TURING-GALLERY-BACK

VERSION 1.0

CODE ANALYSIS

By: default

2020-01-27

CONTENT

Content 1

Introduction 2

Configuration 2

Synthesis 2

Metrics 2

Volume 3

Issues count by severity and type 3

Charts 4

Issues 5

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	A	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	24.0	7.0	201.0	14.3	0.0	0.0

TURING-GALLERY-BACK

VOLUME

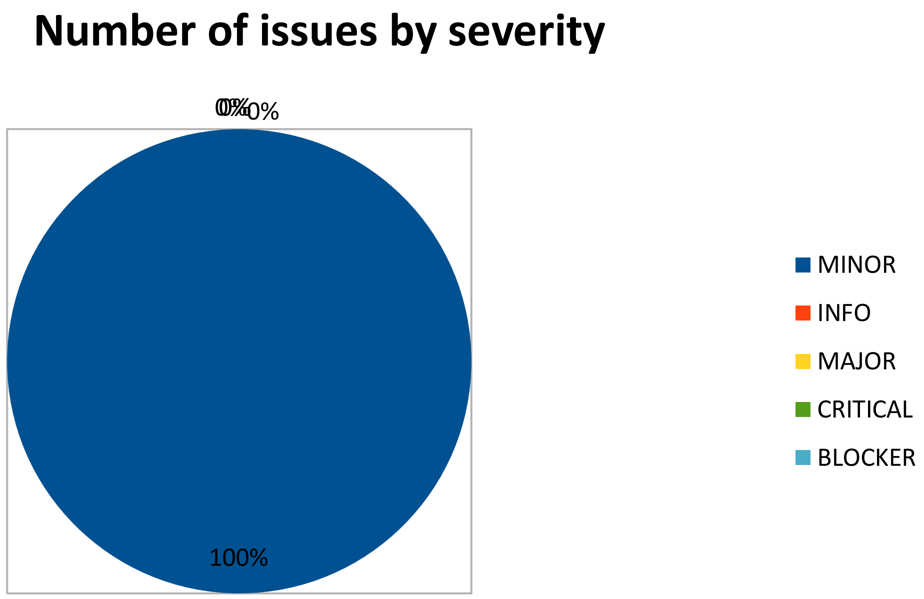
Language	Number
Java	192
HTML	9
Total	201

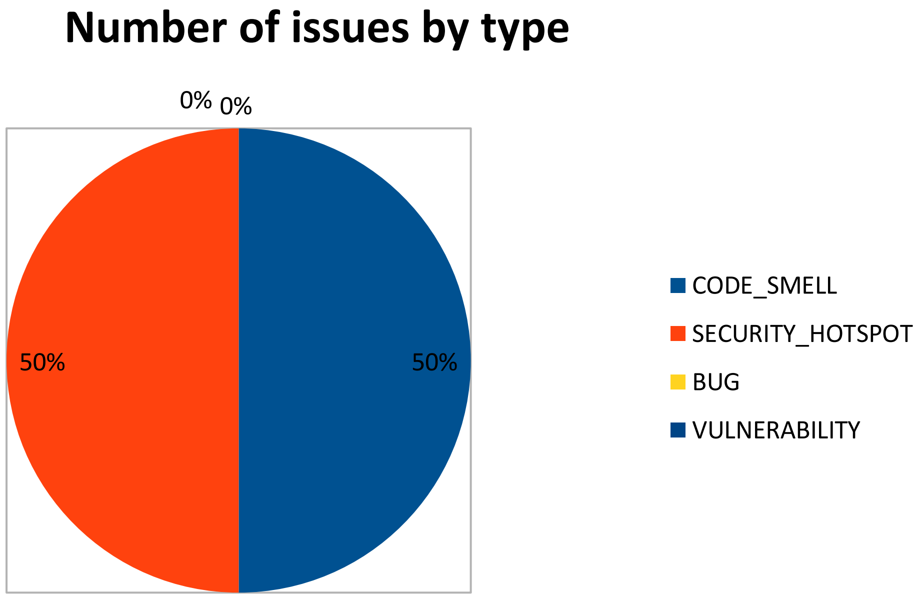
ISSUES COUNT BY SEVERITY AND TYPE

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

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
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	CODE_SMELL	MINOR	1

	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			
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	SECURITY_HOTSPOT	CRITICAL	1

(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; 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