JAN 17, 2016 / JVM,SECURITY

The Java Security Manager: why and how?

Generally, security concerns are boring for developers. I hope this article is entertaining enough for you to read it until the end since it tackles a very serious issue on the JVM.<!--more-->

Quizz

Last year, at Joker conference, my colleague Volker Simonis showed a snippet that looked like the following:

```
public class StrangeReflectionExample {

    public Character aCharacter;

    public static void main(String... args) throws Excep
        StrangeReflectionExample instance = new StrangeR
        Field field = StrangeReflectionExample.class.get
        Field type = Field.class.getDeclaredField("type"
        type.setAccessible(true);
        type.set(field, String.class);
        field.set(instance, 'A');
        System.out.println(instance.aCharacter);
    }
}
```

Now a couple of questions:

1. Does this code compile?

2. If yes, does it run?
3. If yes, what does it display?

Answers below.

This code compiles just fine. In fact, it uses the so-called reflection API (located in the `java.lang.reflect` package) which is fully part of the JDK.

Executing this code leads to the following exception:

```
Exception in thread "main" java.lang.IllegalArgumentExce
        Can not set java.lang.String field ch.frankel.bl
            to java.lang.Character
        at sun.reflect.UnsafeFieldAccessorImpl.throwSetI
        at sun.reflect.UnsafeFieldAccessorImpl.throwSetI
        at sun.reflect.UnsafeObjectFieldAccessorImpl.set
        at java.lang.reflect.Field.set(Field.java:764)
        at ch.frankel.blog.securitymanager.StrangeReflec
        at sun.reflect.NativeMethodAccessorImpl.invoke0(
        at sun.reflect.NativeMethodAccessorImpl.invoke(Na
        at sun.reflect.DelegatingMethodAccessorImpl.invo
        at java.lang.reflect.Method.invoke(Method.java:4
        at com.intellij.rt.execution.application.AppMain
```

So, despite the fact that we defined the type of the `aCharacter` attribute as a `Character` at development time, the reflection API is able to change its type to `String` at runtime! Hence, trying to set it to `'A'` fails.

Avoiding nasty surprises with the Security Manager

Reflection is not the only risky operation one might want to keep in check on the JVM. Reading a file or writing one also belong to

the set of potentially dangerous operations. Fortunately, the JVM has a system to restrict those operations. Unfortunately, it's not set by default.

In order to activate the SecurityManager, just launch the JVM with the `java.security.manager` system property *i.e.* `java -Djava.security.manager` . At this point, the JVM will use the default JRE policy. It's configured in the file located at `%JAVA_HOME%/lib/security/java.policy` (for Java 8). Here's a sample of this file:

```
grant codeBase "file:${java.ext.dirs}/*" {
        permission java.security.AllPermission;
};

grant {
        permission java.lang.RuntimePermission "stopThre
        permission java.net.SocketPermission "localhost:
        permission java.util.PropertyPermission "java.ve
        permission java.util.PropertyPermission "java.ve
        ...
}
```

The first section - `grant codeBase` , is about which code can be executed; the second - `grant` , is about specific permissions.

Regarding the initial problem regarding reflection mentioned above, the second part is the most relevant. One can read the source of the `AccessibleObject.setAccessible()` method:

```
SecurityManager sm = System.getSecurityManager();
if (sm != null) sm.checkPermission(ACCESS_PERMISSION);
setAccessible0(this, flag);
```

Every sensitive method to a Java API has the same check through the Security Manager. You can verify that for yourself in the following code:

- `Thread.stop()`
- `Socket.bind()`
- `System.getProperty()`
- etc.

Using an alternate java.policy file

Using the JRE's policy file is not convenient when one uses the same JRE for different applications. Given the current micro-service trend, this might not be the case. However, with automated provisioning, it might be more convenient to always provision the same JRE over and over and let each application provides its own specific policy file.

To **add** *another* policy file in addition to the default JRE's, thus adding more permissions, launch the JVM with:

```
java -Djava.security.manager -Djava.security.policy=/pat
```

To replace the default policy file with your own, launch the JVM with:

```
java -Djava.security.manager -Djava.security.policy==/pa
```

Note the double equal sign.

Configuring your own policy file

Security configuration can be either based on a:

Black list:

In a black list scenario, everything is allowed but exceptions can be configured to disallow some operations.

White list:

On the opposite, in a white list scenario, only operations that are explicitly configured are allowed. By default, all operations are disallowed.

If you want to create your own policy file, it's suggested you start with a blank one and then launch your app. As soon, as you get a security exception, add the necessary permission is the policy. Repeat until you have all necessary permissions. Following this process will let you have only the minimal set of permissions to run the application, thus implementing the least privilege security principle.

Note that if you're using a container or a server, you'll probably require a lot of those permissions, but this is the price to pay to secure your JVM against abuses.

Conclusion

I never checked policy files in production, but since I never had any complain, I assume the JVM's policy was never secured. This is a very serious problem! I hope this article will raise awareness regarding that lack of hardening - especially since with the latest JVM, you can create and compile Java code on the fly, leading to even more threats.

To go further:

- Default Policy Implementation and Policy File Syntax
- SecurityManager Javadoc

Follow @nicolas_frankel

Nicolas Fränkel

Nicolas Fränkel is a Developer Advocate with 15+ years experience consulting for many different customers, in a wide range of contexts (such as telecoms, banking, insurances, large retail and public sector). Usually working on Java/Java EE and Spring technologies, but with focused interests like Rich Internet Applications, Testing, CI/CD and DevOps. Currently working for Hazelcast. Also double as a teacher in universities and higher education schools, a trainer and triples as a book author.

Read More

bloodyroo7 • a month ago

Thank you for raising awareness!
On a talk on this topic you somehow programmatically collected the minimal grants you needed to execute some code with a custom policy file. How did you manage this?

Reply • Share ›

Nicolas Frankel Mod bloodyroo7 • a month ago

You might want to read the whole focus. Otherwise, just read this post that tackles the subject

1 Reply • Share ›

Eugene Berezhnoy • 4 months ago • edited

Hi, on Java 11 the code above works and displays
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by net.bestengineers.reflection.StrangeReflectionExample (file:/D:/github/gradleplayground/out/production/gradleplayground/) to field java.lang.reflect.Field.type
WARNING: Please consider reporting this to the maintainers of net.bestengineers.reflection.StrangeReflectionExample
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
A

But if you change it this way :
field.set(instance, "A");
It throwa an Exception
Exception in thread "main" java.lang.IllegalArgumentException: Can not set java.lang.Character field net.bestengineers.reflection.StrangeReflectionExample.aCharacter to java.lang.String

So now Java not allow change type of field

Reply • Share ›

— A Java geek —

Jvm

Crafting Java policy files, a practical guide

Synthetic

Signing and verifying a standalone JAR

See all 5 posts →

JAN 31, 2016

HTML

SECURITY

Why you shouldn't trust the HTML password input

This week, I wanted to make a simple experiment. For sure, all applications we develop make use of HTTPS to encrypt the login/password but what happens before? Let's say I typed my login/password but before sending them, I'm called by my colleague and I leave my computer open. My password is protected by the HTML password input, right? It shows stars instead of the real characters. Well, it's stupidly easy to circumvent this. If you use a developer workstation and have develop

NICOLAS FRÄNKEL

JAN 10, 2016

KOTLIN
SPRING BOOT
VAADIN

Playing with Spring Boot, Vaadin and Kotlin

It’s no mystery that I’m a fan of both Spring Boot and Vaadin. When the Spring Boot Vaadin add-on became GA, I was ecstatic. Lately, I became interested in Kotlin, a JVM-based language offered by JetBrains. Thus, I wanted to check how I could develop a small Spring Boot Vaadin demo app in Kotlin - and learn something in the process. Here are my discoveries, in no particular order. Spring needs non-final stuff It seems Spring needs @Configuration classes and @Bean methods to be non

NICOLAS FRÄNKEL