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

blog.nviso.eu/2021/11/03/cobalt-strike-using-process-memory-to-decrypt-traffic-part-3/

November 3, 2021

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.

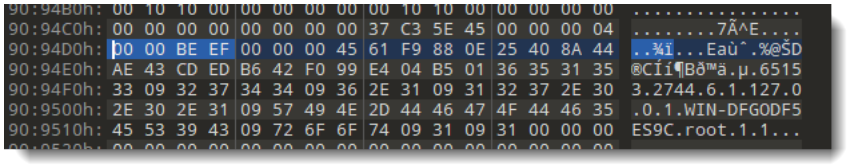

Figure 1: binary editor view of metadata in process memory

Tool cs-extract-key.py can be used to find and decode this metadata, like this:

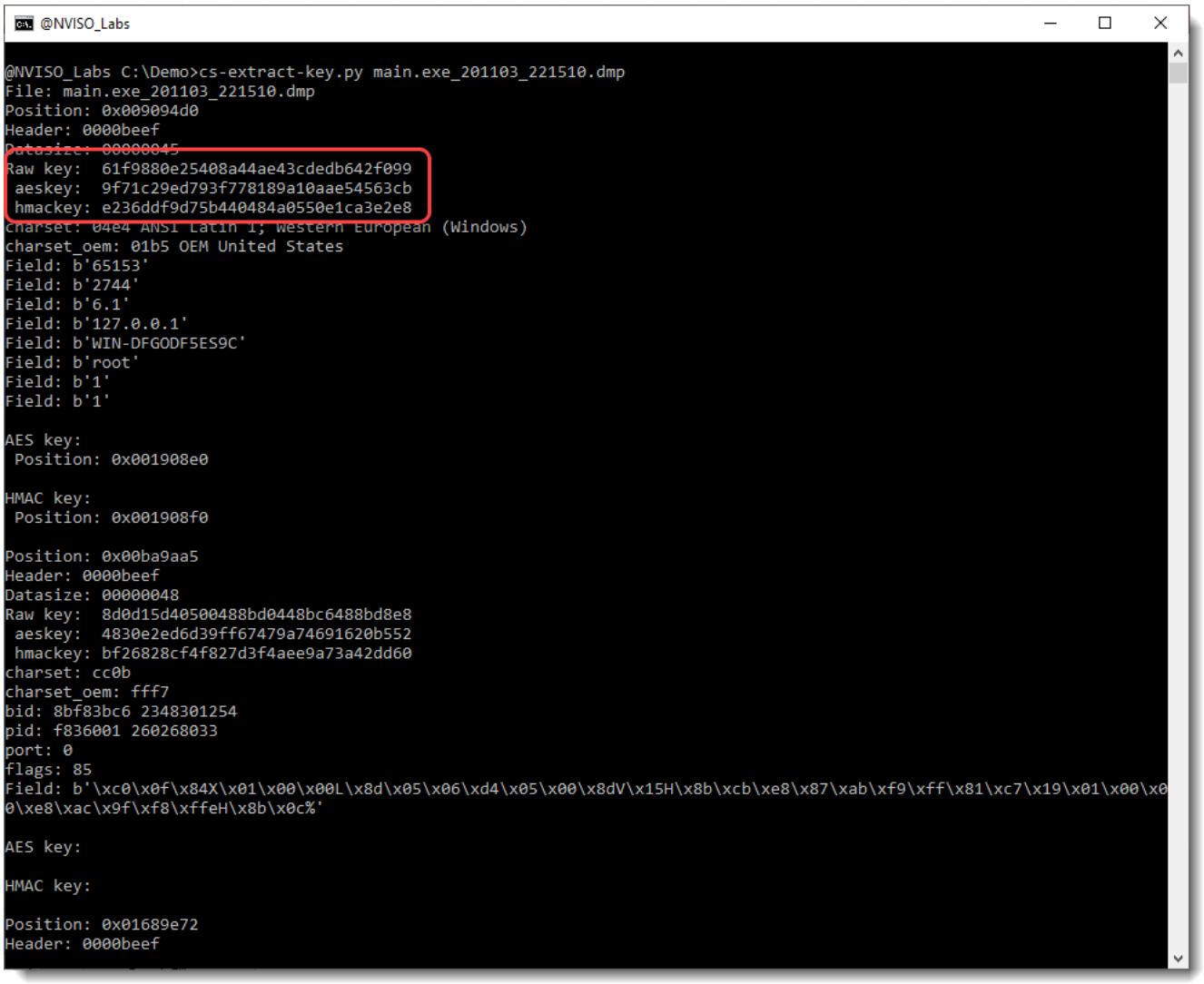

Figure 2: extracted and decoded metadata
The metadata contains the raw key: 16 random bytes. The AES and HMAC keys are derived from this raw key by calculating the SHA256 value of the raw key. The first half of the SHA256 value is the HMAC key, and the second half is the AES key.

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:

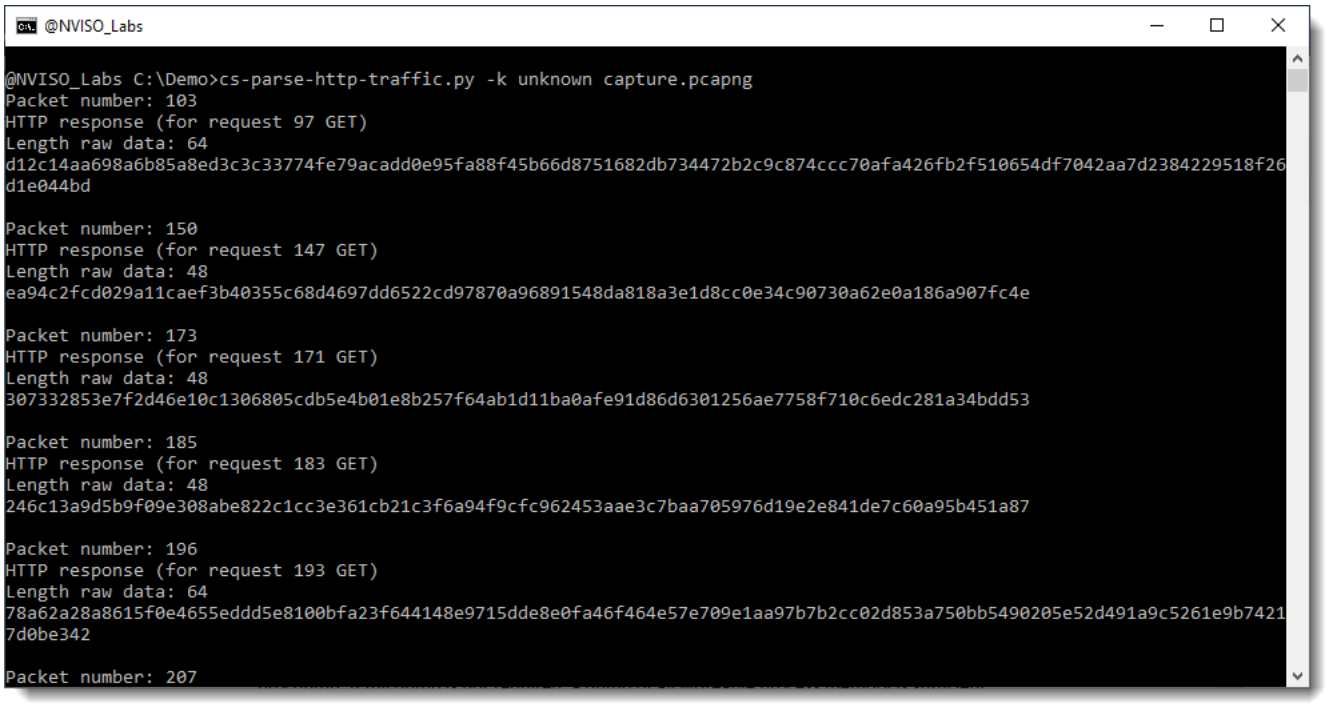

Figure 3: extracting encrypted data from a capture file
Packet 103 is an HTTP response to a GET request (packet 97). The encrypted data of this response is 64 bytes long: d12c14aa698a6b85a8ed3c3c33774fe79acadd0e95fa88f45b66d8751682db734472b2c9c874ccc70afa426fb2f510654df7042aa7d2384229518f26d

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.

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):

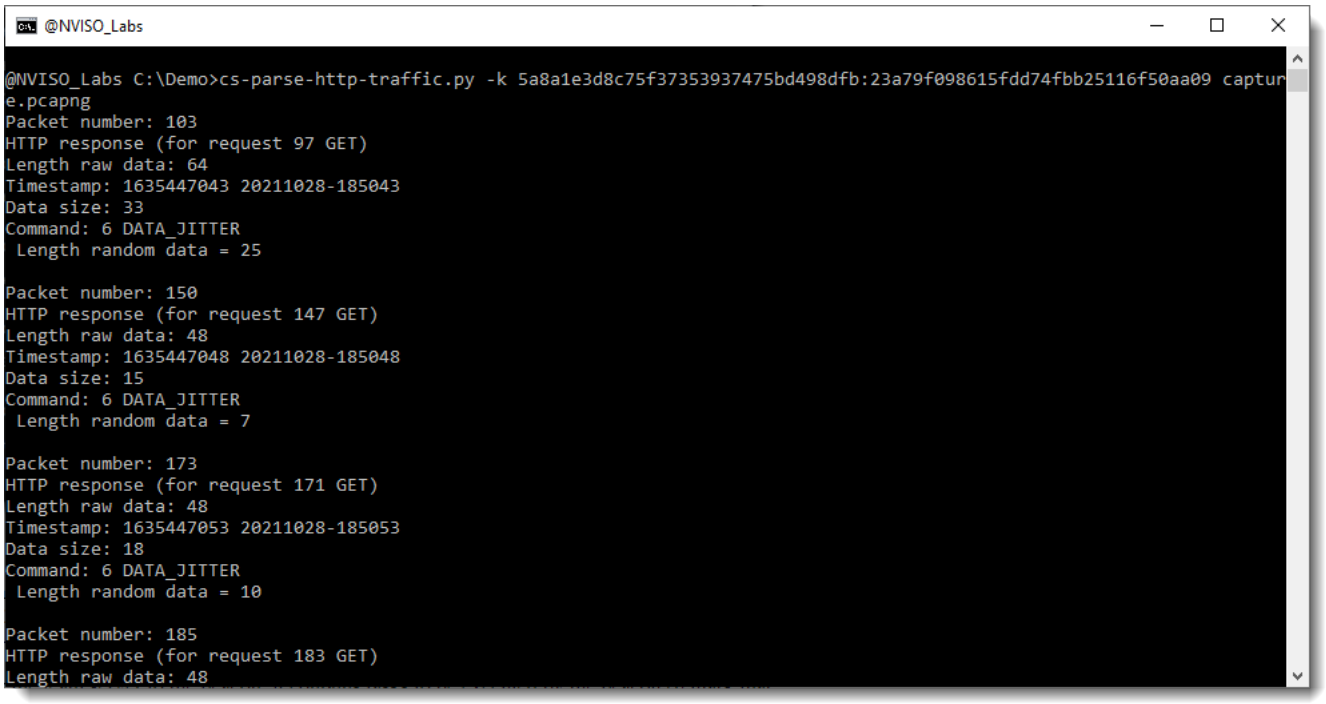

Figure 5: decrypting traffic with HMAC and AES key provided via option -k

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.