2025-08-08

Snapchat is commonly used app amongst social media users. In april 2025 Snapchat announced that they have surpassed 900 million monthly users [https://newsroom.snap.com/q1-2025-monthly-active-users]. One of the principal features are that pictures, videos and messages are usually available for only a short time before they become inaccessible to their recipients. Volatile user data is a feature.

In criminal cases Snapchat is often a important source for forensic artifacts. The built-in feature of short-lived user content and cloud storage of user content may result in law enforcement missing out on important digital evidence. Even the saved media files with metadata known as memories can be difficult to acquire.

This blog post presents an example of how the National Cybercrime Center (NC3) in Norway conducts mobile application reverse code engineering, in this post examplified through Snapchat. We will look at some of the forensics artifacts that are possible to acquire from Snapchat on mobile phones.

In summary, this post demonstrates several capabilities in the forensic analysis of Snapchat, including:

- Recovering users PIN or Password protecting My Eyes Only memories.
- Decrypting a encrypted user content database found on iOS.
- Decrypting Memories stored in Snapchats cloud storage.
- Recovering deleted memories by doing SQLite data forensics.

Warning: As Snapchat is rapidly updated with new functionality the techniques and tools presented here may be outdated.

Table of contents

Related work

- Regnerys, Merrhieu (2021, February 26). Decrypting and extracting juicy data, Snap!. *xperylab*. https://xperylab.medium.com/decrypting-and-extracting-juicy-data-snap-17301aa57a87
- hot3eed (2020, June 18). Reverse Engineering Snapchat (Part I): Obfuscation Techniques. *hot3eed*. https://hot3eed.github.io/2020/06/18/snap_p1_obfuscations.html
- hot3eed (2020, June 22). Reverse Engineering Snapchat (Part II): Deobfuscating the Undeobfuscatable. *hot3eed*. https://hot3eed.github.io/2020/06/18/snap_p1_obfuscations.html
- Miscellaneous working and non-working scripts on github.com/gitlab.com.
    - Some very nice scripts at https://github.com/DFIR-HBG

Reverse code engineering challenges

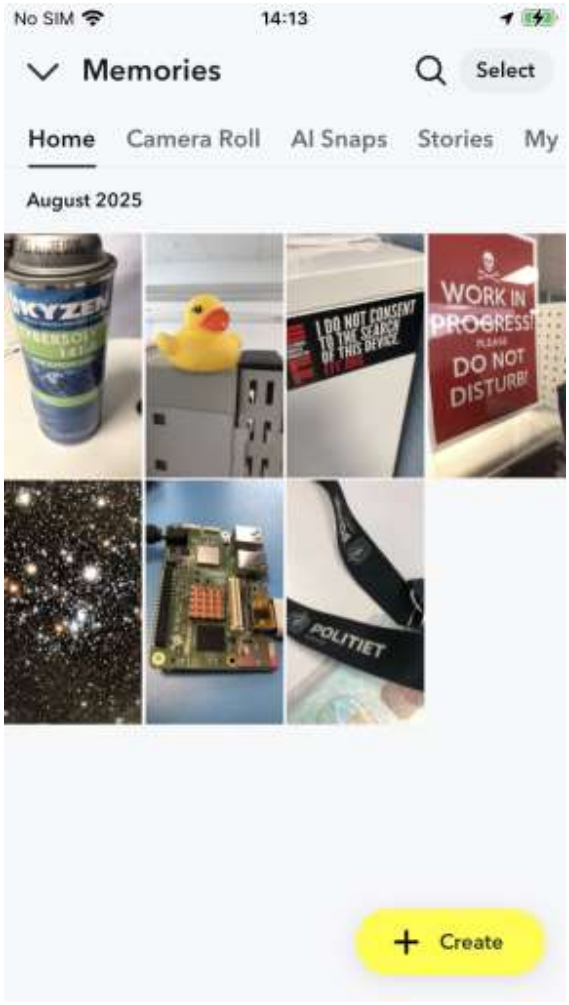
The Snapchat application is updated rapidly, and there are, for some reason, significant differences between the compiled code for iOS and Android platforms. Most of the interesting functionality is implemented in native code, which is considerably more time consuming to reverse engineer than Java or Objective-C. Snapchat, in particular, invests substantial efforts into protecting its intellectual property. For example, in 2017, Snapchat acquired Strong.Code [https://mashable.com/article/snapchat-security-strong-codes-copying-facebook], a company specializing in effective code obfuscation using LLVM. As a result, Snapchat has implemented advanced techniques such as automatic code flow changes, insertion of dead code, dynamic library calls, symbol stripping, loop flattening, and the use of Mixed Boolean Arithmetic (MBA) to obfuscate calculations, all of which make reverse engineering even more difficult.

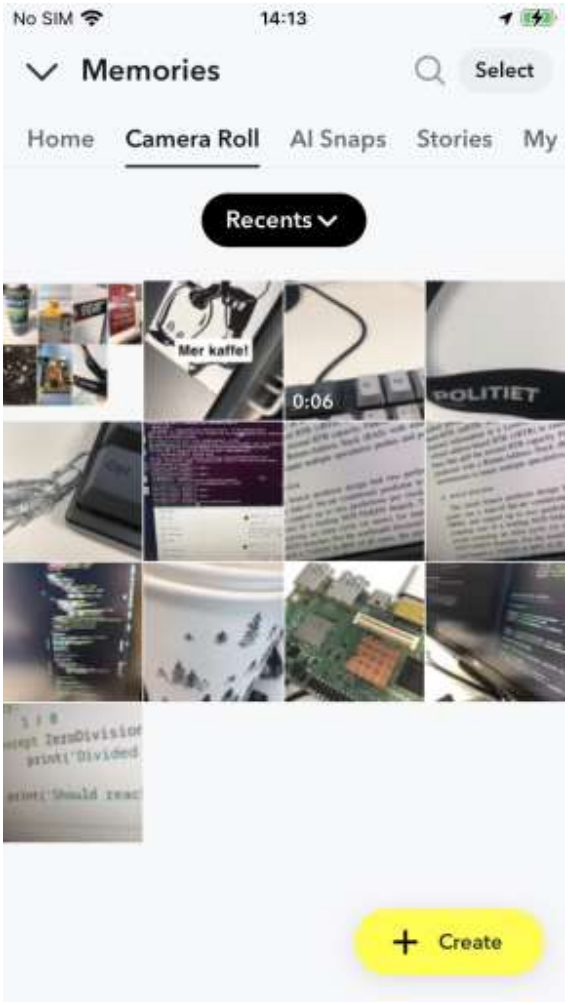
Tools and Techniques

- To understand how Snapchat works on modern phones we conducted research on both a Android and iOS test device with root access. We often rely on our own custom rooting chains to gain access to devices, but also make use of publicly available tools, such as Palera1n to root iPhones and Magisk to root Android phones.
- To disassemble and decompile binary code, we use whichever tools best suit the purpose, e.g. Ida Pro, Ghidra and jadx.
- To dynamically instrument the applications we use Frida.

Snapchat Reverse Code Engineering

Memories are pictures or videos including metadata, stored locally and/or in the Snapchat cloud. A user can choose to keep their memories inside Snapchat, shared with the phones camera roll and/or protected behind a PIN or password. Some memories in Snapchat can be protected using a feature called My Eyes Only (MEO), which adds an extra layer of security through PIN or password protection.





Data at rest

For our purpose of investigating MEO and memories we will take a look at some key databases and keystore elements on Android and iOS:

iOS

On iOS two particular databases er interesting, an encrypted database found here

```
/var/mobile/Containers/Data/Application/<UUID>/Documents/gallery_encrypted_db/<N>/<hash>/gallery.{encrypteddb.
encrypteddb-shm,encrypteddb-wal}
```

and a cleartext database found here:

```
/var/mobile/Containers/Data/Application/<UUID>/Documents/gallery_data_object/<N>/<hash>/scdb-27.{sqlite3,sqlite3-
shm,sqlite3-wal}
```

Several iOS keychain elements are found in Snapchat. Getting data from the iOS keychain can easily be dump the keychain with Frida/Objection:

```
$ objection --gadget="Snapchat" explore
com.toyopagroup.picaboo on (iPhone: 16.7.5) [net] # ios keychain dump --json snapchat_keychain.json
...
Dumped keychain to: snapchat_keychain.json
```

At least two keychain elements are related to memories:

Keychain key	Value
egocipher.key.avoidkeyderivation	Database encryption
com.snapchat.keyservice.persistedkey	Snapchat MEO

`com.snapchat.keyservice.persistedkey` looks like this:

```
{
        "access_control": "None",
        "accessible_attribute": "kSecAttrAccessibleWhenUnlockedThisDeviceOnly",
        "account": "com.snapchat.keyservice.persistedkey",
        "create_date": "2023-09-05 10:44:25 +0000",
        "dataHex": "62706c6973743030d4010203040506070a582476657273696f6e59246......",
        "entitlement_group": "3MY7A92V5W.com.toyopagroup.picaboo",
        "generic": "com.snapchat.keyservice.persistedkey",
        "item_class": "kSecClassGenericPassword",
        "modification_date": "2023-09-05 10:44:25 +0000",
        "service": "com.toyopagroup.picaboo",
},
```

`"62706c6973743030d4010203040506070a582476657273696f6e59246......"` is a NSKeyedArchiver plist object. We can deserialize it using the python package `NSKeyedUnArchiver` [https://pypi.org/project/NSKeyedUnArchiver/] and we find the following keys and values:

Key	Value
masterKey	AES CBC (KEK)
initializationVector	AES IV
userId	A UUID
keyTag	A tag/string
passphrase	Some ciphertext of the MEO pin/password

Android

On Android a related cleartext database is found here:

```
/data/data/com.snapchat.android/databases/memories.{db,db-shm,db-wal}
```

My Eyes Only

iOS

Using IDA Pro we decompile the iOS version of Snapchat. A Snapchat iOS version from May 2025 has a Mach-O file with 225 MB of compiled binary code:

```
$ file Snapchat
Snapchat: Mach-O 64-bit arm64 executable,
flags:<NOUNDEFS|DYLDLINK|TWOLEVEL|BINDS_TO_WEAK|PIE|HAS_TLV_DESCRIPTORS>

$ ls -lah Snapchat
-rwxr-xr-x. 1 user user 225M May  5 11:24 Snapchat
```

Decompiling this much binay code, and likely some well protected binary code, is time consuming and takes several hours in a modern decompiler like IDA Pro.

The Ida Pro database file ends up being 4.2 Giga Bytes of data.

Every iOS app is linked against the Apple Objective-C runtime shared library. This library implements the entire object model supporting Objective-C. To make the object oriented pseudo code in IDA Pro more readable the hex-rays `objc` plugin is helpful. Running the following commands improves the pseudo code readability:

```
idaapi.load_and_run_plugin("objc", 1) //objc classes and method calls
idaapi.load_and_run_plugin("objc", 5) //NSConcreteStackBlocks
                                      //block_layout structure applied to the function's stack frame.
idaapi.load_and_run_plugin("objc", 4) //NSConcreteGlobalBlock (globals)
```

Finding the Memory and MEO functionality within the huge binary file is time-intensive, even with the obective C symbols. Eventually, after looking through the decompiled class hierarchy we find the class `SCKeyService` and it's method `_requestWithAuthorizationRequestHandlerWithPassphrase:`:

IDA View-A | Pseudocode-A

```
1  void __cdecl -[SCKeyService _requestWithAuthorizationRequestHandlerWithPassphrase:](SCKeyService *self, SEL a2, id a3)
2  {
3    id v4; // x19
4    id v5; // x22
5    id v6; // x23
6    id v7; // x24
7    void *v8; // x25
8    id v9; // x21
9    NSData *v10; // x22
10   NSDate *v11; // x23
11   id v12; // x22
12   NSMutableDictionary *authorizationRequestHandlers; // x20
13   void *v14; // x22
14   void *v15; // x0
15   void *v16; // x21
16   id v17; // [xsp+28h] [xbp-48h]
17
18   v4 = objc_retain(a3);
19   if ( self->_persistedKey || (sub_108B97860(self) & 1) != 0 )
20   {
21     v5 = objc_retainAutoreleasedReturnValue((id)sub_108CE4A10(self->_profile));
22     v6 = objc_retainAutoreleasedReturnValue((id)-[SCUnknownGroupParticipant username]_0());
23     v7 = objc_retainAutoreleasedReturnValue((id)sub_108C49748(self->_persistedKey));
24     v8 = objc_retainAutoreleasedReturnValue(objc_msgSend(v4, "passphrase"));
25     v9 = objc_retainAutoreleasedReturnValue((id)sub_10591F2C4(v6, v7, v8, CFSTR("SKSLocal")));
26     objc_release(v8);
27     objc_release(v7);
28     objc_release(v6);
29     objc_release(v5);
30     v10 = objc_retainAutoreleasedReturnValue(-[SCKeyServicePersistedKey passphrase](self->_persistedKey, "passphrase"));
31     LODWORD(v6) = (unsigned int)objc_msgSend(v9, "isEqualToData:", v10);
```

Objective-C method calls are preformed on objects with the method `objc_msgSend` utilizing something called message passing. The second argument is called a selector and contain a useful string that describe the function. However following the calls to the correct method is not straight forward.

We could continue with static reverse engineering, but looking at this dynamically through instrumentation is much easier. Therefore we create a Frida script to hook the relevant methods.

E.g. hooking the `_requestWithAuthorizationRequestHandlerWithPassphrase:` objective C method with Frida:

```
var base_addr = Module.findBaseAddress('Snapchat');
function hook__requestWithAuthorizationRequestHandlerWithPassphrase() {
  try
  {
      let methodName = "- _requestWithAuthorizationRequestHandlerWithPassphrase:";
      let address = ObjC.classes["SCKeyService"][methodName].implementation;
      console.log(`Address to ${methodName} is ${address} (Ida_addr: 0x${(address-base_addr).toString(16)})`);
      Interceptor.attach(address, {
        onEnter: function(args) {
          let a = ObjC.Object(args[0]);
          console.log(`SCKeyService _requestWithAuthorizationRequestHandlerWithPassphrase:(${a.toString()}, `);
          console.log(`\t ${ObjC.Object(args[2]).toString()},`);
        }
      });
  }
  catch(err)
...
}
hook__requestWithAuthorizationRequestHandlerWithPassphrase()
```

We add frida-server to our rooted iPhone, launch Objection (or Frida) and import our Frida hook script. Then we use the application as a normal user and input our PIN to open our My Eyes Only memories, triggering our hook:

```
com.toyopagroup.picaboo on (iPhone: 16.7.5) [net] # import snapchat_unlock_meo_hook.js
Snapchat base address is 0x102920000 (pid: 414})
Address to - requestAuthorizationWithPassphrase:queue:completionHandler: is 0x1081ab4e8 (Ida_addr: 0x588b4e8)
Address to - _requestWithAuthorizationRequestHandlerWithPassphrase: is 0x1081ab880 (Ida_addr: 0x588b880)
Address to scrypt/sub_4424429676 is 0x10a49706c (Ida_addr: 0x107b7706c)

# When we enter the pin (1234) the hook triggers:

com.toyopagroup.picaboo on (iPhone: 16.7.5) [net] #
SCGalleryPrivateGalleryManager requestAuthorizationWithPassphrase:queue:completionHandler:
(<SCKeyService: 0x28360dc20>, 1234,...........
```

The second argument passed to this function is the PIN code we submitted. We know that we are on the right path.

By statically reversing functions called by `_requestWithAuthorizationRequestHandlerWithPassphrase:` we find some relevant encryption and decryption functions. These functions is without the Objective-C symbols and some of them may be obfuscated. However, functions without symbols can still be instrumented by calculating their addresses relative to the application's randomized base address. E.g. we can make a hook for one of the reverse engineered PBKDF2 functions with HMAC-SHA256 as pseudo random function in Frida like this:

```
var base_addr = Module.findBaseAddress('Snapchat')

function hook_sub_100ABB574(){
  try
    {
        let address = base_addr.add(0x100ABB574-0x100000000);
        console.log(`Address to sub_100ABB574 (PBKDF2WithHmacSHA256) is ${address} (Address in ida: 0x${((address-
base_addr)+0x100000000).toString(16)})`);
        var outBuf;

        Interceptor.attach(address, {
          onEnter: function(args) {
            //let a = ObjC.Object(args[2]);
            console.log(`(PBKDF2WithHmacSHA256) sub_100ABB574(\n\t pass=${args[0].readCString()},`);
            console.log(`\t passLen=${args[1]} (len cleartext (${parseInt(args[1],16)}))`);
            console.log(`\t salt=${args[2]} `);
            console.log(`\t saltLen=${args[3]} (${parseInt(args[3],16)})`);
            console.log(`\t iter=${args[4]} (${parseInt(args[4],16)})`);
            console.log(`\t v14=${args[5]} `);
            console.log(`\t keyLen=${args[6]} (${parseInt(args[6],16)})`);
            console.log(`\t outBuf=${args[7]} )`);
```

This password-based key derivation function (PBKDF2) is invoked twice within a single function, each time using a password string as input:

```
    while ( n_iterations != v38 );
    sub_1088FB29C(v36, v34 + (r_blocksizefactor << 7) + ((2 * r_blocksizefactor * (n_iterations - 1)) << 6));
    v39 = 0;
    do
    {
      if ( v33 )
      {
        v40 = v54;
        v41 = v53 + r_blocksizefactor * (v52 + ((*&v36[128 * r_blocksizefactor - 64] & (n_iterations - 1)) << 7));
        v42 = v34;
        v43 = 2 * r_blocksizefactor;
        do
        {
          sub_1088FB26C(v42, v40, v41);
          v41 += 64;
          v40 += 64;
          v42 += 64;
          --v43;
        }
        while ( v43 );
      }
      sub_1088FB29C(v36, v34);
      ++v39;
    }
    while ( v39 != n_iterations );
    v35 = v50;
    v32 = v51 + 1;
    v29 = v53;
    v54 += v50;
  }
  while ( v51 + 1 != v49 );
  v44 = j__EVP_aes_128_cbc();
  v18 = sub_1088FABA8(v46, v47, v53, v45, 1u, v44, a10, output_buf);// Second PBKDF2_HMAC-SHA256 in scrypt
                                                    //
}
else
{
  v18 = 0;
}
sub_107C4B45C();
return v18;
```

The presence of two consecutive PBKDF2 calls in a function that processes user input resembles the structure of the `scrypt` algorithm in pseudocode [https://en.wikipedia.org/wiki/Scrypt]:

Algorithm [edit]

```
Function scrypt
   Inputs: This algorithm includes the following parameters:
      Passphrase:                Bytes    string of characters to be hashed
      Salt:                      Bytes    string of random characters that modifies the hash to protect against
Rainbow table attacks
      CostFactor (N):            Integer  CPU/memory cost parameter - Must be a power of 2 (e.g. 1024)
      BlockSizeFactor (r):       Integer  blocksize parameter, which fine-tunes sequential memory read size and
performance. (8 is commonly used)
      ParallelizationFactor (p): Integer  Parallelization parameter. (1 .. 2^32-1 * hLen/MFlen)
      DesiredKeyLen (dkLen):     Integer  Desired key length in bytes
                                          (Intended output length in octets of the derived key;
                                           a positive integer satisfying dkLen ≤ (2^32- 1) * hLen.)
      hLen:                      Integer  The length in octets of the hash function (32 for SHA256).
      MFlen:                     Integer  The length in octets of the output of the mixing function (SMix below).
Defined as r * 128 in RFC7914.
   Output:
      DerivedKey:                Bytes    array of bytes, DesiredKeyLen long

   Step 1. Generate expensive salt
   blockSize ← 128*BlockSizeFactor  // Length (in bytes) of the SMix mixing function output (e.g. 128*8 = 1024 bytes)

   Use PBKDF2 to generate initial 128*BlockSizeFactor*p bytes of data (e.g. 128*8*3 = 3072 bytes)
   Treat the result as an array of p elements, each entry being blocksize bytes (e.g. 3 elements, each 1024 bytes)
   [B0...Bp-1] ← PBKDF2HMAC-SHA256(Passphrase, Salt, 1, blockSize*ParallelizationFactor)

   Mix each block in B Costfactor times using ROMix function (each block can be mixed in parallel)
   for i ← 0 to p-1 do
      Bi ← ROMix(Bi, CostFactor)

   All the elements of B is our new "expensive" salt
   expensiveSalt ← B0||B1||B2|| ... ||Bp-1  // where || is concatenation

   Step 2. Use PBKDF2 to generate the desired number of bytes, but using the expensive salt we just generated
   return PBKDF2HMAC-SHA256(Passphrase, expensiveSalt, 1, DesiredKeyLen);
```

Hooking this function with Frida we are able to confirm it is in fact `scrypt`. Additionally, we get the hard coded primitives we need to attack the scrypt cipher text off-device:

```
com.toyopagroup.picaboo on (iPhone: 16.5.1) [usb] # import snapchat_unlock_meo_hook.js
Snapchat base address is 0x100148000 (pid: 313})
Address to sub_100ABB7D4 is 0x100c037d4 (Address in ida: 0x100abb7d4)
com.toyopagroup.picaboo on (iPhone: 16.5.1) [usb] #
(scrypt?) sub_100ABB7D4(
         pass=32d46086-2da6-4e9a-9872-5781ef4a2170|1234|PyQI0imJlHq89KPW,
         passLen=0x3a (len cleartext (58))
         salt=SKSLocal
         saltLen=0x8 (8)
         a5=0x1000 (4096) (N iterations)
         a6=0x4 (4) (r block size?)
         a7=0x1 (1) (p parallelism factor?, usually 1)
         a8=0x0 (0) arg_0=0x0
         resultBuffer=0x281325ba0
```

The input to the function first parameter is the cleartext PIN with some additional string data. This string is built in a another function using string concatenation function in Objective-C named `componentsJoinedByString` in it's selector:

```
cleartext_parts = objc_retainAutoreleasedReturnValue(snap_count(&OBJC_CLASS___NSArray));
cleartext_pwd = objc_retainAutoreleasedReturnValue(objc_msgSend(cleartext_parts, "componentsJoinedByString:", CFSTR("|")));
```

Now we know that the `passphrase` in `com.snapchat.keyservice.persistedkey` is the `scrypt` encrypted ciphertext of

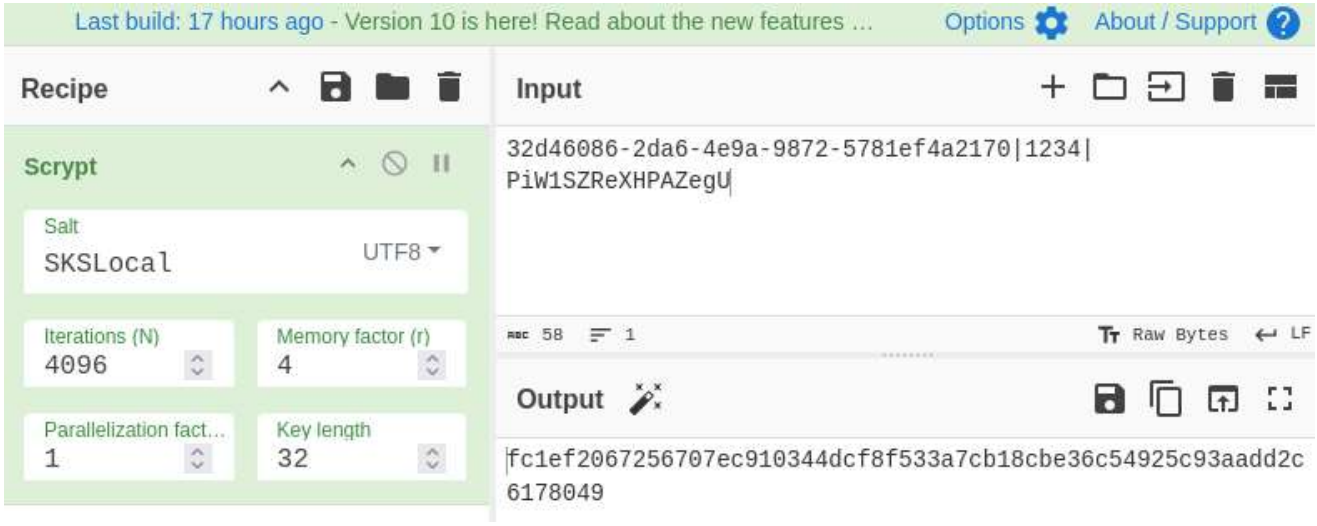
```
userId|[pin/passord]|keyTag
```

E.g.

```
32d46086-2da6-4e9a-9872-5781ef4a2170|9999|PiW1SZReXHPAZegU
```

This is given as input to `scrypt` together with; 4096 iterations (N), a memory factor of 4 (r), parallelization factor of 1 (p), key length of 32 and a hard coded `SKS Local` string as salt.

To verify we use CyberChef [https://gchq.github.io/CyberChef/]:



The output hash is the same as the one in the keychain element `com.snapchat.keyservice.persistedkey`.

Making a simple scrypt brute force script in Python:

```
$ $ python3 snapchat-meo-brute.py --keychain snapchat_keychain.json
08-Sep-23 14:18:00 - keychain userId:
08-Sep-23 14:18:00 -         32d46086-2da6-4e9a-9872-5781ef4a2170 (36 bytes)
08-Sep-23 14:18:00 - keychain keyTag:
08-Sep-23 14:18:00 -         PiW1SZReXHPAZegU (16 bytes)
08-Sep-23 14:18:00 - keychain passphrase:
08-Sep-23 14:18:00 -         b'fc1ef2067256707ec910344dcf8f533a7cb18cbe36c54925c93aadd2c6178049' (32 bytes)
08-Sep-23 14:18:00 - keychain masterKey:
08-Sep-23 14:18:00 -         b'd77e19f45daebba9f0dc32a01b60f8abb9c9c61350a3c6c4df946b254d82be2c' (32 bytes)
08-Sep-23 14:18:00 - keychain initializationVector:
08-Sep-23 14:18:00 -         b'4688d865fc90a194f5cc6ef2c75abcbd' (16 bytes)
08-Sep-23 14:18:00 - snapchat MEO brute force starting!
08-Sep-23 14:18:05 - THE PIN CODE IS: 1234
```

Or we can use hashcat [https://hashcat.net] `scrypt` kernel it the user has set a password.

Android

On Android we have a hash stored within `memories.db`:

```
$ sqlite3 memories.db
sqlite> select * from memories_meo_confidential;
dummy|$2a$06$OymDnyM7uIvmeJVKGcFzsOKnDVMPdVP15iK/9cIxKwAz13l/HUNEe|134Z9F2uu6nw3DKgG2D4q7nJxhNQo8bE35RrJU2Cviw=
|RojYZfyQoZT1zG7yx1q8vQ==
```

The hash, `$2a$06$OymDnyM7uIvmeJVKGcFzsOKnDVMPdVP15iK/9cIxKwAz13l/HUNEe`, is self explanatory:

- `$2a$` -> `bcrypt`
- `$06$` -> input cost, i.e. 2^6 rounds
- `OymDnyM7uIvmeJVKGcFzs` salt
- `OKnDVMPdVP15iK/9cIxKwAz13l/HUNEe` hash/ciphertext

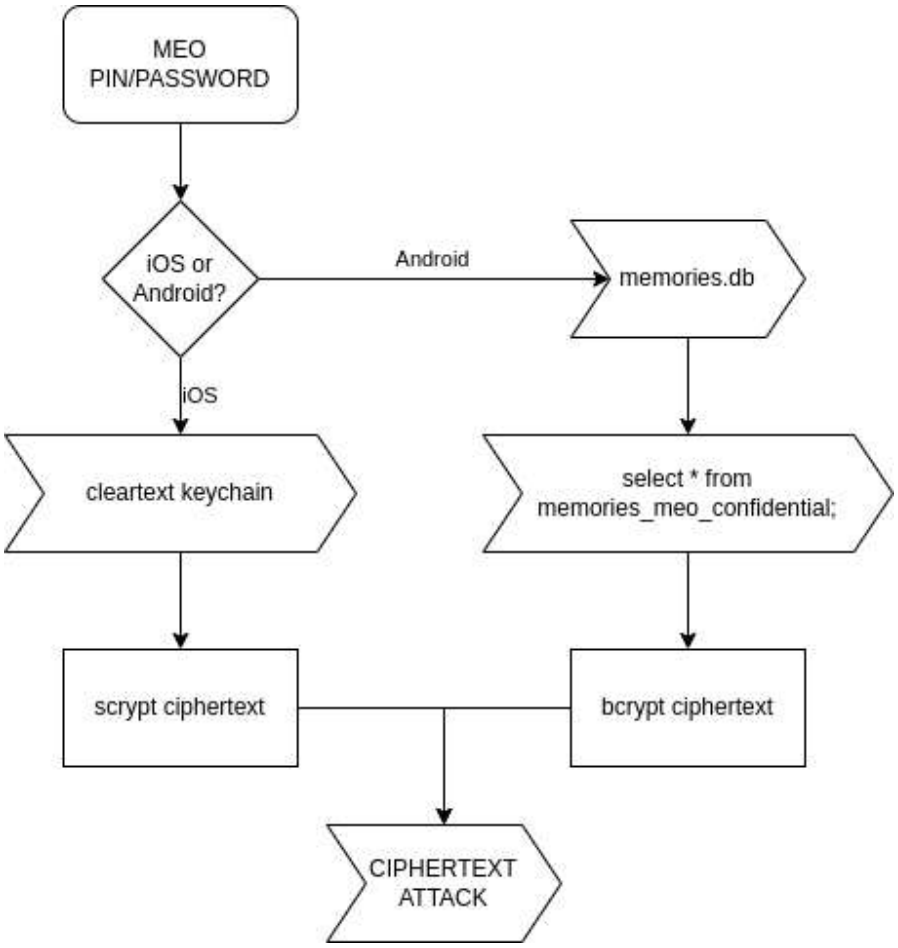
With a bcrypt POC in Python and we can brute force the PIN code:

```
$ python3 snapchat-meo-brute.py --database memories.db
07-Sep-23 16:43:48 - Read the hash $2a$06$B2eiEYcXmX8W98TqXjMEwOfO7IiCXK/Il0wqfQcT1lwQSF9JxLtlG from memories.db
07-Sep-23 16:43:48 - snapchat MEO brute force starting!
07-Sep-23 16:43:48 - THE PIN CODE IS: 1111
07-Sep-23 16:43:55 - snapchat MEO brute force done!
```

In hindsight, we probably should have used the `bcrypt` kernel in hashcat directly instead of writing our own proof-of-concept in Python.

Summary

To summarize the MEO PIN/Password attack on Android and iOS can be approached like this:



Memories Decryption

iOS

The keychain element `egocipher` contain the following data:

```
{
    "access_control": "None",
    "accessible_attribute": "kSecAttrAccessibleWhenUnlockedThisDeviceOnly",
    "account": "egocipher.key.avoidkeyderivation",
    "create_date": "2023-05-04 11:31:54 +0000",
    "dataHex": "9c66fbb7d9a9565b545b8ed87fb327a97582e9bf052acf0158246722644d5ab8",
    "entitlement_group": "3MY7A92V5W.com.toyopagroup.picaboo",
    "generic": "egocipher.key.avoidkeyderivation",
    "item_class": "kSecClassGenericPassword",
    "modification_date": "2023-05-04 11:31:54 +0000",
    "service": "com.toyopagroup.picaboo",
  },
```

From reversing and hooking in a similar manner as before we deduct that `egocipher` is a AES key that uses SQLCipher to encrypt and decrypt the database `gallery.encrypteddb`.

After some reverse code engineering we decrypt `gallery.encrypteddb` using `SQLCipher`. The chiper settings are found hard coded into various functions in the Snapchat Mach-O file:

```
 $ sqlcipher gallery.encrypteddb
sqlite> PRAGMA key="x'242927b1d7424f5c786cc323f7034ba7c158a6fa9c71255445e16b054ebfd8a9'";
ok
sqlite> PRAGMA cipher_page_size=1024;
sqlite> PRAGMA kdf_iter=64000;
sqlite> PRAGMA cipher_hmac_algorithm=HMAC_SHA1;
sqlite> PRAGMA cipher_kdf_algorithm=PBKDF2_HMAC_SHA1;
sqlite> .recover
sqlite> .output gallery.decrypteddb.recovered
sqlite> .dump
sqlite> .exit
$ cat gallery.encrypteddb.recovered | sqlite3 gallery.decrypteddb
```

Inside the decrypted `gallery.encrypteddb` we find a key and IV pair for every snap_id. If the memory is a MEO memory it has a boolean called `encrypted`:

```
sqlite> SELECT snap_id,hex(key),hex(iv) FROM snap_key_iv WHERE encrypted=1 LIMIT 2;
ca8a4da7-bd28-3952-17c7-30bd959d65f9|
        D98EA3A176777A11B992415365763C24D02326637ED77919CCE67F588F0ED3544E024D9E0815741FC167F4E1248CC224|
        EAF68B1165DA0CE15D5CBA72C2974F5E46366E04FA2D4D8A13C4E36093854926
60A5C5A3-0EBB-4DC8-9DB9-A150F94BF4D5|
        E1849B5A3E8E846BF69F9C7A9B2129DDC93D84310EB678ED00A4C52C4D28AD5E0425948FCD4D02AEB0E74A5554283264|
        24C18EEA70DC9934B54899C19A6FD19B2D36131520BB262782C83F060C5EAD30
```

Every MEO memory key and IV is AES CBC encrypted with a `masterkey` and `initializationVector`; both found in the keychain object `com.snapchat.keyservice.persistedkey` (which was a NSKeyedArchiver plist).

In the database `scdb-27.sqlite3` we find a URL to all encrypted memories stored in the SnapChat cloud. These we can decrypt with the keys we previously found for each `snap_id` that matches `zmediaid` or `zsnapid`. If the memory is stored as MEO, we first need the decrypt the key and IV with the `masterkey` and its `initializationVector`.

Let's try to download and decrypt a memory:

```
$ sqlite3 scdb-27.sqlite3
sqlite> SELECT zmediadownloadurl FROM zgallerysnap WHERE zmediaid == '60A5C5A3-0EBB-4DC8-9DB9-A150F94BF4D5' OR
znapid == '60A5C5A3-0EBB-4DC8-9DB9-A150F94BF4D5';

https://cf-st.sc-cdn.net/h/fjAGkc2oIJEnAvVKL3t84?bo=Eg0aABoAMgEISAJQHGAB&uc=28
```

Downloading the encrypted memory from the URL using `curl`:

```
$ curl -o 60A5C5A3-0EBB-4DC8-9DB9-A150F94BF4D5 https://cf-st.sc-cdn.net/h/fjAGkc2oIJEnAvVKL3t84?
bo=Eg0aABoAMgEISAJQHGAB&uc=28
$ file 60A5C5A3-0EBB-4DC8-9DB9-A150F94BF4D5
60A5C5A3-0EBB-4DC8-9DB9-A150F94BF4D5: data
```

Gives us, an encrypted binary blob of data, as expected.

Then we get the encrypted key and iv from gallery.encrypteddb:

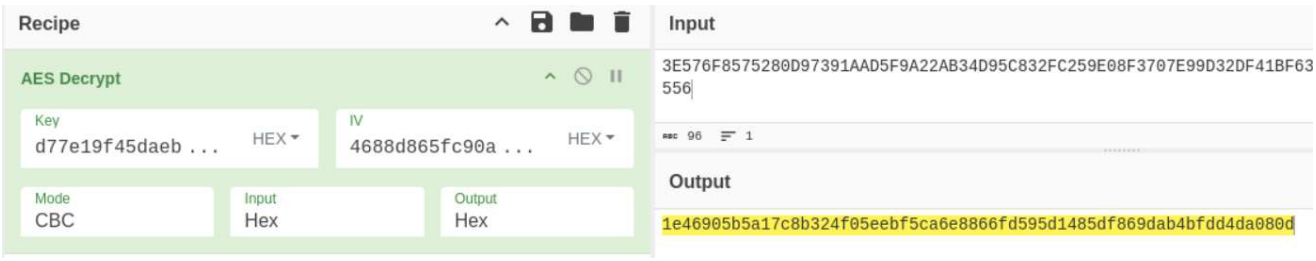
```
sqlite> SELECT snap_id,hex(key),hex(iv) FROM snap_key_iv WHERE snap_id = '60A5C5A3-0EBB-4DC8-9DB9-A150F94BF4D5';
60A5C5A3-0EBB-4DC8-9DB9-A150F94BF4D5|
E1849B5A3E8E846BF69F9C7A9B2129DDC93D84310EB678ED00A4C52C4D28AD5E0425948FCD4D02AEB0E74A5554283264|
24C18EEA70DC9934B54899C19A6FD19B2D36131520BB262782C83F060C5EAD30
```

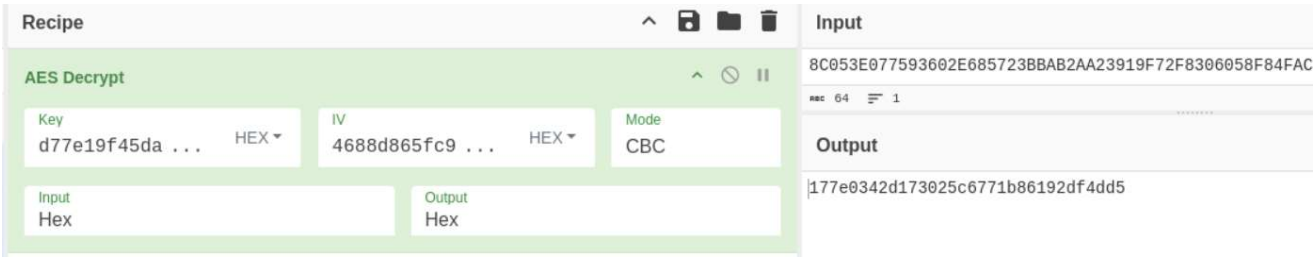
- Encrypted key: `E1849B5A3E8E846BF69F9C7A9B2129DDC93D84310EB678ED00A4C52C4D28AD5E0425948FCD4D02AEB0E74A5554283264`
- Encrypted iv: `24C18EEA70DC9934B54899C19A6FD19B2D36131520BB262782C83F060C5EAD30`

From the keystore `com.snapchat.keyservice.persistedkey` we have the NSKeyedArchive: `62706c6973743030d4010203040506070a582476657273696f6e59246......`

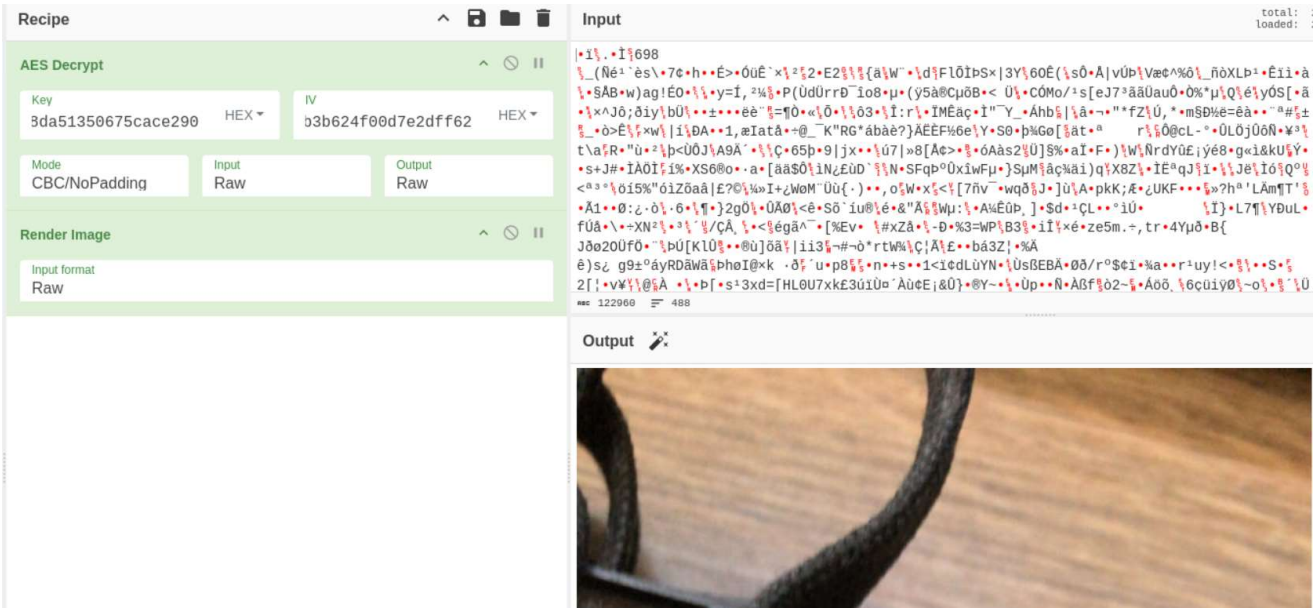
```
$ pip install NSKeyedUnArchiver
$ ipython
In [1]: persistedkey = bytes.fromhex('62706c6973743030d4010203040506070a582476657273696f6e5924....')
In [2]: NSKeyedUnArchiver.unserializeNSKeyedArchiver(persistedkey)
Out[3]:
{'keyTag': 'T+PATGnEu5AstGkQ',
 'masterKey': b'\xd7~\x19\xf4]\xae\xbb\xa9\xf0\xdc2\xa0\x1b`\xf8\xab\xb9\xc9\xc6\x13P\xa3\xc6\xc4\xdf\x94k%
M\x82\xbe,',
 'userId': '32d46086-2da6-4e9a-9872-5781ef4a2170',
 'passphrase': b'1\x8e\xd7\x86T\xd7J\xeb\x19}[\xe8\x96\xc5\xa6\x81p\xaaq\x8eC\xc0\xbaB\x98x\xb7\xd0\xb3\xc3G\xdc',
 'initializationVector': b'F\x88\xd8e\xfc\x90\xa1\x94\xf5\xccn\xf2\xc7Z\xbc\xbd'}
In [4]: NSKeyedUnArchiver.unserializeNSKeyedArchiver(persistedkey).get('masterKey').hex()
Out[4]: 'd77e19f45daebba9f0dc32a01b60f8abb9c9c61350a3c6c4df946b254d82be2c'
In [5]: NSKeyedUnArchiver.unserializeNSKeyedArchiver(persistedkey).get('initializationVector').hex()
Out[5]: '4688d865fc90a194f5cc6ef2c75abcbd'
```

The `masterkey` and `initializationVector` decrypts encrypted key and encrypted iv. Decrypting the key and iv with the AES recipe in CyberChef:





Finally, uploading the encrypted Memory blob from `https://cf-st.sc-cdn.net/h/fjAGkc2oIJEnAvVKL3t84?bo=Eg0aABoAMgEISAJQHGAB&uc=28` to Cyber Chef and decrypt it using our now decrypted AES key and IV (still AES CBC):



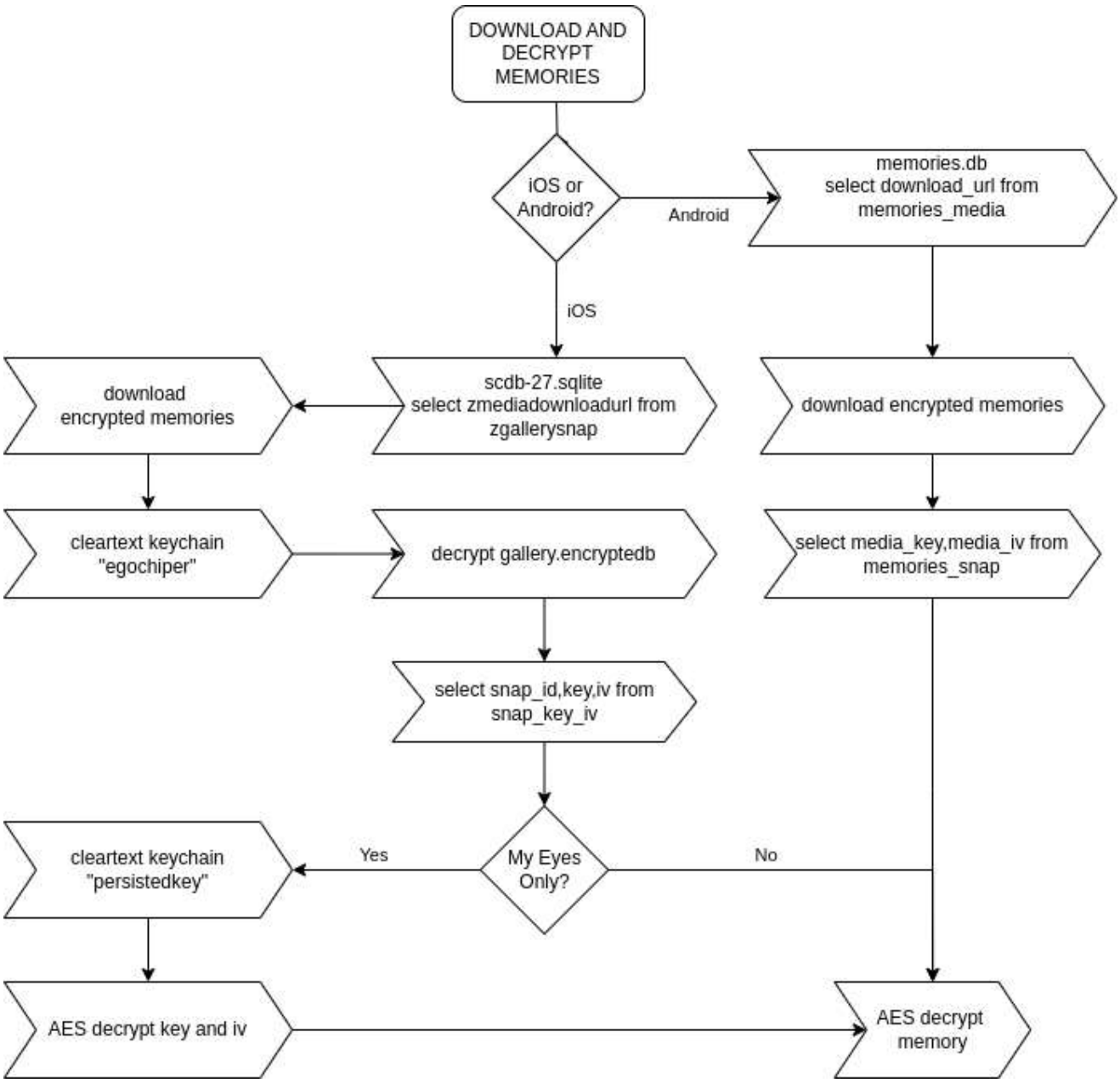
Android

It's almost the same on Android, only easier as Android keystore and SQLCipher encryption is not utilized in Snapchat:

```
 # Same key and iv as iOSs `com.snapchat.keyservice.persistedkey`
$ sqlite3 memories.db "SELECT hex(base64(master_key)),hex(base64(master_key_iv)) FROM memories_me
o_confidential;"
D77E19F45DAEBBA9F0DC32A01B60F8ABB9C9C61350A3C6C4DF946B254D82BE2C|
4688D865FC90A194F5CC6EF2C75ABCBD

# Same url as iOS `scdb-27.sqlite3`
$ sqlite3 memories.db "SELECT download_url  FROM memories_media WHERE
_id='60A5C5A3-0EBB-4DC8-9DB9-A150F94BF4D5';"
https://cf-st.sc-cdn.net/h/fjAGkc2oIJEnAvVKL3t84?bo=Eg0aABoAMgEISAJQHGAB&uc=28

$ curl -o 60A5C5A3-0EBB-4DC8-9DB9-A150F94BF4D5.android.bin https://cf-st.sc-cdn.net/h/fjAGkc2oIJEnAvVKL3t84?
bo=Eg0aABoAMgEISAJQHGAB&uc=28

# Same key and IV as iOS, however the key and IV is not encrypted with the master_key or master_key_iv..
$ sqlite3 memories.db "SELECT hex(base64(media_key)),hex(base64(media_iv)) FROM memories_snap WHERE
_id == '60A5C5A3-0EBB-4DC8-9DB9-A150F94BF4D5';"
37D0FB78B8BB0D4C48DE7204DEF330E36643D225CA60398DA51350675CACE290|
94D44B314CF7D5B3B624F00D7E2DFF62

$ md5sum 60A5C5A3-0EBB-4DC8-9DB9-A150F94BF4D5.android.bin
3aa3ff5a6294a75e7a688ff57bd0adda
$ md5sum 60A5C5A3-0EBB-4DC8-9DB9-A150F94BF4D5.ios.bin
3aa3ff5a6294a75e7a688ff57bd0adda
```

Summary

To summarize the memory decryption on Android and iOS can be approached like this:



Recovering Deleted Memories

SQLite database system uses journaling to keep the integrity of a database if something unexpected happens (power outage to device etc.). There are two methods in use; Rollback Journal and Write Ahead Logs (WAL).

Rollback Journals include information to restore a database back to its original state if a transaction fails. Write Ahead Logs contain updates to the database that are then committed to the main database also protecting the main database for corruptions.

Most applications, including Snapchat, uses Write Ahead Logs.

Both `scdb-27.sqlite3` and `gallery.encrypteddb` has it enabled:

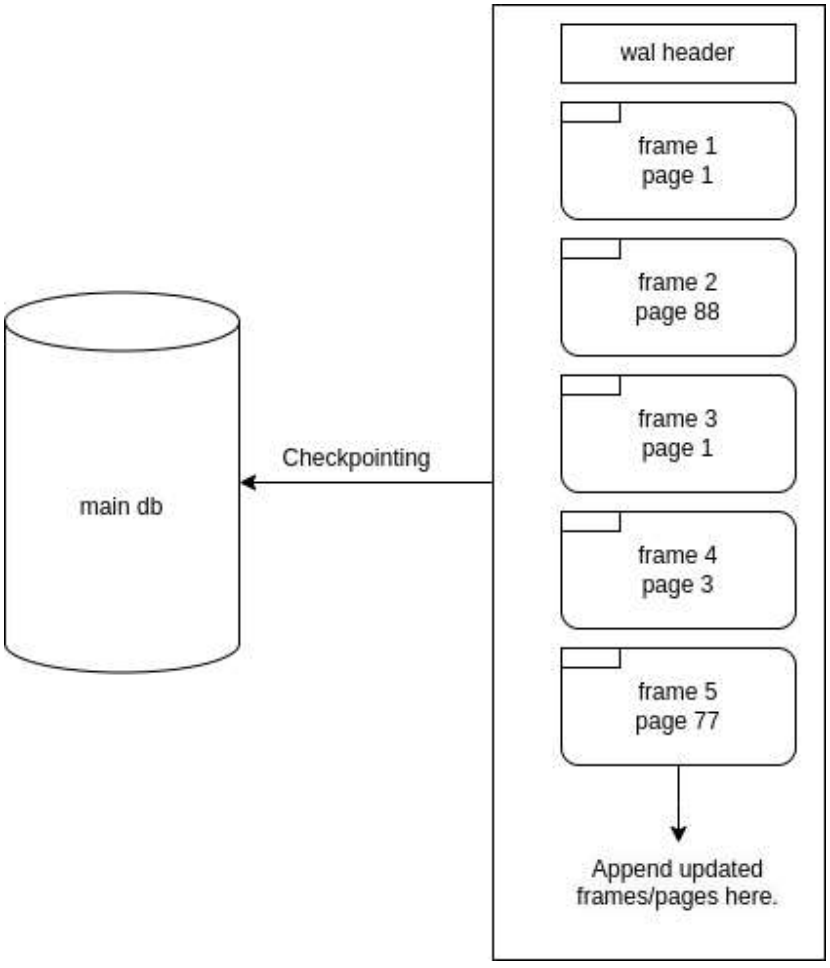
```
$ sqlite3 scdb-27.sqlite3 'PRAGMA journal_mode;'
wal

sqlcipher ../../gallery.encrypteddb
sqlite> PRAGMA key="x'242927b1d7424f5c786cc323f7034ba7c158a6fa9c71255445e16b054ebfd8a9'";
ok
sqlite> PRAGMA cipher_page_size=1024;
sqlite> PRAGMA kdf_iter=64000;
sqlite> PRAGMA cipher_hmac_algorithm=HMAC_SHA1;
sqlite> PRAGMA cipher_kdf_algorithm=PBKDF2_HMAC_SHA1;
sqlite> PRAGMA journal_mode;
wal
```

WAL usage makes it possible for us to recover deleted URLs, keys and IVs from the Snapchat databases.

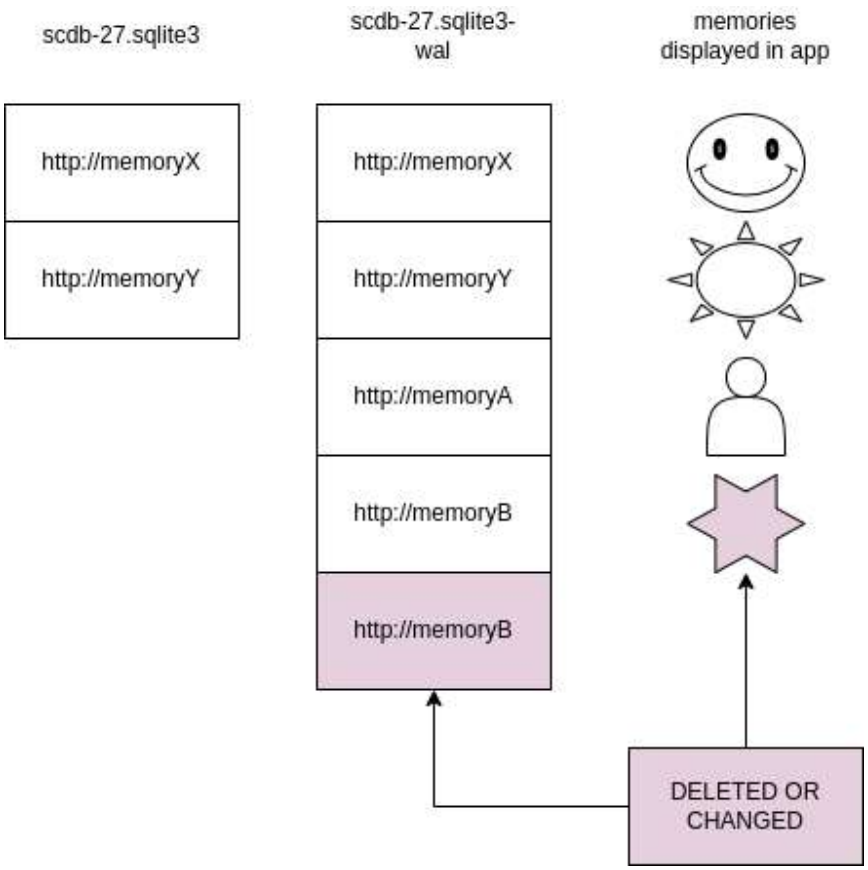
- Modifications (INSERT, DELTE, UPDATE) are written to the WAL file and not directly to the database to speed things up and handle abrupt exits.
- Checkpointing usually happens when the WAL file exceeds 1000 pages or when a database connection closes. 1000 pages means a ~4 MB large WAL file for Snapchat. This can amount to a large amount of deleted memories.

The wal file has a wal header, and every frame in the wal file also has a small header (SQLite WAL frame header). Both cleartext and encrypted databases has these headers. A single frame matches a page in the database. A database page contain only one table or a page that points to other pages for the one table it represent. In the WAL file for an encrypted database, every page is encrypted.

The WAL always grows from start to end. Thereby changes to database pages are added to the WAL file, until checkpointing happens and the file pointer to the WAL is moved to the top again (old data is still present). In the figure below, if Page 1 represents the table containing URLs, we observe that two modifications to this table are recorded in the Write-Ahead Log (WAL) file, specifically in frame 1 and frame 3.



Simplifying this and only looking at the URL table in `scdb-27.sqlite` it may look like this for each memory URL:



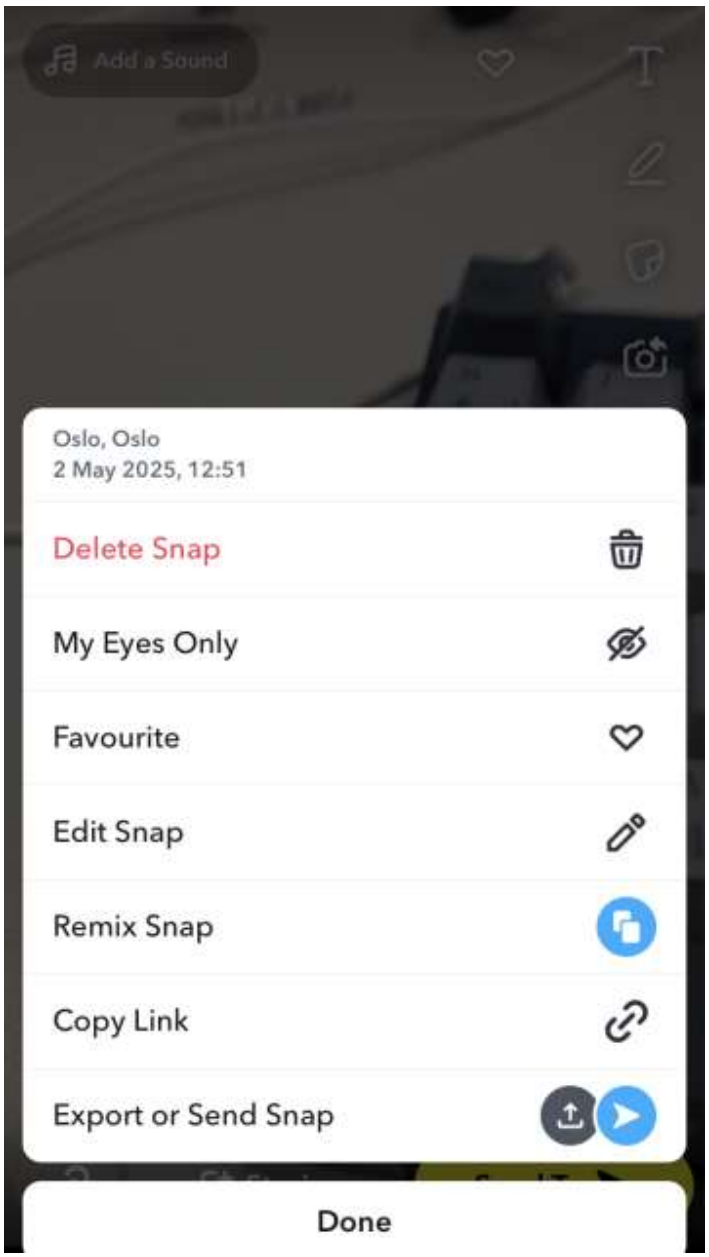
The URL `http://memoryB` is added to Snapchat, not written directly to `scdb-27.sqlite`, but kept in `scdb-27.sqlite-wal` until database checkpointing. Then `http://memoryB` is deleteded by the user, but also this table entry deletion is kept in `scdb-27.sqlite-wal` until checkpointing.

The WAL file acts as a roll-forward journal, storing all changes—including row deletions—that have been committed but not yet applied to the main database file (checkpointing). A programmer can force checkpoint with PRAGMA statements:

```
PRAGMA wal_checkpoint; #forces a checkpoint
```

Lets look at an example on how to recovery of a delteded MEO memory from an iOS device:

1 Use Snapchat and delete a MEO memory.



2. Create a python script that creates a database snapshot of every comit frame from the wal files of `gallery.encrypteddb` and `scdb-27.sqlite3`:

```
./wal_versions.py --db gallery.encrypteddb --wal gallery.encrypteddb-wal  --outfolder wall_versions/
./wal_versions.py --db scdb-27.sqlite3 --wal scdb-27.sqlite3  --outfolder wall_versions/
```

3. In this experiment we get 8 different versions of `gallery.encrypteddb` and 41 versions of `scdb-27.sqlite3`. Then we decrypt each version of `gallery.encrypteddb`:

```
decrypt_dbs.py --db gallery.encrypteddb* --key 242927b1d7424f5c786cc323f7034ba7c158a6fa9c71255445e16b054ebfd8a9
```

4. Then we diff the original db with the version with the last commit frame from the WAL:

```
$ sqldiff gallery.encrypteddb_c8.decrypted.db gallery.encrypteddb_c0.decrypted.db --table snap_key_iv --summary
snap_key_iv: 0 changes, 0 inserts, 2 deletes, 335 unchanged

$ sqldiff scdb-27.sqlite3_c45 scdb-27.sqlite3_c00 --table zgallerysnap --summary
zgallerysnap: 0 changes, 3 inserts, 19 deletes, 130 unchanged

$ sqldiff gallery.encrypteddb_c8.decrypted.db gallery.encrypteddb_c0.decrypted.db --table snap_key_iv
DELETE FROM snap_key_iv WHERE rowid=336;
DELETE FROM snap_key_iv WHERE rowid=337;
```

From the diff we see that 2 deletions in `gallery.encrypteddb-wal` are ready to be checkpointed into `gallery.encryptedb`. Let's look at the rows:

```
$ sqlite3 gallery.encrypteddb_c8.decrypted.db "SELECT snap_id,hex(key),hex(iv),encrypted FROM snap_key_iv WHERE
rowid=336"
2E39E633-ACFA-480F-848B-DCDD177EC4E0|
C040AB99959A506490E6864E31E2628A71C0E5D1AEA1AE2A3FAD1070B21886C3|
320ED0DE7B37835A2F4F9C78B12A0FA0|
0
$ sqlite3 gallery.encrypteddb_c8.decrypted.db "SELECT snap_id,hex(key),hex(iv),encrypted FROM snap_key_iv WHERE
rowid=337"
919CE1B6-423D-4C05-948E-609E1642505B|
F469B5BB1FB95A88DDBE0B5F168E90D9BF9460B89FCD5F71027AB23D29D2989D89DD35843E0C41FE169A884290C2D8D8|
83261D88C789DD2D7482E9E9D4E9A275D7082DB53F82D14D130683128FCA6E35
|1
```

Looks like one MEO memory and one regular memory has been deleted?!

5. Then we get these memories from their URL:

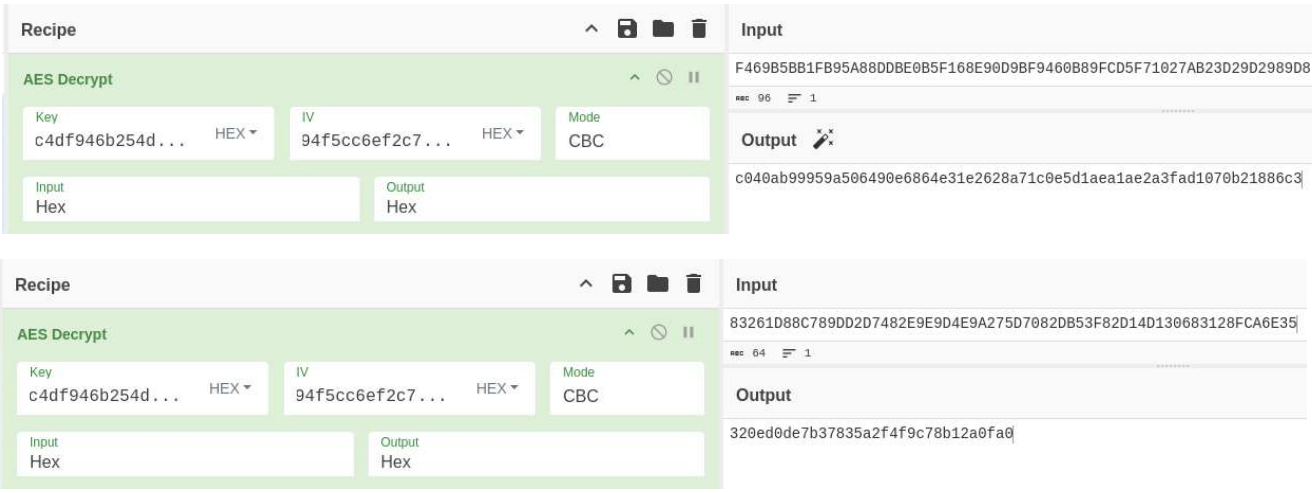
```
$ sqlite3 scdb-27.sqlite3_c45 "SELECT zmediadownloadurl FROM zgallerysnap WHERE zsnapid == '2E39E633-ACFA-480F-
848B-DCDD177EC4E0'
OR zmediaid == '2E39E633-ACFA-480F-848B-DCDD177EC4E0';"

https://cf-st.sc-cdn.net/h/3eH3hDhTzfXqFHdTYHVjg?bo=EgkyAQhIAVALYAE%3D&uc=11

$ sqlite3 scdb-27.sqlite3_c45 "SELECT zmediadownloadurl FROM zgallerysnap WHERE zsnapid == '919CE1B6-423D-4C05-
948E-609E1642505B' OR
zmediaid == '919CE1B6-423D-4C05-948E-609E1642505B';"

https://cf-st.sc-cdn.net/h/3eH3hDhTzfXqFHdTYHVjg?bo=EgkyAQhIAVALYAE%3D&uc=11
```

We see that the URLs are identical. This is how snapchat moves a normal memory to a MEO memory. Since we deleteded a MEO memory it's key and IV is now encrypted in the latest database commit.

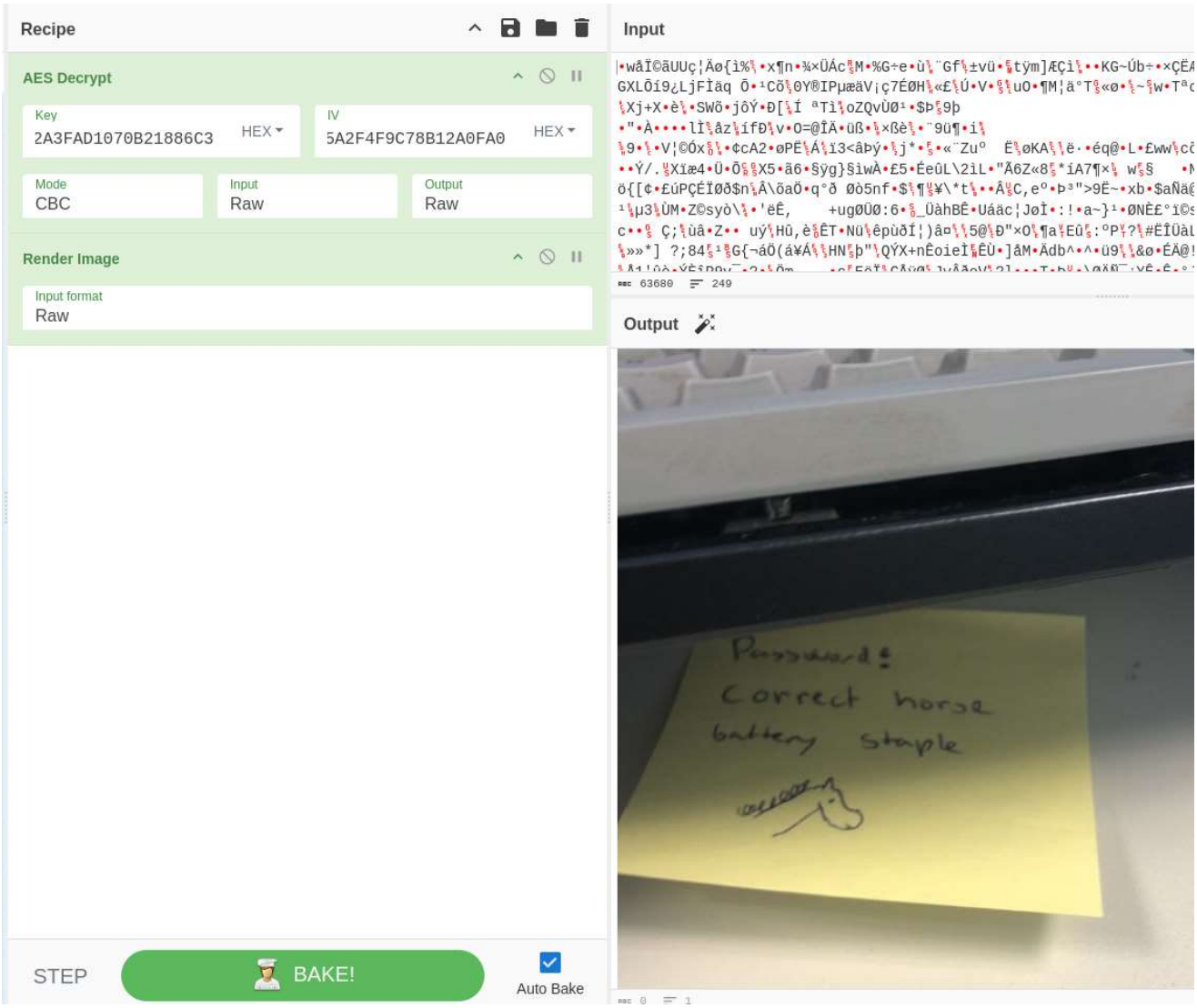
```
curl -o 919CE1B6-423D-4C05-948E-609E1642505B.bin https://cf-st.sc-cdn.net/h/3eH3hDhTzfXqFHdTYHVjg?
bo=EgkyAQhIAVALYAE%3D&uc=11
$ file 919CE1B6-423D-4C05-948E-609E1642505B.bin
919CE1B6-423D-4C05-948E-609E1642505B.bin: data
```

6. In this case we do not even need to decrypt the key and IV since we have its 'cleartext' in the recovered data from the WAL:

```
$ sqlite3 gallery.encrypteddb_c8.decrypted.db "SELECT snap_id,hex(key),hex(iv),encrypted FROM snap_key_iv WHERE
rowid=336"
2E39E633-ACFA-480F-848B-DCDD177EC4E0|
C040AB99959A506490E6864E31E2628A71C0E5D1AEA1AE2A3FAD1070B21886C3|
320ED0DE7B37835A2F4F9C78B12A0FA0|
0
$ sqlite3 gallery.encrypteddb_c8.decrypted.db "SELECT snap_id,hex(key),hex(iv),encrypted FROM snap_key_iv WHERE
rowid=337"
919CE1B6-423D-4C05-948E-609E1642505B|
F469B5BB1FB95A88DDBE0B5F168E90D9BF9460B89FCD5F71027AB23D29D2989D89DD35843E0C41FE169A884290C2D8D8|83261D88C789DD2D74
82E9E9D4E9A275D7082DB53F82D14D130683128FCA6E35
|1
```



7. Finally we recover the deleted MEO memory:



The snapchat database `gallery.encrypteddb` also contain the position of where the memory was made, a nice artifact in many criminal cases.

```
sqlite3 gallery.encrypteddb_c8.decrypted.db
"SELECT latitude, longitude FROM snap_location_table
WHERE snap_id == '919CE1B6-423D-4C05-948E-609E1642505B';"
59.90888582378O3|10.8171173813676
```

Conclusions

- The MEO pin/password are crackable to given a aquisition of snapchat user data:
  - On iOS we need the keychain and two user databases.
  - On Android we need a user database.
- The MEO pin/password are not used to protect the confidentiality of the MEO memories (it's only a GUI thing..).
  - On iOS a static AES key and IV in the keychain is used to encrypt MEO keys and IVs in `gallery.encrypteddb`. URLs are found in `scdb-27.sqlite`
  - On Android AES keys, IVs and URLs are found in the `memories.db` file
- Deleting a memory only deletes the URL for it and its key and IV pair from the local databases:
  - Memories are keep on the servers for some time after deletion (observed ~2 months)
  - No authenication is necessary to download encrypted Memories (curl works).
- The use of SQLite journaling Write-Ahead Log (WAL) makes it possible to recover deleted URLs and key material in the databases.

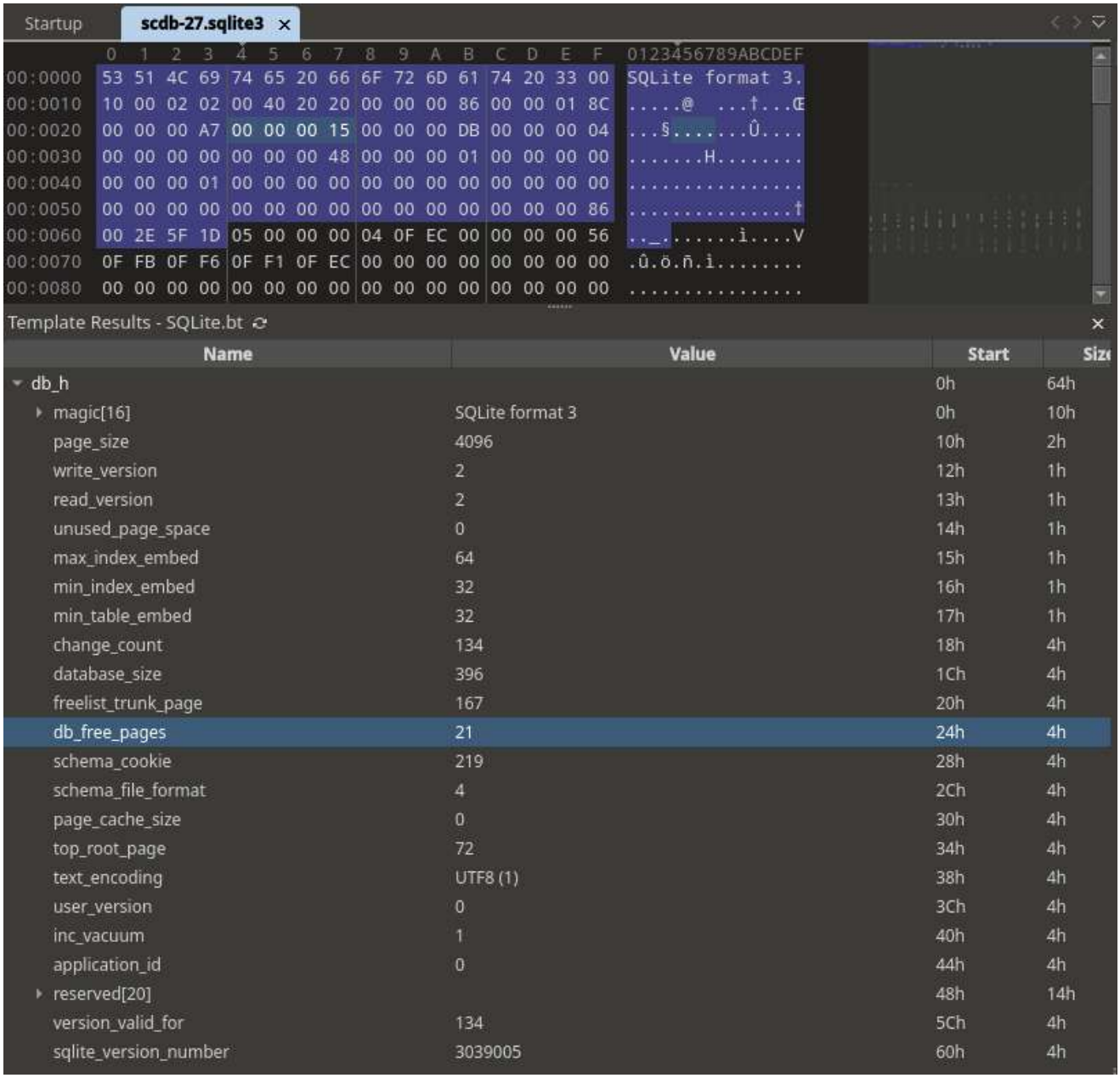
Bonus: More possibilities for database recovery

- SQLite free list: Deleted pages (4096 bytes each) in the database are not immediately removed from the database file. Instead, they are added to a free list. In every page in the free list the old deleted data remaining intact until the pages are reused (e.g when the database grows).

To inspect free lists we can use the sqlite-dissect tool from USA DoD Crime Center (DC3) [https://github.com/dod-cyber-crime-center/sqlite-dissect]:

```
$ pip install sqlite-dissect $ sqlite_dissect --carve --carve-freelists -k scdb-27.sqlite3 -d results
```

The SQLite database header shows us how many free pages that are available in the free list. These pages might contain old and possible deleteded data...


Startup
scdb-27.sqlite3
0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
00:0000 53 51 4C 69 74 65 20 66 6F 72 6D 61 74 20 33 00 SQLite format 3.
00:0010 10 00 02 02 00 40 20 20 00 00 00 86 00 00 01 8C .....@ ...†...Œ
00:0020 00 00 00 A7 00 00 00 15 00 00 00 DB 00 00 00 04 ...§.......Û....
00:0030 00 00 00 00 00 00 00 48 00 00 00 01 00 00 00 00 ....H........
00:0040 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 ....
00:0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 86 ....†
00:0060 00 2E 5F 1D 05 00 00 00 04 0F EC 00 00 00 00 56 .._.......ì....V
00:0070 0F FB 0F F6 0F F1 0F EC 00 00 00 00 00 00 00 00 .û.ö.ñ.ì........
00:0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
Template Results - SQLite.bt
Name
Value
Start
Size
db_h 0h 64h
magic[16] SQLite format 3 0h 10h
page_size 4096 10h 2h
write_version 2 12h 1h
read_version 2 13h 1h
unused_page_space 0 14h 1h
max_index_embed 64 15h 1h
min_index_embed 32 16h 1h
min_table_embed 32 17h 1h
change_count 134 18h 4h
database_size 396 1Ch 4h
freelist_trunk_page 167 20h 4h
db_free_pages 21 24h 4h
schema_cookie 219 28h 4h
schema_file_format 4 2Ch 4h
page_cache_size 0 30h 4h
top_root_page 72 34h 4h
text_encoding UTF8 (1) 38h 4h
user_version 0 3Ch 4h
inc_vacuum 1 40h 4h
application_id 0 44h 4h
reserved[20] 48h 14h
version_valid_for 134 5Ch 4h
sqlite_version_number 3039005 60h 4h