InnoDB Database Forensics

Peter Frühwirt, Markus Huber
Vienna University of Technology
Vienna, Austria
markus.huber@tuwien.ac.at

Martin Mulazzani, Edgar R. Weippl
SBA Research
Vienna, Austria
{mmulazzani|eweippl@sba-research.org}

Abstract— **Whenever data is being processed, there are many places where parts of the data are temporarily stored; thus forensic analysis can reveal past activities, create a (partial) timeline and recover deleted data. While this fact is well known for computer forensics, multiple forensic tools exist to analyze data and the systematic analysis of database systems has only recently begun. This paper will describe the file format of the MySQL Database 5.1.32 with InnoDB Storage Engine. It will further explain with a practical example of how to reconstruct the data found in the file system of any SQL table. We will show how to reconstruct the table as it is, read data sets from the file and how to interpret the gained information.**

Keywords- MySQL, InnoDB, Database and Forensic

I. INTRODUCTION

Today's society would be unimaginable without database systems. The information forms an important commodity. Most of the information is stored and processed in databases. Most of this information can be very precious for its owner as it concerns data regarding patients or customer information, which is covered by the relevant data protection. The results and information gained from database forensics can be used for several reasons. First, for many companies it is relevant to know if the integrity of their data has been compromised or users' privacy has been violated [14]. This means that they need or want to prove, whether or not, their databases has been tampered with. Second, results of database forensics can be used to detect and analyze attacks, understand which vulnerabilities were exploited and to develop preventive countermeasures. Third, modern file systems develop in the direction of database systems and thus database forensic will also become important for file forensics.

Common file systems, such as FAT, NTFS, HFS+ or ext2/ext3 are well understood in the forensic community [8] and are supported by many forensic tools, for example EnCase [23] or The Sleuth Kit [24]. Exotic or new file systems are much harder to analyze as these tools do not support them. This makes it increasingly difficult as new file systems are in constant development. Recent research has been focusing on more advanced file systems that partially rely on data structures found in database systems. These include IBM's JFS [4] or Journaling File Systems [5], ext3 and reiserFS [15]. New and upcoming file systems include Btrfs and ext4, which are already included in many Linux distributions. Btrfs is based on the well known data structure called B-tree [3] that is also used in database indices [2].

Clearly, database systems are bound to leave more extensive traces than "normal" files as they not only store files but, in addition, need indexes, rollback segments and log files. Therefore, the analysis of the structure in the database management system is a precondition for forensic analysis [9, 10, 11].

To analyze the data, it is important to know and understand in detail, how the database is built. Only with this basic knowledge can one prove the consistency of the data. The goal of this paper is to explain how to analyze the database system MySQL with the storage engine InnoDB. This paper shows the structure and architecture of this database system and how one can reconstruct data from the files in the file system. Based on these findings we provide an application that reads and interprets this basic data to facilitate the detection of any unauthorized manipulation.

In the first section of this paper we will explain the basis of a simple SQL-table and how all relevant information like keys, types and others are saved and how they can be read from the files. In the second part of this paper we will show how data in tables can be saved with the help of the InnoDB storage in the files and how to interpret this data. We refer to the MySQL source code in all our explanations[1]. While the MySQL documentation (such as [18-22]) provides a good starting point; none of the presented material has to the best of our knowledge been documented anywhere else.

II. SQL INTERNALS

Storage engines of MySQL store all information of each single table in a .frm file in the directory of the database. The filename is the name of the table [1]. Each .frm file is created by the function create_frm() in the file /sql/table.cc [Appendix 1]. The file size of a single .frm file is limited to 4 GB. If this limit is reached MySQL will automatically truncate the file to prevent errors [13,16].

A. General Table Information

All general information of the file, such as the used version number, is saved at the first bytes of the .frm file. Table 1 shows the structure of the first bytes of a .frm file starting at position 0x0. Table 2 explains the meaning of the bytes. All references to the line numbers of the source code refer to the file /sql/table.cc [Appendix 1].

[1] All information of the data structures and source code refers to the MySQL database management system in the version 5.1.32 for apple-darwin9.5.0 on i386 (Mac OS X 10.5.7 Leopard).

TABLE I. HEXADECIMAL STRUCTURE OF .FRM FILE (0X00000000 - 0X0000004F)

0x00000000	FE 01 0A 0C 03 00 00 10 01 00 00 30 00 00 69 01	.....0..i.
0x00000010	10 01 00 00 00 00 00 00 00 00 00 02 21 00 09 00	.....!...
0x00000020	00 05 00 00 00 00 30 00 02 00 00 00 00 00 00 69	.....0.....i
0x00000030	01 00 00 D4 C3 00 00 10 00 00 00 00 00 00 00 00	.....
0x00000040	2F 2F 00 00 20 00 00 00 00 00 00 00 00 00 00 00	//.. .....

TABLE II. EXPLANATION AND MEANING OF THE HEXADECIMAL VALUES OF THE .FRM FILE (0X00000000 - 0X00000040)

Offset	Length	Value	Meaning	Source Code
0x00	1	FE	fixed value	2454
0x01	1	01	fixed value	2455
0x02	1	0A	FRM VERSION (/include/mysql_version.h [Appendix 2]) + 3 + test(create info->varchar)	2456
0x03	1	0C	Database type (sql/handler.h Z. 258-279) [Appendix 3]) z.B.: DB_TYPE_MYISAM: 9, DB_TYPE_INNODB: 12	2458-2459
0x04	1	03	unknown	
0x05	2	00 00	unknown or undefined	
0x07	2	10 01	IO_SIZE (4096) Definition in include/my_global.h [Appendix 13]	2461
0x09	2	01 00	unknown	
0x0A	4	00 30 00 00	keylength (IO SIZE+key length+reclength+ create_info->extra_size). Table 3 shows exact meaning of the key length (key_length)	2474-2477
0x0E	2	69 01	length of the temporary key, based on key_length	2478-2479
0x10	2	10 01	length of the record	2480
0x12	4	00 00 00 00	create info->max rows (definition of the create_info structure HA_CREATE_INFO in sql/handler.h [Appendix 3] in line 896-924)	2481
0x16	4	00 00 00 00	create_info->min_rows	2482
0x1A	1	00	unused or padding / alignment	-
0x1B	1	02	fixed value (use long pack-fields)	1)
0x1C	2	21 00	key_info_length	1)
0x1E	2	09 00	create_info->table_options	2486-2487
0x20	1	00	fixed value	2488
0x21	1	05	fixed value (frm version number: 5)	2489
0x22	4	00 00 00 00	create_info->avg_row_length	2490
0x26	1	30	create info->default table charset	2491-2492
0x27	1	00	create_info->transactional | create info->page checksum << 2	2493
0x28	1	02	create_info->row_type	2495
0x29	6	00 00 00 00 00 00	RAID Support	2496-2502
0x2F	3	69 01 00 00	key_length	2503
0x33	4	D4 C3 00 00	MYSQL_VERSION_ID (only saved to prevent a warning, because of unaligned key_length of 3 bytes)	2504-2505
0x37	4	10 00 00 00	create_info->extra_size (length of extra data sequence)	2506
0x3B	2	00 00	extra rec buf length (reservation)	-
0x3D	1	00	default_part_db_type (reservation)	-
0x3E	2	00 00	create_info->key_block_size	2511

TABLE III. STRUCTURE OF A KEY DEFINED IN /SQL/UNIREG.CC [APPENDIX 4]

Length	Meaning (all keys)
8	key header
9	key parts(MAX_REF_PARTS)
NAME LEN	name of the key
1	separator NAMES_SEP_CHAR (before name of key)
Length	**Meaning (single key)**
6	key header
1	separator NAMES_SEP_CHAR (after name of key)
9	padding and alignment

Further information will not be saved up to the fixed address 0x1000. This space is filled with null-values. The keys of a table will be saved at position 0x1000. At this point we want to explain how one can reconstruct the following SQL table:

```
CREATE TABLE `Project` (
  `idProject` int(11) NOT NULL
              AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `deleted` tinyint(1) DEFAULT NULL,
  `startTime` date DEFAULT NULL,
  `endTime` date DEFAULT NULL,
  `effort` int(10) DEFAULT NULL,
  PRIMARY KEY (`idProject`)
```

```
) ENGINE=InnoDB DEFAULT CHARSET=latin
AUTO_INCREMENT=352;
```

Appendix 5 contains the Project.frm file of this table.

B. *Primary Key*

Table 4 shows the stored primary key in the file Project.frm [Appendix 5]. This can be found starting at address 0x1000.

TABLE IV. HEXADECIMAL STRUCTURE OF .FRM FILE (0X00000000 - 0X0000004F)

0x00001000	01 01 00 00 0A 00 00 00 04 00 01 00 00 00 01 80	.....
0x00001010	02 00 00 1B 00 04 00 FF 50 52 49 4D 41 52 59 FF	.....PRIM ARY.

All following references to the source code lines are from the file /sql/unireg.cc [Appendix 4]. The byte value in address 0x000001001 indicates how many keys exist in this table. The structure st_key (KEY) of "key" is defined in the file /sql/structs.h [Appendix 8] at line 72-101, the structure st_key_part_info (KEY_PART_INFO) of "key_part" on line 52-69. With this data you can reconstruct the primary key of this table (field: idProject) as follows:

TABLE V. KEY DEFINITIONS (0X00001000 - 0X0000101F)

Offset	Len	Value	Meaning		Source Code
0x04	2	0A 00	KEY HEADER	key->flags ^ HA_NOSAME dupp key and pack fields	521
0x06	2	00 00		key->key_length length of the key	522
0x08	1	00		key->key_parts sum or number of key parts	523
0x09	1	04		key->algorithm key algorithm from include/my_base.h (line 93-99) [Appendix 9]. The value 04 means in this case HA_KEY_ALG_FULLTEXT	524
0x0A	2	00 01		key->block_size	525
0x0B	3	00 00 00		unused (padding, alignment)	526
0x0E	2	01 80	KEY PART	key_part->fieldnr+1+FIELD_NAME_USED	540
0x10	2	20 00		offset (key_part->offset+data offset+1)	541-542
0x12	1	00		fixed value (sorting order)	543
0x13	2	1B 00		key_part->key_type	544
0x15	1	04	TS	key_part->length length of the key parts in bytes	545
0x16	1	00		unused (padding, alignment)	546
0x17	1	FF	separator NAMES_SEP_CHAR (before name of key)		551/555
0x18	7	50 52 49 4D 41 52 59	name of key (PRIMARY)		554
0x1F	1	FF	separator NAMES_SEP_CHAR (after name of key)		555

A table with a primary key over more fields than one would have a similar structure. The following MySQL table is given:

```
CREATE TABLE `pk2` (
  `field1` int(10) NOT NULL,
  `field2` int(10) NOT NULL,
  `field3` int(10) NOT NULL,
  PRIMARY KEY (`field1`,`field3`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

TABLE VI. KEY DEFINITION OF A PRIMARY KEY OVER MORE THAN ONE FIELD (0X00001000 - 0X0000102F)

0x00001000	01 02 00 00 0A 00 00 00 08 00 02 00 00 00 01 80	.....
0x00001010	02 00 00 1B 40 04 00 03 80 0A 00 00 1B 40 04 00	..@....@..
0x00001020	FF 50 52 49 4D 41 52 59 FF 00 00 00 00 00 00 00	.PRIMA RY.....

Table 6 shows the changes in the hexadecimal structure. The definition in the primary key covers more than one field. The value 01 was increased by one. This value stands for the numbers of keys. The two fields of the two parts of the primary key are described by the values 01 02 00 00 1B 40 04 00 and 03 80 0A 00 00 1B 40 04 00. The first byte stands for the number of the field. In this example it would be the fields "field1" (first field of this table) and "field3" (third field of this table).

C. *MySQL Storage Engine*

After the definition of the keys, InnoDB saves null-values as placeholders. After about 0x100 - 0x300 addresses one will find between further null-values a six byte long string. This string stands for the used storage engine. In the case of the InnoDB Storage Engine the string would have the value 49 6E 6E 6F 44 42 (=InnoDB).

D. *Field Definitions*

After the definition of the used MySQL Storage engine up to the address 0x00002100 follow only null-values as placeholders. Starting at this address one will find the definition of the single columns in the table. All references to the line numbers of the source code belong to the file /sql/unireg.cc [Appendix 4].

TABLE VII. HEXADECIMAL STRUCTURE OF THE FIELD HEADER (0X00002100 - 0X0000213F)

0x00002100	01 00 06 00 72 00 29 01 00 00 10 01 32 00 00 00	....r.)...2...
0x00002110	00 00 00 00 00 00 50 00 16 00 04 00 00 00 00 00	......P.....
0x00002120	72 00 07 06 02 14 29 20 20 20 20 20 20 20 20 20	r.....)
0x00002130	20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20	

TABLE VIII. STRUCTURE OF A FIELD HEADER

Offset	Length	Value	Meaning	Source Code
0x00	1	01	fixed value (screens)	726
0x01	1	00	unused	722
0x02	2	06 00	create_fields.elements	727
0x04	2	72 00	info_length	728
0x06	2	29 01	tolength	729
0x08	2	00 00	no_empty	730
0x0A	2	10 00	reclength	731
0x0C	2	32 00	n_length	732
0x0E	2	00 00	int_count	733
0x10	2	00 00	int_parts	734
0x12	2	00 00	int_length	735
0x14	2	50 00	time_stamp_post	736
0x16	2	16 00	columns needed (default: 80)	737
0x18	2	04 00	rows needed (default: 22)	738
0x1A	2	00 00	null_fields	739
0x1C	2	00 00	com_length	740
0x1E	4	00 00 72 00	placeholder for further information	741

TABLE IX. HEXADECIMAL STRUCTURE OF THE FIELDS (0X00002190 - 0X000021EF)

0x00002190	74 00 04 0A 0B 0B 00 02 00 00 1B 00 0F 00 00 03	t.....
0x000021A0	30 00 00 05 05 4A FF 00 06 00 00 00 40 00 00 00	0....J....@..
0x000021B0	0F 08 00 00 06 08 01 01 00 06 01 00 0B 80 00 00	.....
0x000021C0	00 01 30 00 00 07 0A 0A 0A 00 07 01 00 70 80 00	..0.....P..
0x000021D0	00 00 0E 30 00 00 08 08 0A 0A 00 0A 01 00 70 80	...0.....P.
0x000021E0	00 00 00 0E 30 00 00 09 07 0A 0a 00 0D 01 00 1A	....0.....

TABLE X. STRUCTURE OF A FIELD

Offset	Length	Value	Meaning	Source Code
0x00	1	04	field->row	793
0x01	1	0A	field->col	794
0x02	1	0B	field->sc_length	795
0x03	2	0B 00	field->length	796
0x05	3	20 00 00	recpos (field->offset+1 + data_offset)	797-799
0x08	2	1B 00	field->pack_flag	800
0x0A	2	0F 00	field->unireg_check	801
0x0C	1	00	field-->interval_id	802
0x0D	1	03	field->sql_type (siehe /include/mysql_com.h)	803
0x0E	1	30	if the field has got the type MYSQL_TYPE_GEOMETRY, the value will be field->geom_type, else charset field->charset if defined else 0 (numerical)	804-814
0x0F	2	00 00	field->comment.length	815

One can find more information about the MySQL Types at offset 0x0D in the file */include/mysql_com.h* [Appendix 6]. The value of this field will define the type of the column. The information about the type is, in some cases, not enough. For example, the types varchar and varbinary have a type value of 0x0F. In that case, one has to know more about the table. This means that one has to know the flags at offset 0x08. The flag "isBinary" defines in common with the type value of the field the correct type of the field.

TABLE XI. HEXADECIMAL STRUCTURE OF THE FIELD NAMES (0X000021F0 - 0X0000222F)

0x000021F0	80 00 00 00 03 30 00 00 FF 69 64 50 72 6F 6A 65	.....0...idP roje
0x00002200	63 74 FF 6E 61 6D 65 FF 64 65 6C 65 74 65 64 FF	ct.name.d eleted.
0x00002210	73 74 61 72 74 54 69 6D 65 FF 65 6E 64 54 69 6D	startTime .endTim
0x00002220	65 FF 65 66 66 6F 72 74 FF 00	e.effort..

III. DATA STORAGE

In contrast to the storage of the table information in the *.frm* files, the structure of data of every MySQL Storage Engine is different. In this section we will only refer to the InnoDB Storage Engine because space and time is limited.

Per default the InnoDB Storage Engine saves all data of every table in one single file. This can be an advantage for forensic analysis because deleted rows may exist for a long time. Analyzing the structure and architecture of the storage engine in one huge file is not necessarily conductive. One can force InnoDB to save every piece data of a table in one single file by editing the configuration file *my.cnf* and adding an entry "innodb-file-per-table=ON". InnoDB will not split up the existing data file, but it will now create a single data file for every table.

A. *Overview*

The InnoDB storage format has seven parts:

1. Fil Header
2. Page Header
3. Infimum- and Supremum Records
4. User Records
5. Free Space
6. Page Directory
7. Fil Trailer

B. *Fil Header*

The fil header creates together with the fil trailer a container around the pages in the data file. The fil header defines all important data in the file like offsets and checksums.

TABLE XII. HEXADECIMAL STRUCTURE OF A FIL HEADER

0x0000000	35 69 8D 4D 00 00 00 00 ff ff ff ff ff ff ff ff	5i.M.....
0x00000010	00 00 00 00 01 EA 03 31 00 08 00 00 00 00 00 00	.....1.....
0x00000020	00 00 00 00 00 BC 00 00 00 BC 00 00 00 00 00 00	.....

All references to the line numbers of the source code point to file /storage/innodb/include/fil0fil.h [Appendix 7].

TABLE XIII. MEANING OF THE FIL HEADERS

Offset	Length	Value	Meaning	Source Code
0x00	1	04	field->row	793
0x01	1	0A	field->col	794
0x02	1	0B	field->sc_length	795
0x03	2	0B 00	field->length	796
0x05	3	20 00 00	recpos (field->offset+1 + data offset)	797-799
0x08	2	1B 00	field->pack_flag	800
0x0A	2	0F 00	field->unireg_check	801
0x0C	1	00	field-->interval_id	802
0x0D	1	03	field->sql_type (siehe /include/mysql_com.h)	803
0x0E	1	30	if the field has got the type MYSQL_TYPE_GEOMETRY, the value will be field->geom_type, else charset field->charset if defined else 0 (numerical)	804-814
0x0F	2	00 00	field->comment.length	815

The pointers FIL_PAGE_PREV and FIL_PAGE_NEXT are very important for efficient serial queries. These pointers are the direct link between two adjacent pages (linked list). With the help of these two pointers a database management system can read data very quickly. If a serial query is executed, the database management system will start reading the smallest value of this query and read every following user record. If it reaches the end of a page it can quickly jump to the next value at the next page with the help of the pointer FIL_PAGE_NEXT.

C. *Page Header*

The page header creates together with the page directory the infimum- and supremum records, which is the outer casing of the user records. A page is in the hierarchy along with the last and smallest unit before a single data row. The page header contains all important information about a page. It defines important positions at the heap and b-tree. Therefore the page header can be used to pinpoint how many records on this page are deleted and how many bytes are free for further rows.

TABLE XIV. HEXADECIMAL STRUCTURE OF A PAGE HEADER

0x0000C020	00 00 00 00 00 BC 00 02 01 E3 80 09 00 00 00 00	.....
0x0000C030	01 C3 00 02 00 06 00 07 00 00 00 00 00 00 00 00	.....
0x0000C040	00 00 00 00 00 00 00 00 01 02 00 00 00 BC 00 00	.....
0x0000C050	00 02 00 F2 00 00 00 BC 00 00 00 02 00 32 01 00	.....2..

TABLE XV. STRUCTURE OF A PAGE HEADER

Offset	Length	Value	Meaning	Source Code
0x08	2	01 E3	PAGE_N_DIR_SLOTS count of slots in one page directory	38
0x0A	2	80 09	PAGE_HEAP_TOP pointer to the root of the heaps of this records	39
0x0C	2	00 00	PAGE_N_HEAP count of entries in the heap. Bit 15 is a flag for "new-style compact page format"	40-41
0x0E	2	00 00	PAGE_FREE pointer to the start of the "page free record list"	42
0x10	2	01 C3	PAGE_GARBAGE amount of bytes in deleted records	43
0x12	2	00 02	PAGE_LAST_INSERT pointer to the last inserted entry or NULL, if the last action was e.g. a DELETE operation.	44-46
0x14	2	00 06	PAGE_DIRECTION last insert direction (PAGE LEFT, ...)	47
0x16	2	00 07	PAGE_N_DIRECTION amount of last inserts in the same direction	48-49
0x18	8	00 00 00 00 00 00 00 00	PAGE_MAX_TRX_ID last TransaktionID, which made a modification on a user record.	51-57
0x20	2	00 00	PAGE_LEVEL level of the node in the index tree (leafs: 0)	61-62

0x22	8	00 00 00 00 00 00 01 02	PAGE_INDEX_ID IndexID to which this page belongs	63
0x2A	10	00 00 00 BC 00 00 00 02 00 F2	PAGE_BTR_SEG_LEAF File segment header of the leaf at the b-tree. Only defined at the root of the tree or at ibuff-trees.	64-66
0x34	10	00 00 00 BC 00 00 00 02 00 32	PAGE_BTR_SEG_TOP File segment header of non-leafs at a b-tree. Only defined at the root of the b-tree or at ibuf-trees.	74-78

D. Infimum- and Supremum Records

InnoDB creates the infimum and supremum records automatically and will never delete them. The infimum record stands for the lowest possible value of an existing entry of the following data segment. This barrier is used for the navigation and it prevents "get-prev" to disregard the beginning of the data segment. Analogical order, the supremum record stands for the maximum. InnoDB sometimes uses the infimum record to temporarily lock access to records of this segment. For this action, InnoDB will replace the infimum record with a "dummy entry" instead of the correct infimum record.

The usage of the infimum- and supremum entries works in the b-tree of the InnoDB storage engine because the keys are reached in a sorted order.

TABLE XVI. HEXADECIMAL STRUCTURE OF INFIMUM AND SUPREMUM ENTRIES

0x0000C050	00 02 00 F2 00 00 00 BC 00 00 00 02 00 32 01 00	.....2..
0x0000C060	02 00 1C 69 6E 66 69 6D 75 6D 00 08 00 0B 00 00	..infimum..
0x0000C070	73 75 70 72 65 6D 75 6D 11 00 00 00 10 00 34 80	supremum.

E. User Records

The user record contains one single row for each record. The following Table shows the structure of one user record. The offsets of the data in a row have information about the NULL values and the extra bytes (see Table 20) are defined at the beginning of the user record.

TABLE XVII. PART OF THE HEXADECIMAL STRUCTURE OF THE USER RECORD

0x0000C070	73 75 70 72 65 6D 75 6D 11 00 00 00 10 00 34 80	supremum. .....4.
0x0000C080	00 01 59 00 00 00 01 21 3C 80 00 00 00 2D 01 10	..Y..!<....-..
0x0000C090	41 6D 70 65 6C 20 41 70 70 6C 69 6B 61 74 69 6F	Ampel Applikatio
0x0000C0A0	6E 80 8f B2 C1 8F B2 DE 80 00 00 14 10 00 00 00	n.....
0x0000C0B0	18 00 33 80 00 01 5A 00 00 00 01 21 3C 80 00 00	..3...Z..!<...
0x0000C0C0	00 2D 01 1D 43 61 6D 70 69 6E 6f 20 48 6F 6D 65	.- ..Campino Home

To analyze the user records one has to know the exact structure of the table. The length of the data is defined in most cases at the offsets in the beginning of the records. Nevertheless, for some types the definition of length is missing because some types like "date" have got a fixed length.

```
(root@localhost) forensic> DESCRIBE Project;
+-----------+-------------+------+-----+---------+----------------+
| Field     | Type        | Null | Key | Default | Extra          |
+-----------+-------------+------+-----+---------+----------------+
| idProject | int(11)     | NO   | PRI | NULL    | auto_increment |
| name      | varchar(255)| NO   |     | NULL    |                |
| deleted   | tinyint(1)  | YES  |     | NULL    |                |
| startTime | date        | YES  |     | NULL    |                |
| endTime   | date        | YES  |     | NULL    |                |
| effort    | int(10) unsi| YES  |     | NULL    |                |
+-----------+-------------+------+-----+---------+----------------+
6 rows in set (0.00 sec)
```

In this example the field "name" has got a variable length. The length of the outer fields are fixed because of their types as defined in the *Project.frm*. The fields "idProject" and "effort" with the type "int" are four bytes, the fields "startTime" and "endType" are three bytes and the field "deleted" is one byte.

With this information you can read following data of this user record:

TABLE XVIII. STRUCTURE OF A SINGLE USER RECORD

Offset	Length	Value	Meaning
0x08	1	11	length of the first variable field (name), this field could have the length of two bytes.
0x09	1	00	information about the NULL-values. The size of this field is variable and is based on the amount of fields, which can be NULL.
0x10	6	00 00 10 00 34 80	extra bytes (more information about the extra bytes can be found at table 20)
0x20	3	00 01 59	primary key (If nonexistent this field will have the value of an ascending ID, because this is needed for the position at the b-tree)
0x23	6	00 00 00 01 21 3C	transactions ID
0x29	7	80 00 00 00 2D 01 10	rollback data pointer
0x30	11	41 6D 70 65 6C 20 41 70 70 6C 69 6B 61 74 69 6F 6E	first data field (name). The length of this field was defined at the Offset 0x08. content: Ampel Applikation
0x41	1	80	second data field (deleted). The length of this field is defined by its type "tinyint" (1 byte). The value of this field is calculated by the subtraction with 0x80, because this field is unsigned. Value: 0
0x42	3	8F B2 C1	third data field (start time). The length of this field is defined by its type "date" (3 bytes). You will find the exact interpretation of this value in the section "Interpretation of SQL data types". Value: 2009-07-02

0x45	3	8F B2 DE	fourth data field (endTime). see startTime field for explanation. Value: 2009-07-31
0x48	4	80 00 00 14	fifth data field (effort). The length of this field is defined by its type "int" (4 Bytes). Value: 20

The extra bytes 00 00 10 00 34 80 can be interpreted in the binary system as following:

TABLE XIX. BINARY STRUCTURE OF THE EXTRA BYTES

0x0000C050	00 02 00 F2 00 00 00 BC 00 00 00 02 00 32 01 00	.....2..
0x0000C060	02 00 1C 69 6E 66 69 6D 75 6D 00 08 00 0B 00 00	..infimum..
0x0000C070	73 75 70 72 65 6D 75 6D 11 00 00 00 10 00 34 80	supremum.

TABLE XX. STRUCTURE OF THE EXTRA BYTES

Offset	Length	Value	Meaning
0	1	0	*unused*
1	1	0	*unused*
2	1	0	DELETED_FLAG The value 1 shows, that this row was deleted.
3	1	0	MIN_RECORD_FLAG The value 1 shows, that this entry is the predefined minimum.
4	4	0 0 0 0	N_OWNED Amount of entries, which are owned by this record.
8	13	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0	HEAP_NO Sorting number of this entry in the heap of the index page
21	10	0 0 0 0 0 0 0 0 0 1	Amount of fields with variable length at this entry
31	1	0	1BYTE OFFS FLAG This bit is 1, if all fields have an offset with the length of one byte
32	16	0 1 0 1 0 0 1 1 1 0 0 0 0 0 0 0	pointer to the next entry at this page

One can gain a lot of information from the user records. If a user deletes a row, the database management system will not physically delete this record because of performance reasons. Therefore, the database management system will mark this record as deleted with the DELETED_FLAG (Table 20, offset 0x02). This row can be easily recovered. Tests show that the database management system will very quickly replace these deleted rows. After that, it is very difficult to reconstruct this data.

If the user record contains NULL-Values, InnoDB will save the information about the NULL-Values at the bytes after the information regarding the length (Table 18, offset 0x09). If an update query is executed and this query replaces the value of a field with NULL, InnoDB does not replace the value. Instead it will only change a bit in the NULL Information. In this case, the old value can be read from the record even at a later date.

F. Free Space

This part of the InnoDB storage file contains only placeholder (NULL-values) for further user records. If this area does not contain enough space for further rows, InnoDB will automatically create a new page.

G. Page Directory

The page directory contains an amount of pointers between the records. The order of the pointers is equal with the logical order of the records. That means that this order is not equal with the order at the heap. Because of this fact, one can easily implement a binary search through the records.

H. Fil Trailer

The fil trailer can be found at the last 8 bytes of the page. In our example, one can find the fil trailer in the address 0x0000BFF8. The definition of the value FIL_PAGE_END_LSN_OLD_CHKSUM in the source code can be found in the file storage/innobase/include/fil0fil.h [Appendix 7].

TABLE XXI. STRUCTURE OF THE FIL TRAILER

Offset	Length	Value	Meaning	Source Code
0x00	4	22 E1 22 20	checksum of this page (first four bytes of FIL_PAGE_END_LSN_OLD_CHKSUM)	101-104
0x04	4	01 EA 03 31	equal with FIL_PAGE_LSN from table 13 (last four bytes of FIL_PAGE_END_LSN_OLD_CHKSUM)	101-104

IV. INTERPRETATION OF SQL DATA TYPES

Every single data type has its own interpretation of the hexadecimal string in the user records. Because of its complexity, we would like to explain some data types quickly and how one can interpret the value of some important types of formats.

A. Numbers (Tinyint, Smallint, Int and Bigint)

If the field has the attribute "signed", one has to subtract 0x80 (tinyint), 0x80 00 (smallint), 0x80 00 00 00 (int) and 0x80 00 00 00 00 00 (bigint).

B. Float and Decimal

Float and decimal are two data types saved with the same method. The differences between the two data types are length (four and eight bytes) and its precision. MySQL will automatically save fields at the height of accuracy as a decimal instead of a float.

If one wants to calculate the value of the field, one has to calculate the binary value of the field. The last seven bits stand for the exponent. One can get the correct exponent if

one subtracts 1000000_2 from it. The 8th bit from the end of the bit string is the flag for the used sign. If this flag is 1, the real number will be negative. The other bits stand for the mantissa. In contrast to the IEEE 754 standard MySQL uses a seven bit long exponent instead of a eight bit long exponent. Therefore, one cannot use libraries to calculate the results. In case of the type float the mantissa is 3 bytes long, in case of decimal 7 bytes. The exponent with the sign flag has to be in both cases the length of one byte.

C. Date and time

1) Date

Before one can interpret the value of a date field one has to convert the value to the decimal system and subtract 0x800000 if it is greater than 0x800000. If the value is before the following date 01.01.0000 (before the birth of Christ), you can now convert the resulting number "value" to a date object:

```
year = floor(value / 512)
month = floor((value mod 512) / 32)+1
day = (value mod 32) +1;
```

2) Datetime

The first steps of interpreting a datetime field are equal with the interpretation of a date field. One has to convert the number to the decimal system and subtract 0x80 00 00 00 00 00 00 00 if possible. You can now interpret the result "value".

```
year = floor((value / 1.000.000) / 10.000)
month = floor(((value / 1.000.000) mod 10.000) / 100)
day = (value / 1.000.000) mod 100
hour = floor((value mod 1.000.000) / 10.000)
minute = floor((value mod 10.000) / 100)
second = value mod 100
```

3) Timestamp

It is very easy to interpret the timestamp: one has only to convert the hexadecimal string to the decimal system. The format of the timestamp is equal with the POSIX-Standard of UNIX time definition of the year 1969.

4) Time

Before one can interpret the value of the time field one has to convert the hexadecimal string to the decimal system. After that, one has to take the modulus of 1.000.000 and subtract 388.608 from it. Now you can interpret the value of the number:

```
hour = floor(value / 10.000)
minute = floor((value % 10.000) / 100)
second = value % 100
```

5) Year

MySQL starts to count the years starting at the year 1900. This means that you have to add 1900 to the decimal value of the field. It is not possible to save years before the year 1900.

V. Conclusion

This paper shows how the MySQL tables in the .frm files are built and how important information is saved. With that information it is possible to read every important information in a table from this file. This paper describes the entire structure of the file format of the InnoDB Storage Engine. With this information one can read rows from the storage files and can recover some deleted or overwritten rows and detect inconsistencies that may occur from an attacker. For instance, manually altering the data files and circumventing access restrictions imposed by access control mechanisms of the database [6].

To interpret the data one has to know a lot about the MySQL Internals. Therefore, we show how one can convert most of the important SQL types to a readable string. For this interpretation, one needs a lot of information from the relevant tables. The detailed analysis of the .frm files is therefore, very important.

With the data found from the files one can partly recover deleted and updated user records from the file system. The aim of this paper is to identify and name the bytes and interpret them. With that knowledge, it is possible to detect inconsistencies in the database. To prove it in detail you need to know more about the history of the tables. A connection between the binary log files and the user records make it is possible to prove the consistency of the table in detail because of the transaction-ID in the User Records.

Appendix 12 contains an analysis program we created, which can read the data from a MySQL Database with InnoDB Storage Engine. The program reads the bytes and identifies them. The data will then be converted to known structures and objects. Initially, the program will read and reconstruct the table structure from the .frm file. Information about the columns will be saved in an instance of the class /structure/Field.java. This information is used to analyze the structure of the data in the InnoDB storage data file. The program first looks for the fil- and page header. After that, it looks for the infimum and supremum records and from that position it reads the existing user records. For that step, it is important that the program knows the exact table structure analyzed in the first step because it has to know the correct offsets. The program will not only show the data, it will also display the found bytes for further analysis.

This program is a basic software tool and can be extended in many directions. Because of converting the data, known structures and objects should be easier to analyze in more detail. One can build a connection with the binary log, which verifies the consistency of the data. Further uses could involve an extended analysis, which can show the differences between two states of a database. With this extension, one could for example, analyze changes in the storage files after executed operations.

References

[1] Ryan Bannon, Alvin Chin, Faryaaz Kassam and Andrew Roszko InnoDB Concrete Architecture, University of Waterloo 2002

[2] R. Bayer, The universal B-tree for multidimensional indexing: General concepts, Worldwide Computing and Its Applications, vol. 1274, 1997B. Carrier, File System Forensic Analysis, Amsterdam: Addison-Wesley Longman, 2005.

[3] D. Comer, Ubiquitous B-Tree, ACM Computing Surveys (CSUR) archive, vol. 11 , no. 2, June 1979.

[4] K Eckstein, Forensics for Advanced UNIX File Systems, in Proceedings of the 2004 IEEE Information Assurance Workshop, 2004.

[5] K. Eckstein and M. Jahnke, Data hiding in journaling file systems, in Proceedings of the 2005 Digital Forensic Research Workshop (DFRWS), 2005.

[6] Wolfgang Essmayr, Stefan Probst, and Edgar R.Weippl. Role-based access controls: Status, dissemination, and prospects for generic security mechanisms. International Journal of Electronic Commerce Research, 4(1):127-156, 2004.

[7] Eva Gahleitner, Wernher Behrendt, Jürgen Palkoska, and Edgar R. Weippl. On cooperatively creating dynamic ontologies. In Proceedings of the 16th ACM Conference on Hypertext and Hypermedia, Salzburg, Austria, September 2005. ACM. 2004.

[8] K. J. Jones, R. Bejtlich, and C. W. Rose, Real Digital Forensics: Computer Security and Incident Response, Upper Saddle River: Addison-Wesley Professional, 2005.

[9] Oscar Mangisengi, Wolfgang Essmayr, Johannes Huber, and Edgar Weippl. Xmlbased olap query processing in a federated data warehouses. In Proceedings of the 5th International Conference On Enterprise Information Systems (ICEIS), France, April 2003. reprinted as 'Towards XML OLAP Query in Federated Data Warehouses' in a selection of best papers in the book Enterprise Information Systems V, Edited by Olivier Camp, Joaquim B.L. Filipe, Slimane Hammoudi, and Mario Piattini, ISBN 1-4020-1726-X (HB) ISBN 1-4020-2673-0 (e-book) Kluwer, Netherlands,

[10] Martin S. Olivier, On metadata context in Database Forensics, Digital Investigation Volume 5, Issues 3-4, March 2009, Pages 115-123

[11] Kyriacos Pavlou, Richard T. Snodgrass, Forensic Analysis of Database Tampering, International Conference on Management of Data, Proceedings of the 2006 ACM SIGMOD international conference on Management of data, SESSION: Authentication, Pages: 109 - 120

[12] Rudolf Ramler, Klaus Wolfmaier, and Edgar Weippl. From maintenance to evolutionary development of web applications: A pragmatic approach. In Proceedings of ICWE 2004, LNCS Springer, Munich, Germany, July 2004. Springer.

[13] Pachev Sasha. "Understanding MySQL Internals, O'Reilly April 2007

[14] Patrick Stahlberg, Gerome Miklau, and Brian Neil Levine, Threats to Privacy in the Forensic Analysis of Database Systems, University of Massachusetts Amherst 2007

[15] C. Swenson, R. Phillips, and S. Shenoi, File System Journal Forensics, Advances in Digital Forensics III, vol. 242, pp. 231-244, 2007.

[16] Heikki Tuuri, Calvin Sun "InnoDB Internals: InnoDB File Formats and Source Code Structure" MySQL Conference, April 2009

[17] Edgar R. Weippl and Wolfgang Essmayr. Personal trusted devices for web services: Revisiting multilevel security. Mobile Networks and Applications, Kluwer, 8(2):151-157, April 2003. http://www.kluweronline.com/issn/1383-469X.

Internet Sources

[18] Structure of an InnoDB storage engine data file http://forge.mysql.com/wiki/MySQL_Internals_InnoDB (07.07.2009)

[19] InnoDB File Structure and Primary key Tree Structure http://www.bigdbahead.com/?p=142 (07.07.2009)

[20] InnoDB data structure - distribution in multiple files http://www.siusic.com/wphchen/how-to-prevent-innodb-data-file-keep-growing-156.html (07.07.2009)

[21] MySQL Internals File Formats http://forge.mysql.com/wiki/MySQL_Internals_File_Formats (07.07.2009)

[22] MySQL Internals Manual http://faemalia.net/mysqlUtils/mysql-internals.pdf (10.07.2009)

[23] http://www.guidancesoftware.com/. [Accessed: May 29, 2009].

[24] http://www.sleuthkit.org/. [Accessed: May 29, 2009].

VI. APPENDIX

The files can be access at http://www.ifs.tuwien.ac.at/~weippl/download/src.rar.

1. src/sql/table.cc - MySQL 5.1.32 Source Code
   Initialization of the Headers of the .frm files at the function create_frm()
2. src/include/mysql_version.h - MySQL 5.1.32 Source Code
   General definitions like version numbers, version of the protocol etc. (Template)
3. src/sql/handler.h - MySQL 5.1.32 Source Code
   Definition of the constants of the MySQL Storage Engine
4. src/sql/unireg.cc - MySQL 5.1.32 Source Code
   Creates field information of .frm files
5. files/Project.frm - MySQL .frm file of the table Project
   Example used to explain the analysis of the table structure of MySQL
6. src/include/mysql_com.h - MySQL 5.1.32 Source Code
   Definition of the constants of the MySQL data types of fields
7. src/storage/innobase/include/fil0fil.h - MySQL 5.1.32 Source Code
   Definition of the offsets in the InnoDB data file
8. src/sql/structs.h - MySQL 5.1.32 Source Code
   Definition of the structure KEY
9. src/include/my_base.h - MySQL 5.1.32 Source Code
   Key algorithms
10. src/storage/innobase/include/page0page.h - MySQL 5.1.32 Source Code Offsets of the page headers
11. src/storage/innobase/include/page0cur.c - MySQL 5.1.32 Source Code
    Delete function for user records (function page_cur_delete_rec() - line 1340-1439)
12. analyze r - InnoDB data analyzation program
    Self-developed program in Java 1.6.0
13. /src/include/my_global.h - MySQL 5.1.32 Source Code
    IO_SIZE definition