TURING-GALLERY-BACK

VERSION 1.0

CODE ANALYSIS

By: default

2020-02-10

CONTENT

INTRODUCTION

This document contains results of the code analysis of TURING-GALLERY-BACK.

CONFIGURATION

- Quality Profiles
  - Names: Sonar way [Java]; Sonar way [HTML];
  - Files: AW-vB_RS6MX5uumeTZYc.json; AW-vB_Ue6MX5uumeTZfB.json;
- Quality Gate
  - Name: Sonar way
  - File: Sonar way.xml

SYNTHESIS

Quality Gate	Reliability	Security	Maintainability	Coverage	Duplication
ERROR	D	A	A	0.0 %	0.0 %

METRICS

	Cyclomatic Complexity	Cognitive Complexity	Lines of code per file	Comment density (%)	Coverage	Duplication (%)
Min	0.0	0.0	6.0	0.0	0.0	0.0
Max	89.0	29.0	744.0	32.7	0.0	0.0

TURING-GALLERY-BACK

VOLUME

Language	Number
Java	735
HTML	9
Total	744

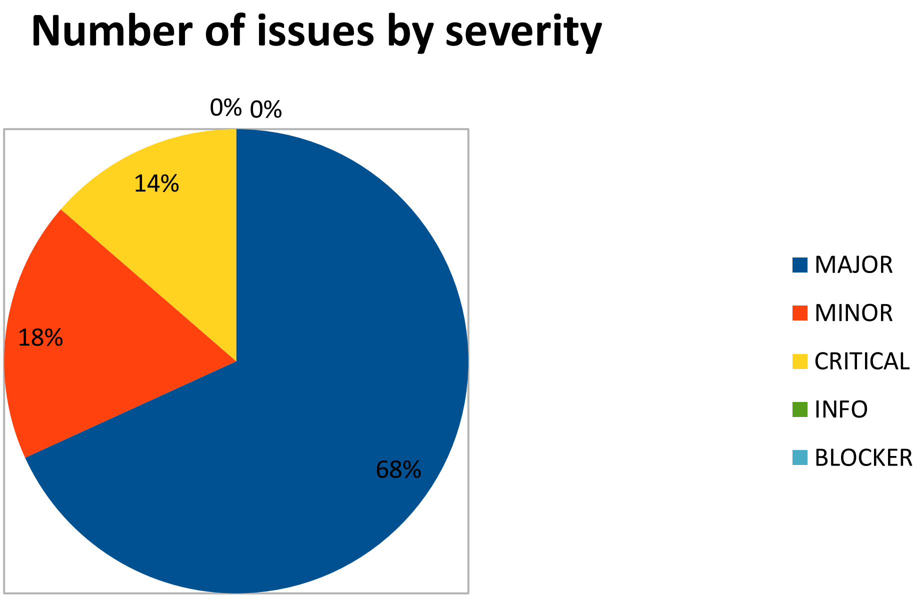
ISSUES COUNT BY SEVERITY AND TYPE

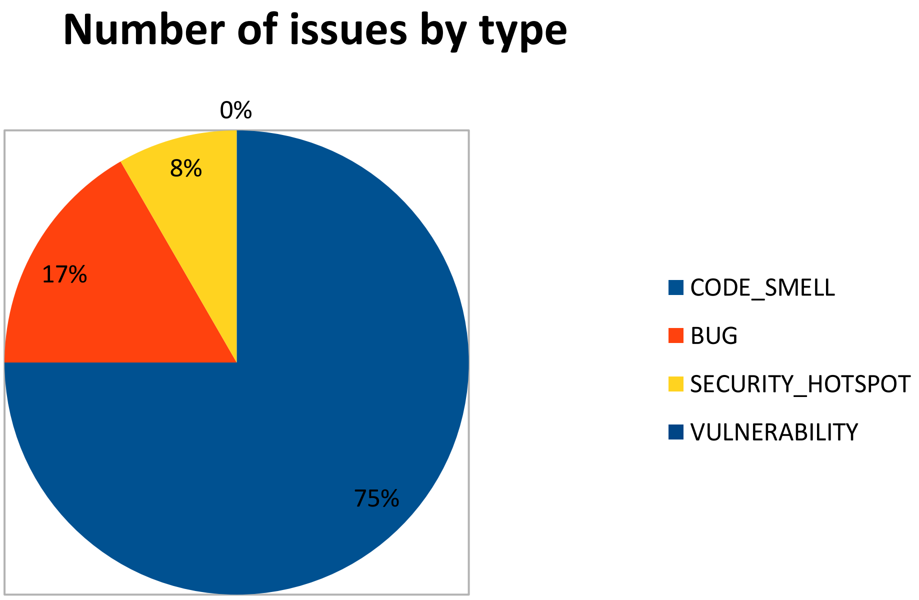
Type	Severity	Number
VULNERABILITY	BLOCKER	0
VULNERABILITY	CRITICAL	0
VULNERABILITY	MAJOR	0
VULNERABILITY	MINOR	0
VULNERABILITY	INFO	0
BUG	BLOCKER	0
BUG	CRITICAL	1
BUG	MAJOR	2
BUG	MINOR	1
BUG	INFO	0
CODE_SMELL	BLOCKER	0
CODE_SMELL	CRITICAL	2
CODE_SMELL	MAJOR	13
CODE_SMELL	MINOR	3
CODE_SMELL	INFO	0

TURING-GALLERY-BACK

SECURITY_HOTSPOT	BLOCKER	0
SECURITY_HOTSPOT	CRITICAL	0
SECURITY_HOTSPOT	MAJOR	0
SECURITY_HOTSPOT	MINOR	0
SECURITY_HOTSPOT	INFO	0

CHARTS





ISSUES

Name	Description	Type	Severity	Number
"Random" objects should be reused	Creating a new Random object each time a random value is needed is inefficient and may produce numbers which are not random depending on the JDK. For better efficiency and randomness, create a single Random, then store, and reuse it. The Random() constructor tries to set the seed with a distinct value every time. However there is no guarantee that the seed will be random or even uniformly distributed. Some JDK will use the current time as seed, which makes the generated numbers	BUG	CRITICAL	1

	not random at all. This rule finds cases where a new Random is created each time a method is invoked and assigned to a local random variable. Noncompliant Code Example public void doSomethingCommon() { Random rand = new Random(); // Noncompliant; new instance created with each invocation int rValue = rand.nextInt(); //... Compliant Solution private Random rand = SecureRandom.getInstanceStrong(); // SecureRandom is preferred to Random public void doSomethingCommon() { int rValue = this.rand.nextInt(); //... Exceptions A class which uses a Random in its constructor or in a static main function and nowhere else will be ignored by this rule. See OWASP Top 10 2017 Category A6 - Security Misconfiguration			
"null" should not be used with "Optional"	The concept of Optional is that it will be used when null could cause errors. In a way, it replaces null, and when Optional is in use, there should never be a question of returning or receiving null from a call. Noncompliant Code Example public void doSomething () { Optional&lt;String&gt; optional = getOptional(); if (optional != null) { // Noncompliant // do something with optional... } } @Nullable // Noncompliant public Optional&lt;String&gt; getOptional() { // ... return null; // Noncompliant } Compliant Solution public void doSomething () { Optional&lt;String&gt; optional = getOptional(); optional.ifPresent(// do something with optional...); } public Optional&lt;String&gt; getOptional() { // ... return Optional.empty(); }	BUG	MAJOR	1
Optional value should only be accessed after calling isPresent()	Optional value can hold either a value or not. The value held in the Optional can be accessed using the get() method, but it will throw a NoSuchElementException if there is no value present. To avoid the exception, calling the isPresent() or ! isEmpty() method should always be done before any call to get(). Alternatively, note that other methods such as orElse(...), orElseGet(...) or orElseThrow(...) can be used to specify what to do with an empty Optional. Noncompliant Code Example Optional&lt;String&gt; value = this.getOptionalValue(); // ... String stringValue = value.get(); // Noncompliant Compliant Solution Optional&lt;String&gt; value = this.getOptionalValue(); // ... if (value.isPresent()) { String stringValue = value.get(); } or Optional&lt;String&gt; value =	BUG	MAJOR	1

	this.getOptionalValue(); // ... String stringValue = value.orElse("default"); See MITRE, CWE-476 - NULL Pointer Dereference			
Math operands should be cast before assignment	When arithmetic is performed on integers, the result will always be an integer. You can assign that result to a long, double, or float with automatic type conversion, but having started as an int or long, the result will likely not be what you expect. For instance, if the result of int division is assigned to a floating-point variable, precision will have been lost before the assignment. Likewise, if the result of multiplication is assigned to a long, it may have already overflowed before the assignment. In either case, the result will not be what was expected. Instead, at least one operand should be cast or promoted to the final type before the operation takes place. Noncompliant Code Example float twoThirds = 2/3; // Noncompliant; int division. Yields 0.0 long millisInYear = 1_000*3_600*24*365; // Noncompliant; int multiplication. Yields 1471228928 long bigNum = Integer.MAX_VALUE + 2; // Noncompliant. Yields -2147483647 long bigNegNum = Integer.MIN_VALUE-1; //Noncompliant, gives a positive result instead of a negative one. Date myDate = new Date(seconds * 1_000); //Noncompliant, won't produce the expected result if seconds &gt; 2_147_483 ... public long compute(int factor){ return factor * 10_000; //Noncompliant, won't produce the expected result if factor &gt; 214_748 } public float compute2(long factor){ return factor / 123; //Noncompliant, will be rounded to closest long integer } Compliant Solution float twoThirds = 2f/3; // 2 promoted to float. Yields 0.6666667 long millisInYear = 1_000L*3_600*24*365; // 1000 promoted to long. Yields 31_536_000_000 long bigNum = Integer.MAX_VALUE + 2L; // 2 promoted to long. Yields 2_147_483_649 long bigNegNum = Integer.MIN_VALUE-1L; // Yields -2_147_483_649 Date myDate = new Date(seconds * 1_000L); ... public long compute(int factor){ return factor * 10_000L; } public float compute2(long factor){ return factor / 123f; } or float twoThirds = (float)2/3; // 2 cast to float long millisInYear = (long)1_000*3_600*24*365; // 1_000 cast to long long bigNum = (long)Integer.MAX_VALUE + 2; long bigNegNum = (long)Integer.MIN_VALUE-1; Date myDate = new Date((long)seconds * 1_000); ... public long compute(long factor){ return factor * 10_000; } public float compute2(float factor){ return factor / 123; } See MITRE, CWE-190 - Integer Overflow or	BUG	MINOR	1

	Wraparound CERT, NUM50-J. - Convert integers to floating point for floating-point operations CERT, INT18-C. - Evaluate integer expressions in a larger size before comparing or assigning to that size SANS Top 25 - Risky Resource Management			
Methods should not be empty	There are several reasons for a method not to have a method body: It is an unintentional omission, and should be fixed to prevent an unexpected behavior in production. It is not yet, or never will be, supported. In this case an UnsupportedOperationException should be thrown. The method is an intentionally-blank override. In this case a nested comment should explain the reason for the blank override. Noncompliant Code Example public void doSomething() { } public void doSomethingElse() { } Compliant Solution @Override public void doSomething() { // Do nothing because of X and Y. } @Override public void doSomethingElse() { throw new UnsupportedOperationException(); } Exceptions Default (no-argument) constructors are ignored when there are other constructors in the class, as are empty methods in abstract classes. public abstract class Animal { void speak() { // default implementation ignored } }	CODE_SMELL	CRITICAL	1
Generic wildcard types should not be used in return parameters	It is highly recommended not to use wildcard types as return types. Because the type inference rules are fairly complex it is unlikely the user of that API will know how to use it correctly. Let's take the example of method returning a "List&lt;? extends Animal&gt;". Is it possible on this list to add a Dog, a Cat, ... we simply don't know. And neither does the compiler, which is why it will not allow such a direct use. The use of wildcard types should be limited to method parameters. This rule raises an issue when a method returns a wildcard type. Noncompliant Code Example List&lt;? extends Animal&gt; getAnimals(){...} Compliant Solution List&lt;Animal&gt; getAnimals(){...} or List&lt;Dog&gt; getAnimals(){...}	CODE_SMELL	CRITICAL	1
Sections of code should not be commented out	Programmers should not comment out code as it bloats programs and reduces readability. Unused code should be deleted and can be retrieved from source control history if required.	CODE_SMELL	MAJOR	2
Generic exceptions	Using such generic exceptions as Error, RuntimeException, Throwable, and Exception	CODE_SMELL	MAJOR	4

should never be thrown	prevents calling methods from handling true, system-generated exceptions differently than application-generated errors. Noncompliant Code Example public void foo(String bar) throws Throwable { // Noncompliant throw new RuntimeException("My Message"); // Noncompliant } Compliant Solution public void foo(String bar) { throw new MyOwnRuntimeException("My Message"); } Exceptions Generic exceptions in the signatures of overriding methods are ignored, because overriding method has to follow signature of the throw declaration in the superclass. The issue will be raised on superclass declaration of the method (or won't be raised at all if superclass is not part of the analysis). @Override public void myMethod() throws Exception {...} Generic exceptions are also ignored in the signatures of methods that make calls to methods that throw generic exceptions. public void myOtherMethod throws Exception { doTheThing(); // this method throws Exception } See MITRE, CWE-397 - Declaration of Throws for Generic Exception CERT, ERR07-J. - Do not throw RuntimeException, Exception, or Throwable			
Standard outputs should not be used directly to log anything	When logging a message there are several important requirements which must be fulfilled: The user must be able to easily retrieve the logs The format of all logged message must be uniform to allow the user to easily read the log Logged data must actually be recorded Sensitive data must only be logged securely If a program directly writes to the standard outputs, there is absolutely no way to comply with those requirements. That's why defining and using a dedicated logger is highly recommended. Noncompliant Code Example System.out.println("My Message"); // Noncompliant Compliant Solution logger.log("My Message"); See CERT, ERR02-J. - Prevent exceptions while logging data	CODE_SMELL	MAJOR	2
Unused "private" fields should be removed	If a private field is declared but not used in the program, it can be considered dead code and should therefore be removed. This will improve maintainability because developers will not wonder what the variable is used for. Note that this rule does not take reflection into account, which means that issues will be raised on private fields that are only accessed using the reflection API. Noncompliant Code Example public class MyClass { private int foo = 42; public int compute(int a) { return a * 42; } }	CODE_SMELL	MAJOR	4

	Compliant Solution public class MyClass { public int compute(int a) { return a * 42; } } Exceptions The Java serialization runtime associates with each serializable class a version number, called serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. A serializable class can declare its own serialVersionUID explicitly by declaring a field named serialVersionUID that must be static, final, and of type long. By definition those serialVersionUID fields should not be reported by this rule: public class MyClass implements java.io.Serializable { private static final long serialVersionUID = 42L; } Moreover, this rule doesn't raise any issue on annotated fields.			
Unused method parameters should be removed	Unused parameters are misleading. Whatever the values passed to such parameters, the behavior will be the same. Noncompliant Code Example void doSomething(int a, int b) { // "b" is unused compute(a); } Compliant Solution void doSomething(int a) { compute(a); } Exceptions The rule will not raise issues for unused parameters: that are annotated with @javax.enterprise.event.Observes in overrides and implementation methods in interface default methods in non-private methods that only throw or that have empty bodies in annotated methods, unless the annotation is @SuppressWarning("unchecked") or @SuppressWarning("rawtypes"), in which case the annotation will be ignored in overridable methods (non-final, or not member of a final class, non-static, non-private), if the parameter is documented with a proper javadoc. @Override void doSomething(int a, int b) { // no issue reported on b compute(a); } public void foo(String s) { // designed to be extended but noop in standard case } protected void bar(String s) { //open-closed principle } public void qix(String s) { throw new UnsupportedOperationException("This method should be implemented in subclasses"); } /** * @param s This string may be use for further computation in overriding classes */ protected void foobar(int a, String s) { // no issue, method is overridable and unused parameter has proper javadoc compute(a); } See CERT, MSC12-C. - Detect and remove code that has no effect or is never executed	CODE_SMELL	MAJOR	1

Field names should comply with a naming convention	Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that field names match a provided regular expression. Noncompliant Code Example With the default regular expression ^[a-z][a-zA-Z0-9]*$: class MyClass { private int my_field; } Compliant Solution class MyClass { private int myField; }	CODE_SMELL	MINOR	1
URIs should not be hardcoded	Hard coding a URI makes it difficult to test a program: path literals are not always portable across operating systems, a given absolute path may not exist on a specific test environment, a specified Internet URL may not be available when executing the tests, production environment filesystems usually differ from the development environment, ...etc. For all those reasons, a URI should never be hard coded. Instead, it should be replaced by customizable parameter. Further even if the elements of a URI are obtained dynamically, portability can still be limited if the path-delimiters are hard-coded. This rule raises an issue when URI's or path delimiters are hard coded. Noncompliant Code Example public class Foo { public Collection&lt;User&gt; listUsers() { File userList = new File("/home/mylogin/Dev/users.txt"); // Non-Compliant Collection&lt;User&gt; users = parse(userList); return users; } } Compliant Solution public class Foo { // Configuration is a class that returns customizable properties: it can be mocked to be injected during tests. private Configuration config; public Foo(Configuration myConfig) { this.config = myConfig; } public Collection&lt;User&gt; listUsers() { // Find here the way to get the correct folder, in this case using the Configuration object String listingFolder = config.getProperty("myApplication.listingFolder"); // and use this parameter instead of the hard coded path File userList = new File(listingFolder, "users.txt"); // Compliant Collection&lt;User&gt; users = parse(userList); return users; } } See CERT, MSC03-J. - Never hard code sensitive information	CODE_SMELL	MINOR	1
Boxed "Boolean" should be avoided in boolean	When boxed type java.lang.Boolean is used as an expression it will throw NullPointerException if the value is null as defined in Java Language Specification §5.1.8 Unboxing Conversion. It is safer to avoid such conversion altogether and handle the null value explicitly. Noncompliant Code Example Boolean b =	CODE_SMELL	MINOR	1

expressions	getBoolean(); if (b) { // Noncompliant, it will throw NPE when b == null foo(); } else { bar(); } Compliant Solution Boolean b = getBoolean(); if (Boolean.TRUE.equals(b)) { foo(); } else { bar(); // will be invoked for both b == false and b == null } See * Java Language Specification §5.1.8 Unboxing Conversion			
Using pseudorandom number generators (PRNGs) is security-sensitive	Using pseudorandom number generators (PRNGs) is security-sensitive. For example, it has led in the past to the following vulnerabilities: CVE-2013-6386 CVE-2006-3419 CVE-2008-4102 When software generates predictable values in a context requiring unpredictability, it may be possible for an attacker to guess the next value that will be generated, and use this guess to impersonate another user or access sensitive information. As the java.util.Random class relies on a pseudorandom number generator, this class and relating java.lang.Math.random() method should not be used for security-critical applications or for protecting sensitive data. In such context, the java.security.SecureRandom class which relies on a cryptographically strong random number generator (RNG) should be used in place. Ask Yourself Whether the code using the generated value requires it to be unpredictable. It is the case for all encryption mechanisms or when a secret value, such as a password, is hashed. the function you use generates a value which can be predicted (pseudo-random). the generated value is used multiple times. an attacker can access the generated value. You are at risk if you answered yes to the first question and any of the following ones. Recommended Secure Coding Practices Use a cryptographically strong random number generator (RNG) like "java.security.SecureRandom" in place of this PRNG. Use the generated random values only once. You should not expose the generated random value. If you have to store it, make sure that the database or file is secure. Sensitive Code Example Random random = new Random(); // Questionable use of Random byte bytes[] = new byte[20]; random.nextBytes(bytes); // Check if bytes is used for hashing, encryption, etc... Compliant Solution SecureRandom random = new SecureRandom(); // Compliant for security-sensitive use cases byte bytes[] = new byte[20]; random.nextBytes(bytes); See OWASP Top 10 2017 Category A3 - Sensitive Data Exposure MITRE, CWE-338 - Use of Cryptographically Weak Pseudo-Random Number	SECURITY_HOTSPOT	CRITICAL	1

	Generator (PRNG) MITRE, CWE-330 - Use of Insufficiently Random Values MITRE, CWE-326 - Inadequate Encryption Strength CERT, MSC02-J. - Generate strong random numbers CERT, MSC30-C. - Do not use the rand() function for generating pseudorandom numbers CERT, MSC50-CPP. - Do not use std::rand() for generating pseudorandom numbers Derived from FindSecBugs rule Predictable Pseudo Random Number Generator			
Using command line arguments is security-sensitive	Using command line arguments is security-sensitive. It has led in the past to the following vulnerabilities: CVE-2018-7281 CVE-2018-12326 CVE-2011-3198 Command line arguments can be dangerous just like any other user input. They should never be used without being first validated and sanitized. Remember also that any user can retrieve the list of processes running on a system, which makes the arguments provided to them visible. Thus passing sensitive information via command line arguments should be considered as insecure. This rule raises an issue when on every program entry points (main methods) when command line arguments are used. The goal is to guide security code reviews. Ask Yourself Whether any of the command line arguments are used without being sanitized first. your application accepts sensitive information via command line arguments. If you answered yes to any of these questions you are at risk. Recommended Secure Coding Practices Sanitize all command line arguments before using them. Any user or application can list running processes and see the command line arguments they were started with. There are safer ways of providing sensitive information to an application than exposing them in the command line. It is common to write them on the process' standard input, or give the path to a file containing the information. Sensitive Code Example This rule raises an issue as soon as there is a reference to argv, be it for direct use or via a CLI library like JCommander, GetOpt or Apache CLI. public class Main { &nbsp;&nbsp;&nbsp; public static void main (String[] argv) { &nbsp; &nbsp; &nbsp; &nbsp; String option = argv[0]; &nbsp;// Questionable: check how the argument is used &nbsp;&nbsp;&nbsp; } } // === JCommander === import&nbsp;com.beust.jcommander.*; public class Main { &nbsp;&nbsp;&nbsp; public static void main (String[] argv) { &nbsp;&nbsp;&nbsp; &nbsp;&nbsp;&nbsp; Main main = new Main(); &nbsp; &nbsp; &nbsp; &nbsp;	SECURITY_HOTSPOT	CRITICAL	1

JCommander.newBuilder() &nbsp; &nbsp; &nbsp; &nbsp; .addObject(main) &nbsp; &nbsp; &nbsp; &nbsp; .build() &nbsp; &nbsp; &nbsp; &nbsp; .parse(argv); // Questionable &nbsp;&nbsp;&nbsp;&nbsp;&nbsp; &nbsp;&nbsp;main.run(); &nbsp;&nbsp; &nbsp;} } // === GNU Getopt === import gnu.getopt.Getopt; public class Main { &nbsp;&nbsp;&nbsp; public static void main (String[] argv) { &nbsp;&nbsp; &nbsp;&nbsp;&nbsp; &nbsp;Getopt g = new Getopt("myprog", argv, "ab"); // Questionable &nbsp;&nbsp;&nbsp; } } // === Apache CLI === import org.apache.commons.cli.*; public class Main { &nbsp;&nbsp;&nbsp; public static void main (String[] argv) { &nbsp;&nbsp; &nbsp;&nbsp;&nbsp; &nbsp;Options options = new Options(); &nbsp;&nbsp;&nbsp;&nbsp;&nbsp; &nbsp; CommandLineParser parser = new DefaultParser(); &nbsp;&nbsp;&nbsp; &nbsp;&nbsp; &nbsp;try { &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp; &nbsp;&nbsp; CommandLine line = parser.parse(options, argv); // Questionable &nbsp;&nbsp;&nbsp;&nbsp;&nbsp; &nbsp; } &nbsp;&nbsp;&nbsp; } } In the case of Args4J, an issue is created on the public void run method of any class using org.kohsuke.args4j.Option or org.kohsuke.args4j.Argument. Such a class is called directly by org.kohsuke.args4j.Starter outside of any public static void main method. If the class has no run method, no issue will be raised as there must be a public static void main and its argument is already highlighted. // === argv4J === import org.kohsuke.args4j.Option; import org.kohsuke.args4j.Argument; public class Main { @Option(name="-myopt",usage="An option") public String myopt;   @Argument(usage = "An argument", metaVar = "&lt;myArg&gt;")   String myarg;   String file;   @Option(name="-file") public void setFile(String file) {   this.file = file;   } String arg2;   @Argument(index=1)   public void setArg2(String arg2) {   this.arg2 = arg2;   } &nbsp;&nbsp; &nbsp;public void run() { // Questionable: This function   myarg; // check how this argument is used &nbsp;&nbsp; &nbsp;} } Exceptions The support of Argv4J without the use of org.kohsuke.argv4j.Option is out of scope as there is no way to know which Bean will be used as the mainclass. No issue will be raised on public static void main(String[] argv) if argv is not referenced in the method. See   OWASP Top 10 2017 Category A1 -

	Injection MITRE, CWE-88 - Argument Injection or Modification MITRE, CWE-214 - Information Exposure Through Process Environment SANS Top 25 - Insecure Interaction Between Components