returned for a query and the data posted, was just the encrypted data.

This encrypted data can be transformed into traffic that looks more benign, using malleable C2 data transforms. In the example we will look at in this blog post, the encrypted data is hidden inside JavaScript code.

But how do we know if a beacon is using such instructions to obfuscate traffic, or not? This can be seen in the analysis results of the [latest version of tool 1768.py](#). Let's take a look at the configuration of the beacon we started with in [part 1](#):

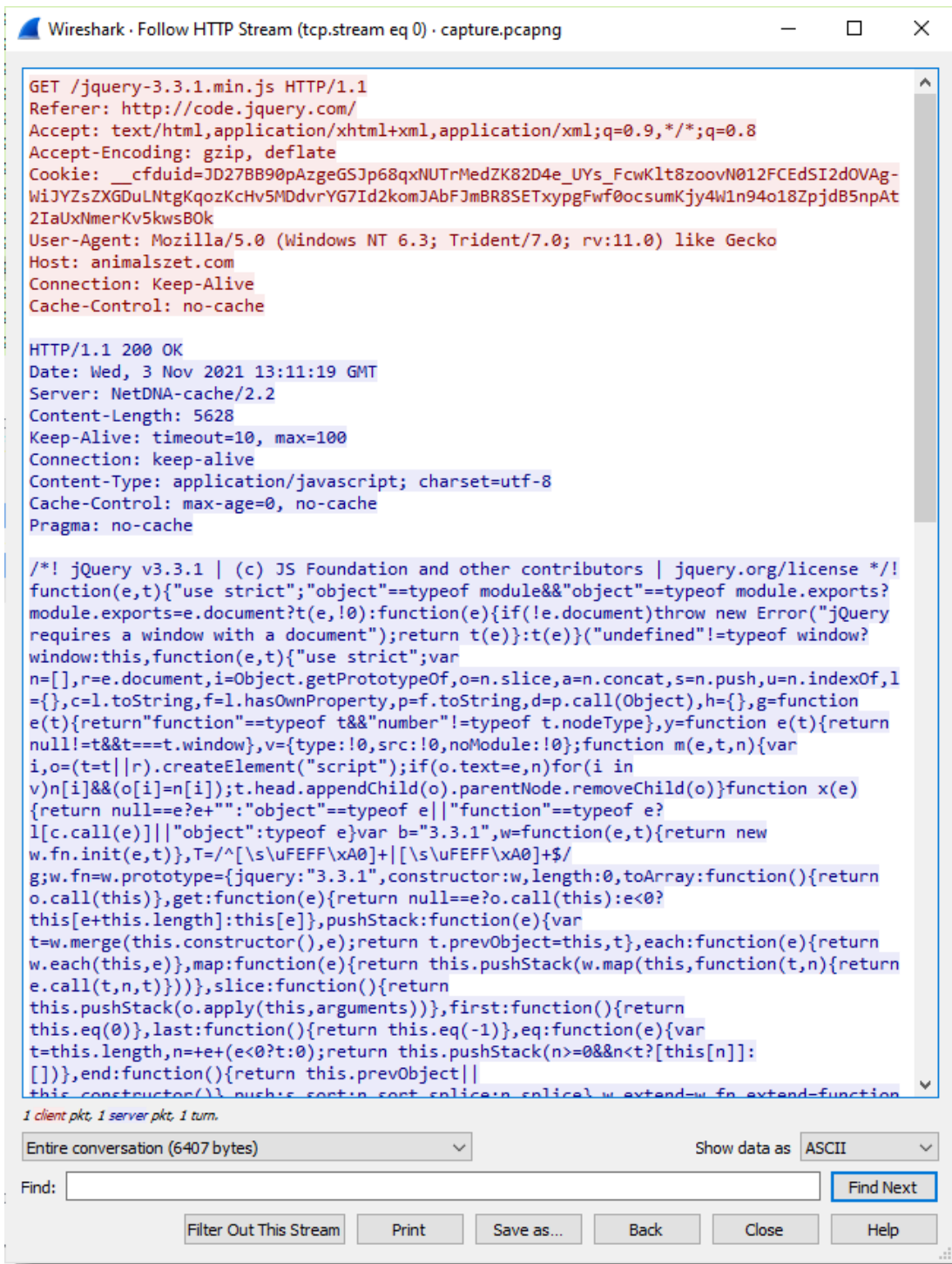


Figure 3: reply transformed with malleable C2 instructions to look like JavaScript code

What we see here, is a GET request by the beacon to the C2 (notice the Cookie with the encrypted metadata) and the reply by the C2. This reply looks like JavaScript code, because of the malleable C2 data transforms that have been used to make it look like JavaScript code.

We copy this reply over to CyberChef in its input field:

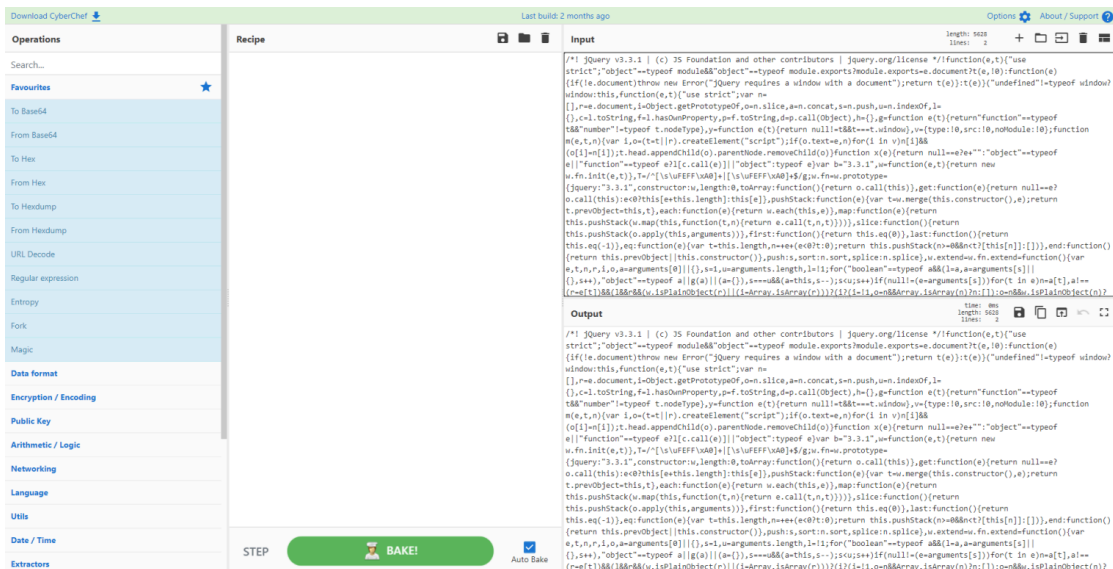


Figure 4: CyberChef with obfuscated input

The instructions we need to follow, to deobfuscate this reply, are listed in tool 1768.py's output:

```
0x000b Malleable_C2_Instructions 0x0003 0x0100
Transform Input: [7:Input,4,1:1522,2:84,2:3931,13,15]
Print
Remove 1522 bytes from end
Remove 84 bytes from begin
Remove 3931 bytes from begin
BASE64 URL
XOR with 4-byte random key
```

Figure 5: decoding instructions

So let's get started. First we need to remove 1522 bytes from the end of the reply. This can be done with a CyberChef drop bytes function and a negative length (negative length means dropping from the end):

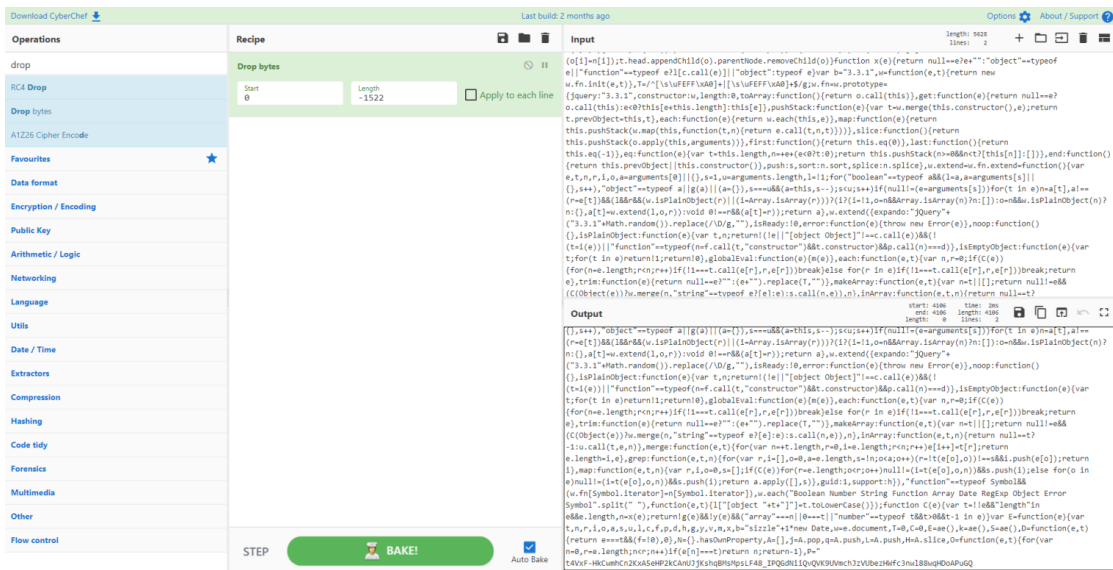


Figure 6: dropping 1522 bytes from the end

Then, we need to remove 84 bytes from the beginning of the reply:

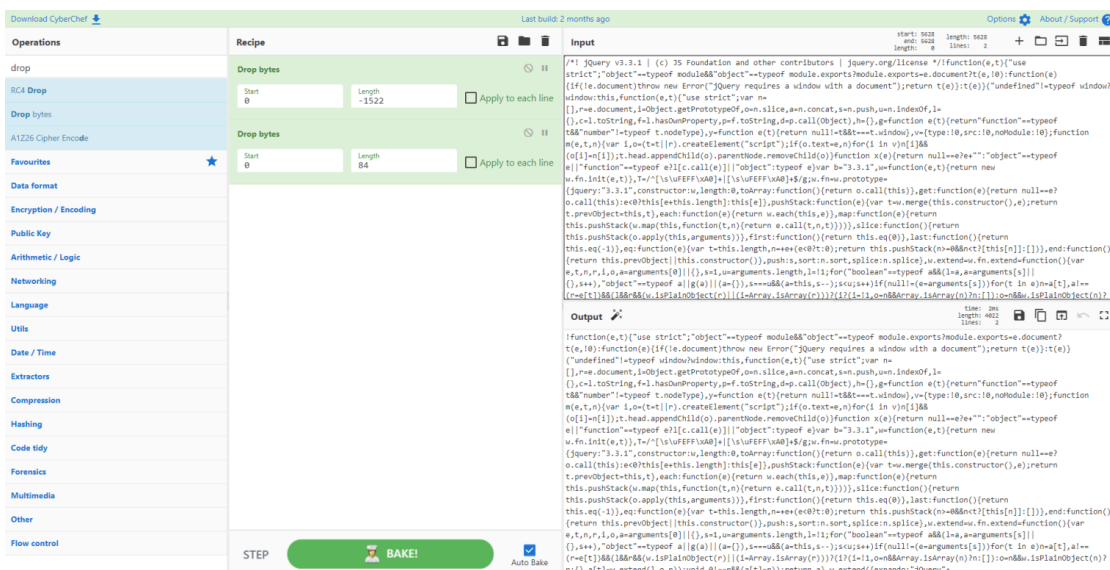


Figure 7: dropping 84 bytes from the beginning

And then also dropping 3931 bytes from the beginning:

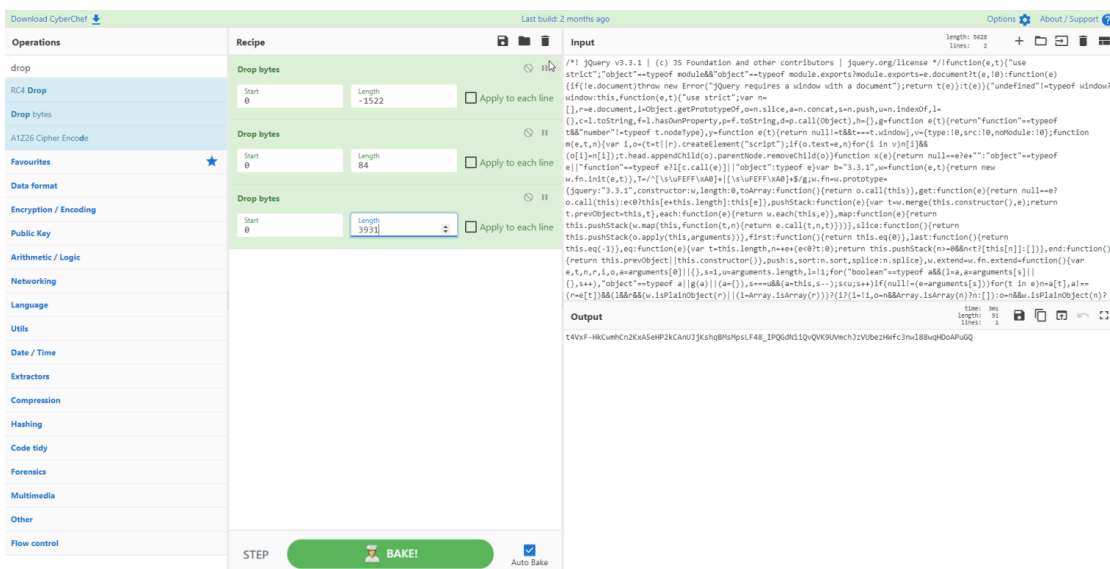


Figure 8: dropping 3931 bytes from the beginning

And now we end up with output that looks like BASE64 encoded data. Indeed, the next instruction is to apply a BASE64 decoding instructions (to be precise: BASE64 encoding for URLs):

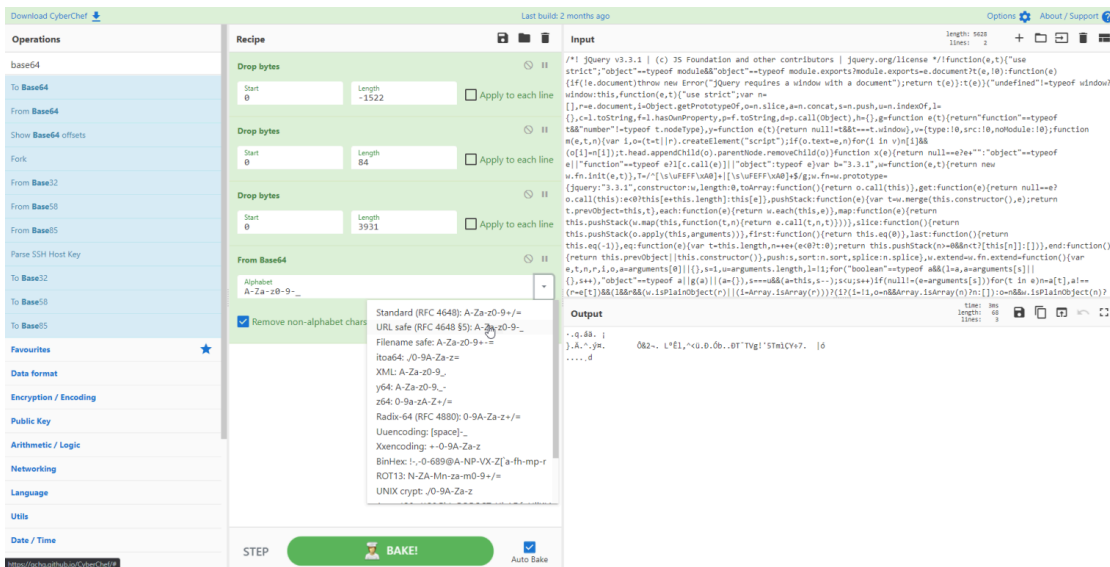


Figure 9: decoding BASE64/URL data

The next instruction is to XOR the data. To do that we need the XOR key. The malleable C2 instruction to XOR, uses a 4-byte long random key, that is prepended to the XORed data. So to recover this key, we convert the binary output to hexadecimal:

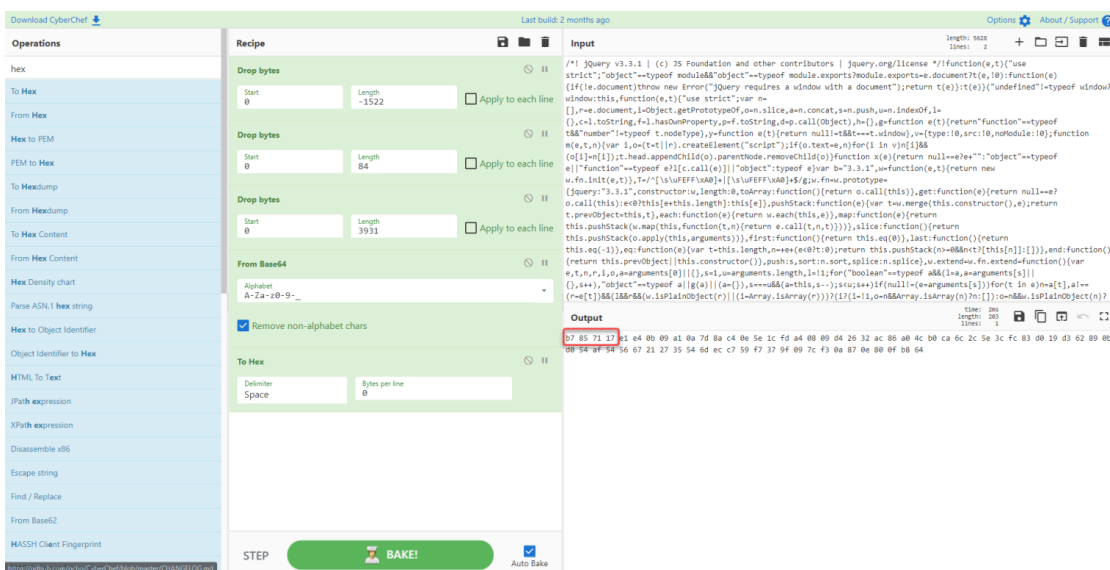


Figure 10: hexadecimal representation of the transformed data

The first 4 bytes are the XOR key: b7 85 71 17

We use that with CyberChef's XOR command:

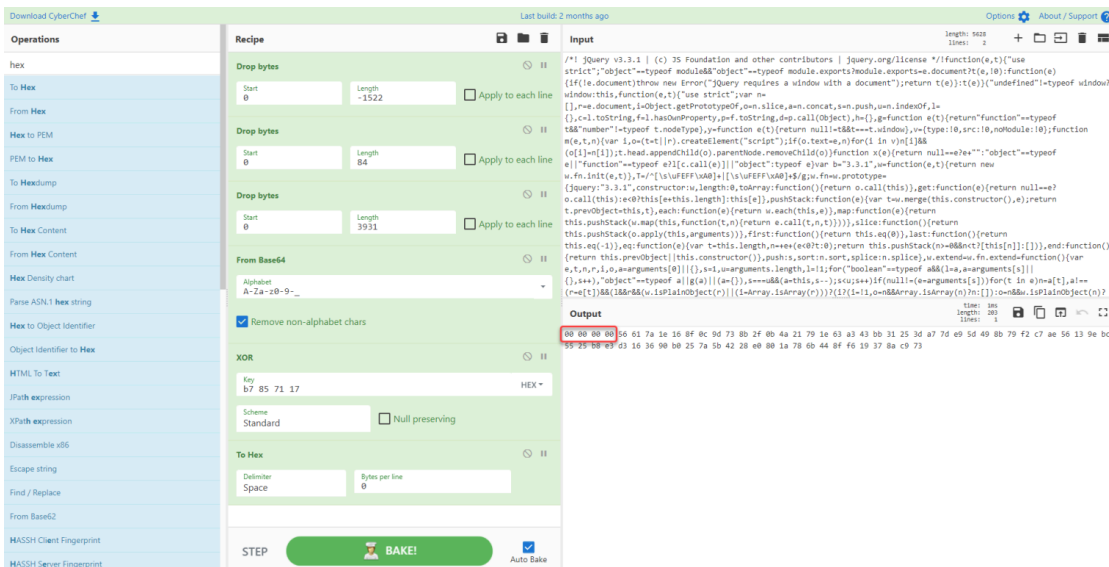


Figure 11: XORed data

Notice that the first 4 bytes are NULL bytes now: that is as expected, XORing bytes with themselves gives NULL bytes.

And finally, we drop these 4 NULL bytes:

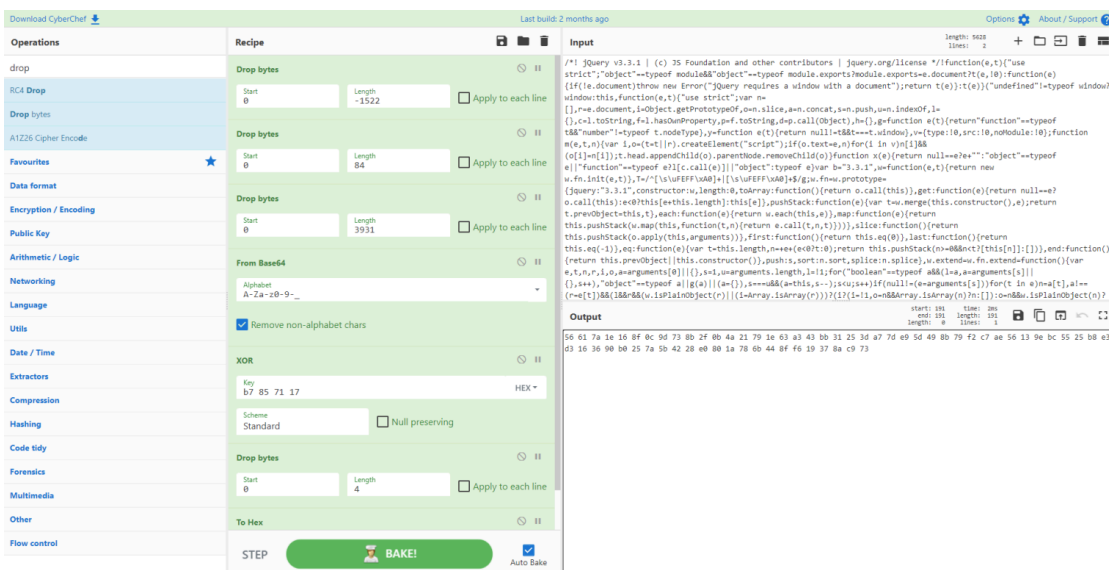


Figure 12: fully transformed data

What we end up with, is the encrypted data that contains the C2 commands to be executed by the beacon. This is the result of deobfuscating the data by following the malleable C2 data transform. Now we can proceed with the decryption using a process memory dump, just like we did in [part 3](#).


```
@NVISIO_Labs
Potential keys = 142
Segment 76a84000 size 1000
Potential keys = 1123
Segment 76aa8000 size 1000
Potential keys = 177
Segment 76c10000 size 1000
Potential keys = 2047
Segment 771c0000 size 1000
Potential keys = 1048
Segment 771c2000 size 1000
Potential keys = 1317
Segment 7efde000 size 1000
Potential keys = 316
Segment 7efdf000 size 1000
Potential keys = 396
Summary:
Keys found:
b'c7a56a38b5282ed9f2a56fe86f'
Files written:
rundll32.exe_211103_141122.dmp.2576000-3.bin
@NVISIO_Labs C:\Demo>
```

Figure 15: analyzing the process memory dump (screenshot 2)

The tool has indeed found a 13-byte long XOR key, and written the decoded section to disk as a file with extension .bin.

This file can now be used with cs-extract-key.py, it's exactly the same command as before, but with the decoded section in stead of the encoded .dmp file:

```
@NVISIO_Labs C:\Demo>cs-extract-key.py -t 56617a1e168f0c9d738b2f0b4a21791e63a343bb31253da77de95d498b79f2c7ae56139ebc5525b8e3d3163690b0257a5b4228e0801a786b448ff619378ac973 rundll32.exe_211103_141122.dmp.2576000-3.bin
File: rundll32.exe_211103_141122.dmp.2576000-3.bin
Searching for AES and HMAC keys
Searching after sha256\x00 string (0x6714)
AES key position: 0x0000cc60
AES Key: eeee0d3b24b0cc2c969b9483ccf66404
HMAC key position: 0x0000ff80
HMAC Key: dbe322c2eead03f1b9b67e52a5bad695
SHA256 raw key: dbe322c2eead03f1b9b67e52a5bad695:eeee0d3b24b0cc2c969b9483ccf66404
Searching for raw key
@NVISIO_Labs C:\Demo>
```

Figure 16: extracting keys from the decoded section

And now we have recovered the cryptographic keys.

Notice that in figure 16, the tool reports finding string sha256\x00, while in the first command (figure 13), this string is not found. The absence of this string is often a good indicator that the beacon uses a sleep mask, and that tool cs-analyze-processdump.py should be used prior to extracting the keys.

Now that we have the keys, we can decrypt the network traffic with tool [cs-parse-http-traffic.py](#):

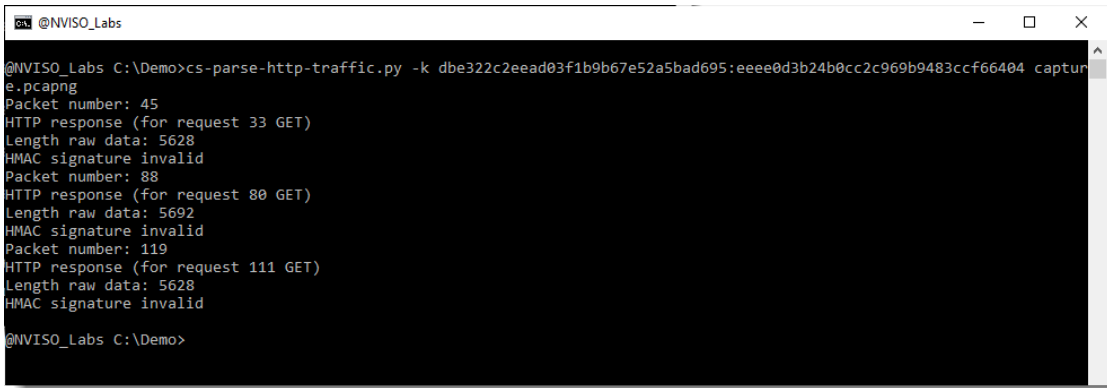


Figure 17: decrypting the traffic fails

This fails: the reason is the malleable C2 data transform. Tool cs-parse-http-traffic.py needs to know which instructions to apply to deobfuscate the traffic prior to decryption. Just like we did manually with CyberChef, tool cs-parse-http-traffic.py needs to do this automatically. This can be done with option -t.

Notice that the output of tool 1768.py contains a short-hand notation of the instructions to execute (between square brackets):

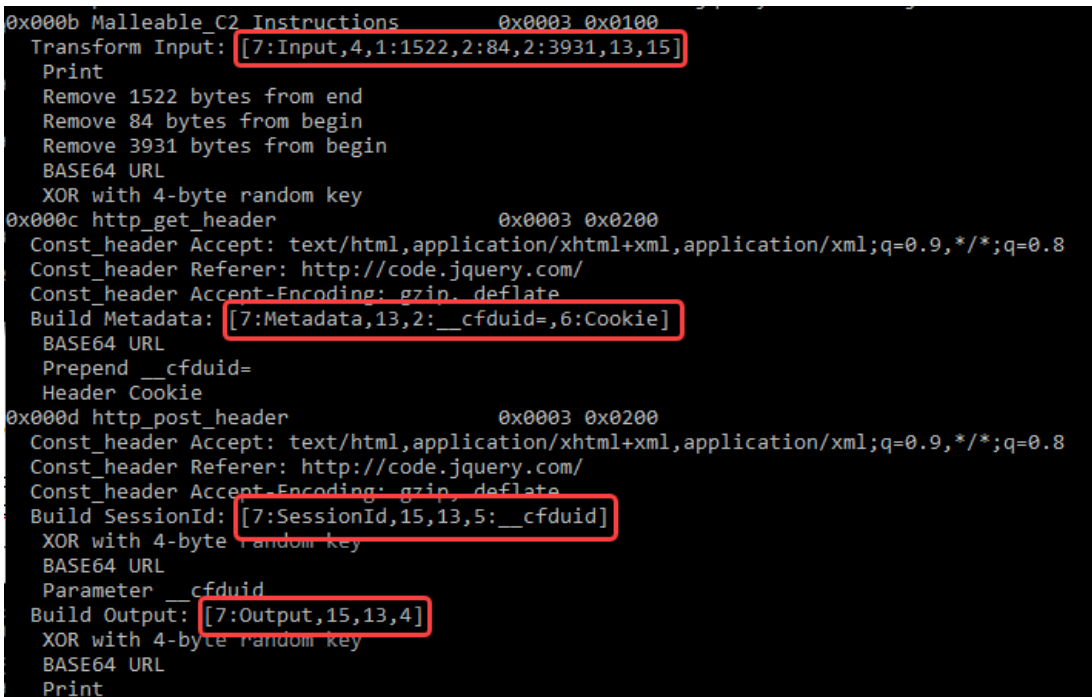


Figure 18: short-hand notations of malleable C2 instructions

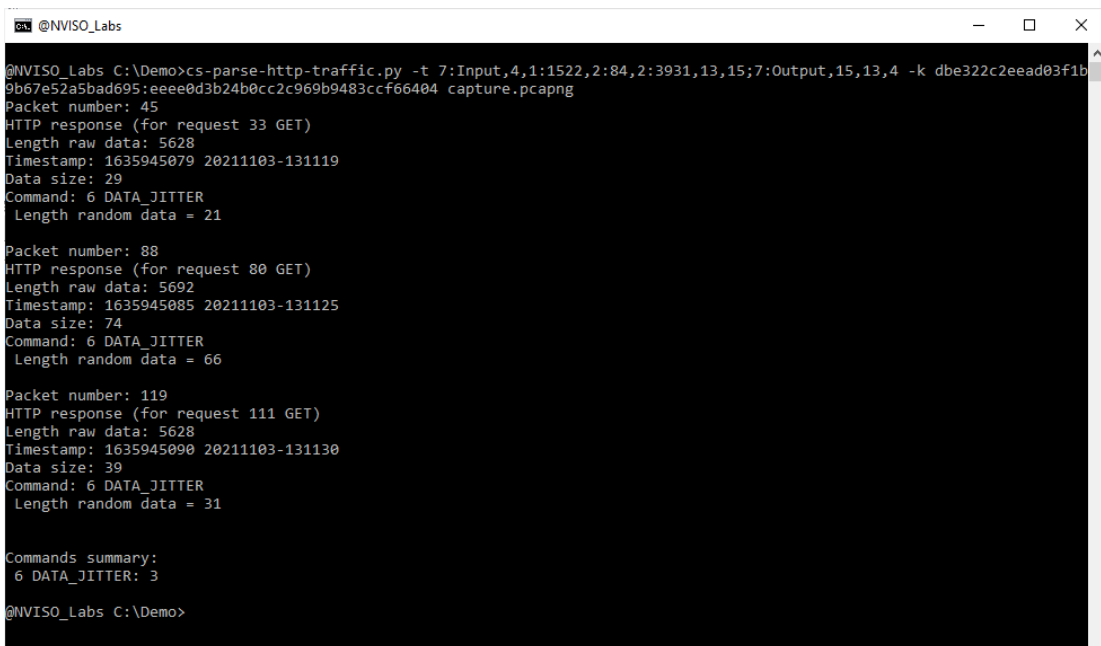
For the tasks to be executed (input), it is:

7:Input,4,1:1522,2:84,2:3931,13,15

And for the results to be posted (output), it is:

7:Output,15,13,4

These instructions can be put together (using a semicolon as separator) and fed via option -t to tool cs-parse-http-traffic.py:



```
@NVISIO_Labs C:\Demo>cs-parse-http-traffic.py -t 7:Input,4,1:1522,2:84,2:3931,13,15;7:Output,15,13,4 -k dbe322c2eed03f1b9b67e52a5bad695:eeee0d3b24b0cc2c969b9483ccf66404 capture.pcapng
Packet number: 45
HTTP response (for request 33 GET)
Length raw data: 5628
Timestamp: 1635945079 20211103-131119
Data size: 29
Command: 6 DATA_JITTER
Length random data = 21

Packet number: 88
HTTP response (for request 80 GET)
Length raw data: 5692
Timestamp: 1635945085 20211103-131125
Data size: 74
Command: 6 DATA_JITTER
Length random data = 66

Packet number: 119
HTTP response (for request 111 GET)
Length raw data: 5628
Timestamp: 1635945090 20211103-131130
Data size: 39
Command: 6 DATA_JITTER
Length random data = 31

Commands summary:
6 DATA_JITTER: 3

@NVISIO_Labs C:\Demo>
```

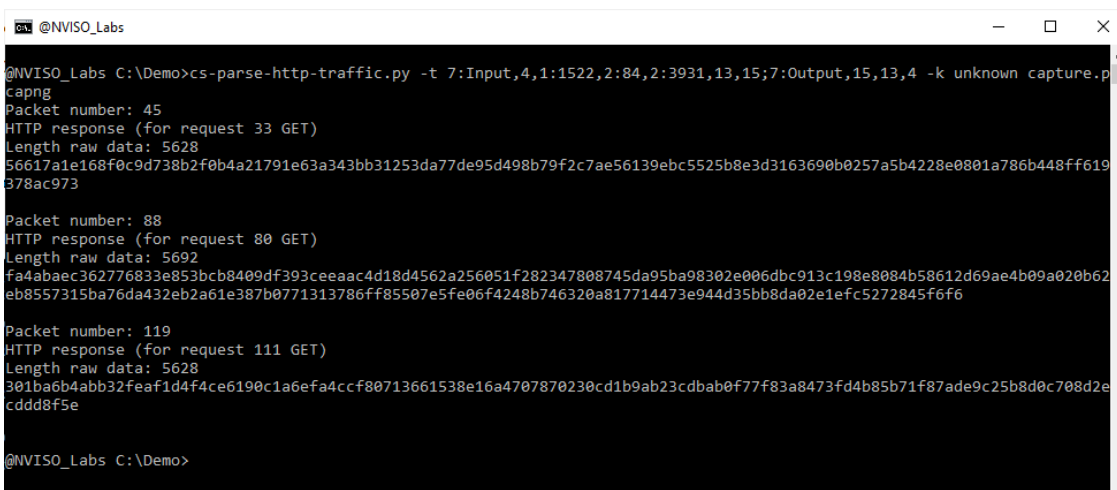
Figure 19: decrypted traffic

And now we finally obtain decrypted traffic. There are no actual commands here in this traffic, just “data jitter”: that is random data of random length, designed to even more obfuscate traffic.

Conclusion

We saw how malleable C2 data transforms are used to obfuscate network traffic, and how we can deobfuscate this network traffic by following the instructions.

We did this manually with CyberChef, but that is of course not practical (we did this to illustrate the concept). To obtain the decoded, encrypted commands, we can also use cs-parse-http-traffic.py. Just like we did in [part 3](#), where we started with an unknown key, we do this here too. The only difference, is that we also need to provide the decoding instructions:



```
@NVISIO_Labs C:\Demo>cs-parse-http-traffic.py -t 7:Input,4,1:1522,2:84,2:3931,13,15;7:Output,15,13,4 -k unknown capture.pcapng
Packet number: 45
HTTP response (for request 33 GET)
Length raw data: 5628
56617a1e168f0c9d738b2f0b4a21791e63a343bb31253da77de95d498b79f2c7ae56139ebc5525b8e3d3163690b0257a5b4228e0801a786b448ff619378ac973

Packet number: 88
HTTP response (for request 80 GET)
Length raw data: 5692
fa4abaec362776833e853bcb8409df393ceeaac4d18d4562a256051f282347808745da95ba98302e006dbc913c198e8084b58612d69ae4b09a020b62eb8557315ba76da432eb2a61e387b0771313786ff85507e5fe06f4248b746320a817714473e944d35bb8da02e1efc5272845f6f6

Packet number: 119
HTTP response (for request 111 GET)
Length raw data: 5628
301ba6b4abb32feaf1d4f4ce6190c1a6efa4ccf80713661538e16a4707870230cd1b9ab23cdbab0f77f83a8473fd4b85b71f87ade9c25b8d0c708d2eccdd8f5e

@NVISIO_Labs C:\Demo>
```

Figure 20: extracting and decoding the encrypted data

And then we can take one of these 3 encrypted data, to recover the keys.

Thus the procedure is exactly the same as explained in [part 3](#), except that option -t must be used to include the malleable C2 data transforms.

About the authors

Didier Stevens is a malware expert working for Nviso. Didier is a SANS Internet Storm Center senior handler and Microsoft MVP, and has developed numerous popular tools to assist with malware analysis. You can find Didier on [Twitter](#) and [LinkedIn](#).

You can follow Nviso Labs on [Twitter](#) to stay up to date on all our future research and publications.

Source: <https://blog.nviso.eu/2021/11/17/cobalt-strike-decrypting-obfuscated-traffic-part-4/>