Fortify Standalone Report Generator

Developer Workbook

akka-persistence-tck

Table of Contents

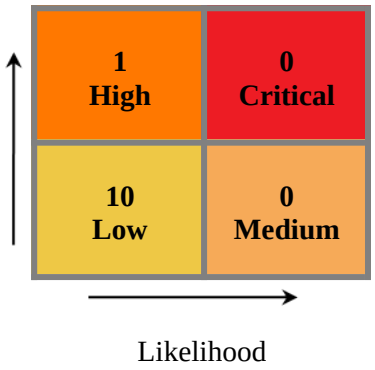
Executive Summary

This workbook is intended to provide all necessary details and information for a developer to understand and remediate the different issues discovered during the akka-persistence-tck project audit. The information contained in this workbook is targeted at project managers and developers.

This section provides an overview of the issues uncovered during analysis.

Project Name:	akka-persistence-tck
Project Version:	
SCA:	Results Present
WebInspect:	Results Not Present
WebInspect Agent:	Results Not Present
Other:	Results Not Present

Issues by Priority



Top Ten Critical Categories

This project does not contain any critical issues

Project Description

This section provides an overview of the Fortify scan engines used for this project, as well as the project meta-information.

SCA

Date of Last Analysis:	Jun 16, 2022, 11:36 AM	**Engine Version:**	21.1.1.0009
Host Name:	Jacks-Work-MBP.local	**Certification:**	VALID
Number of Files:	11	**Lines of Code:**	582

Rulepack Name	Rulepack Version
Fortify Secure Coding Rules, Extended, Java	2022.1.0.0007
Fortify Secure Coding Rules, Core, Scala	2022.1.0.0007
Fortify Secure Coding Rules, Extended, JSP	2022.1.0.0007
Fortify Secure Coding Rules, Core, Android	2022.1.0.0007
Fortify Secure Coding Rules, Extended, Content	2022.1.0.0007
Fortify Secure Coding Rules, Extended, Configuration	2022.1.0.0007
Fortify Secure Coding Rules, Core, Annotations	2022.1.0.0007
Fortify Secure Coding Rules, Community, Cloud	2022.1.0.0007
Fortify Secure Coding Rules, Core, Universal	2022.1.0.0007
Fortify Secure Coding Rules, Core, Java	2022.1.0.0007
Fortify Secure Coding Rules, Community, Universal	2022.1.0.0007

Issue Breakdown by Fortify Categories

The following table depicts a summary of all issues grouped vertically by Fortify Category. For each category, the total number of issues is shown by Fortify Priority Order, including information about the number of audited issues.

Category	Fortify Priority (audited/total)				Total Issues
	Critical	High	Medium	Low	
Code Correctness: Constructor Invokes Overridable Function	0	0	0	0 / 1	0 / 1
Code Correctness: Erroneous String Compare	0	0	0	0 / 1	0 / 1
Code Correctness: Non-Static Inner Class Implements Serializable	0	0	0	0 / 2	0 / 2
Dead Code: Expression is Always false	0	0	0	0 / 4	0 / 4
Dead Code: Expression is Always true	0	0	0	0 / 1	0 / 1
Password Management: Hardcoded Password	0	0 / 1	0	0	0 / 1
Poor Logging Practice: Use of a System Output Stream	0	0	0	0 / 1	0 / 1

Results Outline

Code Correctness: Constructor Invokes Overridable Function (1 issue)

Abstract

A constructor of the class calls a function that can be overridden.

Explanation

When a constructor calls an overridable function, it may allow an attacker to access the `this` reference prior to the object being fully initialized, which can in turn lead to a vulnerability. **Example 1:** The following calls a method that can be overridden.

```
...
class User {
  private String username;
  private boolean valid;
  public User(String username, String password){
    this.username = username;
    this.valid = validateUser(username, password);
  }
  public boolean validateUser(String username, String password){
    //validate user is real and can authenticate
    ...
  }
  public final boolean isValid(){
    return valid;
  }
}
```

Since the function `validateUser` and the class are not `final`, it means that they can be overridden, and then initializing a variable to the subclass that overrides this function would allow bypassing of the `validateUser` functionality. For example:

```
...
class Attacker extends User{
  public Attacker(String username, String password){
    super(username, password);
  }
  public boolean validateUser(String username, String password){
    return true;
  }
}
...
class MainClass{
  public static void main(String[] args){
    User hacker = new Attacker("Evil", "Hacker");
    if (hacker.isValid()){
      System.out.println("Attack successful!");
    }else{
      System.out.println("Attack failed");
    }
  }
}
```

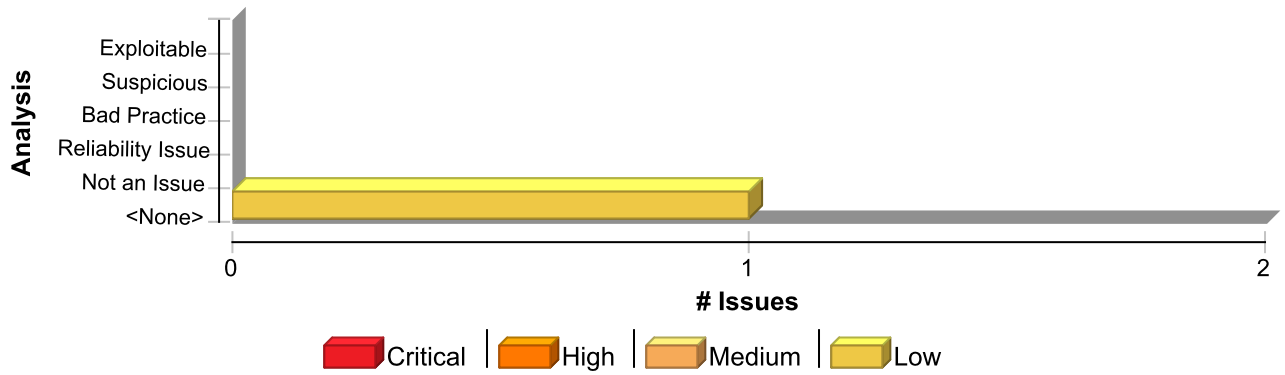
The code in `Example 1` prints "Attack successful!", since the `Attacker` class overrides the `validateUser()` function that is called from the constructor of the superclass `User`, and Java will first look in the subclass for functions called from the constructor.

Recommendation
Constructors should not call functions that can be overridden, either by specifying them as `final`, or specifying the class as `final`. Alternatively if this code is only ever needed in the constructor, the `private` access specifier can be used, or the logic could be placed directly into the constructor of the superclass. **Example 2:** The following makes the class `final` to prevent the function from being overridden elsewhere.

```
...
final class User {
  private String username;
  private boolean valid;
  public User(String username, String password){
    this.username = username;
    this.valid = validateUser(username, password);
  }
  private boolean validateUser(String username, String password){
    //validate user is real and can authenticate
    ...
  }
  public final boolean isValid(){
    return valid;
  }
}
```

This example specifies the class as `final`, so that it cannot be subclassed, and changes the `validateUser()` function to `private`, since it is not needed elsewhere in this application. This is programming defensively, since at a later date it may be decided that the `User` class needs to be subclassed, which would result in this vulnerability reappearing if the `validateUser()` function was not set to `private`.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Code Correctness: Constructor Invokes Overridable Function	1	0	0	1
Total	**1**	**0**	**0**	**1**

Code Correctness: Constructor Invokes Overridable Function	**Low**
Package: akka.persistence.journal	
journal/JournalPerfSpec.scala, line 126 (Code Correctness: Constructor Invokes Overridable Function)	**Low**
Issue Details	

Code Correctness: Constructor Invokes Overridable Function	Low
Package: akka.persistence.journal	
journal/JournalPerfSpec.scala, line 126 (Code Correctness: Constructor Invokes Overridable Function)	**Low**

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionCall: akka$persistence$journal$JournalPerfSpec$$cmdSerializerConfig
Enclosing Method: JournalPerfSpec()
File: journal/JournalPerfSpec.scala:126
Taint Flags:

```
123  * @see [[akka.persistence.journal.JournalSpec]]
124  */
125 abstract class JournalPerfSpec(config: Config)
126  extends JournalSpec(config.withFallback(JournalPerfSpec.cmdSerializerConfig)) {
127
128  private val testProbe = TestProbe()
129
```

Code Correctness: Erroneous String Compare (1 issue)

Abstract

Strings should be compared with the `equals()` method, not `==` or `!=`.

Explanation

This program uses `==` or `!=` to compare two strings for equality, which compares two objects for equality, not their values. Chances are good that the two references will never be equal. **Example 1:** The following branch will never be taken.

```
  if (args[0] == STRING_CONSTANT) {
      logger.info("miracle");
  }
```

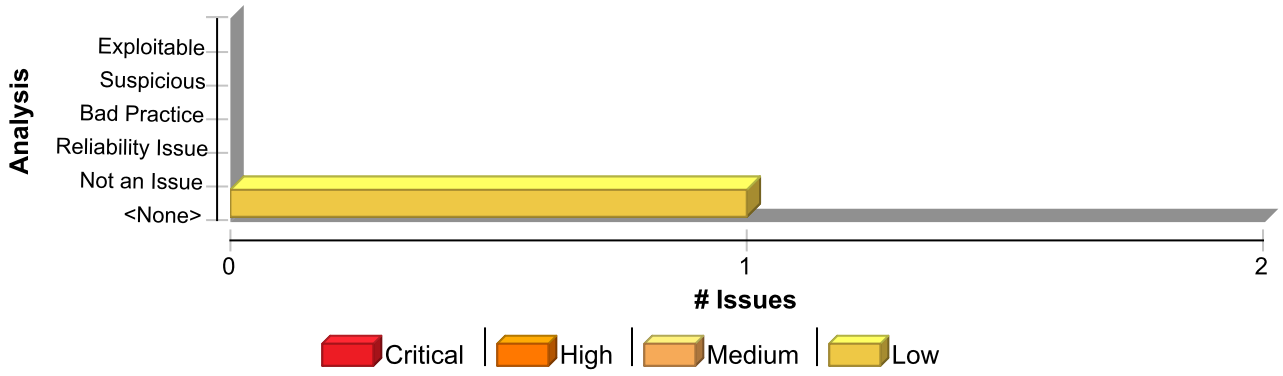
The `==` and `!=` operators will only behave as expected when they are used to compare strings contained in objects that are equal. The most common way for this to occur is for the strings to be interned, whereby the strings are added to a pool of objects maintained by the `String` class. Once a string is interned, all uses of that string will use the same object and equality operators will behave as expected. All string literals and string-valued constants are interned automatically. Other strings can be interned manually be calling `String.intern()`, which will return a canonical instance of the current string, creating one if necessary.

Recommendation

Use `equals()` to compare strings. **Example 2:** The code in `Example 1` could be rewritten in the following way:

```
  if (STRING_CONSTANT.equals(args[0])) {
      logger.info("could happen");
  }
```

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Code Correctness: Erroneous String Compare	1	0	0	1
Total	**1**	**0**	**0**	**1**

Code Correctness: Erroneous String Compare — Low

Package: akka.persistence

TestSerializer.scala, line 32 (Code Correctness: Erroneous String Compare) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: Operation
Enclosing Method: fromBinary()
File: TestSerializer.scala:32
Taint Flags:

```
29 }
30 def fromBinary(bytes: Array[Byte], manifest: String): AnyRef = {
31 verifyTransportInfo()
32 manifest match {
33 case "A" =>
34 val refStr = new String(bytes, StandardCharsets.UTF_8)
35 val ref = system.provider.resolveActorRef(refStr)
```

Code Correctness: Non-Static Inner Class Implements Serializable (2 issues)

Abstract

Inner classes implementing `java.io.Serializable` may cause problems and leak information from the outer class.

Explanation

Serialization of inner classes lead to serialization of the outer class, therefore possibly leaking information or leading to a runtime error if the outer class is not serializable. As well as this, serializing inner classes may cause platform dependencies since the Java compiler creates synthetic fields in order to implement inner classes, but these are implementation dependent, and may vary from compiler to compiler. **Example 1:** The following code allows serialization of an inner class.

```
...
class User implements Serializable {
  private int accessLevel;
  class Registrator implements Serializable {
    ...
  }
}
```

In `Example 1`, when the inner class `Registrator` is serialized, it will also serialize the field `accessLevel` from the outer class `User`.

Recommendation

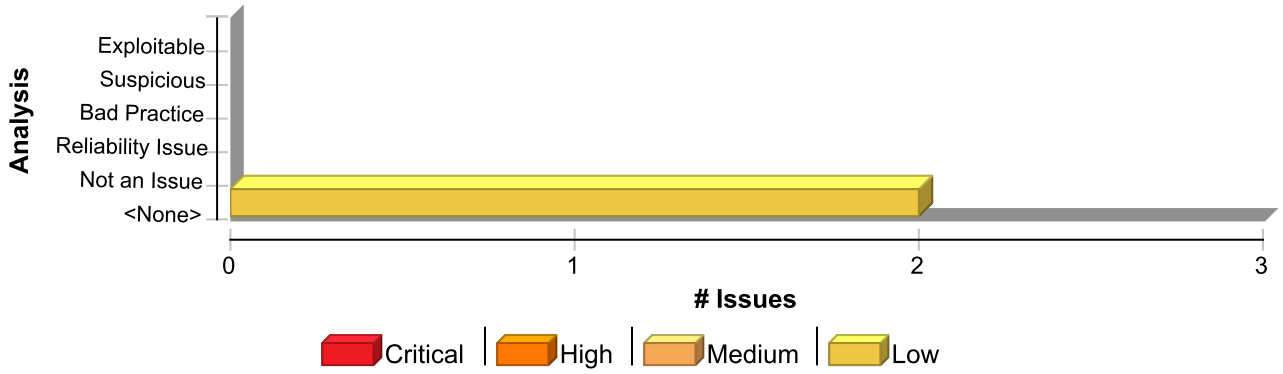
When using inner classes, they should not be serialized, or they should be changed to static-nested classes, since these do not have the drawbacks that non-static inner classes have when serialized. When a nested class is static it inherently has no association with instance variables (including those of the outer class), and would not cause serialization of the outer class. **Example 2:** The following code changes the example in `Example 1`, by stopping the inner class from implementing `java.io.Serializable`.

```
...
class User implements Serializable {
  private int accessLevel;
  class Registrator {
    ...
  }
}
```

Example 2: The following code changes the example in `Example 1`, by making the inner class into a static-nested class.

```
...
class User implements Serializable {
  private int accessLevel;
  static class Registrator implements Serializable {
    ...
  }
}
```

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Code Correctness: Non-Static Inner Class Implements Serializable	2	0	0	2
Total	**2**	**0**	**0**	**2**

Code Correctness: Non-Static Inner Class Implements Serializable — Low

Package: akka.persistence.journal

journal/JournalPerfSpec.scala, line 74 (Code Correctness: Non-Static Inner Class Implements Serializable) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: Class: JournalPerfSpec$Cmd
File: journal/JournalPerfSpec.scala:74
Taint Flags:

```
71 }
72
73 case object ResetCounter
74 case class Cmd(mode: String, payload: Int)
75
76 /**
77 * INTERNAL API
```

Package: akka.persistence.scalatest

scalatest/MayVerb.scala, line 62 (Code Correctness: Non-Static Inner Class Implements Serializable) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Code Correctness: Non-Static Inner Class Implements Serializable	Low
Package: akka.persistence.scalatest	
scalatest/MayVerb.scala, line 62 (Code Correctness: Non-Static Inner Class Implements Serializable)	**Low**

Sink: Class: MayVerb$TestCanceledByFailure
File: scalatest/MayVerb.scala:62
Taint Flags:

```
59 }
60
61 object MayVerb {
62  case class TestCanceledByFailure(msg: String, specialStackTrace: Array[StackTraceElement])
63  extends TestCanceledException(Some(msg), None, 2) {
64  override def getStackTrace = specialStackTrace
65  }
```

Dead Code: Expression is Always false (4 issues)

Abstract

This expression will always evaluate to `false`.

Explanation

This expression will always evaluate to `false`; the program could be rewritten in a simpler form. The nearby code may be present for debugging purposes, or it may not have been maintained along with the rest of the program. The expression may also be indicative of a bug earlier in the method. **Example 1:** The following method never sets the variable `secondCall` after initializing it to `false`. (The variable `firstCall` is mistakenly used twice.) The result is that the expression `firstCall && secondCall` will always evaluate to `false`, so `setUpDualCall()` will never be invoked.

```
public void setUpCalls() {
  boolean firstCall = false;
  boolean secondCall = false;

  if (fCall > 0) {
    setUpFCall();
    firstCall = true;
  }
  if (sCall > 0) {
    setUpSCall();
    firstCall = true;
  }

  if (firstCall && secondCall) {
    setUpDualCall();
  }
}
```

Example 2: The following method never sets the variable `firstCall` to `true`. (The variable `firstCall` is mistakenly set to `false` after the first conditional statement.) The result is that the first part of the expression `firstCall && secondCall` will always evaluate to `false`.

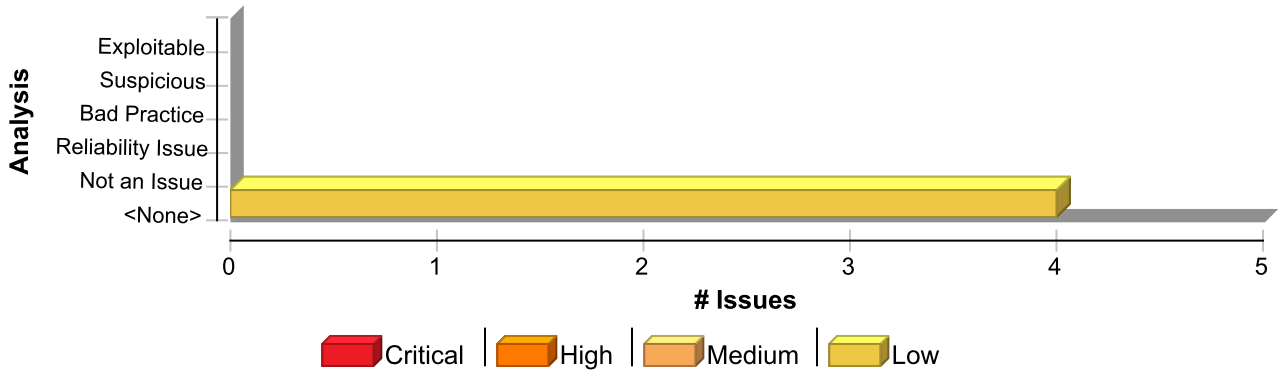
```
public void setUpCalls() {
  boolean firstCall = false;
  boolean secondCall = false;

  if (fCall > 0) {
    setUpFCall();
    firstCall = false;
  }
  if (sCall > 0) {
    setUpSCall();
    secondCall = true;
  }

  if (firstCall && secondCall) {
    setUpForCall();
  }
}
```

Recommendation

In general, you should repair or remove unused code. It causes additional complexity and maintenance burden without

contributing to the functionality of the program.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Dead Code: Expression is Always false	4	0	0	4
Total	**4**	**0**	**0**	**4**

Dead Code: Expression is Always false — Low

Package: akka.persistence.journal

journal/JournalPerfSpec.scala, line 55 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()
File: journal/JournalPerfSpec.scala:55
Taint Flags:

```
52  }
53  if (counter == replyAfter) replyTo ! payload
54
55  case Cmd("n", payload) =>
56  counter += 1
57  require(payload == counter, s"Expected to receive [$counter] yet got: [${payload}]")
58  if (counter == replyAfter) replyTo ! payload
```

journal/JournalPerfSpec.scala, line 41 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Dead Code: Expression is Always false — Low

Package: akka.persistence.journal

journal/JournalPerfSpec.scala, line 41 (Dead Code: Expression is Always false) — Low

Sink: IfStatement
Enclosing Method: applyOrElse()
File: journal/JournalPerfSpec.scala:41
Taint Flags:

```
38 if (counter == replyAfter) replyTo ! d.payload
39 }
40
41 case c @ Cmd("pa", _) =>
42 persistAsync(c) { d =>
43 counter += 1
44 require(d.payload == counter, s"Expected to receive [$counter] yet got: [${d.payload}]")
```

journal/JournalPerfSpec.scala, line 48 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()
File: journal/JournalPerfSpec.scala:48
Taint Flags:

```
45 if (counter == replyAfter) replyTo ! d.payload
46 }
47
48 case c @ Cmd("par", payload) =>
49 counter += 1
50 persistAsync(c) { d =>
51 require(d.payload == counter, s"Expected to receive [$counter] yet got: [${d.payload}]")
```

journal/JournalPerfSpec.scala, line 34 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()
File: journal/JournalPerfSpec.scala:34
Taint Flags:

```
31 var counter = 0
32
```

Dead Code: Expression is Always false	Low
Package: akka.persistence.journal	
journal/JournalPerfSpec.scala, line 34 (Dead Code: Expression is Always false)	**Low**

```
33 override def receiveCommand: Receive = {
34 case c @ Cmd("p", _) =>
35 persist(c) { d =>
36 counter += 1
37 require(d.payload == counter, s"Expected to receive [$counter] yet got: [${d.payload}]")
```

Dead Code: Expression is Always true (1 issue)

Abstract

This expression will always evaluate to `true`.

Explanation

This expression will always evaluate to `true`; the program could be rewritten in a simpler form. The nearby code may be present for debugging purposes, or it may not have been maintained along with the rest of the program. The expression may also be indicative of a bug earlier in the method. **Example 1:** The following method never sets the variable `secondCall` after initializing it to `true`. (The variable `firstCall` is mistakenly used twice.) The result is that the expression `firstCall || secondCall` will always evaluate to `true`, so `setUpForCall()` will always be invoked.

```
public void setUpCalls() {
  boolean firstCall = true;
  boolean secondCall = true;

  if (fCall < 0) {
    cancelFCall();
    firstCall = false;
  }
  if (sCall < 0) {
    cancelSCall();
    firstCall = false;
  }

  if (firstCall || secondCall) {
    setUpForCall();
  }
}
```

Example 2: The following method tries to check the variables `firstCall` and `secondCall`. (The variable `firstCall` is mistakenly set to `true` instead of being checked.) The result is that the first part of the expression `firstCall = true && secondCall == true` will always evaluate to `true`.

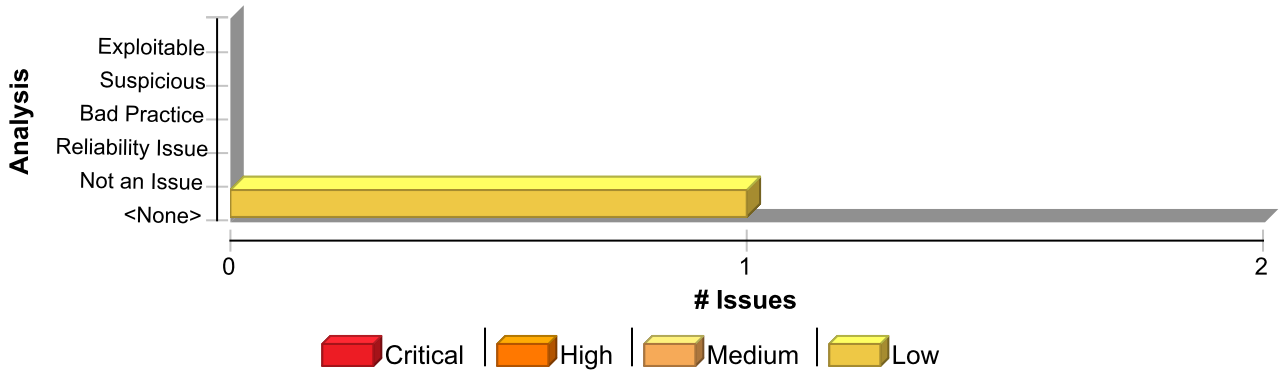
```
public void setUpCalls() {
  boolean firstCall = false;
  boolean secondCall = false;

  if (fCall > 0) {
    setUpFCall();
    firstCall = true;
  }
  if (sCall > 0) {
    setUpSCall();
    secondCall = true;
  }

  if (firstCall = true && secondCall == true) {
    setUpDualCall();
  }
}
```

Recommendation

In general, you should repair or remove unused code. It causes additional complexity and maintenance burden without

contributing to the functionality of the program.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Dead Code: Expression is Always true	1	0	0	1
Total	**1**	**0**	**0**	**1**

Dead Code: Expression is Always true — Low

Package: akka.persistence.journal

journal/JournalSpec.scala, line 97 (Dead Code: Expression is Always true) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: writeMessages()
File: journal/JournalSpec.scala:97
Taint Flags:

```
94  writerUuid = writerUuid)
95
96  val msgs =
97  if (supportsAtomicPersistAllOfSeveralEvents) {
98  (fromSnr to toSnr - 1).map { i =>
99  if (i == toSnr - 1)
100  AtomicWrite(List(persistentRepr(i), persistentRepr(i + 1)))
```

Password Management: Hardcoded Password (1 issue)

Abstract

Hardcoded passwords can compromise system security in a way that is not easy to remedy.

Explanation

It is never a good idea to hardcode a password. Not only does hardcoding a password allow all of the project's developers to view the password, it also makes fixing the problem extremely difficult. After the code is in production, the password cannot be changed without patching the software. If the account protected by the password is compromised, the owners of the system must choose between security and availability. **Example 1:** The following code uses a hardcoded password to connect to a database:

```
...
DriverManager.getConnection(url, "scott", "tiger");
...
```

This code will run successfully, but anyone who has access to it will have access to the password. After the program ships, there is likely no way to change the database user "scott" with a password of "tiger" unless the program is patched. An employee with access to this information can use it to break into the system. Even worse, if attackers have access to the bytecode for the application they can use the `javap -c` command to access the disassembled code, which will contain the values of the passwords used. The result of this operation might look something like the following for `Example 1`:

```
javap -c ConnMngr.class

22: ldc   #36; //String jdbc:mysql://ixne.com/rxsql
24: ldc   #38; //String scott
26: ldc   #17; //String tiger
```

In the mobile environment, password management is especially important given that there is such a high chance of device loss. **Example 2:** The following code uses hardcoded username and password to setup authentication for viewing protected pages with Android's WebView.

```
...
webview.setWebViewClient(new WebViewClient() {
  public void onReceivedHttpAuthRequest(WebView view,
        HttpAuthHandler handler, String host, String realm) {
    handler.proceed("guest", "allow");
  }
});
...
```

Similar to `Example 1`, this code will run successfully, but anyone who has access to it will have access to the password.

Recommendation

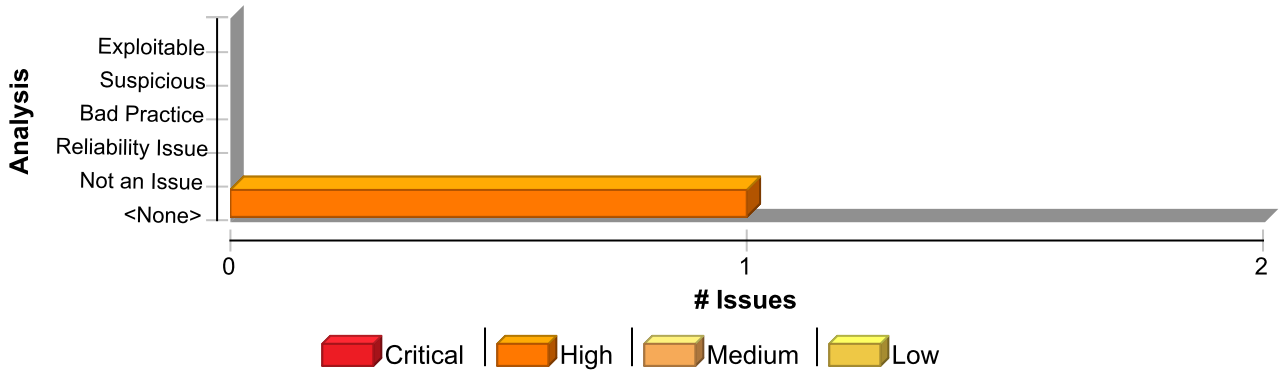
Passwords should never be hardcoded and should generally be obfuscated and managed in an external source. Storing passwords in plain text anywhere on the system allows anyone with sufficient permissions to read and potentially misuse the password. At the very least, hash passwords before storing them. Some third-party products claim the ability to securely manage passwords. For example, WebSphere Application Server 4.x uses a simple XOR encryption algorithm for obfuscating values, but be skeptical about such facilities. WebSphere and other application servers offer outdated and relatively weak encryption mechanisms that are insufficient for security-sensitive environments. Today, the best option for a secure generic solution is to create a proprietary mechanism yourself. For Android, as well as any other platform that uses SQLite database, SQLCipher is a good alternative. SQLCipher is an extension to the SQLite database that provides transparent 256-bit AES encryption of database files. Thus, credentials can be stored in an encrypted database. **Example 3:** The following code demonstrates how to integrate SQLCipher into an Android application after downloading the necessary binaries, and store credentials into the database file.

```
import net.sqlcipher.database.SQLiteDatabase;
```

```
...
  SQLiteDatabase.loadLibs(this);
  File dbFile = getDatabasePath("credentials.db");
  dbFile.mkdirs();
  dbFile.delete();
  SQLiteDatabase db = SQLiteDatabase.openOrCreateDatabase(dbFile,
"credentials", null);
  db.execSQL("create table credentials(u, p)");
  db.execSQL("insert into credentials(u, p) values(?, ?)", new Object[]
{username, password});
...
```

Note that references to `android.database.sqlite.SQLiteDatabase` are substituted with those of `net.sqlcipher.database.SQLiteDatabase`. To enable encryption on the WebView store, you must recompile WebKit with the `sqlcipher.so` library.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Password Management: Hardcoded Password	1	0	0	1
Total	**1**	**0**	**0**	**1**

Password Management: Hardcoded Password	**High**
Package: snapshot	
snapshot/SnapshotStoreSpec.scala, line 118 (Password Management: Hardcoded Password)	**High**

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Structural)

Sink Details

Sink: StringLiteral
Enclosing Method: apply()
File: snapshot/SnapshotStoreSpec.scala:118
Taint Flags:

```
115 senderProbe.ref)
116 senderProbe.expectMsg(LoadSnapshotResult(Some(SelectedSnapshot(metadata(2), s"s-3")), 13))
```

Password Management: Hardcoded Password	High

Package: snapshot

snapshot/SnapshotStoreSpec.scala, line 118 (Password Management: Hardcoded Password)	High

```
117  }
118  "delete a single snapshot identified by sequenceNr in snapshot metadata" in {
119  val md = metadata(2).copy(timestamp = 0L) // don't care about timestamp for delete of single snap
120  val cmd = DeleteSnapshot(md)
121  val sub = TestProbe()
```

Poor Logging Practice: Use of a System Output Stream (1 issue)

Abstract

Using `System.out` or `System.err` rather than a dedicated logging facility makes it difficult to monitor the program behavior.

Explanation

Example 1: The first Java program that a developer learns to write is the following:

```
public class MyClass
  ...
    System.out.println("hello world");
  ...
}
```

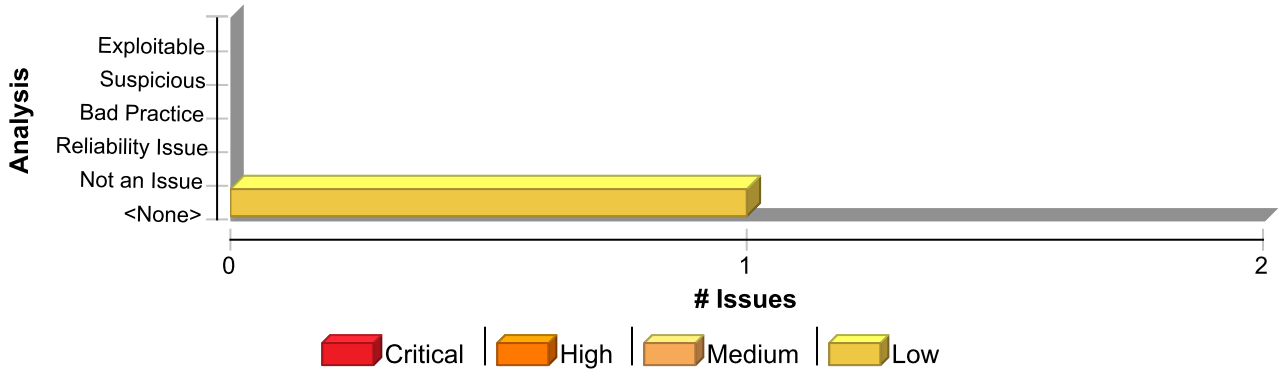
While most programmers go on to learn many nuances and subtleties about Java, a surprising number hang on to this first lesson and never give up on writing messages to standard output using `System.out.println()`. The problem is that writing directly to standard output or standard error is often used as an unstructured form of logging. Structured logging facilities provide features like logging levels, uniform formatting, a logger identifier, timestamps, and, perhaps most critically, the ability to direct the log messages to the right place. When the use of system output streams is jumbled together with the code that uses loggers properly, the result is often a well-kept log that is missing critical information. Developers widely accept the need for structured logging, but many continue to use system output streams in their "pre-production" development. If the code you are reviewing is past the initial phases of development, use of `System.out` or `System.err` may indicate an oversight in the move to a structured logging system.

Recommendation

Use a Java logging facility rather than `System.out` or `System.err`. **Example 2:** For example, you can rewrite the "hello world" program in `Example 1` using log4j as follows:

```
import org.apache.log4j.Logger;
import org.apache.log4j.BasicConfigurator;

public class MyClass {
  private final static Logger logger =
            Logger.getLogger(MyClass.class);

  ...
    BasicConfigurator.configure();
    logger.info("hello world");
  ...
}
```

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Poor Logging Practice: Use of a System Output Stream	1	0	0	1
Total	**1**	**0**	**0**	**1**

Poor Logging Practice: Use of a System Output Stream — Low

Package: akka.persistence.japi.journal

japi/journal/JavaJournalPerfSpec.scala, line 50 (Poor Logging Practice: Use of a System Output Stream) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionCall: println
Enclosing Method: apply()
File: japi/journal/JavaJournalPerfSpec.scala:50
Taint Flags:

```
47 class JavaJournalPerfSpec(config: Config) extends JournalPerfSpec(config) {
48  override protected def info: Informer = new Informer {
49  override def apply(message: String, payload: Option[Any])(implicit pos: Position): Unit =
50  System.out.println(message)
51  }
52
53  override protected def supportsRejectingNonSerializableObjects: CapabilityFlag = CapabilityFlag.on()
```