Using Kaitai Struct to Parse Cobalt Strike Beacon Configs

By Justin Warner

Published: 2021-04-07 · Archived: 2026-04-05 18:10:31 UTC

I have seen a definite uptick in security researchers hunting Cobalt Strike servers, and tweeting/sharing indicators or config data. There are two popular config parsing methods I have seen: the Nmap NSE script written by @notwhickey and the Sentinel One parser by @gal_kristal (yes, I am aware many organizations have custom parsers). In the case of these two public parsers, I sometimes find myself desiring configuration data that the parsers were missing and when engaging with researchers, I discovered a few did not understand that these tools were not extracting the full configs or did not have a complete understanding of how things worked beneath the hood. That led me to tweet this:

Thankfully, in this thread, Lee Holmes (@Lee_Holmes) responded and pointed me to Kaitai Struct (http://kaitai.io/) as a possible resource for building a parser. Kaitai Struct is a declarative language-neutral method of developing parsers for binary structures. It is open-source and includes many samples for popular formats. This blog will not focus on infrastructure hunting, interrogating servers, proper attacker infrastructure setup, or anything of the likes. This blog is to share my journey using Kaitai Struct to comprehensively parse Beacon configs and dive deep into the binary structure of Malleable C2. Tl;dr, Kaitai Struct is awesome and I built a comprehensive proof-of-concept parser (there are *many* improvements that could be made). Realistic outcomes: I don't expect anyone to change. The parser you are using is probably just fine, and you should keep using whatever works best for you :). Just learn how it works and what collection blindness you might have!

Parsing Beacon Configs: Basics

To Start

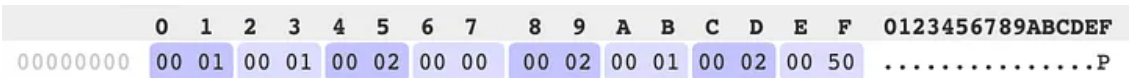
When staged from a server, the Beacon DLL is encoded/packed with a stub loader. You might also find this DLL in memory dumps or in fully-staged Beacon payloads in common malware repositories. Both the NSE script and Sentinel One's parser have the ability to unpack the actual Beacon code and XOR decode the config (0x69 or 0x2e depending on version). I will start from the point that you have decoded config data. We will be focusing on an in-the-wild sample I discovered and uploaded to MalwareBazaar for those who want to follow along. The sample is already unpacked and XOR decoded. The sample uses a popular Amazon Malleable C2 profile created by

(Twitter). To get to the config data in the sample, open it in your favorite hex editor and Ctrl/Cmd-F for the hex string "00 01 00 01 00 02".

Binary Config Structure

Configs are built up of multiple records in the format of (Index# | Type | Length | Value). This is common to the usual “TLV” structure used in binary structures. Within this structure, the type is either 0x1 for short, 0x2 for int, or 0x3 for byte array / blob. The blob (0x3 type) can vary in actual interpretation depending on index/field #, ranging from string to byte array to sub-structures. We can look at a couple of simple examples of this I-TLV in practice from a real world config.

Press enter or click to view image in full size



Two I-TLV structures at the beginning of the sample config.

In the image above, we see two I-TLV structures, the first is (00 01 | 00 01 | 00 02 | 00 00). This structure can be broken down to:

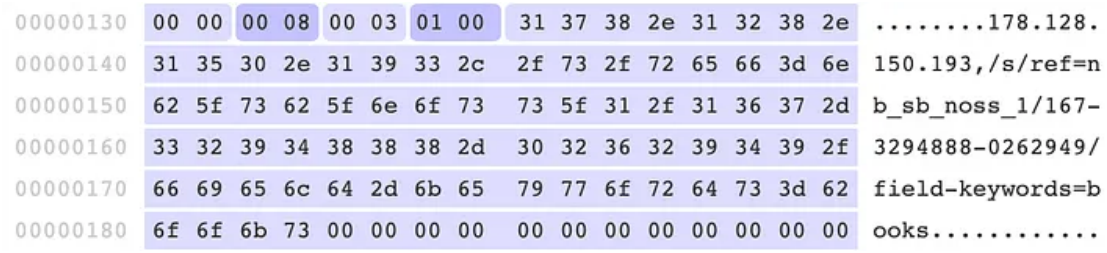
- Index # = 0x01 (Beacon Type) | Type = 0x01 (Short) | Length = 0x02 | Value = 0x00 (HTTP Beacon)

The second is (00 02 | 00 01 | 00 02 | 00 50) which can be broken down to:

- Index # = 0x02 (Port) | Type = 0x01 (Short) | Length = 0x02 | Value = 0x50 (Port 80)

HTTP beacon using port 80, makes sense! Now for the example of a blob value:

Press enter or click to view image in full size



Config field 0x8 showing an example blob structure in the sample data

In the image, we see the structure (00 08 | 00 03 | 01 00| <DATA>). This breaks down to a Field with Index #8 which refers to the C2Server, with 256 bytes of “blob” data. The value is a string containing the IP address and URI of the c2 server. This can be found in the Malleable profile here: `set uri “/s/ref=nb_sb_noss_1/167-3294888-0262949/field-keywords=books”;` .

Get Justin Warner’s stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

Modeling this in Kaitai Struct is relatively straight forward. Our highest level sequence is a “Config” struct that is made up entries. Each entry is the I-TLV structure where the type of value is variable based upon the fieldtype. Within this structure, we handle the “bytes” sections different depending on what config entry it is denoted by the

“index”. Beacon itself knows how to interpret blobs but this knowledge is “assumed” and therefore as the analyst, I have to decide which are arrays, which are strings, and which are substructures. I left a default of a byte array so if it is a new field or an “unknown” one, at least you get the bytes. This is an example of one way we could model this in Kaitai Struct (slightly different than my final version for simplicity):

```
config_entry:
    seq:
      - id: index
        type: u2
      - id: fieldtype
        type: u2
      - id: fieldlength
        type: u2
      - id: fieldvalue
        size: fieldlength
        type:
          switch-on: fieldtype
          cases:
            1: u2
            2: u4
            3: bytesbytes:
    seq:
      - id: byte_val
        type:
          switch-on: _parent._parent.index
          cases:
            11: transform_blocks
            12: req_malleablec2
            13: req_malleablec2
            42: gargle_section
            46: procinj_transform
            47: procinj_transform....... SUB STRUCTURES CONTINUE
```

Beacon Configs: Malleable C2

It might seem simple so far, but things get complicated quick: the Malleable C2 block. These are actually sub-structures that get interpreted and used by Beacon to alter various aspects of the command and control protocols through its own transform language. To understand these, I highly highly recommend reading the Cobalt Strike docs on Malleable C2. Doing binary analysis on these structures actually made me appreciate the beauty of Cobalt Strike. For the sake of this blog, we will only decompose one of the types, the Malleable C2 Request block, which is used in the profile’s protocol sections, specifically in the “client” blocks (used in index #12 and #13 of the binary structure).

Malleable C2 Request Blocks

We will use the following images for our walkthrough. On the left, we have the binary structure that is representing the actual malleable config on the right, inside of the client section. The binary config starts off with the standard I-TLV (0xd | 0x3 | 0x200 | BLOB). Inside the blob is the malleable C2 request structure.

The malleable c2 request structure is made up of commands that have a form of (Command # | Length | Value). The commands at the top level of the client block will either be a _Parameter (#9), _Header (#10), _HostHeader (#16) or BuildTransform (#7). _Parameter, _Header and _HostHeader are sub-structures of format (Length | String) whereas the BuildTransform type is a bit more complicated we will review later. Before going further, let's look at some of the commands in the example above.

- (0xa or _Header| 0xb | "Accept: */*")
- (0xa or _Header | 0x16 | "Content-Type: text/xml")
- (0xa or _Header | 0x20 | "X-Requested-With" "XMLHttpRequest")
- (0x10 or _HostHeader | 0x14 | "Host: www.amazon.com")
- (0x9 or _Parameter | 0xa | "sz=160x600")
- …. And so on

I modeled this structure in Kaitai Struct with two types, the 4-byte command "statement" and the value which will either be a (Length | String) structure or a DataTransform structure (next section).

```
malleable_block:
    seq:
      - id: statement
        type: u4
        enum: transform_actions
      - id: statement_value
        type:
          switch-on: statement
          cases:
            transform_actions::uheader: length_val_string
            transform_actions::uparameter: length_val_string
            transform_actions::build: data_transform
            transform_actions::uhostheader: length_val_string
        if: statement != transform_actions::stop
```

DataTransform Blocks

The DataTransform block is another substructure that is described well in the Cobalt Strike docs in the "Data Transform" section. Within the Malleable C2 profiles, these will be in the "id", "output", or "metadata" blocks and describe how to transform the data CS produces. Profile authors can prepend or append bytes, change encodings, xor mask, etc. For our example, we will focus on the "output" block of the http-post section in the Amazon config.

```
output {
     base64;
```

```
     print;
}
```

Within the binary structure, the DataTransform block is indicated by an command code of 0x7. This substructure will contain a 4-byte code to indicate what section this is and a variable number of Transform Actions to transform data. The Cobalt Strike documentation lists all of the possible transform actions including the termination actions that will always signal the end of transform. The majority of them will simply be a 4 byte action code but some also have variables and therefore will also have a (Length | String) association. Our example above from the Amazon profile is pretty simple and can be broken down like:

- (0x07 = DataTransform) (0x01 = Output Section) (0x03 = Base64) (0x04 = Terminate_Print)

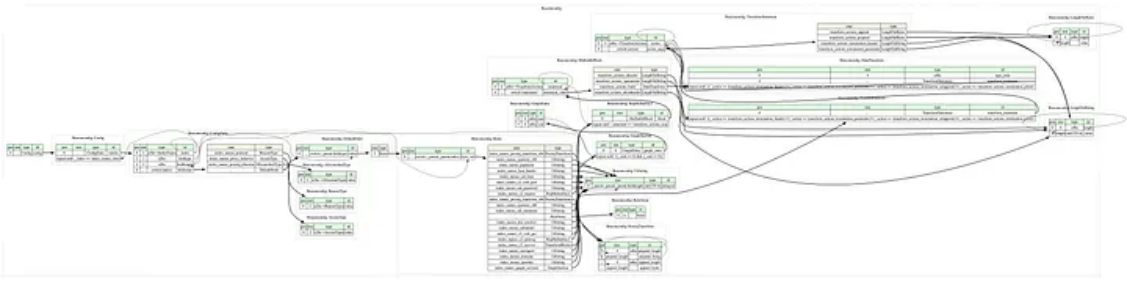
…. And now, in Kaitai Struct :

```
data_transform:
   seq:
     - id: type_code
       type: u4
     - id: transform_statement
       type: transform_statement
       repeat: until
       repeat-until: _.action == <TERMINATION ACTION>
 transform_statement:
   seq:
     - id: action
       type: u4
       enum: transform_actions
     - id: action_args
       type:
         switch-on: action
         cases:
           transform_actions::append: length_val_bytes
           transform_actions::prepend: length_val_bytes
           transform_actions::termination_header: length_val_string
           transform_actions::termination_parameter: length_val_string
       if: <NOT A TERMINATION ACTION>
```

Results & Future Work

My full proof-of-concept KSY file for Beacon config data can be found here: https://gist.github.com/sixdub/a5361168ba7acecf7a7a214bf7e5d3d3.

Press enter or click to view image in full size



GraphViz output of my parser structs. This is not very pretty but so easy to generate!

I also put together a simple static HTML/Javascript util to parse configs using the Kaitai Struct JS lib (I mean… if it makes it easy enough to use in Javascript, it must be okay). Here is a screenshot, please don't judge Kaitai Struct by my HTML:

Press enter or click to view image in full size

Choose File 2acb4fb03f9…256c67b.bin

config:

- PROTOCOL - 8
- PORT - 443
- SLEEPTIME - 60000
- MAXGET - 1048576
- JITTER - 0
- MAXDNS - 255
- PUBKEY - 48,129,159,48,13,6,9,42,134,72,134,247,13,1,1,1,5,0,3,129,141,0,48,129,137,2,129,129,0,167,9,145,214,157,129,10
- DOMAINS - 104.36.231.42,/updates.rss
- USERAGENT - Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0; Touch; MALCJS)
- SUBMITURI - /submit.php
- C2_RECOVER -
  - TERMINATION_PRINT
- C2_REQUEST -
  - BUILD
    - BASE64
    - TERMINATION_HEADER
  - STOP
- C2_POSTREQ -
  - UHEADER
  - BUILD
    - TERMINATION_PARAMETER
  - BUILD
    - TERMINATION_PRINT
  - STOP
- SPAWNTO -
- SPAWNTO_X86 - %windir%\syswow64\rundll32.exe
- SPAWNTO_X64 - %windir%\sysnative\rundll32.exe
- PIPENAME -
- CRYPTO_SCHEME - 0
- DNS_IDLE - 0
- DNS_SLEEP - 0
- C2_VERB_GET - GET

Screenshot of simple HTML/Javascript wrapper POC using my Kaitai Struct Beacon Config Parser. Some wrapping functionality is required to display and pretty print the structure information.

THIS IS A PROOF OF CONCEPT… and far from perfect. This weekend was the *first time* I ever used Kaitai Struct and I really just wanted to demonstrate how to use it for this purpose.

Noted constraints:

- This parser will break if CS adds sub-structures to existing assumed formats such as Malleable C2 Req. For example, if there is a new Malleable Transform Action that has a (length | value) argument, this proof-of-concept will not know about it today. This should handle new “fields” at the higher level just fine, it will treat them as blobs.
- The current parser assumes you start the input at the config start (it does not handle unpacking, xor decoding, or seeking the config).

Possible future work areas:

- I believe my structure definition is overly nested and complicated. I’d bet that it could be optimized many ways by someone with more time in the language. It could definitely be prettier.
- I would love to extend the parser to handle Beacon dlls or stages entirely without the need to first parse the start of the config bytes. I didn’t implement this for the sake of time.
- There are probably more blob structures that have meaning to the Beacon agent and could use sub-structures. I implemented the obvious ones.
- Validate all of the bytes fields that should be strings are properly casted.

Would I do it again?

Overall, I found it simple and intuitive to use. I really appreciated the declarative methodology and the support for multiple languages. I could see this being much easier to maintain as a parser format with a wrapper that just loads the updates. If I were honest, this is potentially a bit of overkill to just hack together a Beacon parser but to build, support, and maintain a long term parsing project, I could totally see the value (or to teach others the structure).

One potential downside was that I still needed to write a wrapper to use and display the Kaitai Struct object (outside of a ruby gem). I did not find a universal toJson type capability that gave you a pretty representation of the resulting object (absent the _root _child _this type object used for traversal).

I could easily see Kaitai Struct being a regular part of my life going forward. I could see direct value in creating definitions for popular forensic artifacts, malware structures, file analysis work, etc. Hope this is helpful or fun!

Happy hacking.
