Understanding Apple's binary property list format

Christos Karaiskos
Feb 27, 2018 · 11 min read

Property lists offer a structured and efficient way to represent and persist hierarchies of objects to disk. They are used extensively in Apple's operating systems to store small amounts of data, with the most well-known example being the *Info.plist,* a file which contains key-value encoded configuration information for bundled executables. Cocoa allows various interchangeable representations for plists, including XML, JSON and binary. The former two have the advantage of being human-readable, while the latter offers the most efficient representation on disk, as well as fast serialization/deserialization. In this blog post, we will take a look at the internal structure of binary plists.

Identifying binary plists

Binary plists (bplists) can be easily identified using the `file` command on macOS.

```
$ file config.plist
config.plist: Apple binary property list
```

Converting between different plist formats can be performed using the `plutil` command, e.g. to convert the above bplist to XML:

```
$ plutil -convert xml1 config.plist
$ file config.plist
config.plist: XML 1.0 document text, ASCII text
```

Binary plist structure

The bplist structure can be found in the comments of the Apple-provided open source CFBinaryPList.c and declarations of ForFoundationOnly.h. It comprises of 4 distinct sections: the header, the object table, the offset table and the trailer.

Header

The file starts off with an 8-byte header, containing the magic "bplist" and the version. For bplist00, the version is 00, but other versions have been known to exist, e.g. bplist15, bplist16. We will only deal with bplist00, which is by far the most common version in Apple's OSs.

Object Table

The 2nd section corresponds to the object table. The object table contains all the objects of the plist. All object types are identified by a single byte, also called a ***marker*** (see Fig. 1). This byte encodes vital metadata about an object, such as its type and size information.

```
HEADER
        magic number ("bplist")
        file format version

OBJECT TABLE
        variable-sized objects

        Object Formats (marker byte followed by additional info in some cases)
        null    0000 0000
        bool    0000 1000                       // false
        bool    0000 1001                       // true
        fill    0000 1111                       // fill byte
        int     0001 nnnn       ...             // # of bytes is 2^nnnn, big-endian bytes
        real    0010 nnnn       ...             // # of bytes is 2^nnnn, big-endian bytes
        date    0011 0011       ...             // 8 byte float follows, big-endian bytes
        data    0100 nnnn       [int]   ...     // nnnn is number of bytes unless 1111 then int count follows, followed by bytes
        string  0101 nnnn       [int]   ...     // ASCII string, nnnn is # of chars, else 1111 then int count, then bytes
        string  0110 nnnn       [int]   ...     // Unicode string, nnnn is # of chars, else 1111 then int count, then big-endian 2-byte uint16_t
                0111 xxxx                       // unused
        uid     1000 nnnn       ...             // nnnn+1 is # of bytes
                1001 xxxx                       // unused
        array   1010 nnnn       [int]   objref* // nnnn is count, unless '1111', then int count follows
                1011 xxxx                       // unused
        set     1100 nnnn       [int]   objref* // nnnn is count, unless '1111', then int count follows
        dict    1101 nnnn       [int]   keyref* objref* // nnnn is count, unless '1111', then int count follows
                1110 xxxx                       // unused
                1111 xxxx                       // unused

OFFSET TABLE
        list of ints, byte size of which is given in trailer
        -- these are the byte offsets into the file
        -- number of these is in the trailer

TRAILER
        byte size of offset ints in offset table
        byte size of object refs in arrays and dicts
        number of offsets in offset table (also is number of objects)
        element # in offset table which is top level object
        offset table offset
```

Fig.1: Binary representation of supported bplist markers (from CFBinaryPList.c)

The marker byte is sometimes enough to fully identify an object. For example, a null value has a marker equal to zero, a boolean value has a marker of 0x08 if false, or 0x09 if true. All other objects can be uniquely identified by their 4 most significant bits (from now on I'll use the terms left-most and right-most instead of MSB and LSB). For example, the 4 left-most bits for an integer are 0001 (0x1), while for a string they are 0110 (0x5).

The remaining right-most 4 bits denote sizing information, i.e., how many bytes the actual value of this type will occupy after the marker. In some cases, if the object is

small enough, the size is encoded immediately in the 4 right-most bits. For example, the ASCII string "Hello" would be encoded as 0x55, and then the actual character values would follow. In other cases, the fill marker (0x0F) is OR-ed with the object marker, denoting that the next bytes encode size information before the actual value bytes. More specifically, if the right-most 4 bits of a marker are equal to 1111 (0xF), the next byte will have the following structure:

- its 4 left-most bits are equal to 0001 (0x1)
- its 4 right-most bits tell us how many bytes we need to encode the object's size. if the 4 right-most bits contain the value ***x***, the size will require ***pow(2, x)*** bytes

Then, ***pow(2, x)*** bytes follow, which should be read in big endian to give us the actual size of the object. After that, the actual values of the object follow. For example, the string "This is a long string" contains 21 ASCII characters. The marker would be **0x5F** followed by the byte **0x10** (since pow(2, 0) = 1, and 1 byte is enough to encode the value 21), then **0x15** (the decimal 21 in hex), then the 21 characters one after the other.

Markers corresponding to objects such as ints, real numbers, strings are immediately followed by a multibyte sequence that represents their actual values (e.g. the individual string characters, as described above). This is not always the case, though. In the case of object containers, such as arrays and dictionaries, the marker byte is followed by **object references** that are simply offsets to the offset table (see next section). Such offsets are ***object_ref_size*** bytes long, as determined by the bplist trailer, and are counted from the beginning of the offset table. Therefore, a container element is just a reference of size ***object_ref_size*** that points back to a position in the offset table, which itself is ***offset_table_offset_size*** bytes long and points back to the object table and specifically to a marker corresponding to the individual object. The examples in the next section will clear up any confusion.

This technique flat-maps the actual multi-level hierarchy and allows all objects to have fixed sizes. Thus, we always know that a marker with value **0xA5** is followed by 5**object_ref_size* bytes. This level of indirection also allows a basic form of compression; when values of a container are exactly the same, they may point to the same offset table offset.

Container examples:

0xA5 — an array of 5 elements. The values of these 5 elements are *not* found immediately after the marker. Instead, after the marker we find 5 object references that act as offsets to the offset table. Following these references gives offsets back to their individual byte-markers in the object table, where the actual values can be found (or if the marker is again a container, the same procedure followed).

0xAF 0x10 0x0F — an array of 15 elements (same as above, but now size info does not fit in 4 bits). The 15 object references follow.

0xD6 — a dictionary of 6 key-value pairs. The actual values of these 6 key-value pairs are *not* found immediately after the marker. Instead, after the marker we find 12 object references that act as offsets to the offset table. First we find the 6 key offsets, then the 6 value offsets grouped together. Following these references gives offsets back to their individual byte-markers in the object table, where the actual values can be found. (or if the marker is again a container, the same procedure followed).

Offset Table

The 3rd section contains offsets to the object table and serves as a way to guide us to the actual values of objects. Each offset is ***offset_table_offset_size*** bytes long, as determined by the bplist **trailer**, and points to the position of a byte marker in the object table. The offset is calculated from the beginning of the file (not the end of the header). The offset table contains ***num_objects*** offsets, denoting how many objects are actually encoded in the object table. Remember that some items might be compressed (encoded once and reused) so when you view the human readable contents of the plist you will probably see more than ***num_objects***.

Trailer

The trailer is 32 bytes long and contains size information. Bytes 0 to 4 are unused, while byte 5 includes the sort version.

Byte 6 tells us how many bytes are needed for each offset table offset (***offset_table_offset_size***). If the plist is large in size, jumping from the offset table to a marker in the object table might require 2 or more bytes. After all, a single byte can only take us 255 bytes from the start of the file and this might not be enough if the number of objects is large.

Similarly, byte 7 tells us how many bytes are needed for each object reference in a container (***object_ref_size***). Again, if the plist is large in size, jumping from the object table to a position in the offset table might require 2 or more bytes.

Bytes 8 to 15 contain the number of objects (***num_objects***) that are encoded. Remember that multibyte numerical values are encoded as big endian.

The ***top_object_offset*** (bytes 16 to 23) tells us the offset from the offset table where we should start from (usually zero, denoting the first item). This position of the offset table contains the offset to the first marker in the object table.

The ***offset_table_start*** (bytes 24 to 31) denotes the start of the offset table, counting from the start of the file.

Example

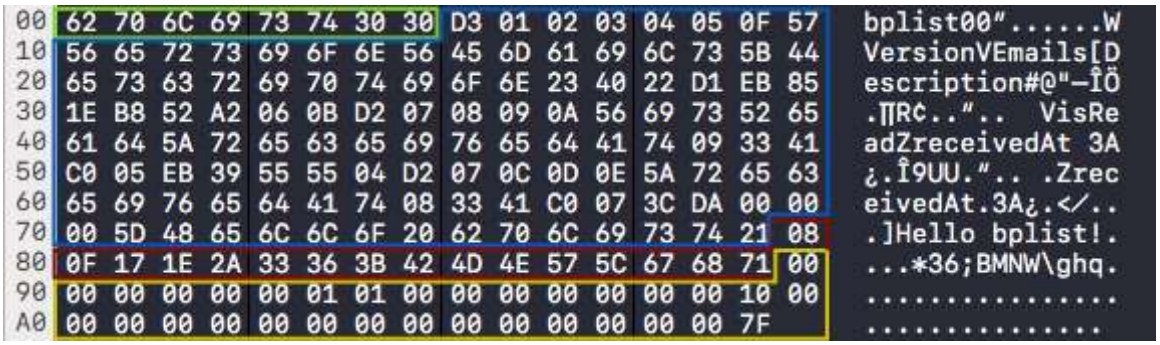
Consider the simple plist below, as seen from within Xcode:

Key	Type	Value
▼ Root	Dictionary	(3 items)
Version	Number	9,41
Description	String	Hello bplist!
▼ Emails	Array	(2 items)
▼ Item 0	Dictionary	(2 items)
isRead	Boolean	YES
receivedAt	Date	14 Jan 2018 at 20:18:26
▼ Item 1	Dictionary	(2 items)
isRead	Boolean	NO
receivedAt	Date	16 Jan 2018 at 20:19:32

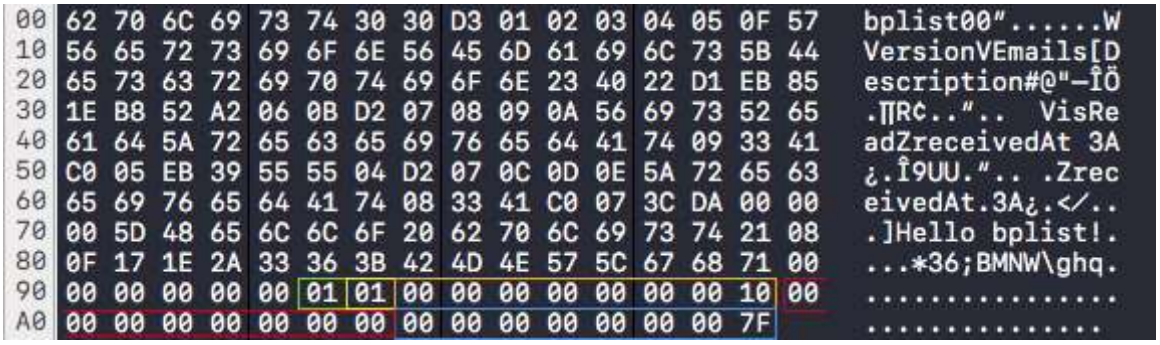
Its XML representation is as follows:

```
<dict>
        <key>Description</key>
        <string>Hello bplist!</string>
        <key>Emails</key>
        <array>
                <dict>
                        <key>isRead</key>
                        <true/>
                        <key>receivedAt</key>
                        <date>2018-01-14T18:18:26Z</date>
                </dict>
                <dict>
                        <key>isRead</key>
                        <false/>
                        <key>receivedAt</key>
                        <date>2018-01-16T18:19:32Z</date>
                </dict>
        </array>
        <key>Version</key>
        <real>9.4100000000000001</real>
</dict>
</plist>
```

The same plist in binary format is shown in Fig.3, where the header (green), object table (blue), offset table (red) and trailer (yellow) are depicted.



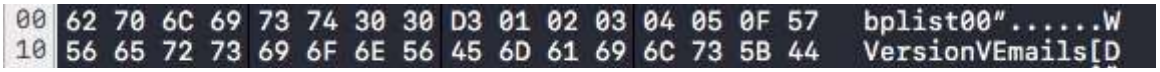
From the first 8 bytes of the header, we immediately identify the plist type to be bplist00. Now, let's take a closer look at the trailer:

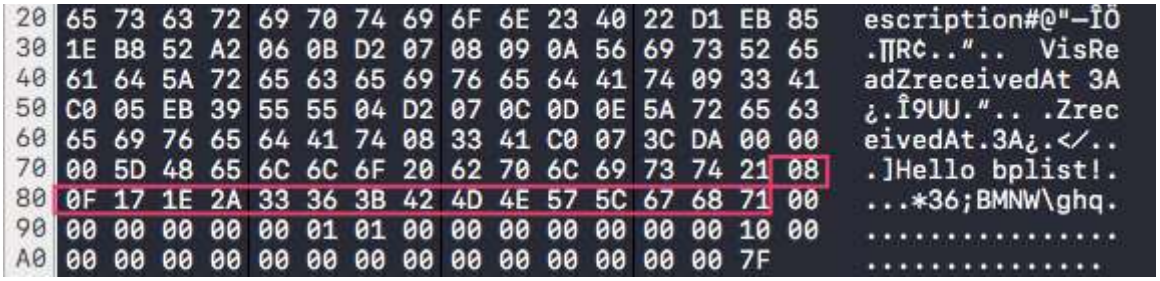


We can immediately deduce the following (Remember, everything is big endian):

- ***offset_table_offset_size***: 0x01 byte (green)
- ***object_ref_size***: 0x01 byte (yellow)
- ***num_objects***: 0x10 (16 objects, orange)
- ***top_object_offset***: 0x00 (red)
- ***offset_table_start***: 0x7F (blue)

Next, let's take a look at the offset table. From the trailer, we already know where the offset table starts (0x7F), how many objects it points to (0x10), what size the offset table slots are (1 byte) and at which offset the first object pointer is located (position 0).





Each offset table entry contains an offset that leads us to its matching marker in the object table (shown below in same color 😳😉):

Let's focus on the markers (…and drop the multicolor):

At a glance, we can see the following 16 objects in order:

1. 0xD3 — A dictionary with 3 key-value pairs
2. 0x57 — An ASCII string with 7 characters
3. 0x56 — An ASCII string with 6 characters
4. 0x5B — An ASCII string with 11 characters
5. 0x23 — A real number of length 8 bytes
6. 0xA2 — An array of 2 elements
7. 0xD2 — A dictionary of 2 key-value pairs

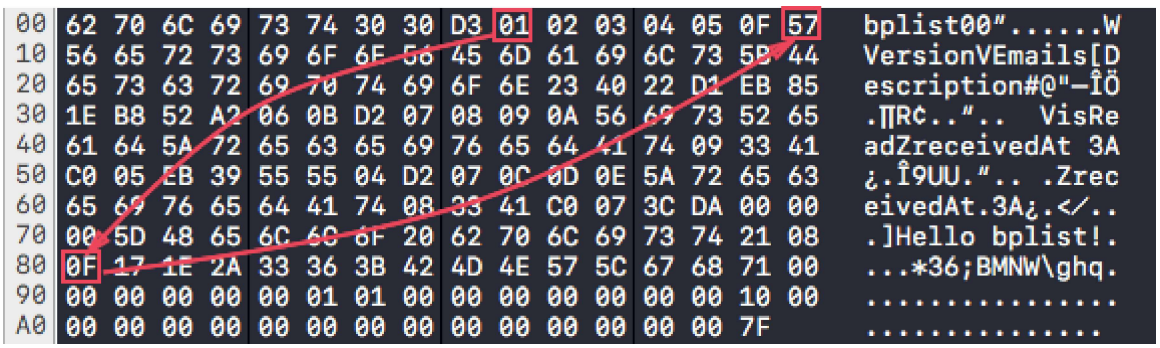
8. 0x56 — An ASCII string with 6 characters
9. 0x5A — An ASCII string with 10 characters
10. 0x09 — A boolean true value
11. 0x33 — A date
12. 0xD2 — A dictionary of 2 key-value pairs
13. 0x5A — An ASCII string with 10 characters
14. 0x08 — A boolean false value
15. 0x33 — A date
16. 0x5D — An ASCII string with 13 characters

Let's follow the path to the first object. Looking at the trailer, the first offset table offset is zero from the start of the offset table, so we look at the offset table start (0x7F). The value contained (0x08) tells us that the first object resides 8 bytes after the beginning of the file. The value there is (0xD3), denoting a dictionary with 3 key-value pairs. At the next position (0x09) start the key references (not the actual keys). These 3 keys are offsets to the offset table. The offsets are 0x01, 0x02, 0x03. After that, the 3 values appear. These are again offsets to the offset table: 0x04, 0x05, 0x0F.

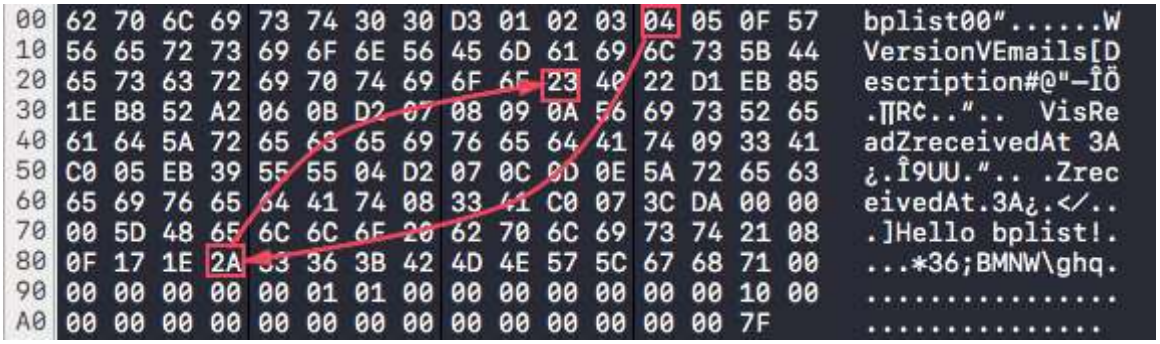
Let's find the 1st key of the dictionary. We move 1 position from our 0xD3 marker, finding the value 0x01. Then, we take the offset table start (0x7F) and add this value, resulting to 0x80. At 0x80 we find the value 0x0F, which takes us to the marker 0x57, denoting a string with 7 bytes:

0x56 0x65 0x72 0x73 0x69 0x6F 0x6E

yielding the dictionary key "Version".



To get its value, we move from our 0xD3 marker, 3 positions to the right and follow the same procedure, resulting to the marker 0x23, which is an 8-byte long real number.

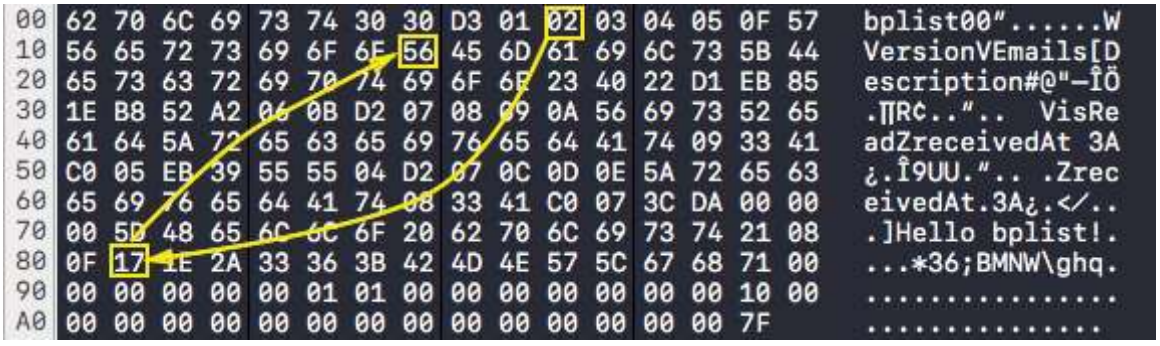


Its value is calculated from the next 8 bytes read in big endian:

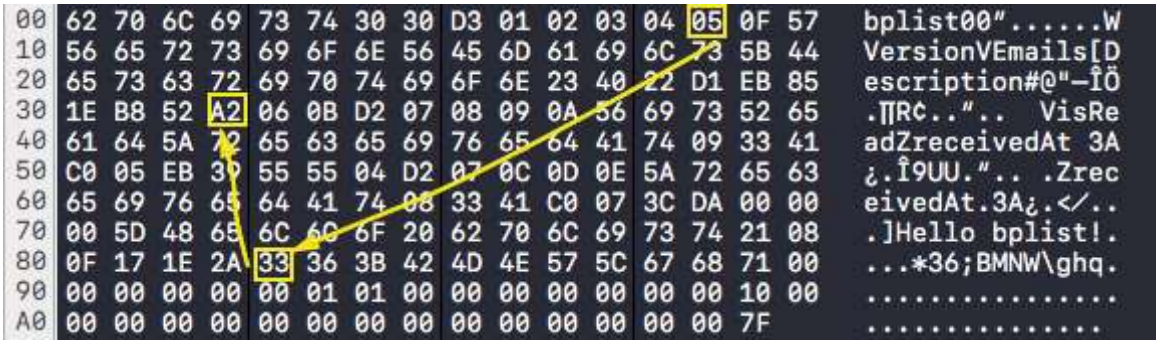
0x40 0x22 0xD1 0xEB 0x85 0x1E 0xB8 0x52

which corresponds to 9.41 in IEEE754 Double precision 64-bit format. So we have our first key-value pair read: **"Version" : 9.41**

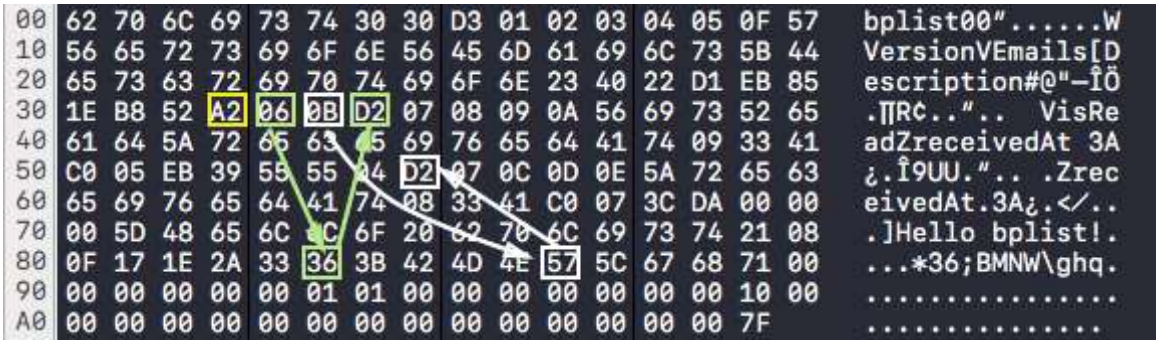
Let's continue with the second key, which contains the value 0x02 and leads us to the marker 0x56. The marker denotes an ASCII string with 6 characters that follow, yielding the key "Emails".



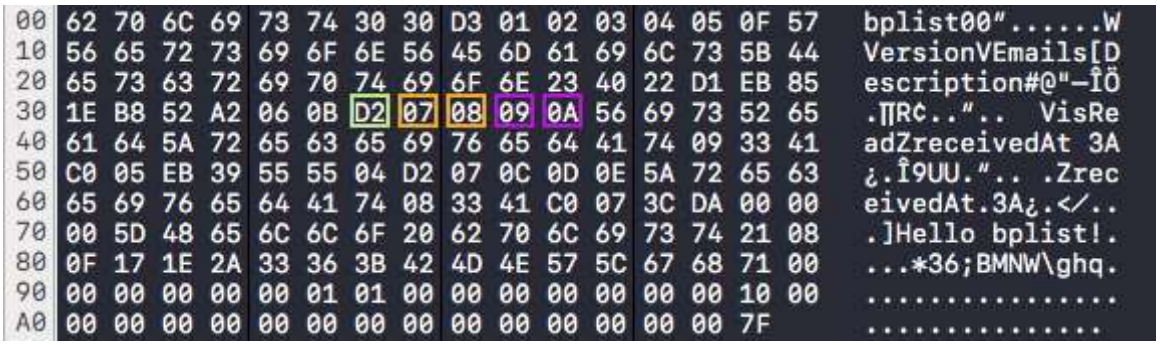
The corresponding value results to the marker 0xA2, which is an array of 2 elements:



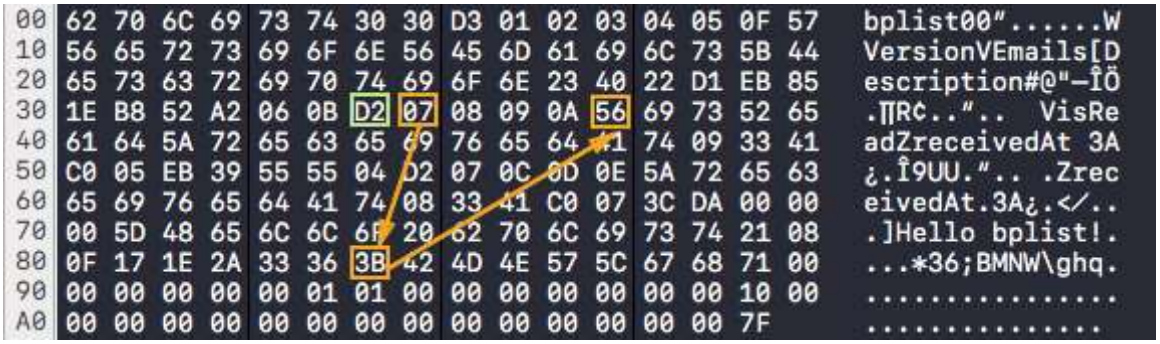
The next two bytes contain the object refs that point at the offset table (follow the green and white paths):



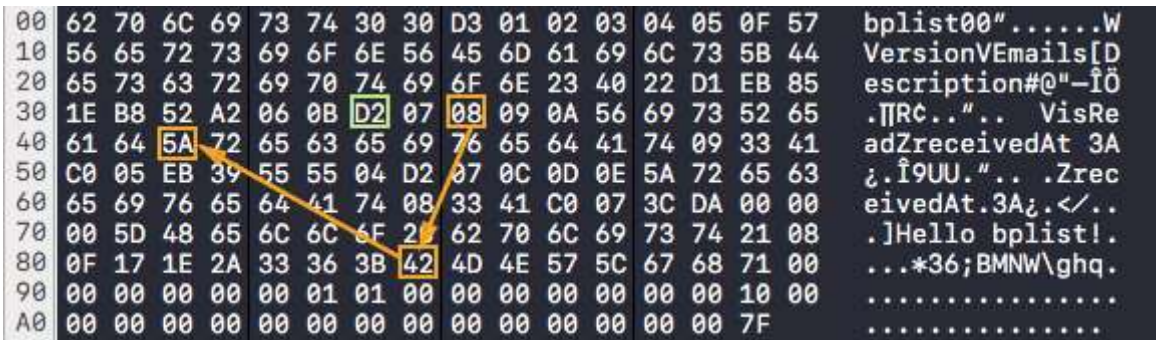
Each offset table offset ends up pointing to a dictionary of 2 key-value pairs. Let's look at the first dictionary:



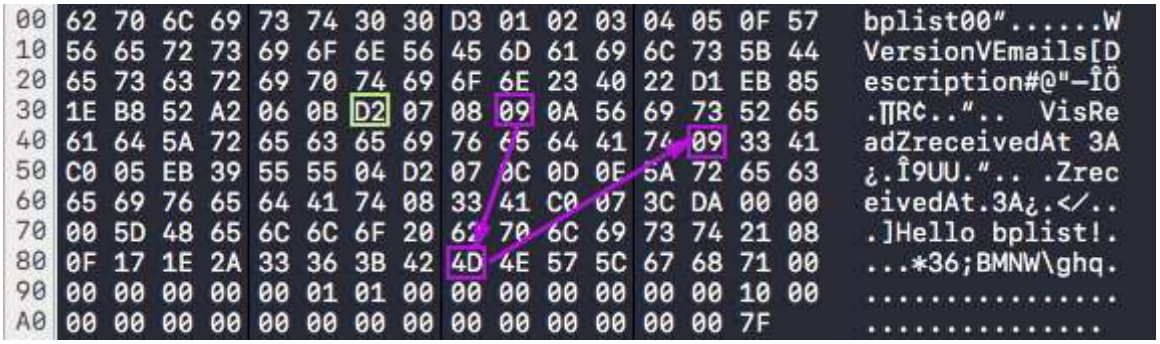
The first key points to the marker 0x56 and the string "isRead"



The second key points to the marker 0x5A and the string "receivedAt"



The first value points to the marker 0x09 which is a Bool with value true.



The second value points to the marker 0x33 which is a date with an 8-byte Core Data timestamp which corresponds to Sun Jan 14 18:18:26 2018 UTC or Sun Jan 14 20:18:26 2018 local time Athens, Greece, as seen in Xcode.

You should by now be able to apply the same pattern to decode the final dictionary at 0x57.

It is worth observing that the dictionary key "isRead" is only saved once for both dictionaries, while the other key "receivedAt" is saved twice, despite being the exact same characters. If you count the human-readable elements of the XML representation you will find 17 objects, while the binary version contains 16 encoded objects. The difference is the compressed "isRead" key.

Conclusion

Binary plists offer a compact way of efficiently storing objects in a structured and transportable manner. In this blog post, we took a peek at the internal structure of the most common bplist version 00.