Timing Attacks for Recovering Private Entries From Database Engines

Ariel Futoransky, Damián Saura, and Ariel Waissbein[1]

Core Security Technologies, Humboldt 1967, Cda. de Buenos Aires 1414, Argentina.

Abstract. Dynamic content for Web applications is typically managed through database engines, including registration information, credit cards medical records and other private information. The web applications typically interface with web users and allow them to make only certain queries from the database while they safeguard the privacy where expected, for example, they may allow to add data in a column of the database but not to view the complete contents of this column. We will describe a new technique that allows executing a timing attack which recovers entries from a private column in a database and only requires the ability to insert data in this private column. During the presentation, we will show the experiments that led us to developing exploit code for the MySQL engine that demonstrates this technique, give details for the audience to understand the underlying algorithm, analyze the results and discuss future work. We will also discuss how to protect from or detect this exploit.

Vulnerability Description

Database management systems are prone to a design vulnerability in their indexing and insertion/update algorithms.

An attacker which is allowed to insert new entries in a database table and time their response can exploit this vulnerability and recover all the entries from the table under attack.

The problem derives from an information leak caused by the data structure used for indexing table columns and the underlying data manipulation and storing functions.

This vulnerability is present in the MySQL v5 when used with the InnoDB storage engine. Earlier versions have not been checked. This vulnerability could also affect every database configuration where a table column is indexed by a data structure and algorithms set which leak information, including but not limited to, those that implement the B-tree data structure. This has not been checked.

Vulnerable systems

MySQL is vulnerable when used with InnoDB storage system. MySQL is the most popular free and open source database engine, with a great success in

web applications, it is the database engine in the LAMP/WAMP platforms (Linux/Windows-Apache-MySQL-PHP), and used in the popular *Youtube*, *flickr*, *Wikipedia* and *Bugzilla*.

If a MySQL engine is configured with the InnoDB storage engine (see [MyS07]), a nonempty table is configured to store one of its fields using clustered indexes (i.e., as primary keys) and the attacker is able to insert records, then the attack is successful and the attacker is able to retrieve the complete table field.

We have not tested our attack technique against other database engines or configurations. However, we conjecture that it is possible to successfuly instantiate our attack against most database management systems that store data indexed by primary keys and using the B-trees data structure (and possibly other data engine configurations which leak information).

Nonvulnerable systems

The attack technique we describe in this article was only instantiated to attack the MySQL engine v5.0.26-5.0.37 when used with the InnoDB storage system. Other MySQL versions and configurations were not tested.

Stating what scenarios are nonvulnerable to this attack technique is a difficult task. Clearly, this attack cannot be applied to table columns that are indexed by data structures (and insertion algorithms) which do not leak information. Each systems and configuration should be examined carefuly before claiming it is nonvulnerable.

Presumably, databases that can only be accessed remotely and through noisy channels are not vulnerable to this attack. In the next sections we will give advices which allow to determine whether a channel is noisy.

Solution/mitigation

In order to secure a MySQL-InnoDB engine the easiest countermeasure to implement is not indexing the table columns that you want to protect; so first, one should make sure if this is an option. However this might be impractical, and certain applications might require other countermeasures. One could use transaction throttling, e.g., limiting the number of data manipulation operations that a given user is allowed to make (or doing this for each IP address). Also, one could monitor every command that is sent to the MySQL server (at network level), examine every data manipulation operation and block a user when more than $n \gg 1$ consecutive inserts are made; or simply insert small time delays after each data manipulation operation (see, e.g., the Appendix section).

A more disruptive countermeasure is redesigning or reconfiguring index storage functionalities. To stop the attack from working, no information should be leaked after table inserts. This means that every data manipulation operations should require the same amount of time (or independently distributed values).

Notice that these countermeasures can be applied to the MySQL-InnoDB system but also to other database management system implementations.

Technical description

Preliminaries: MySQL & InnoDB

Generally speaking a **database** is a structured collection of data. In a **relational database** ([Cod69,Cod70]), data is structured in different sets of n-relations and data extraction is done through the evaluation of predicates. Relations are collected in **tables**, each member of a table (i.e., an n-relation) is called a **record**, and each of the n elements that define the relation is called a **field record**. Often, tables are pictured as bidimensional arrays, of an arbitrary number of rows and n columns.

A **database management system (DbMS)** is a collection of programs that enable you to store, modify and extract information from databases. Its main capabilities can be summarized in persistent storage, a programming interface and transaction management. A DbMS handles two types of users, authorized users and administrators, where authorized users connect to make queries and/or modify the content of tables, while administrators can also create and configure tables and the DbMS settings. We are specially concerned with the **MySQL** database management system, which uses a **storage engine** for storing (it can be configured to use the InnoDB engine or others) and a variant of the SQL language for its programming interface. Currently, it is being distributed in its version 5.0.

Structured Query Language (SQL) ([CB74]), is a computer language designed for interacting with the DbMS and it is inspired in relational database management systems. Roughly speaking, the SQL language can be divided in the following classes of commands: **Queries** (e.g., operations that extract data from the database); **Data manipulation** (e.g., insert, update, merge); **Transaction controls** (e.g., begin work, commit, rollback); **Data definition operations** (e.g., create, drop, alter); etcetera.

In DbMSs data is stored in the hard disk. Queries and data manipulation commands are processed by a **query compiler**, which forwards optimized instructions to an **execution environment**. When required to access the tables in the database, the execution environment will ask for a given index (or file, or record) to the **Index/file/record manager**, which in turn will command the **buffer manager** for **disk pages** (or **blocks**), which in turn a retrieved (as a whole) from **storage** by the **storage manager**. (These last three components are loosely refered to as the storage engine.) Disk pages are the smallest amount of data that can be retrieved from (permanent) storage. (See [GMUW00] for more on this.)

The computer reads/writes in one of these disk pages at a time, and typically goes from one to the other during its many processes —taking more time to go from one page disk to a page disk that is further in the hard disk than one that is nearer. These disk pages, and the information contained therein, must be efficiently located so that the insert, delete and select operations can be made efficiently.

Historically and up to this date several data structures have been proposed for storing indexes, with B-trees ([BU77], [BM71]) being the most popular (for strings and small pieces of data; other type of data, such as spatial data, are better approached by other data structures). MySQL stores permanent data in the hard disk using a storage engine (these include the index, buffer and storage managers mentioned above), and which can be selected from list including InnoDB, MyISAM, MEMORY/HEAP. The first three allow only the B-tree data structure for primary indexes, while the latter also allows the use of the hash tables.

We are concerned with InnoDB, which stores indexes using a variant of the B-tree data structure. Each node is stored in a page disk. In **B-trees**, data is organized in blocks and these blocks in a tree. Each node contains n search-key values and $n+1$ pointers, for a fixed integer n that is called the block length and whose value is determined by the DbMS ([GMUW00] and [MyS07, Section 14.2.13]). At the root, except in a border case, there are at least two pointers: one pointing to each block below.

Each leaf consists of $n+1$ pointers, with the last of these pointers pointing to the block of leafs next to the right, and the other pointers being either empty or pointing to data records. Pointers are used from left to right, and at least $\lfloor (n+1)/2 \rfloor$ should be used.

In internal nodes, the $n+1$ pointers point to blocks in the next level and at least $\lceil (n+1)/2 \rceil$ of these should be used. Pointers are ordered increasingly and they represent consecutive nodes in the level immediately after. Explicitly, for $j+1$ pointers used there exist j keys, $K_1, \ldots, K_j$ such that all the keys in the first node are smaller than K_1, all the keys in the second node are between K_1 and K_2, and so forth.

According to design principles, after data manipulations (e.g., insertions, updates or deletions), we might require to add or delete nodes. When a node (i.e., a disk page) is added to the tree —say, because a key was inserted and it belonged to the range of an already full node— we call this action a **a node split** or a **split**. There are several design and implementation decisions which are particular to InnoDB. For example ([MyS07, Section 14.2.13.1]): "If index records are inserted in a sequential order (ascending or descending), the resulting index pages are about 15/16 full. If records are inserted in a random order, the pages are from 1/2 to 15/16 full. If the fill factor of an index page drops below 1/2, InnoDB tries to contract the index tree to free the page."

In analyzing the MySQL-InnoDB engines, we discovered that the following properties, essential to our attack, are verified:

- Starting from a node with ordered key values $[A, r, s]$ where A represents a sequence of keys (respectively $[s, r, A]$) and adding keys $r+1, r+2, \ldots$ (respectively $r-1, r-2, \ldots$) until there is a split after adding the key $r+m$ (respectively $r-m$), then as a result two nodes are produced $[A, r, r+1, \ldots, r+m-1]$ and $[r+m, s]$ (respectively $[s, r+m]$ and $[r-(m-1), \ldots, r-1, r, A]$).

– Starting from a node with ordered keys $[A, r, S]$ (respectively $[S, r, A]$) where A and S represent sequences of keys, the size of S is bigger than 1, and adding values $r+1, r+2, \ldots$ (respectively $r-1, r-2, \ldots$) until there is a split after adding the key $r+m$ (respectively $r-m$), then as a result two nodes are produced $[A, r, r+1, \ldots, r+m]$ and $[S]$ (respectively $[S]$ and $[r-m, \ldots, r-1, r, A]$).

These properties were discovered experimentally through instrumentation of the MySQL/InnoDB source code. We have described the instrumentation and the effect of this instrumentation in the Appendix section.

Attack Design

Let us fix a MySQL configuration with the InnoDB storage engine, a nonempty table and a table column. Let us assume that this column is indexed by primary keys (which use the B-tree index structure) and a user (the attacker) has the ability to make INSERT operations over this table column, and measure the time they take. The attack can be executed by a probabilistic[1]algorithm, which receives as input a value x_0 in the range of the table column under attack (e.g., $x_0 = 0$ if the table column stores integers, or the string $x_0 = a$ if it stores strings), and returns the smallest value y in this table column which is larger than x_0.

For the sake of simplicity, let us assume that the table column under attack stores integer numbers.

If we want to retrieve all the keys in the table column, we simply start by executing the algorithm a first time with $x_0 = 0$, computing the smallest key y in the column, next set $x_0 = y + 1$ and execute the algorithm once again. Continuing in this way, we are able to retrieve all the keys form the table column under attack.

The algorithm uses as parameters, a step base b and an exponent r. It starts with an input x_0 and returns a key y. Let us write the b-base representation of y as $y = \sum k_i b^r$, with $0 \leq b_i < b$. The algorithm first computes k_0 (i.e., an interval $[k_0 \cdot b^r, (k_0 + 1) \cdot b^r]$, with $0 \leq k_1 < b$, such that y belongs to this interval). Next, for $\ell = 1, 2 \ldots$, it will recursively compute the digits $k_1, k_2, \ldots$ of y (e.g., it computes nested intervals containing y and of size $b^{r-\ell}$). Each step requires making a number of inserts proportional to n, the number of entries that fit in a page disk[2], and, $|y|$, the size of the key y.

When the size of the interval $b^{r-\ell}$ is smaller than the page size n, the algorithm continues searching in this interval examining all its members. Notice that, since indexes are stored as primary keys, when the attack attempts to insert a key with the same value as an existent key, it will receive an error. This error indicates that the attack was successful and a key was found.

[1] The algorithm might fail in some cases, depending on the input parameters and the keys in the table. Describing these cases is out of the scope of this work.

[2] In InnoDB page disks are of size 16kB, and depending on the length of the data stored in these keys one can determine this value n.

We designed and developed the attack in two steps which can be analyzed separately. First, we assumed that a split detection algorithm was available (i.e., an algorithm that can decide whether an insert operation produced a split or not), and designed an attack that was able to recover any number of fields from the table under attack. Next, we designed and implemented a split detection algorithm which can be composed with the previous algorithm.

Let us assume, for now, that for each insert made, the user also knows whether this operation produced a node split. This assumption will be waived, and replaced by a split detection algorithm in the next section. Then, the following algorithm recovers y.

Algorithm 1:
Input: x (starting point), b (base of the steps) and B (a power of b which is the biggest step size).
Output: A key y.

```
# simple tuning
# The page containing key y
For i=1 to 3 do:
  {
  Insert a,a+1,a+2... until a split occurs.
  Set a as the next value after the split.
  }

# First step computes the interval I=[k B, (k+1)B] that contains y
Repeat
  {
  Insert a,a+1,a+2... until a split occurs
  Set k the number of inserts made
  a = a + B
  } Until k != PageSize

# Tuning so that there is one page whose leftmost value is the sought key
Insert a-2B-1,a-2B-2,a-2B-3,... until a split occurs


# Second step, narrowing.
Set a = a-3B.
B = B/b.
Repeat
  {
  Repeat
    {
    Insert a,a+1,a+2... until a split occurs
```

```
  Set k the number of inserts made
  a = a + B
  } Until k != PageSize
a = a - B + k
B = B / b
If B < n
  {
  Insert entries a,a+1,a+2... until key is found
  }
}
```

During the execution of the algorithm described in the previous page, we only fall in the four cases of page splits we describe above (except perhaps in the first step). We can argument that in this way, the algorithm correctly computes the key y.

More generally, assume that during the run of the above algorithm we add $a, a+1, \ldots$ and miss a split or more, but eventually correctly detect a split, then the algorithm also correctly retrieves the key.

Attack Implementation

In order to finish the attack, we experimented with MySQL and InnoDB and discovered that splits were statistically noticeable only by measuring the time taken by the insert, but not with %100 accuracy.

Our split detection algorithm starts by estimating a "threshold" value t_* such that most of the inserts that produce splits take more time than t_* and most of the inserts that do not produce splits take smaller time than t_*. In fact, our algorithm simply records the time taken by the last few inserts made, and computes the 90th percentile, and sets this value as t_*. (Experimentally, we discovered that the 90th percentile has this property when storing the time taken by a few thousand inserts.)

Consider consecutive ascending or descending inserts which took time $t_1, t_2 \ldots$ respectively. Let i be such that t_i, t_{i+n}, t_{i+2n} are all bigger than t_*. Then, our algorithm detects a split at t_{i+2n}. (From the experimental data, one can easily argue that the probability of "thee inserts taking time greater than t_* and being separated by $n-1$ inserts" correspond to a node splitting event is close to 1.)

We tested our split detection algorithm and discovered its accuracy was sufficient for executing the attack successfully. On the other hand, sometimes the number of inserts made in order to detect a split were much larger than expected (e.g., $i + 2n \gg 3n$).

Finally, we composed the attack algorithm with this split detection procedure (using the remark made after **Algorithm 1**) and executed several successful attacks. Explicitly, we fixed a computer and executed the attack against different scenarios. We executed our attack against a table with a single column of 64 bit

integers with 1, 101 and 1001. The results of these attacks are summarized in the following table.

# of keys	Result	# of inserts	time elapsed
1	Success 3/3	14291	09:48
1	Success	14864	11:13
1	Success	13145	10:52
101	Success	13145	10:54
101	Success	13145	10:53
101	Success	13145	10:11
1001	Success	12858	09:56
1001	Failed	10590	08:34
1001	Failed	20094	15:47
1001	Success	12592	08:33
1001	Success	15723	11:09

References

[BM71] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. In *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access, November 15-16, 1970, Rice University, Houston, Texas, USA (Second Edition with an Appendix)*. ACM, 1971.

[BU77] Rudolf Bayer and Karl Unterauer. Prefix b-trees. *ACM Trans. Database Syst.*, 2(1):11–26, 1977.

[CB74] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured english query language. In Randall Rustin, editor, *Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, May 1-3, 1974, Vol. 1*, pages 249–264. ACM, 1974.

[Cod69] Edgar F. Codd. Derivability, redundancy and consistency of relations stored in large data banks. IBM Research Report, San Jose, California RJ599, 1969.

[Cod70] Edgar F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[GMUW00] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. Perntice Hall, 2000.

[MyS07] MySQL. *MySQL 5.0 Reference Manual*, 2007.

Appendix: MySQL Instrumentation

In order to design our attack, we instrumented the InnoDB code so that we were able to detect with %100 accuracy when a node was split, and to know what keys were included in each node. Also, we configured MySQL to store data (and indexes) in a ramdrive, instead of the hard disk. This allowed us to remove noises coming from undetermined delays.

Below, we copied the most important changes that appear by making a *diff* between the instrumented and the original code. In fact, we only modified the function `btr_page_split_and_insert` (in `\innobase\btr\btr0btr.c`). The name is self explanatory: this function is called every time an insert is made to a full page.

First, we added a significant delay after the split is made.

```
...
  heap = mem_heap_create(1024);
  n_uniq = dict_index_get_n_unique_in_tree(cursor->index);
func_start:
  if(strcmp(cursor->index->name, "Index_0") == 0) // if this is
                                          //  the index we are working with
   {
    Sleep(100);
   }
  mem_heap_empty(heap);
  offsets = NULL;
  tree = btr_cur_get_tree(cursor);
...
```

Second, at the beginning of the function we added some code that dumps the b-tree into a file (`c:\pages.txt`).

```
  ...
FILE *f;
ulong keyAux;
int keyIsNumber = TRUE; // this is a constant that needs to be modified
                        // indicating if our column data has numbers or strings
byte *bufAux;
tree = btr_cur_get_tree(cursor);
if(strcmp(cursor->index->name, "Index_0") == 0) // if this is
                // the index we are working with
  {
   f = fopen("c:\\pages.txt", "at");
   if(f != NULL)
    {
     fprintf(f, "=== BEFORE SPLIT TREE");
      // dump the tree
     btr_print_tree(f, tree, 1000);
```

```
     fclose(f);
    }
  }
...
```

Finally, we added at the end of the function (just before it returns) code that dumps the b-tree into the file if a split is made (in this way, we have the state of the b-tree before and after a split).

```
...
if(strcmp(cursor->index->name, "Index_0") == 0) // if this
                // is the index we are working with
  {
   f = fopen("c:\\pages.txt", "at");
   if(f != NULL)
    {
     // show the value of the insert that generated the split
     if(keyIsNumber)
      {
        keyAux = 0;
        for(i = 0; i < tuple->fields->len; i++)
         {
           keyAux += (((ulong)((byte*)tuple->fields->data)[i])
                       << ((tuple->fields->len-i-1)*tuple->fields->len));
         }
        fprintf(f, "=== SPLITED INSERTING: %ld\n", keyAux);
      }
     else
      {
        bufAux = malloc(tuple->fields->len + 1);
        memcpy(bufAux, tuple->fields->data, tuple->fields->len);
        bufAux[tuple->fields->len] = 0;
        fprintf(f, "=== SPLITED INSERTING: %s\n", bufAux);
        free(bufAux);
      }
     btr_print_tree(f, tree, 1000);
     fclose(f);
    }
  }
  ...
```