Evaluating Quality of Security Testing of the JDK *

Padmanabhan Krishnan
Oracle Labs, Brisbane
paddy.krishnan@oracle.com

Jerome Loh
Oracle Labs, Brisbane[1]
zhengxiang.loh@uq.net.au

Rebecca O'Donoghue
Oracle Labs, Brisbane
rebecca.odonoghue@oracle.com

Larissa Meinicke
ITEE, The University of Queensland
l.meinicke@uq.edu.au

1. Introduction

The Java security model provides language-level access control to security-sensitive resources and actions. Proper use of this model is the responsibility of the programmer, and errors may arise in its use. As there is no formal model of the desired security properties, techniques such as verification are not directly applicable. So testing is the only practical way to detect such errors. However, one wishes to have some guarantee that the tests themselves are thorough enough to catch all errors in the use of the security model. So the first step is to determine the quality of test suites that check for security properties.

Mutation testing [7] can be used to measure the effectiveness of a test suite. The general idea is to seed faults into the original program to derive mutations and check whether the test suite can distinguish the behaviour of each of the mutations against the original program. The percentage of mutations that a test suite can distinguish can be used as a measure of effectiveness of the test suite. As a result, a tester can determine the causes for not distinguishing a mutation to improve the test suite.

As the number of potential faults in a program can be very large, generating mutations that represent all the faults is not useful in practice. For instance the mutation operators developed for object-oriented languages [1, 4] are too general. In order to reduce the number of generated mutations, one usually assumes a competent programmer hypothesis where the possible mutations are derived from a set of typical errors a competent programmer can make. For instance, not all variable replacement operators may be relevant in a given situation. Otherwise, mutations will, typically, not represent real faults [6].

In this position paper, we outline the issues related to mutation testing in the evaluation of test suites that are relevant for security in the JDK. That is, we do not focus on user's applications but on the library. This is because any security issue in the JDK has an impact on a large number of applications.

The key questions relate to identifying representative mutation operators and ensuring that they can be used in an automated process that scales to industrial size code.

2. Mutation Operators

In a security context, mutation testing has been applied to detect string formatting and buffer overflow vulnerabilities in C programs [10, 11]. But these mutation operators are not related to any security model. Martin and Xie [9] and Traon et al. [12] have extended mutation testing to testing access control policies. Both studies utilise a system based on rules that either permit or deny sensitive actions based on the caller's identity, or role. Bertolino et al. [3] describe mutation operators that are relevant for XACML. Conceptually, these mutation operators are similar to what is desired for Java. However, we need operators that directly manipulate the constructs that deal with security. These include the checkPermission and doPrivileged invocations [5]—the details of which are not discussed here.

Semantically, the basic mutation operators we consider can be classified as either narrowing or widening the set of permissions associated with the original code fragment. The narrowing operators either check for more permissions or remove permissions that are available to the code. One can check for more permissions by replacing the permission in a checkPermission call (say p) with permission q such that

* Java and JDK are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

[1] Jerome Loh was enrolled at the University of Queensland

ISBN [to be supplied]. . . $15.00

q implies p. The widening operators either check for fewer permissions or in effect grant more permissions to the code. Replacing a permission p in a checkPermission call with a permission q such that p implies q results in widening. Note that it is possible to have mutation operators that widen a narrowing (or narrow a widening). Similarly, the mutations of a doPrivileged will involve changing the access control contexts and permissions passed to it. Note that we do not consider changing the privileged action as that represents a behavioural change which is not security specific.

3. Open Issues

A number of challenges need to be addressed before a proper evaluation can be performed. They include scalability and the choice of mutation operators, and tool support.

The first is related to scalability and automation. The JDK has more than 2 million lines of Java code. So the number of mutations generated must be limited but if they do not cover the relevant security properties the mutation operators will not be useful. In principle, there are an unbounded number of options when replacing permissions or access control contexts. This is because, parameters to permissions are strings. So exploring all potential replacements for a given mutation operator is not practical.

A more precise formulation of the competent programmer hypothesis will help in identifying the set of relevant permissions or ACCs. One could find the minimal set of distinct mutations [2] and thus eliminate mutations which are themselves equivalent or subsumed by other mutations. The problem of reducing the set of mutations is NP-complete and thus approximations of the minimal set will be required.

The second challenge is to integrate `jtreg`, the testing infrastructure used for the JDK, with our mutation process. For this we need to identify a suitable mutation testing tool. Although there are tools such as *muJava*[2], [8] Jumble[3] and PIT[4], none of these can be used directly. Based on some initial experimentation, we are exploring adapting PIT to suit our requirements. A related challenge arises because PIT and `jtreg` are written in Java. As we are mutating the JDK, we have to ensure that the behaviour of PIT and `jtreg` are not altered by the mutation process. We would need an isolation mechanism to effect this.

4. Summary

In this note we have outlined the need for security-specific mutation operators and their categorisation into impact classes. While we have described in the context of the JDK, this would be required for any programming language that supports security. We also have described the engineering challenges related to scalability, existing infrastructures and tool support. To summarise, the general questions are

- What are language specific security mutation operators that are applicable to libraries that provide security services?
- How to ensure these mutation operators can be used for large programs?
- How to overcome some of the engineering challenges so that these mutation operators are used in practice?

A more long-term goal is to use the results of the evaluation to generate the required tests to improve security coverage.

[2] https://cs.gmu.edu/ offutt/mujava/

[3] http://jumble.sourceforge.net/

[4] http://pitest.org/

References

[1] R.T. Alexander, J.M. Bieman, S. Ghosh, and B. Ji. Mutation of Java objects. In *13th International Symposium on Software Reliability Engineering (ISSRE)*, pages 341–351, 2002.

[2] P. Ammann, M. E. Delamaro, and J. Offutt. Establishing theoretical minimal sets of mutants. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 21–30, 2014.

[3] A. Bertolino, S. Daoudagh, F. Lonetti, and E. Marchetti. XACMUT: XACML 2.0 mutants generator. In *Proc. of 8th International Workshop on Mutation Analysis*, pages 28–33. IEEE, 2013.

[4] P. Chevalley and P. Thévenod-Fosse. A mutation analysis tool for Java programs. *International Journal on Software Tools for Technology Transfer*, 5(1):90–103, 2003.

[5] L. Gong, G. Ellison, and M. Dageforde. *Inside Java 2 Platform Security*. The Java Series. Addison Wesley, 2003.

[6] R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *Proceedings of the IEEE 25th International Symposium on Software Reliability Engineering, 2014 (ISSRE'14)*, pages 189–200, 2014.

[7] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[8] Y-S. Ma, J. Offutt, and Y-R. Kwon. MuJava : An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[9] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *Proceedings of the 16th International Conference on World Wide Web*, pages 667–676, 2007.

[10] H. Shahriar and M. Zulkernine. Mutation-based testing of buffer overflow vulnerabilities. In *Computer Software and Applications (COMPSAC)*, pages 979–984, 2008.

[11] H. Shahriar and M. Zulkernine. Mutation-based testing of format string bugs. In *High Assurance Systems Engineering Symposium (HASE)*, pages 229–238, 2008.

[12] Y. L. Traon, T. Mouelhi, and B. Baudry. Testing security policies: Going beyond functional testing. In *18th IEEE International Symposium on Software Reliability (ISSRE)*, pages 93–102, 2007.