Utilizing RacerD (Infer) for Statistical Analysis and Inference of Potential Concurrency Bugs

Aim: Explore data races in real-world Java projects and locate four distinct data races.
Project Investigated: Apache Tomcat (url: https://github.com/apache/tomcat)

Key Definitions

`Race Condition.` A property of an *algorithm* (or program) that is manifested in displaying anomalous outcomes or behavior becuase of the unfortunate ordering of events.

`Data Race.` A property of an *execution* of a program which contains at least two conflicting (non-volatile) accesses to a shared variable that are not properly coordinated/synchronous (i.e. not ordered by a 'happens-before' relation)

*****

Background on Apache Tomcat

Apache Tomcat is an open source project that implements Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies. Java Servlet (also known as Jakarta Servlet) and WebSocket are softwares that allow programmers to connect to and extend the capabilities of a server. Many developers around the world leverage this technology to implement containers to host web application. JavaServer Pages is a large collection of frameworks that assist developers in creating dynamically generated web pages based on HTML, XML, and other document types. All in all, Apache Tomcat is a large-scale project that assists programmers daily in both front-end and back-end.

Though the scale and complexity of the project makes it difficult to comprehend the precise purpose behind each java file, I could observe (from studying a few scripts) that the benefits of concurrent programming are significantly leveraged in this project. Many of the scripts I have studied were related to the server-side operations (e.g. establishing network connection, application/transport layer protocols, creating datagrams/packets, etc...) which all used concurrent programming. For instance, I observed that the implementation of the HTTP/2 connection was facilitated by multiple threads (synchronized by socket). This makes sense as programmers are always looking to increase the maximal utilization of the bandwidth through faster and more efficient connections and packet transmissions: I would assume that involving as many threads and as much computing power as gatherable is favorable. I also learned that the implementation of JavaServer Pages (e.g. xml encoding, xml mapping, etc...) leverages concurrency to optimize its efficiency and speed.

Data Races identified by RacerD (Infer)

1. Asynchronous Counter

Error Report:

```
#613
java/org/apache/catalina/ha/session/DeltaManager.java:1272: warning: Thread
Safety Violation
```

```
Unprotected write. Non-private method ‘DeltaManager.handleALL_SESSION_DATA(...)‘
writes to field ‘this.counterReceive_EVT_ALL_SESSION_DATA‘ outside of
synchronization.
 Reporting because another access to the same memory occurs on a background
 thread, although this access may not.
1270.      protected void handleALL_SESSION_DATA(SessionMessage msg,Member sender)
1271.              throws ClassNotFoundException, IOException {
1272. >        counterReceive_EVT_ALL_SESSION_DATA++;
1273.          if (log.isDebugEnabled()) {
1274.              log.debug(sm.getString("deltaManager.receiveMessage.allSessi-
                  -onDataBegin", getName()));
```

In the above instance, RacerD has identified the location in our code (L1272 in DeltaManager.java) which it suspects will cause data race in our execution. The reason it believes that this line infringes thread safety is because it suspects that a thread that calls the method DeltaManager.handleALL_SESION_DATA(...) will be writing to the shared data field ‘this.counterReceive_EVT_ALL_SESSION_DATA’ without multi-thread synchronization.

To analyze and verify whether this error message raised by RacerD is a true error that can cause a data race or a false positive, I examined the DeltaManager.java script (and linked scripts).

Script/Lines of interest:

```java
private long counterReceive_EVT_ALL_SESSION_DATA = 0;
// ...
public long getCounterReceive_EVT_ALL_SESSION_DATA() {
    return counterReceive_EVT_ALL_SESSION_DATA;
}
public synchronized void resetStatistics() {
    // ...
    counterReceive_EVT_ALL_SESSION_DATA = 0;
}
protected void handleALL_SESSION_DATA(SessionMessage msg,Member sender)
    throws ClassNotFoundException, IOException {
        // ...
        counterReceive_EVT_ALL_SESSION_DATA++;
        // ...
}
```

My Verdict: As suspected, RacerD has successfully identified a potential breach to Thread Safety. As can be seen in the code block above, `counterReceive_EVT_ALL_SESSION_DATA` is declared as long type and is not attached with any volatile annotation. Furthermore, although several operations read (e.g. `getCounterReceive_EVT_ALL_SESSION_DATA()` and write (e.g. `resetStatistics()`) to the same field, there is no lock (e.g. Reentrant Lock) that ensures mutually exclusive access to the field. This means that one thread could have execute the method `handleALL_SESSION_DATA()`, thereby writing and incrementing the interested field from – say – 0 to 1 and another thread may read the value of the field afterwards and still see its content as zero (asynchronous!). Hence, RacerD is correct in its assessment of line 1272 as a likely cause of data race.

Proposed Solution: One method to resolve this source of data race is to synchronize the methods that access the interested field (just like how the engineers have done with resetStatistics(). However, there is a classic and elegant solution to this error: using AtomicIntegers to ensure mutually exclusion and thread safety.

```
private AtomicInteger counterReceive_EVT_ALL_SESSION_DATA = new
    AtomicInteger(0);
// ...
public int getCounterReceive_EVT_ALL_SESSION_DATA() {
    return counterReceive_EVT_ALL_SESSION_DATA.get();
}
public synchronized void resetStatistics() {
    // ...
    counterReceive_EVT_ALL_SESSION_DATA.set(0); // bc it is synchronized, it
        need not use the atomic method but it doesn't hurt!
}
protected void handleALL_SESSION_DATA(SessionMessage msg,Member sender)
    throws ClassNotFoundException, IOException {
        // ...
        counterReceive_EVT_ALL_SESSION_DATA.getAndIncrement();
        // ...
}
```

I have implemented these changes and ran infer on the project. I have successfully verified that the error (potential data race) is resolved thanks to the proposed solution.

2. Indirect, unprotected write outside synchronization

Error Report:

```
#360
java/org/apache/jasper/JspC.java:1160: warning: Thread Safety Violation
  Unprotected write. Non-private method `JspC.generateWebMapping(...)`
  indirectly writes to field `clctxt.className` outside of synchronization.
 Reporting because this access may occur on a background thread.
1158.           }
1159.
1160. >         String className = clctxt.getServletClassName();
1161.           String packageName = clctxt.getServletPackageName();
1162.
```

In this report, RacerD has identified location L1160 in JspC.java as a potential cause of a data race during execution. The reason is that a thread which calls the method JspC.generateWebMapping(...) will be indirectly writing to the shared data field "clctxt.className' outside of synchronization.

To verify whether this error message raised by RacerD is a true error that threatens Thread Safety or a false positive, I examined the JspC.java script and the JspCompilationContext.java script (which defines the method getServletClassName())

Script/Lines of interest:

```
// JspC.java
public void generateWebMapping( String file, JspCompilationContext clctxt )
        throws IOException
    {
        // ...
        String className = clctxt.getServletClassName();
    }

// JspCompilationContext.java
private String className;
// ...
public String getServletClassName() {
    if (className != null) {
        return className;
    }

    if (isTagFile) {
        className = tagInfo.getTagClassName();
        int lastIndex = className.lastIndexOf('.');
        if (lastIndex != -1) {
            className = className.substring(lastIndex + 1);
        }
    } else {
        int iSep = jspUri.lastIndexOf('/') + 1;
        className = JspUtil.makeJavaIdentifier(jspUri.substring(iSep));
    }
    return className;
}
        // ...
}
```

My Verdict: As suspected, RacerD has once again successfully identified a potential breach to Thread Safety. As Infer has implied, the method call `generateWebMapping()` does 'indirectly write' to a shared field by calling the method call `getServletClassName()` which is defined in JspCompilationContext.java. This affected/interested field is 'className' which is declared as a String (no volatile annotation). Although not captured in the code block above, the shared private variable 'className' is read and written to in many related methods described in JspCompilationContext.java. These functions seem to be designed to run concurrently in a multi-thread environment as there is evidence of synchronization in other parts of these functions. As correctly noted by Infer, line 1160 of JspC.java may likely result in a non-synchronized write to 'className', thereby influencing and causing unexpected/undesired behaviors from other methods whose behaviors are determined by the value they read from 'className'. Therefore, I believe that RacerD is once again correct in its assessment of line 1160 as a potential cause of data race.
Proposed Solution: One solution that resolves the error message of Infer is to attach a volatile annotation to the 'className' variable (declaration).

```
// JspCompilationContext.java
private volatile String className;
```

I have implemented these changes and ran infer on the project. I have successfully verified that the error (potential data race) is resolved thanks to the proposed solution.

This simple, one-line solution indeed resolves the error from arising in successive builds/tests with Infer. However, given how critical 'className' is in the JspCompilationContext class and its many methods, it is a more robust solution to implement a lock for className such that reads and writes to the field is mutually exclusive and the content of the field is accurate at every moment throughout the execution (not demonstrated in this report for sake of concision).

3. Write to shared field outside synchronization (`Synchronized`-required)

Error Report:

```
#509
java/org/apache/catalina/session/DataSourceStore.java:139: warning: Thread
Safety Violation
  Unprotected write. Non-private method ‘DataSourceStore.getName()‘ writes
  to field ‘this.name‘ outside of synchronization.
 Reporting because another access to the same memory occurs on a background
 thread, although this access may not.
137.                     }
138.                 }
139. >               name = "/" + engineName + "/" + hostName + contextName;
140.             }
141.             return name;
```

This time, RacerD has identified location L139 in DataSourceStore.java as a loophole for data race during execution. The reason is that a thread which calls the method DataSourceStore.getName(...) will be writing to a shared data field "DataSourceStore.name' outside of synchronization (which may collide with the methods of other threads that also access the field, resulting in undesired behaviors).

Let us consider if this error message is a true error that breaches Thread Safety or a false positive by looking at DataSourceStore.java script.

Script/Lines of interest:

```
private String name = null;
// ...
public String getName() {
    if (name == null) {
        Container container = manager.getContext();
        String contextName = container.getName();
        if (!contextName.startsWith("/")) {
            contextName = "/" + contextName;
        }
        String hostName = "";
        String engineName = "";

        if (container.getParent() != null) {
            Container host = container.getParent();
```

```
            hostName = host.getName();
            if (host.getParent() != null) {
                engineName = host.getParent().getName();
            }
        }
        name = "/" + engineName + "/" + hostName + contextName;
    }
    return name;
```

My Verdict: RacerD has once again successfully proven itself by correctly identified a potential breach to Thread Safety. The threads' non-synchronized writing to the field can impact the behavior of other threads and the entire program in unfortunate ways (e.g. less efficient, unexpected behavior, etc...) – I will demonstrate one such scenario. Consider the situation in which two threads have both called the method getName() (for the same DataSourceStore class instance) at a moment during the execution where `name == null`. As the getName() method is not synchronized and the reading of and writing to the field 'name' is not synchronized, both threads will read the content of 'name' as null and will proceed to execute the successive instructions of the method. Without loss of generality, assume that the first thread executes `Container container = manager.getContext();` and thereby writes the container item to its own local variable. Then, before the second thread also executes the line, the value of manager.getContext() is changed due to a write operation of another thread (operating within manager). Hence, when the second thread executes the line, it saves a different container value in its local variable 'container'. Assume that both threads proceed through the successive if-branches and the first thread executes `name = "/" + engineName + "/" + hostName + contextName;` – thereby writing to the shared field `this.name`. However, before it can return the `this.name` which it wrote, the second thread may run `name = "/" + engineName + "/" + hostName + contextName;` itself and override the shared value of 'name', causing both threads to return the 'name' value that was generated later. This is an undesirable outcome of the method getName() as the code specifies in the beginning that if 'name' is not 'null', threads should just return the defined 'name'. In contrast, without the synchronization on 'name' we can observe a scenario in which a name is essentially discarded/overwritten. This is clearly a situation in which the execution of the program is not properly coordinated due to conflicting accesses to a shared variable. Therefore, I believe that RacerD is once again correct in its diagnosis.

Proposed Solution: A solution could be to wrap the method of getName() in synchronized. Then, for each instance of the DataSourceStore class, there will be a maximum of one thread executing getName() at any given moment. This will prevent the situation where a newly defined 'name' is immediately overwritten by another thread. If one wants to take a more conservative, paranoid approach, they can add a volatile annotation to the variable 'name'. This is not entirely neccessary if getName() is synchronized (as access to 'name' occurs only in this method) and the errors will be resolved without it. In contrast, the contrary (i.e. adding volatile annotation but not synchronized) is not going to be enough to resolve the data race as the previously described scenario can still occur even when the data field of 'name' is volatile.

```
private volatile String name = null; // optional
// ...
public synchronized String getName() {
```

```
    if (name == null) {
        Container container = manager.getContext();
        String contextName = container.getName();
        if (!contextName.startsWith("/")) {
            contextName = "/" + contextName;
        }
        String hostName = "";
        String engineName = "";

        if (container.getParent() != null) {
            Container host = container.getParent();
            hostName = host.getName();
            if (host.getParent() != null) {
                engineName = host.getParent().getName();
            }
        }
        name = "/" + engineName + "/" + hostName + contextName;
    }
    return name;
}
```

I have implemented these changes and have successfully verified that the error (potential data race) is resolved thanks to the proposed solution.

4. Classic Counter Problem

Error Report:

```
#384
java/org/apache/tomcat/websocket/DigestAuthenticator.java:71: warning: Thread
Safety Violation
  Unprotected write. Non-private method ‘DigestAuthenticator.getAuthorization(...)‘
  writes to field ‘this.nonceCount‘ outside of synchronization.
 Reporting because this access may occur on a background thread.
69.
70.                 cNonce = cnonceGenerator.nextLong();
71. >               nonceCount++;
72.             }
73.
```

RacerD has identified location L71 in DigestAuthenticator.java as a potential cause for data race during execution. The justification is that threads that call this method will be writing to the shared data field “this.nonceCount’ outside of synchronization.

Let us consider if this error message is a true error that breaches Thread Safety or a false positive by looking at DataSourceStore.java script.

Script/Lines of interest:

```
private int nonceCount = 0;
// ...
public String getAuthorization(String requestUri, String WWWAuthenticate,
```

```
    Map<String, Object> userProperties) throws AuthenticationException {
    // ...
    nonceCount++;
    // ...
    challenge.append("nc=" + String.format("%08X",
        Integer.valueOf(nonceCount)));
}
// ...
private String calculateRequestDigest(String requestUri, String userName,
    String password,
    String realm, String nonce, String qop, String algorithm)
    throws NoSuchAlgorithmException {
    // ...
    preDigest.append(String.format("%08X", Integer.valueOf(nonceCount)));
    // ...
}
```

My Verdict: RacerD has correctly identified a potential breach to Thread Safety, again. This is the classic asynchronous counter example. Consider a situation where two threads are both executing 'DigestAuthenticator.getAuthorization(...)'. Since the shared field 'nonceCount' is neither atomic or volatile, even if one thread executes the `nonceCount++` operation (incrementing the nonceCount value from 3 to 4, for e.g.), it is very likely that the second thread will read the same value that the previous thread read and essentially not increment the shared counter (e.g. reads 3 and increments to 4, again). This is not the behavior we expect from the execution of our program and is suggestive that there is a lack of synchronization between our threads – hence, I too agree with Infer's diagnosis that the non-synchronous execution of the line 'nonceCount++' (and the nature of nonceCount as a whole) is a threat against Thread Safety and predictor for data race. *Proposed Solution:* As before, I would chose to re-declare 'nonceCount' as an atomic integer such that we can ensure mutual exclusion in a non-blocking manner. This would ensure that every reading of the value of 'nonceCount' is accurate and representative, and this will allow the count to be precisely kept without nullified increments.

```
private AtomicInteger nonceCount = new AtomicInteger(0);
// ...
public String getAuthorization(String requestUri, String WWWAuthenticate,
    Map<String, Object> userProperties) throws AuthenticationException {
    // ...
    nonceCount.getAndIncrement();
    // ...
    challenge.append("nc=" + String.format("%08X", nonceCount.get()));
}
// ...
private String calculateRequestDigest(String requestUri, String userName,
    String password,
    String realm, String nonce, String qop, String algorithm)
    throws NoSuchAlgorithmException {
    // ...
    preDigest.append(String.format("%08X", nonceCount.get()));
    // ...
}
```

I have implemented these changes and have successfully verified that the error (potential data race) is resolved thanks to the proposed solution.

5. Bonus: Interesting Finding and Further Investigation of Atomic Data Types

Error Report:

```
#300
java/org/apache/coyote/http2/Http2UpgradeHandler.java:525: warning: Thread
Safety Violation
  Read/Write race. Non-private method ‘Http2UpgradeHandler.executeQueued-
  Stream()‘ reads without synchronization from ‘this.streamConcurrency‘.
  Potentially races with write in method ‘Http2UpgradeHandler.upgradeDispatch(...)‘.
 Reporting because another access to the same memory occurs on a
 background thread, although this access may not.
523.
524.       void executeQueuedStream() {
525. >         if (streamConcurrency == null) {
526.               return;
527.           }
```

More Script/Lines for reference:

```
private AtomicInteger streamConcurrency = null;
// ...
@Override
public void init(WebConnection webConnection) {
    // ...
    if (protocol.getMaxConcurrentStreamExecution() <
        localSettings.getMaxConcurrentStreams()) {
        streamConcurrency = new AtomicInteger(0);
        // ...
        }
    // ...
}
void processStreamOnContainerThread(StreamProcessor streamProcessor,
    SocketEvent event) {
    StreamRunnable streamRunnable = new StreamRunnable(streamProcessor, event);
    if (streamConcurrency == null) {
        socketWrapper.execute(streamRunnable);
    } else {
        if (getStreamConcurrency() <
            protocol.getMaxConcurrentStreamExecution()) {
            increaseStreamConcurrency();
            socketWrapper.execute(streamRunnable);
        } else {
            queuedRunnable.offer(streamRunnable);
        }
    }
}
```

```
private int increaseStreamConcurrency() {
    return streamConcurrency.incrementAndGet();
}

private int decreaseStreamConcurrency() {
    return streamConcurrency.decrementAndGet();
}

private int getStreamConcurrency() {
    return streamConcurrency.get();
}

void executeQueuedStream() {
    if (streamConcurrency == null) {
        return;
    }
    decreaseStreamConcurrency();
    if (getStreamConcurrency() < protocol.getMaxConcurrentStreamExecution()) {
        StreamRunnable streamRunnable = queuedRunnable.poll();
        if (streamRunnable != null) {
            increaseStreamConcurrency();
            socketWrapper.execute(streamRunnable);
        }
    }
}
```

Initially, I was surprised to see that the data type of 'streamConcurrency' – the reading of which is the source of the error – is actually an AtomicInteger (and not an Integer or long). Then, I wondered if RacerD may have finally given out a false positive. However, the more carefully I observed the methods and operations done on the AtomicInteger, the more questions I had: Is there a difference between calling an AtomicInteger variable like a regular variable (e.g. 'ABC') vs. calling it through its designated method (e.g. 'ABC.get()') other than the output type (atomic int vs. int)? Are we limited to using the provided, pre-defined atomic methods if we want to ensure atomicity and mutual exclusion? For instance, consider line 525 which is the line RacerD has pointed out. At first, I naively imagined that the comparison operation (i.e. '==') is atomic because it was written in one line. However, upon further consideration, I realized that there is at least three steps to a comparison: 1. reading the first value, 2. reading the second value, and 3. comparing the two. Hence, even if two atomic integers are compared using the '==', the operation may not be atomic as a whole. However, in our given line, we see that the right side of the equality comparison operator is a NULL which is an unchanging value – could this allow for the comparison to be atomic here?

Furthermore, I am intrigued by how the AtomicInteger type can house a 'NULL' (as demonstrated by the engineers above). I assume they are using it as a sentinel value, but it seems to come with non-trivial costs in mutual exclusion and synchronization (i.e. cannot 'NULL.set(...)') – what could be a better sentinel value for AtomicIntegers? AtomicInteger(MAX_INT)? AtomicInteger(MIN_INT)? Or maybe something else?