CIOFECA FORENSICS

MOBILE APPLE NOTES

Revisiting Apple Notes (5): Encrypted Notes

31 Jul 2020 · 29 mins read

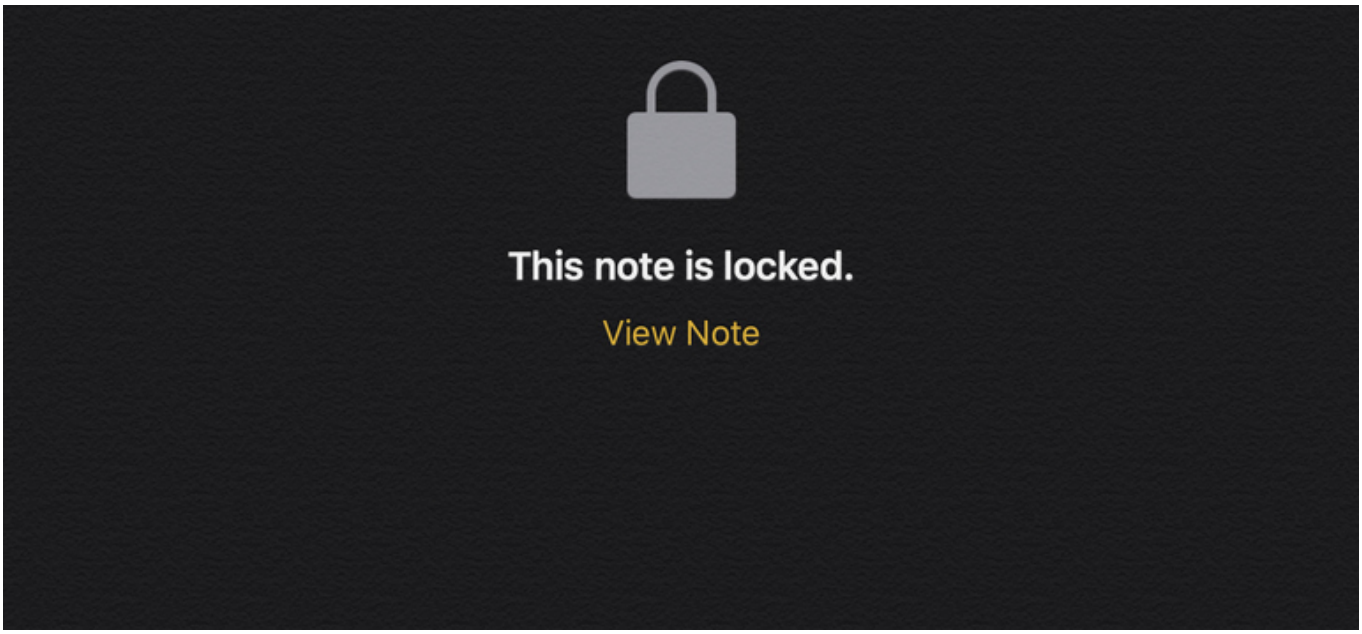
TL;DR: Apple Notes allows users to encrypt note contents at rest and the Apple Cloud Notes Parser now supports parsing of encrypted content.

Background

Apple Notes has allowed users to encrypt their note's contents at rest in the NoteStore database since iOS 9.3. While some commercial forensics tools can unlock notes, I am unaware of free, open source tools in the community which do so and adding this functionality was a challenge posed to me by Heather Mahalik a few years ago during a SANS FOR585. The foundations which Apple uses to carry out the encryption are well documented standards, but initial attempts at putting them together when the parser was written in Perl failed. The struggles with debugging are what led to completely rewriting the parser into an object-oriented language in late 2019 and early 2020. With the parser rewritten, each of the foundational blocks could be implemented discretely and built into the overall program, leading to decryption of encrypted notes in July of 2020.

This article is a detailed look at what is encrypted, how it is encrypted, and how to decrypt it. If you are just looking at Apple notes for the first time, I would recommend starting with the other entries in the Apple Notes category to understand how it works under normal circumstances before tackling encrypted notes. Throughout this article, I will mainly refer to the `ZICCLOUDSYNCINGOBJECT` table when I reference the NoteStore.sqlite database and hence will not be writing that table name every time. I will intentionally include a table name when referencing other tables, such as `ZICNOTEDATA.ZDATA`.

CIOFECA FORENSICS



Password Recovery

It must be said from the outset that the Apple Cloud Notes Parser is not intended to be a password cracker for Apple Notes. This software is to be used to backup or recover notes which you legally have the password to. With that said, Apple's documentation says that "if you forgot your password, Apple can't help you regain access to your locked notes." While true, if you do not have the password for the encrypted note, but do still have the database, you could use a program like HashCat or John the Ripper to potentially recover it. Both of those programs support password recovery on Apple Notes and Apple File System (APFS) encryption. The rest of this article assumes you either created the note and know the password for the note or have a legal reason for reading the note and have the password.

Face ID and Touch ID can also be used to unlock notes, but these do not change the underlying password. All they do is unlock the password you set up in Settings->Notes->Password and enter it automatically, so the password is still able to be recovered from the database. .

Decrypting with Apple Cloud Notes Parser

While I hope the detailed explanations below allow many others to implement note decryption, Apple Cloud Notes Parser aims to make it easy. To tell the parser which passwords to try, a new argument has been added which expects a file path pointing to a file with one password on each line.

```
-w, --password-file FILE File with plaintext passwords, one per line.
```

For example:

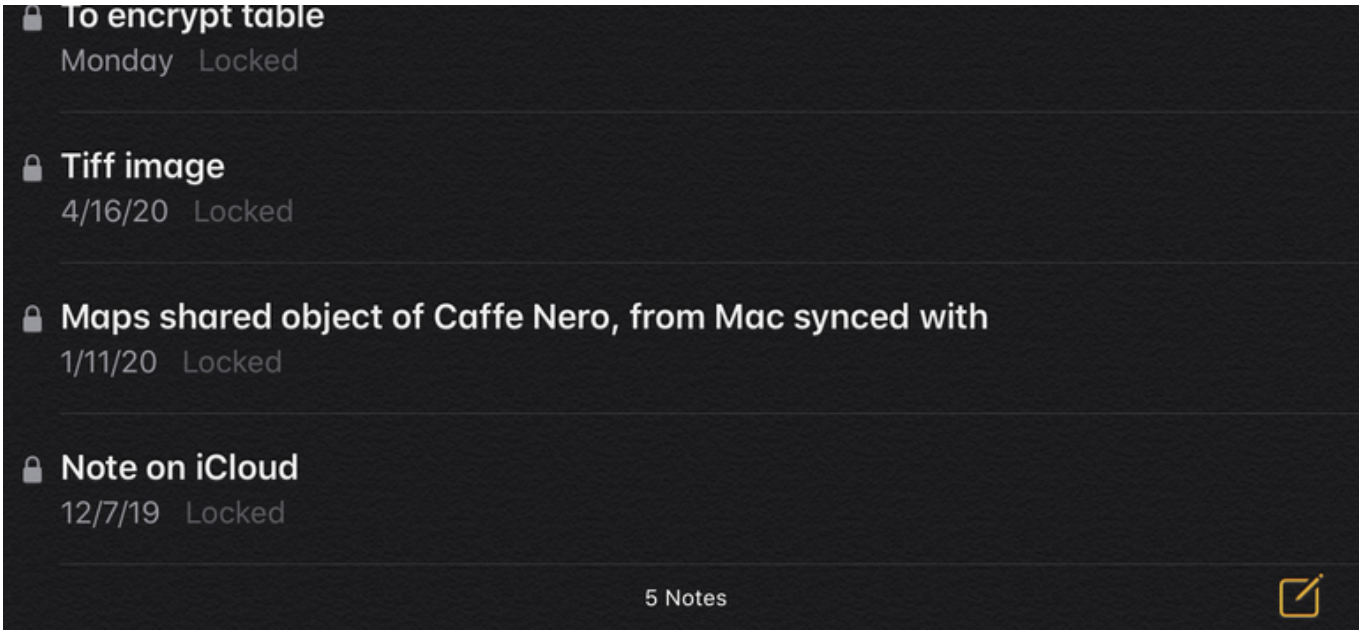
CIOFECA FORENSICS

```
password
password1

notta@cuppa ~/apple_cloud_notes_parser $ ruby notes_cloud_ripper.rb --itunes-dir ~/p

Starting Apple Notes Parser at Wed Jul 29 19:48:43 2020
Storing the results in ./output/2020_07_29-19_48_43

Created a new AppleBackup from iTunes backup: /home/notta/phone_rips/iphone/notes_20
Guessed Notes Version: 13
Guessed Notes Version: 8
Added 4 passwords to the AppleDecrypter from /home/notta/apple_could_notes_parser/pa
Updated AppleNoteStore object with 70 AppleNotes in 11 folders belonging to 2 accour
Updated AppleNoteStore object with 0 AppleNotes in 1 folders belonging to 1 accounts
Adding the ZICNOTEDATA.ZPLAINTEXT and ZICNOTEDATA.ZDECOMPRESSEDDATA columns, this ta

Successfully finished at Wed Jul 29 19:48:48 2020
```

What is Encrypted?

When I say that users can encrypt their note's contents, I need to be careful to be precise. For the most part, what is encrypted is what is found in the `ZICNOTEDATA.ZDATA` column in the NoteStore.sqlite database. This leaves most of the note metadata unencrypted, and even some of the content (specifically the `ZTITLE1` column) unencrypted. Aside from the `ZICNOTEDATA.ZDATA` column, contents of attached files end up encrypted and some of the metadata about objects ends up encypted in the `ZENCRYPTEDVALUESJSON` column, such as filenames and URLs. Notice in the below screenshot how, even though the notes are all locked, you can see the day they were created and the title.

CIOFECA FORENSICS



According to Apple[1], users can encrypt notes containing "images, sketches, tables, maps, and websites." Each of those functions slightly differently in how they encrypt, but the general rules I've seen are:

1. If the normal version of the attachment writes to disk, such as an image or a thumbnail of a webpage, that file will contain encrypted data
2. If the normal version of the attachment has a user-generated filename, such as an image, that filename will be encrypted and stored in the `ZENCRYPTEDVALUESJSON` column but the file will use the `ZIDENTIFER` column as its filename
3. If the normal version of the attachment has a Notes-generated filename, such as a sketch, that filename will remain the same
4. If the normal version of the attachment uses the `ZMERGEABLEDATA1` column, such as a table, that blob will be encrypted and stored in the `ZENCRYPTEDVALUESJSON` column
5. Files are encrypted on disk using the `ZASSETCRYPTOTAG` and `ZASSETCRYPTOINITIALIZATIONVECTOR` columns
6. Fallback images, such as for sketches, are encrypted on disk using the `ZFALLBACKIMAGECRYPTOTAG` and `ZFALLBACKIMAGECRYPTOINITIALIZATIONVECTOR` columns

How is it Encrypted?

Like much of Apple's products, specific details on the inner workings of Apple Notes are hard to come by and they use functionality that is not part of the larger libraries Apple makes available to most developers[2]. The bulk of what is officially known comes from a brief blurb that has been copied over a few different Apple pages over the years[1]. The blurb describes the underlying foundation of their encryption, but doesn't get quite far enough for someone unfamiliar with the implementation of these foundations to implement it on their own. That is possibly why this

CIOFECA FORENSICS

> When a user secures a note, a 16-byte key is derived from the user's passphrase using PBKDF2 and SHA256. The note and all of its attachments are encrypted using AES-GCM. New records are created in Core Data and CloudKit to store the encrypted note, attachments, tag, and initialization vector. After the new records are created, the original unencrypted data is deleted. Attachments that support encryption include images, sketches, tables, maps, and websites. Notes containing other types of attachments can't be encrypted, and unsupported attachments can't be added to secure notes.
>
> ...
>
> To change the passphrase on a secure note, the user must enter the current passphrase, as Touch ID and Face ID aren't available when changing the passphrase. After choosing a new passphrase, the Notes app rewraps the keys of all existing notes in the same account that are encrypted by the previous passphrase.

What Changes During Encryption?

During testing of how to decrypt, we wanted to see what exactly happened when a user encrypted a note. I used a MacOS computer to create a note, copied the NoteStore.sqlite database off, then encrypted the note and copied it off again. This is the output of sqldiff'ing the files:

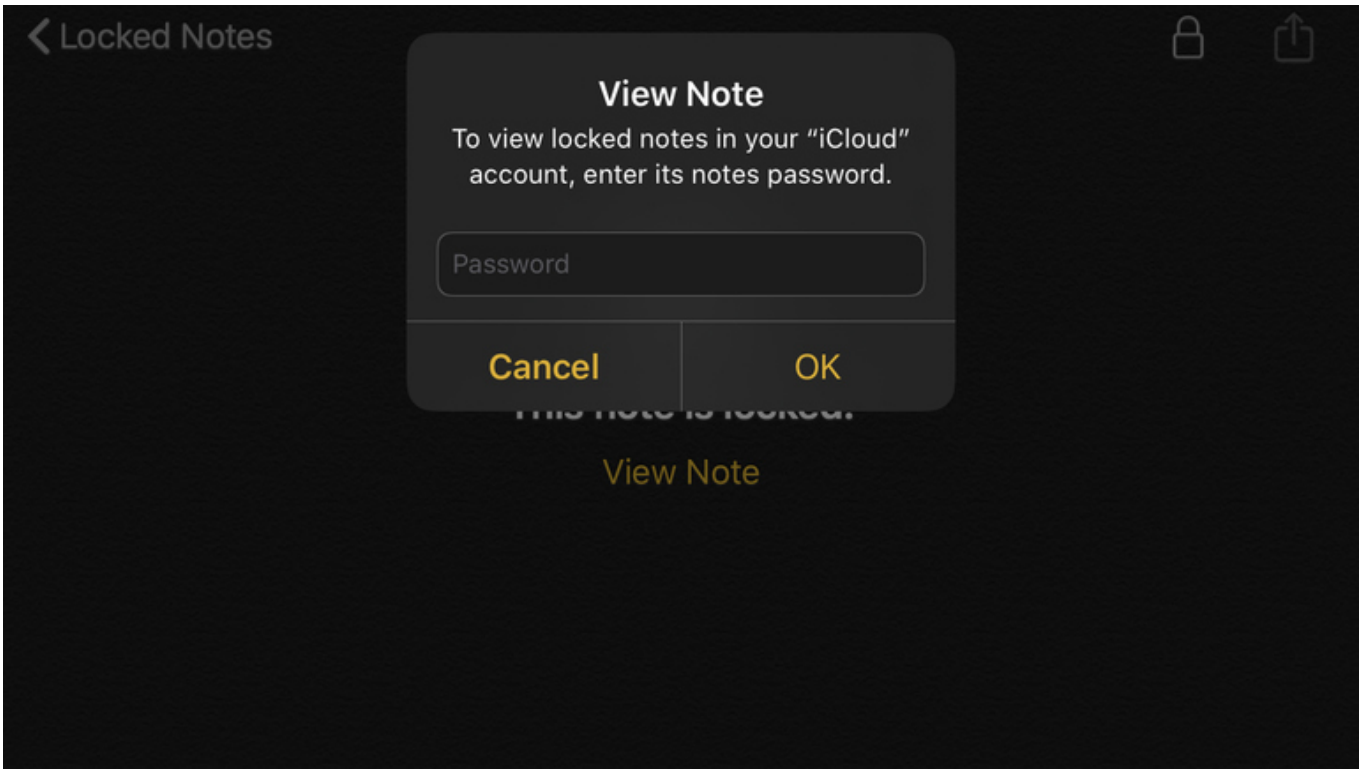
```
notta@cuppa ~/notestores $ sqldiff MacNoteStore_preencryption.sqlite MacNoteStore_pc
UPDATE ZICCLOUDSYNCINGOBJECT SET Z_OPT=8, ZMARKEDFORDELETION=1, ZSNIPPET=NULL, ZTITL
UPDATE ZICCLOUDSYNCINGOBJECT SET Z_OPT=4 WHERE Z_PK=123;
INSERT INTO ZICCLOUDSYNCINGOBJECT(Z_PK,Z_ENT,Z_OPT,ZCRYPTOITERATIONCOUNT,ZISPASSWORD
UPDATE ZICNOTEDATA SET Z_OPT=4, ZDATA=x'1f8b0800000000000013e36010ea67e460106090ea60
INSERT INTO ZICNOTEDATA(Z_PK,Z_ENT,Z_OPT,ZNOTE,ZCRYPTOINITIALIZATIONVECTOR,ZCRYPTOTA
```

The note I created initially was `ZICCLOUDSYNCINGOBJECT.Z_PK=121` and `ZICNOTEDATA=41` so you can see how Apple marks the note for deletion and clears out the fields which might have information that should be protected. You can also see the overwriting of `ZICNOTEDATA.ZDATA` with a new protobuf and the insertion of a new entry into `ZICCLOUDSYNCINGOBJECT` and `ZICNOTEDATA` with the encrypted information.

How is it Decrypted?

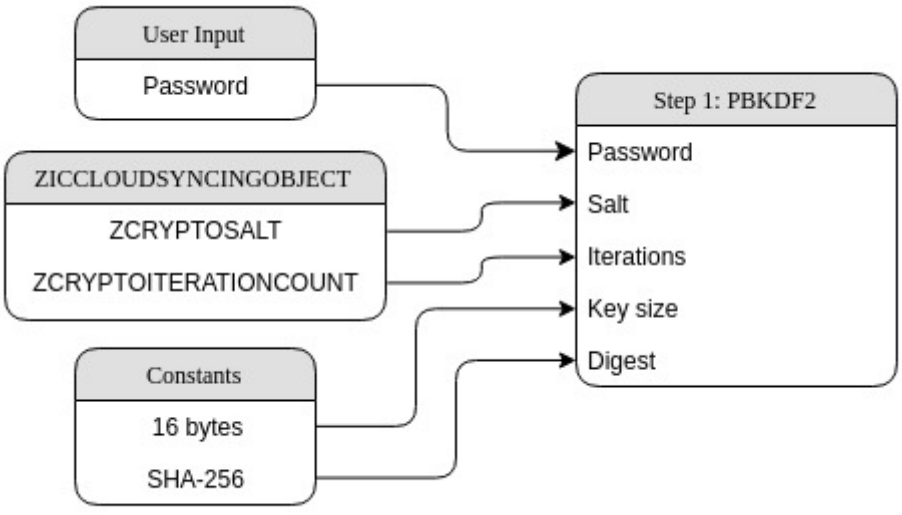
The above probably is more than sufficient for someone that knows cryptography, but for the layman such as myself, I found I needed a lot of time reading the RFCs and looking at common implementations to understand it. Below I will describe the three basic steps that are taken to

CIOFECA FORENSICS



Step 1: Derive the Password-Based Key

Apple makes the starting point clear in their description, the use of Password-Based Key Derivation Function 2 (PBKDF2) with the SHA-256 digest algorithm to make a 16-byte key. However, PBKDF2 requires two other parameters to run, the number of iterations and the password salt. Both of those values are found in the `ZICCLOUDSYNCINGOBJECT` table, in the obviously named `ZCRYPTOITERATIONCOUNT` [3] and `ZCRYPTOSALT` columns.



This SQLite query would let you would pull the necessary values out of the database to carry out step 1 for all encrypted notes:

CIOFECA FORENSICS

```
FROM ZICCLOUDSYNCINGOBJECT
WHERE ZICCLOUDSYNCINGOBJECT.ZISPASSWORDPROTECTED=1
```

PK	Salt (in hex)	Iterations
17	1165106b6b288bda1e6ecb18e65c7876	20000

In Ruby, this is what a method might look like[4] if you wanted to pass in the user's password and the crypto_salt and crypto_iterations from the database to derive the key:

```
require 'openssl'

def derive_key(password, crypto_salt, crypto_iterations)
  return OpenSSL::PKCS5.pbkdf2_hmac(password,
                                    crypto_salt,
                                    crypto_iterations,
                                    16, # Apple uses 16-byte keys
                                    OpenSSL::Digest::SHA256.new) # Apple uses SHA-25
end

key_encrypting_key = derive_key("password",
                                "\x11\x65\x10\x6b\x6b\x28\x8b\xda\x1e\x6e\xcb\x18\xe
                                20000)
# key_encrypting_key => "\x65\x2b\xe8\x61\x43\x34\x8a\x6e\x01\x06\x09\x58\x80\xbc\xf
```

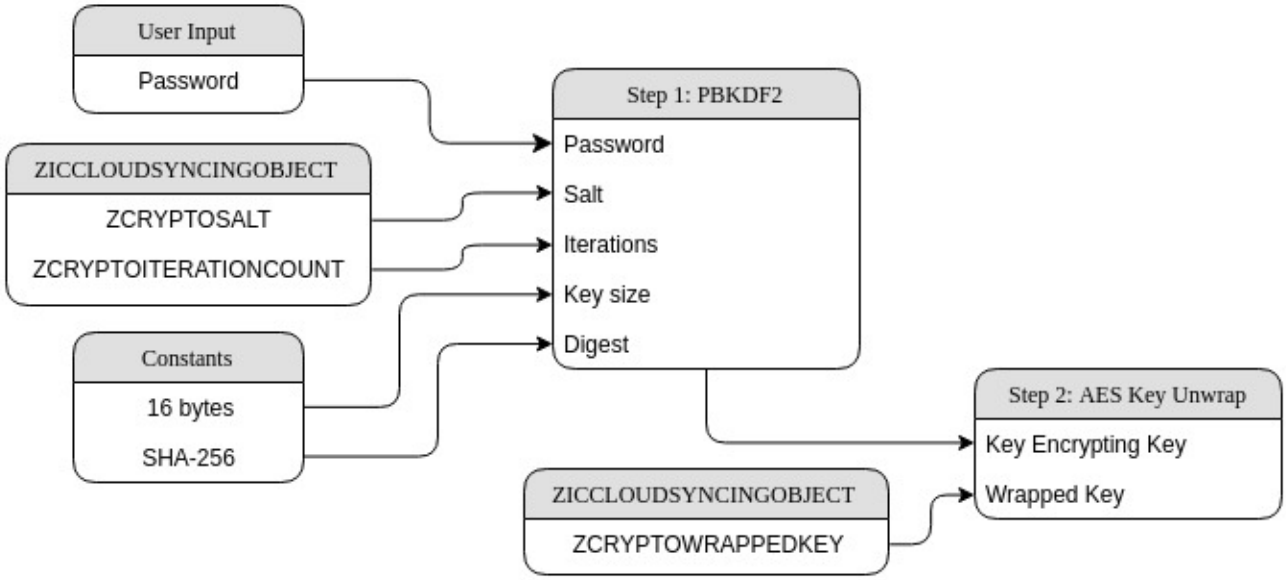
Step 2: Unwrap the Encryption Key

In the last part of the quoted text above, Apple says it "rewraps" all of the keys for existing notes when you change the password. This indicates that, similar to the APFS protections, Apple isn't reencrypting all the data, they simply are rewrapping the key to decrypt the data. There are advantages to this, including being able to "delete" data very quickly by simply deleting the decryption key. One downside is a layman such as myself could easily waste some time trying to "decrypt" the key, instead of "unwrap" it[5]. Another "downside" (from Apple's perspective) or advantage (from the perspective of someone who lost their password) is that it is relatively easy to try offline password attacks if you have the database, since you never have to actually decrypt all the data, just unwrap the key.

To unwrap the decryption key, you need an implementation of the AES Key Wrap algorithm. Do *not* use AES-GCM, even though it is mentioned in the Apple specs as that is not a key wrapping algorithm and will take you down many, wrong, rabbit holes[5]. Do *not* be confused with the 24-byte wrapped key in your database when the Apple blurb says there should be a 16-byte key, the AES Key Wrap adds an extra 8-bytes on to the key material during wrapping.

CIOFECA FORENSICS

1 using PBKDF2.



This SQLite query would let you would pull the necessary values out of the database to carry out step 2 for all encrypted notes:

```
SELECT ZICCLOUDSYNCINGOBJECT.Z_PK,
  ZICCLOUDSYNCINGOBJECT.ZCRYPTOWRAPPEDKEY
FROM ZICCLOUDSYNCINGOBJECT
WHERE ZICCLOUDSYNCINGOBJECT.ZISPASSWORDPROTECTED=1
```

PK	Wrapped Key (in hex)
17	98c0e56b43b507e60c5465ec5e1bb0c74b756f7d4f4a9bff

Building on the Ruby example above, this is how one could use the `aes_key_wrap` gem to unwrap the wrapped key:

```
require 'aes_key_wrap'

# From Step 1
key_encrypting_key = "\x65\x2b\xe8\x61\x43\x34\x8a\x6e\x01\x06\x09\x58\x80\xbc\xf3\x

def unwrap_key(wrapped_key, key_encrypting_key)
  return AESKeyWrap.unwrap(wrapped_key, key_encrypting_key)
end

unwrapped_key = unwrap_key("\x98\xc0\xe5\x6b\x43\xb5\x07\xe6\x0c\x54\x65\xec\x5e\x1b
                           key_encrypting_key)
```

CIOFECA FORENSICS

Side Quest 2.5: Get the Right Library

While I wasted a lot of time learning the difference between an AES Key Wrap and AES-GCM in step 2, a very specific Ruby versioning issue burned about a week of my time for step 3. My development box has been around a long time and it still runs Ruby 2.3 and if you read the main page for the OpenSSL gem it clearly says:

> NOTE: If you are using Ruby 2.3 (and not Bundler), you must activate the gem version of openssl, otherwise the default gem packaged with the Ruby installation will be used

The layman who glossed over that warning to get into the specifics for the gem would likely pay for that mistake later[5]. While the first two steps worked fine, step 3 kept generating bad decrypts and throwing an `OpenSSL::Cipher::CipherError` because the authentication tag seemed not to be right. My troubleshooting involved all sorts of crazy changes, odd assumptions about maybe Apple using this value instead of that value, and wasting hours of a good friend's time trying to get the code to work. Finally, I was going down the path of openssl being too old and yet again was googling for specific Ruby and OpenSSL issues when one of the cached pages had the note about Ruby 2.3 on it. As soon as I actually read that warning and added the appropriate line to the AppleDecrypter class to explicitly used the OpenSSL gem instead of the built-in library, things worked immediately.

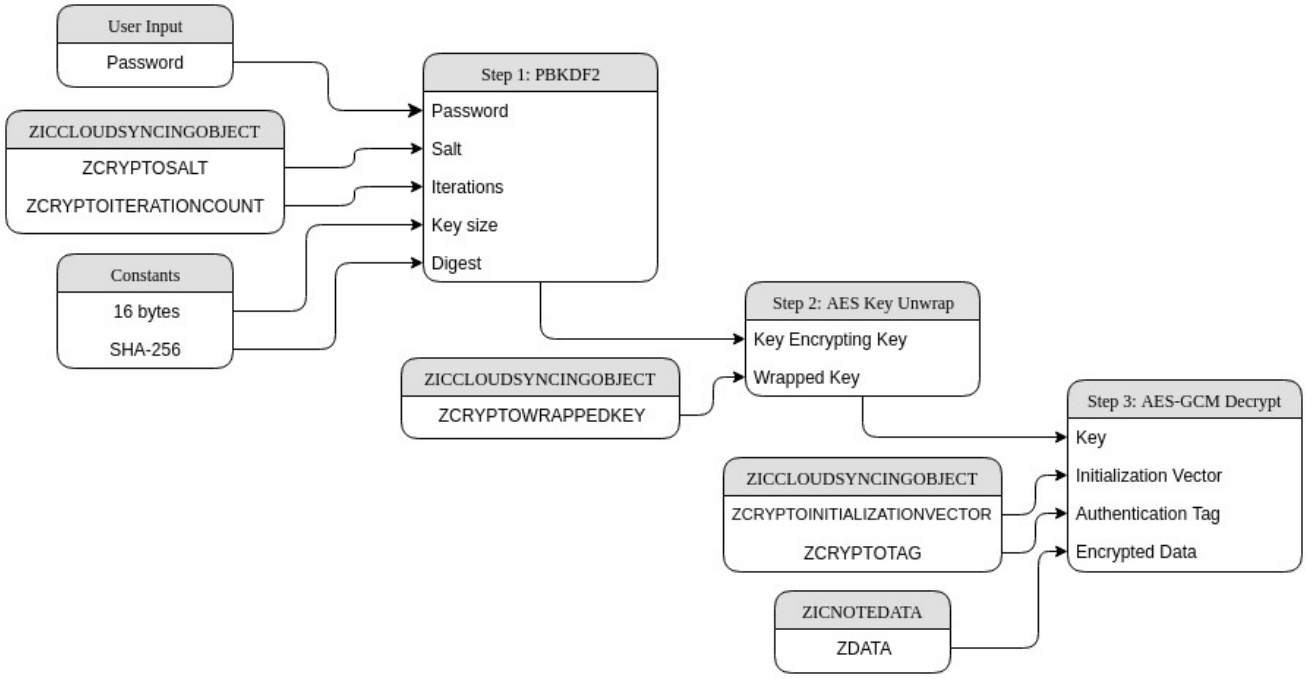
The most important thing you can look at if you implement this in another language is the IV size. My code was failing because the AES-GCM specifications state an IV of 96-bits should be used, but the IV in Apple Notes is 128-bits. This would not be a problem except the built-in openssl library in Ruby's standard libraries from Ruy 2.3 appears to only take the first 12-bytes of the IV if you give it a longer value. Whatever library or language you use, make sure you can tell your algorithm that you are giving it a 128-bit IV. Using only part of the IV will, guaranteed, waste a week of your life[5].

Step 3: Decrypt the Note

The third and final step of decrypting is to actually decrypt the content. Per Apple's blurb, the note content is encrypted with AES-GCM. Apple says it stores the encrypted note, the note's attachments, the tag, and the initialization vector before deleting the original note. Functionally, what that means in the database is that a new entry is created in `ZICNOTEDATA` and `ZICCLOUDSYNCINGOBJECT` for a new note (as well as for any attachments) and the old ones are "marked for deletion."[6] Even though the row is not immediately deleted (just marked as such for future cleanup), the `ZICNOTEDATA.ZDATA` column for the original note is overwritten as well.

CIOFECA FORENSICS

ZICNOTEDATA table, and appear to be consistent in both places.



This SQLite query would let you would pull the necessary values out of the database to carry out step 3 for all encrypted notes:

```
SELECT ZICCLOUDSYNCINGOBJECT.Z_PK,
  ZICCLOUDSYNCINGOBJECT.ZCRYPTOINITIALIZATIONVECTOR,
  ZICCLOUDSYNCINGOBJECT.ZCRYPTOTAG,
  ZICNOTEDATA.ZDATA
FROM ZICCLOUDSYNCINGOBJECT, ZICNOTEDATA
WHERE ZICCLOUDSYNCINGOBJECT.ZISPASSWORDPROTECTED=1 AND
  ZICNOTEDATA.ZNOTE=ZICCLOUDSYNCINGOBJECT.ZCLOUDSTATE
```

PK	IV (in hex)	Tag (in hex)
17	151f64de7be34d15dacdaea9b33471f9	806bf2bbd3bf83cf1240b03e7c4d6ab1

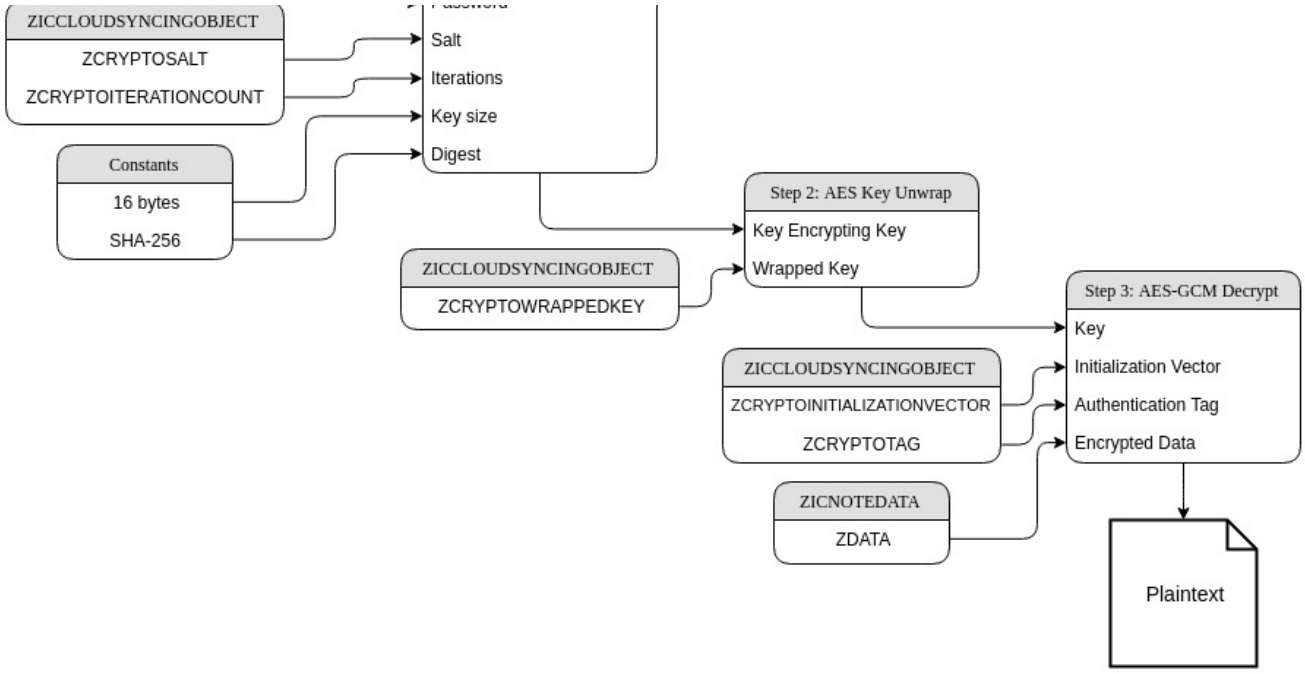
PK	ZDATA (in hex)
17	131b03571fc9ec47ef58e58e21fce5c10aa73a62b9e58a743bcdcc3aff1ea8ab 9964f4535b8597735f3da5f6ae63b9370625a20d633e9cf2986d4d118989124f 0ddfee956e47cb5cbc3617c520b075620b37ae4056f3a1af83351fda634dfb44 6055c75f7143a5600149db333893c0ecb0ef3944e2a64542e9a4375bf1526898 58fed8b21aded0eab0afb11190

CIOFECA FORENSICS

```
gem 'openssl'
require 'openssl'

# From Step 2
unwrapped_key = "\x02\x3a\xae\x7c\x45\x0a\x28\x3b\x23\xe3\xd7\xc1\x41\x6a\xd6\x44"

def decrypt_aes_gcm(key, iv, tag, data)
  decrypter = OpenSSL::Cipher.new('aes-128-gcm').decrypt
  decrypter.iv_len = 16
  decrypter.key = key
  decrypter.iv = iv
  decrypter.auth_tag = tag
  plaintext = decrypter.update(data) + decrypter.final
end

iv = "\x15\x1f\x64\xde\x7b\xe3\x4d\x15\xda\xcd\xae\xa9\xb3\x34\x71\xf9"
tag = "\x80\x6b\xf2\xbb\xd3\xbf\x83\xcf\x12\x40\xb0\x3e\x7c\x4d\x6a\xb1"
encrypted_data = "\x13\x1b\x03\x57\x1f\xc9\xec\x47\xef\x58\xe5\x8e\x21\xfc\xe5\xc1"
                 "\x0a\xa7\x3a\x62\xb9\xe5\x8a\x74\x3b\xcd\xcc\x3a\xff\x1e\xa8\xab"
                 "\x99\x64\xf4\x53\x5b\x85\x97\x73\x5f\x3d\xa5\xf6\xae\x63\xb9\x37"
                 "\x06\x25\xa2\x0d\x63\x3e\x9c\xf2\x98\x6d\x4d\x11\x89\x89\x12\x4f"
                 "\x0d\xdf\xee\x95\x6e\x47\xcb\x5c\xbc\x36\x17\xc5\x20\xb0\x75\x62"
                 "\x0b\x37\xae\x40\x56\xf3\xa1\xaf\x83\x35\x1f\xda\x63\x4d\xfb\x44"
                 "\x60\x55\xc7\x5f\x71\x43\xa5\x60\x01\x49\xdb\x33\x38\x93\xc0\xec"
                 "\xb0\xef\x39\x44\xe2\xa6\x45\x42\xe9\xa4\x37\x5b\xf1\x52\x68\x98"
                 "\x58\xfe\xd8\xb2\x1a\xde\xd0\xea\xb0\xaf\xb1\x11\x90"

plaintext = decrypt_aes_gcm(unwrapped_key, iv, tag, encrypted_data)

# plaintext = "\x1f\x8b\x08\x00\x00\x00\x00\x00\x00\x13\xe3\x60\x10\x9a" +
#             "\xc1\xc8\xc1\x20\xc0\x20\x35\x91\x51\x48\xde\x35\x2f\xb9" +
#             "\xa8\xb2\xa0\x24\x35\x45\xa1\x24\xb3\x24\x27\x95\x8b\x0b" +
#             "\x21\x90\x94\x9f\x52\x29\x25\xc0\xc5\x02\x52\x0b\x54\x0d" +
#             "\xa6\x35\x18\xc1\x22\x8c\x40\x11\x79\x29\x30\xad\xc1\x24" +
#             "\x25\xc6\xc5\x01\x94\xfb\x0f\x04\xfc\x40\x75\x70\xb6\x92" +
#             "\x0c\x97\x14\x97\xc0\xbb\x7f\x02\xb7\xa2\x2a\x9d\x55\x3b" +
#             "\x76\xe5\x9e\x7a\xf4\x72\xfb\x1b\x21\x26\x0e\x79\x20\x66" +
#             "\xd4\xe2\xe0\x10\x10\x02\x9a\x29\xc1\xa8\x05\xe2\xb1\x71" +
#             "\xf0\x09\x31\x49\x30\x02\x00\xd1\x69\x5a\x2d\x9d\x00\x00\x00"
```

Look at that output! `0x1f 0x8b`, that's the start of a GZip file! At this point, our normal processing could take over as we would be expecting a GZipped protobuf in the `ZICNOTEDATA.ZDATA` column.

CIOFECA FORENSICS



Step 4: Repeat

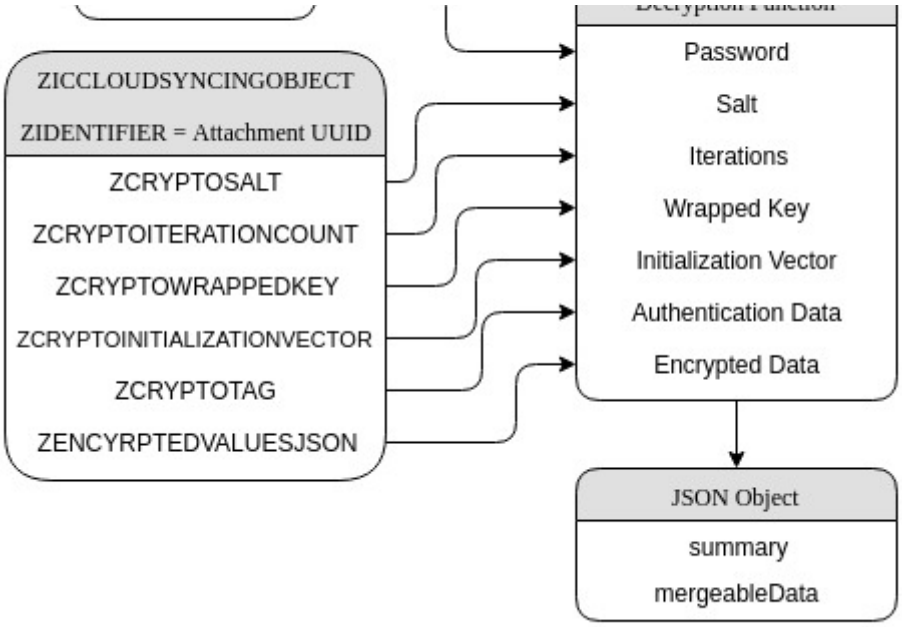
Ok, I lied, our normal processing can't quite take over yet because as it parses the protobuf it will start to hit encrypted attachments and run into problems. As you hit attachments in an encrypted note, you'll need to do the same 3 steps above for that object. Each type of attachment, as noted at the top is different in how you have to decryp it. The attachment will have different cryptographic variables from the note, but the same underlying password. Here are brief rundowns of how each encrypted attachment behaves:

How are Attachments Handled?

Tables

Tables are the most straight forward as all they do is take the value that would normally be in `ZMERGEABLEDATA1`, base64 encoded it, put it into a JSON object along with the value that used to be in the `ZSUMMARY` column, encrypt that JSON, and insert it into the `ZENCRYPTEDVALUESJSON` column. The same row will have all of your cryptographic settings and the content.

CIOFECA FORENSICS




```
{
  "summary":"This\nIs\nFantastic\nEncryption\n",
  "mergeableData":"H4sIAAAAAAAAE7VU30/TUBReu63r7oaUwvhx5YWCc6mBYE2M8Q0HKMjGLEONiQ+ju
}
```

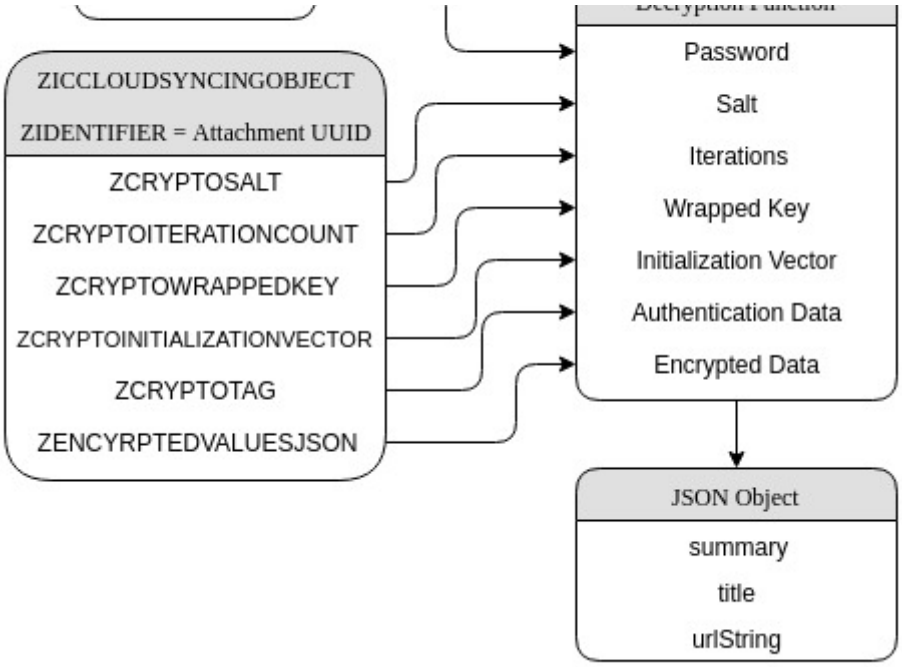

The right SQL statement to pull all the variables necessary to decrypt this object is:

```
SELECT ZICCLOUDSYNCINGOBJECT.ZCRYPTOINITIALIZATIONVECTOR, ZICCLOUDSYNCINGOBJECT.ZCRY
  ZICCLOUDSYNCINGOBJECT.ZCRYPTOSALT, ZICCLOUDSYNCINGOBJECT.ZCRYPTOITERATIONCOUNT,
  ZICCLOUDSYNCINGOBJECT.ZCRYPTOVERIFIER, ZICCLOUDSYNCINGOBJECT.ZCRYPTOWRAPPEDKEY,
  ZICCLOUDSYNCINGOBJECT.ZENCRYPTEDVALUESJSON
FROM ZICCLOUDSYNCINGOBJECT
WHERE ZICCLOUDSYNCINGOBJECT.ZIDENTIFIER="[your table's UUID]"
```

URLs

URLs are also fairly easy and behave much like tables in that they have the sensitive information in a JSON object in the `ZENCRYPTEDVALUESJSON` column. The exception is instead of having mergeable data they have the actual URL with its summary and title.

CIOFECA FORENSICS




```
{
  "summary":"Washington St\nWellesley Hills MA 02481\nUnited States",
  "title":"Caffè Nero",
  "urlString":"https://maps.apple.com/?address=339%20Washington%20St,%20Wellesley%20
}
```

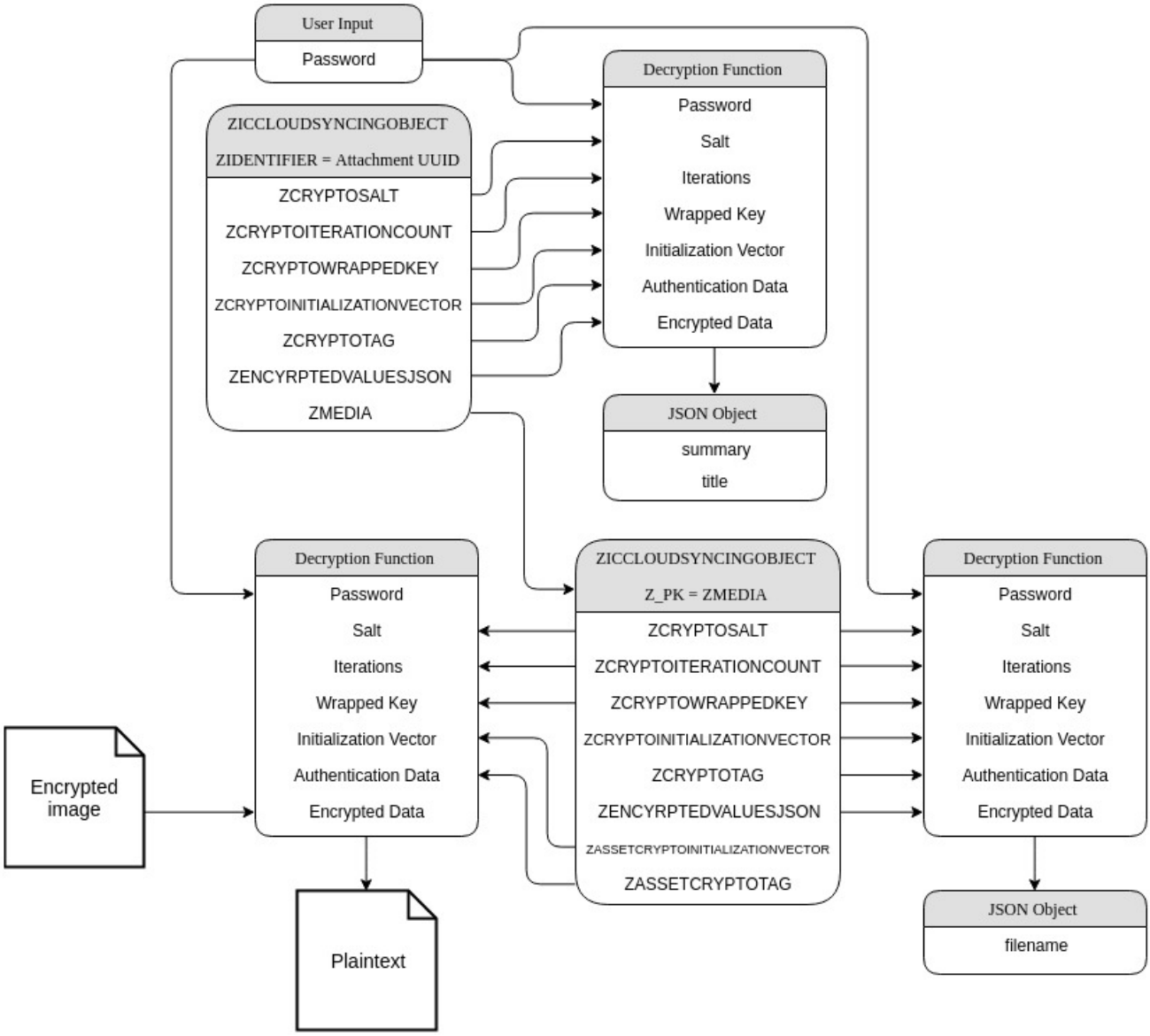

The right SQL statement to pull all the variables necessary to decrypt this object is:

```
SELECT ZICCLOUDSYNCINGOBJECT.ZCRYPTOINITIALIZATIONVECTOR, ZICCLOUDSYNCINGOBJECT.ZCRY
  ZICCLOUDSYNCINGOBJECT.ZCRYPTOSALT, ZICCLOUDSYNCINGOBJECT.ZCRYPTOITERATIONCOUNT,
  ZICCLOUDSYNCINGOBJECT.ZCRYPTOVERIFIER, ZICCLOUDSYNCINGOBJECT.ZCRYPTOWRAPPEDKEY,
  ZICCLOUDSYNCINGOBJECT.ZENCRYPTEDVALUESJSON
FROM ZICCLOUDSYNCINGOBJECT
WHERE ZICCLOUDSYNCINGOBJECT.ZIDENTIFIER="[your URL's UUID]"
```

Images

Images are a bit harder to deal with because of how many rows they go across in the `ZICCLOUDSYNCINGOBJECT` table. You'll have the attachment's row, then a row for the media, then potentially a lot of rows for thumbnails. While the image data is encrypted within the file on disk, the image's filename is kept in the `ZENCRYPTEDVALUESJSON` column on the media's row. Importantly, the correct cryptographic settings to decrypt the contents within the file on disk are in the `ZASSETCRYPTOTAG` and `ZASSETCRYPTOINITIALIZATIONVECTOR` columns for the media

CIOFECA FORENSICS



On the Image object itself

The right SQL statement to pull all the variables necessary to decrypt this object's JSON is:

```
SELECT ZICCLOUDSYNCINGOBJECT.ZCRYPTOINITIALIZATIONVECTOR, ZICCLOUDSYNCINGOBJECT.ZCRY
  ZICCLOUDSYNCINGOBJECT.ZCRYPTOSALT, ZICCLOUDSYNCINGOBJECT.ZCRYPTOITERATIONCOUNT,
  ZICCLOUDSYNCINGOBJECT.ZCRYPTOVERIFIER, ZICCLOUDSYNCINGOBJECT.ZCRYPTOWRAPPEDKEY,
  ZICCLOUDSYNCINGOBJECT.ZENCRYPTEDVALUESJSON
FROM ZICCLOUDSYNCINGOBJECT
WHERE ZICCLOUDSYNCINGOBJECT.ZIDENTIFIER="[your image's UUID]"
```

```
{
  "summary":"    ",
  "title":"CF-Favicon.tiff"
}
```

CIOFECA FORENSICS

The right SQL statements to pull all the variables necessary to decrypt the actual file on disk and the JSON is as follows. Note, the image's filename will be the `ZIDENTIFIER` from the second query.

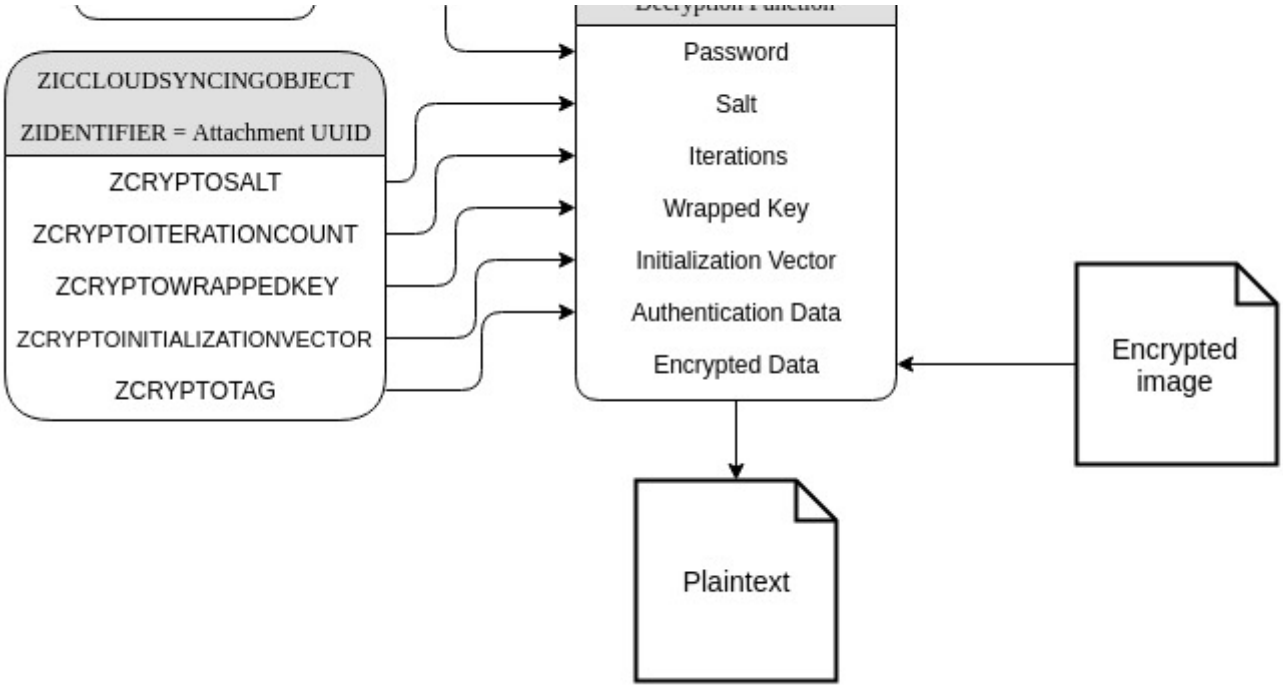
```
SELECT ZICCLOUDSYNCINGOBJECT.ZMEDIA
FROM ZICCLOUDSYNCINGOBJECT
WHERE ZICCLOUDSYNCINGOBJECT.ZIDENTIFIER="[your image's UUID]"

SELECT ZICCLOUDSYNCINGOBJECT.ZASSETCRYPTOINITIALIZATIONVECTOR,
  ZICCLOUDSYNCINGOBJECT.ZASSETCRYPTOTAG,
  ZICCLOUDSYNCINGOBJECT.ZCRYPTOINITIALIZATIONVECTOR, ZICCLOUDSYNCINGOBJECT.ZCRYPTOTA
  ZICCLOUDSYNCINGOBJECT.ZCRYPTOSALT, ZICCLOUDSYNCINGOBJECT.ZCRYPTOITERATIONCOUNT,
  ZICCLOUDSYNCINGOBJECT.ZCRYPTOVERIFIER, ZICCLOUDSYNCINGOBJECT.ZCRYPTOWRAPPEDKEY,
  ZICCLOUDSYNCINGOBJECT.ZENCRYPTEDVALUESJSON, ZICCLOUDSYNCINGOBJECT.ZIDENTIFIER
FROM ZICCLOUDSYNCINGOBJECT
WHERE ZICCLOUDSYNCINGOBJECT.Z_PK=[your ZMedia result above]
```

```
{
  "filename":"CF-Favicon.tiff"
}
```

Thumbnails

After discussing images, thumbnails make a lot more sense. They follow basically the same process, except they only have the one row of information, which simplifies things. Their files on disk are also encrypted, but using the normal `ZCRYPTOTAG` and `ZCRYPTOINITIALIZATIONVECTOR` columns.

CIOFECA FORENSICS



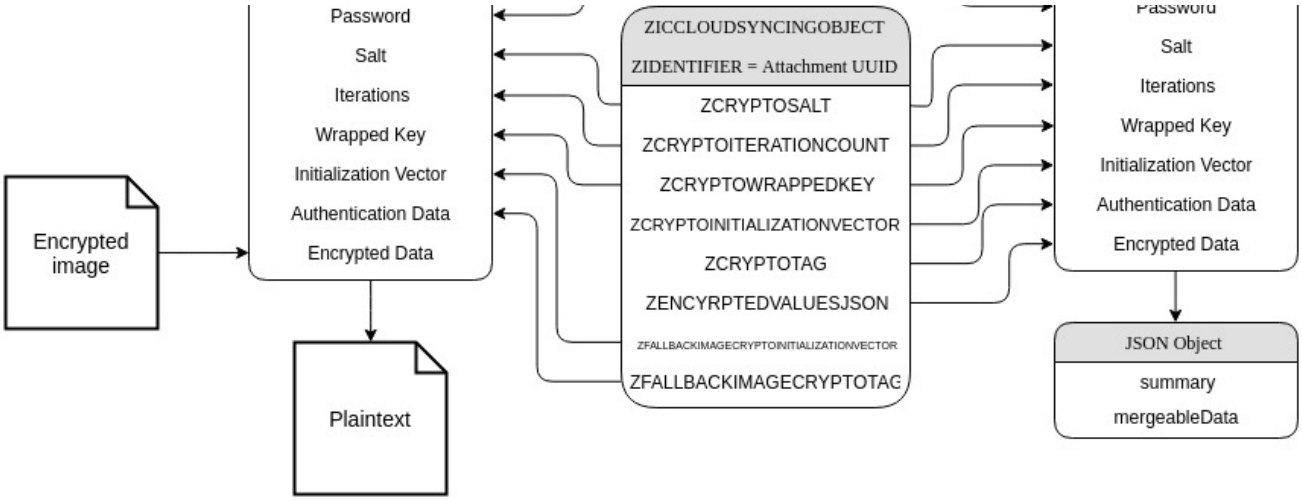
The right SQL statement to pull all the variables necessary to decrypt this object's JSON is:

```
SELECT ZICCLOUDSYNCINGOBJECT.ZCRYPTOINITIALIZATIONVECTOR, ZICCLOUDSYNCINGOBJECT.ZCRY
  ZICCLOUDSYNCINGOBJECT.ZCRYPTOSALT, ZICCLOUDSYNCINGOBJECT.ZCRYPTOITERATIONCOUNT,
  ZICCLOUDSYNCINGOBJECT.ZCRYPTOVERIFIER, ZICCLOUDSYNCINGOBJECT.ZCRYPTOWRAPPEDKEY
FROM ZICCLOUDSYNCINGOBJECT
WHERE ZICCLOUDSYNCINGOBJECT.ZIDENTIFIER="[your thumbnail's UUID]"
```

Sketches

Sketches can be done either like tables parsing a protobuf from `ZENCRYPTEDVALUESJSON`, or images decrypting a file on disk from the `ZMEDIA` row, or both if you want to get everything. The only big difference is when you are parsing a file from disk using fallback images the tag and IV are in the `ZFALLBACKIMAGECRYPTOTAG` and `ZFALLBACKIMAGECRYPTOINITIALIZATIONVECTOR` columns.

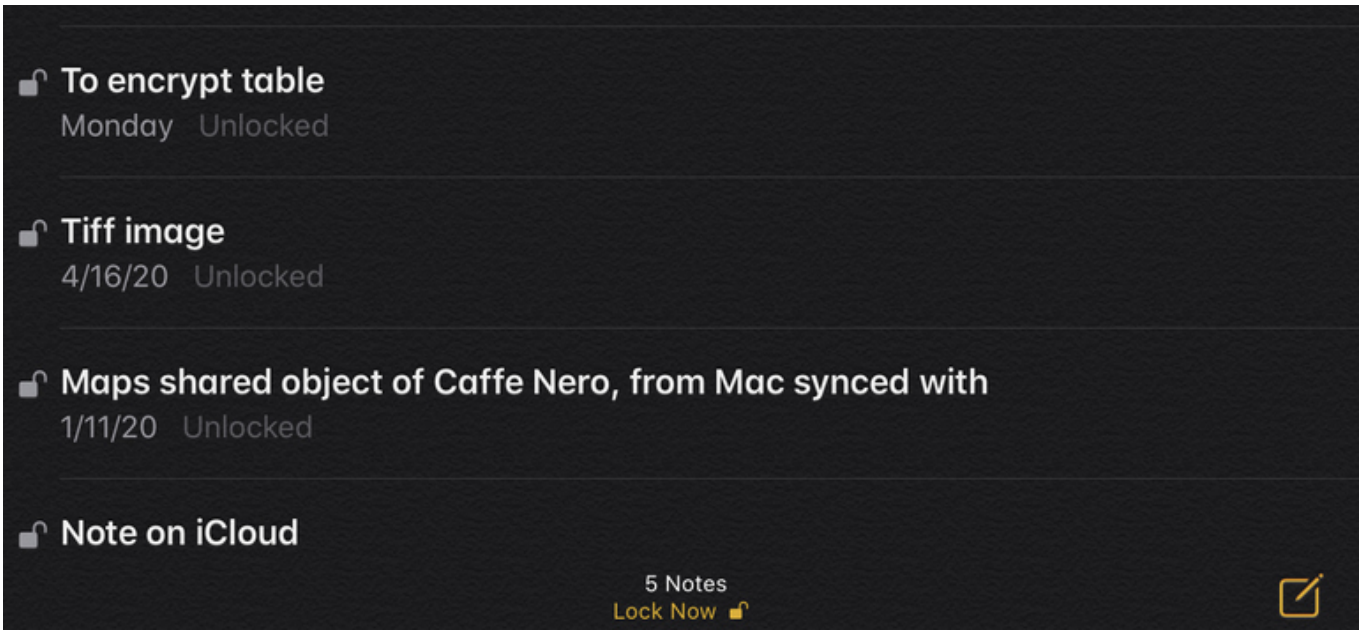
CIOFECA FORENSICS



The right SQL statements to pull all the variables necessary to decrypt the actual file on disk and the JSON are as follows.

```
SELECT ZICCLOUDSYNCINGOBJECT.ZMEDIA
FROM ZICCLOUDSYNCINGOBJECT
WHERE ZICCLOUDSYNCINGOBJECT.ZIDENTIFIER="[your sketch's UUID]"

SELECT ZICCLOUDSYNCINGOBJECT.ZFALLBACKIMAGECRYPTOINITIALIZATIONVECTOR,
  ZICCLOUDSYNCINGOBJECT.ZFALLBACKIMAGECRYPTOTAG,
  ZICCLOUDSYNCINGOBJECT.ZCRYPTOINITIALIZATIONVECTOR, ZICCLOUDSYNCINGOBJECT.ZCRYPTOTA
  ZICCLOUDSYNCINGOBJECT.ZCRYPTOSALT, ZICCLOUDSYNCINGOBJECT.ZCRYPTOITERATIONCOUNT,
  ZICCLOUDSYNCINGOBJECT.ZCRYPTOVERIFIER, ZICCLOUDSYNCINGOBJECT.ZCRYPTOWRAPPEDKEY,
  ZICCLOUDSYNCINGOBJECT.ZENCRYPTEDVALUESJSON, ZICCLOUDSYNCINGOBJECT.ZIDENTIFIER
FROM ZICCLOUDSYNCINGOBJECT
WHERE ZICCLOUDSYNCINGOBJECT.Z_PK=[your ZMedia result above]
```

CIOFECA FORENSICS



Conclusion

Apple's use of well-documented, public algorithms for its encryption of data at rest in the Notes application allows for straight forward decryption of notes, if the password is known. Even if the password is not known, public tools such as John the Ripper can easily recover the password you need to continue your investigation within Apple notes if you have the legal authority to do so. It is my hope that this article is useful for others in implementing the decrypting of Apple notes in other forensics frameworks and for your personal use on your own data.

Footnotes

1. Apple Platform Security, Secure features in Notes app ↩ ↩2

2. For example, Apple Notes have been encrypted with AES-GCM since iOS 9, whereas CryptoKit only picked up that functionality in iOS 13. ↩

3. You'll notice that things which are not encrypted also have `ZCRYPTOITERATIONCOUNT` set, do not use that field to check for encryption, use the `ZISPASSWORDPROTECTED` field instead. ↩

4. While these examples will work in IRB if you have the required gems installed, they obviously are not fully fleshed out, check for errors, etc. See the AppleDecrypter class in the Apple Cloud Notes Parser for a better implementation to steal for other projects. ↩

5. This happened. ↩ ↩2 ↩3 ↩4

6. Note that even though the note is marked for deletion and the `ZICNOTEDATA.ZDATA` column is overwritten, not all artifacts are deleted. In one test example, the thumbnails for

CIOFECA FORENSICS

Previous
‹ Never Trust Apple: Network ...