JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Hardening MySQL on POSIX compatible systems

Seminararbeit

Netzwerke und Sicherheit

LVA-Nummer: 351.094, WS2007

Angefertigt am *Institut für Anwendungsorientierte Wissensverarbeitung*

Betreuung:

a. Univ.-Prof. Dr. Josef Küng

Eingereicht von:

Sonnleitner Erik

Linz, December 2007

Contents

Abstract

This paper describes how to secure a POSIX compliant environment through setting up filesystem access restrictions and access control lists, designing `chroot` sandboxes for peuso-virtualization, make use of strong cryptography on network- and filesystem level, as well as protecting MySQL from exploitable buffer overflows. After ensuring operating system secureness, I'll discuss some security related configuration attributes of the database server, before going into detail of the internal privilege management of the DBMS.

1 Introduction

The enormous global increase of information which is to be stored, forces certain approaches of archiving and restoring data, while keeping track of numerous valuable and essential preconditions, e. g. data integrity.

Relational databases are still the common way of accomplishing the storage of masses of information, although its conceptional basics reach back to 1970, where E. F. Codd firstly introduced this method of data handling [Cod70].

As global networking dramatically increased the past decades, the TCP/IP protocol stack has become very popular and nowadays builds the fundamental backbone of the Internet. As conclusion to this tendency, also the way of controlling and operating relational database systems mostly relies on the mentioned protocol suites, with all advantages and disadvantages, inherently given by using them.

Accessability and reliability of information services is often constrained by providing them over the Internet, which should be seen as naturally untrusted and insecure network, since not only permitted persons are able to try to establish connections.

With the aspect of Unix-like system environments in mind, I'll figure out how to secure and harden database systems primarily on Linux, taking MySQL 5 as example, since this software is commonly used and widespread, especially over the Internet, for it is Open Source Software. Except for the description of filesystem encryption, all examples should work also on other POSIX compliant operating systems than Linux.

The language of given sourcecodes should be clear from the context they are mentioned. However, shell scripts are written using the Bourne Again Shell (`/bin/bash`), and most sources are plain C. When shell command examples are given, every line is prefixed with either `#` or `$`. While the hash indicates that the following statement has to be called as `root` user, the dollarsign commands doesn't need administrative permissions.

2 Hardening the Unix environment

Common Unix-like systems offer a wide range of security related tools and methods for obtaining access restrictions. The configuration of certain software packages like databases is assuredly to be done carefully and with respect to secureness. Nevertheless, a system-wide security model for protecting information and information services should begin (at least) at operating system level.

A perfectly configured Oracle Database Server, including DMBS account and role management etc., won't be useful if everybody may be able to simply copy the raw data from the filesystem for obtaining the desired information quickly and easily.

For more in-depth information about Unix and the Unix system environment, I'd refer to [SWF05], [Amb07] and [Bau02].

2.1 Filesystem access restrictions and Access Control Lists

Most suitable filesytems available on POSIX environments provide mechanisms of restricing methods of access in an abrasive way, using (at least) three types of access mode codes, and three ways of describing for whom those modes apply.

The basic filesystem rights are

- read (→ `'r'`)
- write (→ `'w'`), and
- execute (→ `'x'`)

which can be individually referred to

- the user which is the owner of the filesystem object, e.g. a file or a directory (→ `'u'`),

- the group of persons which belong to the (main) group of the owner (→ `’g’`), and
- all others (→ `’o’`).

Taking the major configuration file of MySQL, which is normally found at `/etc/mysql/my.cnf`, the filesystem rights are given as following:

```
$ ls -lh /etc/mysql/my.cnf
-rw-r--r-- 1 root root 3.7K 2007-07-18 00:14 /etc/mysql/my.cnf
```

The access rights are shown in the string `-rw-r--r--`. Disregarding the first `-` character, Unix returns basically a nine-character string, which is to be read in triples, as `rw-|r--|r--`. The first triple describes the permissions of the owner, the second the permissions of the owner’s group and the third triple refers to all other users. Therefore, only the owner of the file (the `root` user, the administrator) is allowed to modify the file because of the write permission – users in the same group as well as all other system users may only read the object.

The upcoming columns, both entitled as `root` describe the owner of the object, and group membership belonging of the object. As we see, the `my.cnf` file is owned by the user `root` and belongs to the system group `root`.

The configuration files should always belong to the `root` user, and only permit `root` to write on these objects, since nobody else should be able to modify its contents in any way. The right permission settings may be assured by

```
# chown -R root:root /etc/mysql/
# chmod 0644 /etc/mysql/my.cnf
```

In dependency on what other configuration files MySQL actually is referring to, the `chmod` command may also be applied to other items inside the `/etc/mysql/` directory.

Storage data MySQL stores the actual data (tables, etc.) in `/var/lib/mysql` or `$MYSQL/data` by default. In contrast to the configuration files, the data storage files should not be owned by the administrator, but by a completely unprivileged user, normally called `mysql`, which isn’t allowed to to anything else inside the Unix system as what is absolutely necessary. Besides the administrator of course, nobody should be able to read and/or modify these objects, therefore we completely revoke any rights of the `others` user section and just let `mysql` read and write.

Moreover, the `mysql` user should by no means be able to invoke a command shell. This assures that crackers arn't be able to login at the server system, even if this user has been hacked. Revoking command shells is done within `/etc/passwd`, by changing the last column of the `mysql` user from `/bin/bash` to `/bin/false`. The program given here will be invoked when a user has been successfully authenticated by the system.

Logfiles MySQL commonly logs every event, relevant to the database. Absolutely no other users than `root` and `mysql` should be able to read or write the logs, preventing the leaking of information out of the logfiles. For example, certain queries like `GRANT` may offer sensitive information like user passwords, which are stored plaintext inside the protocol files. The logs are normally owned by the `mysql` user, since MySQL needs to write the events here (in contraty to the configuration files, only the administrator should be able to modify, not the MySQL system).

Access control lists ACLs, or *Access control lists* offer a very granular method of defining and granting permissions. As opposed to the standard Unix filesystem rights, POSIX ACLs are not built-in in the filesystem device driver (as done in `ext2/3`, `reiserfs`, `xfs`, etc.).

The usage of ACLs offers mechanisms for setting up per-user-permissions of single filesystem objects and therefore provide fine-grained definitions of access restrictions, if needed. The corresponding POSIX commands are `getfacl` for viewing ACLs, and `setfacl` for setting up an ACL. These features may be useful to add certain permissions to other users (e. g. automatic logfile analyzers). The following example quickly shows the usage of `setfacl`, allowing the user `syslog` to write on the MySQL log files:

```
# setfacl -m user:syslog:-w- /var/log/mysql/*
```

2.2 Designing a `chroot`-jail

Even when accurately managing user- and group-memberships as well as read and write permissions to the relevant MySQL filesystem objects, we should assure, that, in case of a successful attack, the system environment does not get compromised in any way. Numerous attacks have been reported on this topic. When talking about attacks, we now commonly mean attacks from within the database system, when users or programs try to gain sensitive system parameters like the `/etc/shadow` file or logfiles via outfoxing the DMBS.

That's why we need to create a sandbox-like environment where MySQL runs within and is restricted to. In terms of POSIX systems, this is called a *change root*-environment, or *chroot-jail* named by the corresponding command `chroot`. In the early Eighties when nowadays keywords like virtualization havn't been born, Bill Joy introduced the concept of the `chroot` command which can be seen as forerunner of an virtual system environment.

`chroot` basically repositions the global root directory (`/`) via remapping it into a specific directory of any directory within the filesystem tree. Any commands, applications, users etc. which act within the `chroot`-environment actually don't know that they are working in a sandbox and should have no chance for accessing any part of the filesystem outside the jailed area.

Designing the sandbox Since the jailed environment won't be able to access the rest of the filesystem, *all* relevant system objects like binaries, libraries, the directory structure, logs, etc. have to be copied into the sandbox.

The easiest way to accomplish this, is to get an official static build of MySQL, which doesn't mandatorily rely on external dymanic libraries (shared objects, respectively) and defines the right directory structure. The first step is to download and unpack the package, as shown here by example of MySQL 5.0.45:

```
$ export MYSQL_CHROOT=/chroot/mysql
# mkdir -p $MYSQL_CHROOT
# cd $MYSQL_CHROOT
$ wget http://$SERVER/mysql-5.0.45-linux-i686.tar.gz
$ tar xfz mysql-5.0.45-linux-i686.tar.gz
$ MYSQL_CHROOT=$MYSQL_CHROOT/mysql-5.0.45-linux-i686
$ cd $MYSQL_CHROOT
```

We have now prepared a basically functional MySQL environment. Nevertheless, we want to have at least a working shell, as well as some system-wide configuration files needed by MySQL. Therefore we need to copy `/bin/bash` to the sandbox. Since the Linux Bash also depends on certain libraries, it's necessary to find out which libraries are needed, using the `ldd` command:

```
$ ldd /bin/bash
linux-gate.so.1 =>  (0xffffe000)
libncurses.so.5 => /lib/libncurses.so.5 (0xb7f8f000)
libdl.so.2 => /lib/i686/cmov/libdl.so.2 (0xb7f8b000)
libc.so.6 => /lib/i686/cmov/libc.so.6 (0xb7e42000)
/lib/ld-linux.so.2 (0xb7fd9000)
```

Now we'll just need to copy the given objects in the corresponding directories of the sandbox. This can be done manually file by file, or simply with the following piece of code:

```
$ for i in `ldd /bin/bash | awk '{print␣$3}' | egrep '^/.*'`; do
  mkdir -p "./`dirname␣$i`";
  cp $i ./`dirname $i`;
done
cp /bin/bash ./bin
```

Since MySQL also uses some shell scripts, it will also need the following files:

```
$ for i in /bin/hostname /bin/chown /bin/chmod /bin/touch
     /bin/date /bin/rm /usr/bin/tee /usr/bin/dirname
     /etc/passwd /etc/group /lib/librt.so.1 /lib/libthread.so.0; do
  mkdir -p "./`dirname␣$i`";
  cp $i ./`dirname $i`;
done
```

We can now initially start the MySQL Server inside the chroot-environment by calling:

```
# chroot $MYSQL_CHROOT /bin/mysqld_safe
```

The `chroot` command now repositions the global root node `/` for the command `mysqld_safe`. If an attacker forces to gain access of the system behind the database server, he's limited to MySQL's root directroy, which is represented by the `$MYSQL_CHROOT` environment variable, and pointing to `/chroot/mysql` of the real filesystem behind the sandbox.

2.3 Modern virtualization approaches

Since `chroot` can be seen as an old-school pseudo-virtualisation, just keeping the MySQL server in a sandbox of an existing system, modern approaches have shown that virtualization and para-virtualization are leading the way of running multiple operating system kernels on one machine.

Therefore, there is no need of creating a sandbox, since every server-system may run in a completely isolated full featured Unix system, while all of these (virtual) servers are run on one single physical server.

The most common ways of aquiring an virtual server environment are currently the open-source project Xen as well as the comparable closed-source software VMWare ESX Server. Basically, those projects provide a so called Hypervisor, which can be seen as an additional abstraction layer, between the system's hardware and the operating sytstem's kernels. The hypervisor manages to devide the system resources by the running kernels, independent on which operating systems are used above the hypervisor, without producing much overhead in comparison to natively running the virtualized operating systems.

Since the installation of MySQL on a virtual server is done exactly like a normal installation, I won't provide more information on this topic within this paper, but I'd refer to [SBZD07].

Another way of performing system restrictions are security suites like the NSA SELinux, as well as Novell AppArmor. Those applications aim to spy and restrict the behaviour of certain programs and what they are trying to perform on the filesystem as well as via system calls.

3 Cryptographic appliances

3.1 Encrypting network traffic

For encrypting network traffic, there are several differnet ways. One may use

- OpenSSL,
- OpenSSH tunneling, or

- OpenVPN tunneling

All cryptographic implementations are available for every platform MySQL is capable of, and all three use strong encryption. Using OpenSSL deserves some MySQL internal configuration, and is based on certificates. This may be a good choice if there already is a public-key-infrastructure (PKI) available.

OpenVPN provides a link between two trusted private networks, over an untrusted (mostly non-private) network (normally the Internet). This needs an OpenVPN gateway server, which should commonly not be run on the same machine as the MySQL daemon does due to security reasons. Setting up an VPN tunnel is normally done to encrypt the whole network traffic between two parties, and deserves deeper knowledge of configuring a VPN gateway. Therefore, I won't provide information on this variant, which can be obtained from [BLTR06].

An OpenSSH tunnel is easy to setup and maintain, as well as secure and well-known to most Unix users.

3.1.1 Using OpenSSL

For using OpenSSL encryption, the MySQL server has to be capable of understanding OpenSSL. Most standard MySQL packages of the common Linux distributions already offer OpenSSL-enabled MySQL services out of the box. If not, you may compile the sources of MySQL manually and run the `configure` script with the option `--with-vio --with-openssl`. OpenSSL activation forces the environment variable `have_openssl` to be set to `YES`. This can be checked by

```
mysql> SHOW VARIABLES LIKE '%openssl%';
+---------------+-------+
  | Variable_name | Value |
  +---------------+-------+
  | have_openssl  | YES   |
  +---------------+-------+
```

Since the OpenSSL encryption implementation of MySQL sustains upon certificates, we need to create

1. a Certificate Authority (CA) key and certificate,
2. a server encryption key, as well es a server certificate request

3. a client encryption key, as well as a client certificate request

The following shellscript will do this for us (OpenSSL binaries have to be installed):

```
1  #!/bin/bash
2  DIR=‘pwd‘/openssl
3  PRIV=$DIR/private
4
5  mkdir $DIR $PRIV $DIR/newcerts
6  cp /usr/lib/ssl/openssl.cnf $DIR
7  replace ./demoCA $DIR -- $DIR/openssl.cnf
8
9  openssl req -new -x509 -keyout $PRIV/cakey.pem -out $DIR/cacert.pem
10     -config $DIR/openssl.cnf
11
12 openssl req -new -keyout $DIR/server-key.pem -out $DIR/server-req.pem
13     -days 3600 -config $DIR/openssl.cnf
14
15 # openssl rsa -in $DIR/server-key.pem -out $DIR/server-key.pem
16
17 openssl ca  -policy policy_anything -out $DIR/server-cert.pem
18     -config $DIR/openssl.cnf -infiles $DIR/server-req.pem
19
20 openssl req -new -keyout $DIR/client-key.pem -out $DIR/client-req.pem
21     -days 3600 -config $DIR/openssl.cnf
22
23 # openssl rsa -in $DIR/client-key.pem -out $DIR/client-key.pem
24
25 openssl ca  -policy policy_anything -out $DIR/client-cert.pem
26     -config $DIR/openssl.cnf -infiles $DIR/client-req.pem
```

Lines 1 - 6 create a useable directory structure for storing the resulting keys and certificates. Be sure to call this script from a safe location; keys are normally stored in `/etc/mysql/keys` or something similar.

Line 8 and 9 generate a local Certificate Authority for signing the certificates which are to be created.

Lines 11 and 12 create an encryption key for the MySQL server and a certificate request, which is to be signed afterwards. The certificate will be valid for 3600 days.

Line 14 is optional and would remove the passphrase from the server key. This means that it's not necessary to give the passphrase every time the MySQL server is restartet. This behaviour may be seen as security risk, depending on where the (unencrypted) key will be stored.

Lines 16 and 17 will sign the previously generated server certificate with our local CA instance.

Lines 19 and 20 create a client key and certificate request.

Line 22 is optional, see Line 14.

Lines 24 and 25 sign the client certificate with our local CA instance.

We finally have to tell MySQL where our encryption keys and certificates are stored, which is done in `my.cnf`. We need entries for both, server and client. Note that the client configuration as well as the client and CA certificates have to be available on all clients who wish to encrypt MySQL related network traffic.

```
[client]
ssl-ca=$DIR/cacert.pem
ssl-cert=$DIR/client-cert.pem
ssl-key=$DIR/client-key.pem
<...>

[mysqld]
ssl-ca=$DIR/cacert.pem
ssl-cert=$DIR/server-cert.pem
ssl-key=$DIR/server-key.pem
<...>
```

`$DIR` is to be replaced by the chosen key and certificate directory.

3.1.2 Using OpenSSH

Encrypting network traffic using OpenSSH is done via tunnelling. The advantages of this method are:

- An existing MySQL configuration has not to be altered
- There is no administrative overhead for creating and maintaining certificates and keys
- The tunnel itself is transparant to MySQL since SSH does everything on its own
- Easy setup

However, there are several points which may be seen as disadvantages:

- The tunnelling mechanism itself has to be done on the client(s), which leads to decentralized administration
- The calling client(s) require to have a valid system user on the box where the OpenSSH server is running
- The server machine must run an OpenSSH server[1], the clients must have the `ssh` binary installed

The basic idea is that the `ssh` binary on the client(s) opens a socket which is bound to a specific port (3307 in the following example). `ssh` encrypts all the traffic, coming through this port and sends it to the OpenSSH server which will perform the decryption transparently and redirect the unecrypted traffic to the port, the MySQL server is listening on.

The MySQL TCP connection a client tries to establish, is done to `localhost` instead of the MySQL server, to the port number bound my `ssh`.

On the client side, the following command will set up our OpenSSH tunnel:

```
ssh -L 3307:<MySQL server address>:3306
  <username>@<OpenSSH server address>
```

The clients can now connect through `localhost` the get in touch with the MySQL server:

```
mysql -u <mysql_username> -p -h 127.0.0.1 -P 3307
```

Note: The OpenSSH server doesn't mandatorily have to run on the same machine as the database server does. If OpenSSH runs on server `A` and MySQL on server `B`, we need to set up an packet redirection, which can be done using `iptables` on machine `A`:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
iptables
  -t nat
  -A PREROUTING
  -p tcp
  --dport 3306
```

[1]This is the easiest way, but not unconditionally necessary

```
    -j DNAT
    --to-destination <address of MySQL server>
  iptables
    -t nat
    -A POSTROUTING
    -p tcp
    -d <address of MySQL server>
    --dport 3306
    -j MASQUERADE
```

The statement in line 1 just activates IP packet forwarding in the Linux kernel. The second command activates traffic redirection from the OpenSSH server (where the iptables rulebase is active) to the MySQL database server. Finally, with the third command, we activate masquerading to ensure that responses of the MySQL server are correctly translated and redirected to the calling host (e.g. the MySQL client).

3.2 Encrypting databases on filesystem level

As long as the MySQL server is up and running, and keeping track of incoming queries to provide stored data, the database files have to be unencrypted and readable. It's primarily the job of the DMBS, to only allow authorized users to read and/or write data of certain tables.

Nevertheless, if a harddisk (including backups, tapes, etc.) gets stolen, the stored data is world-readable from every external system. If needed, encryption can solve this problem. Using encryption on filesystem level is quite easy in nowadays 2.6 Linux kernels. The following steps need to have `losetup` and `cryptsetup` installed on the System, as well as a kernel which has been built with `CONFIG_DM_CRYPT` and `CONFIG_BLK_DEV_DM` support (which most of the current kernels have). Most Unices offer the use of encryption, but most of them are not platform independent. When this is a criteria, Truecrypt[2] may be a good choice.

MySQL stores its data in the `$MYSQL_CHROOT/data` directory, we will now encrypt. We will proceed with the following steps:

1. We generate a file with completely randomized content, with the maximum size of the MySQL storage tables (in the following example, 100MiB). If the

[2]www.truecrypt.org

reserved space points out to be too few, we can simply create a bigger one and transfer the encrypted data later.

2. We create a new loopback-device, which is capable of handling our crypted dataimage as harddisk partition.
3. We connect the loopback-device with a so called crypto-target, which encrypts everything which is written onto the target, and decrypts everything which is read from the target, as long as the crypto-target is enabled.
4. Format the crypted data container with a filesystem of our choice (ReiserFS in this case).
5. Mount the crypted container, as it's ready to use.

These steps are done via the following commands:

```
1  # dd if=/dev/urandom of=$MYSQL_CHROOT/data.crypt
2  # losetup /dev/loop0 $MYSQL_CHROOT/data.crypt
3  # cryptsetup -y create mysql_data /dev/loop0
4    Enter passphrase: Passphrase
5    Verify passphrase: Passphrase
6  # mkreiserfs /dev/mapper/mysql_data
7  # mount /dev/mapper/mysql_data $MYSQL_CHROOT/data
```

Now, before starting up the MySQL database server for everyday use, we have to enforce step 2, 3 and 7. Detailed information about the theoretical backgrounds to cryptography may be found in the wonderful reference of Bruce Schneier [Sch05], as well as [Ert03] and [Wae03]. Information on practical filesystem encryption is found in [Pac05].

4 Protection against stack-smashing attacks

Since MySQL has been written in C (and partly C++), the code is implicitly based upon pointer arithmetics and therefore offers a broad spectrum of possible buffer-overflow vulnerabilities. The most common form of buffer overlows are stack-based smashing attacks, since they're normally much easier to produce than heap-based overflows.

Todays high-level programming languages like Java and C# follow a conceptional hiding of pointers to the developer, which, spoken generally, leads to more secure code since overflows nearly always sustain upon exploitable pointer structures. Nevertheless I'm going to figure out some possibly insecure code-snippes of the current MySQL version, before describing howto avoid attacks on them.

Here's an outtake of `mysql-5.0.45/libmysql/libmysql.c:693`

```
1  my_bool STDCALL mysql_change_user(MYSQL *mysql, const char *user,
2                  const char *passwd, const char *db)
3  {
4    char buff[512],*end=buff;
5    int rc;
6    DBUG_ENTER("mysql_change_user");
7
8    if (!user)
9      user="";
10   if (!passwd)
11     passwd="";
12
13   /* Store user into the buffer */
14   end=strmov(end,user)+1;
```

This code is always executed when the calling application intends to change the current MySQL (DBMS-) user. Like shown in line 4, memory for a character buffer `buff` is statically allocated with a size of 512 bytes. When strings have to be passed to a function, C only passes pointers to the beginning of the string, which should be terminated by a NULL-byte (00000000_2), to indicate where the string ends. The function `strmov` at line 14, which does basically the same like ANSI `strcpy`, copies the username (passed to `mysql_change_user()`) in the allocated buffer. However, since the size of the corresponding username has never been checked to be less than 512 bytes, this code represents a classical stack-based buffer overflow.

Moreover, C doesn't has a built-in exception management. If a function fails, is in most cases only shown by the return value. Therefore, not checking the return values of certain, possibly critical, and especially memory mapping functions can be very dangerous and may lead to segmentation faults. The following piece of code shows this (`mysql-5.0.45/innobase/log/log0recv.c:3081`):

```
  log_dir_len = strlen(log_dir);
  /* reserve space for log_dir, "ib_logfile" and a number */
  name = memcpy(mem_alloc(log_dir_len + ((sizeof logfilename) + 11)),
    log_dir, log_dir_len);
  memcpy(name + log_dir_len, logfilename, sizeof logfilename);
```

This code is part of the InnoDB sources, which attempts to be an journaling ACID-compatible database backend. The developer wants to put the `log_dir` string into a newly created buffer called `name`. The memory allocation of `name` is done within the `memcpy` call, and the return value is not checked against 0, which would indicate that the memory allocation has failed. In such a situation, the MySQL database server process will probably get killed by the System, since writing to unallocated memory normally leads to a segmentation fault.

Protection Meanwhile, there are several ways to make C code more secure. One attempt is *libsafe*, which has the advantage that existing programs doesn't have to be changed in any way. Libsafe is a library, that intercepts all system calls of the Standard C Libaray, which are known to be insecure when not exactly checking parameters, return values, etc. [BTS99]. However, it only checks against *attacks*, meaning code or strings, trying to overwrite the return pointer of a function, which is always pushed on the stack when calling; this is normally done to execute homebrewn code, like opening a (root)-shell on the system. In that case, libsafe immediately sends a `SIGKILL` signal to the program. This forces the shutdown of the database server on the one hand, but protects against illegal code execution on the other hand, which is definitly the less bad option.

Libsafe is a libary, which is used with the `LD_PRELOAD` functionality of POSIX systems, allowing to preload specific libraries, which may be called by a program, and relink the call to other functions and/or libraries (which are defined in the `LD_PRELOAD` environment variable). Simply install libsafe and force to set the environment correctly:

```
# LD_PRELOAD=libsafe.so.1
# export LD_PRELOAD
```

Note that `LD_PRELOAD` does only take effect on dynamically linked executables. Because of this, it's not capable of protecting statically linked binaries as shown in the chroot-jail to make the operation easier.

More on buffer overflows and stack smashing can be found in the classical, groundbreaking paper [One], as well as [Fos05] and [Eri03].

5 Security related configuration attributes

The `my.cnf` file may contain a rich set of possible configuration attributes and values, which can change the behaviour of the MySQL server dramatically. The whole file is basically split up into a couple of different sections, each describing the configuration of a specific MySQL executable which is written within bracketes, e.g. *mysqld*, *mysqldump*, *client*, etc. We will further focus on `mysqld` only. The whole set of configuration attributes can be archieved in the MySQL sample configuration files, usually found in `$MYSQL/support-files/`.

Connectivity Securing a database server strongly depends on what is expected from the server. One of the most important questions is the need for remote access to the service. If our database server is just needed by local services, we can achieve a very effective security enhancement by disabling TCP/IP networking of our MySQL instance. This is done by activating the `skip-networking` option. If passed, connections are limited to either UNIX sockets or named pipes.

The `max_connections` defines the maximum of concurrent connections to the server. Note that one of the given amount is always reserved for users with SUPER privileges. Related to this, `max_connect_errors` defines the maximum of errors which may result upon or during connection establishment per user, before he/she is being banned. Setting this value to about 10 should prevent brute-force attacks.

Logging Turning on the `log` parameter, makes MySQL enable full query logging. This means, that every MySQL query (even ones with incorrent syntax) is getting logged. This is either good for debugging reasons on the one hand, and very interesting on detecting certain database attacks like SQL-injections on the other hand.

Transactions and ACIDness `transaction_isolation` defines how MySQL is reacting, if `SELECT` statements are queried upon possibly uncommitted rows and/or tables (*dirty read*). From the security perspective, it's advisable that this value is set to `REPEATABLE-READ` or `SERIALIZABLE`, since both ensure ACID-compatiblity.

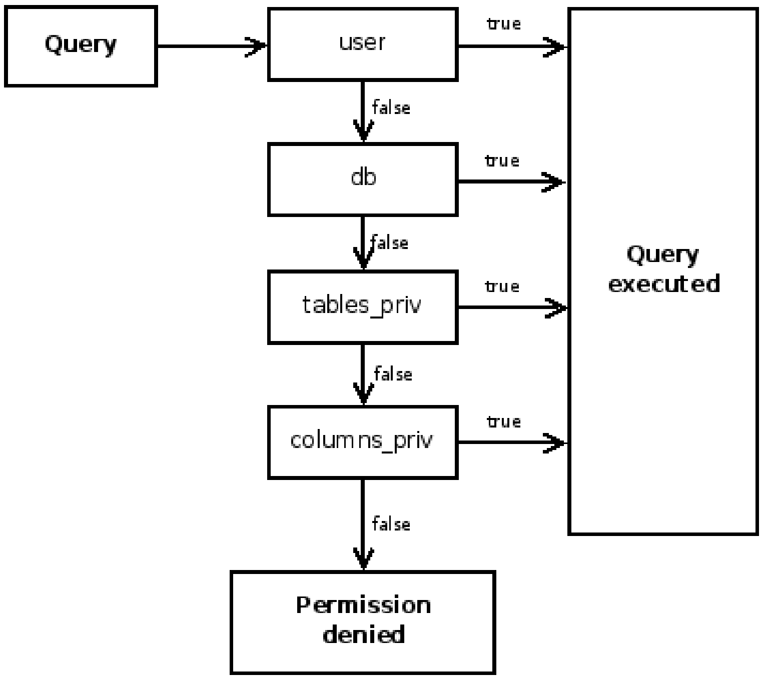
To guarantee ACID compliance, the instance of MySQL has to use a backend, supporting transactions. This is normally done via the InnoDB engine, so it's a good idea to set `default_table_type` to `InnoDB`. The probably most important factor due to the performance of this storing engine, is the `innodb_buffer_pool_size`, which caches indexes and row data of InnoDB tables. On a pure high-performance database server, MySQL AB recommends to set this value up to 80% of the available physical memory. In a maximum address-space of 4GiB on a 32 bit architecture, this value may reach more than 3GiB of memory.

Others The MySQL syntax defines a `LOAD DATA` statement, which provids reading files directly from the filesystem into a table. This command can be very useful for certain administration tasks, but does offer a high potential of attacks. The use of this statement can be prevented by setting `load-infile` to `0` in the configuration file.

6 Access control and privilege management

6.1 General management table structures

MySQL has a built-in access control and privilege management, once more implemented as a relational model in a separate database. Even after freshly installing a database instance, MySQL automatically creates the `mysql` database which holds 6 tables – 5 of them play a certain role of wether a user is allowed to access database objects (table, row, column, etc) or not. Those access rules may be built upon username, connecting host or the requested database.



The `user` table The `user` table is the most important one, since it (besides numerous other things) defines users, their passwords, and the hosts they are allowed to connect from, so are the first 3 columns. The `host` column also accepts wildcards, like `%` as the regular expression `(.*)`. The password is *never* stored in plain text, but normally hashed via the MD5 algorithm. Note that a `user`/`host`-pair is used as primary key.

After those initial values, the `user` table is followed by about two dozen boolean values, giving a more granular description of the permissions granted to the user. The names, like `Insert_priv`, `Update_priv`, etc. are self-speaking. Since those rights have no restriction to certain tables or databases, they should be avoided and set to `N`, whereever possible, for using more restricting levels of access.

When a query is being processed, the permissions of the `user` table are checked at first, and the query is immediately granted if the user has sufficient permissions on this layer. The following listing completes the available columns of the `user` table:

```
mysql> use mysql;
Database changed

mysql> desc user;
+-----------------------+----------------+------+-----+
| Field                 | Type           | Null | Key |
+-----------------------+----------------+------+-----+
| Host                   | char(60)        | NO   | PRI |
| User                   | char(16)        | NO   | PRI |
| Password               | char(41)        | NO   |     |
| Select_priv            | enum('N','Y')   | NO   |     |
| Insert_priv            | enum('N','Y')   | NO   |     |
| Update_priv            | enum('N','Y')   | NO   |     |
| Delete_priv            | enum('N','Y')   | NO   |     |
| Create_priv            | enum('N','Y')   | NO   |     |
| Drop_priv              | enum('N','Y')   | NO   |     |
| Reload_priv            | enum('N','Y')   | NO   |     |
| Shutdown_priv          | enum('N','Y')   | NO   |     |
| Process_priv           | enum('N','Y')   | NO   |     |
| File_priv              | enum('N','Y')   | NO   |     |
| Grant_priv             | enum('N','Y')   | NO   |     |
| References_priv        | enum('N','Y')   | NO   |     |
| Index_priv             | enum('N','Y')   | NO   |     |
| Alter_priv             | enum('N','Y')   | NO   |     |
| Show_db_priv           | enum('N','Y')   | NO   |     |
| Super_priv             | enum('N','Y')   | NO   |     |
| Create_tmp_table_priv  | enum('N','Y')   | NO   |     |
| Lock_tables_priv       | enum('N','Y')   | NO   |     |
| Execute_priv           | enum('N','Y')   | NO   |     |
| Repl_slave_priv        | enum('N','Y')   | NO   |     |
| Repl_client_priv       | enum('N','Y')   | NO   |     |
| Create_view_priv       | enum('N','Y')   | NO   |     |
| Show_view_priv         | enum('N','Y')   | NO   |     |
| Create_routine_priv    | enum('N','Y')   | NO   |     |
| Alter_routine_priv     | enum('N','Y')   | NO   |     |
| Create_user_priv       | enum('N','Y')   | NO   |     |
| ssl_type               | enum('','ANY','X509',
                           'SPECIFIED') | NO   |     |
| ssl_cipher             | blob              | NO   |
| x509_issuer            | blob              | NO   |
| x509_subject           | blob              | NO   |
| max_questions          | int(11) unsigned  | NO   |
| max_updates            | int(11) unsigned  | NO   |
| max_connections        | int(11) unsigned  | NO   |
| max_user_connections   | int(11) unsigned  | NO   |
+------------------------+------------------+------+
37 rows in set (0.01 sec)
```

As listed, `user` additionally defines four columns related to cryptographic methods like ciphers and certificates, and four columns used for user-specific limitations on the database, we will inspect later.

The `db` table The `db` table is checked (only), if the `user` table doesn't define enough permissions for a user to fully process the query. `db` again defines username, connecting host, and numerous privileges on a certain database, given by the column `Db`. This table is only processed, if

1. the user doesn't has sufficient permissions in the `user` table, and
2. the user wants to set up a query on a database, defined in the `db` table.

The `host` table This is basically the same as the `db` table, but acting on actual hosts, the query may come from and may be restricted to.

The `tables_priv` and `columns_priv` table The `tables_priv` table exactly defines the permissions of users on per-table-basis, who may or may not set up select, insert, update, delete, create, drop, grant, references, index and alter commands. Also the Grantor, the timestamp of the `GRANT`-statement and of course username, database name and hostname are stored here. This is possibly the table where user-based restrictions should be done.

In comparison, the `columns_priv` table is structured like `tables_priv`, but holds less permissions and additionally defines a `column_name` column, telling us to which column the restriction/permission is refering.

6.2 Access management via SQL

All permissions and restrictions stored in the `mysql` database, are classically managed via SQL, mainly using `GRANT` and `REVOKE` statements.

A `GRANT` statement consists of the permissions which are to be set, as well as the database and table it is refering to, and a user/hostname pair. For example:

```
GRANT SELECT, UPDATE on mysql.user TO root@localhost IDENTIFIED BY 'pass
```

The `REVOKE` command is used adequatly. For a detailed description on `GRANT` and `REVOKE` you may consider having a look on the official MySQL reference [Vas04].

There is no main difference between setting up permissions via the tables inside the `mysql` database using DML or typing SQL `GRANT` and `REVOKE` statements. However, while the latter version will activate the permissions immediately, privilege settings applied by direct DML, deserve reloading the values. This can be done via `FLUSH PRIVILEGES`.

There a several privileges only used for database administration, namely

- `PROCESS`, allowing the user to perform the `processlist` command,
- `SHUTDOWN`, allowing the user to shutdown the MySQL server via the `shutdown` command,
- `SUPER`, allowing the user to perform the `kill` command for killing certain MySQL threads,
- `RELOAD`, allowing the user to perform `flush-hosts`, `flush-logs`, `flush-privileges`, `flush-status`, `flush-tables`, `flush-threads`, `refresh` as well as `reload` commands.

Note, that these privileges are commonly *not* used via SQL-statements, but through using the `mysqladmin` shell command. This is a security related model, since a user who intends to force privilege escalation atempts on the MySQL server, will not be able to use this commands inside the standard MySQL shell. The above rights should be reduced to an absolute minimum of users.

6.3 Setting up connection limits

As shown in the table description of `user`, there are several options MySQL offers to limit certain resources of specific users.

This includes three main clauses:

- The `MAX_QUERIES_PER_HOUR` clause defines a maximum set of queries which may be processed on per user and per host basis. For example, the statement `GRANT SELECT on *.* TO root WITH MAX_QUERIES_PER_HOUR` will limit the maximum queries available to user `root` to an amount of five per hour.
- `MAX_UPDATES_PER_HOUR`, controls the maximum amount of DML statements per hour, and
- `MAX_CONNECTIONS_PER_HOUR` controls the maximum of connection establishments per hour.

All of those clauses cannot be applied on per-table or per-database basis, since they have to be stated via `*.*`. Every mentioned limitation is internally represented by counters, corresponding to the time (per hour). Those counter may easily be reset by invoking the command `FLUSH USER_RESOURCES` (the user which tries to flush, will need the `RELOAD` privilege). This statement will *not* remove the defined resource limits, but reset the counters.

7 Conclusion

There is no absolute security for applications. The offered methods and technologies mentioned in this paper, can help making the environment much more secure where the MySQL daemon is running.

We may use technologies like sandboxing and virtualization for isolating the MySQL processes from the environment, the database server is running in. This minimizes the possible negative consequences, if the daemon is getting compromised.

The deployment and use of cryptographic routines for ciphering physical data and network traffic, reduces the risks of sniffing and man-in-the-middle attacks, as well as securing the whole data covered by the database if the data directory itself gets theft.

A very big disadvantage of using programming languages which explicitely make use of pointers like C or C++, is the possibility of buffer overflows and attacks using this as basis. That's not a conceptional mistake of MySQL, but makes the spectrum of possible attacks much wider. Using certain external software for checking those leaks is highly recommended. In such a case, the database server will just be terminated - which is not a desirable consequence, but far better than having an up and running but compromised instance.

References

[AB05] MySQL AB. Inside mysql 5.0 - a dba's perspective, 2005.

[Ale06] Michael Alexander. *Netzwerke und Netzwerksicherheit.* Huehtig Telekommunikation, 2006. (ISBN 3826650484).

[Amb07] Eric Amberg. *Linux-Server mit Debian.* mitp, 2007. (ISBN 3826615875).

[Bau02] Michael Bauer. *Building secure servers with Linux.* O'Reilly, 2002. (ISBN 0596002173).

[BLTR06] Johannes Bauer, Albrecht Liebscher, and Klaus Thielking-Riechert. *OpenVPN. Grundlagen, Konfiguration, Praxis.* Dpunkt Verlag, 2006. (ISBN 3898643964).

[BTS99] Arash Baratloo, Timothy Tsai, and Navjog Singh. libsafe manual page. libsafe library, 1999.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. Communications of the ACM 13 (6), 377-387, 1970.

[Eri03] Jon Erickson. *Hacking - the art of exploitation.* No starch press, 2003. (ISBN 1593270070).

[Ert03] Wolfgang Ertel. *Angewandte Kryptographie.* Hanser Fachbuchverlag, 2003. (ISBN 3446223045).

[Fos05] James Foster. *Buffer overflow attacks.* Syngres Media, 2005. (ISBN 1932266674).

[Gri05] Lenz Grimmer. Mysql backup and security, 2005.

[Kre04] Juergen Kreileder. Chrooting mysql on debian, 2004.

[MBBS07] Keith Murphy, Peter Brawley, Dan Buettner, and Baron Schwartz. Mysql magazine, 2007. Issue 1.

[One] Aleph One. Smashing the stack for fun and profit. Phrack magazine vol 49, File 14 of 16.

[Pac05] Lars Packshies. *Praktische Kryptographie unter Linux.* Open source press, 2005. (ISBN: 3937514066).

[PW07] Johannes Ploetner and Steffen Wendzel. *Netzwerksicherheit.* Galileo press, 2007. (ISBN 3898428286).

[SBZD07] Henning Sprang, Timo Benk, Jaroslaw Zdrzalek, and Ralph Dehner. *Xen. Virtualisierung unter Linux.* Open source press, 2007. (ISBN 3937514295).

[Sch05] Bruce Schneier. *Angewandte Kryptographie. Algorithmen, Protokolle und Sourcecode in C.* Pearson Studium, 2005. (ISBN 0471117099).

[SR07] M. Stipcevic and B. Medved Rogina. Quantum random number generator. Rudjer Boskovic Institute, Bijenicka, Zagreb, Croata, 2007.

[SWF05] Ellen Siever, Aaron Weber, and Stephen Figgins. *Linux in a nutshell.* O'Reilly, 2005. (ISBN 0596009305).

[Vas04] Vikram Vaswani. *MySQL: The complete reference.* Mcgraw-Hill Professional, 2004. (ISBN 0072224770).

[Wae03] Dietmar Waetjen. *Kryptographie. Grundlagen, Algorithmen, Protokolle.* Spektrum Adakemischer Verlag, 2003. (ISBN 3827414318).