How to shoot yourself in the foot building a Java Agent

javacodegeeks.com/2017/06/shoot-foot-building-java-agent.html

Over the years of building Plumbr, we have encountered many challenging problems. Among others, making the Plumbr Java Agent perform reliably without endangering the customers' applications, is a particularly tricky one. To safely gather all the required telemetry from a live system poses a huge set of problems to tackle. Some of them are quite straightforward while some of them are fiendishly non-obvious.

In this blog post, we would like to share with you several examples demonstrating the complexities encountered in building the support for some seemingly simple aspects our Agents need to cope with. The examples are simplified a bit, but are extracted from real world problems we needed to solve some time ago. In fact, these are just the tip of the iceberg awaiting those who try to use byte code instrumentation or JVMTI.

Example 1: Instrumenting a simple web application

Let's start with a very simple hello world web app:

```
1  @Controller
2  public class HelloWorldController {
3
4      @RequestMapping("/hello")
5      @ResponseBody
6      String hello()
       {
7          return "Hello, world!";
8      }
9  }
```

If we start the application and access the relevant controller, we will see this:

```
1  $ curl localhost:8080/hello
2  Hello, world!
```

As a simple exercise, let us change the returned value to "Hello, transformed world". Naturally, our real java agent would not do such a thing to your application: our goal is to monitor without changing the observed behaviour. But bear with us for the sake of keeping this demo short and concise. To change the returned response, we will be using ByteBuddy:

```
01              ServletAgent
    public class {
02
03                    premain(String arguments, Instrumentation instrumentation)
     public static void {
04     new AgentBuilder.Default()
05           .type(isSubTypeOf(Servlet.class))
06                        ) -
             .transform((>
07             builder.method(named("service"))
08                    .intercept(
09                      MethodDelegation.to(Interceptor.class)
10                    )
11           ).installOn(instrumentation);
12   }
13
14  }
```

What's happening here:

1. As is typical for java agents, we supply a pre-main method. This will be executed before the actual application starts. If you are curious for more, ZeroTurnaround has an excellent post for more info on how instrumenting java agents work.
2. We find all classes that are sub-classes of the Servlet class. The Spring magic eventually unfolds into a Servlet as well.
3. We find a method named "service"
4. We intercept calls to that method and delegate them to our custom interceptor that simply prints "Hello, transformed world!" to the ServletOutputStream.
5. Finally, we tell ByteBuddy to instrument the classes loaded into the JVM according to the rules above

Alas, if we try to run this, the application no longer starts, throwing the following error:

```
1  java.lang.NoSuchMethodError:
   javax.servlet.ServletContext.getVirtualServerName()Ljava/lang/String;
```

```
2
   at
   org.apache.catalina.authenticator.AuthenticatorBase.startInternal(AuthenticatorBase.java:
   1137)
```

```
3      at org.apache.catalina.util.LifecycleBase.start(LifecycleBase.java:150)
```

What happened? We only touched the “service” method on the “Servlet” class, but now the JVM cannot find another method on another class. Fishy. Let’s try to see where that class is being loaded from in both cases. To do that, we can add the -XX:+TraceClassLoading argument to the JVM startup script. Without the java agent, the class in question is loaded from Tomcat:

```
1  [Loaded javax.servlet.ServletContext from jar:file:app.jar!/BOOT-INF/lib/tomcat-embed-
   core-
   8.5.11.jar!/]
```

However, if we enable the java agent again, then it’s loaded from elsewhere:

```
1  [Loaded javax.servlet.ServletContext from
   file:agent.jar]
```

Aha! Indeed, our agent has a direct dependency on the servlet API defined in the Gradle build script:

```
1  agentCompile"javax.servlet:servlet-api:2.5"
```

Sadly, this version does not match the one that is expected by Tomcat, hence the error. We used this dependency to specify which classes to instrument: *isSubTypeOf*(Servlet.**class**), but this also caused us to load an incompatible version of the servlet library. It is actually not that easy to get rid of this: to check if a class that we are trying to instrument is a subtype of another type, we have to know all of its parent classes or interfaces.

While the information on the direct parent is present in the bytecode, the transitive inheritance is not. In fact, the relevant classes may not even have been loaded yet when we’re instrumenting. To work around that, we have to figure out the entire class hierarchy of the client’s application at runtime. Gathering the class hierarchy efficiently is a difficult feat that has a lot of pitfalls on its own, but the lesson here is clear: instrumentation should not load classes that the client’s application may want to load as well, especially coming from non-compatible versions.

This is just a little baby dragon that has strayed away from the legions awaiting you when you try to instrument bytecode or try to mess with classloaders. We’ve seen many many more issues: classloading deadlocks, verifier errors, conflicts between multiple agents, native JVM structure bloating, you name it!

Our agent, however, does not limit itself to using the Instrumentation API. To implement some of the features, we have to go deeper.

Example 2: Using JVMTI to gather information about classes

There are many different ways one could take to figure out the type hierarchy, but in this post let us focus on just one of them – JVMTI, the JVM Tool Interface. It allows us to write some native code that can access the more lower-level telemetry and tooling functions of the JVM. Among other things, one may subscribe for JVMTI callbacks for various events happening in the application or the JVM itself. The one we’re currently interested in is the ClassLoad callback. Here’s an example of how we could use it to subscribe to class loading events:

```
01            register_class_loading_callback(jvmtiEnv* jvmti)
    static void {
```

```
02     jvmtiEventCallbacks
       callbacks;

03     jvmtiError
       error;

04

05                         ,
       memset(&callbacks,0sizeof(jvmtiEventCallbacks));

06

07     callbacks.ClassLoad =
       on_class_loaded;

08

09     (*jvmti)->SetEventCallbacks(jvmti, &callbacks,
       sizeof(callbacks));

10
    (*jvmti)->SetEventNotificationMode(jvmti, JVMTI_ENABLE, JVMTI_EVENT_CLASS_LOAD,
    (jthread)NULL);

11  }
```

This will make the JVM execute the on_class_loaded function, defined by us, at an early stage in class loading. We can then write this function so it calls a java method of our agent via JNI like so:

```
1  void
   JNICALL on_class_loaded(jvmtiEnv *jvmti, JNIEnv* jni, jthread thread, jclass klass)
   {

2      (*jni)->CallVoidMethod(jni, agent_in_java, on_class_loaded_method,
       klass);

3  }
```

In the Java Agent, for simplicity's sake, we will just print the name of the class:

```
1                   onClassLoaded(Class clazz)
   public static void {

2                       "Hello,   +
       System.out.println("         clazz);

3  }
```

Close your eyes for a minute and try to imagine what could *possibly* go wrong here.

Many of you probably thought that this is simply going to crash. After all, every mistake you make in native code has a chance of bringing your entire application down with a segfault. However, in this particular example, we're actually going to get some JNI errors and a java exception:

```
1 Error: A JNI error has occurred, please check your installation and try again
2 Error: A JNI error has occurred, please check your installation and try again
3 Hello,class java.lang.Throwable$PrintStreamOrWriter
4 Hello,class java.lang.Throwable$WrappedPrintStream
5 Hello,class java.util.IdentityHashMap
6 Hello,class java.util.IdentityHashMap$KeySet
7 Exception in thread "main" java.lang.NullPointerException
8   At JvmtiAgent.onClassLoaded(JvmtiAgent.java:23)
```

Let us put the JNI errors aside for now and focus on the java exception. It's surprising. What could be null here? Not that many options, so let's just check them and run again:

```
01 public static void onClassLoaded(Class clazz) {
02     if(System.out == null){
03         throw new AssertionError("System.out is null");
04     }
05
06     if(clazz == null){
07         throw new AssertionError("clazz is null");
08     }
```

```
09
10                      "Hello,   +
      System.out.println("        clazz);
11  }
```

But alas, we still get the same exception:

```
1  Exception in
   thread             "main" java.lang.NullPointerException
2    At JvmtiAgent.onClassLoaded(JvmtiAgent.java:31)
```

Let us hold this for a moment, and make another simple change to the code:

```
1                  onClassLoaded(Class clazz)
   public static void {
2                      "Hello,   +
     System.out.println("        clazz.getSimpleName());
3  }
```

This seemingly insignificant change in output format makes for a dramatic change in behaviour:

```
1  Error: A JNI error has occurred, please check your installation
   and                                                        try again
2  Error: A JNI error has occurred, please check your installation
   and                                                        try again
3  Hello,
   WrappedPrintWriter
4  Hello,
   ClassCircularityError
5  #
6  # A fatal error has been detected by the Java Runtime
   Environment:
7  #
8  #  Internal Error                                    ,
   (systemDictionary.cpp:                 806), pid=82384tid=  0x0000000000001c03
```

```
9  #  guarantee((!class_loader.is_null())) failed: dup
   definition                                          for bootstrap loader?
```

Ah, finally a crash! What a delight! In fact, this gives us a lot of information very helpful in pinpointing the root cause. Specifically, the now apparent ClassCircularityError and the internal error message are very revealing. If you were to look at the relevant part of the source code of the JVM, you would find an immensely complicated and intermingled algorithm for resolving classes. It does work on its own, fragile as it is, but is easily broken by doing something unusual like overriding ClassLoader.loadClass or throwing in some JVMTI callbacks.

What we're doing here is sneaking class loading into the middle of loading classes, and that does seem like a risky business. Skipping the troubleshooting that would take a blog post of its own and involves a lot of native digging, let's just outline what's happening in the first example:

1. We try to load a class, e.g. launcher.LauncherHelper
2. To print it out, we try to load the io.PrintStream class, recursing to the same method. Since the recursion happens through the JVM internals and JVMTI and JNI, we do not see it in any stack traces.
3. Now have to print out the PrintStream as well. But it's not quite loaded yet, so we get a JNI error
4. Now we go on and try to continue printing. To concatenate strings, we need to load lang.StringBuilder. The same story repeats.
5. Finally, we get a null pointer exception because of the not-quite-loaded classes.

Well, that is quite complicated. But after all, the JVMTI doc says quite explicitly that we should exercise extreme caution:

"This event is sent at an early stage in loading the class. As a result the class should be used carefully. Note, for example, that methods and fields are not yet loaded, so queries for methods, fields, subclasses, and so on will not give correct results. See "Loading of Classes and Interfaces" in the Java Language Specification. For most purposes the ClassPrepare event will be more useful."

Indeed, if we were to use this callback, then there would be no such difficulties. However, when designing a Java Agent for monitoring purposes, we are sometimes forced to go to the very dark areas of the JVM to support the product features we need with overhead low enough for production deployments.

Take-away

These examples demonstrated how some seemingly innocent set-ups and naive approaches to building a java agent can blow up in your face in surprising ways. In fact, the above barely scratches the surface of what we've discovered over the years.

Couple this with the sheer number of different platforms such agents would need to run flawlessly (different JVM vendors, different Java versions, different operating systems) and the already complex task becomes even more challenging.

However, with due diligence and proper monitoring, building a reliable java agent is a task that can be tackled by a team of dedicated engineers. We confidently run Plumbr Agent in our own production and don't lose any sleep over it.

Reference: How to shoot yourself in the foot building a Java Agent from our JCG partner Gleb Smirnov at the Plumbr Blog blog.