

Cobalt Strike: Using Process Memory To Decrypt Traffic – Part 3



Blogpost series: [Cobalt Strike: Decrypting Traffic](#)

We decrypt Cobalt Strike traffic with cryptographic keys extracted from process memory.

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. And in [part 2](#), we decrypted Cobalt Strike traffic starting with a private RSA key. In this blog post, we will explain how to decrypt Cobalt Strike traffic if you don't know the private RSA key but do have a process memory dump.

Cobalt Strike network traffic can be decrypted with the proper AES and HMAC keys. In [part 2](#), we obtained these keys by decrypting the metadata with the private RSA key. Another way to obtain the AES and HMAC key, is to extract them from the process memory of an active beacon.

One method to produce a process memory dump of a running beacon, is to use Sysinternals' tool [procdump](#). A full process memory dump is not required, a dump of all writable process memory is sufficient.

Example of a command to produce a process dump of writable process memory: "procdump.exe -mp 1234", where -mp is the option to dump writable process memory and 1234 is the process ID of the running beacon. The process dump is stored inside a file with extension .dmp.

For Cobalt Strike version 3 beacons, the unencrypted metadata can often be found in memory by searching for byte sequence 0x0000BEEF. This sequence is the header of the unencrypted metadata. The earlier in the lifespan of a process the process dump is taken, the more likely it is to contain the unencrypted metadata.

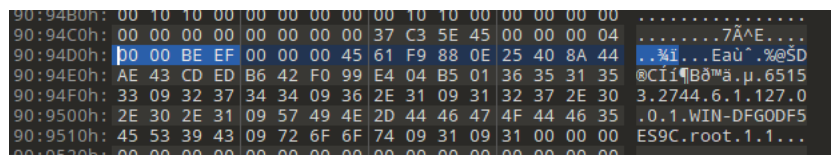


Figure 1: binary editor view of metadata in process

memory

Tool [cs-extract-key.py](#) can be used to find and decode this metadata, like this:

```
@NVIISO_Labs C:\Demo>cs-extract-key.py main.exe_201103_221510.dmp
File: main.exe_201103_221510.dmp
Position: 0x009094d0
Header: 0000beef
Datasize: 00000045
Raw key: 61f9880e25408a44ae43cde4b642f099
aeskey: 9f71c29ed793f778189a10aae54563cb
hmackey: e236ddf9d75b440484a0550e1ca3e2e8
charset: 04e4 ANSI Latin 1; Western European (Windows)
charset_oem: 01b5 OEM United States
Field: b'65153'
Field: b'2744'
Field: b'6.1'
Field: b'127.0.0.1'
Field: b'WIN-DFG0DF5ES9C'
Field: b'root'
Field: b'1'
Field: b'1'

AES key:
Position: 0x001908e0

HMAC key:
Position: 0x001908f0

Position: 0x00ba9aa5
Header: 0000beef
Datasize: 00000048
Raw key: 8d0d15d40500488bd0448bc6488bd8e8
aeskey: 4830e2ed6d39ff67479a74691620b552
hmackey: bf26828cf4f827d3f4aee9a73a42dd60
charset: cc0b
charset_oem: fff7
bid: 8bf83bc6 2348301254
pid: f836001 260268033
port: 0
flags: 85
Field: b'\xc0\x0f\x84X\x01\x00\x00L\x8d\x05\x06\xd4\x05\x00\x8dV\x15H\x8b\xcb\xe8\x87\xab\xf9\xff\x81\xc7\x19\x01\x00\x00\xe8\xac\x9f\xf8\xffH\x8b\x0c%'

AES key:

HMAC key:

Position: 0x01689e72
Header: 0000beef
```

Figure 2: extracted and decoded metadata

The metadata contains the raw key: 16 random bytes. The AES and HMAC keys are derived from this raw key by calculating the SHA256 value of the raw key. The first half of the SHA256 value is the HMAC key, and the second half is the AES key.

These keys can then be used to decrypt the captured network traffic with tool [cs-parse-http-traffic.py](#), like explained in [Part 2](#).

Remark that tool [cs-extract-key.py](#) is likely to produce false positives: namely byte sequences that start with 0x0000BEEF, but are not actual metadata. This is the case for the example in figure 2: the first instance is indeed valid metadata, as it contains a recognizable machine name and username (look at Field: entries). And the AES and HMAC key extracted from that metadata, have also been found at other positions in process memory. But that is not the case for the second instance (no recognizable names, no AES and HMAC keys found at other locations). And thus that is a false positive that must be ignored.

For Cobalt Strike version 4 beacons, it is very rare that the unencrypted metadata can be recovered from process memory. For these beacons, another method can be followed. The AES and HMAC keys can be found in writable process memory, but there is no header that clearly identifies these keys. They are just 16-byte long sequences, without any distinguishable features. To extract these keys, the method consists of performing a kind of dictionary attack. All possible 16-byte long, non-null sequences found in process memory, will be used to try to decrypt a piece of encrypted C2 communication. If the decryption succeeds, a valid key has been found.

This method does require a process memory dump and encrypted data.

This encrypted data can be extracted using tool [cs-parse-http-traffic.py](#) like this: `cs-parse-http-traffic.py -k unknown capture.pcapng`

With an unknown key (-k unknown), the tool will extract the encrypted data from the capture file, like this:

```
@NVISIO_Labs C:\Demo>cs-parse-http-traffic.py -k unknown capture.pcapng
Packet number: 103
HTTP response (for request 97 GET)
Length raw data: 64
d12c14aa698a6b85a8ed3c3c33774fe79acadd0e95fa88f45b66d8751682db734472b2c9c874ccc70afa426fb2f510654df7042aa7d2384229518f26d1e044bd

Packet number: 150
HTTP response (for request 147 GET)
Length raw data: 48
ea94c2fcd029a11caef3b40355c68d4697dd6522cd97870a96891548da818a3e1d8cc0e34c90730a62e0a186a907fc4e

Packet number: 173
HTTP response (for request 171 GET)
Length raw data: 48
307332853e7f2d46e10c1306805cdb5e4b01e8b257f64ab1d11ba0afe91d86d6301256ae7758f710c6edc281a34bdd53

Packet number: 185
HTTP response (for request 183 GET)
Length raw data: 48
246c13a9d5b9f09e308abe822c1cc3e361cb21c3f6a94f9cfc962453aae3c7baa705976d19e2e841de7c60a95b451a87

Packet number: 196
HTTP response (for request 193 GET)
Length raw data: 64
78a62a28a8615f0e4655eddd5e8100bfa23f644148e9715dde8e0fa46f464e57e709e1aa97b7b2cc02d853a750bb5490205e52d491a9c5261e9b74217d0be342

Packet number: 207
```

Figure 3: extracting encrypted data from a capture file
Packet 103 is an HTTP response to a GET request (packet 97). The encrypted data of this response is 64 bytes long:
d12c14aa698a6b85a8ed3c3c33774fe79acadd0e95fa88f45b66d8751682db734472b2c9c874ccc70afa426fb2f510654df7042aa7d2384229518f26d

This is encrypted data, sent by the team server to the beacon: it contains tasks to be executed by the beacon (remark that in these examples, we look at encrypted traffic that has not been transformed, we will cover traffic transformed by malleable instructions in an upcoming blog post).

We can attempt to decrypt this data by providing tool cs-extract-key.py with the encrypted task (option -t) and the process memory dump: cs-extract-key.py -t
d12c14aa698a6b85a8ed3c3c33774fe79acadd0e95fa88f45b66d8751682db734472b2c9c874ccc70afa426fb2f510654df7042aa7d2384229518f26d
rundll32.exe_211028_205047.dmp.

```
@NVISIO_Labs C:\Demo>cs-extract-key.py -t d12c14aa698a6b85a8ed3c3c33774fe79acadd0e95fa88f45b66d8751682db734472b2c9c874ccc70afa426fb2f510654df7042aa7d2384229518f26d1e044bd rundll32.exe_211028_205047.dmp
File: rundll32.exe_211028_205047.dmp
Searching for AES and HMAC keys
Searching after sha256\x00 string (0x6a8179)
AES key position: 0x006ae6c5
AES Key: 23a79f098615fdd74fbb25116f50aa09
HMAC key position: 0x006b19e5
HMAC Key: 5a8a1e3d8c75f37353937475bd498dfb
SHA256 raw key: 5a8a1e3d8c75f37353937475bd498dfb:23a79f098615fdd74fbb25116f50aa09
Searching for raw key

@NVISIO_Labs C:\Demo>
```

Figure 4: extracting AES and HMAC keys from process memory
The recovered AES and HMAC key can then be used to decrypt the traffic (-k HMACkey:AESkey):

```
@NVISO_Labs C:\Demo>cs-parse-http-traffic.py -k 5a8a1e3d8c75f37353937475bd498dfb:23a79f098615fdd74fbb25116f50aa09 capture.pcapng
Packet number: 103
HTTP response (for request 97 GET)
Length raw data: 64
Timestamp: 1635447043 20211028-185043
Data size: 33
Command: 6 DATA_JITTER
Length random data = 25

Packet number: 150
HTTP response (for request 147 GET)
Length raw data: 48
Timestamp: 1635447048 20211028-185048
Data size: 15
Command: 6 DATA_JITTER
Length random data = 7

Packet number: 173
HTTP response (for request 171 GET)
Length raw data: 48
Timestamp: 1635447053 20211028-185053
Data size: 18
Command: 6 DATA_JITTER
Length random data = 10

Packet number: 185
HTTP response (for request 183 GET)
Length raw data: 48
```

Figure 5: decrypting traffic with HMAC and AES key provided via option -k

The decrypted tasks seen in figure 5, are “data jitter”. Data jitter is a Cobalt Strike option, that sends random data to the beacon (random data that is ignored by the beacon). With the default Cobalt Strike beacon profile, no random data is sent, and data is not transformed using malleable instructions. This means that with such a beacon profile, no data is sent to the beacon as long as there are no tasks to be performed by the beacon: the Content-length of the HTTP reply is 0.

Since the absence of tasks results in no encrypted data being transmitted, it is quite easy to determine if a beacon received tasks or not, even when the traffic is encrypted. An absence of (encrypted) data means that no tasks were sent. To obfuscate this absence of commands (tasks), Cobalt Strike can be configured to exchange random data, making each packet unique. But in this particular case, that random data is useful to blue teamers: it permits us to recover the cryptographic keys from process memory. If no random data would be sent, nor actual tasks, we would never see encrypted data and thus we would not be able to identify the cryptographic keys inside process memory.

Data sent by the beacon to the team server contains the results of the tasks executed by the beacon. This data is sent with a POST request (default), and is known as a callback. This data too can be used to find decryption keys. In that case, the process is the same as shown above, but the option to use is -c (callback) in stead of -t (tasks). The reason the options are different, is that the way the data is encrypted by the team server is slightly different from the way the data is encrypted by the beacon, and the tool must be told which way to encrypt the data was used.

Some considerations regarding process memory dumps

For a process memory dump of maximum 10MB, the “dictionary” attack will take a couple of minutes.

Full process dumps can be used too, but the dictionary attack can take much longer because of the larger size of the dump. Tool cs-extract-key.py reads the process memory dump as a flat file, and thus a larger file means more processing to be done.

However, we are working on a [tool](#) that can parse the data structure of a dump file and extract / decode memory sections that are most likely to contain keys, thus speeding up the key recovery process.

Remark that beacons can be configured to encode their writable memory while they are not active (sleeping): in such cases, the AES and HMAC keys are encoded too, and can not be recovered using the methods described here. The dump parsing tool we are working on will handle this situation too.

Finally, if the method explained here for version 3 beacons does not work with your particular memory dump, try the method for version 4 beacons. This method works also for version 3 beacons.

Conclusion

Cryptographic keys are required to decrypt Cobalt Strike traffic. The best situation is to have the corresponding private RSA key. If that is not the case, HMAC and AES keys can be recovered using a process memory dump and capture file with encrypted traffic.

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.