**Project 2 Report**

**CS 40800**

Group 13:

Connor Brown: [brow1325@purdue.edu](mailto:brow1325@purdue.edu)

Jakob Molskness: [jmolskn@purdue.edu](mailto:jmolskn@purdue.edu)

Nathan Adduci: [adducin@purdue.edu](mailto:adducin@purdue.edu)

**Part 1**

**(b) Finding and Explaining False Positives**

In general, one reason false positives occur under this strategy is because static analysis has a limited sample size from which to determine which pairs of functions are commonly called together. With no training data, it is very difficult to distinguish between pairs that are actually connected, and pairs of functions that are individually common but have very little to do with one another. For example, Java’s String.substring() and String.indexOf() are both extremely common and could easily exceed the confidence threshold for each other, but having one without the other would very rarely indicate a bug. Because static analysis cannot tell the intent behind function calls, it naturally identifies false positives when analyzing code. Another reason false positives occur is the similarity between code that is buggy and code that simply employs bad practices. Again, because both deviate from what would normally be observed, without knowing the intent behind a section of code it is difficult to know which is which. For example, when working in an environment that employs a logging system, any scope that contains a file I/O function will almost certainly also contain a call to the logger incase of an IOException. If one such scope did not have a logging statement, it would be reported as a bug despite not affecting the behavior of the program from the user’s perspective. In this scenario, it is impossible to know whether the logging statement was left off intentionally or not. The default behavior should be to report the potential bug and let the developer decide, but this uncertainty will inevitably cause some false positives to be reported as well.

The first false positive we identified from httpd is the pair (strlen, strcmp). One instance of this occurs on line 5311 of “server/mpm/core.c:core\_dump\_config()”, in which strncmp is called without strlen also being called. This is not a bug because strncmp is being called against a string literal, so the max length of comparison is always known. When the second string being compared is shorter than this, strncmp() will stop at the null terminator, giving well defined and expected behavior without needing to call strlen. Additionally, although it will cause an illegal memory access if the string is not null terminated, calling strlen() on it would have the same result, so this issue is not resolved by including a call to strlen(). This pair caused three bug reports, all of a similar nature.

The second false positive we identified from https is the pair (apr\_array\_make, apr\_array\_push). One instance of this occurs on line 257 of modules/core/test/mod\_so.c in the function load\_module(). Because this is a test directory, many of the resources are generated before the tests themselves are executed. In this case, the apr\_array is created via a function called on line 438, outside the scope of load\_module but still always upstream of it in the control flow of the program. This is a false positive not because the two functions shouldn’t be called together, but because pipair didn’t know they actually are called together, because of the convoluted flow of the test files. Expanding the scope as is done in part c may help to eliminate this being provided as a false positive. This pair produced 19 unique bug reports, most of which occured in test files similar to this one.

Pairs from Call Graph for (strlen, strcmp):

* bug: strncmp in core\_dump\_config, pair: (strlen, strncmp), support: 7, confidence: 70.00%
* bug: strncmp in core\_check\_config, pair: (strlen, strncmp), support: 7, confidence: 70.00%
* bug: strncmp in ap\_core\_translate, pair: (strlen, strncmp), support: 7, confidence: 70.00%

Pairs from Call Graph for (apr\_array\_make, apr\_array\_push):

* bug: apr\_array\_push in ap\_copy\_method\_list, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 85.71%
* bug: apr\_array\_push in load\_module, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 85.71%
* bug: apr\_array\_push in ap\_add\_per\_dir\_conf, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 85.71%
* bug: apr\_array\_make in ap\_init\_virtual\_host, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 94.38%
* bug: apr\_array\_push in ap\_regname, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 85.71%
* bug: apr\_array\_push in ap\_if\_walk\_sub, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 85.71%
* bug: apr\_array\_push in set\_server\_alias, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 85.71%
* bug: apr\_array\_push in ap\_file\_walk, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 85.71%
* bug: apr\_array\_push in set\_define, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 85.71%
* bug: apr\_array\_push in uniq\_field\_values, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 85.71%
* bug: apr\_array\_make in so\_sconf\_create, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 94.38%
* bug: apr\_array\_make in create\_core\_server\_config, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 94.38%
* bug: apr\_array\_make in ap\_make\_method\_list, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 94.38%
* bug: apr\_array\_push in set\_protocols, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 85.71%
* bug: apr\_array\_push in ap\_method\_list\_add, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 85.71%
* bug: apr\_array\_push in ap\_location\_walk, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 85.71%
* bug: apr\_array\_make in prep\_walk\_cache, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 94.38%
* bug: apr\_array\_push in ap\_directory\_walk, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 85.71%
* bug: apr\_array\_push in ap\_add\_per\_url\_conf, pair: (apr\_array\_make, apr\_array\_push), support: 84, confidence: 85.71%

**(c) Inter-Procedural Analysis**

We decided to do a program which expands the scope of our coverage graph. In order to accomplish this, we added an expansion to the original coverage graph in the method getExpandedGraph. For every node, it looped through the HashSet which contained all its child nodes, then added the child nodes of those child nodes in the original coverage graph. This is repeated E times, where E is the requested expansion depth from the command line arguments. This solution didn’t reduce the chance of false positives, but did increase the bug detection rate substantially.

In order to test this, we ran a series of cases with increasing expansion depth, starting at 0, 1, 2, 10 and going up to 100. By the time we reached 10, we had expanded our scope to its maximum, so we didn’t feel the need to expand past 100. We took the average runtime of 5 runs using System.currentTimeMillis();

|  |  |  |  |
| --- | --- | --- | --- |
| Levels of Expansion | Avg Runtime (ms) | Bugs @ 3, 65% | Bugs @ 10, 80% |
| 0 | 177.6 | 253 | 25 |
| 1 | 561.2 | 2982 | 141 |
| 2 | 1608 | 14790 | 3211 |
| 10 | 2690.2 | 32258 | 8853 |
| 100 | 2831.4 | 32258 | 8853 |

The data from our table shows that the average runtime as we expand grows quickly. Our bugs also continue to grow until we reach the max depth at which point the amount of bugs does not change. Expanding any further does not result in different call graphs. This helps to show that by expanding our thresholds we can find more bugs and be more confident that they are not false positives.

**(d) Improving the Solutions**

A solution to reduce false positives would be to consider the order functions are called in as a factor in calculating confidence. If two arbitrary functions are in fact correlated, they will almost always be called in a particular order. This can occur either because one utilizes the output of the other or because one’s operations must occur before the second’s. Park et al stated in 2016 in their work on improving JavaScript bug detectors that “through preliminary experiments we have found for the correctness properties that we focus on, the execution order of event handlers is often not crucial for the analysis precision.” They go on further to explain that this helped them to find and reduce the number of false positives that resulted from analyzing the order.

Najako and Kume found in 1991 in their work on analyzing the order of operations in software that, “when a group of results has theoretically no correlation with a group of causes, it is appropriate to independently classify the causes corresponding to each group of results.” Similarly, under this method we aim to identify when two functions should be considered independent or correlated, as this can provide valuable insight into whether a deviation from this order should be considered a bug.

Our implementation is to use the idea of the order to determine the likelihood that the bug detected is a false positive or not. By having a check where the pair order of A,B does not matter and it is over the total number of occurrences of the pair (A,B). This gives us a formula of that we use to determine a likelihood of false positive.

Works Cited

Baloglu, B. (2016). How to find and fix software vulnerabilities with coverity static analysis. *2016 IEEE Cybersecurity Development (SecDev)*. doi: 10.1109/secdev.2016.041. Retrieved from <https://insights.sei.cmu.edu/sei_blog/2016/06/prioritizing-alerts-from-static-analysis-to-find-and-fix-code-flaws.html>

Nakajo, T., & Kume, H. (1991). A case history analysis of software error cause-effect relationships. *IEEE Transactions on Software Engineering*, *17*(8), 830–838. doi: 10.1109/32.83917. Retrieved from <https://www.computer.org/csdl/journal/ts/1991/08/e0830/13rRUwjXZLo>

Nam, J., Wang, S., Xi, Y., & Tan, L. (2019). A bug finder refined by a large set of open-source projects. *Information and Software Technology*, *112*, 164–175. doi: 10.1016/j.infsof.2019.04.014. Retrieved from <https://www.cs.purdue.edu/homes/lintan/publications/feefin-ist19.pdf>

Park, J., Lim, I., & Ryu, S. (2016). Battles with false positives in static analysis of JavaScript web applications in the wild. *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE 16*. doi: 10.1145/2889160.2889227. Retrieved from <https://ieeexplore.ieee.org/abstract/document/7883289>

Pradel, M., & Gross, T. R. (2012). Leveraging test generation and specification mining for automated bug detection without false positives. *2012 34th International Conference on Software Engineering (ICSE)*. doi: 10.1109/icse.2012.6227185. Retrieved from <https://dl.acm.org/doi/abs/10.5555/2337223.2337258>

**Part 2**

1. **Resolving bugs in Apache Tomcat**

**Outstanding Issue #1:**

|  |  |
| --- | --- |
| **File:** | org.​apache.​catalina.​ha.​session.​DeltaSession |
| **Function:** | setMaxInactiveInterval(int, boolean) |
| **Category** | Concurrent data access violations |
| **Classification:** | Bug |
| **Justification:** | (Faulty lines: 280-282) The statement in question first checks whether an object is null, and if it is not, obtains a lock calls one of the object’s member functions. If the object is set to null between the time the check is made and the time the program obtains the lock, the function call will generate a NullPointerException. Since the try{} wrapping the operation does not include a catch{} clause, we believe this behavior is unintentional and constitutes a bug. To fix this, the lock should be obtained before checking if the object is null (essentially moving the if() inside the try{} block). |

**Outstanding Issue #2:**

|  |  |
| --- | --- |
| **File:** | org.​apache.​catalina.​session.​JDBCStore |
| **Function:** | remove(java.​lang.​String) |
| **Category** | Null pointer dereferences |
| **Classification:** | False positive |
| **Justification:** | The code does a check to ensure an object is not null, then calls a function if so. This appears to be the correct behavior, as, unless the object is being concurrently written to, the reference will always be non-null when used. Even if it is not, however, the close() function itself has defined behavior for when the reference is null. Because of this, we believe there is no bug or defect present and the outstanding issue is a result of a false positive. |

**Outstanding Issue #3:**

|  |  |
| --- | --- |
| **File:** | org.​apache.​catalina.​ha.​deploy.​FarmWarDeployer |
| **Function:** | start() |
| **Category** | Null pointer dereferences |
| **Classification:** | Intentional |
| **Justification:** | The non-null checks are correctly done, so there is no risk of a NullPointerException being generated in this code segment. However, when both engine and host are non-null, the constructor will be called twice with only the second result being used. This fall through may be intentional, though it is hard to tell. While not directly harmful to the program’s state, this does affect performance. To fix this, the if() on line 177 should be changed to an else if() to ensure only one is executed. Additionally, if the second is preferred to the first when both are possible (as would happen in the fall-through case), the order of the if() statements should be reversed after doing the else if(). |

**Outstanding Issue #4:**

|  |  |
| --- | --- |
| **File:** | org.​apache.​catalina.​tribes.​group.​ChannelCoordinator |
| **Function:** | sendMessage(org.​apache.​catalina.​tribes.​Member[], org.​apache.​catalina.​tribes.​ChannelMessage, org.​apache.​catalina.​tribes.​group.​InterceptorPayload) |
| **Category** | Class hierarchy inconsistencies |
| **Classification:** | False positive |
| **Justification:** | The function in question calls the sendMessage member function of the ChannelSender class, while the superclass function calls the sendMessage found in the ChannelInterrupter class. This difference seems to have been done intentionally to get the different behavior of the two classes, making this a false positive. |

**Outstanding Issue #5:**

|  |  |
| --- | --- |
| **File:** | org.​apache.​tomcat.​util.​buf.​MessageBytes |
| **Function:** | equalsIgnoreCase(java.​lang.​String) |
| **Category** | Null pointer dereferences |
| **Classification:** | Bug |
| **Justification:** | (Faulty lines: 330-331) When both strValue and s are null, the if statement on line 330 evaluates to false. As a result line 331 executes, which calls a member function of strValue. Because strValue is null in this case this will cause a NullPointerException, which is a bug. This bug can be fixed by checking whether strValue on its own is null before executing line 331. |

**Outstanding Issue #6:**

|  |  |
| --- | --- |
| **File:** | org.​apache.​jk.​common.​ChannelUn |
| **Function:** | init() |
| **Category** | Null pointer dereferences |
| **Classification:** | False positive |
| **Justification:** | If apr is null, the or operation inside the if() will short circuit before dereferencing apr. Because of this, there is no risk of a NullPointerException being thrown from this line. Taking advantage of short circuiting like this can frequently reduce code complexity and for this reason should not be considered bad practice, making this a false positive. |

**Outstanding Issue #7:**

|  |  |
| --- | --- |
| **File:** | org.apache.jasper.compiler.JDTCompiler$#1 |
| **Function:** | isPackage(java.lang.String) |
| **Category** | Resource leaks |
| **Classification:** | Bug |
| **Justification:** | (Faulty line: 235) The InputStream object “is” may have been initialized when it goes out of scope on line 235. However, the member function close() is not called on it before returning. In Java, calling close() is required on initialized InputStream objects per the Eclipse documentation. If this is not done, over time this could lead to an OutOfMemoryException being thrown, which is a bug. To fix this, the program should evaluate the value of “is” then, if necessary, call is.close() before returning. |

**Outstanding Issue #8:**

|  |  |
| --- | --- |
| **File:** | org.​apache.​catalina.​ha.​deploy.​FarmWarDeployer |
| **Function:** | start() |
| **Category** | Null pointer dereferences |
| **Classification:** | False positive |
| **Justification:** | The null check in question returns, so in all remaining execution paths left in the function, the value of the object engine is non-null. So, when it is dereferenced on line 159, it is guaranteed to be non-null, making this a false positive. |

**Outstanding Issue #9:**

|  |  |
| --- | --- |
| **File:** | org.​apache.​catalina.​core.​StandardWrapper |
| **Function:** | getServlet() |
| **Category** | Concurrent data access violations |
| **Classification:** | False positive |
| **Justification:** | It is not possible to add a lock to the function in question without significantly altering the behavior of the function. Instead, the lock should be around the places that call the function (which, from what I could find, it is). Encapsulation like this is generally good practice, so this is a false positive. |

**Outstanding Issue #10:**

|  |  |
| --- | --- |
| **File:** | org.​apache.​naming.​resources.​VirtualDirContext |
| **Function:** | scanForTlds(java.​io.​File) |
| **Category** | Null pointer dereferences |
| **Classification:** | Bug |
| **Justification:** | (Faulty lines: 192-193) The method listfiles() has a defined behavior of returning null in some cases. However, its result is immediately dereferenced on line 193, which, if null was returned, will cause a NullPointerException. This exception is not caught, making it a bug. |

**Outstanding Issue #11:**

|  |  |
| --- | --- |
| **File:** | org.​apache.​catalina.​ha.​tcp.​SimpleTcpCluster |
| **Function:** | messageReceived(org.​apache.​catalina.​ha.​ClusterMessage) |
| **Category** | Null pointer dereferences |
| **Classification:** | Bug |
| **Justification:** | When message is null, accepted will always be false and lines 912 and 913 will be executed whenever debugging is turned on. Because message is dereferenced in these lines, this situation will cause a NullPointerException. Since the value of message was checked to be non-null in other places, we do not believe this to be intentional, making this a bug. This can be fixed by checking that null is not null before executing the debugging statements. |

**Outstanding Issue #12:**

|  |  |
| --- | --- |
| **File:** | org.​apache.​catalina.​ssi.​SSIServletExternalResolver |
| **Function:** | getServletContextAndPathFromVirtualPath(java.​lang.​String) |
| **Category** | Null pointer dereferences |
| **Classification:** | Bug |
| **Justification:** | (Faulty line: 419) The code passes the return value function getContextPath, which may be null, to the method getPathWithoutContext. This, in turn, passes its first argument to Java’s String.startsWith() method, which will throw a NullPointerException if the given argument is null. As a result, when getContextPath returns null an uncaught exception will be thrown, which is a bug. This can be fixed by checking that getContextPath did not return null before passing it on. |

**Outstanding Issue #13:**

|  |  |
| --- | --- |
| **File:** | org.​apache.​jk.​common.ChannelNioSocket |
| **Function:** | ​init() |
| **Category** | Resource leaks |
| **Classification:** | False positive |
| **Justification:** | Coverity is arguing that an instance of the ServerSocketClass class needs to be closed before it goes out of scope. However, the ServerSocketClass doesn’t have any member method that would do this, making this impossible to do and as such not a bug. |

**Outstanding Issue #14:**

|  |  |
| --- | --- |
| **File:** | org.​apache.​catalina.​ha.​session.​SimpleTcpReplicationManager |
| **Function:** | messageReceived(org.​apache.​catalina.​ha.​session.​SessionMessage, org.​apache.​catalina.​tribes.​Member) |
| **Category** | Null pointer dereferences |
| **Classification:** | Bug |
| **Justification:** | (Faulty line: 614) If an exception is encountered in the readSession function, it returns null. This null value is then dereferenced on line 614, which will cause a NullPointerException. This can be fixed by checking that session does not equal null along with the other check on line 613. |

**Outstanding Issue #15:**

|  |  |
| --- | --- |
| **File:** | org.​apache.​catalina.​ha.​session.​DeltaRequest. |
| **Function:** | readExternal(java.​io.​ObjectInput) |
| **Category** | Null pointer dereferences |
| **Classification:** | Bug |
| **Justification:** | (Faulty lines: 234, 238) The check for non-null on line 238 implies action can be null and that if it is, this will not result in desired behavior (since it is given a value if null). However, reset(), which is called on line 234, dereferences action. So, if it is possible for it to be null on line 238, it will have already caused a NullPointerException on line 234. If it is not, the check is unnecessary and slows performance. In the first scenario, this can be fixed by checking that action is not null before calling reset. In the second, the check should be removed from line 238. |

**Outstanding Issue #16:**

|  |  |
| --- | --- |
| **File:** | org.​apache.​jk.​server.​JkMain |
| **Function:** | ​loadPropertiesFile() |
| **Category** | Resource leaks |
| **Classification:** | Bug |
| **Justification:** | (Faulty line: 455) FileInputStream objects must be closed in Java; line 455 of the file calls the FileInputStream constructor without saving the result, making it impossible to close and thus creating a memory leak. To fix this, the return value should be stored in an object then closed once load() returns. |

**Outstanding Issue #17:**

|  |  |
| --- | --- |
| **File:** | org.​apache.​catalina.​ssi.​SSIProcessor |
| **Function:** | process(java.​io.​Reader, long, java.​io.​PrintWriter) |
| **Category** | Null pointer dereferences |
| **Classification:** | Bug |
| **Justification:** | (Faulty lines: 127-128) The function parseCmd returns null when command is null. When it does so, the dereference on lines 127 and 128 will generate a NullPointerException. To fix this, the result of line 121 should be checked to be non-null before executing line 127. |

**Outstanding Issue #18:**

|  |  |
| --- | --- |
| **File:** | org.​apache.​jasper.​compiler.​JspDocumentParser |
| **Function:** | startElement(java.​lang.​String, java.​lang.​String, java.​lang.​String, org.​xml.​sax.​Attributes) |
| **Category** | Null pointer dereferences |
| **Classification:** | Bug |
| **Justification:** | (Faulty line: 274) The function checkPrefices(), called on line 274, dereferences the passed value of attrs without catching any exceptions. However, the check on line 294 implies that there is a valid chance that attrs is null. So, if attrs can be null this state will generate an Exception, and if it cannot be then the check is unnecessary and adversely impacts performance. To fix this, either the check for null should be removed or line 274 should be wrapped in a check for whether attrs is null. |

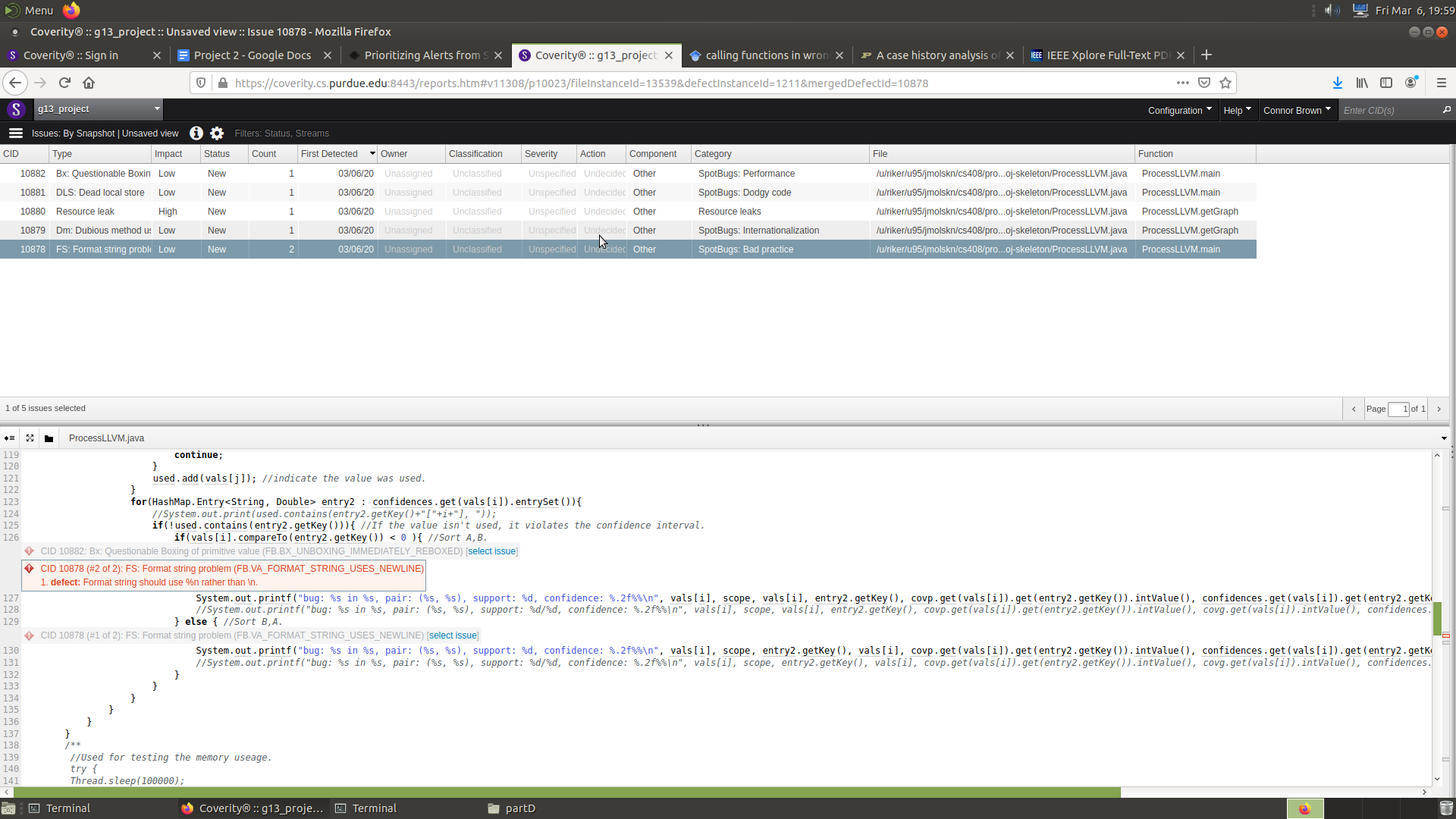
**(b) Analyzing Your Own Code**

One bug reported by Coverity, shown in Figure 1, was our use of “\n” in two print format strings (lines 127 and 130) instead of “%n”. We as a group had never used the percent instead of the backslash, and were confused about why this would be a problem given this is how Purdue teaches it. However, after further research it turns out the official Oracle JavaSE Number documentation does recommend using %n for portability reasons. While \n works fine on UNIX systems, it can produce unexpected behavior on Windows and other NT kernel systems, while %n will perform as expected in both environments. To fix this, we would replace the two instances of \n with %n in order to improve the portability of our code.

Another bug Coverity reported in Figure 2 was a resource leak caused by not closing an open BufferedReader. This is something that absolutely should be done which we simply forgot to do. To fix this, we would add a line just before the return statement of getGraph(), which reads in the data from the output of opt, to close the reader (e.g. reader.close()). This would solve the issue and prevent any memory leaks. While Java expects all input and output readers to be closed after they are no longer needed, this is easy to forget and does cost resources if the program is long running. While this fix is relatively unimportant in our use case, it is easy to imagine scenarios where it could be performance critical, making this a valuable catch by Coverity.

Overall, we were very impressed with Coverity’s ability to identify and explain problems in code that we originally thought to be bug free. Both of these are bugs that unit testing would have been unlikely to uncover, especially if they were run only in a UNIX environment.

**Figure 1:**



**Figure 2:**

