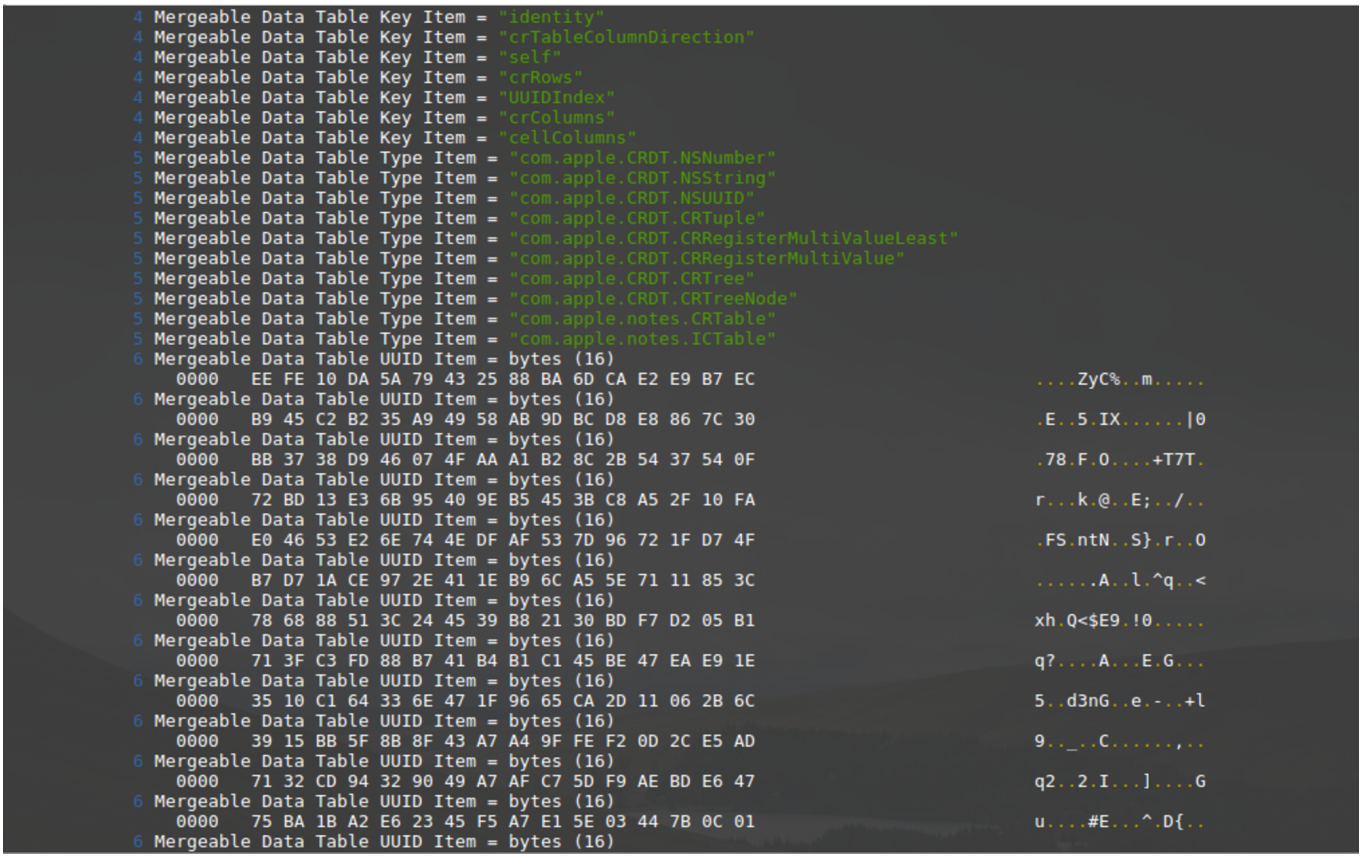
CIOFECA FORENSICS

MOBILE APPLE NOTES

Revisiting Apple Notes (3): Embedded Tables

14 Jan 2020 · 36 mins read



TL;DR: Apple Notes has a few bespoke embedded objects which are messier than the Easy Embedded Objects previously explained. This post covers how to piece back together a more complex embedded object, the Apple Notes Table.

Background

If you haven't read the previous post about Easy Embedded Objects, please go do that now. It will explain the background assumptions of this work and show how this works on some very straight forward objects. These are *not* those, so you'll want to build on that knowledge.

Below are the steps to rebuild `com.apple.notes.table` objects. These are specific to Notes and get much more involved as a result, hence their own post. If you still don't want to do this by hand, feel free to check out the Apple Cloud Notes Parser, which handles the below.

Warning: This gets fairly technical. I highly recommend not doing this by hand, but use this post to understand what is happening under the hood and fact check your tool output. I also highly recommend a good espresso (or two) before beginning. Andiamo!

com.apple.notes.table

The Type UTI of `com.apple.notes.table` represents a columned table embedded in the Note. This type took me the longest to figure out how to parse, and previous work by dunhamsteve was invaluable in sorting out what I was seeing, although it didn't quite describe the right answer for current table types in iOS 13.

Table Structure

Before going into how to rebuild tables, I want to address why this is needed. Table textual content can be pulled out of the underlying protobuf fairly easily (after you figure out it is a protobuf and know where to fetch text from), however the structure is the annoying part. Why worry about structure? Because until the user enters data into one of the table cells, it doesn't appear to even record an empty string which means if you just scan for strings, you might know text exists, but not where it goes. That also means that you can't even figure out how large the table is based on how many cells have text, let along where they go.

Why does that matter? Imagine a table where you only know the user entered three strings: "Things to do today", "Things not to do today", and "kill everybody." Obviously this is an over the top example, but which column "kill everybody" falls in makes a difference. More realistically, you could imagine a table with names and money, knowing which were debits, which were credits, and who did what would be important. Without rebuilding the table, you can't do that reliably.

Why is this the case, when the other embedded objects and textual formatting in a Note are much simpler to understand and parse? Apple can't just store a basic list of rows and columns because of the iCloud integration and the ability to share Notes. Two users can make edits to the same Note and Apple needs to be able to put those together, such as a user adding a column and another user adding a row at the same time. That seems trivial, but if you imagine a 2x2 table which has a row added at the start and a column added at the start, the question of where the original four cells move to when both users' edits are combined quickly becomes non trivial. To be able to tell what goes where when that happens, Apple breaks the table down into rows, columns, and then mappings of cells to those.

For example, imagine a table with two columns, `Column1` and `Column2`, and two rows, `Row1` and `Row2`. Inside are four cells, `Cell1` which belongs to `Column1` and `Row1`, `Cell2` which

belongs to `Column2` and `Row1`, `Cell3` which belongs to `Column2` and `Row2`, and `Cell4` which belongs to `Column2` and `Row2`.

	Column1	**Column2**
Row1	Cell1	Cell2
Row2	Cell3	Cell4

Now if a user adds a new row, `Row3`, at the start of the table, there is no question what happens to each of the cells that exist. Why? Because they don't belong to the new row that is on top. So we would have two new cells, `Cell5` which belongs to `Column1` and `Row3` and `Cell6` which belongs to `Column2` and `Row3`. We haven't changed the mappings of any other cells yet.

	Column1	**Column2**
Row3	Cell5	Cell6
Row1	Cell1	Cell2
Row2	Cell3	Cell4

If, at the same time, another user was deleting `Column2`, Apple would still be able to know exactly which cells to delete. `Cell6`, `Cell2`, and `Cell4` all belong to `Column2`, so they would all die.

	Column1
Row3	Cell5
Row1	Cell1
Row2	Cell3

I don't want to get too far into data types and theory, hopefully this example simply gets across why Apple would make this much more complex than the other aspects of Notes and why you should care to understand it enough to rebuild properly (or use a tool that does).

Rebuilding Tables

The first step to putting a `com.apple.notes.table` back together is finding it. We'll use this as an example Note:

```
root:
    2 <document> = document:
        3 Note = note:
            2 Note Text = "Table title\n\nAfter the table"
            5 Attribute Run = attribute_run: (1 Length = 12)
```

```
5 Attribute Run = attribute_run:
    1 Length = 1
    12 Attachment Info = attachment_info:
        1 Attachment Identifier = "CD0CE698-2765-4C55-B53C-CB8E8C4C5609"
        2 Type UTI = "com.apple.notes.table"
5 Attribute Run = attribute_run: (1 Length = 16)
```

As we know from the previous blog post, there's an attachment in the middle of this Note, with a UUID of `CD0CE698-2765-4C55-B53C-CB8E8C4C5609` and Type UTI of `com.apple.notes.table`. As before, we can pull the right information out of the ZICCLOUDSYNCINGOBJECT table using that UUID, but what we pull is where this gets really different.

```
SELECT ZICCLOUDSYNCINGOBJECT.ZMERGEABLEDATA1
FROM ZICCLOUDSYNCINGOBJECT
WHERE ZICCLOUDSYNCINGOBJECT.ZIDENTIFIER="CD0CE698-2765-4C55-B53C-CB8E8C4C5609"
```

The `ZICCLOUDSYNCINGOBJECT.ZMERGEABLEDATA1` field holds the key for all of the `com.apple.*` types (yes, even the `com.apple.drawing.2` type I hinted at in the last post). For `com.apple.notes.table` objects, this field holds a GZipped protobuf that is similar to, but different from the overall Notes protobuf format. The good news is you already know the first two steps, this value needs to be gunzipped and parsed into its corresponding parts. The bad news is after that, you get ~390 lines to read, about 40 times as much as the Note itself. Let's look at this one, section by section, to understand what they do, with these caveats:

- As with the previous post I'm editing the parts of the protobuf I display to remove a lot of unnecessary information. Because this gets so involved, though, you can download a copy for reference.
- I will generally be presenting these in the same order I needed to in the rebuild_table method of the AppleNotesEmbeddedTable object in Apple Cloud Notes Parser
- My naming on the protobuf parsing is not necessarily the most clear. Please know those are wholly names I've chosen as I reveresed the format, and don't believe they are part of an Apple standard somewhere. Cleaning them up is on the todo list.
- I fully expect the protobuf examples given to be referenced as you read the text. The only way this made sense to me was as I followed along with my finger and finally got to the end, I'm trying to set up the same sorts of situations for the reader.

Key Items

First we need to understand the key items, we will later use this to understand what type of map entry we are looking at in to protobuf.

```
4 Mergeable Data Table Key Item = "identity"
4 Mergeable Data Table Key Item = "crTableColumnDirection"
4 Mergeable Data Table Key Item = "self"
4 Mergeable Data Table Key Item = "crRows"
```

```
4 Mergeable Data Table Key Item = "UUIDIndex"
4 Mergeable Data Table Key Item = "crColumns"
4 Mergeable Data Table Key Item = "cellColumns"
```

Without getting into protobuf specifics, field 4 under the `Mergeable Data Table Data` message is a repeatable field. This means there can be any number of entries and, while the order matters for parsing, the order is not guaranteed to be the same, table to table. By that I mean that in this specific table, "crRows" was in position 3 (0-based ordering, of course), but in another table, it might swap places with "crColumns". What you need to do with this section is build an Array that maps the position (i.e. 0) to the entry (i.e. "identity") as you'll refer to that later.

Type Items

Next we can do the same thing to the type items.

```
5 Mergeable Data Table Type Item = "com.apple.CRDT.NSNumber"
5 Mergeable Data Table Type Item = "com.apple.CRDT.NSString"
5 Mergeable Data Table Type Item = "com.apple.CRDT.NSUUID"
5 Mergeable Data Table Type Item = "com.apple.CRDT.CRTuple"
5 Mergeable Data Table Type Item = "com.apple.CRDT.CRRegisterMultiValueL
5 Mergeable Data Table Type Item = "com.apple.CRDT.CRRegisterMultiValue"
5 Mergeable Data Table Type Item = "com.apple.CRDT.CRTree"
5 Mergeable Data Table Type Item = "com.apple.CRDT.CRTreeNode"
5 Mergeable Data Table Type Item = "com.apple.notes.CRTable"
5 Mergeable Data Table Type Item = "com.apple.notes.ICTable"
```

This looks a lot like before and we need to do the same thing: build an Array mapping the position to the entry. In this case, our 0-th entry would be "com.apple.CRDT.NSNumber", our 1st would be "com.apple.CRDT.NSString", and so on.

UUID Items

Next we repeat with the internal UUIDs, in our early example, these would be things like `Row1`, `Column1`, and `Cell1`, but in a nice binary format.

```
6 Mergeable Data Table UUID Item = bytes (16)
    0000   EE FE 10 DA 5A 79 43 25 88 BA 6D CA E2 E9 B7 EC
6 Mergeable Data Table UUID Item = bytes (16)
    0000   B9 45 C2 B2 35 A9 49 58 AB 9D BC D8 E8 86 7C 30
6 Mergeable Data Table UUID Item = bytes (16)
    0000   BB 37 38 D9 46 07 4F AA A1 B2 8C 2B 54 37 54 0F
6 Mergeable Data Table UUID Item = bytes (16)
    0000   72 BD 13 E3 6B 95 40 9E B5 45 3B C8 A5 2F 10 FA
6 Mergeable Data Table UUID Item = bytes (16)
    0000   E0 46 53 E2 6E 74 4E DF AF 53 7D 96 72 1F D7 4F
6 Mergeable Data Table UUID Item = bytes (16)
    0000   B7 D7 1A CE 97 2E 41 1E B9 6C A5 5E 71 11 85 3C
```

```
6 Mergeable Data Table UUID Item = bytes (16)
    0000   78 68 88 51 3C 24 45 39 B8 21 30 BD F7 D2 05 B1
6 Mergeable Data Table UUID Item = bytes (16)
    0000   71 3F C3 FD 88 B7 41 B4 B1 C1 45 BE 47 EA E9 1E
6 Mergeable Data Table UUID Item = bytes (16)
    0000   35 10 C1 64 33 6E 47 1F 96 65 CA 2D 11 06 2B 6C
6 Mergeable Data Table UUID Item = bytes (16)
    0000   39 15 BB 5F 8B 8F 43 A7 A4 9F FE F2 0D 2C E5 AD
6 Mergeable Data Table UUID Item = bytes (16)
    0000   71 32 CD 94 32 90 49 A7 AF C7 5D F9 AE BD E6 47
6 Mergeable Data Table UUID Item = bytes (16)
    0000   75 BA 1B A2 E6 23 45 F5 A7 E1 5E 03 44 7B 0C 01
6 Mergeable Data Table UUID Item = bytes (16)
    0000   7F EB 40 43 05 25 4E ED 9F E4 4F 16 C4 3C EF CF
```

Yet again, we just need an Array and we will be storing the actual bytes above. Where needed in this post, I will refer to them using the hex representation given, not the string representation listed on the right.

At this point, we have arrays for the key items, type items, and UUIDs to refer to as we parse through the meat of the protobuf. We could say, for example, that `key_items[0]` is “identity”, `type_items[0]` is “com.apple.CRDT.NSNumber”, and `uuid_items[0]` is “EEFE10DA5A79432588BA6DCAE2E9B7EC”.

Table Objects

Finally, we need to add all of the `Mergeable Data Table Object` messages, which are repeatable field 3 in the `Mergeable Data Table Data` message, to another Array, let’s call it `table_objects`. Let’s look at just the first one to understand what we’re dealing with.

```
3 Mergeable Data Table Object = mergeable_data_table_object:
    13 Table Map = mergeable_data_table_custom_map:
        1 Type = 9
        3 Map Entry = map_entry:
            1 Key = 0
            2 Value = object_id:
                4 String Value = "00000000-0000-0000-0000-000000000000"
        3 Map Entry = map_entry:
            1 Key = 1
            2 Value = object_id(6 Object Index = 1)
        3 Map Entry = map_entry:
            1 Key = 3
            2 Value = object_id(6 Object Index = 3)
        3 Map Entry = map_entry:
            1 Key = 5
            2 Value = object_id(6 Object Index = 10)
        3 Map Entry = map_entry:
            1 Key = 6
            2 Value = object_id(6 Object Index = 17)
```

In this case, the `Mergeable Data Table Object` has exactly one message under it, field 13, which is a `Table Map`. `Table Maps` always have an integer called `Type` in field 1 and then have a repeatable field 3 called `Map Entry`. We will be looking up the `Type` integer in our `type_items` Array and the `Key` in the `Map Entry` in our `key_item` Array. Make sense? It didn't to me either at first, so let's look deeper.

In this case, we have a `Type` of 9. If we refer back to our list of `type_items`, the bottom one would be index 9 (entry number 10): "com.apple.notes.ICTable". This tells us this specific `Table Map` is an ICTable, great! Next we can look at each `Map Entry` to figure out what they are.

- The first has a `Key` of 0, which would be the first entry in our `key_items`, or "identity". Its value is all 0's, formatted as a UUID, that doesn't appear too helpful.
- The second has a `Key` of 1, which would be "crTableColumnDirection". That proves to be helpful should you deal with the large swaths of the world that don't write left-to-right.
- The third has a `Key` of 3, which is "crRows", which is how we identify all the rows in the table. Its value is an `Object ID` message which has only one field under it, an `Object Index` set to 3. Everytime you see an `Object Index` you are going to use the resulting value to look up a `Mergeable Data Table Object` in the `table_object` Array. In this case, whatever is in `table_objects[3]` is "crRows"
- The fourth has a `Key` of 5, which is "crColumns" and how we identify all the columns in the table. Its value is the `Object ID` with `Object Index` of 10, so `table_objects[10]` is our "crColumns".
- The fifth and final entry has a `Key` of 6, which is "cellColumns" and how we identify the mappings of cells to columns and rows. In this case, the object at `table_objects[17]` is what we want.

Brief Summary

That's a lot to take in, so for a quick summary, at this point we have an Array of all the key items, an Array of all the type items, an Array of all the UUIDs, and an Array of all the table objects. We also know how to deal with a `Map Entry` message, by looking the key up in our key Array and potentially using an `Object Index` to go pull out the table object it is referring to. We essentially have all the information we need at this point, we just need to stitch it back together.

Identifying Rows and Columns

To put this back together I start by finding the "com.apple.notes.ICTable" (which we accidentally did by looking at the first item, but don't assume it will always be there, loop over all items until you find the right `Type` based on the `type_items` Array). At this point I loop over each of the `Map Entry` message under it, as we did in our example above. I check the `Key` of each of them and handle both the "crRows" and "crColumns" very similarly. Let's look at the object that "crRows" pointed to, it was in object 3 (but note that the initial '3' in this section is referring to field 3 of a previous message, you'll see that on all of these `Mergeable Data Table Objects`):

```
3 Mergeable Data Table Object = mergeable_data_table_object:
    16 Ordered Set = ordered_set:
        1 Ordering = ordered_set_ordering:
            1 Array = ordered_set_ordering_array:
                1 Contents = note:
                    2 Note Text = ""
                    5 Attribute Run = attribute_run(1 Length = 1)
                    5 Attribute Run = attribute_run(1 Length = 1)
                    5 Attribute Run = attribute_run(1 Length = 1)
                2 Attachments = ordered_set_ordering_array_attachments:
                    1 Index = 0
                    2 UUID = bytes (16)
                        0000   BB 37 38 D9 46 07 4F AA A1 B2 8C 2B 54 37
                2 Attachments = ordered_set_ordering_array_attachments:
                    1 Index = 1
                    2 UUID = bytes (16)
                        0000   78 68 88 51 3C 24 45 39 B8 21 30 BD F7 D2
                2 Attachments = ordered_set_ordering_array_attachments:
                    1 Index = 2
                    2 UUID = bytes (16)
                        0000   E0 46 53 E2 6E 74 4E DF AF 53 7D 96 72 1F
            2 Contents = dictionary:
                1 Dictionary Element = dictionary_element:
                    1 Key = object_id(6 Object Index = 5)
                    2 Value = object_id(6 Object Index = 4)
                1 Dictionary Element = dictionary_element:
                    1 Key = object_id(6 Object Index = 7)
                    2 Value = object_id(6 Object Index = 6)
                1 Dictionary Element = dictionary_element:
                    1 Key = object_id(6 Object Index = 9)
                    2 Value = object_id(6 Object Index = 8)
        2 Elements = dictionary:
            1 Dictionary Element = dictionary_element:
                1 Key = object_id(6 Object Index = 5)
                2 Value = object_id(6 Object Index = 5)
            1 Dictionary Element = dictionary_element:
                1 Key = object_id(6 Object Index = 7)
                2 Value = object_id(6 Object Index = 7)
            1 Dictionary Element = dictionary_element:
                1 Key = object_id(6 Object Index = 9)
                2 Value = object_id(6 Object Index = 9)
```

This table object has one field under it, 16, which is an `Ordered Set`. The only places these `Ordered Sets` come up are the "crRows" and "crColumns" (that I've seen) and they serve help us understand where the rows and columns go.

As you look under the `Ordering` message, you'll see an Array with field 1 being a Note! We parsed a Note to find the UUID of this com.apple.notes.table, opened the table and got another Note, how crazy is that? Now, this isn't a real Note, this is just using the same protobuf as you'd find in `ZICNOTEDATA`, including using the Unicode character for a replacement to identify the

rows. We can tell from that there are three rows because there are three replacement characters.

Unlike a normal Note, however, we don't see what to replace it with in the Note itself, for that information we look lower at the `Attachments` repeated field 2 and see three UUIDs with indexes. Index 0, for example, is "BB3738D946074FAAA1B28C2B5437540F". That should look familiar, it is one of the entries in our `uuid_items` Array from earlier. So this tells us that the first row is called "BB3738D946074FAAA1B28C2B5437540F", the second is called "786888513C244539B82130BDF7D205B1", and the third "E04653E26E744EDFAF537D96721FD74F".

Here comes the really annoying part that stumped me for so long. How do you take the knowledge that `Row1` is "BB3738D946074FAAA1B28C2B5437540F" and use that to actually display data in the right place? What you end up doing is keeping track of pointers back to the correct row index. What I mean by that is we look at "BB3738D946074FAAA1B28C2B5437540F" and we know that it is in position 2 (remember, 0-based indexing) of our `uuid_items` Array. So then we would record something like `row_index[2] = 0`. Meaning, if I ever look up the UUID that is in position 2 from a pointer in this protobuf, I need to know that gets spit out as the first row. We would also add in `row_index[6] = 1` and `row_index[4] = 2` based on the following `Attachments`. All of this gives us a way to go from the row's UUID (or rather its index since numbers are far nicer to type than a lot of hex) to where we need to spit it out on the screen.

Sadly, that's not enough, as you'll never find pointers directly to any of these UUIDs. This is what drove me crazy and the answer is in the next section of `Ordering`, the `Contents` in field 2. These `Dictionary Elements` all have a `Key` and a `Value`, both of which are `Object Indexes`. What it is saying is something in the `table_object` in the `Key` equals something in the `table_object` in the `Value`. We already know we have to go find those objects, so here are objects 4 and 5, respectively:

```
3 Mergeable Data Table Object = mergeable_data_table_object:
    13 Table Map = mergeable_data_table_custom_map:
        1 Type = 2
        3 Map Entry = map_entry:
            1 Key = 4
            2 Value = object_id: (2 Unsigned Integer Value = 1)
3 Mergeable Data Table Object = mergeable_data_table_object:
    13 Table Map = mergeable_data_table_custom_map:
        1 Type = 2
        3 Map Entry = map_entry:
            1 Key = 4
            2 Value = object_id: (2 Unsigned Integer Value = 2)
```

Thankfully we see `Table Maps` and we already know how to deal with that. `Table Object` 4 is `Type` 2, which going back to way earlier we know is "com.apple.CRDT.NSUUID". It has a `Map Entry` with `Key` of 4 ("UUIDIndex") and a value that is the number 1. This means the item in

our UUID item array in index 1 is what this entire object refers to: "B945C2B235A94958AB9DBCD8E8867C30".

`Table Object` 5 is also `Type 2` and also has a `Map Entry` with `Key 4`, but its value is 2. That means it refers to the UUID at index 2 in our `uuid_items` Array: "BB3738D946074FAAA1B28C2B5437540F", now we finally have something referencing one of our rows!

Taken altogether, we would use that first `Dictionary Element` to insert another row into our pointers that says `row_index[1] = 0`. Meaning that if we get the UUID in index 1, that *also* refers to the first row (index 0). We would run through that whole process with the other two `Dictionary Elements` as well, to end up with this pseudocode for our `row_item` pointers:

```
row_items[2] = 0
row_items[1] = 0
row_items[5] = 1
row_items[6] = 1
row_items[3] = 2
row_items[4] = 2
```

Brief Summary

That is a *lot* to work through and took a few hours of reading what others were saying and stepping through bit by bit to get it. To summarize again, the "crRows" and "crColumns" entries require you to not just note what the row and column UUIDs are, but what their index is in the `uuid_items` Array you built and note all of the other UUIDs that can point to the same place. For the sake of brevity, assume we've now done the same on the "crColumns" entry which has exactly the same structure and we have built these Hashes for ourselves telling us exactly which UUID indices map to which rows and columns for output.

```
row_items[2] = 0
row_items[1] = 0
row_items[5] = 1
row_items[6] = 1
row_items[3] = 2
row_items[4] = 2
column_items[9] = 0
column_items[10] = 0
column_items[11] = 1
column_items[12] = 1
column_items[7] = 2
column_items[8] = 2
```

Identifying Cells

With our knowledge of which UUIDs point to which rows and columns we are so close to being able to build this table. We can certainly at this point flesh out the size of the table (this is a

3x3). To finish this off, let's look at our "cellColumns" object, remember we'd previously identified it was in index 17 of our `table_items` Array.

```
3 Mergeable Data Table Object = mergeable_data_table_object:
    6 Dictionary = dictionary:
        1 Dictionary Element = dictionary_element:
            1 Key = object_id(6 Object Index = 21)
            2 Value = object_id(6 Object Index = 18)
```

After all the rest, this doesn't look scary at all, but just wait. To understand this, you need to know that the "cellColumns" object is made up of a `Dictionary` which has a repeatable field `Dictionary Element`. Each of these `Dictionary Elements` represents a column. That seems odd because we said this is a 3x3 table which should have 3 columns. Remember the warning at the start of this post, Apple only remembers which cells actually have text. In this case, only one column is needed to track that because only one column had any cells with text in them.

For each and every column, then, we will have yet another `Key - Value` pair saying that something in the key `Object Index` is equal to something in the value `Object Index`. This should be old hat now, we go grab indixes 18 and 21 respectively to see what's in them:

```
3 Mergeable Data Table Object = mergeable_data_table_object:
    6 Dictionary = dictionary:
        1 Dictionary Element = dictionary_element:
            1 Key = object_id(6 Object Index = 20)
            2 Value = object_id(6 Object Index = 19)
3 Mergeable Data Table Object = mergeable_data_table_object:
    13 Table Map = mergeable_data_table_custom_map:
        1 Type = 2
        3 Map Entry = map_entry:
            1 Key = 4
            2 Value = object_id:
                2 Unsigned Integer Value = 11
```

Our `Key` in this case was 21, the second message above, which already looks familiar. We know the `Type` of 2 and `Map Entry Key` of 4 means we're looking up a UUID, specifically index 11. Recall above we noted that UUID index 11 is one of the column UUIDs that points to the second column (index 1 in a 0-based world): `column_index[11] = 1`.

To know what we're equating this column to, we have to look at object 18, the top one above. This is another `Dictionary` but this time the `Dictionary Elements` listed are the rows representing each cell in that column with a value. In this case we see that a `Key` of 20 equals a `Value` of 19. Last time, I promise you, let's go pull those objects:

```
3 Mergeable Data Table Object = mergeable_data_table_object:
    10 Table Note = note:
        2 Note Text = "3x3 middle"
```

```
            5 Attribute Run = attribute_run:
                1 Length = 10
        3 Mergeable Data Table Object = mergeable_data_table_object:
            13 Table Map = mergeable_data_table_custom_map:
                1 Type = 2
                3 Map Entry = map_entry:
                    1 Key = 4
                    2 Value = object_id:
                        2 Unsigned Integer Value = 5
```

Taking the `Key` of 20 first, the bottom entry is very obviously saying that the UUID in index 5 is our Key. Well, the UUID in index 5 is one of the ones we know to refer to the second row: `row_index[5] = 1`. This puts our target dead center of the 3x3 table because `row_index[5] = 1` and `column_index[11] = 1`, but what is it?

Looking at the `Value` field of 19 we see… another Note! And this time, it's a real one! We can quickly see that the text for this field is "3x3 middle" which seems ironic until you know this was all contrived to test a theory.

With that, we now know how to properly display the Note given at the start:

Table title

	3x3 middle	

After the table

Stitching it All Together

Larger tables obviously have more rows and columns, you'll have to do a *lot* more object lookups. They'll certainly have a lot more cells to look up, but as you repeat that last step over all the cells, looking up each row and column in your lookup table, it all will fall into place. If you follow these steps, you'll be able to pull out things many others wouldn't even know existed. I am linking to the actual methods that do each of the below steps in case reading code is more how you learn.

- Build your key items
- Buld your type items
- Build your UUID items
- Use the above to identify your row translations
- Use the above to identify your column translations
- Use all of the above to parse your cells by…
    - Looping over each column `Dictionary Element` …

- Then looping over each row `Dictionary Element` ...
  - Then putting specific Note text into a specific cell

Conclusion

Thanks for sticking with me through all that, I struggled to find the right ways to explain it and hope this was at least somewhat clearer than busting into that protobuf yourself. I hope it is useful knowledge for the forensic examiner to understand how to view this data if they can't actually load the Notes database onto their phone because there is no way any of this information could be accidentally found or properly understood in context. If nothing else, this will be useful for me in 6 months when I'm trying to remember exactly why I did what I did.

Previous
‹ **Revisiting Apple Notes (2):...**

Next
Revisiting Apple Notes (4):... ›

© 2020 Ciofeca Forensics. All rights reserved.