4/22/22

Project 5 Analysis on Vulnerabilities in Spotify's GitHub-Java-Client

WHAT ARE THE VULNERABILITIES AND WHAT DO THEY MEAN

MARQUIS, ALEX

Table of Contents

Introduction

On the 18th of April 2022, Alexander Marquis ran a Parasoft Jtest static code analysis scan on the Spotify GitHub-Java-Client project and generated a report of the findings (Spotify *Spotify/github-java-client*). This report identified a total of 368 findings within the Spotify GitHub-Java-Client project. A finding in this context is a line of code that is a potential a security flaw. A security flaw is a flaw in code that code lead to a potentially exploitable vulnerability. These findings must be further examined to determine whether the line in question is vulnerable and the exploitability of it if it is vulnerable. The Parasoft Jtest report categorizes findings by the security rule that was potentially violated. The report generated for the Spotify GitHub-Java-Client project broke the 368 findings down into potential violations of 20 separate security rules. This report examines the findings from 10 of those 20 security rules to further understand whether those findings were vulnerable, what the found vulnerabilities represent, how serious they're consequences can be, and how it should be prioritized for mitigation.

SEI CERT and the SEI CERT Oracle Coding Standard for Java

The Parasoft Jtest scan used to identify potential vulnerabilities in the Spotify GitHub-Java-Client project was configured to use the built-in security pack known as *CERT for Java* which bases its security rules on the *SEI CERT Oracle Coding Standard for Java* (McManus et al. *SEI CERT*). This standard provides rules and recommendations for developing and maintaining secure software in Java and was developed by *Carnegie Mellon University's Software Engineering Institute's* (SEI) *Computer Emergency Response Team* (CERT) (McManus et al. *SEI CERT*). The rules in the *SEI CERT Oracle Coding Standard for Java* are requirements that are meant to be followed in order to increase the security, reliability, dependability, robustness, resilience, availability, and maintainability of a program (Flynn et al. *Rules versus Recommendations*). Failure to follow these rules is likely to result in security flaws or defects that may affect the safety, reliability, or security of a system. Unless the rule specifies an exception, these rules are meant to always be enforced. Recommendations are guidelines that, when followed, should increase the safety, reliability, and or security of software system. Rules are meant to provide normative requirements for code whereas recommendations simply provide guidance. The violation of a recommendation does not necessarily indicate that there is a defect in the code (Rules and Recs).

SEI CERT Risk Analyses

SEI CERT performs a risk analysis for each rule and recommendation and provides the results to help indicate to developers the potential consequences of failing to address a violation of a rule or recommendation in the software system along with the expected cost of remediation (Svoboda et al. *How this Coding Standard*). This information is broken down into 3 distinct categories: severity, likelihood, and remediation cost with each one of those categories being broken down into 3 distinct levels (Svoboda et al. *How this Coding Standard*). Severity informs of just how drastic the consequences of failing to comply with a rule or recommendation can be and is broken down into the levels of low with a value of 1, medium with a value of 2, and high with a value of 3. Low often describes consequences like denial-of-service attacks or abnormal terminations. Medium characterizes consequences such as data integrity violations or data leakage. High describes the attacks that take the form of executing arbitrary code (Svoboda et al. *How this Coding Standard*). The likelihood category is meant to describe how likely it is that the flaw introduced by failing to comply with a rule or recommendation leads to an exploitable vulnerability and is broken up into the levels of unlikely with a value of 1, probable with a value of 2, and likely with a value of 3 (Svoboda et al. *How this Coding Standard*). These levels are meant to be used to provide a general idea of how likely it is for the consequences described in the severity section to occur.

Remediation cost describes the estimated cost in both time and resources to bring a program into compliance with a rule or recommendation. Remediation cost is broken up into the levels of high with a value of 1, medium with a value of 2, and low with a value of 3 (Svoboda et al. *How this Coding Standard*). High remediation cost vulnerabilities need to be both manually detected and corrected. Medium remediation cost vulnerabilities can be detected automatically but still need to be corrected manually. Low remediation cost vulnerabilities can both be detected and corrected automatically (Svoboda et al. *How this Coding Standard*).

The SEI CERT risk analyses also provide priority and level as derivative categories of the previous three categories. The priority category seeks to provide guidance to developers when prioritizing which vulnerabilities should be mitigated. The priority category's value is derived by multiplying the values from the previous three categories together and adding the capitalized letter "P" in front of the product (Svoboda et al. *How this Coding Standard*). Due to the available values in the other three categories, priority can only take the values of P1, P2, P3, P4, P6, P8, P9, P12, P18, and P27. High values indicate a higher level of priority while the opposite is true for lower values (Svoboda et al. *How this Coding Standard*). Level is the final category present in SEI CERT's risk assessments for violations of their rules and vulnerabilities. Level is a derivative category of priority and can take three values: L1 at level 1, L2 at level 2, and L3 at level 3. Levels are meant to provide a broader way to prioritize vulnerabilities so that they can be more effectively grouped together. Level 1 (L1) encapsulates the priority levels of P12-p27, level 2 (L2) has P6-P9, and level 3 uses P1-P4 (Svoboda et al. *How this Coding Standard*).

Review of Findings by Violation of Security Rules

(CERT.SER03.SIF-1) Possible Exposure of Sensitive Information in Serializable Object

The report generated by the Parasoft Jtest scan identified two findings for violations of security rule CERT.SER03.SIF-1 with both findings being located in the same file. Security rule *CERT.SER03.SIF-1* states that any non-transient, non-final instance field that is part of serializable class may expose confidential information held within those fields. Therefore, the data within those fields must either be ensured to not be sensitive or be encrypted, or the fields must either be removed or made transient and final. This security rule is based on the SEI CERT rule *SER03-J. Do not serialize unencrypted sensitive data* (Mohindra et al. SER03-J). This SEI CERT rule states that since unencrypted serialized data can be exposed to attackers; unencrypted sensitive data should not be serialized (Mohindra et al. SER03-J). Exposed serialized data can allow attackers to discover sensitive information that they are not meant to have access to and could allow for modification of the serialized data in a malicious way (Mohindra et al. SER03-J). SEI CERT's risk assessment for violations of this rule put the severity at medium, the likelihood at likely, and the remediation cost at high (Mohindra et al. SER03-J). Using these factors SEI CERT's risk assessment for violations of this rule indicate that they have a priority of P6 making them fall within level L2.

SEI CERT Risk Assessment for Violations of Rule SER03-J				
Severity	**Likelihood**	**Remediation Cost**	**Priority**	**Level**
Medium	Likely	High	P6	L2

When looking at the code that is being pointed to as the source of the potential security flaw in the */github-client/src/main/java/com/spotify/github/v3/exceptions/RequestNotOkException.java* file, it appears to be a false-positive at best or an easy remediation at worst. The `RequestNotOkException` class is designed to setup a custom exception for when a HTTP request to GitHub fails. Exceptions are meant to be serializable so that debugging information can be transferred across programs and webservices as needed. The specific data fields that are being indicated as being vulnerable, the `statusCode` and `path` variables on lines 40 and 41 respectively, do not appear to be storing sensitive information as the URI and HTTP response status code should not contain any unencrypted sensitive information. However, even though the URI shouldn't contain unencrypted sensitive information it is possible that it does. If the URI is not to be trusted to not contain unencrypted sensitive information then this vulnerability can be easily mitigated by making the `path` variable on line 41 transient. The status code should truly never contain sensitive information, so it does not need to be made transient and doing so may hinder debugging efforts due to insufficient logging. SEI's assessment for the remediation cost of this vulnerability is not particularly accurate in this case since the mitigation for this vulnerability is as easy as appending the word key word `transient` to the front of the `path` variable's declaration. For this reason, it is recommended that this vulnerability's remediation cost be treated as low causing the new priority to be P12 and the level to be L1.

File***: /github-client/src/main/java/com/spotify/github/v3/exceptions/RequestNotOkException.java***

```
38 public class RequestNotOkException extends GithubException {
39
40     private final int statusCode;
41     private final String path;
```

(CERT.ERR00.LGE-2) Failure to Properly Log Exceptions

Two findings were discovered under security rule CERT.ERR00.LGE-2 by the Parasoft Jtest static code analysis scan. This security rule states that since logging is so effective at allowing developers to quickly and effectively track down bugs that all caught exceptions should either be logged or rethrown. SEI CERT rule *ERR00-J. Do not suppress or ignore checked exceptions* that states that checked exceptions should not be suppressed or ignored serves as the basis of security rule CERT.ERR00.LGE-2 (Mohindra et al. *ERR00-J*). CERT.ERR00.LGE-2 builds off of that requirement by requiring that all caught exceptions either be logged or rethrown. SEI CERT's risk assessment for violations of rule *ERR00-J* determined that the severity is ranked at low, the likelihood as probable, and the remediation cost at medium (Mohindra et al. *ERR00-J*). The low severity for this rule is indicative of the fact that this vulnerability is not directly exploitable, rather it helps track, monitor, and stop other vulnerabilities and exploits. The likelihood of probable for this type of vulnerability means that is probable that other vulnerabilities and or exploits will occur and go unnoticed due to the insufficient logging. The medium remediation costs are indicative of the fact that both code will need to be modified and the new logging will require more space and overhead to perform. These factors lead SEI CERT vulnerabilities stemming from a failure to comply with rule *ERR00-J* a priority of P4 and deem them as being in level L3 (Mohindra et al. *ERR00-J*). The quite low priority is a result of the fact that this flaw is not directly exploitable making the time and cost to mitigate it not as worthwhile as it may be for other vulnerabilities.

SEI CERT Risk Assessment for Violations of Rule ERR00-J				
Severity	**Likelihood**	**Remediation Cost**	**Priority**	**Level**
Low	Probable	Medium	P4	L3

When actually looking at the code that is being indicated as being the source of the errors it appears that the first finding that's located in the */github-client/src/test/java/com/spotify/github/v3/clients/GitHubClientTest.java* file on line 129 is a false positive. This is believed to be the case because the code being pointed to is part of a test as indicated by the name of the file being GitHubClientTest.java and it being located under the */github-client/src/test/* directory. It is testing whether the contents of the exception contain the expected information from the test. While this exception could still be logged, the code in question is only run during tests and all relevant information for the exception would be found in the test results. Therefore, this is deemed as being a false-positive as logging would appear to provide little to no benefit in this case and would cost additional overhead and storage resources.

File: ***/github-client/src/test/java/com/spotify/github/v3/clients/GitHubClientTest.java***

```
129   } catch (ExecutionException e) {
130       assertThat(e.getCause() instanceof RequestNotOkException, is(true));
131       RequestNotOkException e1 = (RequestNotOkException) e.getCause();
132       assertThat(e1.statusCode(), is(409));
133       assertThat(e1.getMessage(), containsString("Merge Conflict"));
134   }
```

The second finding under the CERT.ERR00.LGE-2 security rule is located in the */github-client/src/main/java/com/spotify/github/Parameters.java* file on line 69 and appears to have more merit to it. Inside a method called `serialize()`, designed for serialization of class files, when trying to wrap the result of invoking a method in an `Optional` object it is placed inside of a try block. If this try block fails for any reason the catch block simply returns an empty `Optional` object but does nothing with the exception. This exception should be logged as it is not occurring within the confines of a test so the information it holds should be available for review inside of log files. While this will increase the overhead of the program and the storage requirements for logging it will increase security and maintainability by providing more information to developers and testers.

File: ***/github-client/src/main/java/com/spotify/github/Parameters.java***

```
59 try {
60    final Object invocationResult = method.invoke(this);
...
66    return invocationResult instanceof Optional
67       ? (Optional) invocationResult
68       : Optional.ofNullable(invocationResult);
69 } catch (Exception e) {
70    return Optional.empty();
71 }
```

(CERT.LCK06.INSTLOCK) Usage of Instance Lock to Protect Static Shared Data.

A single finding was uncovered for security rule CERT.LCK06.INSTLOCK by the Parasoft Jtest static code analysis scan. Security rule CERT.LCK06.INSTLOCK is derived from SEI CERT rule *LCK06-J. Do not use an instance lock to protect shared static data* which much as it's title implies, states that instance locks are insufficient when trying to protect shared static data (Mohindra et al. *LCK06-J*). This is because if multiple instances of the class are created the shared state unprotected from concurrent access (Mohindra et al. *LCK06-J*). The risk assessment performed by SEI CERT for violations of this rule determined the severity of the consequences of a an exploit based on a failure to follow this rule to be medium (Mohindra et al. *LCK06-J*). This rating in conjunction with this security rule indicates that that internal values and states within the program maybe inconsistent across multiple class instances which is often a data integrity violation (Mohindra et al. *LCK06-J*). It was also determined that the likelihood of a security flaw stemming from a violation of this rule leading to an exploitable vulnerability and therefore putting data integrity within the program at the whim of an attacker was probable (Mohindra et al. *LCK06-J*). The risk assessment also found rated the remediation cost as being medium, indicating that a moderate amount of time and resources will need to be dedicated to ensuring the security of the program (Svoboda et al. *How this Coding Standard*). These factors lead SEI CERT to deem vulnerabilities caused by a failure to comply with rule LCK06-J to have a priority of P8 making them part of level L2.

SEI CERT Risk Assessment for Violations of Rule LCK06-J				
Severity	**Likelihood**	**Remediation Cost**	**Priority**	**Level**
Medium	Probable	Medium	P8	L2

When examining the source code in the */github-client/src/test/java/com/spotify/github/opencensus/TestExportHandler.java* file, the finding under security rule CERT.LCK06.INSTLOCK can be located on line 52 where the static class variable LOG is accessed while it is only being held under an instance lock. The instance lock can be seen in its creation on line 46 and its usage on line 50. While SEI CERT deems this to have a remediation cost of medium, in this particular case I would have to advise differently. This code can be mitigated by simply implementing the static keyword in the initialization of the lock variable making it a static variable (Mohindra et al. *LCK06-J*). Doing so makes it so all class instances share the lock and thus concurrent access is not possible which maintains data integrity (Mohindra et al. *LCK06-J*). Since the mitigation for this security flaw is in reality very cheap and easy to implement it is recommended that this vulnerability's remediation cost be treated as low and therefore its priority be treated as a P12.

File: ***/github-client/src/test/java/com/spotify/github/opencensus/TestExportHandler.java***

```
42 class TestExportHandler extends SpanExporter.Handler {
43    private static final Logger LOG =
LoggerFactory.getLogger(TestExportHandler.class);
44
45    private final List<SpanData> receivedSpans = new ArrayList<>();
46    private final Object lock = new Object();
47
48    @Override
49    public void export(final Collection<SpanData> spanDataList) {
50       synchronized (lock) {
51       receivedSpans.addAll(spanDataList);
52       LOG.info("received {} spans, {} total", spanDataList.size(),
receivedSpans.size());
53            }
54        }
```

(CERT.ERR00.UCATCH-3) Use a Caught Exception in the "catch" Block

Security rule CERT.ERR00.UCATCH-3 was identified to have a single finding based on the Parasoft Jtest scan ran by Alexander Marquis. CERT.ERR00.UCATCH-3 much like security rule CERT.ERR00.LGE-2, is derived from the SEI CERT rule *ERR00-J. Do not suppress or ignore checked exceptions* (Mohindra et al. *ERR00-J*). This rule states that checked exceptions should not be ignored or suppressed (Mohindra et al. *ERR00-J*). Security rule CERT.ERR00.UCATCH-3 is a more specific form of the SEI CERT rule that states that the exceptions that are caught within a catch block should be used. A risk assessment of violations of SEI CERT rule ERR00-J was performed by SEI CERT and it was found that security flaws caused by a failure to follow SEI CERT rule *ERR00-J* had a low severity, a high likelihood, and a medium remediation cost (Mohindra et al. *ERR00-J*). These ratings for this type of security flaw indicate that abnormal program termination or lack of availability may be possible and if so are probably exploitable, and that mitigating the vulnerability would cost a moderate amount of time and resources. These factors combined for SEI CERT to assign vulnerabilities caused by a failure to follow SEI CERT rule *ERR00-J* with a priority of P4 and therefore a level of L3 (Mohindra et al. *ERR00-J*).

SEI CERT Risk Assessment for Violations of Rule ERR00-J				
Severity	**Likelihood**	**Remediation Cost**	**Priority**	**Level**
Low	Probable	Medium	P4	L3

When looking to the source of the finding in the */github-client/src/main/java/com/spotify/github/Parameters.java* file on line 69 as indicated by the Parasoft Jtest scan, it can be see that an exception is caught and nothing is done with it. As was mentioned in the section over the findings for security rule CERT.ERR00.LGE-2, the method which the code in question takes place is designed for serializing class files. When trying to wrap the result of invoking a method in an `Optional` object it is placed inside of a try block. If this try block fails for any reason the catch block simply returns an empty `Optional` object. This exception should be logged as it is not useless information. Doing so would mitigate findings under both the CERT.ERR00.UCATCH-3 and CERT.ERR00.LGE-2 security rules. SEI CERT's risk assessment appears to be particularly accurate when describing this vulnerability; especially when it comes to remediation costs. Logging software introduces much cost, both in terms of time and resources for implementation, maintenance, and operation. Logs must be stored, program resources must be dedicated to logging, developers must spend time implementing logging. The time and resource cost on this, while not immense, is not to be taken lightly.

File: ***/github-client/src/main/java/com/spotify/github/Parameters.java***

```
59 try {
60    final Object invocationResult = method.invoke(this);
...
66    return invocationResult instanceof Optional
67       ? (Optional) invocationResult
68       : Optional.ofNullable(invocationResult);
69 } catch (Exception e) {
70    return Optional.empty();
71 }
```

(CERT.EXP00.NASSIG-3) Ensure Method and Constructor Return Values are Used

The report generated by the Parasoft Jtest scan ran by Alexander Marquis identified four findings under security rule CERT.EXP00.NASSIG-3. Security rule CERT.EXP00.NASSIG-3 is based off of the SEI CERT rule *EXP00-J. Do not ignore values returned by methods* (Mohindra et al. *EXP00-J*). This rule states that since methods can return values to communicate the failure or success of updating local objects or fields that ignoring or failing to properly respond to the returned values from invoked methods is a security flaw (Mohindra et al. *EXP00-J*). The risk assessment for violations of this rule performed by SEI CERT determined that exploits on this vulnerability can pose a medium level of severity as it can lead to unexpected program behavior (Mohindra et al. *EXP00-J*, Svoboda et al. *How this Coding Standard*). Having a non-updated state when the program logic relies on it being properly updated is seen as being probable to cause unexpected program behavior (Mohindra et al. *EXP00-J*). Remediation cost for fixing vulnerabilities caused by violating this rule is ranked as being medium (Mohindra et al. *EXP00-J*). This is because while this vulnerability can potentially be detected by static code analysis software mitigating it will require knowledge of the method being used, what it's return value represents, and how to properly handle it in the context of the program. These factors lead SEI CERT to classify vulnerabilities stemming from violations of EXP00-J to have a priority of P8 and to be of level L2.

SEI CERT Risk Assessment for Violations of Rule EXP00-J				
Severity	**Likelihood**	**Remediation Cost**	**Priority**	**Level**
Medium	Probable	Medium	P8	L2

When observing the source of the first two findings in the */github-client/src/main/java/com/spotify/github/opencensus/OpenCensusTracer.java* file on lines 47 and 48 as indicated by the Parasoft Jtest scan, it can be seen that while this is a violation of this SEI CERT rule, it is not a particularly exploitable one. All that is occurring here is that the program is ensuring that the method parameters `path` and `future` are not holding null values. However, even though it takes more operations and will likely eventually be filtered out by the compiler, it is still recommended that instead of just invoking the `requireNonNull` method with the path and future parameters, that those variables be assigned to result of invoking those methods on themselves. This is recommended for readability reasons and the security flaws in these findings are not exploitable but may lead to future mistake and misunderstandings. Since this security flaw is not exploitable and is easy to fix, the new recommended priority for this mitigation is P2.

File: ***/github-client/src/main/java/com/spotify/github/opencensus/OpenCensusTracer.java***

```
43 private Span internalSpan(
44     final String path,
45     final String method,
46     final CompletionStage<?> future) {
47 requireNonNull(path);
48 requireNonNull(future);
```

NEEDS EDITING span() doesn't return future it returns internal span

The next two findings for security rule CERT.EXP00.NASSIG-3 are located in the */github-client/src/test/java/com/spotify/github/opencensus/OpenCensusTracerTest.java* file on lines 61 and 87. It can be seen that these lines are parts of tests as indicated by the filename of *OpenCensusTracerTest.java* and the fact that the file is located under the */github-client/src/test/* directory. These findings in invoke the `span()` method but do nothing with the returned `span` objects. This seems to have been done because the `future` objects that the `span` objects are set to close after completing are still accessible from within their methods. While these flaws are not exploitable it is still recommended that variables to store the `span` objects be created within smaller required scopes to improve readability and make clearer the total lifespan of the `span` objects.

File: ***/github-client/src/test/java/com/spotify/github/opencensus/OpenCensusTracerTest.java***

```
056   public void testTrace_CompletionStage_Simple() throws Exception {
057      Span rootSpan = startRootSpan();
058      final CompletableFuture<String> future = new CompletableFuture<>();
059      OpenCensusTracer tracer = new OpenCensusTracer();
060
061      tracer.span("path", "GET", future);
...
082   public void testTrace_CompletionStage_Fails() throws Exception {
083      Span rootSpan = startRootSpan();
084      final CompletableFuture<String> future = new CompletableFuture<>();
085      OpenCensusTracer tracer = new OpenCensusTracer();
086
087      tracer.span("path", "POST", future);
```

(CERT.MET02.DPRAPI-3) Do not use deprecated APIs

A single finding of the security rule CERT.MET02.DPRAPI-3 was detected by the Parasoft Jtest scan run by Alexander Marquis. This security rule takes after the SEI CERT rule *MET02-J. Do not use deprecated or obsolete classes or methods* (Mohindra et al. *MET02-J*). This SEI CERT rule, much as the title would imply, states that deprecated and obsolete classes and methods should never be used. Doing so can result in unexpected behavior from the program (Mohindra et al. *MET02-J*). However, SEI CERT's risk analysis of vulnerabilities stemming from violations of SEI CERT rule MET02-J states that it has a severity of low and likelihood of unlikely (Mohindra et al. *MET02-J*). These factors combine to mean that the consequences for not patching this is likely quite low and the likelihood of those consequences occurring are also quite low (Mohindra et al. *MET02-J*). The remediation cost is deemed as being medium because also detection of deprecated classes and methods can be easy and automated; detection of obsolete classes and methods is usually a manual task (Mohindra et al. *MET02-J*). In addition, implementation of the newer version of the class or method usually takes time and can require restructuring of the code to fit the new method or class's operation.

SEI CERT Risk Assessment for Violations of Rule MET02-J				
Severity	**Likelihood**	**Remediation Cost**	**Priority**	**Level**
Low	Unlikely	Medium	P2	L3

Observing the code that is pointed to as being the source of the finding in the */github-client/src/main/java/com/spotify/github/v3/clients/GitDataClient.java* file on line 153, it can be seen that the usage of the deprecated `listReferences` method occurs only in another overloaded and deprecated `listReferences` method. These methods are deprecated, not really used inside the code outside of one of the deprecated methods using another deprecated method. Since both methods are deprecated and should not be included into any future programs they are only left in the program for backwards compatibility. While the output of these deprecated methods may lead to erroneous program behavior, they are not actively implemented in the program in an exploitable way, do not have severe consequences, are unlikely to exploited in the first place, and the cost to remove these methods may be very high depending on who is relying on these deprecated methods. These factors in addition to the fact that a newer version of the method is already offered have lead to the determination that this mitigation in particular is deemed to have a priority of P1 and a level of L3.

File: ***/github-client/src/main/java/com/spotify/github/v3/clients/GitDataClient.java***

```
144    @Deprecated
145    public CompletableFuture<List<Reference>> listReferences(final String ref) {
146        final String path = format(REFERENCE_URI, owner, repo,
ref.replaceAll("refs/", ""));
147        return github.request(path, LIST_REFERENCES);
148    }
149
150    /** List references. (Replaced by listMatchingReferences for github
enterprise version > 2.18) */
151    @Deprecated
152    public CompletableFuture<List<Reference>> listReferences() {
153        return listReferences("");
154    }
```

(CERT.SER07.RRSC-3) Define a "readResolve" Method for All Instances of Serializable Types

The Parasoft Jtest static code analysis scan performed by Alexander Marquis detected a single finding for security rule CERT.SER07.RRSC-3. This security ruleis based on the SEI CERT rule *SER07-J. Do not use the default serialized form for classes with implementation-defined invariants* (Mohindra et al. *SER07-J*). This rule states that since the default serialized form lacks enforcement of class invariants, programs must not use this default form (Mohindra et al. *SER07-J*). Deserialization creates a new instance of a class without having to invoke the class's constructors, subsequently any input validation that is occurring in the constructor of a class must also be occurring during the deserialization process (Mohindra et al. *SER07-J*). Since the class instance is constructed without a constructor it may break the invariant associated with singleton classes and as such all classes that implements the `java.io.Serializable` interface should also implement a proper `readResolve` method to enforce singleton semantics (Mohindra et al. *SER07-J*). SEI CERT's risk assessment of the vulnerabilities stemming from violations of SEI CERT rule found that they had a severity of medium, a likelihood of probable, and a remediation cost of high. The severity of medium combined with deserialization makes it likely that exploitations will violate data integrity by creating a second instance of what is supposed to be a singleton (Mohindra et al. *SER07-J*). The likelihood indicates that this is likely to be exploited and cause the break in data integrity. However, the remediation cost is high, indicating that it will take much time and resources to properly mitigate this bug due to the varied nature and purposes for serialization. These factors lead SEI CERT to assign vulnerabilities that stem from the violation of rule SER07-J as having a priority of 4 and level of 3.

SEI CERT Risk Assessment for Violations of Rule SER07-J				
Severity	**Likelihood**	**Remediation Cost**	**Priority**	**Level**
Medium	Probable	High	P4	L3

The source code pointed to by the Parasoft Jtest scan for this finding in the */github-client/src/main/java/com/spotify/github/jackson/GitHubInstantJsonSerializer.java* file on line 31 it can be clearly seen that there is glaring lack of the readResolve() method. This indicates that the singleton nature of the class may not be preserved and is therefore vulnerable to data integrity violations. This can be mitigated by implementing a read resolve method that references the `INSTANCE` constant pointing to the single instance of the singleton.

File: ***/github-client/src/main/java/com/spotify/github/jackson/GitHubInstantJsonSerializer.java***

```
31 class GitHubInstantJsonSerializer extends StdSerializer<GitHubInstant> {
32
33    static final GitHubInstantJsonSerializer INSTANCE = new
GitHubInstantJsonSerializer();
34
35    private GitHubInstantJsonSerializer() {
36       super(GitHubInstant.class);
37    }
38
39    @Override
40    public void serialize(
...
49    }
50 }
```

(CERT.SEC05.ARM-4) Avoid Using Reflection Methods

Security rule CERT.SEC05.ARM-4 was identified as having a single finding by the Parasoft Jtest scan ran by Alexander Marquis. This security rule is based off of SEI CERT rule *SEC05-J. Do not use reflection to increase accessibility of classes, methods, or fields* which states that since Java reflection API includes methods that allow normally inaccessible fields to become accessible, that it complicates security analysis and can easily introduce vulnerabilities and for that reason should be avoided whenever possible (Mohindra et al. *SEC05-J*). SEI CERT's risk analysis for vulnerabilities stemming from violations of this rule were determined to have a severity of high, a likelihood of probable, and a remediation cost of medium. These factors have lead SEI CERT to classify these vulnerabilities as having a priority of P12 and level of L1. This is because allowing classes, methods, or fields to become accessible can lead to breakage of data encapsulation, leakage of sensitive information, and privilege escalations attacks (Mohindra et al. *SEC05-J*). This combined with the fact that it is a well-known vulnerability that is not hard to exploit, and it is very likely to be exploited (Mohindra et al. *SEC05-J*). The Parasoft Jtest scan run determined security rule CERT.SEC05.ARM-4 to have a severity of low/4. This is largely due to the fact that reflection methods are often used when necessary which causes it to be flagged when there is no suitable alternative.

SEI CERT Risk Assessment for Violations of Rule SEC05-J				
Severity	**Likelihood**	**Remediation Cost**	**Priority**	**Level**
High	Probable	Medium	P12	L1

This security flaw does not appear to be exploitable as evidenced by the source code pointed as the source of the security flaw by the Parasoft Jtest scan in the */github-client/src/main/java/com/spotify/github/Parameters.java* file on line 60. It can be seen that the `Method.invoke` method is only operating on public methods gotten from the `getMethods` method invoked on line 51 of the same file. Although this security flaw is not exploitable it is still recommended that it be mitigated. This is because the author of the code acknowledges that there is a way to accomplish the same task without using Java reflection in his Javadoc for the serialize method that encapsulates lines 48 through 80. Because reflection is not necessary in this case and it increases the complexity of the security analysis of the code it is recommended that this security flaw be mitigated to no longer include use of the Java reflection API.

File: ***/github-client/src/main/java/com/spotify/github/Parameters.java***

```
48 default String serialize() {
49    return Arrays.stream(this.getClass().getInterfaces())
50       .filter(Parameters.class::isAssignableFrom)
51       .map(Class::getMethods)
52       .flatMap(Arrays::stream)
53       // Filter out any method defined in this interface.
54       .filter(method -> !method.getDeclaringClass().equals(Parameters.class))
55       .collect(
56          toMap(
57             Method::getName,
58             method -> {
```

(CERT.MSC03.HCCK-1) Avoid Using Hard-Coded Cryptographic Keys

The Parasoft Jtest scan was able to identify four findings under the security rule CERT.MSC03.HCCK-1. The Parasoft Jtest page for this security rule states that findings for this rule will be flagged for each instance of a hardcoded cryptographic key that is composed of strictly hexadecimal values. This security rule derives from the SEI CERT rule *MSC03-J. Never hard code sensitive information* which, much as its title implies, states that sensitive information, such as cryptographic keys, has the potential to expose sensitive information to attackers and increases the complexity of managing and accommodating changes to the code (Mohindra et al. *MSC03-J*). SEI CERT's risk analysis of vulnerabilities stemming from violations of this rule states that they have a severity of high, a likelihood of probable, and a remediation cost of medium (Mohindra et al. *MSC03-J*). This results in a priority of P12 and a level of 1 (Mohindra et al. *MSC03-J*). When sensitive information such as cryptographic keys become available to attackers, remote exploitation is not only possible but probable (Mohindra et al. *MSC03-J*).

SEI CERT Risk Assessment for Violations of Rule SEC05-J				
Severity	**Likelihood**	**Remediation Cost**	**Priority**	**Level**
High	Probable	Medium	P12	L1

The first finding under this security rule points to the code on line 41 of the */github-client/src/test/java/com/spotify/github/v3/activity/events/StatusEventTest.java* file as being the source of the potential security flaw. However, from the file name alone we can begin to get the intuition that since this is coming from a test file, as evidenced by the name *StatusEventTest.java* and the fact that its located in the */github-client/src/test* directory, that it is likely that this hard coded cryptographic key is only used during testing. Upon examination of the code in question it can be seen that the hardcoded cryptographic key in question is in assertion from a test. It is verifying that mock status event that is stored in the */github-client/src/test/resources/com/spotify/github/v3/activity/events/fixtures/status_event.json* file has the same cryptographic key after deserialization as it did before. This cryptographic key is not valid and therefore could not be used to cause actual damages.

File: ***/github-client/src/test/java/com/spotify/github/v3/activity/events/StatusEventTest.java***

```
34 @Test
35 public void testDeserialization() throws IOException {
36     String fixture =
37         Resources.toString(
38             getResource(this.getClass(), "fixtures/status_event.json"),
defaultCharset());
39     final StatusEvent statusEvent = Json.create().fromJson(fixture,
StatusEvent.class);
40     assertThat(statusEvent.context(), is("default"));
41     assertThat(statusEvent.sha(),
is("9049f1265b7d61be4a8904a9a27120d2064dab3b"));
42     assertThat(statusEvent.state(), is("success"));
43 }
```

The next finding for security rule CERT.MSC03-HCCK-1 was located on line 43 of the */github-client/src/test/java/com/spotify/github/v3/prs/ReviewTest.java* file. Much as the previous finding, the name of the file including the word test along with being stored in the */github-client/src/test* directory indicates that this file is likely only used for and during testing. Examining the code in question illuminates the fact the that the hard coded cryptographic key is in an assertion for a test. The test is verifying that the mock review object stored in the */github-client/src/test/resources/com/spotify/github/v3/prs/review.json* file has the expected commit ID after deserialization. Since this hardcoded ID is not an actual commit ID, revealing this information to an attacker does not pose a threat.

File: ***/github-client/src/test/java/com/spotify/github/v3/prs/ReviewTest.java***

```
34 @Test
35 public void testDeserialization() throws IOException {
36    String fixture =
37    Resources.toString(
38       getResource(this.getClass(), "review.json"),
39          defaultCharset());
40    final Review review =
41       Json.create().fromJson(fixture, Review.class);
42    assertThat(review.state(), is(ReviewState.APPROVED));
43    assertThat(review.commitId(),
is("ecdd80bb57125d7ba9641ffaa4d7d2c19d3f3091"));
44    assertThat(review.id(), is(80));
45 }
```

The next two findings are both located in the */github-client/src/test/java/com/spotify/github/v3/repos/PushCommitTest.java* file on lines 48 and 49. Like the previous two findings, the fact that the source of these flags are in the */github-client/src/test* directory with file names including the word test indicate that these are likely not actual hardcoded cryptographic keys, but mock ones used for testing. Examination of the code reveals this to be the same case as the last two findings with the hardcoded cryptographic keys really just being assertions for the pushCommit objects id and treeID method results after deserialization.

File: ***/github-client/src/test/java/com/spotify/github/v3/repos/PushCommitTest.java***

```
44 @Test
45 public void testDeserialization() throws IOException {
46    final PushCommit pushCommit = Json.create().fromJson(fixture,
PushCommit.class);
47    assertThat(pushCommit.modified().get(0), is("README.md"));
48    assertThat(pushCommit.id(), is("0d1a26e67d8f5eaf1f6ba5c57fc3c7d91ac0fd1c"));
49    assertThat(pushCommit.treeId(),
is("f9d2a07e9488b91af2641b26b9407fe22a451433"));
50    assertThat(pushCommit.message(), is("Update README.md"));
51 }
```

(CERT.NUM12.CLP-2) Do Not Cast Primitive Data Types to Lower Precision

Security rule CERT.NUM12.CLP-2 was identified as having seven findings by the Parasoft Jtest scan ran by Alexander Marquis. This security flaw is described by the Parasoft Jtest scan as any instance of casting a primitive data type into a lower precision data type. This security flaw is based off of SEI CERT rule *NUM12-J. Ensure conversions of numeric types to narrower types do not result in lost or misinterpreted data* (Svoboda *NUM12-J*). This rule states that since casting from a higher primitive type to a lower primitive type may result in the loss or misinterpretation of data, that all narrowing conversions should be guaranteed safe by range-checking the value before being cast (Svoboda *NUM12-J*). It specifies that primitive conversion are allowed in cases where the value of the wider type is within the range of the narrower type (Svoboda *NUM12-J*). The risk analysis performed by SEI CERT found vulnerabilities that stem from a violation of this rule have a severity of low, a likelihood of unlikely, and a remediation of cost of medium causing them to be classified as a priority of P12 Level L3.

SEI CERT Risk Assessment for Violations of Rule SEC05-J				
Severity	**Likelihood**	**Remediation Cost**	**Priority**	**Level**
High	Probable	Medium	P12	L1

All seven findings for this security flaw can be sourced to the */github-client/src/main/java/com/spotify/github/v3/clients/PKCS1PEMKey.java* file on lines 71 through 74. Examination of these lines allows us to see that all converted values are only two hexadecimal characters long meaning that they can definitely fit within the narrower byte type. Since the values being casted into a narrower type will definitely fit within the range of that narrower type this fits the exception of the SEI CERT rule and is unlikely to cause problems.

File: ***/github-client/src/main/java/com/spotify/github/v3/clients/PKCS1PEMKey.java***

```
67 private static byte[] toPkcs8(final byte[] pkcs1Bytes) {
68    final int pkcs1Length = pkcs1Bytes.length;
69    final int totalLength = pkcs1Length + 22;
70       byte[] pkcs8Header = new byte[] {
71          0x30, (byte) 0x82, (byte) ((totalLength >> 8) & 0xff), (byte)
(totalLength & 0xff), // Sequence + total length
72          0x2, 0x1, 0x0, // Integer (0)
73          0x30, 0xD, 0x6, 0x9, 0x2A, (byte) 0x86, 0x48, (byte) 0x86, (byte) 0xF7,
0xD, 0x1, 0x1, 0x1, 0x5, 0x0, // Sequence: 1.2.840.113549.1.1.1, NULL
74          0x4, (byte) 0x82, (byte) ((pkcs1Length >> 8) & 0xff), (byte)
(pkcs1Length & 0xff) // Octet string + length
75 };
```

Conclusion

As can be seen by the review of the findings from the Parasoft Jtest report generated by running a static code analysis scan on the Spotify GitHub-Java-Client project, many of the findings are either not actively exploitable or are false positives. This is to be expected when running automated static code analysis as the software must choose between either being over or under active in its flagging of possible vulnerabilities. It is often decided to err on the side of caution and slightly over flag possible vulnerabilities and have the development team manually review these findings to determine the true vulnerabilities. This helps developers hone in on areas of interest that could be vulnerable however, detecting these false flags does cost developer time and resources. The balancing act that occurs when choosing how to setup the security criteria for an automated scan is complex and automated scans should never be trusted to identify all vulnerabilities within a project nor should all findings be inherently seen as being vulnerable.

While the Parasoft Jtest scan found many security flaws that are not actively exploitable, these flaws should not be discounted entirely. For example, the lack of logging found for the second finding under the CERT.ERR00.LGE-2 security rule is not actively exploitable on its own. However as the OWASP foundation has said “insufficient logging and monitoring is the bedrock of nearly every major incident” (*A10:2017-insufficient logging*). Proper logging can allow for attackers, vulnerabilities, and exploits to be detected and responded to in a timely manner and perhaps even before damages occur. Non-exploitable flaws are often just methods of accomplishing something that are overly ambiguous or complex causing unnecessary confusion and possible mistakes in the future. These flaws, while not as important as the actively exploitable vulnerabilities, are still important and mitigation of them should not be written off.

Of all of the findings reviewed in this report, the finding for security rule CERT.LCK06.INSTLOCK is the most important to mitigate should be assigned the highest priority. This is because this vulnerability is actively exploitable and the consequences for the exploit can be data integrity violations. This is because multiple instances of a singleton class can be created through this vulnerability. The multiple instances of what is supposed to be a singleton class can allow the attacker to manipulate read values and have access to values that should not be accessible.

References

"A10:2017-Insufficient Logging & Monitoring." *OWASP Top Ten 2017*, OWASP Foundation, 2017, https://owasp.org/www-project-top-ten/2017/A10_2017-Insufficient_Logging%2526Monitoring.

Flynn, Lori, et al. "Rules versus Recommendations (Java)." *Confluence*, 14 Apr. 2021, https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487386.

McManus, Joe, et al. "SEI CERT Oracle Coding Standard for Java." *Confluence*, 11 June 2018, https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java.

Mohindra, Dhruv, et al. "ERR00-J. Do Not Suppress or Ignore Checked Exceptions." *Confluence*, 6 Aug. 2021, https://wiki.sei.cmu.edu/confluence/display/java/ERR00-J.+Do+not+suppress+or+ignore+checked+exceptions.

Mohindra, Dhruv, et al. "EXP00-J. Do Not Ignore Values Returned by Methods ." *Confluence*, 24 Jan. 2022, https://wiki.sei.cmu.edu/confluence/display/java/EXP00-J.+Do+not+ignore+values+returned+by+methods.

Mohindra, Dhruv, et al. "LCK06-J. Do Not Use an Instance Lock to Protect Shared Static Data." *Confluence*, 15 Mar. 2022, https://wiki.sei.cmu.edu/confluence/display/java/LCK06-J.+Do+not+use+an+instance+lock+to+protect+shared+static+data.

Mohindra, Dhruv, et al. "MET02-J. Do Not Use Deprecated or Obsolete Classes or Methods." *Confluence*, 18 May 2021, https://wiki.sei.cmu.edu/confluence/display/java/MET02-J.+Do+not+use+deprecated+or+obsolete+classes+or+methods.

Mohindra, Dhruv, et al. "MSC03-J. Never Hard Code Sensitive Information." *Confluence*, 6 Aug. 2021, https://wiki.sei.cmu.edu/confluence/display/java/MSC03-J.+Never+hard+code+sensitive+information.

Mohindra, Dhruv, et al. "SEC05-J. Do Not Use Reflection to Increase Accessibility of Classes, Methods, or Fields." *Confluence*, 25 Jan. 2022, https://wiki.sei.cmu.edu/confluence/display/java/SEC05-J.+Do+not+use+reflection+to+increase+accessibility+of+classes%2C+methods%2C+or+fields.

Mohindra, Dhruv, et al. "SER03-J. Do Not Serialize Unencrypted Sensitive Data." *Confluence*, 6 Aug. 2021, https://wiki.sei.cmu.edu/confluence/display/java/SER03-J.+Do+not+serialize+unencrypted+sensitive+data.

Mohindra, Dhruv, et al. "SER07-J. Do Not Use the Default Serialized Form for Classes with Implementation-Defined Invariants." *Confluence*, 6 Aug. 2021, https://wiki.sei.cmu.edu/confluence/display/java/SER07-J.+Do+not+use+the+default+serialized+form+for+classes+with+implementation-defined+invariants.

Spotify. “Spotify/Github-Java-Client: A Java Client to Github API.” *GitHub*, 15 Apr. 2022, https://github.com/spotify/github-java-client.

Svoboda, David, et al. “How This Coding Standard Is Organized.” *Confluence*, 14 Apr. 2021, https://wiki.sei.cmu.edu/confluence/display/c/How+this+Coding+Standard+is+Organized.

Svoboda, David. “NUM12-J. Ensure Conversions of Numeric Types to Narrower Types Do Not Result in Lost or Misinterpreted Data.” *Confluence*, 18 Jan. 2022, https://wiki.sei.cmu.edu/confluence/display/java/NUM12-J.+Ensure+conversions+of+numeric+types+to+narrower+types+do+not+result+in+lost+or+misinterpreted+data.

Appendix A: Parasoft Jtest Report

PARASOFT

Parasoft Jtest Report

Jtest 2021.2.1

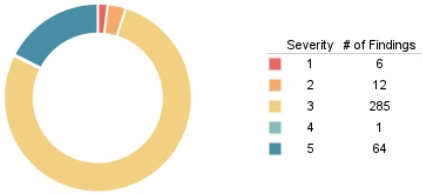
Session Summary

Build ID:	2022-04-18
Test Configuration:	builtin://CERT for Java
Started:	2022-04-18T09:08:32-06:00
Performed on:	Alex-Desktop by Turno_000
Session Tag:	${scontrol_branch}-win32_x86_64
Project:	

Static Analysis	Severity 1 Findings: 6

Summary - Static Analysis

Findings



Total Findings: 368

Details - Static Analysis

Static Analysis

Module	Findings			Files		Lines	
	suppressed	total	per 10,000 lines	checked	total	checked	total
github-client	0	368	250	189	189	14671	14671
Total [0:00:07]	**0**	**368**	**250**	**189**	**189**	**14671**	**14671**

All Findings by Severity

Category | Severity

[6] Severity 1 - Highest
- [4] Avoid using hard-coded cryptographic keys (CERT.MSC03.HCCK-1)
- [2] Inspect instance fields of serializable objects to make sure they will not expose sensitive information (CERT.SER03.SIF-1)

[12] Severity 2 - High
- [2] Ensure all exceptions are either logged with a standard logger or rethrown (CERT.ERR00.LGE-2)
- [1] Do not use an instance lock to protect shared static data (CERT.LCK06.INSTLOCK-2)
- [9] Do not cast primitive data types to lower precision (CERT.NUM12.CLP-2)

[285] Severity 3 - Medium
- [1] Use a caught exception in the "catch" block (CERT.ERR00.UCATCH-3)
- [7] Do not throw exception types which are too general or are unchecked exceptions (CERT.ERR07.NTERR-3)
- [19] Avoid declaring methods to throw general or unchecked Exception types (CERT.ERR07.NTX-3)
- [4] Ensure method and constructor return values are used (CERT.EXP00.NASSIG-3)
- [1] Do not use deprecated APIs (CERT.MET02.DPRAPI-3)
- [2] Do not perform bitwise and arithmetic operations on the same data (CERT.NUM01.NCBAV-3)
- [45] Provide mutable classes with copy functionality (CERT.OBJ04.MUCOP-3)
- [45] Provide mutable classes with copy functionality (CERT.OBJ05.MUCOP-3)
- [45] Provide mutable classes with copy functionality (CERT.OBJ06.MUCOP-3)
- [7] Create a 'serialVersionUID' for all 'Serializable' classes (CERT.SER00.DUID-3)
- [1] Define a "readResolve" method for all instances of Serializable types (CERT.SER07.RRSC-3)
- [41] Do not use String concatenation in an Internationalized environment (CERT.STR00.COS-3)
- [67] Use the optional java.util.Locale parameter (CERT.STR02.CCL-3)

[1] Severity 4 - Low
- [1] Avoid using reflection methods (CERT.SEC05.ARM-4)

[64] Severity 5 - Lowest
- [64] Make your classes noncloneable (CERT.OBJ07.MCNC-5)

Appendix B: Project Import into IntelliJ