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

Tuesday, 10 May 2011

An analysis of the record structure within SQLite databases

My two previous posts Carving SQLite databases from unallocated clusters and SQLite Pointer Maps pages looked at the structure of SQLite databases as a whole. Information contained in those posts may hopefully facilitate the carving of complete SQLite databases. This post is aimed at examining the potential of carving individual records within an SQLite database but should be read in conjunction with the Carving SQLite databases from unallocated clusters post.

Background

Carving individual SQLite database records in certain circumstances may be more fruitful than carving whole databases. There are in fact a number of applications that do exactly this for some types of SQLite database. For example Firefox 3 History Recovery (FF3HR) is an application written to recover Firefox records. A paper entitled *Forensic analysis of the Firefox 3 Internet history and recovery of deleted SQLite records* written by *Murilo Tito Pereira* also deals with the recovery of Firefox records.

SQLite databases can be considered as a mini file system in their own right. Within this file system are areas that are marked as free that may contain deleted data. Record based recovery may help identify records that have been deleted but are still contained within the parent SQLite database. More obviously record based recovery is indicated where only deleted and partially overwritten databases are available. However for record based recovery to be useful the data you wish to recover must be stored within one table within the SQLite database concerned. If it is necessary to query two or more tables to extract useful data record based recovery is probably not going to be appropriate.

Table Record

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

Factory Clock

Please Subscribe To This B

Posts

Comments

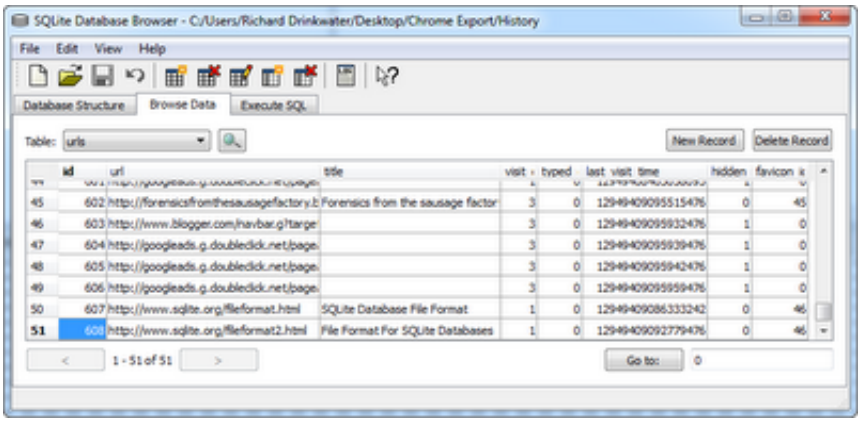

Figure 1

Figure 1 shows, as viewed with the *SQLite Database Browser* software, a record within the Google Chrome History file URL table at row ID 608. This record is stored within the Google Chrome History SQLite database within a B-tree table leaf node in an area known as a cell. It can be seen from the column headers that this record consists of an *id*, a *url*, a *title*, a *visit_count*, a *typed_count*, a *last_visit_time*, a *hidden* flag and lastly a *favicon _id*. To aid viewing I will repeat the record data below:

- **ID**
  608
- **URL**
  http://www.sqlite.org/fileformat2.html
- **title**
  File Format For SQLite Databases
- **visit_count**
  1
- **typed_count**
  0
- **last_visit_time**
  12949409092779476
- **hidden**
  0
- **favicon_id**
  46

The **urls** table is stored within one table B-tree which will consist of a root page and possibly a number of internal and leaf node pages. I have established that the data representing the record detailed above is stored in a cell that exists within a B-tree table leaf node database page.

Cells

Lets recap some of the information dealt with in the earlier post Carving SQLite databases from unallocated clusters. SQLite databases are divided into equal sized pages, the size of which is detailed in two bytes, decoded as a 16 bit integer big endian, at offset 16 of the database file within the

database header. Most of an SQLite database consists of B-tree structures consisting of one or more B-tree pages. Each B-tree page has either an 8 or 12 byte page header (depending on whether it is a leaf or internal node).

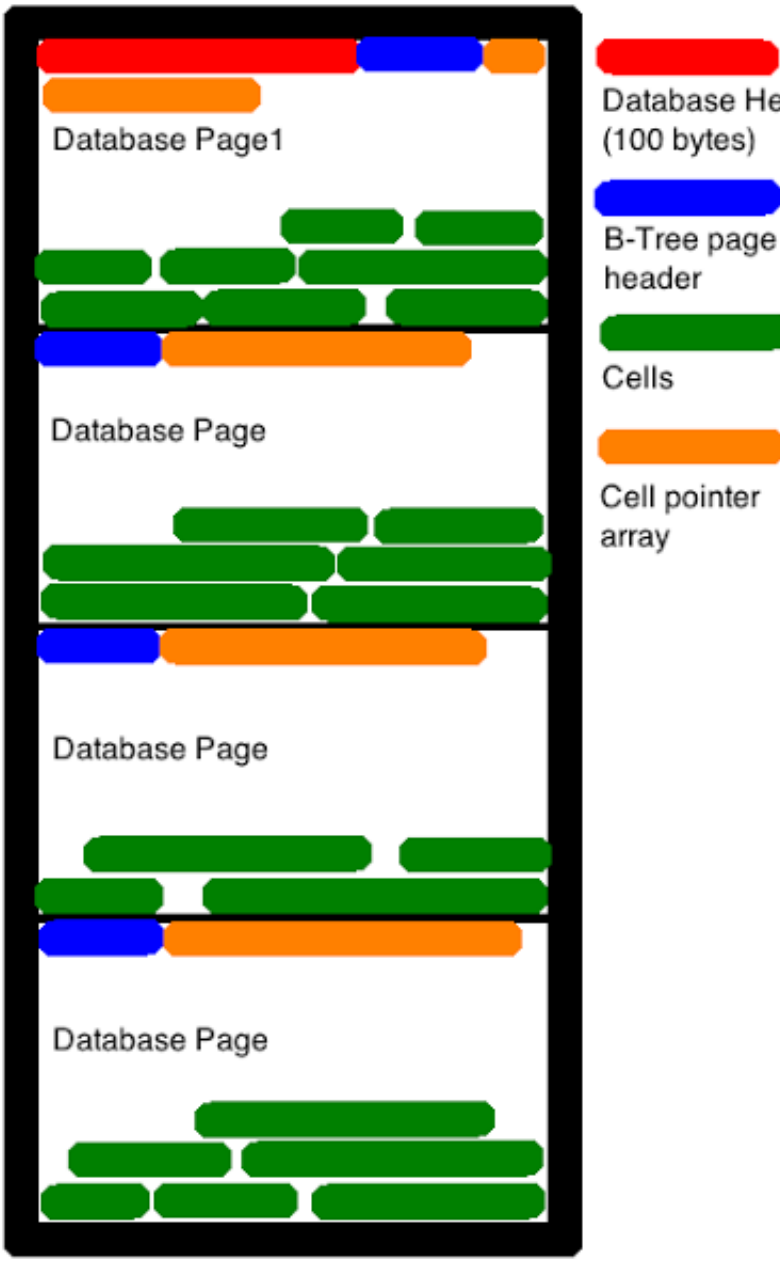

Figure 2

As can be seen in Figure 2 the cells tend to be at the end of each database page in an area referred to as the *cell content area*. These cells are used to store the database records, one record per cell. The first cell to be written in a database page is stored at the end of the page and additional cells work back towards the start of the page.

The number of cells and their location within a database page is stored within the B-Tree page header at the following offsets.

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)

An analysis of the record structure within SQLite ...

SQLite Pointer Maps pages

► April (1)

► January (1)

► 2010 (21)

► 2009 (25)

► 2008 (24)

- **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 the 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

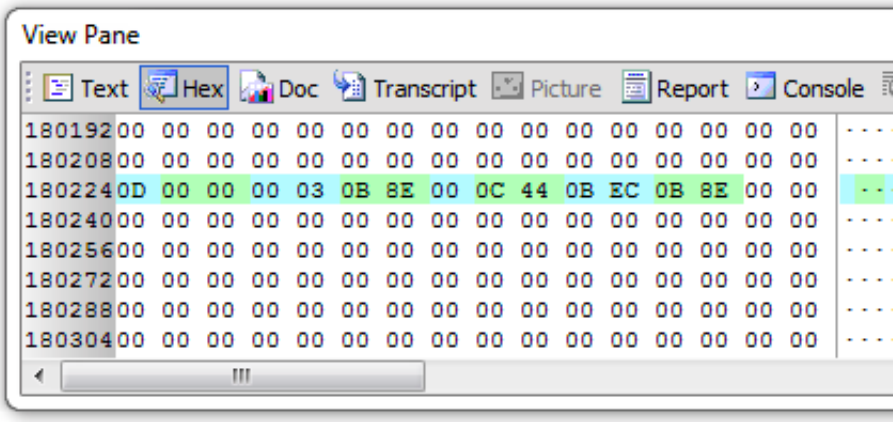

Figure 3 Page Header and Cell Pointer array

Figure 3 shows the page header of the B-tree leaf node page that contains the record detailed in Figure 1 above. 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 **0B 8E** decoded big endian give a value of 2958 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 **0C 44**, **0B EC** and **0B 8E** are the cell pointer array for this page. The SQLite.org file format notes helpfully state that *the cell pointer array of a b-tree page immediately follows the b-tree page header. Let K be the number of cells on the b-tree. The cell pointer array consists of K 2-byte integer offsets to the cell contents. The cell pointers are arranged in key order with left-most cell (the cell with the smallest key) first and the right-most cell (the cell with the largest key) last*. The *key* value referred to is the row ID. In this case we have three cells and therefore three offsets which when decoded big endian are 3140, 3052 and 2958. These offsets allow us to find the start of each cell, it is worth pointing out that there may be free blocks or fragments between each cell so we can not use the offsets to determine the length of each cell.

The record detailed in Figure 1 is contained within the cell at offset 2958 within the page. We will decode the contents of this cell but first we better look at the make up of a cell.

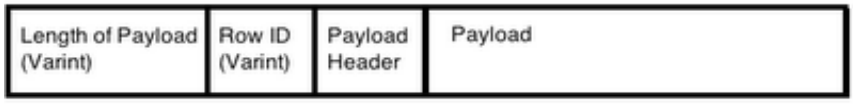

Figure 4 Cell make up

Figure 4 indicates four areas of interest. The *payload* is the data forming the record as detailed in this example in Figure 1 and as suggested in the diagram it is stored in a serialized way with all the relevant data concatenated together. The *payload header* details how we can identify each field within the concatenated data (*see the Payload Header section below for details of how this works*). The Row ID number and the *Payload* length are stored using *variable length integers* known as *varints.* To successfully decode the Cell and Payload headers we have to be able to decode a *varint.*

Varint

http://www.sqlite.org/fileformat2.html and http://www.sqlite.org/fileformat.html#varint_format provides some detail in respect to how *varints* are structured. I will try here to simplify things and provide a few example decodings when we decode the cell relating to the record detailed at figure 1.

- *Varints* are variable length integers between 1 and 9 bytes in length depending on the value stored
- They are a static Huffman encoding of 64-bit twos-complement integers that uses less space for small positive values
- Where the most significant bit of byte 1 is set this indicates that byte 2 is required, where the most significant bit of byte 2 is set this indicates that byte 3 is required, and so on
- *Varints* are big-endian: bits taken from the earlier byte of the *varint* are the more significant than bits taken from the later bytes
- Seven bits are used from each of the first eight bytes present, and, if present, all eight from the final ninth byte

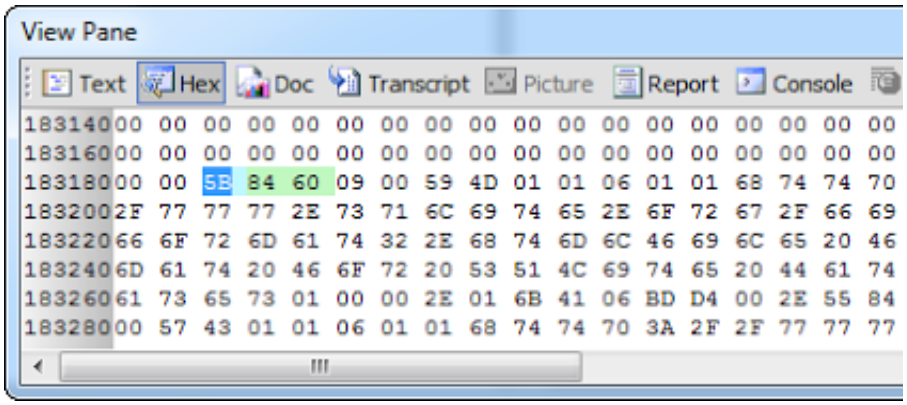

Figure 5

Figure 5 shows the beginning of the cell at offset 2958 within the page. As shown in figure 4 the first value is the payload length represented by a *varint*. The first byte is **5B**. We have to establish the value of the most significant bit and this can be done by converting the hex 5B to binary:

Hex	UInt8	Int8	Binary
5b	91	91	01011011

It can be seen that in this case the most significant bit is zero and therefore not set. This *varint* is only one byte long and indicates that the payload length is 91 bytes. The payload length is the length in bytes of both the payload header and the payload.

The next byte is **84**. Converting this to binary:

Hex	UInt8	Int8	Binary
84	132	-124	10000100

The most significant bit here has the value of one and therefore is set. This indicates that this *varint* includes, at least, the next byte **60** which converted to binary:

Hex	UInt8	Int8	Binary
60	96	96	01100000

It can be seen that the most significant bit is zero and therefore not set. This byte therefore is not followed by another and is the last byte of this *varint*. To establish the value of this *varint* we now have to take the least significant 7 bits of each of the two contributing bytes and concatenate them together:

00001001100000

We discard the leading zeros and convert the binary 1001100000 to decimal, giving a value of 608. This *varint* represents the row ID and we can see in figure 1 that the row ID is confirmed as 608.

The calculation we have carried out can be represented by a formula. If we say that the value of the *varint* is **N** and the unsigned integer value of the first byte is **x** and the unsigned integer value of the second byte is **y** we can use the formula:

N = (**x**-128) x 128 + **y**

If we substitute the value of our unsigned integers 132 and 96 into the formula:

(132-128) x 128 + 96 = 608

This formula works for two byte *varints* that can represent a maximum value of 16383. I suspect we are not likely to encounter larger *varints* in the SQLite databases we have an interest in with the possible exception of SQLite databases used to store browser cache. It is also worth noting that the most significant bit if included and allowed to contribute to the unsigned integer value would have a value of 128 (hence the [x-128]). Therefore if the first byte of a *varint* is less than 128 you can exclude the possibility of there being a second byte in the *varint*.

Payload Header and Payload

We have already looked at two of the four areas of interest within a cell, the payload length and row ID. Next up is the *Payload Header* and *Payload*.

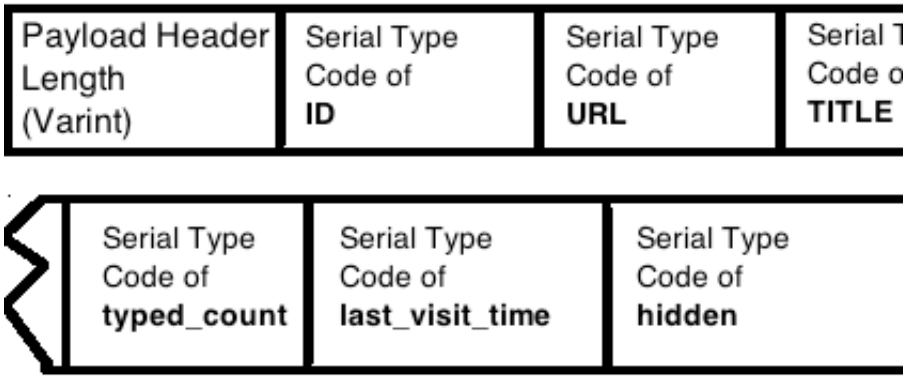

Figure 6 Payload Header make up

Figure 6 shows the make up of the *payload header* of a record within the URLs table of the Google Chrome History SQLite database. The *payload* is the data forming the record stored in a serialized way with all the relevant data concatenated together. The *payload header* details how we can identify each field within the concatenated data and will vary from table to table, the contents of which is dictated by the fields required in each record. All payload headers will have however a *Payload Header Length* followed by one or more *Serial Type Codes*. The *Serial Type Code* is used to denote the type of data found in a field within the payload and it's length. All possible Serial Type Codes are *varints* and are detailed in a chart provided by SQLite.org at Figure 7:

Serial Type Codes Of The Record Format

Serial Type	Content Size	Meaning
0	0	NULL
1	1	8-bit twos-complement integer
2	2	Big-endian 16-bit twos-complement integer
3	3	Big-endian 24-bit twos-complement integer
4	4	Big-endian 32-bit twos-complement integer
5	6	Big-endian 48-bit twos-complement integer
6	8	Big-endian 64-bit twos-complement integer
7	8	Big-endian IEEE 754-2008 64-bit floating point number
8	0	Integer constant 0. Only available for schema format 4 and higher.
9	0	Integer constant 1. Only available for schema format 4 and higher.
10,11		*Not used. Reserved for expansion.*
N≥12 and even	(N-12)/2	A BLOB that is (N-12)/2 bytes in length
N≥13 and odd	(N-13)/2	A string in the database encoding and (N-13)/2 bytes in length. The nul terminator is omitted.

Figure 7

Lets have a look at the Payload Header of our example record detailed in Figure 1.

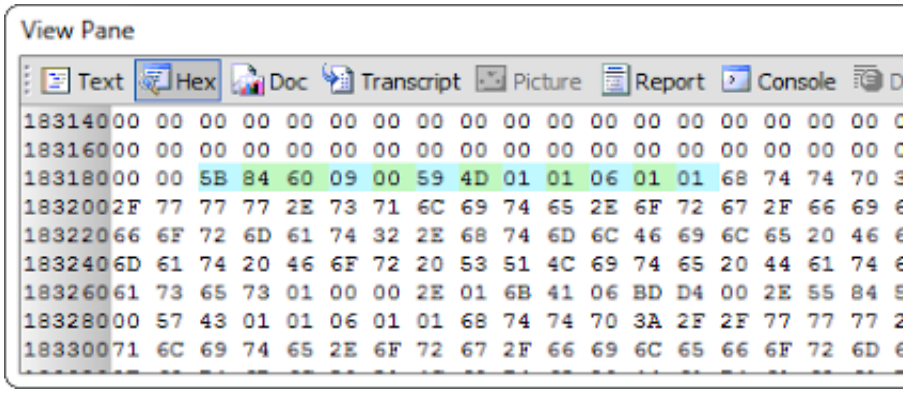

Figure 8

Figure 8 shows highlighted in blue and green the first three elements of the Cell make up shown in Figure 4 - the *Payload Length*, the *Row ID* and the *Payload Header.* We have already decoded the *Payload Length* **5B** and the *Row ID* **84 60**. The next byte **h09** denotes the length of the *Payload Header* which is in this case 9 bytes (including the *Payload Header Length* byte). It can be seen therefore that the remaining 8 bytes shown in hex are **00**, **59**, **4D**, **01**, **01**, **06**, **01** and **01**. These bytes represent *varints* so we have to consider that a value may be represented by more than one byte, however in this case the unsigned integer value of each byte is less than 128. We can conclude therefore that each *varint* is only a single byte in length. To determine what each *varint* indicates we have to consult the Serial Type Code chart shown at figure 7. Each Serial Type Code details the type and length of the data in the *payload* that follows the *payload header*. The multi byte integers are decoded big endian.

- **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 (see figure 1) 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.*
- **59** This serial type code has a value of 89 which is greater than 13 and an odd number. The chart indicates therefore that this field is a text string (89-13)/2 bytes in length [38 bytes]
- **4D** This serial type code has a value of 77 which is greater than 13 and an odd number. The chart indicates therefore that this field is a text string (77-13)/2 bytes in length [32 bytes]
- **01** This serial type code has a value of 1 indicating the next field is an 8 bit integer using 1 byte
- **01** This serial type code has a value of 1 indicating the next field is an 8 bit integer using 1 byte
- **06** This serial type code has a value of 6 indicating the next field is an 64 bit integer using 8 bytes
- **01** This serial type code has a value of 1 indicating the next field is an 8 bit integer using 1 byte
- **01** This serial type code has a value of 1 indicating the next field is an 8 bit integer using 1 byte

It can be seen that our *payload* is 82 bytes in length (38+32+1+1+8+1+1). The *payload header* was 9 bytes and therefore the overall *payload length* is 91 (82+9) bytes, as previously calculated, and represented by the byte **5B**.

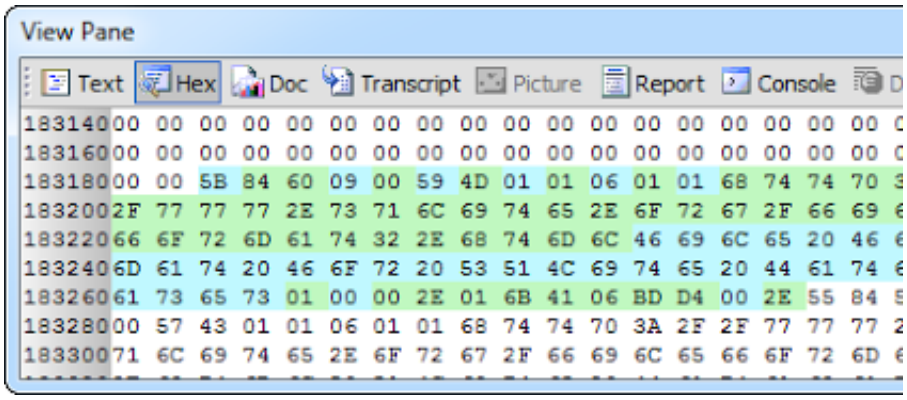

Figure 9

Figure 9 shows each element of the payload highlighted alternately in green and blue. The first element is **http://www.sqlite.org/fileformat2.html** 38 bytes in length, the next element is **File Format For SQLite Databases** 32 bytes in length. The next element is represented by the byte **01** which denotes the visit_count of 1. This is followed by the byte **00** denoting the typed_count of 0. Next are the eight bytes **00 2E 01 6B 41 06 BD D4** decoded as a 64 bit integer big endian giving a value of 12949409092779476, the last_visit_time (stored in the Google format). The next byte is **00**, the hidden flag, followed lastly by **2E** decoded as 46, the favicon_ID. The next record in this case immediately follows at offset 3052 within the database page.

Notes

I have glossed over some possible combinations of data found in stored records in order to try and simplify things a little. It is possible for a record to require more space than the space available in a cell within one database page. In this eventuality pages known as overflow pages come into play. I will leave any commentary on this to another day :-)

Carving Considerations

It can be seen that each record of the Google Chrome History URL table may vary in content and length. This precludes simple carving of records using known headers. It may be possible to define a scheme to assist with carving however by focussing on the parameters of each element of the record. It is clear that for the Google Chrome History URL table the scheme would be fairly complicated, allowing for very large URLs and Page titles which may well induce many false positives. For databases using a simpler record structure things are a bit easier. A presentation presented by Alex Caithness of CCL details an approach that can be adopted for carving iPhone calls.db databases.

Deleted Data within Live Databases

This area will require another blog post on another day! I am aware of two programs possibly written to recover this deleted data. SQL Undeleter from Chirashi Security and Epilog from CCL. If the developers will let me test these programs out I will report the results to you.

References

http://www.sqlite.org/fileformat.html
http://www.sqlite.org/fileformat2.html
http://www.ccl-forensics.com/images/f3%20presentation3.pdf
http://mobileforensics.wordpress.com/2011/04/30/sqlite-records/

Posted by DC1743 at 18:11

+1 Recommend this on Google

2 comments:

Pernille said...

This comment has been removed by the author.
10 May 2011 21:20

DBC said...

Great work on these database files. I wish I had this a month ago when I was digging through the same processes to carve out records from unalloc.

11 May 2011 03:37

Post a Comment

Links to this post

Create a Link

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)