Fortify Standalone Report Generator

Developer Workbook

akka-testkit

Table of Contents

Executive Summary
Project Description
Issue Breakdown by Fortify Categories
Results Outline

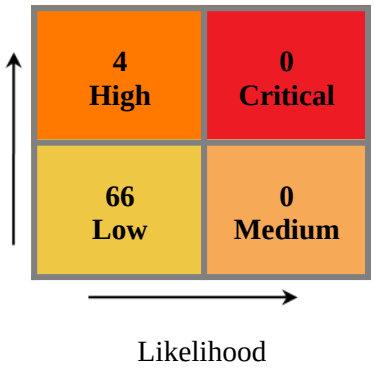
Executive Summary

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-testkit 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-testkit
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, 12:06 PM	**Engine Version:**	21.1.1.0009
Host Name:	Jacks-Work-MBP.local	**Certification:**	VALID
Number of Files:	44	**Lines of Code:**	2,461

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	
Access Specifier Manipulation	0	0 / 1	0	0	0 / 1
Code Correctness: Call to System.gc()	0	0	0	0 / 1	0 / 1
Code Correctness: Constructor Invokes Overridable Function	0	0	0	0 / 6	0 / 6
Code Correctness: Non-Static Inner Class Implements Serializable	0	0	0	0 / 9	0 / 9
Dead Code: Expression is Always false	0	0	0	0 / 16	0 / 16
Dead Code: Expression is Always true	0	0	0	0 / 4	0 / 4
Insecure Randomness	0	0 / 2	0	0	0 / 2
J2EE Bad Practices: Sockets	0	0	0	0 / 6	0 / 6
J2EE Bad Practices: Threads	0	0	0	0 / 16	0 / 16
Poor Error Handling: Overly Broad Catch	0	0	0	0 / 2	0 / 2
Poor Style: Value Never Read	0	0	0	0 / 1	0 / 1
Redundant Null Check	0	0	0	0 / 1	0 / 1
System Information Leak	0	0	0	0 / 1	0 / 1
System Information Leak: Internal	0	0	0	0 / 2	0 / 2
Unchecked Return Value	0	0	0	0 / 1	0 / 1
Unreleased Resource: Synchronization	0	0 / 1	0	0	0 / 1

Results Outline

Access Specifier Manipulation (1 issue)

Abstract

The method call changes an access specifier.

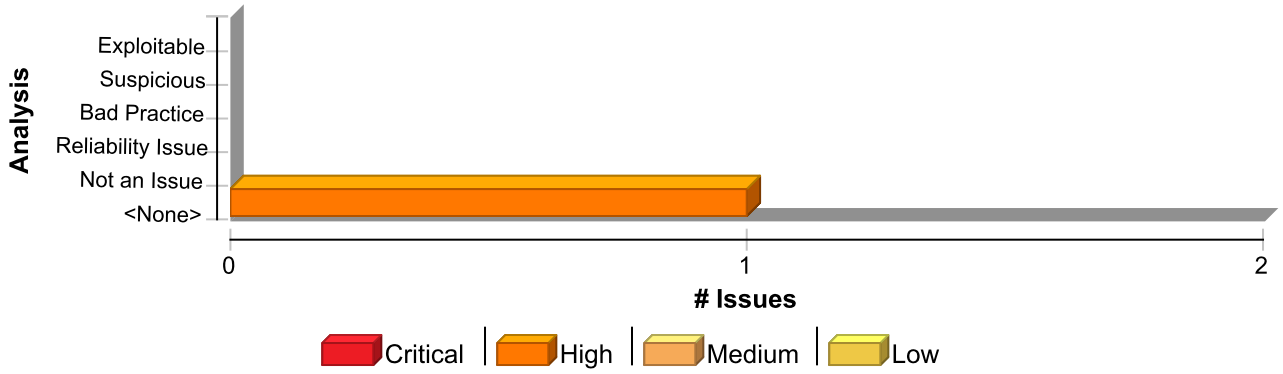
Explanation

The AccessibleObject API allows the programmer to get around the access control checks provided by Java access specifiers. In particular it enables the programmer to allow a reflected object to bypass Java access controls and in turn change the value of private fields or invoke private methods, behaviors that are normally disallowed.

Recommendation

Access specifiers should only be changed by a privileged class using arguments that an attacker cannot set. All occurrences should be examined carefully.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Access Specifier Manipulation	1	0	0	1
Total	**1**	**0**	**0**	**1**

Access Specifier Manipulation	**High**
Package: akka.testkit.metrics	
test/scala/akka/testkit/metrics/FileDescriptorMetricSet.scala, line 32 (Access Specifier Manipulation)	**High**

Issue Details

Kingdom: Input Validation and Representation
Scan Engine: SCA (Semantic)

Sink Details

Sink: setAccessible()

Access Specifier Manipulation	High
Package: akka.testkit.metrics	
test/scala/akka/testkit/metrics/FileDescriptorMetricSet.scala, line 32 (Access Specifier Manipulation)	**High**

Enclosing Method: akka$testkit$metrics$FileDescriptorMetricSet$$invoke()
File: test/scala/akka/testkit/metrics/FileDescriptorMetricSet.scala:32
Taint Flags:

```
29
30 private def invoke(name: String): Long = {
31 val method = os.getClass.getDeclaredMethod(name)
32 method.setAccessible(true)
33 method.invoke(os).asInstanceOf[Long]
34 }
35 }
```

Code Correctness: Call to System.gc() (1 issue)

Abstract

Explicit requests for garbage collection are a bellwether indicating likely performance problems.

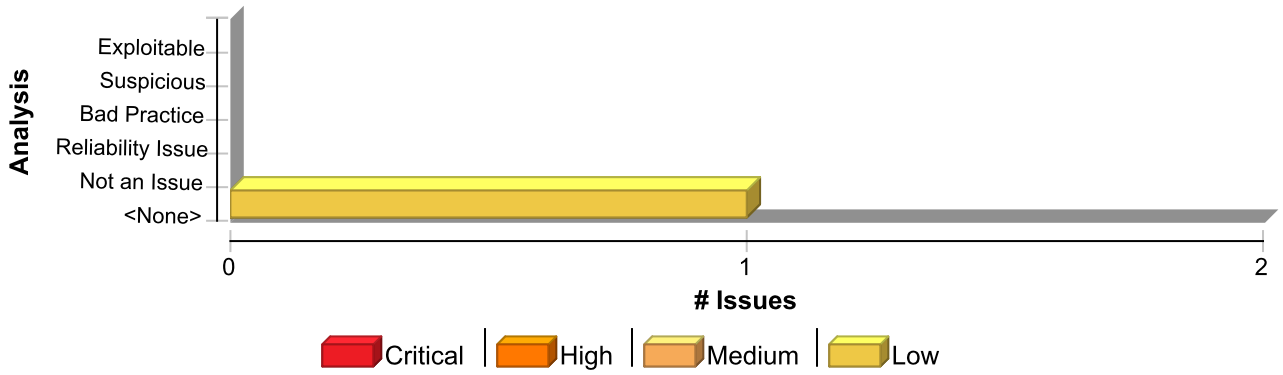
Explanation

At some point in every Java developer's career, a problem surfaces that appears to be so mysterious, impenetrable, and impervious to debugging that there seems to be no alternative but to blame the garbage collector. Especially when the bug is related to time and state, there may be a hint of empirical evidence to support this theory: inserting a call to `System.gc()` sometimes seems to make the problem go away. In almost every case we have seen, calling `System.gc()` is the wrong thing to do. In fact, calling `System.gc()` can cause performance problems if it is invoked too often.

Recommendation

When it seems as though calling `System.gc()` has solved a problem, look for other explanations, particularly ones that involve time and interaction between threads, processes, or the JVM and the operating system. I/O buffering, synchronization, and race conditions are all likely culprits.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Code Correctness: Call to System.gc()	1	0	0	1
Total	**1**	**0**	**0**	**1**

Code Correctness: Call to System.gc()	**Low**
Package: akka.testkit.metrics	
test/scala/akka/testkit/metrics/MetricsKitOps.scala, line 72 (Code Correctness: Call to System.gc())	**Low**

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Structural)

Sink Details

Code Correctness: Call to System.gc()	Low
Package: akka.testkit.metrics	
test/scala/akka/testkit/metrics/MetricsKitOps.scala, line 72 (Code Correctness: Call to System.gc())	**Low**

Sink: FunctionCall: gc
Enclosing Method: gc()
File: test/scala/akka/testkit/metrics/MetricsKitOps.scala:72
Taint Flags:

```
69 /** Yet another delegate to `System.gc()` */
70 def gc(): Unit = {
71 if (forceGcEnabled)
72 System.gc()
73 }
74
75 /**
```

Code Correctness: Constructor Invokes Overridable Function (6 issues)

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.

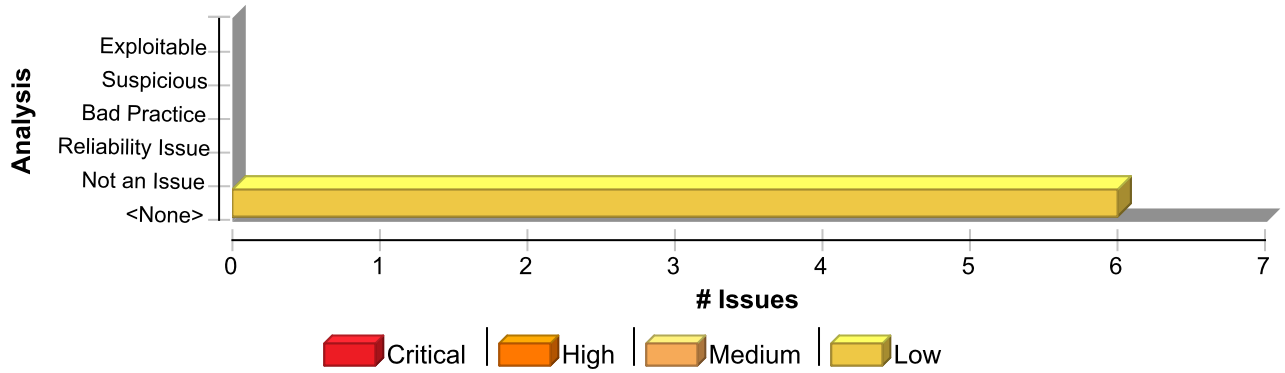
Recommendation

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	6	0	0	6
Total	**6**	**0**	**0**	**6**

Code Correctness: Constructor Invokes Overridable Function	**Low**
Package: akka.testkit	
main/scala/akka/testkit/CallingThreadDispatcher.scala, line 359 (Code Correctness: Constructor Invokes Overridable Function)	**Low**
Issue Details	

Code Correctness: Constructor Invokes Overridable Function — Low

Package: akka.testkit

main/scala/akka/testkit/CallingThreadDispatcher.scala, line 359 (Code Correctness: Constructor Invokes Overridable Function) — Low

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionCall: q
Enclosing Method: CallingThreadMailbox()
File: main/scala/akka/testkit/CallingThreadDispatcher.scala:359
Taint Flags:

```
356 * This is only a marker to be put in the messageQueue's stead to make error
357 * messages pertaining to violated mailbox type requirements less cryptic.
358 */
359 override val messageQueue: MessageQueue = q.get
360
361 override def enqueue(receiver: ActorRef, msg: Envelope): Unit = q.get.enqueue(receiver, msg)
362 override def dequeue(): Envelope =
```

test/scala/akka/testkit/AkkaSpec.scala, line 71 (Code Correctness: Constructor Invokes Overridable Function) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionCall: mapToConfig
Enclosing Method: AkkaSpec()
File: test/scala/akka/testkit/AkkaSpec.scala:71
Taint Flags:

```
68
69 def this(s: String) = this(ConfigFactory.parseString(s))
70
71 def this(configMap: Map[String, _]) = this(AkkaSpec.mapToConfig(configMap))
72
73 def this() = this(ActorSystem(TestKitUtils.testNameFromCallStack(classOf[AkkaSpec], "".r), AkkaSpec.testConf))
74
```

main/scala/akka/testkit/TestActorRef.scala, line 71 (Code Correctness: Constructor Invokes Overridable Function) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Code Correctness: Constructor Invokes Overridable Function — Low

Package: akka.testkit

main/scala/akka/testkit/TestActorRef.scala, line 71 (Code Correctness: Constructor Invokes Overridable Function) — Low

Sink Details

Sink: FunctionCall: props
Enclosing Method: TestActorRef()
File: main/scala/akka/testkit/TestActorRef.scala:71
Taint Flags:

```
68 if (_props.deploy.dispatcher == Deploy.NoDispatcherGiven) CallingThreadDispatcher.Id
69 else _props.dispatcher)
70
71 val dispatcher = _system.dispatchers.lookup(props.dispatcher)
72
73 // we need to start ourselves since the creation of an actor has been split into initialization and starting
74 underlying.start()
```

test/scala/akka/testkit/AkkaSpec.scala, line 64 (Code Correctness: Constructor Invokes Overridable Function) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionCall: testConf
Enclosing Method: AkkaSpec()
File: test/scala/akka/testkit/AkkaSpec.scala:64
Taint Flags:

```
61 with ScalaFutures {
62
63 def this(config: Config) =
64 this(
65 ActorSystem(
66 TestKitUtils.testNameFromCallStack(classOf[AkkaSpec], "".r),
67 ConfigFactory.load(config.withFallback(AkkaSpec.testConf))))
```

main/scala/akka/testkit/SocketUtil.scala, line 22 (Code Correctness: Constructor Invokes Overridable Function) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionCall: liftedTree1
Enclosing Method: SocketUtil()

Code Correctness: Constructor Invokes Overridable Function — Low

Package: akka.testkit

main/scala/akka/testkit/SocketUtil.scala, line 22 (Code Correctness: Constructor Invokes Overridable Function) — Low

File: main/scala/akka/testkit/SocketUtil.scala:22
Taint Flags:

```
19
20 val RANDOM_LOOPBACK_ADDRESS = "RANDOM_LOOPBACK_ADDRESS"
21
22 private val canBindOnAlternativeLoopbackAddresses = {
23 try {
24 SocketUtil.temporaryServerAddress(address = "127.20.0.0")
25 true
```

test/scala/akka/testkit/AkkaSpec.scala, line 73 (Code Correctness: Constructor Invokes Overridable Function) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionCall: testConf
Enclosing Method: AkkaSpec()
File: test/scala/akka/testkit/AkkaSpec.scala:73
Taint Flags:

```
70
71 def this(configMap: Map[String, _]) = this(AkkaSpec.mapToConfig(configMap))
72
73 def this() = this(ActorSystem(TestKitUtils.testNameFromCallStack(classOf[AkkaSpec], "".r), AkkaSpec.testConf))
74
75 implicit val patience: PatienceConfig =
76 PatienceConfig(testKitSettings.SingleExpectDefaultTimeout.dilated, Span(100, Millis))
```

Code Correctness: Non-Static Inner Class Implements Serializable (9 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

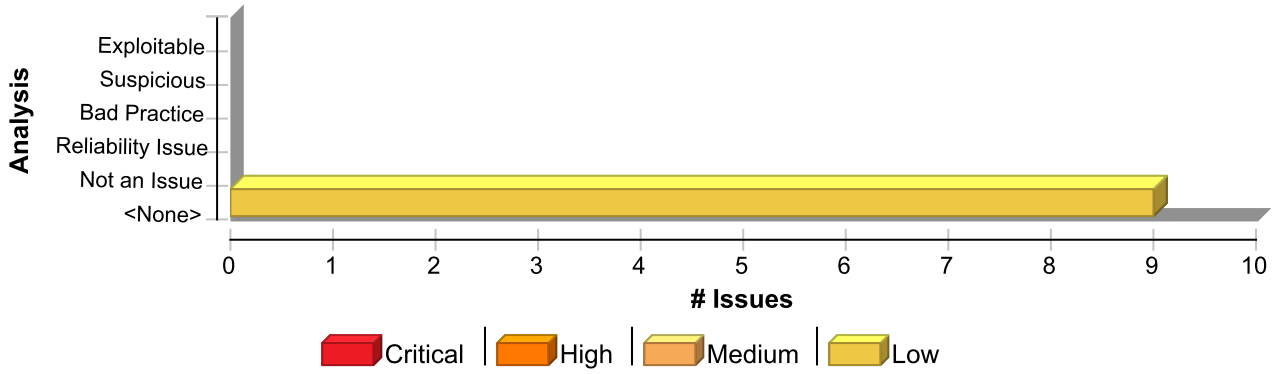
When using inner classes, they should not be serialized, or they should be changed to static-nested classes, since these do not have the drawbacks that non-static inner classes have when serialized. When a nested class is static it inherently has no association with instance variables (including those 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	9	0	0	9
Total	**9**	**0**	**0**	**9**

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

Package: akka.testkit

main/scala/akka/testkit/TestKit.scala, line 47 (Code Correctness: Non-Static Inner Class Implements Serializable) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: Class: TestActor$UnWatch
File: main/scala/akka/testkit/TestKit.scala:47
Taint Flags:

```
44
45 final case class SetIgnore(i: Ignore) extends NoSerializationVerificationNeeded
46 final case class Watch(ref: ActorRef) extends NoSerializationVerificationNeeded
47 final case class UnWatch(ref: ActorRef) extends NoSerializationVerificationNeeded
48 final case class SetAutoPilot(ap: AutoPilot) extends NoSerializationVerificationNeeded
49 final case class Spawn(props: Props, name: Option[String] = None, strategy: Option[SupervisorStrategy] = None)
50 extends NoSerializationVerificationNeeded {
```

main/scala/akka/testkit/TestKit.scala, line 48 (Code Correctness: Non-Static Inner Class Implements Serializable) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: Class: TestActor$SetAutoPilot

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

Package: akka.testkit

main/scala/akka/testkit/TestKit.scala, line 48 (Code Correctness: Non-Static Inner Class Implements Serializable) — Low

File: main/scala/akka/testkit/TestKit.scala:48
Taint Flags:

```
45 final case class SetIgnore(i: Ignore) extends NoSerializationVerificationNeeded
46 final case class Watch(ref: ActorRef) extends NoSerializationVerificationNeeded
47 final case class UnWatch(ref: ActorRef) extends NoSerializationVerificationNeeded
48 final case class SetAutoPilot(ap: AutoPilot) extends NoSerializationVerificationNeeded
49 final case class Spawn(props: Props, name: Option[String] = None, strategy: Option[SupervisorStrategy] = None)
50 extends NoSerializationVerificationNeeded {
51 def apply(context: ActorRefFactory): ActorRef = name match {
```

test/scala/akka/testkit/TestActorRefSpec.scala, line 107 (Code Correctness: Non-Static Inner Class Implements Serializable) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: Class: TestActorRefSpec$WrappedTerminated
File: test/scala/akka/testkit/TestActorRefSpec.scala:107
Taint Flags:

```
104 * Forwarding `Terminated` to non-watching testActor is not possible,
105 * and therefore the `Terminated` message is wrapped.
106 */
107 final case class WrappedTerminated(t: Terminated)
108
109 }
110
```

test/scala/akka/testkit/TestEventListenerSpec.scala, line 67 (Code Correctness: Non-Static Inner Class Implements Serializable) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: Class: TestEventListenerSpec$AnError
File: test/scala/akka/testkit/TestEventListenerSpec.scala:67
Taint Flags:

```
64 }
65 }
66
```

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

Package: akka.testkit

test/scala/akka/testkit/TestEventListenerSpec.scala, line 67 (Code Correctness: Non-Static Inner Class Implements Serializable) — Low

```
67 private class AnError extends Exception
68 private def errorNoCause = Error(self.path.toString, this.getClass, "this is an error")
69 private def errorWithCause(cause: Throwable) = Error(cause, self.path.toString, this.getClass, "this is an error")
70
```

main/scala/akka/testkit/TestKit.scala, line 61 (Code Correctness: Non-Static Inner Class Implements Serializable) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: Class: TestActor$RealMessage
File: main/scala/akka/testkit/TestKit.scala:61
Taint Flags:

```
58 def msg: AnyRef
59 def sender: ActorRef
60 }
61 final case class RealMessage(msg: AnyRef, sender: ActorRef) extends Message
62 case object NullMessage extends Message {
63 override def msg: AnyRef = throw IllegalActorStateException("last receive did not dequeue a message")
64 override def sender: ActorRef = throw IllegalActorStateException("last receive did not dequeue a message")
```

main/scala/akka/testkit/TestKit.scala, line 45 (Code Correctness: Non-Static Inner Class Implements Serializable) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: Class: TestActor$SetIgnore
File: main/scala/akka/testkit/TestKit.scala:45
Taint Flags:

```
42 def run(sender: ActorRef, msg: Any): AutoPilot = sys.error("must not call")
43 }
44
45 final case class SetIgnore(i: Ignore) extends NoSerializationVerificationNeeded
46 final case class Watch(ref: ActorRef) extends NoSerializationVerificationNeeded
47 final case class UnWatch(ref: ActorRef) extends NoSerializationVerificationNeeded
48 final case class SetAutoPilot(ap: AutoPilot) extends NoSerializationVerificationNeeded
```

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

Package: akka.testkit

main/scala/akka/testkit/TestKit.scala, line 49 (Code Correctness: Non-Static Inner Class Implements Serializable) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: Class: TestActor$Spawn
File: main/scala/akka/testkit/TestKit.scala:49
Taint Flags:

```
46 final case class Watch(ref: ActorRef) extends NoSerializationVerificationNeeded
47 final case class UnWatch(ref: ActorRef) extends NoSerializationVerificationNeeded
48 final case class SetAutoPilot(ap: AutoPilot) extends NoSerializationVerificationNeeded
49 final case class Spawn(props: Props, name: Option[String] = None, strategy: Option[SupervisorStrategy] = None)
50 extends NoSerializationVerificationNeeded {
51 def apply(context: ActorRefFactory): ActorRef = name match {
52 case Some(n) => context.actorOf(props, n)
```

main/scala/akka/testkit/TestKit.scala, line 46 (Code Correctness: Non-Static Inner Class Implements Serializable) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: Class: TestActor$Watch
File: main/scala/akka/testkit/TestKit.scala:46
Taint Flags:

```
43 }
44
45 final case class SetIgnore(i: Ignore) extends NoSerializationVerificationNeeded
46 final case class Watch(ref: ActorRef) extends NoSerializationVerificationNeeded
47 final case class UnWatch(ref: ActorRef) extends NoSerializationVerificationNeeded
48 final case class SetAutoPilot(ap: AutoPilot) extends NoSerializationVerificationNeeded
49 final case class Spawn(props: Props, name: Option[String] = None, strategy: Option[SupervisorStrategy] = None)
```

Package: akka.testkit.metrics

test/scala/akka/testkit/metrics/MetricKeyDSL.scala, line 9 (Code Correctness: Non-Static Inner Class Implements Serializable) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Code Correctness: Non-Static Inner Class Implements Serializable	Low
Package: akka.testkit.metrics	
test/scala/akka/testkit/metrics/MetricKeyDSL.scala, line 9 (Code Correctness: Non-Static Inner Class Implements Serializable)	**Low**

Sink Details

Sink: Class: MetricKeyDSL$MetricKey
File: test/scala/akka/testkit/metrics/MetricKeyDSL.scala:9
Taint Flags:

```
6
7 trait MetricKeyDSL {
8
9  case class MetricKey private[MetricKeyDSL] (path: String) {
10
11  import MetricKey._
12
```

Dead Code: Expression is Always false (16 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`.

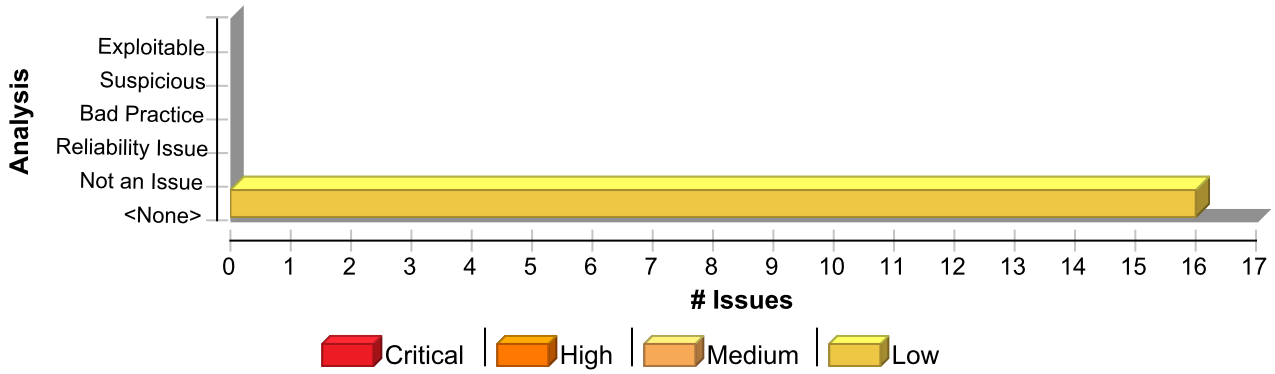
```
public void setUpCalls() {
  boolean firstCall = false;
  boolean secondCall = false;

  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	16	0	0	16
Total	**16**	**0**	**0**	**16**

Dead Code: Expression is Always false — Low

Package: akka.testkit

test/scala/akka/testkit/AkkaSpecSpec.scala, line 75 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()
File: test/scala/akka/testkit/AkkaSpecSpec.scala:75
Taint Flags:

```
72 val davyJones = otherSystem.actorOf(Props(new Actor {
73 def receive = {
74 case m: DeadLetter => locker :+= m
75 case "Die!" => sender() ! "finally gone"; context.stop(self)
76 }
77 }), "davyJones")
78
```

test/scala/akka/testkit/TestActorRefSpec.scala, line 74 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Dead Code: Expression is Always false — Low

Package: akka.testkit

test/scala/akka/testkit/TestActorRefSpec.scala, line 74 (Dead Code: Expression is Always false) — Low

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()
File: test/scala/akka/testkit/TestActorRefSpec.scala:74
Taint Flags:

```
71 class SenderActor(replyActor: ActorRef) extends TActor {
72
73 def receiveT = {
74 case "complex" => replyActor ! "complexRequest"
75 case "complex2" => replyActor ! "complexRequest2"
76 case "simple" => replyActor ! "simpleRequest"
77 case "complexReply" => {
```

test/scala/akka/testkit/TestActorRefSpec.scala, line 60 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()
File: test/scala/akka/testkit/TestActorRefSpec.scala:60
Taint Flags:

```
57
58 class WorkerActor() extends TActor {
59 def receiveT = {
60 case "work" =>
61 sender() ! "workDone"
62 context.stop(self)
63 case replyTo: Promise[_] => replyTo.asInstanceOf[Promise[Any]].success("complexReply")
```

test/scala/akka/testkit/TestActorRefSpec.scala, line 45 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()

Dead Code: Expression is Always false — Low

Package: akka.testkit

test/scala/akka/testkit/TestActorRefSpec.scala, line 45 (Dead Code: Expression is Always false) — Low

File: test/scala/akka/testkit/TestActorRefSpec.scala:45
Taint Flags:

```
42 var replyTo: ActorRef = null
43
44 def receiveT = {
45 case "complexRequest" => {
46 replyTo = sender()
47 val worker = TestActorRef(Props[WorkerActor]())
48 worker ! "work"
```

test/scala/akka/testkit/TestActorRefSpec.scala, line 54 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()
File: test/scala/akka/testkit/TestActorRefSpec.scala:54
Taint Flags:

```
51 val worker = TestActorRef(Props[WorkerActor]())
52 worker ! sender()
53 case "workDone" => replyTo ! "complexReply"
54 case "simpleRequest" => sender() ! "simpleReply"
55 }
56 }
57
```

test/scala/akka/testkit/TestActorRefSpec.scala, line 205 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()
File: test/scala/akka/testkit/TestActorRefSpec.scala:205
Taint Flags:

```
202 override def supervisorStrategy =
```

Dead Code: Expression is Always false — Low

Package: akka.testkit

test/scala/akka/testkit/TestActorRefSpec.scala, line 205 (Dead Code: Expression is Always false) — Low

```
203 OneForOneStrategy(maxNrOfRetries = 5, withinTimeRange = 1 second)(List(classOf[ActorKilledException]))
204
205 def receiveT = { case "sendKill" => ref ! Kill }
206 }))
207
208 boss ! "sendKill"
```

test/scala/akka/testkit/TestProbeSpec.scala, line 161 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()
File: test/scala/akka/testkit/TestProbeSpec.scala:161
Taint Flags:

```
158 probe.ref ! "done"
159
160 probe.fishForMessage() {
161 case "fishForMe" => true
162 case _ => false
163 }
164
```

test/scala/akka/testkit/TestFSMRefSpec.scala, line 21 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()
File: test/scala/akka/testkit/TestFSMRefSpec.scala:21
Taint Flags:

```
18 val fsm = TestFSMRef(new Actor with FSM[Int, String] {
19 startWith(1, "")
20 when(1) {
21 case Event("go", _) => goto(2).using("go")
```

Dead Code: Expression is Always false — Low

Package: akka.testkit

test/scala/akka/testkit/TestFSMRefSpec.scala, line 21 (Dead Code: Expression is Always false) — Low

```
22  case Event(StateTimeout, _) => goto(2).using("timeout")
23  }
24  when(2) {
```

test/scala/akka/testkit/TestProbeSpec.scala, line 123 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: run()
File: test/scala/akka/testkit/TestProbeSpec.scala:123
Taint Flags:

```
120  probe.setAutoPilot(new TestActor.AutoPilot {
121  def run(sender: ActorRef, msg: Any): TestActor.AutoPilot =
122  msg match {
123  case "stop" => TestActor.NoAutoPilot
124  case x => testActor.tell(x, sender); TestActor.KeepRunning
125  }
126  })
```

test/scala/akka/testkit/TestActorRefSpec.scala, line 53 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()
File: test/scala/akka/testkit/TestActorRefSpec.scala:53
Taint Flags:

```
50  case "complexRequest2" =>
51  val worker = TestActorRef(Props[WorkerActor]())
52  worker ! sender()
53  case "workDone" => replyTo ! "complexReply"
54  case "simpleRequest" => sender() ! "simpleReply"
55  }
56  }
```

Dead Code: Expression is Always false — Low

Package: akka.testkit

test/scala/akka/testkit/TestFSMRefSpec.scala, line 25 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()
File: test/scala/akka/testkit/TestFSMRefSpec.scala:25
Taint Flags:

```
22 case Event(StateTimeout, _) => goto(2).using("timeout")
23 }
24 when(2) {
25 case Event("back", _) => goto(1).using("back")
26 }
27 }, "test-fsm-ref-1")
28 fsm.stateName should ===(1)
```

test/scala/akka/testkit/TestActorRefSpec.scala, line 80 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()
File: test/scala/akka/testkit/TestActorRefSpec.scala:80
Taint Flags:

```
77 case "complexReply" => {
78 counter -= 1
79 }
80 case "simpleReply" => {
81 counter -= 1
82 }
83 }
```

test/scala/akka/testkit/TestActorRefSpec.scala, line 50 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Dead Code: Expression is Always false — Low

Package: akka.testkit

test/scala/akka/testkit/TestActorRefSpec.scala, line 50 (Dead Code: Expression is Always false) — Low

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()
File: test/scala/akka/testkit/TestActorRefSpec.scala:50
Taint Flags:

```
47 val worker = TestActorRef(Props[WorkerActor]())
48 worker ! "work"
49 }
50 case "complexRequest2" =>
51 val worker = TestActorRef(Props[WorkerActor]())
52 worker ! sender()
53 case "workDone" => replyTo ! "complexReply"
```

test/scala/akka/testkit/TestActorRefSpec.scala, line 76 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()
File: test/scala/akka/testkit/TestActorRefSpec.scala:76
Taint Flags:

```
73 def receiveT = {
74 case "complex" => replyActor ! "complexRequest"
75 case "complex2" => replyActor ! "complexRequest2"
76 case "simple" => replyActor ! "simpleRequest"
77 case "complexReply" => {
78 counter -= 1
79 }
```

test/scala/akka/testkit/TestActorRefSpec.scala, line 77 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()

Dead Code: Expression is Always false — Low

Package: akka.testkit

test/scala/akka/testkit/TestActorRefSpec.scala, line 77 (Dead Code: Expression is Always false) — Low

File: test/scala/akka/testkit/TestActorRefSpec.scala:77
Taint Flags:

```
74 case "complex" => replyActor ! "complexRequest"
75 case "complex2" => replyActor ! "complexRequest2"
76 case "simple" => replyActor ! "simpleRequest"
77 case "complexReply" => {
78 counter -= 1
79 }
80 case "simpleReply" => {
```

test/scala/akka/testkit/TestActorRefSpec.scala, line 75 (Dead Code: Expression is Always false) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: applyOrElse()
File: test/scala/akka/testkit/TestActorRefSpec.scala:75
Taint Flags:

```
72
73 def receiveT = {
74 case "complex" => replyActor ! "complexRequest"
75 case "complex2" => replyActor ! "complexRequest2"
76 case "simple" => replyActor ! "simpleRequest"
77 case "complexReply" => {
78 counter -= 1
```

Dead Code: Expression is Always true (4 issues)

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`.

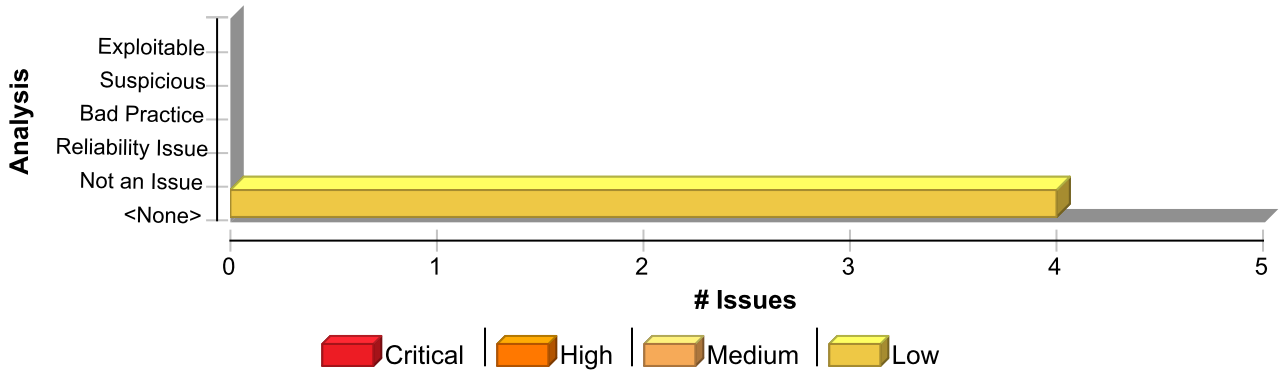
```
public void setUpCalls() {
  boolean firstCall = false;
  boolean secondCall = false;

  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	4	0	0	4
Total	**4**	**0**	**0**	**4**

Dead Code: Expression is Always true — Low

Package: akka.testkit

main/scala/akka/testkit/TestKit.scala, line 997 (Dead Code: Expression is Always true) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: poll()
File: main/scala/akka/testkit/TestKit.scala:997
Taint Flags:

```
994  if (!p) {
995  val toSleep = stop - now
996  if (toSleep <= Duration.Zero) {
997  if (noThrow) false
998  else throw new AssertionError(s"timeout $max expired")
999  } else {
1000  Thread.sleep((toSleep min interval).toMillis)
```

main/scala/akka/testkit/TestKit.scala, line 342 (Dead Code: Expression is Always true) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Dead Code: Expression is Always true — Low

Package: akka.testkit

main/scala/akka/testkit/TestKit.scala, line 342 (Dead Code: Expression is Always true) — Low

Sink: IfStatement
Enclosing Method: poll()
File: main/scala/akka/testkit/TestKit.scala:342
Taint Flags:

```
339 else null.asInstanceOf[A]
340 }
341
342 if (!failed) result
343 else {
344 Thread.sleep(t.toMillis)
345 poll((stop - now) min interval)
```

Package: akka.testkit.metrics

test/scala/akka/testkit/metrics/MetricsKit.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: reportMetrics()
File: test/scala/akka/testkit/metrics/MetricsKit.scala:97
Taint Flags:

```
94 * HINT: this operation can be costy, run outside of your tested code, or rely on scheduled reporting.
95 */
96 def reportMetrics(): Unit = {
97 if (reportMetricsEnabled)
98 reporters.foreach { _.report() }
99 }
100
```

test/scala/akka/testkit/metrics/MetricsKitOps.scala, line 71 (Dead Code: Expression is Always true) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: IfStatement
Enclosing Method: gc()
File: test/scala/akka/testkit/metrics/MetricsKitOps.scala:71

Dead Code: Expression is Always true	Low
Package: akka.testkit.metrics	
test/scala/akka/testkit/metrics/MetricsKitOps.scala, line 71 (Dead Code: Expression is Always true)	**Low**

Taint Flags:

```
68
69 /** Yet another delegate to `System.gc()` */
70 def gc(): Unit = {
71 if (forceGcEnabled)
72 System.gc()
73 }
74
```

Insecure Randomness (2 issues)

Abstract

Standard pseudorandom number generators cannot withstand cryptographic attacks.

Explanation

Insecure randomness errors occur when a function that can produce predictable values is used as a source of randomness in a security-sensitive context. Computers are deterministic machines, and as such are unable to produce true randomness. Pseudorandom Number Generators (PRNGs) approximate randomness algorithmically, starting with a seed from which subsequent values are calculated. There are two types of PRNGs: statistical and cryptographic. Statistical PRNGs provide useful statistical properties, but their output is highly predictable and form an easy to reproduce numeric stream that is unsuitable for use in cases where security depends on generated values being unpredictable. Cryptographic PRNGs address this problem by generating output that is more difficult to predict. For a value to be cryptographically secure, it must be impossible or highly improbable for an attacker to distinguish between the generated random value and a truly random value. In general, if a PRNG algorithm is not advertised as being cryptographically secure, then it is probably a statistical PRNG and should not be used in security-sensitive contexts, where its use can lead to serious vulnerabilities such as easy-to-guess temporary passwords, predictable cryptographic keys, session hijacking, and DNS spoofing. **Example:** The following code uses a statistical PRNG to create a URL for a receipt that remains active for some period of time after a purchase.

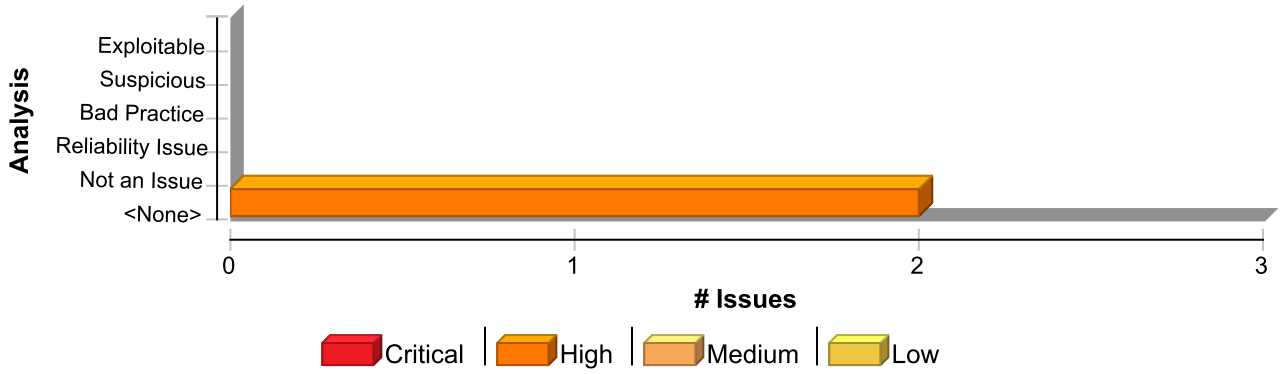
```
def GenerateReceiptURL(baseUrl : String) : String {
    val ranGen = new scala.util.Random()
    ranGen.setSeed((new Date()).getTime())
    return (baseUrl + ranGen.nextInt(400000000) + ".html")
}
```

This code uses the `Random.nextInt()` function to generate "unique" identifiers for the receipt pages it generates. Since `Random.nextInt()` is a statistical PRNG, it is easy for an attacker to guess the strings it generates. Although the underlying design of the receipt system is also faulty, it would be more secure if it used a random number generator that did not produce predictable receipt identifiers, such as a cryptographic PRNG.

Recommendation

When unpredictability is critical, as is the case with most security-sensitive uses of randomness, use a cryptographic PRNG. Regardless of the PRNG you choose, always use a value with sufficient entropy to seed the algorithm. (Do not use values such as the current time because it offers only negligible entropy.) The Java language provides a cryptographic PRNG in `java.security.SecureRandom`. As is the case with other algorithm-based classes in `java.security`, `SecureRandom` provides an implementation-independent wrapper around a particular set of algorithms. When you request an instance of a `SecureRandom` object using `SecureRandom.getInstance()`, you can request a specific implementation of the algorithm. If the algorithm is available, then it is given as a `SecureRandom` object. If it is unavailable or if you do not specify a particular implementation, then you are given a `SecureRandom` implementation selected by the system. Sun provides a single `SecureRandom` implementation with the Java distribution named `SHA1PRNG`, which Sun describes as computing: "The SHA-1 hash over a true-random seed value concatenated with a 64-bit counter which is incremented by 1 for each operation. From the 160-bit SHA-1 output, only 64 bits are used [1]." However, the specifics of the Sun implementation of the `SHA1PRNG` algorithm are poorly documented, and it is unclear what sources of entropy the implementation uses and therefore what amount of true randomness exists in its output. Although there is speculation on the Web about the Sun implementation, there is no evidence to contradict the claim that the algorithm is cryptographically strong and can be used safely in security-sensitive contexts.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Insecure Randomness	2	0	0	2
Total	**2**	**0**	**0**	**2**

Insecure Randomness — High

Package: main.scala.akka.testkit

main/scala/akka/testkit/SocketUtil.scala, line 86 (Insecure Randomness) — High

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Semantic)

Sink Details

Sink: nextInt()
Enclosing Method: apply()
File: main/scala/akka/testkit/SocketUtil.scala:86
Taint Flags:

```
83 case RANDOM_LOOPBACK_ADDRESS =>
84 // JDK limitation? You cannot bind on addresses matching the pattern 127.x.y.255,
85 // that's why the last component must be < 255
86 if (canBindOnAlternativeLoopbackAddresses) s"127.20.${Random.nextInt(256)}.${Random.nextInt(255)}"
87 else "127.0.0.1"
88 case other =>
89 other
```

main/scala/akka/testkit/SocketUtil.scala, line 86 (Insecure Randomness) — High

Issue Details

Kingdom: Security Features
Scan Engine: SCA (Semantic)

Sink Details

Sink: nextInt()
Enclosing Method: apply()
File: main/scala/akka/testkit/SocketUtil.scala:86
Taint Flags:

Insecure Randomness	High
Package: main.scala.akka.testkit	
main/scala/akka/testkit/SocketUtil.scala, line 86 (Insecure Randomness)	**High**

```
83 case RANDOM_LOOPBACK_ADDRESS =>
84 // JDK limitation? You cannot bind on addresses matching the pattern 127.x.y.255,
85 // that's why the last component must be < 255
86 if (canBindOnAlternativeLoopbackAddresses) s"127.20.${Random.nextInt(256)}.${Random.nextInt(255)}"
87 else "127.0.0.1"
88 case other =>
89 other
```

J2EE Bad Practices: Sockets (6 issues)

Abstract

Socket-based communication in web applications is prone to error.

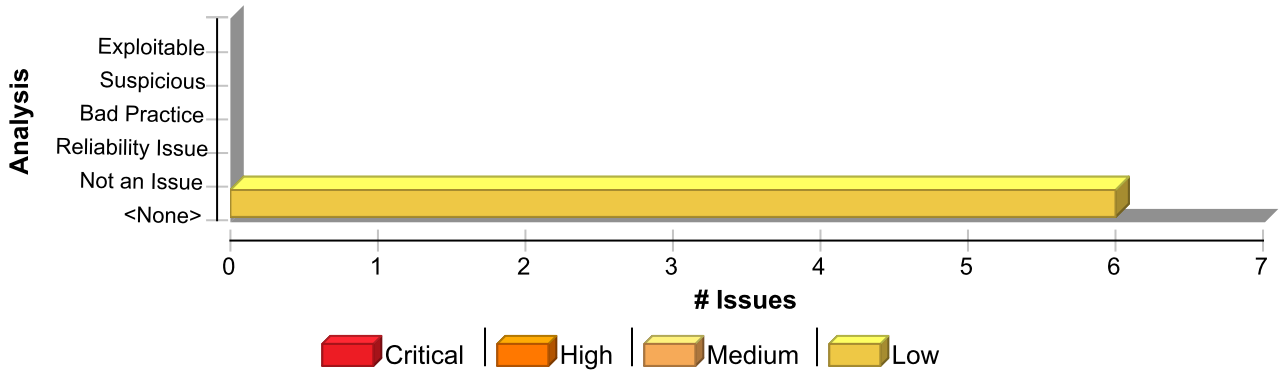
Explanation

The J2EE standard permits the use of sockets only for the purpose of communication with legacy systems when no higher-level protocol is available. Authoring your own communication protocol requires wrestling with difficult security issues, including: - In-band versus out-of-band signaling - Compatibility between protocol versions - Channel security - Error handling - Network constraints (firewalls) - Session management Without significant scrutiny by a security expert, chances are good that a custom communication protocol will suffer from security problems. Many of the same issues apply to a custom implementation of a standard protocol. While there are usually more resources available that address security concerns related to implementing a standard protocol, these resources are also available to attackers.

Recommendation

Replace a custom communication protocol with an industry standard protocol or framework. Consider whether you can use a protocol such as HTTP, FTP, SMTP, CORBA, RMI/IIOP, EJB, or SOAP. Consider the security track record of the protocol implementation you choose.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
J2EE Bad Practices: Sockets	6	0	0	6
Total	**6**	**0**	**0**	**6**

J2EE Bad Practices: Sockets	**Low**
Package: akka.testkit	
main/scala/akka/testkit/SocketUtil.scala, line 51 (J2EE Bad Practices: Sockets)	**Low**

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Semantic)

Sink Details

J2EE Bad Practices: Sockets — Low

Package: akka.testkit

main/scala/akka/testkit/SocketUtil.scala, line 51 (J2EE Bad Practices: Sockets) — Low

Sink: InetSocketAddress()
Enclosing Method: findBoth()
File: main/scala/akka/testkit/SocketUtil.scala:51
Taint Flags:

```
48 val tcpPort = SocketUtil.temporaryLocalPort(udp = false)
49 val ds: DatagramSocket = DatagramChannel.open().socket()
50 try {
51 ds.bind(new InetSocketAddress("localhost", tcpPort))
52 tcpPort
53 } catch {
54 case NonFatal(_) => findBoth(tries - 1)
```

main/scala/akka/testkit/SocketUtil.scala, line 122 (J2EE Bad Practices: Sockets) — Low

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Semantic)

Sink Details

Sink: InetSocketAddress()
Enclosing Method: notBoundServerAddress()
File: main/scala/akka/testkit/SocketUtil.scala:122
Taint Flags:

```
119 port
120 }
121
122 def notBoundServerAddress(address: String): InetSocketAddress = new InetSocketAddress(address, 0)
123
124 def notBoundServerAddress(): InetSocketAddress = notBoundServerAddress("127.0.0.1")
125 }
```

main/scala/akka/testkit/SocketUtil.scala, line 116 (J2EE Bad Practices: Sockets) — Low

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Semantic)

Sink Details

Sink: InetSocketAddress()
Enclosing Method: temporaryUdpIpv6Port()
File: main/scala/akka/testkit/SocketUtil.scala:116
Taint Flags:

```
113
114 def temporaryUdpIpv6Port(iface: NetworkInterface) = {
```

J2EE Bad Practices: Sockets — Low

Package: akka.testkit

main/scala/akka/testkit/SocketUtil.scala, line 116 (J2EE Bad Practices: Sockets) — Low

```
115 val serverSocket = DatagramChannel.open(StandardProtocolFamily.INET6).socket()
116 serverSocket.bind(new InetSocketAddress(iface.getInetAddresses.nextElement(), 0))
117 val port = serverSocket.getLocalPort
118 serverSocket.close()
119 port
```

Package: main.scala.akka.testkit

main/scala/akka/testkit/SocketUtil.scala, line 100 (J2EE Bad Practices: Sockets) — Low

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Semantic)

Sink Details

Sink: InetSocketAddress()
Enclosing Method: apply()
File: main/scala/akka/testkit/SocketUtil.scala:100
Taint Flags:

```
97 } else {
98 val ss = ServerSocketChannel.open().socket()
99 ss.bind(addr)
100 (ss, new InetSocketAddress(address, ss.getLocalPort))
101 } catch {
102 case NonFatal(ex) =>
103 throw new RuntimeException(s"Binding to $addr failed with ${ex.getMessage}", ex)
```

main/scala/akka/testkit/SocketUtil.scala, line 96 (J2EE Bad Practices: Sockets) — Low

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Semantic)

Sink Details

Sink: InetSocketAddress()
Enclosing Method: apply()
File: main/scala/akka/testkit/SocketUtil.scala:96
Taint Flags:

```
93 try if (udp) {
94 val ds = DatagramChannel.open().socket()
95 ds.bind(addr)
96 (ds, new InetSocketAddress(address, ds.getLocalPort))
97 } else {
98 val ss = ServerSocketChannel.open().socket()
99 ss.bind(addr)
```

J2EE Bad Practices: Sockets	Low
Package: main.scala.akka.testkit	
main/scala/akka/testkit/SocketUtil.scala, line 96 (J2EE Bad Practices: Sockets)	**Low**

main/scala/akka/testkit/SocketUtil.scala, line 92 (J2EE Bad Practices: Sockets)	Low

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Semantic)

Sink Details

Sink: InetSocketAddress()
Enclosing Method: apply()
File: main/scala/akka/testkit/SocketUtil.scala:92
Taint Flags:

```
89  other
90  }
91
92  val addr = new InetSocketAddress(address, 0)
93  try if (udp) {
94  val ds = DatagramChannel.open().socket()
95  ds.bind(addr)
```

J2EE Bad Practices: Threads (16 issues)

Abstract

Thread management in a web application is forbidden in some circumstances and is always highly error prone.

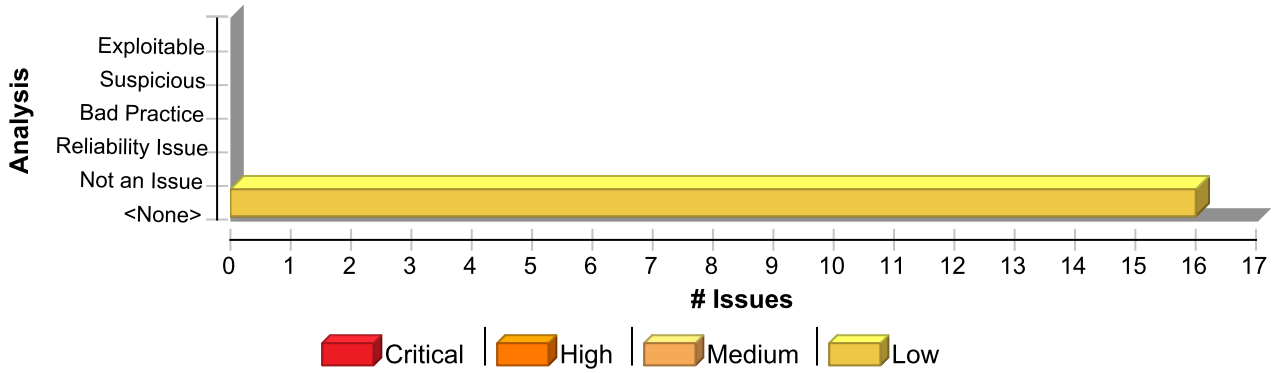
Explanation

Thread management in a web application is forbidden by the J2EE standard in some circumstances and is always highly error prone. Managing threads is difficult and is likely to interfere in unpredictable ways with the behavior of the application container. Even without interfering with the container, thread management usually leads to bugs that are hard to detect and diagnose like deadlock, race conditions, and other synchronization errors.

Recommendation

Avoid managing threads directly from within the web application. Instead use standards such as message driven beans and the EJB timer service that are provided by the application container.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
J2EE Bad Practices: Threads	16	0	0	16
Total	**16**	**0**	**0**	**16**

J2EE Bad Practices: Threads	Low
Package: akka.testkit	
test/scala/akka/testkit/CoronerSpec.scala, line 73 (J2EE Bad Practices: Threads)	**Low**

Issue Details

Kingdom: Time and State
Scan Engine: SCA (Semantic)

Sink Details

Sink: run()
Enclosing Method: lockingThread()
File: test/scala/akka/testkit/CoronerSpec.scala:73
Taint Flags:

J2EE Bad Practices: Threads — Low

Package: akka.testkit

test/scala/akka/testkit/CoronerSpec.scala, line 73 (J2EE Bad Practices: Threads) — Low

```
70 def lockingThread(name: String, initialLocks: List[ReentrantLock]): LockingThread = {
71 val ready = new Semaphore(0)
72 val proceed = new Semaphore(0)
73 val t = new Thread(new Runnable {
74 def run =
75 try recursiveLock(initialLocks)
76 catch { case _: InterruptedException => () }
```

main/scala/akka/testkit/TestKit.scala, line 380 (J2EE Bad Practices: Threads) — Low

Issue Details

Kingdom: Time and State
Scan Engine: SCA (Semantic)

Sink Details

Sink: sleep()
Enclosing Method: poll()
File: main/scala/akka/testkit/TestKit.scala:380
Taint Flags:

```
377 }
378
379 if (instantNow < stop) {
380 Thread.sleep(t.toMillis)
381 poll((stop - now) min interval)
382 } else {
383 result
```

main/scala/akka/testkit/TestKit.scala, line 344 (J2EE Bad Practices: Threads) — Low

Issue Details

Kingdom: Time and State
Scan Engine: SCA (Semantic)

Sink Details

Sink: sleep()
Enclosing Method: poll()
File: main/scala/akka/testkit/TestKit.scala:344
Taint Flags:

```
341
342 if (!failed) result
343 else {
344 Thread.sleep(t.toMillis)
345 poll((stop - now) min interval)
346 }
```

J2EE Bad Practices: Threads — Low

Package: akka.testkit

main/scala/akka/testkit/TestKit.scala, line 344 (J2EE Bad Practices: Threads) — Low

```
347 }
```

main/scala/akka/testkit/TestKit.scala, line 301 (J2EE Bad Practices: Threads) — Low

Issue Details

Kingdom: Time and State
Scan Engine: SCA (Semantic)

Sink Details

Sink: sleep()
Enclosing Method: poll()
File: main/scala/akka/testkit/TestKit.scala:301
Taint Flags:

```
298 def poll(t: Duration): Unit = {
299 if (!p) {
300 assert(now < stop, s"timeout ${_max} expired: $message")
301 Thread.sleep(t.toMillis)
302 poll((stop - now) min interval)
303 }
304 }
```

main/scala/akka/testkit/CallingThreadDispatcher.scala, line 241 (J2EE Bad Practices: Threads) — Low

Issue Details

Kingdom: Time and State
Scan Engine: SCA (Semantic)

Sink Details

Sink: run()
Enclosing Method: executeTask()
File: main/scala/akka/testkit/CallingThreadDispatcher.scala:241
Taint Flags:

```
238 }
239 }
240
241 protected[akka] override def executeTask(invocation: TaskInvocation): Unit = { invocation.run() }
242
243 /*
244 * This method must be called with this thread's queue.
```

test/scala/akka/testkit/CoronerSpec.scala, line 94 (J2EE Bad Practices: Threads) — Low

Issue Details

J2EE Bad Practices: Threads — Low

Package: akka.testkit

test/scala/akka/testkit/CoronerSpec.scala, line 94 (J2EE Bad Practices: Threads) — Low

Kingdom: Time and State
Scan Engine: SCA (Semantic)

Sink Details

Sink: start()
Enclosing Method: lockingThread()
File: test/scala/akka/testkit/CoronerSpec.scala:94
Taint Flags:

```
91  }
92  }
93  }, name)
94  t.start()
95  LockingThread(name, t, ready, proceed)
96  }
97
```

main/scala/akka/testkit/TestKit.scala, line 755 (J2EE Bad Practices: Threads) — Low

Issue Details

Kingdom: Time and State
Scan Engine: SCA (Semantic)

Sink Details

Sink: sleep()
Enclosing Method: expectNoMsg_internal()
File: main/scala/akka/testkit/TestKit.scala:755
Taint Flags:

```
752  while (left.toNanos > 0 && elem == null) {
753  //Use of (left / 2) gives geometric series limited by finish time similar to (1/2)^n limited by 1,
754  //so it is very precise
755  Thread.sleep(pollInterval.toMillis min (left / 2).toMillis)
756  left = leftNow
757  if (left.toNanos > 0) {
758  elem = queue.peekFirst()
```

main/scala/akka/testkit/CallingThreadDispatcher.scala, line 347 (J2EE Bad Practices: Threads) — Low

Issue Details

Kingdom: Time and State
Scan Engine: SCA (Semantic)

Sink Details

Sink: ThreadLocal()

J2EE Bad Practices: Threads — Low

Package: akka.testkit

main/scala/akka/testkit/CallingThreadDispatcher.scala, line 347 (J2EE Bad Practices: Threads) — Low

Enclosing Method: CallingThreadMailbox$$anon$1()
File: main/scala/akka/testkit/CallingThreadDispatcher.scala:347
Taint Flags:

```
344 val system = _receiver.system
345 val self = _receiver.self
346
347 private val q = new ThreadLocal[MessageQueue]() {
348 override def initialValue = {
349 val queue = mailboxType.create(Some(self), Some(system))
350 CallingThreadDispatcherQueues(system).registerQueue(CallingThreadMailbox.this, queue)
```

main/scala/akka/testkit/ExplicitlyTriggeredScheduler.scala, line 61 (J2EE Bad Practices: Threads) — Low

Issue Details

Kingdom: Time and State
Scan Engine: SCA (Semantic)

Sink Details

Sink: sleep()
Enclosing Method: timePasses()
File: main/scala/akka/testkit/ExplicitlyTriggeredScheduler.scala:61
Taint Flags:

```
58 // Give dispatchers time to clear :(. See
59 // https://github.com/akka/akka/pull/24243#discussion_r160985493
60 // for some discussion on how to deal with this properly.
61 Thread.sleep(100)
62
63 val newTime = currentTime.get + amount.toMillis
64 if (log.isDebugEnabled)
```

main/scala/akka/testkit/TestKit.scala, line 1000 (J2EE Bad Practices: Threads) — Low

Issue Details

Kingdom: Time and State
Scan Engine: SCA (Semantic)

Sink Details

Sink: sleep()
Enclosing Method: poll()
File: main/scala/akka/testkit/TestKit.scala:1000
Taint Flags:

```
997 if (noThrow) false
```

J2EE Bad Practices: Threads — Low

Package: akka.testkit

main/scala/akka/testkit/TestKit.scala, line 1000 (J2EE Bad Practices: Threads) — Low

```
998  else throw new AssertionError(s"timeout $max expired")
999  } else {
1000 Thread.sleep((toSleep min interval).toMillis)
1001 poll()
1002 }
1003 } else true
```

test/scala/akka/testkit/Coroner.scala, line 150 (J2EE Bad Practices: Threads) — Low

Issue Details

Kingdom: Time and State
Scan Engine: SCA (Semantic)

Sink Details

Sink: sleep()
Enclosing Method: printReport()
File: test/scala/akka/testkit/Coroner.scala:150
Taint Flags:

```
147 // If we look too soon, we've seen the JVM report a thread
148 // (in that case deadlock-thread-a from CoronerSpec)
149 // waiting on the lock while no thread seemed to be holding it
150 Thread.sleep(300)
151
152 def dumpAllThreads: Seq[ThreadInfo] =
153 threadMx.dumpAllThreads(threadMx.isObjectMonitorUsageSupported, threadMx.isSynchronizerUsageSupported).toSeq
```

test/scala/akka/testkit/Coroner.scala, line 122 (J2EE Bad Practices: Threads) — Low

Issue Details

Kingdom: Time and State
Scan Engine: SCA (Semantic)

Sink Details

Sink: start()
Enclosing Method: watch()
File: test/scala/akka/testkit/Coroner.scala:122
Taint Flags:

```
119 watchedHandle.finished()
120 }
121 }
122 new Thread(new Runnable { def run = triggerReportIfOverdue(duration) }, "Coroner").start()
123 watchedHandle.waitForStart()
124 watchedHandle
125 }
```

J2EE Bad Practices: Threads — Low

Package: akka.testkit

test/scala/akka/testkit/Coroner.scala, line 122 (J2EE Bad Practices: Threads) — Low

test/scala/akka/testkit/Coroner.scala, line 122 (J2EE Bad Practices: Threads) — Low

Issue Details

Kingdom: Time and State
Scan Engine: SCA (Semantic)

Sink Details

Sink: run()
Enclosing Method: watch()
File: test/scala/akka/testkit/Coroner.scala:122
Taint Flags:

```
119 watchedHandle.finished()
120 }
121 }
122 new Thread(new Runnable { def run = triggerReportIfOverdue(duration) }, "Coroner").start()
123 watchedHandle.waitForStart()
124 watchedHandle
125 }
```

Package: main.scala.akka.testkit

main/scala/akka/testkit/ExplicitlyTriggeredScheduler.scala, line 87 (J2EE Bad Practices: Threads) — Low

Issue Details

Kingdom: Time and State
Scan Engine: SCA (Semantic)

Sink Details

Sink: run()
Enclosing Method: apply()
File: main/scala/akka/testkit/ExplicitlyTriggeredScheduler.scala:87
Taint Flags:

```
84 scheduledTasks(runTo).headOption match {
85 case Some((task, time)) =>
86 currentTime.set(time)
87 val runResult = Try(task.runnable.run())
88 scheduled.remove(task)
89
90 if (runResult.isSuccess)
```

J2EE Bad Practices: Threads — Low

Package: test.scala.akka.testkit

test/scala/akka/testkit/CoronerSpec.scala, line 115 (J2EE Bad Practices: Threads) — Low

Issue Details

Kingdom: Time and State
Scan Engine: SCA (Semantic)

Sink Details

Sink: interrupt()
Enclosing Method: apply()
File: test/scala/akka/testkit/CoronerSpec.scala:115
Taint Flags:

```
112
113 val (_, report) = captureOutput(Coroner.printReport("Deadlock test", _))
114
115 a.thread.interrupt()
116 b.thread.interrupt()
117
118 report should include("Coroner's Report")
```

test/scala/akka/testkit/CoronerSpec.scala, line 116 (J2EE Bad Practices: Threads) — Low

Issue Details

Kingdom: Time and State
Scan Engine: SCA (Semantic)

Sink Details

Sink: interrupt()
Enclosing Method: apply()
File: test/scala/akka/testkit/CoronerSpec.scala:116
Taint Flags:

```
113 val (_, report) = captureOutput(Coroner.printReport("Deadlock test", _))
114
115 a.thread.interrupt()
116 b.thread.interrupt()
117
118 report should include("Coroner's Report")
119
```

Poor Error Handling: Overly Broad Catch (2 issues)

Abstract

The catch block handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.

Explanation

Multiple catch blocks can get repetitive, but "condensing" catch blocks by catching a high-level class such as `Exception` can obscure exceptions that deserve special treatment or that should not be caught at this point in the program. Catching an overly broad exception essentially defeats the purpose of Java's typed exceptions, and can become particularly dangerous if the program grows and begins to throw new types of exceptions. The new exception types will not receive any attention. **Example:** The following code excerpt handles three types of exceptions in an identical fashion.

```
  try {
    doExchange();
  }
  catch (IOException e) {
    logger.error("doExchange failed", e);
  }
  catch (InvocationTargetException e) {
    logger.error("doExchange failed", e);
  }
  catch (SQLException e) {
    logger.error("doExchange failed", e);
  }
```

At first blush, it may seem preferable to deal with these exceptions in a single catch block, as follows:

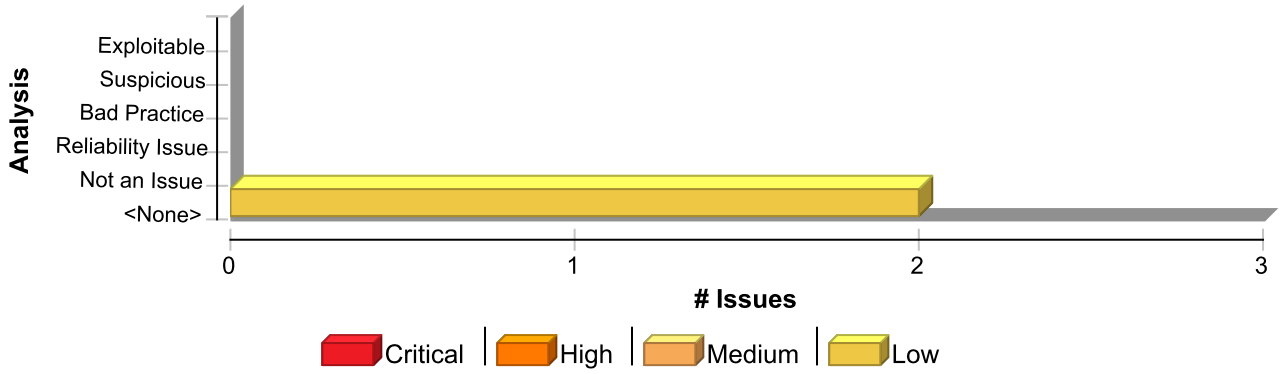
```
  try {
    doExchange();
  }
  catch (Exception e) {
    logger.error("doExchange failed", e);
  }
```

However, if `doExchange()` is modified to throw a new type of exception that should be handled in some different kind of way, the broad catch block will prevent the compiler from pointing out the situation. Further, the new catch block will now also handle exceptions derived from `RuntimeException` such as `ClassCastException`, and `NullPointerException`, which is not the programmer's intent.

Recommendation

Do not catch broad exception classes such as `Exception`, `Throwable`, `Error`, or `RuntimeException` except at the very top level of the program or thread.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Poor Error Handling: Overly Broad Catch	2	0	0	2
Total	**2**	**0**	**0**	**2**

Poor Error Handling: Overly Broad Catch — Low

Package: akka.testkit

main/scala/akka/testkit/TestKitUtils.scala, line 25 (Poor Error Handling: Overly Broad Catch) — Low

Issue Details

Kingdom: Errors
Scan Engine: SCA (Structural)

Sink Details

Sink: CatchBlock
Enclosing Method: isAbstractClass()
File: main/scala/akka/testkit/TestKitUtils.scala:25
Taint Flags:

```
22 try {
23 Modifier.isAbstract(Class.forName(className).getModifiers)
24 } catch {
25 case _: Throwable => false // yes catch everything, best effort check
26 }
27 }
28
```

Package: main.scala.akka.testkit

main/scala/akka/testkit/TestEventListener.scala, line 587 (Poor Error Handling: Overly Broad Catch) — Low

Issue Details

Kingdom: Errors
Scan Engine: SCA (Structural)

Sink Details

Poor Error Handling: Overly Broad Catch	Low
Package: main.scala.akka.testkit	
main/scala/akka/testkit/TestEventListener.scala, line 587 (Poor Error Handling: Overly Broad Catch)	**Low**

Sink: CatchBlock
Enclosing Method: apply()
File: main/scala/akka/testkit/TestEventListener.scala:587
Taint Flags:

```
584 filters.exists(f =>
585 try {
586 f(event)
587 } catch { case _: Exception => false })
588
589 def addFilter(filter: EventFilter): Unit = filters ::= filter
590
```

Poor Style: Value Never Read (1 issue)

Abstract

The variable's value is assigned but never used, making it a dead store.

Explanation

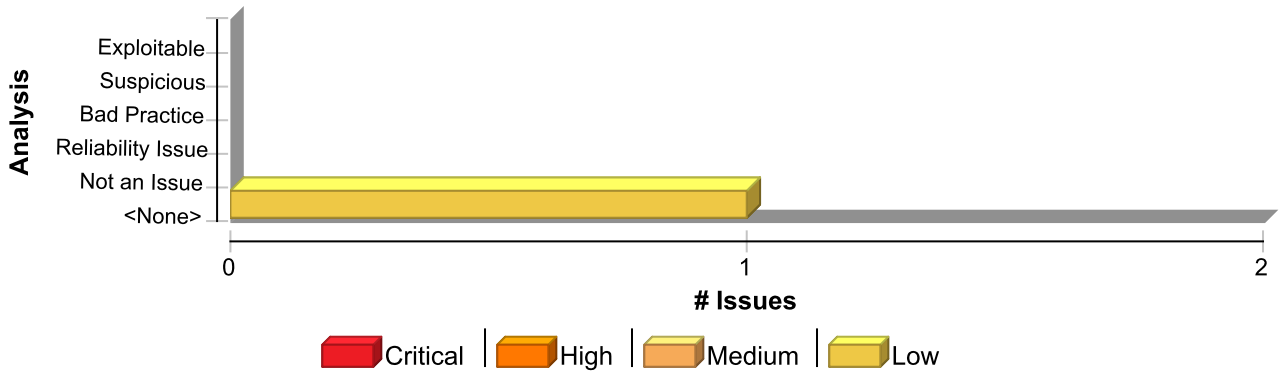
This variable's value is not used. After the assignment, the variable is either assigned another value or goes out of scope. **Example:** The following code excerpt assigns to the variable `r` and then overwrites the value without using it.

```
r = getName();
r = getNewBuffer(buf);
```

Recommendation

Remove unnecessary assignments in order to make the code easier to understand and maintain.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Poor Style: Value Never Read	1	0	0	1
Total	**1**	**0**	**0**	**1**

Poor Style: Value Never Read	**Low**
Package: akka.testkit	
main/scala/akka/testkit/TestActorRef.scala, line 43 (Poor Style: Value Never Read)	**Low**

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: VariableAccess: disregard
Enclosing Method: TestActorRef()
File: main/scala/akka/testkit/TestActorRef.scala:43
Taint Flags:

Poor Style: Value Never Read	Low

Package: akka.testkit

main/scala/akka/testkit/TestActorRef.scala, line 43 (Poor Style: Value Never Read)	Low

```
40 o.getClass)
41 }
42 case s =>
43 _system.log.error(
44 "trying to attach child {} to unknown type of supervisor {}, this is not going to end well",
45 name,
46 s.getClass)
```

Redundant Null Check (1 issue)

Abstract

The program can dereference a null-pointer, thereby causing a null-pointer exception.

Explanation

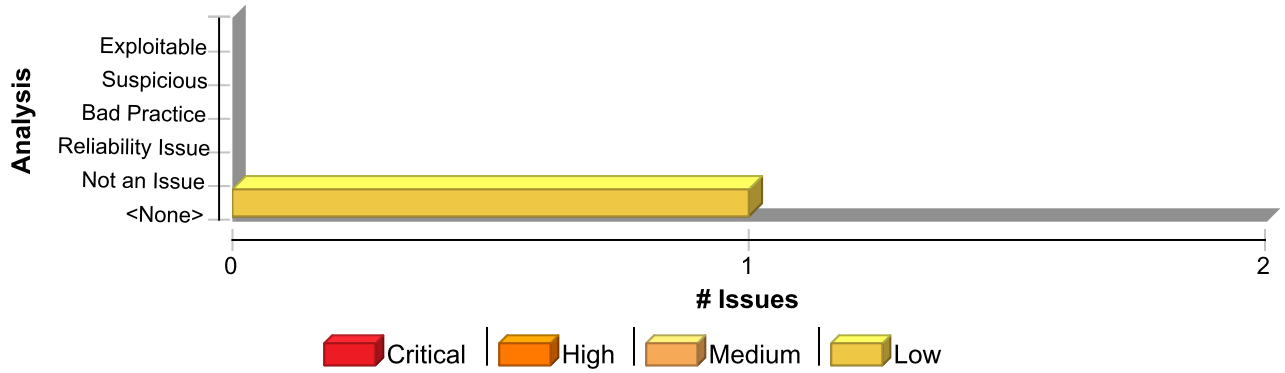
Null-pointer exceptions usually occur when one or more of the programmer's assumptions is violated. Specifically, dereference-after-check errors occur when a program makes an explicit check for `null`, but proceeds to dereference the object when it is known to be `null`. Errors of this type are often the result of a typo or programmer oversight. Most null-pointer issues result in general software reliability problems, but if attackers can intentionally cause the program to dereference a null-pointer, they can use the resulting exception to mount a denial of service attack or to cause the application to reveal debugging information that will be valuable in planning subsequent attacks. **Example 1:** In the following code, the programmer confirms that the variable `foo` is `null` and subsequently dereferences it erroneously. If `foo` is `null` when it is checked in the `if` statement, then a `null` dereference will occur, thereby causing a null-pointer exception.

```
if (foo == null) {
    foo.setBar(val);
    ...
}
```

Recommendation

Implement careful checks before dereferencing objects that might be `null`. When possible, abstract `null` checks into wrappers around code that manipulates resources to ensure that they are applied in all cases and to minimize the places where mistakes can occur.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Redundant Null Check	1	0	0	1
Total	**1**	**0**	**0**	**1**

Redundant Null Check	Low

Package: akka.testkit

main/scala/akka/testkit/TestKit.scala, line 762 (Redundant Null Check) Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Control Flow)

Sink Details

Sink: Dereferenced : elem
Enclosing Method: expectNoMsg_internal()
File: main/scala/akka/testkit/TestKit.scala:762
Taint Flags:

```
759 }
760 }
761
762 if (elem ne null) {
763 // we pop the message, such that subsequent expectNoMessage calls can
764 // assert on the next period without a message
765 queue.pop()
```

System Information Leak (1 issue)

Abstract

Revealing system data or debugging information helps an adversary learn about the system and form a plan of attack.

Explanation

An information leak occurs when system data or debug information leaves the program through an output stream or logging function. **Example 1:** The following code writes an exception to the standard error stream:

```
try {
    ...
} catch (Exception e) {
    e.printStackTrace();
}
```

Depending upon the system configuration, this information can be dumped to a console, written to a log file, or exposed to a remote user. For example, with scripting mechanisms it is trivial to redirect output information from "Standard error" or "Standard output" into a file or another program. Alternatively, the system that the program runs on could have a remote logging mechanism such as a "syslog" server that sends the logs to a remote device. During development, you have no way of knowing where this information might end up being displayed. In some cases, the error message provides the attacker with the precise type of attack to which the system is vulnerable. For example, a database error message can reveal that the application is vulnerable to a SQL injection attack. Other error messages can reveal more oblique clues about the system. In `Example 1`, the leaked information could imply information about the type of operating system, the applications installed on the system, and the amount of care that the administrators have put into configuring the program. Here is another scenario, specific to the mobile world. Most mobile devices now implement a Near-Field Communication (NFC) protocol for quickly sharing information between devices using radio communication. It works by bringing devices to close proximity or simply having them touch each other. Even though the communication range of NFC is limited to just a few centimeters, eavesdropping, data modification and various other types of attacks are possible, since NFC alone does not ensure secure communication. **Example 2:** The Android platform provides support for NFC. The following code creates a message that gets pushed to the other device within the range.

```
...
public static final String TAG = "NfcActivity";
private static final String DATA_SPLITTER = "__:DATA:__";
private static final String MIME_TYPE = "application/my.applications.mimetype";
...
public NdefMessage createNdefMessage(NfcEvent event) {
    TelephonyManager tm =
(TelephonyManager)Context.getSystemService(Context.TELEPHONY_SERVICE);
    String VERSION = tm.getDeviceSoftwareVersion();
    String text = TAG + DATA_SPLITTER + VERSION;
    NdefRecord record = new NdefRecord(NdefRecord.TNF_MIME_MEDIA,
            MIME_TYPE.getBytes(), new byte[0], text.getBytes());
    NdefRecord[] records = { record };
    NdefMessage msg = new NdefMessage(records);
    return msg;
}
...
```

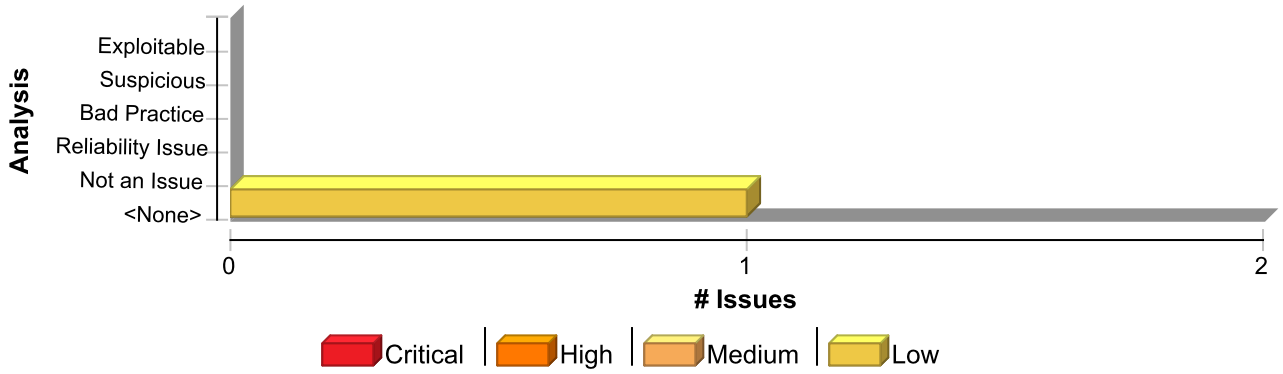
NFC Data Exchange Format (NDEF) message contains typed data, a URI, or a custom application payload. If the message contains information about the application, such as its name, MIME type, or device software version, this information could be leaked to an eavesdropper. In `Example 2`, Fortify Static Code Analyzer reports a System Information Leak vulnerability on the return statement.

Recommendation

Write error messages with security in mind. In production environments, turn off detailed error information in favor of

brief messages. Restrict the generation and storage of detailed output that can help administrators and programmers diagnose problems. Debug traces can sometimes appear in non-obvious places (embedded in comments in the HTML for an error page, for example). Even brief error messages that do not reveal stack traces or database dumps can potentially aid an attacker. For example, an "Access Denied" message can reveal that a file or user exists on the system. If you are concerned about leaking system data via NFC on an Android device, you could do one of the following three things. Do not include system data in the messages pushed to other devices in range, encrypt the payload of the message, or establish a secure communication channel at a higher layer.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
System Information Leak	1	0	0	1
Total	**1**	**0**	**0**	**1**

System Information Leak — Low

Package: akka.testkit

test/scala/akka/testkit/Coroner.scala, line 108 (System Information Leak) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Semantic)

Sink Details

Sink: printStackTrace()
Enclosing Method: akka$testkit$Coroner$$triggerReportIfOverdue()
File: test/scala/akka/testkit/Coroner.scala:108
Taint Flags:

```
105 catch {
106 case NonFatal(ex) => {
107 out.println("Error displaying Coroner's Report")
108 ex.printStackTrace(out)
109 }
110 }
111 }
```

System Information Leak: Internal (2 issues)

Abstract

Revealing system data or debugging information helps an adversary learn about the system and form a plan of attack.

Explanation

An internal information leak occurs when system data or debug information is sent to a local file, console, or screen via printing or logging. **Example 1:** The following code writes an exception to the standard error stream:

```
try {
    ...
} catch (Exception e) {
    e.printStackTrace();
}
```

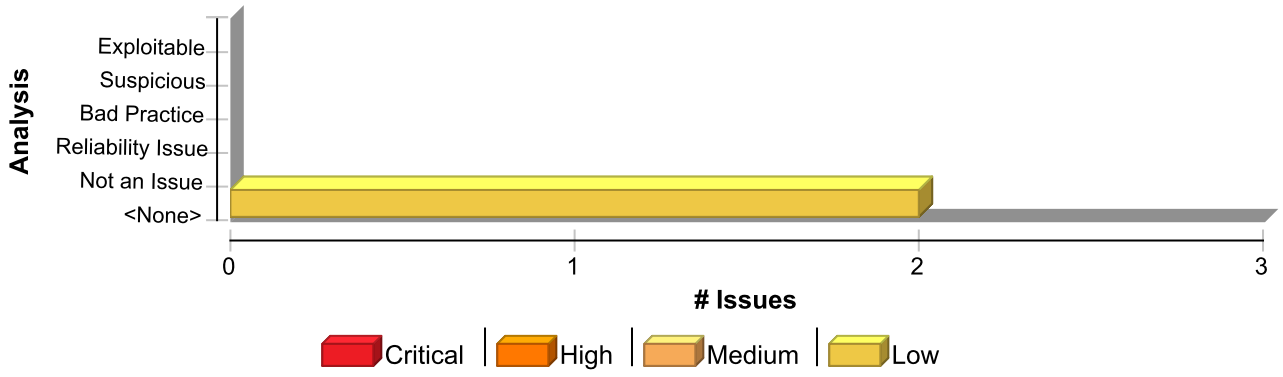
Depending upon the system configuration, this information can be dumped to a console, written to a log file, or exposed to a user. In some cases, the error message provides the attacker with the precise type of attack to which the system is vulnerable. For example, a database error message can reveal that the application is vulnerable to a SQL injection attack. Other error messages can reveal more oblique clues about the system. In `Example 1`, the leaked information could imply information about the type of operating system, the applications installed on the system, and the amount of care that the administrators have put into configuring the program. Information leaks are also a concern in a mobile computing environment. **Example 2:** The following code logs the stack trace of a caught exception on the Android platform.

```
...
try {
  ...
} catch (Exception e) {
    Log.e(TAG, Log.getStackTraceString(e));
}
...
```

Recommendation

Write error messages with security in mind. In production environments, turn off detailed error information in favor of brief messages. Restrict the generation and storage of detailed output that can help administrators and programmers diagnose problems. Debug traces can sometimes appear in non-obvious places (embedded in comments in the HTML for an error page, for example). Even brief error messages that do not reveal stack traces or database dumps can potentially aid an attacker. For example, an "Access Denied" message can reveal that a file or user exists on the system.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
System Information Leak: Internal	2	0	0	2
Total	**2**	**0**	**0**	**2**

System Information Leak: Internal — Low

Package: akka.testkit

test/scala/akka/testkit/Coroner.scala, line 138 (System Information Leak: Internal) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Data Flow)

Source Details

Source: java.lang.management.RuntimeMXBean.getUptime()
From: akka.testkit.Coroner.printReport
File: test/scala/akka/testkit/Coroner.scala:138

```
135  val memMx = ManagementFactory.getMemoryMXBean()
136  val threadMx = ManagementFactory.getThreadMXBean()
137
138  println(s"""#Coroner's Report: $reportTitle
139  #OS Architecture: ${osMx.getArch()}
140  #Available processors: ${osMx.getAvailableProcessors()}
141  #System load (last minute): ${osMx.getSystemLoadAverage()}
```

Sink Details

Sink: java.io.PrintStream.println()
Enclosing Method: printReport()
File: test/scala/akka/testkit/Coroner.scala:138
Taint Flags: NUMBER, SYSTEMINFO

```
135  val memMx = ManagementFactory.getMemoryMXBean()
136  val threadMx = ManagementFactory.getThreadMXBean()
137
138  println(s"""#Coroner's Report: $reportTitle
139  #OS Architecture: ${osMx.getArch()}
140  #Available processors: ${osMx.getAvailableProcessors()}
141  #System load (last minute): ${osMx.getSystemLoadAverage()}
```

test/scala/akka/testkit/Coroner.scala, line 138 (System Information Leak: Internal) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Data Flow)

Source Details

System Information Leak: Internal	Low

Package: akka.testkit

test/scala/akka/testkit/Coroner.scala, line 138 (System Information Leak: Internal) **Low**

Source: java.lang.management.RuntimeMXBean.getStartTime()
From: akka.testkit.Coroner.printReport
File: test/scala/akka/testkit/Coroner.scala:142

```
139  #OS Architecture: ${osMx.getArch()}
140  #Available processors: ${osMx.getAvailableProcessors()}
141  #System load (last minute): ${osMx.getSystemLoadAverage()}
142  #VM start time: ${new Date(rtMx.getStartTime())}
143  #VM uptime: ${rtMx.getUptime()}ms
144  #Heap usage: ${memMx.getHeapMemoryUsage()}
145  #Non-heap usage: ${memMx.getNonHeapMemoryUsage()}""".stripMargin('#'))
```

Sink Details

Sink: java.io.PrintStream.println()
Enclosing Method: printReport()
File: test/scala/akka/testkit/Coroner.scala:138
Taint Flags: NUMBER, SYSTEMINFO

```
135  val memMx = ManagementFactory.getMemoryMXBean()
136  val threadMx = ManagementFactory.getThreadMXBean()
137
138  println(s"""#Coroner's Report: $reportTitle
139  #OS Architecture: ${osMx.getArch()}
140  #Available processors: ${osMx.getAvailableProcessors()}
141  #System load (last minute): ${osMx.getSystemLoadAverage()}
```

Unchecked Return Value (1 issue)

Abstract

Ignoring a method's return value can cause the program to overlook unexpected states and conditions.

Explanation

It is not uncommon for Java programmers to misunderstand `read()` and related methods that are part of many `java.io` classes. Most errors and unusual events in Java result in an exception being thrown. (This is one of the advantages that Java has over languages like C: Exceptions make it easier for programmers to think about what can go wrong.) But the stream and reader classes do not consider it unusual or exceptional if only a small amount of data becomes available. These classes simply add the small amount of data to the return buffer, and set the return value to the number of bytes or characters read. There is no guarantee that the amount of data returned is equal to the amount of data requested. This behavior makes it important for programmers to examine the return value from `read()` and other IO methods to ensure that they receive the amount of data they expect. **Example:** The following code loops through a set of users, reading a private data file for each user. The programmer assumes that the files are always exactly 1 kilobyte in size and therefore ignores the return value from `read()`. If an attacker can create a smaller file, the program will recycle the remainder of the data from the previous user and handle it as though it belongs to the attacker.

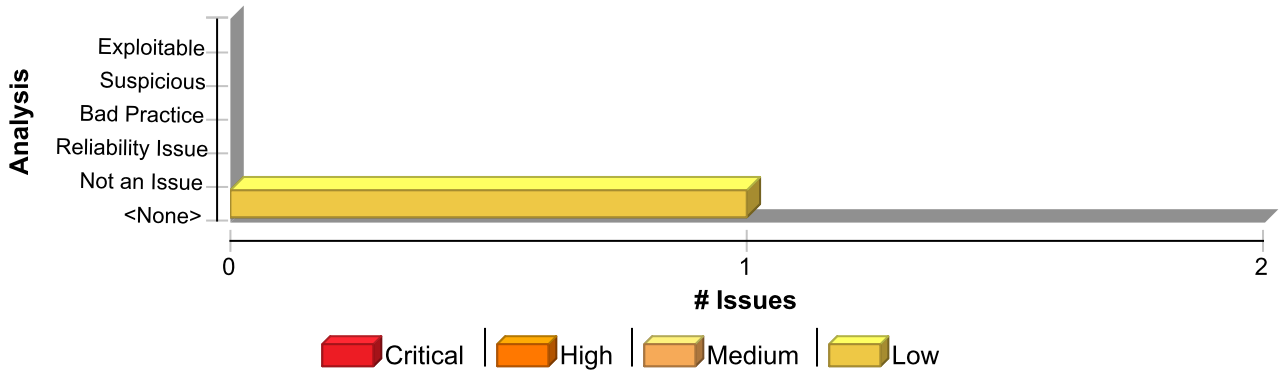
```
FileInputStream fis;
byte[] byteArray = new byte[1024];
for (Iterator i=users.iterator(); i.hasNext();) {
    String userName = (String) i.next();
    String pFileName = PFILE_ROOT + "/" + userName;
    FileInputStream fis = new FileInputStream(pFileName);
    fis.read(byteArray); // the file is always 1k bytes
    fis.close();
    processPFile(userName, byteArray);
}
```

Recommendation

```
  FileInputStream fis;
  byte[] byteArray = new byte[1024];
  for (Iterator i=users.iterator(); i.hasNext();) {
    String userName = (String) i.next();
    String pFileName = PFILE_ROOT + "/" + userName;
    fis = new FileInputStream(pFileName);
    int bRead = 0;
    while (bRead < 1024) {
        int rd = fis.read(byteArray, bRead, 1024 - bRead);
        if (rd == -1) {
          throw new IOException("file is unusually small");
        }
        bRead += rd;
    }
    // could add check to see if file is too large here
    fis.close();
    processPFile(userName, byteArray);
  }
```

Note: Because the fix for this problem is relatively complicated, you might be tempted to use a simpler approach, such as checking the size of the file before you begin reading. Such an approach would render the application vulnerable to a file system race condition, whereby an attacker could replace a well-formed file with a malicious file between the file size check and the call to read data from the file.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Unchecked Return Value	1	0	0	1
Total	**1**	**0**	**0**	**1**

Unchecked Return Value	Low
Package: akka.testkit	
main/scala/akka/testkit/CallingThreadDispatcher.scala, line 266 (Unchecked Return Value)	**Low**

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Semantic)

Sink Details

Sink: interrupted()
Enclosing Method: throwInterruptionIfExistsOrSet()
File: main/scala/akka/testkit/CallingThreadDispatcher.scala:266
Taint Flags:

```
263 def throwInterruptionIfExistsOrSet(intEx: InterruptedException): Unit = {
264 val ie = checkThreadInterruption(intEx)
265 if (ie ne null) {
266 Thread.interrupted() // clear interrupted flag before throwing according to java convention
267 throw ie
268 }
269 }
```

Unreleased Resource: Synchronization (1 issue)

Abstract

The program fails to release a lock it holds, which might lead to deadlock.

Explanation

The program can potentially fail to release a system resource. Resource leaks have at least two common causes: - Error conditions and other exceptional circumstances. - Confusion over which part of the program is responsible for releasing the resource. Most unreleased resource issues result in general software reliability problems. However, if an attacker can intentionally trigger a resource leak, the attacker may be able to launch a denial of service by depleting the resource pool. **Example 1:** The following code establishes a lock before `performOperationInCriticalSection()`, but fails to release the lock if an exception is thrown in that method.

```
  ReentrantLock  myLock = new ReentrantLock();

  myLock.lock();
  performOperationInCriticalSection();
  myLock.unlock();
```

This category was derived from the Cigital Java Rulepack.

Recommendation

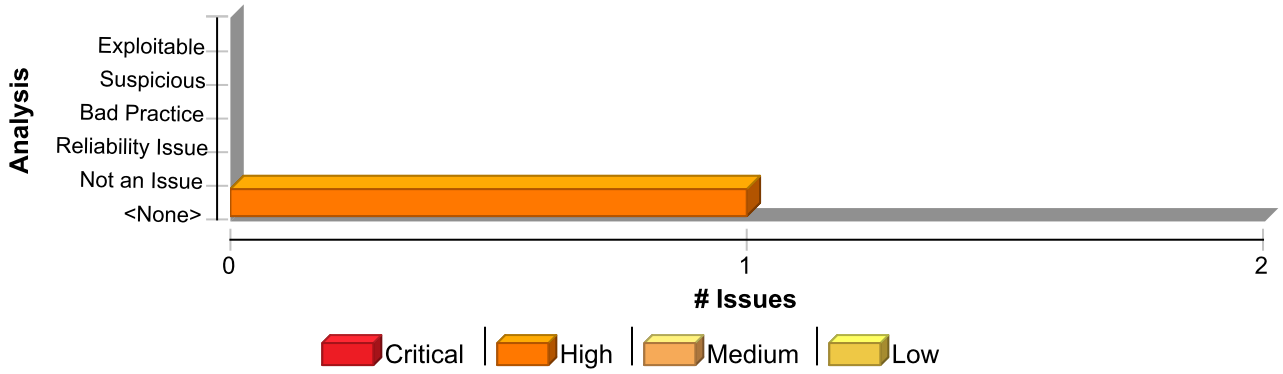
Because resource leaks can be hard to track down, establish a set of resource management patterns and idioms for your software and do not tolerate deviations from your conventions. One good pattern for addressing the error handling mistake in this example is to release the lock in a `finally` block. **Example 2:** The following code will always release the lock.

```
  ReentrantLock  myLock = new ReentrantLock();

  try {
    myLock.lock();
    performOperationInCriticalSection();
    myLock.unlock();
  }
  finally {
    if (myLock != null) {
      myLock.unlock();
    }
  }
```

Example 3: If using Kotlin, it is advisable to use the `withLock` function, removing the possibility of forgetting to unlock.

```
  val myLock = ReentrantLock()
  myLock.withLock {
    performOperationInCriticalSection()
  }
```

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Unreleased Resource: Synchronization	1	0	0	1
Total	**1**	**0**	**0**	**1**

Unreleased Resource: Synchronization	**High**
Package: akka.testkit	
main/scala/akka/testkit/CallingThreadDispatcher.scala, line 306 (Unreleased Resource: Synchronization)	**High**

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Control Flow)

Sink Details

Sink: gotLock = mbox.ctdLock().tryLock(...) : locked
Enclosing Method: runQueue()
File: main/scala/akka/testkit/CallingThreadDispatcher.scala:306
Taint Flags:

```
303 if (!mbox.ctdLock.isHeldByCurrentThread) {
304 var intex = interruptedEx
305 val gotLock = try {
306 mbox.ctdLock.tryLock(50, TimeUnit.MILLISECONDS)
307 } catch {
308 case ie: InterruptedException =>
309 Thread.interrupted() // clear interrupted flag before we continue, exception will be thrown later
```