Forensics from the sausage factory

I worked at the coal face of a UK computer forensics lab and performed production line forensics - day in day out - welcome to the sausage factory

Saturday, 2 July 2011

SQLite overflow pages and other loose ends...

This is the fourth post dealing with the elements making up SQLite databases and complements the previous three:

- Carving SQLite databases from unallocated clusters
- SQLite Pointer Maps pages
- An analysis of the record structure within SQLite databases

We will remember from these previous posts that:

- The entire database file is divided into equally sized *pages* - SQLite database files always consist of an exact number of *pages*
- The *page* size is always a power of two between 512 (2^9) and 65536 (2^{16}) bytes
- All multibyte integer values are read big endian
- The *page* size for a database file is determined by the 2 byte integer located at an offset of 16 bytes from the beginning of the database file
- *Pages* are numbered beginning from 1, not 0
- Therefore to navigate to a particular *page* when you have a *page* number you have to calculate the offset from the start of the database using the formula: *offset = (page number-1) x page size*

and that the database may have the following possible page types:

- An index B-Tree internal node
- An index B-Tree leaf node
- A table B-Tree internal node
- A table B-Tree leaf node
- An overflow page
- A freelist page
- A pointer map page
- The locking page

In this post I am going to take a closer look at Overflow pages.

Overflow pages are required when a record within a database

Where to find me

No longer at the sausage factory but you may find me on my bike looking for work!

My name is Richard Drinkwater. I am a freelance computer forensics consultant and welcome enquiries sent to DC1743 (at) gmail dot com

LinkedIn profile

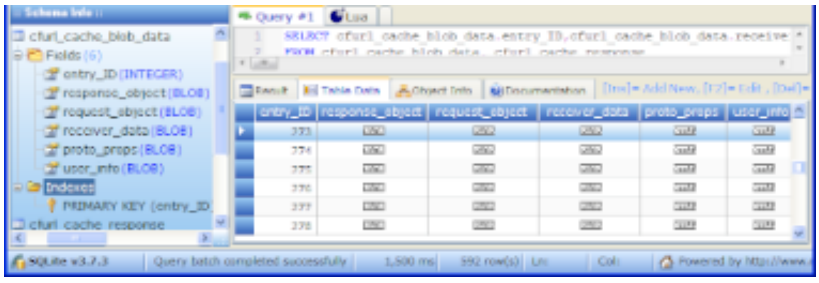
requires more space than that available within a cell in one database page. One SQLite database of forensic interest is the Cache.db file maintained by the Apple Safari web browser. One of the tables within this database is entitled **cfurl_cache_blob_data** which uses the *receiver_data* field to store the cached item itself (e.g. cached jpgs, gifs, pngs, html et al) as a BLOB. A BLOB is a **B**inary **L**arge **OB**ject. These cached objects often require overflow pages and we can demonstrate the mechanics of them by walking through a record within Cache.db.

If you run a file carver across a Cache.db file searching for pictures you are likely to carve out a number corrupt pictures as shown in the example within Figure 1.

Double click to enlarge

Figure 1

It can be seen that this picture starts at File offset 4583317 within Cache.db. By examining the two bytes at offset 16 within this SQLite database we have established that the database page size is 1024 bytes. The record that contains this picture has six fields as shown in Figure 2.



Double click to enlarge

Websites I go back to

Computer Forensics Resources

Digital Detective Forums

Forensicwiki

Macosxforensics

Slyck

Wotsit

Blogs I read

Leo Leporte

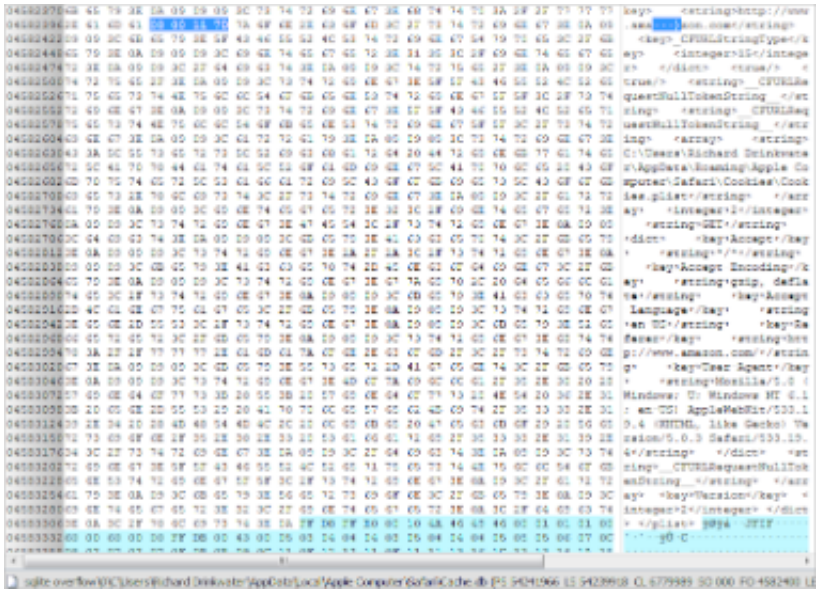
Figure 2

As discussed in my earlier post An analysis of the record structure within SQLite databases the data making up this record is store in a serialised way (the data representing field 1 is immediately followed by the data representing field 2 and so on with no delimiters). It can be seen therefore that a cell storing a record within the **cfurl_cache_blob_data** table is almost bound to overflow the 1024 byte database page.

In our example our corrupt picture starts at FO 4583317. To calculate the database page it is stored in we divide the offset by the page size 4583317/1024=4475.8955078125 and round up to establish the page number. Our corrupt picture header is in page 4476.

The SQLite.org file format states that *overflow pages are chained together using a singly linked list. The first 4 bytes of each overflow page is a big-endian unsigned integer value containing the page number of the next page in the list. The remaining usable database page space is available for record data.*

We know that our picture is likely to be stored in a number of overflow pages and we can establish the next page by looking at the first 4 bytes of the page that is in. Using the formula *offset = (page number-1) x page size* I can calculate that the offset of these 4 bytes at the start of the page is 4475 x 1024=4582400. This offset can be seen in Figure 3.



Double click to enlarge

Figure 3

The 4 bytes in hex are 00 00 11 7D which decoded big endian is 4477. The next page therefore in the linked list is page 4477. The first 4 bytes of this page found at offset 4583424 in

Lance Mueller's blog

Harlan's blog

Happy as a monkey

Andre Ross's blog

Podcasts I listen to

Cyberspeak

MacBreak Weekly

Blog Archive

▶ 2013 (1)

▶ 2012 (4)

▼ 2011 (5)

▼ July (1)

SQLite overflow pages and other loose ends...

▶ May (2)

▶ April (1)

▶ January (1)

▶ 2010 (21)

▶ 2009 (25)

▶ 2008 (24)

hex are 00 00 11 7E which decoded big endian is 4478. The first 4 bytes of this page found at offset 4584448 in hex are 00 00 11 7F which decoded big endian is 4479. The first four bytes of this page found at offset 4585472 are in hex 00 00 00 00. This value signifies the last page in the linked list.

I can be seen in this example that our corrupt picture starts in page 4476 and overflows into pages 4477, 4478 and 4479. Obviously the overflow pages are contiguous in this case, so in theory at least, if I copy the data from the jpg header of the corrupt picture to the jpg footer and edit out the 'corruption' I should end up with a complete picture. The corruption was likely caused by the overflow page values at the start of each page so using a hex editor I can remove these and *hey presto:*

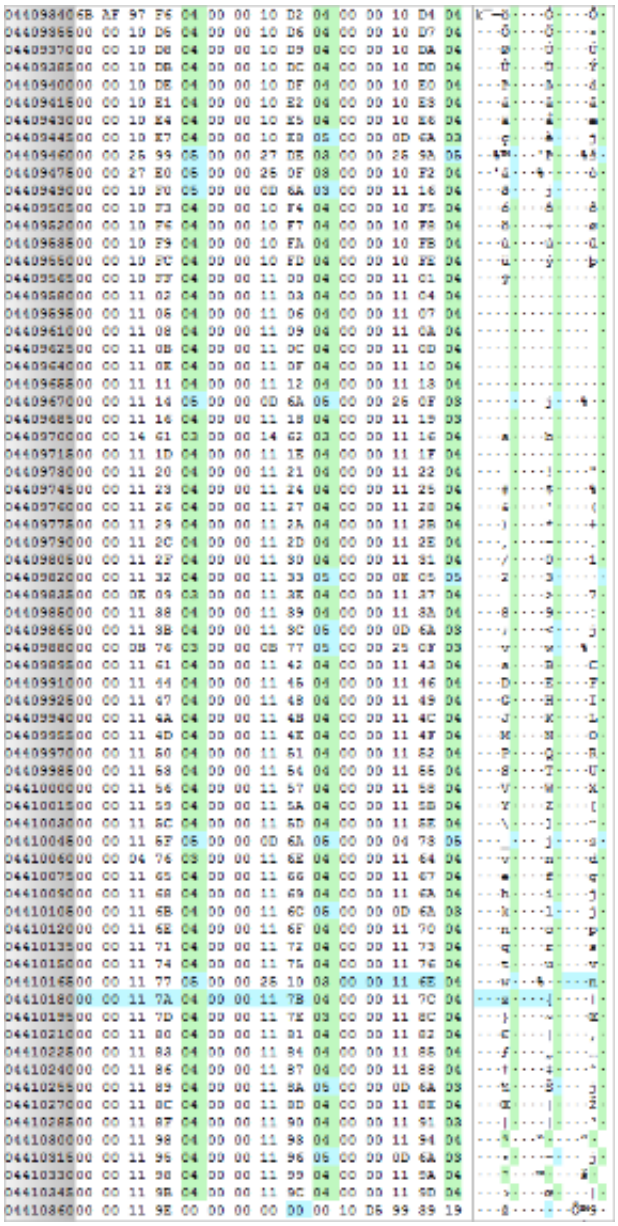
Essentially what we have done here is start in the middle of the record and work forwards to the end. Because overflow pages are chained together using a linked list this is relatively straightforward.

But what do we do if we want to locate the earlier pages in the record? This is a little more complicated because each overflow page does not contain the page number of the previous page. The Safari cache.db SQLite 3 database is an auto-vacuum database so we could utilise Pointer Map pages to locate the parent page of the page (4476) where our corrupt picture header is stored. You will recall from my previous post that Pointer Map pages store a 5 byte record relating to every page that **follows** the Pointer Map *page.* Pointer Map pages found in Safari Cache.db files will have a lot of entries that relate to overflow pages. The 5 byte records are structured with 1 byte indicating a Page Type and then 4 bytes, decoded big endian, referencing the parent page number as follows:

- **0x01** 0x00 0x00 0x00 0x00
  This record relates to a B-tree root page which obviously does not have a parent page, hence the page number being indicated as zero.
- **0x02** 0x00 0x00 0x00 0x00
  This record relates to a free page, which also does not have a parent page.
- **0x03** 0xVV 0xVV 0xVV 0xVV (where VV indicates a variable)
  This record relates to the first page in an overflow chain. The parent page number is the number of the B-Tree page containing the B-Tree cell to which the overflow chain belongs.

- **0x04** 0xVV 0xVV 0xVV 0xVV (where VV indicates a variable)
  This record relates to a page that is part of an overflow chain, but not the first page in that chain. The parent page number is the number of the previous page in the overflow chain linked-list.
- **0x05** 0xVV 0xVV 0xVV 0xVV (where VV indicates a variable)
  This record relates to a page that is part of a table or index B-Tree structure, and is not an overflow page or root page. The parent page number is the number of the page containing the parent tree node in the B-Tree structure.

We can expect to see a lot of Page Types 0x03 and 0x04. So how do we find the pointer map page? We know that the Pointer Map page may contain up to (database page size/5) records (rounded down if necessary) - in this case 1024/5=204.8 so there are 204 records in each Pointer Map page. The first Pointer Map page is Page 2. This is followed by 204 pages and then another Pointer Map page, page 207, followed by 204 pages and then another Pointer Map page, page 412 and so on. In other words there is a Pointer Map page every 205th page, starting page 2. In our example we know that our corrupt picture header is in database page 4476 and the applicable Pointer Map page is prior to it. To calculate the applicable Pointer Map page number we divide 4476 by 205 = 21.834146341463415, round down to 21 and multiply by 205 and then add 2 which equals 4307. The applicable Pointer Map page for page 4476 of the database is page 4307. Using the formula *offset = (page number-1) x page size* I can calculate that the offset to this page is 4409344. This page can be seen in Figure 4. Each Page Type flag where it references an Overflow page is bookmarked in green, other Page Types are in blue. The first 5 byte record relates to database page 4308, the second 5 byte record page 4309 and so on. The record for page 4476 is the 169th record on the Pointer Map page (4476-4307).



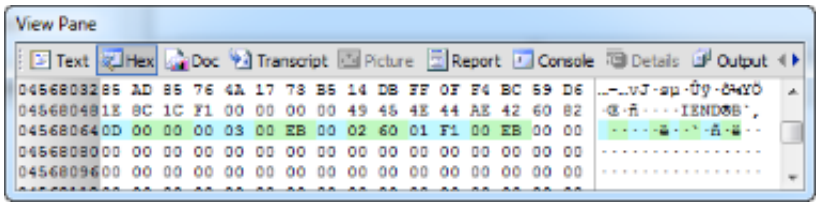
Double click to enlarge

Figure 4

It can be seen that the 169th record is in hex **04 00 00 11 7B**. This record has a flag 0x04 which indicates that this record relates to a page that is part of an overflow chain, but not the first page in that chain. The parent page number is 00 00 11 7B decoded big endian to page 4475. The record for page 4475 is the 168th record on the page **04 00 00 11 7A.** This record also indicates that the page is part of an overflow chain but not the first page. The parent page number is 00 00 11 7A decoded big endian to page 4474. The record for page 4474 is the 167th record on the page **03 00 00 11 6E.** This record has a flag 0x03 which indicates that this record relates to the first page in an overflow chain. The parent page number is the number of the B-Tree page containing the B-Tree cell to which the overflow chain belongs. The parent page number is 00 00 11 6E decoded big endian to page 4462.

Using the formula *offset = (page number-1) x page size* I can

calculate that the offset to the beginning of this page is 4568064. We can now decode the page header (detailed more fully in the post An analysis of the record structure within SQLite databases) shown in Figure 5.



Double click to enlarge

Figure 5

Figure 5 shows the page header of the B-tree leaf node page. The first byte **0D** is a flag indicating the page is a table B-tree leaf node. The next two bytes **00 00** indicate that there are no free blocks within the page. The next two bytes **00 03** read big endian indicate that there are three cells stored on the page. The next two bytes at offset 5 within the page header **00 EB** decoded big endian give a value of 235 which is the byte offset of the first byte of the cell content area relative to the start of the page. The last byte of the eight byte page header **00** is used to indicate the number of fragmented free bytes on the page, in this case there are none. The remaining highlighted three pairs of bytes **02 60**, **01 F1** and **00 EB** are the cell pointer array for this page. These three values are offsets to the start of each cell when decoded big endian are 608, 497 and 235 respectively. We will focus on the cell at offset 235. At offset 235 we find two Varints representing the *Length of Payload* and *Row ID* (see Figure 6).

Double click to enlarge

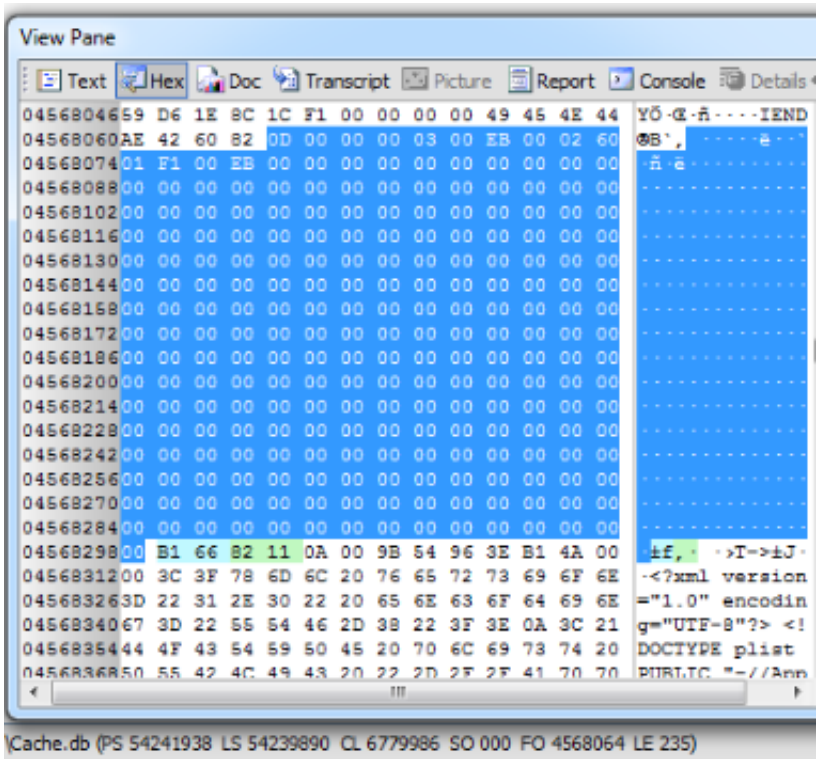


Figure 6

The varints are **B1 66** and **82 11**. The calculation needed to decode them follows in Figure 7:

OFFSET 235

Payload length

File Record

Char	Hex	UInt8	Int8	Binary
±	b1	177	-79	10110001

File Record

Char	Hex	UInt8	Int8	Binary
f	66	102	102	01100110

01100011100110 = 6374 bytes

Row ID

File Record

Char	Hex	UInt8	Int8	Binary
,	82	130	-126	10000010

File Record

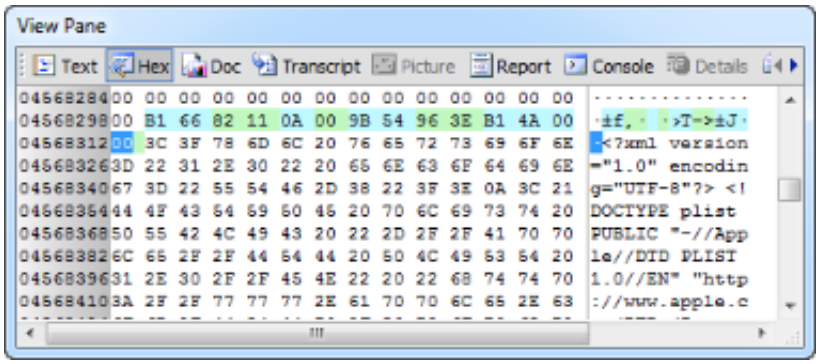
Char	Hex	UInt8	Int8	Binary
	11	17	17	00010001

00000100010001 = 273

Double click to enlarge

Figure 7

Following the *Length of Payload* and *Row ID* are varints representing *the Length of the Payload Header* and the serial type codes of the *entry_ID, response_object, request_object, receiver_data, proto_props* and *user_info* fields respectively as shown in Figure 8:



Double click to enlarge

Figure 8

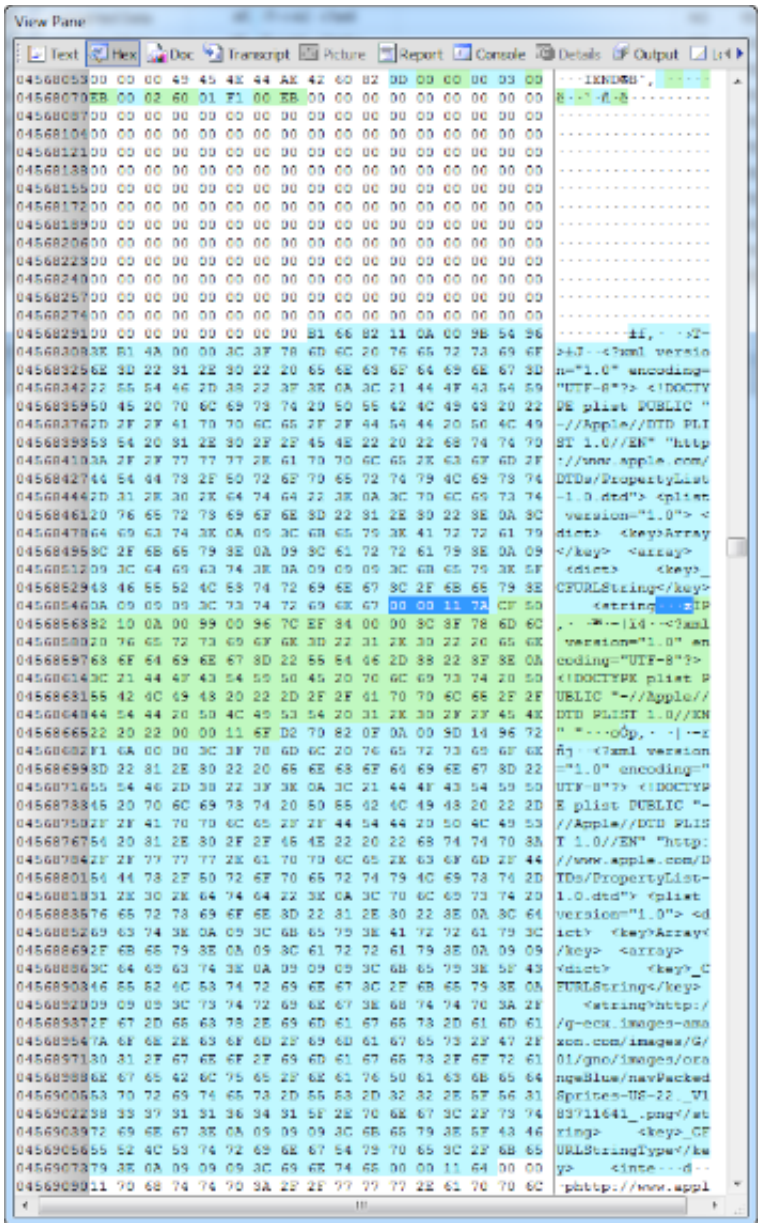
Figure 8 shows highlighted in blue and green the first three elements of the Cell make up - the *Payload Length*, the *Row ID* and the *Payload Header.* We have already decoded the *Payload Length* **B1 66** and the *Row ID* **82 11**. The next byte **h0A** denotes the length of the *Payload Header* which is in this case 10 bytes (including the *Payload Header Length* byte). It can be seen therefore that the remaining 9 bytes contain the *varints* ***00, 9B 54, 96 3E, B1 4A, 00*** *and* ***00***. To determine what each *varint* indicates we have to consult the Serial Type Code chart detailed in the post An analysis of the record structure within SQLite databases . Each Serial Type Code details the type and length of the data in the *payload* that follows the *payload header*.

- **00** This serial type code indicates that the first field is NULL and the content length is 0 bytes. We know that the first field in our record relates to Row ID however the SQLite.org file format states *If a database table column is declared as an INTEGER PRIMARY KEY, then it is an alias for the rowid field, which is stored as the table B-Tree key value. Instead of duplicating the integer value in the associated record, the record field associated with the INTEGER PRIMARY KEY column is always set to an SQL NULL.*
- **9B 54** This serial type code has a value of 3540 which is greater than 12 and an even number. The chart indicates therefore that this field is a BLOB (3540-12)/2 bytes in length [1764 bytes]
- **96 3E** This serial type code has a value of 2878 which is greater than 12 and an even number. The chart indicates therefore that this field is a BLOB (2878-12)/2 bytes in length [1433 bytes]
- **B1 4A** This serial type code has a value of 6346 which is greater than 12 and an even number. The chart indicates

therefore that this field is a BLOB (6346-12)/2 bytes in length [3167 bytes]

- **00** This serial type code indicates that the field is NULL
- **00** This serial type code indicates that the field is NULL

The serial type code for the *response_object* indicates that this field is a BLOB 1764 bytes in length. The entire database page would not be big enough to store this BLOB and the cell is even less capable. Figure 9 shows each cell highlighted alternately blue and green:



Double click to enlarge

Figure 9

The first blue shaded area is the cell the elements of which are duscussed above. The last four bytes of this cell highlighted in darker blue is the hex value **00 00 11 7A** which when decoded

big endian gives the value 4474. This is the page number of the first Overflow page for this cell and is consistent with the information found in the Pointer Map page discussed above.

Next Post

Following on from my earlier SQLite blog posts James Crabtree has been kind enough to code a Varint decoder and Alex Caithness of CCL has supplied me with his fully featured SQLite record recovery tool EPILOG. I'll review this software next time. Thanks to James and CCL.

References

http://www.sqlite.org/fileformat.html
http://www.sqlite.org/fileformat2.html

Posted by DC1743 at 23:10

No comments:

Post a Comment

Links to this post

Create a Link

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)