Fortify Standalone Report Generator

Developer Workbook

akka-discovery

Table of Contents

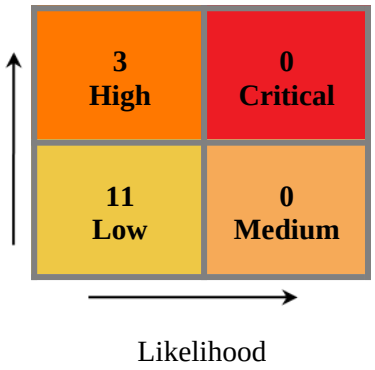
Executive Summary

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-discovery 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-discovery
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, 11:25 AM	**Engine Version:**	21.1.1.0009
Host Name:	Jacks-Work-MBP.local	**Certification:**	VALID
Number of Files:	5	**Lines of Code:**	280

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 / 7	0 / 7
Code Correctness: Erroneous String Compare	0	0	0	0 / 1	0 / 1
Code Correctness: Non-Static Inner Class Implements Serializable	0	0	0	0 / 1	0 / 1
Often Misused: Authentication	0	0 / 3	0	0	0 / 3
Poor Error Handling: Empty Catch Block	0	0	0	0 / 1	0 / 1
Redundant Null Check	0	0	0	0 / 1	0 / 1

Results Outline

Code Correctness: Constructor Invokes Overridable Function (7 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");
    }
  }
}
```

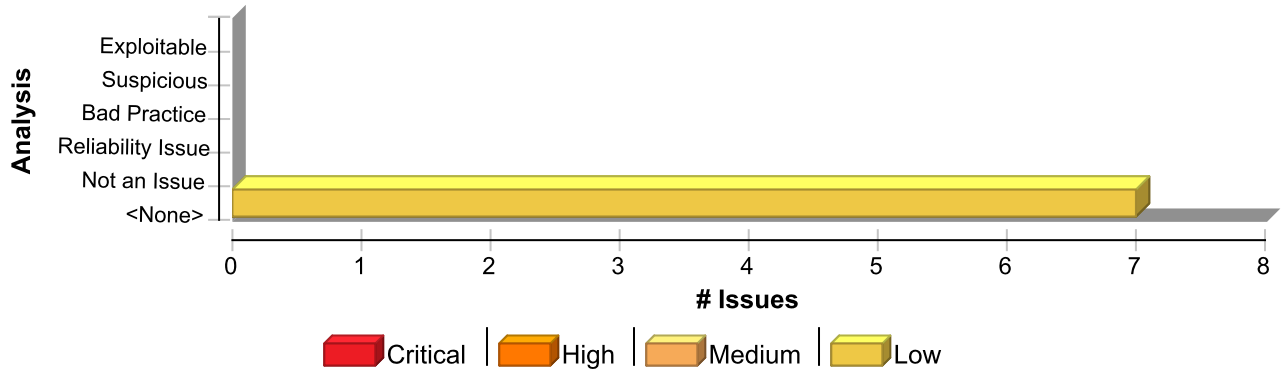
The code in `Example 1` prints "Attack successful!", since the `Attacker` class overrides the `validateUser()` function that is called from the constructor of the superclass `User`, and Java will first look in the subclass for functions called from the constructor.

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	7	0	0	7
Total	**7**	**0**	**0**	**7**

Code Correctness: Constructor Invokes Overridable Function	**Low**
Package: akka.discovery	
Discovery.scala, line 17 (Code Correctness: Constructor Invokes Overridable Function)	**Low**
Issue Details	

Code Correctness: Constructor Invokes Overridable Function — Low

Package: akka.discovery

Discovery.scala, line 17 (Code Correctness: Constructor Invokes Overridable Function) — Low

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionCall: checkClassPathForOldDiscovery
Enclosing Method: Discovery()
File: Discovery.scala:17
Taint Flags:

```
14
15 final class Discovery(implicit system: ExtendedActorSystem) extends Extension {
16
17  Discovery.checkClassPathForOldDiscovery(system)
18
19  private val implementations = new ConcurrentHashMap[String, ServiceDiscovery]
20  private val factory = new JFunction[String, ServiceDiscovery] {
```

Package: akka.discovery.aggregate

aggregate/AggregateServiceDiscovery.scala, line 56 (Code Correctness: Constructor Invokes Overridable Function) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionCall: settings
Enclosing Method: AggregateServiceDiscovery()
File: aggregate/AggregateServiceDiscovery.scala:56
Taint Flags:

```
53  private val settings =
54  new AggregateServiceDiscoverySettings(system.settings.config.getConfig("akka.discovery.aggregate"))
55
56  private val methods = {
57  val serviceDiscovery = Discovery(system)
58  settings.discoveryMethods.map(mech => (mech, serviceDiscovery.loadServiceDiscovery(mech)))
59  }
```

Package: akka.discovery.config

config/ConfigServiceDiscovery.scala, line 58 (Code Correctness: Constructor Invokes Overridable Function) — Low

Issue Details

Kingdom: Code Quality

Code Correctness: Constructor Invokes Overridable Function — Low

Package: akka.discovery.config

config/ConfigServiceDiscovery.scala, line 58 (Code Correctness: Constructor Invokes Overridable Function) — Low

Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionCall: log
Enclosing Method: ConfigServiceDiscovery()
File: config/ConfigServiceDiscovery.scala:58
Taint Flags:

```
55 private val resolvedServices = ConfigServicesParser.parse(
56 system.settings.config.getConfig(system.settings.config.getString("akka.discovery.config.services-path")))
57
58 log.debug("Config discovery serving: {}", resolvedServices)
59
60 override def lookup(lookup: Lookup, resolveTimeout: FiniteDuration): Future[Resolved] = {
61 Future.successful(resolvedServices.getOrElse(lookup.serviceName, Resolved(lookup.serviceName, Nil)))
```

config/ConfigServiceDiscovery.scala, line 58 (Code Correctness: Constructor Invokes Overridable Function) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionCall: resolvedServices
Enclosing Method: ConfigServiceDiscovery()
File: config/ConfigServiceDiscovery.scala:58
Taint Flags:

```
55 private val resolvedServices = ConfigServicesParser.parse(
56 system.settings.config.getConfig(system.settings.config.getString("akka.discovery.config.services-path")))
57
58 log.debug("Config discovery serving: {}", resolvedServices)
59
60 override def lookup(lookup: Lookup, resolveTimeout: FiniteDuration): Future[Resolved] = {
61 Future.successful(resolvedServices.getOrElse(lookup.serviceName, Resolved(lookup.serviceName, Nil)))
```

Package: akka.discovery.dns

dns/DnsServiceDiscovery.scala, line 94 (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.discovery.dns

dns/DnsServiceDiscovery.scala, line 94 (Code Correctness: Constructor Invokes Overridable Function) — Low

Sink Details

Sink: FunctionCall: ec
Enclosing Method: DnsServiceDiscovery()
File: dns/DnsServiceDiscovery.scala:94
Taint Flags:

```
91
92  private implicit val ec: MessageDispatcher = system.dispatchers.internalDispatcher
93
94  dns.ask(AsyncDnsManager.GetCache)(Timeout(30.seconds)).onComplete {
95  case Success(cache: SimpleDnsCache) =>
96  asyncDnsCache = OptionVal.Some(cache)
97  case Success(other) =>
```

dns/DnsServiceDiscovery.scala, line 94 (Code Correctness: Constructor Invokes Overridable Function) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionCall: dns
Enclosing Method: DnsServiceDiscovery()
File: dns/DnsServiceDiscovery.scala:94
Taint Flags:

```
91
92  private implicit val ec: MessageDispatcher = system.dispatchers.internalDispatcher
93
94  dns.ask(AsyncDnsManager.GetCache)(Timeout(30.seconds)).onComplete {
95  case Success(cache: SimpleDnsCache) =>
96  asyncDnsCache = OptionVal.Some(cache)
97  case Success(other) =>
```

dns/DnsServiceDiscovery.scala, line 75 (Code Correctness: Constructor Invokes Overridable Function) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: FunctionCall: initializeDns
Enclosing Method: DnsServiceDiscovery()

Code Correctness: Constructor Invokes Overridable Function	Low
Package: akka.discovery.dns	
dns/DnsServiceDiscovery.scala, line 75 (Code Correctness: Constructor Invokes Overridable Function)	**Low**

File: dns/DnsServiceDiscovery.scala:75
Taint Flags:

```
72 import ServiceDiscovery._
73
74 private val log = Logging(system, classOf[DnsServiceDiscovery])
75 private val dns = initializeDns()
76
77 // exposed for testing
78 private[dns] def initializeDns(): ActorRef = {
```

Code Correctness: Erroneous String Compare (1 issue)

Abstract

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

Explanation

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

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

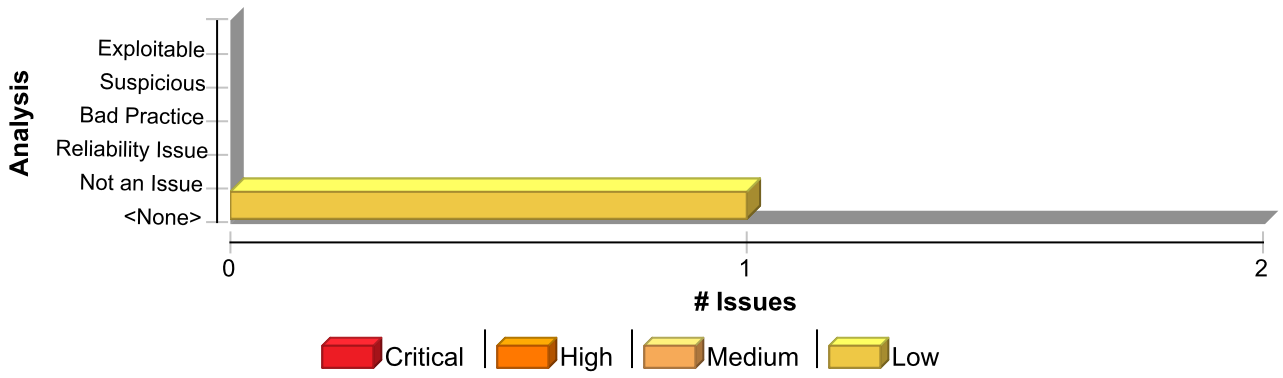
The `==` and `!=` operators will only behave as expected when they are used to compare strings contained in objects that are equal. The most common way for this to occur is for the strings to be interned, whereby the strings are added to a pool of objects maintained by the `String` class. Once a string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

Recommendation

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

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

Issue Summary



Engine Breakdown

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

Code Correctness: Erroneous String Compare — Low

Package: akka.discovery

Discovery.scala, line 25 (Code Correctness: Erroneous String Compare) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: Operation
Enclosing Method: _defaultImplMethod$lzycompute()
File: Discovery.scala:25
Taint Flags:

```
22 }
23
24 private lazy val _defaultImplMethod =
25 system.settings.config.getString("akka.discovery.method") match {
26 case "<method>" =>
27 throw new IllegalArgumentException(
28 "No default service discovery implementation configured in " +
```

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

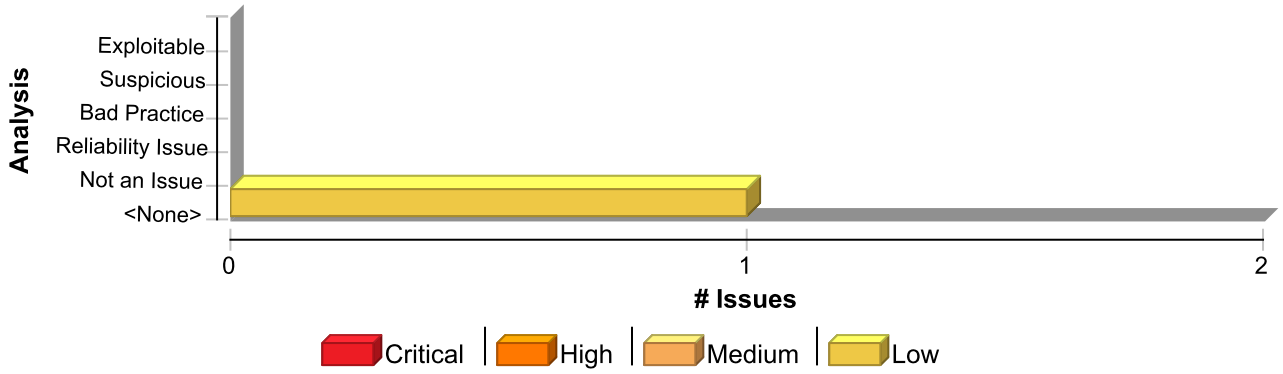
When using inner classes, they should not be serialized, or they should be changed to static-nested classes, since these do not have the drawbacks that non-static inner classes have when serialized. When a nested class is static it inherently has no association with instance variables (including those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

ServiceDiscovery.scala, line 28 (Code Correctness: Non-Static Inner Class Implements Serializable) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: Class: ServiceDiscovery$DiscoveryTimeoutException
File: ServiceDiscovery.scala:28
Taint Flags:

```
25 *
26 * It is up to each implementation to implement timeouts.
27 */
28 final class DiscoveryTimeoutException(reason: String) extends RuntimeException(reason)
29
30 object Resolved {
31 def apply(serviceName: String, addresses: immutable.Seq[ResolvedTarget]): Resolved =
```

Often Misused: Authentication (3 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.

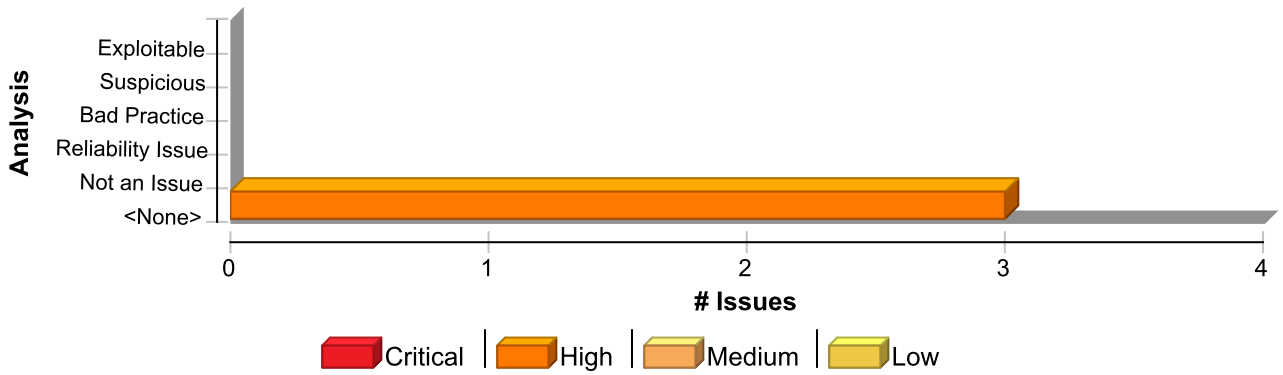
```
String ip = request.getRemoteAddr();
InetAddress addr = InetAddress.getByName(ip);
if (addr.getCanonicalHostName().endsWith("trustme.com")) {
trusted = true;
}
```

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	3	0	0	3
Total	**3**	**0**	**0**	**3**

Often Misused: Authentication — High

Package: <none>

ServiceDiscovery.scala, line 73 (Often Misused: Authentication) — High

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Semantic)

Sink Details

Sink: getAddress()
Enclosing Method: apply()
File: ServiceDiscovery.scala:73
Taint Flags:

```
70 import scala.annotation.nowarn
71 @nowarn("msg=deprecated")
72 private implicit val inetAddressOrdering: Ordering[InetAddress] =
73 Ordering.by[InetAddress, Iterable[Byte]](_.getAddress)
74
75 implicit val addressOrdering: Ordering[ResolvedTarget] = Ordering.by { t =>
76 (t.address, t.host, t.port)
```

Package: akka.discovery.dns

dns/DnsServiceDiscovery.scala, line 160 (Often Misused: Authentication) — High

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Semantic)

Sink Details

Sink: getHostAddress()
Enclosing Method: applyOrElse()
File: dns/DnsServiceDiscovery.scala:160
Taint Flags:

```
157 def ipRecordsToResolved(resolved: DnsProtocol.Resolved): Resolved = {
158 val addresses = resolved.records.collect {
159 case a: ARecord => ResolvedTarget(cleanIpString(a.ip.getHostAddress), None, Some(a.ip))
160 case a: AAAARecord => ResolvedTarget(cleanIpString(a.ip.getHostAddress), None, Some(a.ip))
161 }
162 Resolved(lookup.serviceName, addresses)
163 }
```

Often Misused: Authentication	High
Package: akka.discovery.dns	
dns/DnsServiceDiscovery.scala, line 159 (Often Misused: Authentication)	**High**

Issue Details

Kingdom: API Abuse
Scan Engine: SCA (Semantic)

Sink Details

Sink: getHostAddress()
Enclosing Method: applyOrElse()
File: dns/DnsServiceDiscovery.scala:159
Taint Flags:

```
156
157 def ipRecordsToResolved(resolved: DnsProtocol.Resolved): Resolved = {
158 val addresses = resolved.records.collect {
159 case a: ARecord => ResolvedTarget(cleanIpString(a.ip.getHostAddress), None, Some(a.ip))
160 case a: AAAARecord => ResolvedTarget(cleanIpString(a.ip.getHostAddress), None, Some(a.ip))
161 }
162 Resolved(lookup.serviceName, addresses)
```

Poor Error Handling: Empty Catch Block (1 issue)

Abstract

Ignoring an exception can cause the program to overlook unexpected states and conditions.

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 method call can never fail" and "it doesn't matter if this call fails". When a programmer ignores an exception, they implicitly state that they are operating under one of these assumptions. **Example 1:** The following code excerpt ignores a rarely-thrown exception from `doExchange()`.

```
try {
  doExchange();
}
catch (RareException e) {
  // this can never happen
}
```

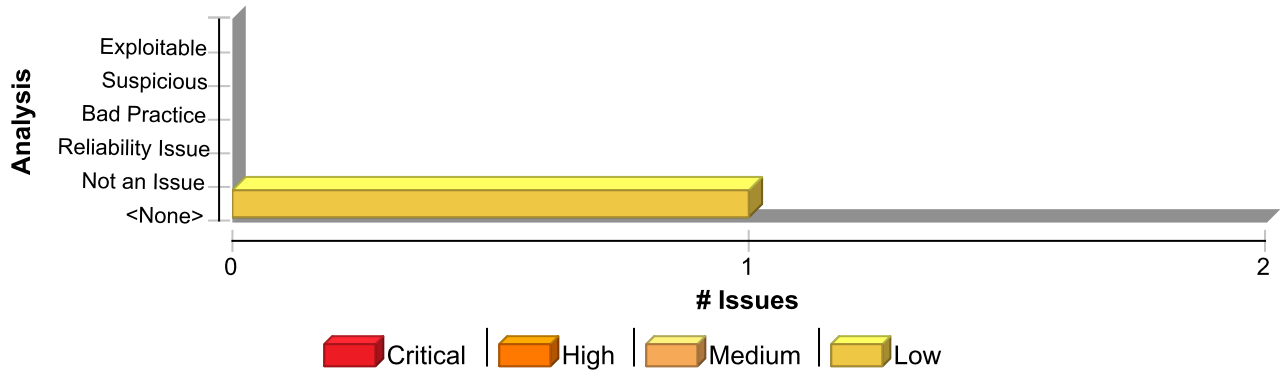
If a `RareException` were to ever be thrown, the program would continue to execute as though nothing unusual had occurred. The program records no evidence indicating the special situation, potentially frustrating any later attempt to explain the program's behavior.

Recommendation

At a minimum, log the fact that the exception was thrown so that it will be possible to come back later and make sense of the resulting program behavior. Better yet, abort the current operation. If the exception is being ignored because the caller cannot properly handle it but the context makes it inconvenient or impossible for the caller to declare that it throws the exception itself, consider throwing a `RuntimeException` or an `Error`, both of which are unchecked exceptions. As of JDK 1.4, `RuntimeException` has a constructor that makes it easy to wrap another exception. **Example 2:** The code in `Example 1` could be rewritten in the following way:

```
try {
  doExchange();
}
catch (RareException e) {
  throw new RuntimeException("This can never happen", e);
}
```

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Poor Error Handling: Empty Catch Block	1	0	0	1
Total	**1**	**0**	**0**	**1**

Poor Error Handling: Empty Catch Block — Low

Package: akka.discovery

Discovery.scala, line 123 (Poor Error Handling: Empty Catch Block) — Low

Issue Details

Kingdom: Errors
Scan Engine: SCA (Structural)

Sink Details

Sink: CatchBlock
Enclosing Method: checkClassPathForOldDiscovery()
File: Discovery.scala:123
Taint Flags:

```
120 case _: ClassCastException =>
121 throw new RuntimeException(
122 "Old version of Akka Discovery from Akka Management found on the classpath. Remove `com.lightbend.akka.discovery:akka-discovery` from the classpath..")
123 case _: ClassNotFoundException =>
124 // all good
125 }
126 }
```

Redundant Null Check (1 issue)

Abstract

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

Explanation

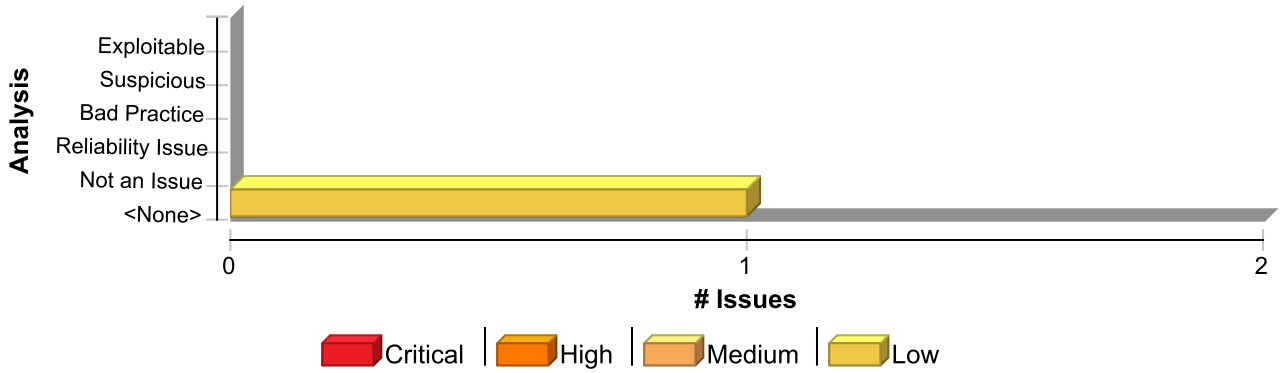
Null-pointer exceptions usually occur when one or more of the programmer's assumptions is violated. Specifically, dereference-after-check errors occur when a program makes an explicit check for `null`, but proceeds to dereference the object when it is known to be `null`. Errors of this type are often 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.discovery

ServiceDiscovery.scala, line 140 (Redundant Null Check) Low

Issue Details

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

Sink Details

Sink: Dereferenced : serviceName
Enclosing Method: Lookup()
File: ServiceDiscovery.scala:140
Taint Flags:

```
137 extends NoSerializationVerificationNeeded {
138
139 require(serviceName != null, "'serviceName' cannot be null")
140 require(serviceName.trim.nonEmpty, "'serviceName' cannot be empty")
141
142 /**
143 * Which port for a service e.g. Akka remoting or HTTP.
```