Fortify Standalone Report Generator

Developer Workbook

akka-remote-tests

Table of Contents

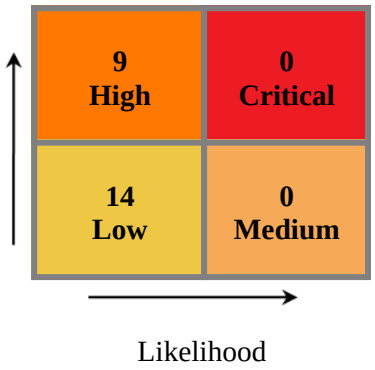
Executive Summary

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-remote-tests 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-remote-tests
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, 11:44 AM	**Engine Version:**	21.1.1.0009
Host Name:	Jacks-Work-MBP.local	**Certification:**	VALID
Number of Files:	7	**Lines of Code:**	533

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 / 2	0 / 2
Code Correctness: Non-Static Inner Class Implements Serializable	0	0	0	0 / 1	0 / 1
Denial of Service	0	0	0	0 / 2	0 / 2
Denial of Service: Regular Expression	0	0 / 2	0	0	0 / 2
J2EE Bad Practices: Leftover Debug Code	0	0	0	0 / 1	0 / 1
J2EE Bad Practices: Sockets	0	0	0	0 / 2	0 / 2
J2EE Bad Practices: Threads	0	0	0	0 / 4	0 / 4
Missing Check against Null	0	0	0	0 / 2	0 / 2
Often Misused: Authentication	0	0 / 2	0	0	0 / 2
Often Misused: Boolean.getBoolean()	0	0 / 2	0	0	0 / 2
Path Manipulation	0	0 / 3	0	0	0 / 3

Results Outline

Code Correctness: Constructor Invokes Overridable Function (2 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");
    }
  }
}
```

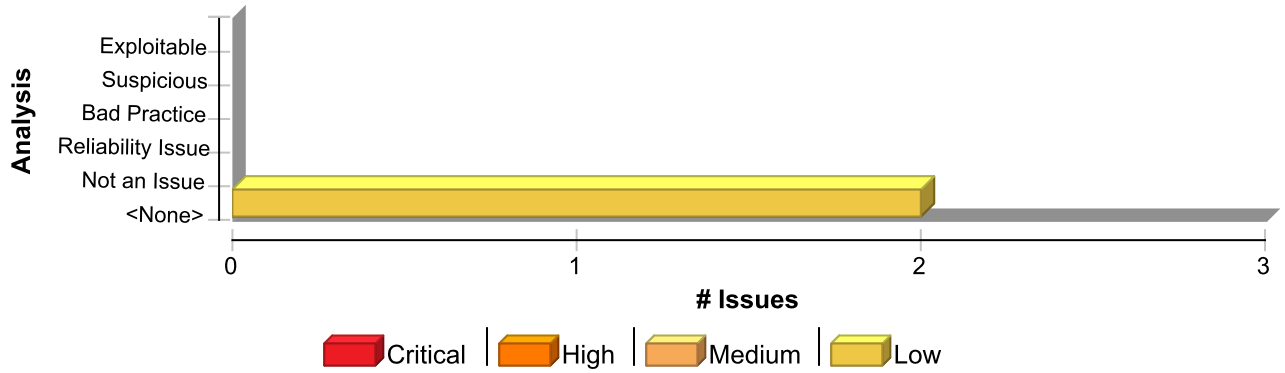
The code in `Example 1` prints "Attack successful!", since the `Attacker` class overrides the `validateUser()` function that is called from the constructor of the superclass `User`, and Java will first look in the subclass for functions called from the constructor.

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	2	0	0	2
Total	**2**	**0**	**0**	**2**

Code Correctness: Constructor Invokes Overridable Function	**Low**
Package: akka.remote.testconductor	
akka/remote/testconductor/BarrierSpec.scala, line 26 (Code Correctness: Constructor Invokes Overridable Function)	**Low**
Issue Details	

Code Correctness: Constructor Invokes Overridable Function — Low

Package: akka.remote.testconductor

akka/remote/testconductor/BarrierSpec.scala, line 26 (Code Correctness: Constructor Invokes Overridable Function) — Low

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionCall: config
Enclosing Method: BarrierSpec()
File: akka/remote/testconductor/BarrierSpec.scala:26
Taint Flags:

```
23 """
24 }
25
26 class BarrierSpec extends AkkaSpec(BarrierSpec.config) with ImplicitSender {
27
28  import BarrierCoordinator._
29  import BarrierSpec._
```

akka/remote/testconductor/ControllerSpec.scala, line 24 (Code Correctness: Constructor Invokes Overridable Function) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionCall: config
Enclosing Method: ControllerSpec()
File: akka/remote/testconductor/ControllerSpec.scala:24
Taint Flags:

```
21 """
22 }
23
24 class ControllerSpec extends AkkaSpec(ControllerSpec.config) with ImplicitSender {
25
26  val A = RoleName("a")
27  val B = RoleName("b")
```

Code Correctness: Non-Static Inner Class Implements Serializable (1 issue)

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

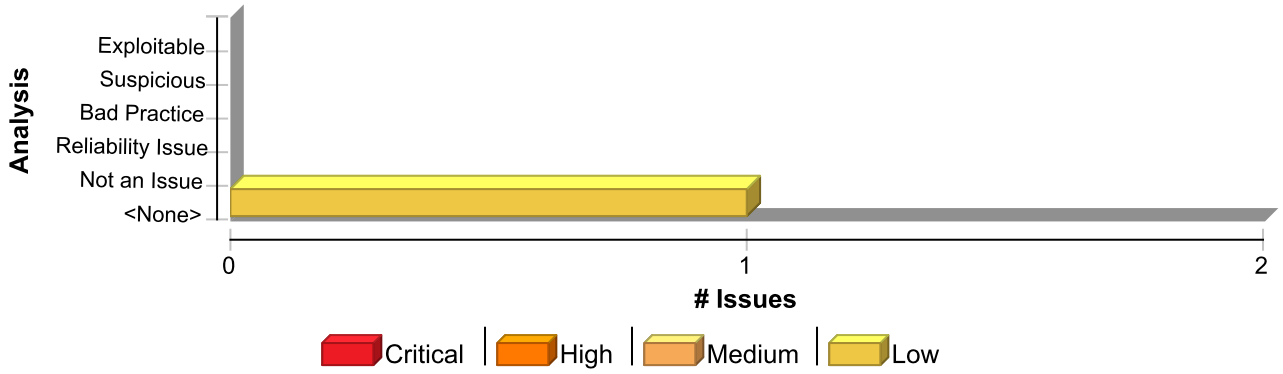
When using inner classes, they should not be serialized, or they should be changed to static-nested classes, since these do not have the drawbacks that non-static inner classes have when serialized. When a nested class is static it inherently has no association with instance variables (including those 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	1	0	0	1
Total	**1**	**0**	**0**	**1**

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

Package: akka.remote.testconductor

akka/remote/testconductor/BarrierSpec.scala, line 17 (Code Correctness: Non-Static Inner Class Implements Serializable) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: Class: BarrierSpec$Failed
File: akka/remote/testconductor/BarrierSpec.scala:17
Taint Flags:

```
14 import akka.testkit.{ AkkaSpec, EventFilter, ImplicitSender, TestProbe, TimingTest }
15
16 object BarrierSpec {
17  final case class Failed(ref: ActorRef, thr: Throwable)
18  val config = """
19  akka.testconductor.barrier-timeout = 5s
20  akka.actor.provider = remote
```

Denial of Service (2 issues)

Abstract

An attacker could cause the program to crash or otherwise become unavailable to legitimate users.

Explanation

Attackers may be able to deny service to legitimate users by flooding the application with requests, but flooding attacks can often be defused at the network layer. More problematic are bugs that allow an attacker to overload the application using a small number of requests. Such bugs allow the attacker to specify the quantity of system resources their requests will consume or the duration for which they will use them. **Example 1:** The following code allows a user to specify the amount of time for which a thread will sleep. By specifying a large number, an attacker may tie up the thread indefinitely. With a small number of requests, the attacker may deplete the application's thread pool.

```
  int usrSleepTime = Integer.parseInt(usrInput);
  Thread.sleep(usrSleepTime);
```

Example 2: The following code reads a String from a zip file. Because it uses the `readLine()` method, it will read an unbounded amount of input. An attacker may take advantage of this code to cause an `OutOfMemoryException` or to consume a large amount of memory so that the program spends more time performing garbage collection or runs out of memory during some subsequent operation.

```
  InputStream zipInput = zipFile.getInputStream(zipEntry);
  Reader zipReader = new InputStreamReader(zipInput);
  BufferedReader br = new BufferedReader(zipReader);
  String line = br.readLine();
```

Recommendation

Validate user input to ensure that it will not cause inappropriate resource utilization. **Example 3:** The following code allows a user to specify the amount of time for which a thread will sleep just as in `Example 1`, but only if the value is within reasonable bounds.

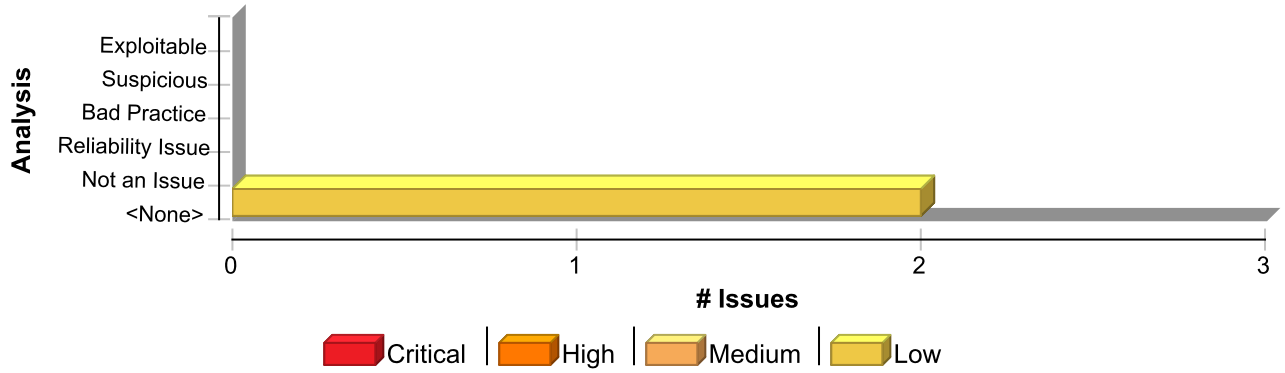
```
  int usrSleepTime = Integer.parseInt(usrInput);
  if (usrSleepTime >= SLEEP_MIN &&
      usrSleepTime <= SLEEP_MAX) {
    Thread.sleep(usrSleepTime);
  } else {
    throw new Exception("Invalid sleep duration");
  }
}
```

Example 4: The following code reads a String from a zip file just as in `Example 2`, but the maximum string length it will read is `MAX_STR_LEN` characters.

```
  InputStream zipInput = zipFile.getInputStream(zipEntry);
  Reader zipReader = new InputStreamReader(zipInput);
  BufferedReader br = new BufferedReader(zipReader);
  StringBuffer sb = new StringBuffer();
  int intC;
  while ((intC = br.read()) != -1) {
    char c = (char) intC;
    if (c == '\n') {
      break;
    }
    if (sb.length() >= MAX_STR_LEN) {
      throw new Exception("input too long");
    }
    sb.append(c);
  }
```

```
String line = sb.toString();
```

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Denial of Service	2	0	0	2
Total	**2**	**0**	**0**	**2**

Denial of Service — Low

Package: akka.remote.testkit

akka/remote/testkit/LogRoleReplace.scala, line 104 (Denial of Service) — Low

Issue Details

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

Sink Details

Sink: readLine()
Enclosing Method: processLines()
File: akka/remote/testkit/LogRoleReplace.scala:104
Taint Flags:

```
101 def processLines(line: String): Unit =
102 if (line ne null) {
103 out.println(processLine(line))
104 processLines(in.readLine)
105 }
106
107 processLines(in.readLine())
```

akka/remote/testkit/LogRoleReplace.scala, line 107 (Denial of Service) — Low

Issue Details

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

Denial of Service	Low

Package: akka.remote.testkit

akka/remote/testkit/LogRoleReplace.scala, line 107 (Denial of Service) — Low

Sink Details

Sink: readLine()
Enclosing Method: process()
File: akka/remote/testkit/LogRoleReplace.scala:107
Taint Flags:

```
104 processLines(in.readLine)
105 }
106
107 processLines(in.readLine())
108 }
109
110 def processLine(line: String): String = {
```

Denial of Service: Regular Expression (2 issues)

Abstract

Untrusted data is passed to the application and used as a regular expression. This can cause the thread to overconsume CPU resources.

Explanation

There is a vulnerability in implementations of regular expression evaluators and related methods that can cause the thread to hang when evaluating regular expressions that contain a grouping expression that is itself repeated. Additionally, any regular expression that contains alternate subexpressions that overlap one another can also be exploited. This defect can be used to execute a Denial of Service (DoS) attack. **Example:**

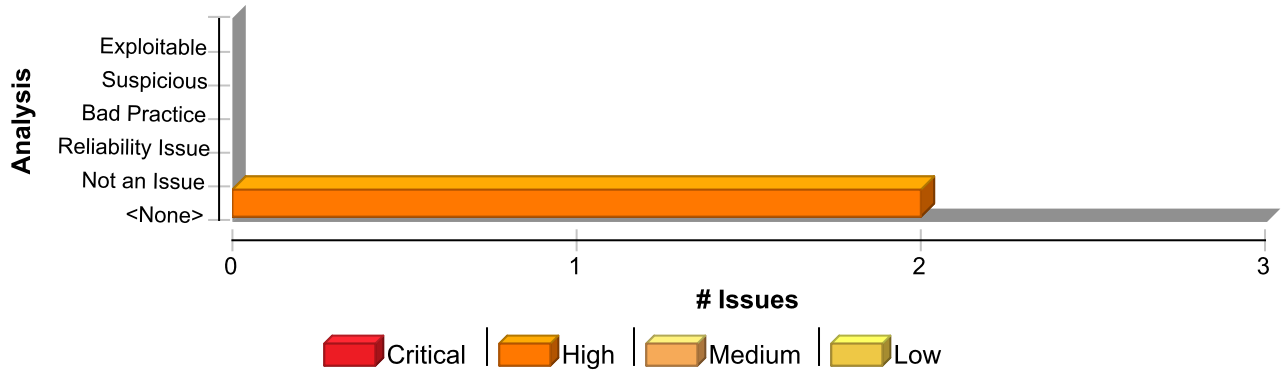
```
(e+)+
([a-zA-Z]+)*
(e|ee)+
```

There are no known regular expression implementations that are immune to this vulnerability. All platforms and languages are vulnerable to this attack.

Recommendation

Do not allow untrusted data to be used as regular expression patterns.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Denial of Service: Regular Expression	2	0	0	2
Total	**2**	**0**	**0**	**2**

Denial of Service: Regular Expression	**High**
Package: akka.remote.testkit	
akka/remote/testkit/LogRoleReplace.scala, line 139 (Denial of Service: Regular Expression)	**High**

Issue Details

Kingdom: Input Validation and Representation
Scan Engine: SCA (Data Flow)

Denial of Service: Regular Expression — High

Package: akka.remote.testkit

akka/remote/testkit/LogRoleReplace.scala, line 139 (Denial of Service: Regular Expression) — High

Source Details

Source: java.io.BufferedReader.readLine()
From: akka.remote.testkit.LogRoleReplace.process
File: akka/remote/testkit/LogRoleReplace.scala:107

```
104  processLines(in.readLine)
105  }
106
107  processLines(in.readLine())
108  }
109
110  def processLine(line: String): String = {
```

Sink Details

Sink: java.lang.String.replaceAll()
Enclosing Method: apply()
File: akka/remote/testkit/LogRoleReplace.scala:139
Taint Flags: NO_NEW_LINE, START_CHECKED_STRING, STREAM, VALIDATED_PORTABILITY_FLAW_FILE_SEPARATOR

```
136  private def replaceLine(line: String): String = {
137  var result = line
138  for ((from, to) <- replacements) {
139  result = result.replaceAll(from, to)
140  }
141  result
142  }
```

akka/remote/testkit/LogRoleReplace.scala, line 139 (Denial of Service: Regular Expression) — High

Issue Details

Kingdom: Input Validation and Representation
Scan Engine: SCA (Data Flow)

Source Details

Source: java.io.BufferedReader.readLine()
From: akka.remote.testkit.LogRoleReplace.processLines
File: akka/remote/testkit/LogRoleReplace.scala:104

```
101  def processLines(line: String): Unit =
102  if (line ne null) {
103  out.println(processLine(line))
104  processLines(in.readLine)
```

Denial of Service: Regular Expression	High
Package: akka.remote.testkit	
akka/remote/testkit/LogRoleReplace.scala, line 139 (Denial of Service: Regular Expression)	**High**

```
105  }
106
107  processLines(in.readLine())
```

Sink Details

Sink: java.lang.String.replaceAll()
Enclosing Method: apply()
File: akka/remote/testkit/LogRoleReplace.scala:139
Taint Flags: NO_NEW_LINE, START_CHECKED_STRING, STREAM, VALIDATED_PORTABILITY_FLAW_FILE_SEPARATOR

```
136  private def replaceLine(line: String): String = {
137  var result = line
138  for ((from, to) <- replacements) {
139  result = result.replaceAll(from, to)
140  }
141  result
142  }
```

J2EE Bad Practices: Leftover Debug Code (1 issue)

Abstract

Debug code can create unintended entry points in a deployed web application.

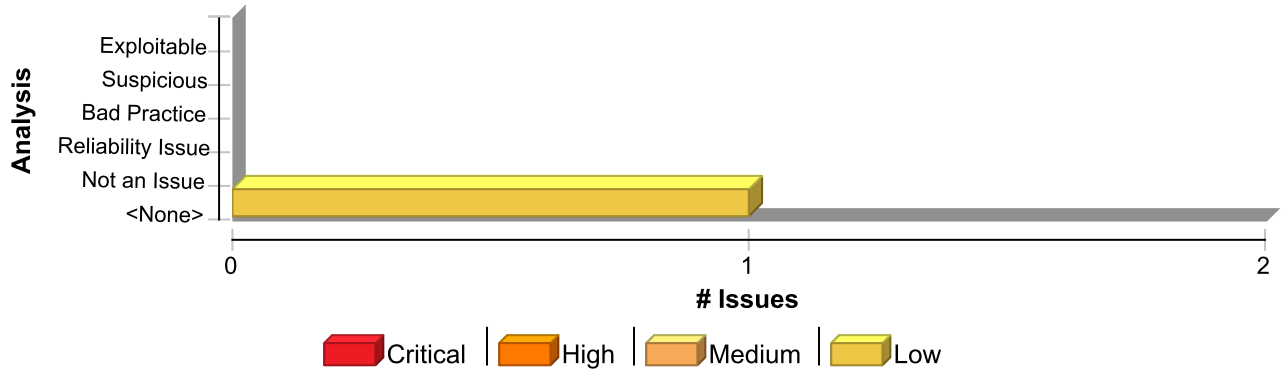
Explanation

A common development practice is to add "back door" code specifically designed for debugging or testing purposes that is not intended to be shipped or deployed with the application. When this sort of debug code is accidentally left in the application, the application is open to unintended modes of interaction. These back door entry points create security risks because they are not considered during design or testing and fall outside of the expected operating conditions of the application. The most common example of forgotten debug code is a `main()` method appearing in a web application. Although this is an acceptable practice during product development, classes that are part of a production J2EE application should not define a `main()`.

Recommendation

Remove debug code before deploying a production version of an application. Regardless of whether a direct security threat can be articulated, it is unlikely that there is a legitimate reason for such code to remain in the application after the early stages of development.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
J2EE Bad Practices: Leftover Debug Code	1	0	0	1
Total	**1**	**0**	**0**	**1**

J2EE Bad Practices: Leftover Debug Code	Low
Package: akka.remote.testkit	
akka/remote/testkit/LogRoleReplace.scala, line 45 (J2EE Bad Practices: Leftover Debug Code)	**Low**

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Structural)

Sink Details

J2EE Bad Practices: Leftover Debug Code	Low

Package: akka.remote.testkit

akka/remote/testkit/LogRoleReplace.scala, line 45 (J2EE Bad Practices: Leftover Debug Code) — Low

Sink: Function: main
Enclosing Method: main()
File: akka/remote/testkit/LogRoleReplace.scala:45
Taint Flags:

```
42 * You can also replace the contents of the clipboard instead of using files
43 * by supplying `clipboard` as argument
44 */
45 def main(args: Array[String]): Unit = {
46 val replacer = new LogRoleReplace
47
48 if (args.length == 0) {
```

J2EE Bad Practices: Sockets (2 issues)

Abstract

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

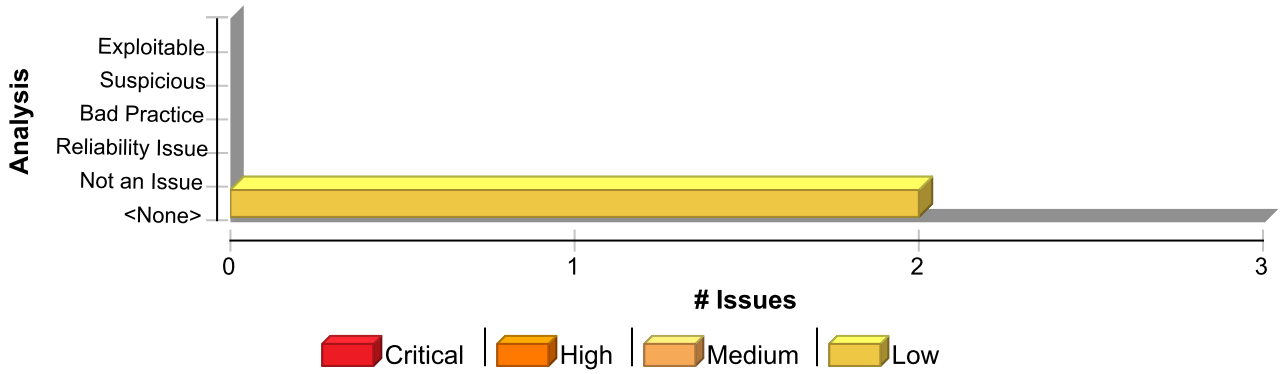
Explanation

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	2	0	0	2
Total	**2**	**0**	**0**	**2**

J2EE Bad Practices: Sockets	Low
Package: akka.remote.testconductor	
akka/remote/testconductor/BarrierSpec.scala, line 549 (J2EE Bad Practices: Sockets)	**Low**

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Semantic)

Sink Details

J2EE Bad Practices: Sockets — Low

Package: akka.remote.testconductor

akka/remote/testconductor/BarrierSpec.scala, line 549 (J2EE Bad Practices: Sockets) — Low

Sink: InetSocketAddress()
Enclosing Method: BarrierSpec$$anon$1()
File: akka/remote/testconductor/BarrierSpec.scala:549
Taint Flags:

```
546 private def withController(participants: Int)(f: ActorRef => Unit): Unit = {
547 system.actorOf(Props(new Actor {
548 val controller =
549 context.actorOf(Props(classOf[Controller], participants, new InetSocketAddress(InetAddress.getLocalHost, 0)))
550 controller ! GetSockAddr
551 override def supervisorStrategy = OneForOneStrategy() {
552 case x => testActor ! Failed(controller, x); SupervisorStrategy.Restart
```

akka/remote/testconductor/ControllerSpec.scala, line 32 (J2EE Bad Practices: Sockets) — Low

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Semantic)

Sink Details

Sink: InetSocketAddress()
Enclosing Method: apply()
File: akka/remote/testconductor/ControllerSpec.scala:32
Taint Flags:

```
29 "A Controller" must {
30
31 "publish its nodes" in {
32 val c = system.actorOf(Props(classOf[Controller], 1, new InetSocketAddress(InetAddress.getLocalHost, 0)))
33 c ! NodeInfo(A, AddressFromURIString("akka://sys"), testActor)
34 expectMsg(ToClient(Done))
35 c ! NodeInfo(B, AddressFromURIString("akka://sys"), testActor)
```

J2EE Bad Practices: Threads (4 issues)

Abstract

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

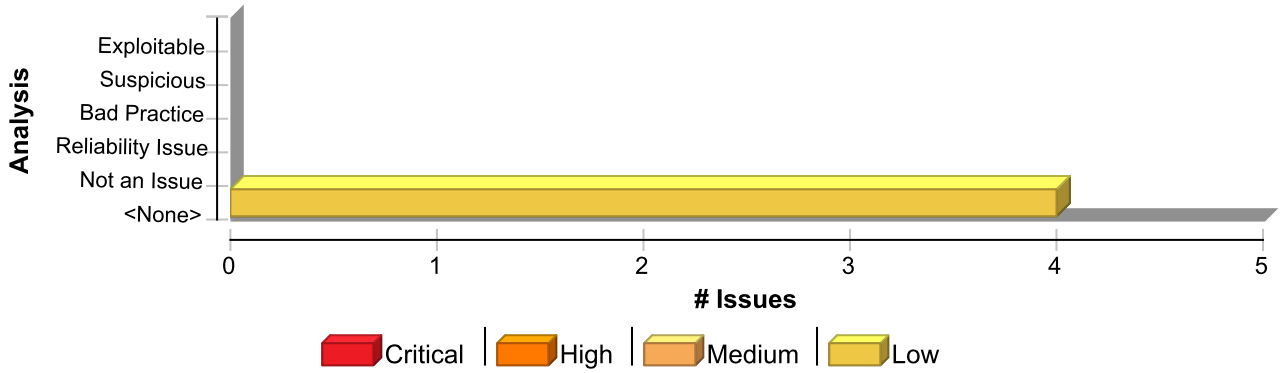
Explanation

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

J2EE Bad Practices: Threads	Low
Package: akka.remote.testconductor	
akka/remote/testconductor/BarrierSpec.scala, line 268 (J2EE Bad Practices: Threads)	**Low**

Issue Details

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

Sink Details

Sink: sleep()
Enclosing Method: apply()
File: akka/remote/testconductor/BarrierSpec.scala:268
Taint Flags:

J2EE Bad Practices: Threads — Low

Package: akka.remote.testconductor

akka/remote/testconductor/BarrierSpec.scala, line 268 (J2EE Bad Practices: Threads) — Low

```
265 EventFilter.warning(start = "cannot remove", occurrences = 1).intercept {
266 b ! Remove(A)
267 }
268 Thread.sleep(5000)
269 }
270 }
271
```

akka/remote/testconductor/BarrierSpec.scala, line 531 (J2EE Bad Practices: Threads) — Low

Issue Details

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

Sink Details

Sink: sleep()
Enclosing Method: apply()
File: akka/remote/testconductor/BarrierSpec.scala:531
Taint Flags:

```
528 a.send(barrier, EnterBarrier("bar23", Option(2 seconds)))
529 b.send(barrier, EnterBarrier("bar23", Option(10 seconds)))
530 EventFilter[BarrierTimeout](occurrences = 1).intercept {
531 Thread.sleep(4000)
532 }
533 c.send(barrier, EnterBarrier("bar23", None))
534 a.expectMsg(ToClient(BarrierResult("bar23", false)))
```

akka/remote/testconductor/BarrierSpec.scala, line 507 (J2EE Bad Practices: Threads) — Low

Issue Details

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

Sink Details

Sink: sleep()
Enclosing Method: apply()
File: akka/remote/testconductor/BarrierSpec.scala:507
Taint Flags:

```
504 a.send(barrier, EnterBarrier("bar22", Option(10 seconds)))
505 b.send(barrier, EnterBarrier("bar22", Option(2 seconds)))
506 EventFilter[BarrierTimeout](occurrences = 1).intercept {
507 Thread.sleep(4000)
508 }
509 c.send(barrier, EnterBarrier("bar22", None))
```

J2EE Bad Practices: Threads	Low

Package: akka.remote.testconductor

akka/remote/testconductor/BarrierSpec.scala, line 507 (J2EE Bad Practices: Threads) **Low**

```
510 a.expectMsg(ToClient(BarrierResult("bar22", false)))
```

akka/remote/testconductor/BarrierSpec.scala, line 432 (J2EE Bad Practices: Threads) **Low**

Issue Details

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

Sink Details

Sink: sleep()
Enclosing Method: apply()
File: akka/remote/testconductor/BarrierSpec.scala:432
Taint Flags:

```
429 b.expectMsg(ToClient(Done))
430 a.send(barrier, EnterBarrier("bar18", Option(2 seconds)))
431 EventFilter[BarrierTimeout](occurrences = 1).intercept {
432 Thread.sleep(4000)
433 }
434 b.send(barrier, EnterBarrier("bar18", None))
435 a.expectMsg(ToClient(BarrierResult("bar18", false)))
```

Missing Check against Null (2 issues)

Abstract

The program can dereference a null-pointer because it does not check the return value of a function that might return `null`.

Explanation

Just about every serious attack on a software system begins with the violation of a programmer's assumptions. After the attack, the programmer's assumptions seem flimsy and poorly founded, but before an attack many programmers would defend their assumptions well past the end of their lunch break. Two dubious assumptions that are easy to spot in code are "this function call can never fail" and "it doesn't matter if this function call fails". When a programmer ignores the return value from a function, they implicitly state that they are operating under one of these assumptions.
Example 1: The following code does not check to see if the string returned by `getParameter()` is `null` before calling the member function `compareTo()`, potentially causing a `null` dereference.

```
String itemName = request.getParameter(ITEM_NAME);
    if (itemName.compareTo(IMPORTANT_ITEM)) {
        ...
    }
    ...
```

Example 2:. The following code shows a system property that is set to `null` and later dereferenced by a programmer who mistakenly assumes it will always be defined.

```
System.clearProperty("os.name");
...
String os = System.getProperty("os.name");
if (os.equalsIgnoreCase("Windows 95") )
    System.out.println("Not supported");
```

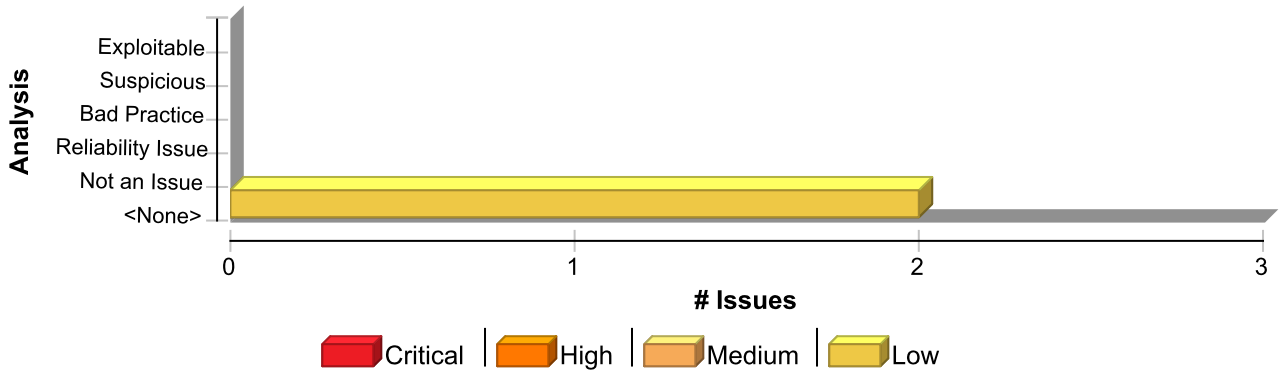
The traditional defense of this coding error is: "I know the requested value will always exist because.... If it does not exist, the program cannot perform the desired behavior so it doesn't matter whether I handle the error or simply allow the program to die dereferencing a `null` value." But attackers are skilled at finding unexpected paths through programs, particularly when exceptions are involved.

Recommendation

If a function can return an error code or any other evidence of its success or failure, always check for the error condition, even if there is no obvious way for it to occur. In addition to preventing security errors, many initially mysterious bugs have eventually led back to a failed method call with an unchecked return value. Create an easy to use and standard way for dealing with failure in your application. If error handling is straightforward, programmers will be less inclined to omit it. One approach to standardized error handling is to write wrappers around commonly-used functions that check and handle error conditions without additional programmer intervention. When wrappers are implemented and adopted, the use of non-wrapped equivalents can be prohibited and enforced by using custom rules.
Example 3: The following code implements a wrapper around `getParameter()` that checks the return value of `getParameter()` against `null` and uses a default value if the requested parameter is not defined.

```
String safeGetParameter (HttpRequest request, String name)
{
    String value = request.getParameter(name);
    if (value == null) {
        return getDefaultValue(name)
    }
    return value;
}
```

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Missing Check against Null	2	0	0	2
Total	**2**	**0**	**0**	**2**

Missing Check against Null — Low

Package: akka.remote.testkit

akka/remote/testkit/LogRoleReplace.scala, line 107 (Missing Check against Null) — Low

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Control Flow)

Sink Details

Sink: readLine() : BufferedReader.readLine may return NULL
Enclosing Method: process()
File: akka/remote/testkit/LogRoleReplace.scala:107
Taint Flags:

```
104 processLines(in.readLine)
105 }
106
107 processLines(in.readLine())
108 }
109
110 def processLine(line: String): String = {
```

akka/remote/testkit/LogRoleReplace.scala, line 104 (Missing Check against Null) — Low

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Control Flow)

Sink Details

Sink: readLine() : BufferedReader.readLine may return NULL

Missing Check against Null	Low
Package: akka.remote.testkit	
akka/remote/testkit/LogRoleReplace.scala, line 104 (Missing Check against Null)	**Low**

Enclosing Method: processLines()
File: akka/remote/testkit/LogRoleReplace.scala:104
Taint Flags:

```
101 def processLines(line: String): Unit =
102 if (line ne null) {
103 out.println(processLine(line))
104 processLines(in.readLine)
105 }
106
107 processLines(in.readLine())
```

Often Misused: Authentication (2 issues)

Abstract

Attackers may spoof DNS entries. Do not rely on DNS names for security.

Explanation

Many DNS servers are susceptible to spoofing attacks, so you should assume that your software will someday run in an environment with a compromised DNS server. If attackers are allowed to make DNS updates (sometimes called DNS cache poisoning), they can route your network traffic through their machines or make it appear as if their IP addresses are part of your domain. Do not base the security of your system on DNS names. **Example:** The following code uses a DNS lookup to determine whether an inbound request is from a trusted host. If an attacker can poison the DNS cache, they can gain trusted status.

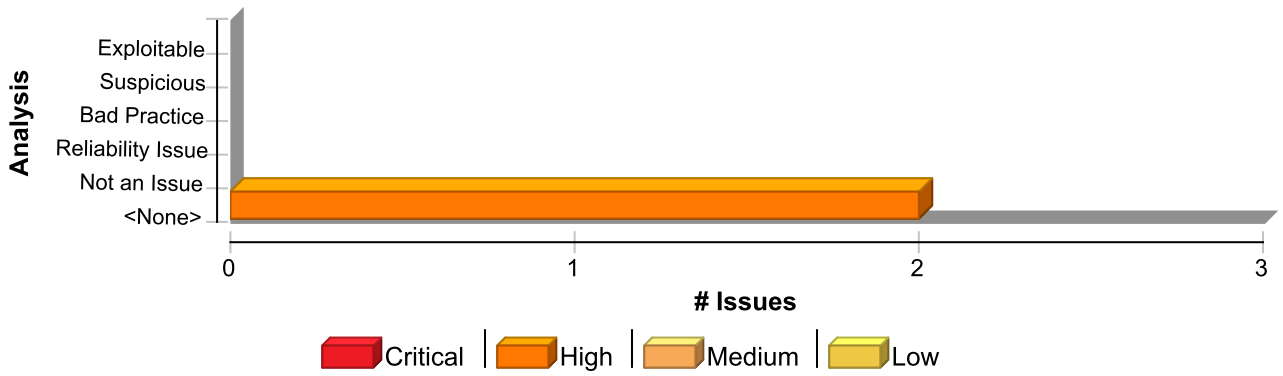
```
String ip = request.getRemoteAddr();
InetAddress addr = InetAddress.getByName(ip);
if (addr.getCanonicalHostName().endsWith("trustme.com")) {
trusted = true;
}
```

IP addresses are more reliable than DNS names, but they can also be spoofed. Attackers may easily forge the source IP address of the packets they send, but response packets will return to the forged IP address. To see the response packets, the attacker has to sniff the traffic between the victim machine and the forged IP address. In order to accomplish the required sniffing, attackers typically attempt to locate themselves on the same subnet as the victim machine. Attackers may be able to circumvent this requirement by using source routing, but source routing is disabled across much of the Internet today. In summary, IP address verification can be a useful part of an authentication scheme, but it should not be the single factor required for authentication.

Recommendation

You can increase confidence in a domain name lookup if you check to make sure that the host's forward and backward DNS entries match. Attackers will not be able to spoof both the forward and the reverse DNS entries without controlling the nameservers for the target domain. This is not a foolproof approach however: attackers may be able to convince the domain registrar to turn over the domain to a malicious nameserver. Basing authentication on DNS entries is simply a risky proposition. While no authentication mechanism is foolproof, there are better alternatives than host-based authentication. Password systems offer decent security, but are susceptible to bad password choices, insecure password transmission, and bad password management. A cryptographic scheme like SSL is worth considering, but such schemes are often so complex that they bring with them the risk of significant implementation errors, and key material can always be stolen. In many situations, multi-factor authentication including a physical token offers the most security available at a reasonable price.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Often Misused: Authentication	2	0	0	2
Total	**2**	**0**	**0**	**2**

Often Misused: Authentication — High

Package: akka.remote.testconductor

akka/remote/testconductor/ControllerSpec.scala, line 32 (Often Misused: Authentication) — High

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Semantic)

Sink Details

Sink: getLocalHost()
Enclosing Method: apply()
File: akka/remote/testconductor/ControllerSpec.scala:32
Taint Flags:

```
29 "A Controller" must {
30
31 "publish its nodes" in {
32 val c = system.actorOf(Props(classOf[Controller], 1, new InetSocketAddress(InetAddress.getLocalHost, 0)))
33 c ! NodeInfo(A, AddressFromURIString("akka://sys"), testActor)
34 expectMsg(ToClient(Done))
35 c ! NodeInfo(B, AddressFromURIString("akka://sys"), testActor)
```

akka/remote/testconductor/BarrierSpec.scala, line 549 (Often Misused: Authentication) — High

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Semantic)

Sink Details

Sink: getLocalHost()
Enclosing Method: BarrierSpec$$anon$1()
File: akka/remote/testconductor/BarrierSpec.scala:549
Taint Flags:

```
546 private def withController(participants: Int)(f: ActorRef => Unit): Unit = {
547 system.actorOf(Props(new Actor {
548 val controller =
549 context.actorOf(Props(classOf[Controller], participants, new InetSocketAddress(InetAddress.getLocalHost, 0)))
550 controller ! GetSockAddr
551 override def supervisorStrategy = OneForOneStrategy() {
552 case x => testActor ! Failed(controller, x); SupervisorStrategy.Restart
```

Often Misused: Boolean.getBoolean() (2 issues)

Abstract

The method `Boolean.getBoolean()` is often confused with `Boolean.valueOf()` or `Boolean.parseBoolean()` method calls.

Explanation

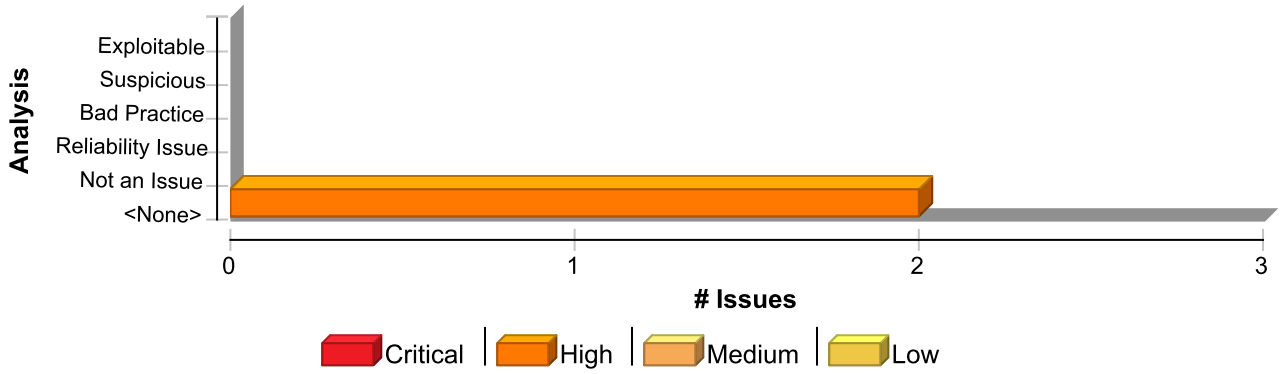
In most cases, a call to `Boolean.getBoolean()` is often misused as it is assumed to return the boolean value represented by the specified string argument. However, as stated in the Javadoc `Boolean.getBoolean(String)` method "Returns true if and only if the system property named by the argument exists and is equal to the string 'true'." Most often what the developer intended to use was a call to `Boolean.valueOf(String)` or `Boolean.parseBoolean(String)` method. **Example 1:** The following code will not behave as expected. It will print "FALSE" as `Boolean.getBoolean(String)` does not translate a String primitive. It only translates system property.

```
...
String isValid = "true";
if ( Boolean.getBoolean(isValid) ) {
    System.out.println("TRUE");
}
else {
    System.out.println("FALSE");
}
...
```

Recommendation

Please ensure that you intend to call the method `Boolean.getBoolean(String)` and the specified string argument is a system property. Else the method call you are most likely looking for is `Boolean.valueOf(String)` or `Boolean.parseBoolean(String)`.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Often Misused: Boolean.getBoolean()	2	0	0	2
Total	**2**	**0**	**0**	**2**

Often Misused: Boolean.getBoolean() — High

Package: org.scalatest.extra

org/scalatest/extra/QuietReporter.scala, line 15 (Often Misused: Boolean.getBoolean()) — High

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Semantic)

Sink Details

Sink: getBoolean()
Enclosing Method: QuietReporter()
File: org/scalatest/extra/QuietReporter.scala:15
Taint Flags:

```
12 class QuietReporter(inColor: Boolean, withDurations: Boolean = false)
13  extends StandardOutReporter(withDurations, inColor, false, true, false, false, false, false, false, false, false) {
14
15  def this() = this(!getBoolean("akka.test.nocolor"), !getBoolean("akka.test.nodurations"))
16
17  override def apply(event: Event): Unit = event match {
18  case _: RunStarting => ()
```

org/scalatest/extra/QuietReporter.scala, line 15 (Often Misused: Boolean.getBoolean()) — High

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Semantic)

Sink Details

Sink: getBoolean()
Enclosing Method: QuietReporter()
File: org/scalatest/extra/QuietReporter.scala:15
Taint Flags:

```
12 class QuietReporter(inColor: Boolean, withDurations: Boolean = false)
13  extends StandardOutReporter(withDurations, inColor, false, true, false, false, false, false, false, false, false) {
14
15  def this() = this(!getBoolean("akka.test.nocolor"), !getBoolean("akka.test.nodurations"))
16
17  override def apply(event: Event): Unit = event match {
18  case _: RunStarting => ()
```

Path Manipulation (3 issues)

Abstract

Allowing user input to control paths used in file system operations could enable an attacker to access or modify otherwise protected system resources.

Explanation

Path manipulation errors occur when the following two conditions are met: 1. An attacker can specify a path used in an operation on the file system. 2. By specifying the resource, the attacker gains a capability that would not otherwise be permitted. For example, the program might give the attacker the ability to overwrite the specified file or run with a configuration controlled by the attacker. **Example 1:** The following code uses input from an HTTP request to create a file name. The programmer has not considered the possibility that an attacker could provide a file name such as "`../../tomcat/conf/server.xml`", which causes the application to delete one of its own configuration files.

```
String rName = request.getParameter("reportName");
File rFile = new File("/usr/local/apfr/reports/" + rName);
...
rFile.delete();
```

Example 2: The following code uses input from a configuration file to determine which file to open and echo back to the user. If the program runs with adequate privileges and malicious users can change the configuration file, they can use the program to read any file on the system that ends with the extension `.txt`.

```
fis = new FileInputStream(cfg.getProperty("sub")+".txt");
amt = fis.read(arr);
out.println(arr);
```

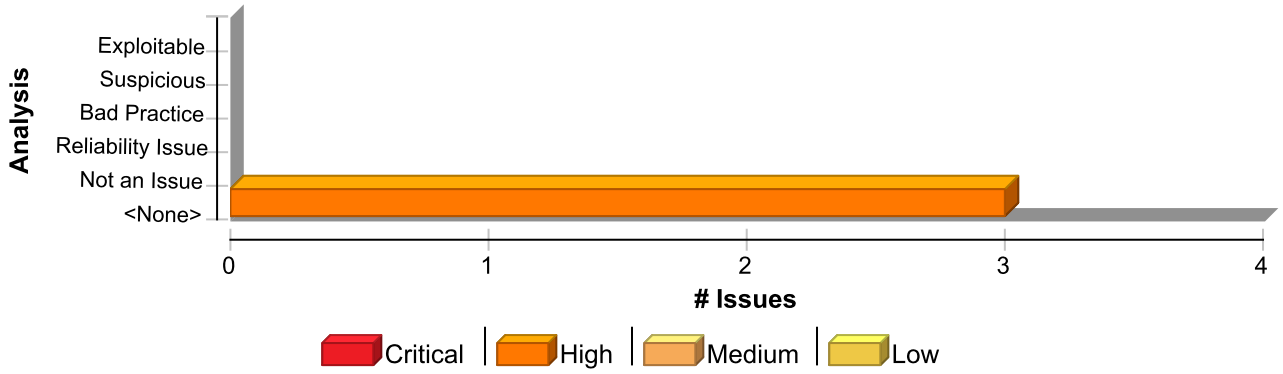
Some think that in the mobile environment, classic vulnerabilities, such as path manipulation, do not make sense -- why would the user attack themself? However, keep in mind that the essence of mobile platforms is applications that are downloaded from various sources and run alongside each other on the same device. The likelihood of running a piece of malware next to a banking application is high, which necessitates expanding the attack surface of mobile applications to include inter-process communication. **Example 3:** The following code adapts `Example 1` to the Android platform.

```
...
        String rName = this.getIntent().getExtras().getString("reportName");
        File rFile = getBaseContext().getFileStreamPath(rName);
...
        rFile.delete();
...
```

Recommendation

The best way to prevent path manipulation is with a level of indirection: create a list of legitimate values from which the user must select. With this approach, the user-provided input is never used directly to specify the resource name. In some situations this approach is impractical because the set of legitimate resource names is too large or too hard to maintain. Programmers often resort to implementing a deny list in these situations. A deny list is used to selectively reject or escape potentially dangerous characters before using the input. However, any such list of unsafe characters is likely to be incomplete and will almost certainly become out of date. A better approach is to create a list of characters that are permitted to appear in the resource name and accept input composed exclusively of characters in the approved set.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Path Manipulation	3	0	0	3
Total	**3**	**0**	**0**	**3**

Path Manipulation — High

Package: akka.remote.testkit

akka/remote/testkit/LogRoleReplace.scala, line 73 (Path Manipulation) — High

Issue Details

Kingdom: Input Validation and Representation
Scan Engine: SCA (Data Flow)

Source Details

Source: main(0)
File: akka/remote/testkit/LogRoleReplace.scala:28

```
25 * Utility to make log files from multi-node tests easier to analyze.
26 * Replaces jvm names and host:port with corresponding logical role name.
27 */
28 object LogRoleReplace extends ClipboardOwner {
29
30 /**
31 * Main program. Use with 0, 1 or 2 arguments.
```

Sink Details

Sink: java.io.FileWriter.FileWriter()
Enclosing Method: main()
File: akka/remote/testkit/LogRoleReplace.scala:73
Taint Flags: ARGS

```
70 }
71
72 } else if (args.length == 2) {
73 val outputFile = new PrintWriter(new FileWriter(args(1)))
74 val inputFile = new BufferedReader(new FileReader(args(0)))
75 try {
```

Path Manipulation — High

Package: akka.remote.testkit

akka/remote/testkit/LogRoleReplace.scala, line 73 (Path Manipulation) — High

```
76 replacer.process(inputFile, outputFile)
```

akka/remote/testkit/LogRoleReplace.scala, line 65 (Path Manipulation) — High

Issue Details

Kingdom: Input Validation and Representation
Scan Engine: SCA (Data Flow)

Source Details

Source: main(0)
File: akka/remote/testkit/LogRoleReplace.scala:28

```
25 * Utility to make log files from multi-node tests easier to analyze.
26 * Replaces jvm names and host:port with corresponding logical role name.
27 */
28 object LogRoleReplace extends ClipboardOwner {
29
30 /**
31 * Main program. Use with 0, 1 or 2 arguments.
```

Sink Details

Sink: java.io.FileReader.FileReader()
Enclosing Method: main()
File: akka/remote/testkit/LogRoleReplace.scala:65
Taint Flags: ARGS

```
62 }
63
64 } else if (args.length == 1) {
65 val inputFile = new BufferedReader(new FileReader(args(0)))
66 try {
67 replacer.process(inputFile, new PrintWriter(new OutputStreamWriter(System.out)))
68 } finally {
```

akka/remote/testkit/LogRoleReplace.scala, line 74 (Path Manipulation) — High

Issue Details

Kingdom: Input Validation and Representation
Scan Engine: SCA (Data Flow)

Source Details

Source: main(0)
File: akka/remote/testkit/LogRoleReplace.scala:28

```
25 * Utility to make log files from multi-node tests easier to analyze.
```

Path Manipulation	High
Package: akka.remote.testkit	
akka/remote/testkit/LogRoleReplace.scala, line 74 (Path Manipulation)	**High**

```
26  * Replaces jvm names and host:port with corresponding logical role name.
27  */
28 object LogRoleReplace extends ClipboardOwner {
29
30  /**
31  * Main program. Use with 0, 1 or 2 arguments.
```

Sink Details

Sink: java.io.FileReader.FileReader()
Enclosing Method: main()
File: akka/remote/testkit/LogRoleReplace.scala:74
Taint Flags: ARGS

```
71
72  } else if (args.length == 2) {
73  val outputFile = new PrintWriter(new FileWriter(args(1)))
74  val inputFile = new BufferedReader(new FileReader(args(0)))
75  try {
76  replacer.process(inputFile, outputFile)
77  } finally {
```