Anthony Wilcox
Benjamin Tullis
Dalena Pham

CS 362 - URLValidator Bug Report and Process

Note to the grader: All tests, including unit tests, are found in UrlValidatorTest.java

Testing Methodology

As per specifications, we first conducted manual testing as you'll see outlined further in our Manual Testing section. We then conducted ad hoc code reviews on an individual basis then compared our finding during our weekly meetings held over Google Hangouts. Following this we conducted white box unit testing, and non-functional unit testing through automated tests inside of the mandated domain partitioning using known true and false url portion-values, and then passed their components to the 'isvalid' function (unit test through three domain partitions). Finally ending by asserting our known results as either true or false.
In addition to implementing manual tests, ad hoc code review, and code review comparisons, we used the outcome of our code reviews to create automated test results inside of our domain partitions that would better lend themselves to fault localization based upon our inspection. We then ran our automated unit tests to find bugs still not located up to this point in our testing. Much of our most helpful feedback was obtained by examining associated gcov files after having used large 'n' values to ensure code coverage. This gave us information on what code we were missing and how we could refine our tests further.

Manual Testing

In manual testing, we attempted a variety of invalid and valid URLs.
Examples of valid URLs we tested:
- http://www.amazon.com
- http://somewhere.com/pathxyz/file(1).html
- http://www.google.com:80/test1?action=view

Examples of invalid URLs we tested:
- example.com
- example.c
- file://C:\\some.file

Failure 1 Found

When doing manual testing, we noticed two particular failures:

```
Expected: true Returned: false
Failed Test: http://www.google.com:80/test1?action=view
Expected: true Returned: false
Failed Test: ftp://www.google.com:80/test1?action=view
```

Debugging Process

1. Created a breakpoint for debugging at the first test failure :
   a. `manualHelper(true, urlVal, "http://www.google.com:80/test1?action=view");`
2. Ran Debug of URLValidatorTest as a JUnit Test
3. Step into `manualHelper`
4. Step into line:
   a. `boolean result = urlVal.isValid(testURL);`
5. Discover within URLValidator method `isValid` line 315, `isValidQuery` returns a false value instead of an expected true value:
   a. `if (!isValidQuery(urlMatcher.group(PARSE_URL_QUERY))) {`
6. Step into `isValidQuery` to investigate
7. Debug details: `variable query : “action=view”(id=46)`
8. Line 447 of URLValidator in method `isValidQuery` produces incorrect output:
   a. `return !QUERY_PATTERN.matcher(query).matches();`
   b. Returns false instead of true (reverses correct value) because if the query matches the `QUERY_PATTERN` than the result should be true, not false.
9. Finish Debugging, change line 447 to:
   a. `return QUERY_PATTERN.matcher(query).matches();`
10. Fixing this line produces expected results for both previous failed tests:
    a. `Expected: true Returned: true`
    b. `Expected: true Returned: true`

Input Partitioning/ Unit Tests

We did two partitions with the first partition testing and expecting only false results from `isValid()` and the second partition expecting only true results. In other words, we tested invalid URLs and valid URLs. We felt it was redundant to do manual testing with input partitioning and aimed our input partitioning tests to also be programming based unit tests that build URLs that fit into the two partitions of either valid or invalid.

Invalid URLs

In our first partition, we iterated over the length of each false object array pairing it with true members from each other object arrays to build URL’s. To help uncover edge cases, we used a for loop for each object array. Then, we partitioned the false values only into that array, concatenating the rest of the string url with TRUE values but finally asserting that the url as a whole is still false. We saw that the multiple for loops were necessary because we paired each false result pair object with all true objects and asserted that it is still false. In this way, fault localization becomes easier as we would have better specifications for assertion failures.

Valid URLs
Our attack plan to uncover the widest breadth of valid URLs was much easier as we are not switching true and false statements as in our first partition. We felt only one for loop was needed to build the Valid URLs. We tried to string the longest possible object array, iterating over all it's possible indices and then combining that with the components in the other object arrays. In this way, it is possible to check out some boundary cases.

Failure 2 Found
While testing invalid URLs, we found the following line failed it's assertion:
`junit.framework.AssertionFailedError: Should be invalid:`
`http://256.256.256.256:80/test1?action=view`

This URL should evaluate to FALSE however it returns a TRUE result.

Debugging Process
1. Created breakpoint where assertion failure manifested:
   a. `assertFalse("Should be invalid: " + buffer, urlVal.isValid(buffer.toString()));`
   b. This is particularly found when the "authority" url component is supposed to be false.
2. Ran Debug of URLValidatorTest as a JUnit Test
3. Step into `isValid`
   a. Value being passed is: "`http://256.256.256.256:80/test1?action=view`"
   b. Scheme: `"http"`
   c. Authority: `"256.256.256.256:80"`
4. Step into `isValidAuthority` passed authority from above
   a. Expect that `isValidAuthority` should return FALSE, because only the authority portion of URL is false.
5. Walk through `isValidAuthority`, eventually stepping into line 386 of UrlValidator:
   a. `if (!inetAddressValidator.isValid(hostLocation)) {`
6. Step into line 64 of InetAddressValidator:
   a. `return isValidInet4Address(inetAddress);`
   b. passed value of `"256.256.256.256"`
7. Enter for loop to `"// verify that address subgroups are legal"` on line 80
8. Line 94, enter conditional: `if (iIpSegment > 255)`

9. Line 96 returns true if the condition is true when it should be returning false.
    a. Source: “Networks with subnet masks of at least 24 bits, i.e. Class C networks in classful networking, and networks with CIDR suffixes /24 to /32 (255.255.255.0–255.255.255.255) may not have an address ending in 0 or 255.” - IPv4 wikipedia page
10. Exit debugging, change line 96 from `return true` to `return false`

Failure 3 Found

In the same vein, when testing valid URLs, the following line also failed it’s assertion:

```
junit.framework.AssertionFailedError: Should be valid:
ftp://go.com:65535/test1?action=edit&mode=up
```

This URL should evaluate to TRUE but `isValid()` returns a FALSE value.

Debugging Process

1. Created breakpoint where assertion failure manifested:
    a. `assertTrue("Should be valid: " + buffer, urlVal.isValid(buffer.toString()));`
2. Ran Debug of URLValidatorTest as a JUnit Test
3. Step into `isValid`
    a. Value being passed is: “`ftp://go.com:65535/test1?action=edit&mode=up`”
4. In this Unit Test Partition, all values are supposed to true. So, we look for abnormal behavior where “`return false`” is invoked.
5. Eventually step into `isValidAuthority`
    a. Where `authority = “go.com:65535”`
6. Notice that lines 393 - 397 of UrlValidator.java causes `isValidAuthority` within `isValid` to return false - behavior that my be triggered by a bug:

```
a.      if (port != null) {
b.          if(!PORT_PATTERN.matcher(port).matches()){
c.              return false;
d.          }
e.      }
```

    f. where `port = :65535`
7. Check out `PORT_PATTERN` on line 159 of UrlValidator.java:
    a. `private static final Pattern PORT_PATTERN = Pattern.compile(PORT_REGEX);`
8. Investigate `PORT_REGEX` from line 158 of UrlValidator.java:
    a. `private static final String PORT_REGEX = "^:(\\d{1,3})$";`
        i. This line would suggest that the largest port number can only have at max 3 digits, however this is wrong as the largest port number

is 65535 (the largest port number is an unsigned short 2^16-1: 65535), which has 5 digits.

   ii. Therefore, we suspect this is the buggy code.

9. Exit debugging, change line 158 from:
   a. `private static final String PORT_REGEX = "^:(\\d{1,3})$";` to:
   b. `private static final String PORT_REGEX = "^:(\\d{1,5})$";`

Teamwork

To collaborate, our team mostly made use of various Google tools. We mainly used Google Drive to store our codes and documents and communicated via Google Hangouts. After establishing ourselves as a group, we set up a meeting time to layout our goals and plan for this project. It was decided that we would divide our work based on each group member's strengths and comfort level of doing certain tasks. Benjamin is proficient with debugging, Anthony with writing test code and Dalena with documentation. While these were the main roles and responsibilities we assigned ourselves, no one was limited to doing just one task and often helped other members with coding, debugging and documentation where necessary.

When writing test code, one person was responsible for a portion of the required test cases (manual testing, input partitioning, etc). To begin with, we as a team would get together on Google Hangouts and discuss and write pseudocode for what we would expect a test to look like. Then the person in charge of writing the test case would translate the pseudocode into working code, releasing their rough draft to the group. We would all then jump in on google docs and add/fix what we felt was needed before official tests were run. Then we would run tests, document if we saw any strange behavior and discuss/share if we had truly found a bug. If we thought we found a bug, the debugger would step in and localize the exact line where the bug took place. Typically, this sometimes involved 2 or more members debugging the same test code because some bugs are hard to find! Finally, a bug report from the member who successfully found the bug was done. Ideally, we wanted each person to find at least one bug, and such was the case.

In summary our division of work involved assigning ‘experts’ for the main responsibilities of the work, discussing closely with each other and collaboratively stepping in help find the bugs.

Debugging + Agan’s Rules

1. Understand the system

At first we questioned why the correct version of the validator was released to us instead of the buggy version. Now we realize that it had to do with our correctly understanding the system. We realized that had the buggy version been released first it would have been more difficult to build a mental model of the correct bug-free behavior due to the experienced ‘buggy’ behavior. “Debugging something you don’t understand is pointlessly hard.” Operating under those conditions would have made a correction unnecessarily difficult to overcome. In the real world we have the advantage of hearing the vision of stakeholders and/or the feedback of the product owners which we don’t have here. Spending time with the source code, learning the ins and outs proved to be the most important part of the debugging process.

2. Make it fail

In order to fix something you have to know what is wrong with it or in other words, you can’t debug a problem that you can’t produce. At first this was mostly trial and error however once we discovered a failure, refining the input that caused the failure became the purpose so that we could reliably make the system fail, showing that there was indeed a bug or an issue of some sort present.

3. Quit thinking and look

This proved to not be so much of a problem as one might think. We have the advantage of being somewhat less familiar with the code base (even after Rule #1) when compared to a developer who has been working on the same piece(s) of code for a year or more. In the latter situation it is easy to assume that you know everything there is to know about what is happening under the hood, primarily because you wrote all or most of it. When one is less comfortable and familiar with the code he/she is less likely to forgo turning over particular ‘rocks’ mainly because he/she is not as certain that it couldn’t be that particular ‘rock’. In many cases examining the code was really the only thing we could do based on our limited and relatively unfamiliar knowledge of the code. But still great experience nonetheless.

4. Divide and Conquer

This truly is at the heart of debugging, it was fun to communicate as a group and discuss where we thought a particular problem might reside; in a line of code, a function, a particular file. Then, in private we would rule out particular areas of code based on our tests. Even though the system we were working on may not be large enough to speak in terms larger segmentation like that of namespaces, projects, packages, jobs, stored procedures, and the like it was still a great experience to have the same types of conversations and experience.

5. Change one thing at a time

How this particular rule is approached can make debugging a productive and proficient process or it can make it a frustrating and time-wasting one. In light of looking at the code more and not over-thinking it as we learned from rule #3 it is always a good practice to look and not just think. We were lucky in that we did not fall into the trap of changing too many variables simultaneously in hopes of ‘nailing’ the failure. Thorough, methodical and one-at-a-time changes proved best here. By changing one variable or condition at time we were able to systematically narrow down the possibilities of where the problematic code lived.

6. Keep an audit trail

This was a relatively easy process as we followed rule #5 above. When we made small changes it was easy to document the deltas in a clean and orderly way, and then to pass our repository around when needed.

7. Check the plug

Luckily we did not have any problems here, but this can be so frustrating and something that does not readily come to mind when a problem arises. Tools aren’t perfect and neither are those who create them. At times tested code can be changed to meet changing requirements. When the tests that test this code are not updated failures are introduced. This can be frustrating when a particular test suite contains many of out of date, failing tests that blur the true result of the code’s functionally. When all else comes short of solving a problem this is always a good route to go.

8. Get a fresh view

Our weekly meetings were very beneficial in this regard. As we got together and discussed our tasks new ideas were generated and current ones were refined. Although none of us are true ‘experts’ in this regard our combined intellect, effort and brainstorming led us closer to discovering the solution.

9. If you didn’t fix it, it ain’t fixed

Implementing a fix that corrects the problem was a satisfying experience. To ensure that we had actually fixed the problem and not just understood a symptom we would implement the fix, run the previously failing tests to see them pass, toggle the fix, and run the tests again to see them fail. In this manner we were able to see direct, positive results that validated that we had indeed found the bug.

Bug Reports

```
Title: isValidQuery error

Class: Semi-Serious Bug

Date: 11/24/2015
Reported By: Dalena Pham
Email: phamdal@oregonstate.edu

Is it reproducible: YES

Description
===========
When given a URL that contains a query segment, such as
“http://www.google.com:80/test1?action=view” where the query segment is
the string succeeding the ? character, isValidQuery in UrlValidator.java
evaluates valid query strings as invalid, thereby incorrectly returning a
false value when a true value is expected.

Expected Results (when running manualHelper(true, urlVal,
"http://www.google.com:80/test1?action=view"); )
----------------
Expected: true Returned: true

Actual Results
--------------
Expected: true Returned: false
Failed Test: http://www.google.com:80/test1?action=view

Suggested Fixes
-----------------
Change line 447 in UrlValidator.java from:
        return !QUERY_PATTERN.matcher(query).matches();
to:
        return QUERY_PATTERN.matcher(query).matches();
```

Title: iIpSegment Bug Report

Class: Semi-Serious

Date: 11/25/2015
Reported By: Anthony Wilcox
Email: wilcoant@oregonstate.edu

Is it reproducible: YES

Description

When checking the validity of an IP address in isValidAuthority of UrlValidator.java, the method isValidInet4Address in InetAddressValidator.java returns the incorrect boolean when evaluating if the subgroups of the passed inet4Address is a valid entry. This situation comes up when we are assessing if the authority portion of a Url is valid or not. In particular, the method returns true if a subgroup holds a value larger than 255. IPv4 addresses should not end in 255 and values of 255 and above are not allowed.

Expected Results

```
assertFalse("Should be invalid: " + buffer,
urlVal.isValid(buffer.toString()));
where:
     buffer = http://256.256.256.256:80/test1?action=view
returns a FALSE value and thus, passes assertion.
```

Actual Results

```
assertion fails:
     junit.framework.AssertionFailedError: Should be invalid:
     http://256.256.256.256:80/test1?action=view
test instead returns a TRUE value where a FALSE value was expected.
```

Suggested Fixes

Change line 96 in InetAddressValidator.java from:

```
     return true;
```

to:

```
     return false;
```

```
Title: Port_Regex Bug Report

Class: Semi-Serious

Date: 11/25/2015
Reported By: Benjamin Tullis
Email: tullisb@oregonstate.edu

Is it reproducible: YES

Description
===========
When supplied a valid port number, such as “:65535” isValidAuthority()
method within UrlValidator.java incorrectly reports this as an invalid
port number and causes the entire URL to be marked invalid. The bug only
allowed at a maximum of 3 digits to make up a port number. The largest
port number allowed is actually 65535, which is 5 digits. This leads to
the bug we see when ports with more than 3 digits are incorrectly
reported.

Expected Results
----------------
assertTrue("Should be invalid: " + buffer,
urlVal.isValid(buffer.toString()));
where:
     buffer = “ftp://go.com:65535/test1?action=edit&mode=up”
returns a TRUE value and thus, passes assertion.

Actual Results
--------------
assertion fails:
     junit.framework.AssertionFailedError: Should be valid:
     ftp://go.com:65535/test1?action=edit&mode=up
test instead returns a FALSE value where a TRUE value was expected.

Suggested Fixes
-----------------
Change line 158 from UrlValidator.java from:
     private static final String PORT_REGEX = "^:(\\d{1,3})$";
to:
     private static final String PORT_REGEX = "^:(\\d{1,5})$”;
```