HOTSOURCE

# JVMTI

## The JVM Tool Interface

how tools you use utilize it and how it helps you gather information about JVM apps

Kevin Watzal

# ABOUT ME

HOTSOURCE

Passionate about JVM since school

Developer at HOTSOURCE

Studied IT-Security

JVMTI was the main focus of my master thesis

Who of you have heard of JVMTI before?

HOTSOURCE

# NULL POINTER EXCEPTION
# IN JAVA 11

```
Exception in thread "main" java.lang.NullPointerException
    at io.kay.Main.main(Main.java:7)
```

```
5 public static void main(String[] args) {
6 // 3 possibilities which could be null in this line
7 var value =
  Helper.getMap().get("data").getData().toLowerCase();
8 System.out.println(value);
9 }
```

JVMTI to the rescue!

# JVM TOOL INTERFACE

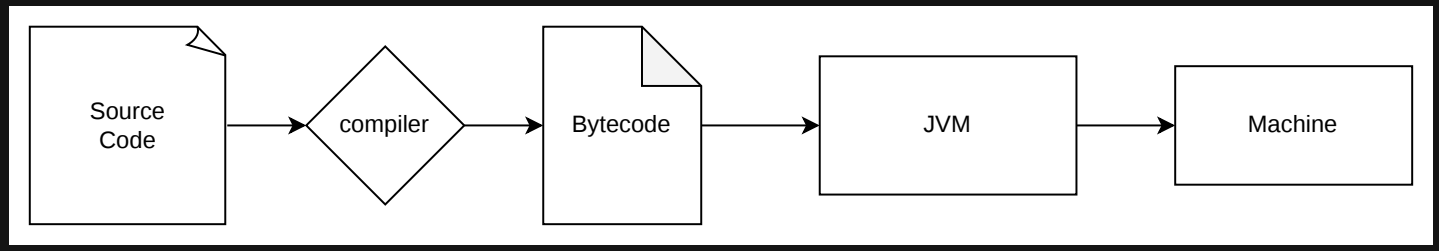
An API which allows to create **agents** to:

- Actively inspect the state of the JVM
- Passively get notified about certain events
- Alter the execution of the running application

HOTSOURCE

# LET'S TALK BASICS

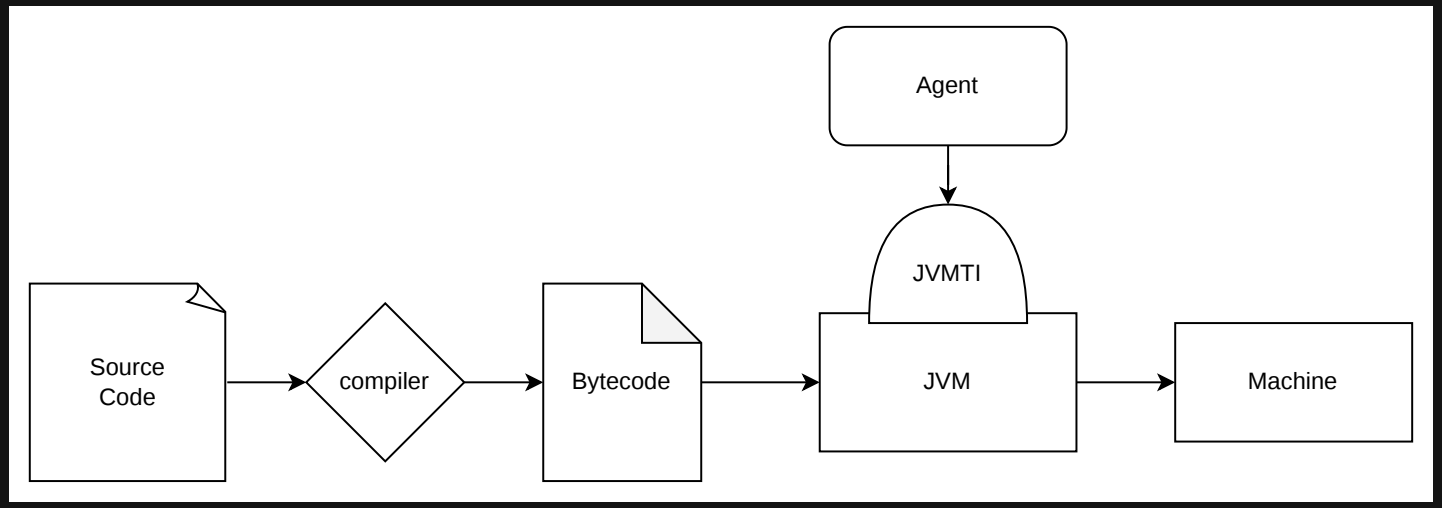
What do you need to run a program in the JVM?




Bytecode & JVM are standardized

The JVM is natively compiled for the machine it runs on

# WHERE'S JVMTI NOW?



# THERE IS A CATCH, THOUGH

Since the JVM is natively compiled, the agent must be natively compiled as well

**That means writing code in C/C++**

Just like bytecode and the JVM,
JVMTI is **standardized** as well

# CAUTION!

JVMs are **not required** to implement it

Everything from now on references Oracle's JVMs
(and was tested with OpenJDK)

# WHICH TOOLS USE JVMTI?

- Debuggers (e.g. IntelliJ, JPDA/JDWP)
- Profilers (e.g. DataDog, Lightrun)
- Security Analyzers (back in Applet-time)
- Bypassing security measures (e.g. license checks)
- Specific solutions for specific problems

# JVMTI CAN BE USEFUL FOR:

- Tracking of threads, objects, garbage collection
- Inspect field access/modification or class loading
- Custom Debugging
- Modifying execution flow or values
- Combine with JNI(Java Native Interface)
- Bytecode instrumentation

# THE JVM OPERATES ON BYTECODE

and so does JVMTI

# DECLARE YOUR CAPABILITIES

Tell the JVM what your agent wants to do

```
jvmtiCapabilities capabilities;
capabilities.can_generate_exception_events = 1;
capabilities.can_get_bytecodes = 1;
capabilities.can_get_constant_pool = 1;
(*jvmti)->AddCapabilities(jvmti, &capabilities);
```

# EVENTS/CALLBACKS

Register your callbacks to **receive events**

```
void MethodEntryCallback(jvmtiEnv *jvmti,
                         JNIEnv *jni,
                         jthread thread,
                         jmethodID method) {
}
//...
jvmtiEventCallbacks callbacks;
callbacks.MethodEntry = (void *) &MethodEntryCallback; // link ca

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

// activate - does not need to be on agent startup
```

# ATTACH THE AGENT TO THE JVM AT STARTUP

```
make all # compile first
java -agentpath:./agent.so -jar application.jar
```

HOTSOURCE

# LET'S START WITH EXAMPLES!

# CVE 2024-38819

Allows for path-traversal in Spring Boot under certain circumstances

Is detectable via a JVMTI agent

# BEWARE

Don't simply add any (untrusted) agent to your jar

JVMTI applied to Spring applications may be more complex

JVMTI adds runtime overhead to your application

Callbacks are **blocking** operations