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

Wednesday, 27 April 2011

Carving SQLite databases from unallocated clusters

Have you missed me?

Background

Carving SQLite databases from unallocated clusters is problematic because although these types of database have a header, there is no footer and the length of the file is not normally stored within the file either. Given that SQLite databases are used by so many programs now (e.g.Firefox, Google Chrome and numerous Mac OSX and IoS applications) to store data of forensic interest it would be useful to recover them from unallocated clusters. I am aware that there has been some comment in this area within our industry blogs and forums. A paper entitled *Forensic analysis of the Firefox 3 Internet history and recovery of deleted SQLite records* written by *Murilo Tito Pereira* became available in 2009 (at a cost) but my interest was piqued more recently when *Rasmus Riis* (who I believe works for Law Enforcement in Denmark) posted an enscript he had written - *Chrome SQlite db finder v1.4* to Guidance Software's enscript download center. *Rasmus Riis's* approach to carving SQLite files used by Google Chrome is to identify the header and then carry out additional checking throughout the file for known values within *page* headers. I had mixed results using this enscript and also wondered whether it could be adapted to search for other SQLite databases, in particular the SMS.db used by the iPhone. So as a result I took a closer look at the problem.

SQLite Database Structure and File Format

The SQLite file format is well documented at http://www.sqlite.org/fileformat.html and also at http://www.sqlite.org/fileformat2.html. What I will try to do here is pick out the salient points that may be relevant to carving SQLite files from unallocated clusters, and also provide some commentary where it may be useful.

Size and numbering

- 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

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

Factory Clock

Please Subscribe To This B

Posts

Comments

(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*

Possible Page Types
Each page is used exclusively to store one of the following:

- 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

However not every database will include all of these items.

The first *page* (*page* 1)
The first *page* of the database is a special page for two reasons; it contains within the first 100 bytes of the *page* the **database header** and it also contains the Database Schema (the structure of the database [tables, indexes etc.] described in formal language).

The database header begins with the following 16 byte sequence:

0x53 0x51 0x4c 0x69 0x74 0x65 0x20 0x66 0x6f 0x72 0x6d 0x61 0x74 0x20 0x33 0x00

which when read as UTF-8 encoded text reads *SQLite format 3* followed by a nul-terminator byte.

Other significant values at offsets within the **database header** are as follows:

- **Offset 16 2 bytes 16 bit integer read big endian** Page Size
- **Offset 28 4 bytes 32 bit integer read big endian** The logical size of the database in pages which is only populated when the database was last written by SQLite version 3.7.0 or later. This field is only valid if it is nonzero and in all examples of SQLite databases I have examined this value was zero, so unfortunately not as exciting as it first appears!
- **Offset 32 4 bytes 32 bit integer read big endian** Page number of first freelist trunk page
- **Offset 36 4 bytes 32 bit integer read big endian** Total number of freelist pages including freelist trunk pages

- **Offset 52 4 bytes 32 bit integer read big endian**
  The highest numbered root page number if the database is auto-vacuum capable, for non-auto-vacuum databases, this value is always zero. The majority of the databases we are likely to be interested in are non-auto-vacuum databases.

B-Tree *pages*

The SQLite.org file format notes helpfully state that:

A large part of any SQLite database file is given over to one or more B-Tree structures. A single B-Tree structure is stored using one or more database pages. Each page contains a single B-Tree node. The pages used to store a single B-Tree structure need not form a contiguous block.

So from a carving perspective we note that most of what we wish to carve are B-Tree pages but they are not necessarily stored contiguously. We can however identify B-Tree pages because they have a *page* header 8 bytes in length if the page is a leaf node page and 12 bytes in length if the page is an internal node page. In all *pages,* with the exception of *page* 1, the header starts at the beginning of the page at offset 0. On *page* 1 the header starts at offset 100.

The first byte of all B-Tree *page* headers is a flag field, each flag is detailed as follows:

- **0x02**
  flag indicating index B-Tree internal node
- **0x0A**
  flag indicating index B-Tree leaf node
- **0x05**
  flag indicating table B-Tree internal node
- **0x0D**
  flag indicating table B-Tree leaf node

These flags therefore allow us to potentially identify B-Tree pages (of all types) by examining the first byte of each *page* to see if it is either 0x02, 0x0A, 0x05 or 0x0D.

The B-Tree *page* header also contains some other potentially useful values (offsets from start of page):

- **Offset 1 2 bytes 16 bit integer read big endian**
  Byte offset of first block of free space on this page (0 if no free blocks)
- **Offset 3 2 bytes 16 bit integer read big endian**
  Number of entries (cells) on this page
- **Offset 5 2 bytes 16 bit integer read big endian**
  Byte offset of the first byte of the cell content area relative to the start of the page. If this value is zero, then it should be interpreted as 65536

Freelist pages

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)

► May (2)

▼ April (1)

Carving SQLite databases from unallocated clusters...

► January (1)

► 2010 (21)

► 2009 (25)

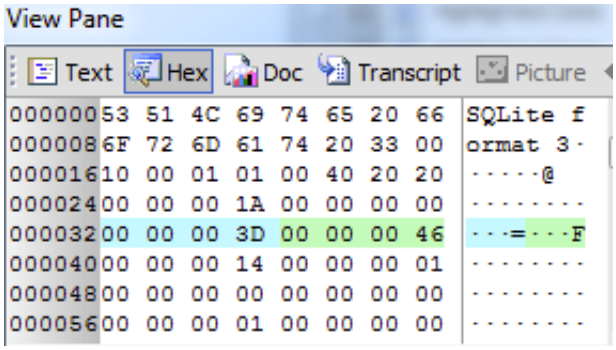
► 2008 (24)

Once again the SQLite.org file format notes help us out:

A database file might contain one or more pages that are not in active use. Unused pages can come about, for example, when information is deleted from the database. Unused pages are stored on the freelist and are reused when additional pages are required.

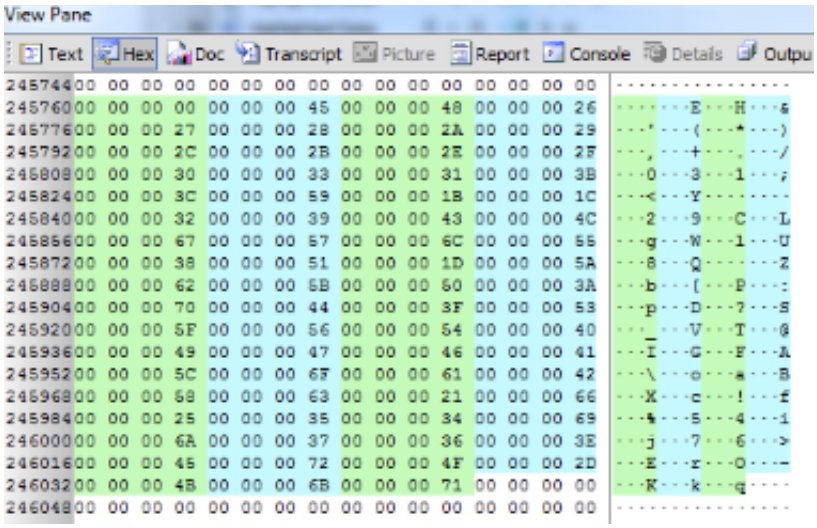
Each page in the freelist is classified as a freelist trunk page or a freelist leaf page. All trunk pages are linked together into a singly linked list. The first four bytes of each trunk page contain the page number of the next trunk page in the list, formatted as an unsigned big-endian integer. If the trunk page is the last page in the linked list, the first four bytes are set to zero.

The operation of the freelists might be better understood by a quick forensic examination of them:



The four bytes highlighted in blue are at offset 32 within the database header and decoded as a 32 bit integer big endian give a decimal value of 61 - this is the page number of the first freelist trunk page. The four bytes highlighted in green are at offset 36 within the database header and decoded as a 32 bit integer big endian give a decimal value of 70 - this is the total number of free pages including freelist trunk pages.

The page number of the first freelist trunk page is 61. To calculate the offset from the start of the database for this page we use the formula ***offset = (page number-1) x page size*** which in this case is **(61-1) x 4096 = 245760.**



The offset 24570 takes us to start of the first freelist trunk page. There we find an array of 4 byte big endian integers. The first four bytes (highlighted in green) decoded big endian denote the page number of the next freelist trunk page - in this case the value of the first four bytes is zero indicating that there is no more free freelist trunk pages. The second four bytes (highlighted in blue), in this case **0x00 0x00 0x00 0x45** when decoded as a 32 bit integer big endian give a value of decimal 69 - this is the number of leaf page pointers to follow. The remaining 69 blocks of 4 bytes (alternately highlighted in green and blue in the screen shot) each represent, when decoded as a 32 bit integer big endian, the page number of a free page. Examination of each free page revealed that the entire page had been zeroed out - although this may be a function of the application using the database, not a function of SQLite itself.

Pointer map pages

Pointer map pages will only exist if the database is auto-vacuum capable and the value within the 4 bytes of the database header at offset 52 is non zero. In a database with pointer map pages, the first pointer map page is page 2. The first byte of a pointer map page is one of five values 0x01, 0x02, 0x03, 0x04 or 0x05. Many of the databases we have a forensic interest in are not auto-vacuum capable and therefore do not have pointer map pages, however where they do (iPhone SMS.db for example) it is possible to calculate the length of the of the SQLite database by extrapolating information from the pointer map page entries. I will cover this in my next blog post.

Locking Page

The locking page is the database page that starts at byte offset 2^{30} (1,073,741,824) and always remains unused (i.e all zeros). Most databases will not be big enough (> 1GB) to require a locking page.

Overflow pages

Once again the SQLite.org file format notes help us out:

Sometimes, a database record stored in either an index or table B-Trees is too large to fit entirely within a B-Tree cell. In this case part of the record is stored within the B-Tree cell and the remainder stored on one or more overflow pages. The 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 can calculate that for the first byte to be any value other than 0x00 there must be at least 16,777,216 pages within the database (0x01 0x00 0x00 0x00 decoded big endian). At a page size of 4096 bytes this would equate to a database size of 64 Gigabytes. In most cases therefore we can discount the first byte of overflow pages being anything other than 0x00.

So how does this help with carving SQLite databases then?

We can use the database header and the first byte value of each *page* thereafter to determine whether the data within each *page* size block is valid. So from a carving perspective we can identify the first *page* with the database header, calculate the *page* size, read the next *page* and validate the first byte and so on until the *first byte validation* fails.

This essentially is what *Rasmus Riis's Chrome SQlite db finder v1.4* enscript does for SQLite databases created by the Google Chrome web browser. His description of the enscripts functionality states that it checks the first byte of each *page,* other than the first page, for the values *13,10, 2, 5 or 0.* Convert these values into hex and you get 0x0D, 0x0A, 0x02, 0x05 and 0x00. The first four of these values are the flags found at the first byte of the B-Tree page headers discussed above. Additionally he checks for 0x00 which may well be the first byte of either a free page or an overflow page. The script does not however allow 0x01, 0x03 or 0x04 to be the first *page* byte. This therefore does not allow for an auto-vacuum capable database. The databases used by Google Chrome are not auto-vacuum capable, however other databases such as the iPhone SMS.db database are.

*Rasmus's en*script also carries out a test of the two bytes at offset 100 within the first database *page*. The enscript according to the description in the download center looks for the values 2243, 3853 or 3331. The coding however shows that it checks for 3343 or 3853 or 3331 decoded big endian. Converted to hex the first two bytes would be **0x0D 0x0F** or **0x0F 0x0D** or **0x0D 0x03.** I am not sure what *Rasmus* had in mind for the second value 3853 (which is **0x0D 0x0F** converted little endian) but focussing on the first and the third pairs of two bytes the script is taking the flag byte and the first byte of the two bytes used to store the Byte offset of first block of free space on the *page.* The first *page* of all SQLite databases beyond the 100 byte database header store the

database schema and are therefore constant (in other words because each particular database has a unique set of tables and indexes the first page of a particular database does not change from instance to instance). Because the database schema does not change the combining of the first and second bytes seems to work in order to identify Chrome databases.

This lead me on to consider whether an analysis of the following values within the B-Tree page header of the first page containing the database schema would allow the identification of other types of SQLite databases. The following table appears to suggest that it would be possible to establish a *fingerprint* for each database type. I have not collected enough test data to be certain about this yet.

Type	pg1 B-tree Page Header Offset 1	pg 1 B-tree Page Header Offset 3	pg1 B-tree Page Header Offset 5	Page Size
Google Chrome History	0F FC	00 15	05 06	4096
Google Chrome downloads.sqlite	00 00	00 01	02 55	1024
Firefox places.sqlite	00 00	00 01	0F FB	4096
iPhone SMS.db	0F FC	00 11	02 45	4096
Safari Top Sites	0F FC	00 02	0E AA	4096

With regards to *Rasmus's en*script because he was kind enough to share and also not Enpack it is possible to make some small changes to the code to allow it to parse unallocated for all types of SQLite.db. I am in touch with my programming friends to create a script that can carve and identify SQLite databases from the *fingerprint* discussed above and also include a greater amount of error checking.

Other stuff of note
When you examine data within SQLite databases have you noticed that most of the meaningful stuff is bunched up at the end of each *page*? The SQLite.org file format notes help us out here:

The total amount of free space on a b-tree page consists of the size of the unallocated region plus the total size of all freeblocks plus the number of fragmented free bytes. SQLite may from time to time reorganize a b-tree page so that there are no freeblocks or fragment bytes, all unused bytes are contained in the unallocated space region, and all cells are packed tightly at the end of the page. This is called "defragmenting" the b-tree page.

To Do
I have not as yet covered the part the journal files may play in the recovery of SQLite data from unallocated, a future post perhaps.

Thanks
to *Rasmus Riis* for sharing his enscript
and to Chris Vaughan for his help in firming up some of the theory.

References

http://www.sqlite.org/fileformat.html
http://www.sqlite.org/fileformat2.html
http://www.sqlite.org/requirements.html
http://www.sqlite.org/hlr30000.html
https://support.guidancesoftware.com/forum/downloads.php?do=file&id=1116

Posted by DC1743 at 19:57

5 comments:

Anonymous said...

Welcome back and fascinating post - thanks!

One comment: Wouldn't it be easier to carve individual records since the headers are fairly well defined for each database type AND you will get complete records out rather than partial pages possibly in the wrong order?

I think a carver for records would be fairly easy.

Regards,
James

4 May 2011 10:22

Anonymous said...

Looking into this a bit further, is it possible that Chrome uses a modified record structure? The following test data from a history record doesn't make sense against the spec:

0A 00 81 11 0D 01 01 06 01 01 68 74 74 70 3A 2F 2F 77 77 77 2E 6F 70 65 72 61 2E 63 6F 6D 2F 64 6F 77 6E 6C 6F 61 64 2F 67 65 74 2E 70 6C 3F 69 64 3D 33 33 34 32 33 26 74 68 61 6E 6B 73 3D 74 72 75 65 26 73 75 62 3D 74 72 75 65 00 00 00 2D FD 09 83 93 96 00 01 00

The \x81 gives a field length of (129 - 13) / 2 = 56. However the actual string in there is "http://www.opera.com/download/get.pl?id=33423&thanks=true&sub=true" which is longer at 66!

However, Firefox SQLite records can be carved easily from their headers.

Regards,
James

4 May 2011 10:54

DC1743 said...

This post was intended to be generic and cover all potential sqlite databases. Carving individual records may be successful where all the required information is within one table.

Richard

4 May 2011 12:18

DC1743 said...

With regards to the chrome example your Huffman decoding is incorrect. Because 0x81 is greater than 128 you have to take into account the next byte 0x11. Call these bytes x and y and use the formula (x-128) x 128 + y to calculate N.

(129-128) x 128 + 17 = 145

Then use the formula you used (N-13)/2

(145-13)/2 = 66

4 May 2011 23:02

Anonymous said...

Thanks Richard,
I didn't read the SQL record format paper through the first time. I just finished working on a 'variable length integer' decoder function - nightmare!

Regards,
James

5 May 2011 18:54

Post a Comment

Links to this post

Create a Link

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)