UTM Computing Proceedings
Innovation in Computing Technology and Applications
Volume: 2| Year: 2017 |ISBN: 978-967-0194-95-0

A Study on Automated Mutation Testing for Java Applications: Security Aspect

Tan Wei Long [1] and Radziah Mohamad [2]

Faculty of Computing, Universiti Teknologi Malaysia (UTM), Malaysia

[1] weilong1113@gmail.com, [2] radziahm@utm.my

Abstract. On the cutting edge, many of our daily activities as well as enterprise activities are performed through the use of various applications and systems. Most of them operates remotely via access to internet. Following their rapid growth, security has been imposed as a crucial issue. Security testing on exploiting vulnerabilities of applications and systems is a must against malicious attack. Traditional security testing approaches work on line coverage of an application. Hence, mutation testing is extended to ensure quality of an application and minimize the faults of it. This paper discusses an overview of mutation testing approaches and its components. On the other hand, security aspects in relative to different mutation operators are discussed. A few of the mutation operators which are Return Values Mutator, Void Method Calls Mutator and Negate Conditionals Mutator are related to security cases according to their behaviors and functions. The results of this study will help researchers to identify the suitable mutation testing approaches as well as their mutation operators used against security aspect, and to develop appropriate tools in mutation testing against security aspect.

Keywords: Mutation testing, security testing, Java applications, PIT, mutation operators.

1 Introduction

In recent years, Information and Communication Technology (ICT) has emerged as a blooming industry as following the trend of globalization. Massive amount of deliveries of information are flowing from time to time every single second. Communication happens at all time, data transversed and delivered is vary in types and extent of their confidentiality. Throughout the years, this has urged a need in security against malicious attack to the private data we have. Security of an application or a system cannot be guaranteed, especially those that can be reached via access to internet. Leaking of information might happens when intended people with misaligned intention launch attack on the applications which hold confidential data and steal them. From studies done by other researchers, most of the applications are not flawless [1]. There are still some immaturities in the approaches used to test the security aspect of the applications. Enhanced approaches which can detect and reject faults are required to have a more concrete and complete testing on them. In this context, traditional security testing approaches are insufficient in ensuring a quality application with minimized faults in it. They cover up to the extent of line coverage. Hence, mutation testing is induced to be discussed here.

Mutation testing is a type of white box testing used to make syntactic changes in a program and examine whether the mutated program can fail against the test suite for it. A mutation is a change in a program while the mutated program in this context is called mutant. In this study, several mutation testing approaches on security aspect towards Java application were examined. They were used to perform mutation tests on several subjects under test and the results were recorded and discussed. This study aims to investigate the existing mutation testing approaches in Java applications. In addition, mutation testing approaches are presented with their related cases in the context of security. As of the scope of this study, connection in between the approaches and the security cases are identified. Experiments were done to the extent of unit level. Mutation engines from tools and plugins acted as the main component in the tests.

The layout of this study is as follows: Section 2 discusses the state-of-art in mutation testing approaches and security aspects in Java applications. Section 3 discusses the mutation testing approaches used in the cases. Section 4 presents the results of the tests performed while Section 5 discusses the outcomes of the results obtained with respect to their security aspects. Lastly, the study is concluded in Section 6.

2 Related Works

2.1 Security in Java applications

Java has emerged as one of the most common programming languages in which 3 billion devices worldwide run on it [2]. Java is used by many companies in one way or other. For mobile applications, Java is one of the main technology of choice to build them. With its slogan that quotes 'Write once, run anywhere', cross-platform benefits of Java are illustrated. Java is a platform independent programming language that runs on any operating systems.

Extensive use of Java in Enterprise applications makes it one of the targets for malicious attack. It is described as an avenue of exploitation. There are more than 50 percent of attacks towards Java applications [3]. Security patches and updates for fixing the vulnerabilities are not frequent and not effective in all situations. This is due to cost that an enterprise has to bear for the patches across all its device, as well as compatibility issues that come with the applications and their environments. Throughout the years, Java has been a large and growing security risk on its own.

2.2 Mutation Testing

Currently, testing approaches for Java applications use simple coverage which tests against line coverage. Evaluation for the quality of the applications are not sufficiently done through traditional testing approaches. Hence, mutation testing is introduced to tackle the issues out of the scope of traditional testing approaches. It is one of the approaches that can detect test safety gaps in a way that goes beyond code coverage. The idea of mutation testing is to make syntactic changes to original codes of a program or application [4]. The program that undergoes changes is mutated and it is called mutant. Test sets against the program is then executed on the mutant to determine whether the mutant behave differently from the original codes, that is producing different outcomes from the original codes. If the test sets detect and reject the mutant, the mutant is said to be killed successfully, which in another word the test sets match the complexity of the codes.

Mutation testing also acts as an indicator to how strong is the given test sets and a guide for software testers to design more effective tests. Since it is a fault-based testing approaches, it detects and helps a program to be free from specific faults [5]. The most essential component in mutation testing is the mutation operators or mutators. Each of the mutation operator carries different types of mutations such as replacement of arithmetic operators, changing the return

Table 1. Original codes and Mutated codes

Original program	Mutant program

void min(int x,int y){
int minval=x;
if(y<x) minval=y;
return minval;
}
	void min(int x,int y){
	int minval=x;
	if(y>x) minval=y;
	return minval;
	}

value of a data type and so on. By the use of the mutation operators, faults are injected into program codes and yields a set of mutants. From Table 1, condition of the *if statement* is mutated in which '<' is replaced by '>', this yields a mutant from the original program.

3 Methodology

3.1 Tools

Many open source tools are available for performing mutation testing on various applications on different platform and programming languages. Nonetheless, many of them are idle and do not receive continuous support and development. One of the tools which is actively in development today is PIT. It is an open source mutation testing tool for Java. Similar to other mutation testing tools, PIT runs unit tests against automatically modified versions of application code. If a unit test does not fail in this situation, it may indicate an issue with the test suite. As of the 0.29 release PIT is also the first generally available incremental mutation testing system, with the option to amortize the cost of analysis by storing a history of results.

Experiments carried out in this study were involving the use of PIT mutation engine as in the form of plugin in Eclipse IDE and also integration with Maven in native command line. Currently there are some built in mutation operators which come with PIT in the package. Mutations are performed on the byte code generated by the compiler which in general a much faster and easier approach if compared to performing mutation on source code. The mutation operators are stable that they will not be detected too easily.

With the use of PIT mutation engine, mutations are being induced into Java Virtual Machine (JVM) byte code [6]. If a test carried out fails against the test cases, it is said to be KILLED, which in another word, says, mutation is KILLED. Otherwise, if the test succeeds, the mutations is said to SURVIVED. In addition, there are other outcomes which include TIMED OUT if a mutant is caught in an exception of infinite loop, and NO COVERAGE in which there are no tests that exercised the line of code where the mutation was created.

A test suite is said to be faulty if a unit test does not fail on a mutant. Traditional test coverage measures the line of code which is executed by the tests, it however, does not check whether the tests are able to detect faults in the codes. By the means of mutation test, quality of tests can be enhanced. Test suites are measured by the percentage of mutants that they kill.

3.2 Subjects Under Test

Java applications as the subject under test in this study includes open source applications which are Kata Bank application as well as Rest Stub Server. Mutation testing was performed on the two subjects with PITClipse which is PIT integration in Eclipse IDE and PIT with Maven framework respectively.

Mutation Operators from PIT	Related Mutation Operators
RETURN_VALS_ MUTATOR	Intent Payload Replacement (IPR) [8]
	Intent Target Replacement (ITR) [4]
VOID_METHOD_CALL_ MUTATOR	RIHA (replace innerHTML property with text node) [7]
	ADES (add escape function calls) [7]
NEGATE_CONDITIONALS_MUTATOR	PPR ((permission to prohibition) replaces a rule of permission by a prohibition) [9]
	PRP ((prohibition to permission) replaces a rule prohibiting by

Table 2. Mutation 4		permission) [9]	Related Operators
		Negation of expression in where conditions (NEGC) [9]	
		Set multiple query execution flags true (MQFT) [9]	

Results

Mutation testing was performed on both subjects. Results of mutations were tabulated and analyzed further in details. Furthermore, mutation operators which were involved in the cases were filtered and listed. They were then being related to the security aspect of different stories. In current technologies market, there were many testing techniques and methods for exploiting vulnerabilities in an application. However, security remains a main issue as there are possibilities of malicious attack despite of it passes all functional tests. Several mutation operators have been proposed by different researchers to cope with vulnerabilities in security aspects. Here the related mutation operators from PIT were matched with other mutation operators proposed by other researchers as shown in Table 2.

5 Discussion

As shown in Table 2, three mutation operators from PIT were related to other proposed mutation operators which can be linked to certain security aspects. In this section, the connection among the mutation operators as well as their security aspects are discussed.

5.1 Return Values Mutator

Return Values Mutator (RETURN_VALS_MUTATOR) is designed to mutate the return values of method calls. For instance, it replaces unmutated return value ‘true’ with ‘false’ and vice versa. There are two mutation operators that were matched with Return Values Mutator in which they have similarities in mutating codes. Both of the mutation operators involve the change of data types as well as parameters. An Intent is an abstraction of an operation among Android components that are usually used to launch an activity and transmit data or messages between activities. It carries different data types as key-value pairs. Intent Payload Replacement (IPR) mutates parameter to default value that depends on its underlying data type. Objects of types such as int, short, long, etc., are replaced by the value of zero, while String objects are replaced by empty strings, and boolean variables are replaced by both true and false. Invalid intent messages trigger a privileged protected API call. This could possibly expose a potential security vulnerability, because invalid intent is not rejected and forces the application to perform a privileged action [7].

Next, Intent Target Replacement (ITR) looks up all the classes within the same package of the current class, and then replaces the target of each Intent with all possible classes. This ensures the design of test cases that verify the success of target activity or service launched after executed the Intent.

5.2 Void Method Call Mutator

As the name suggests, Void Method Call mutator (VOID_METHOD_CALL_MUTATOR) removes method calls to void methods. Here the mutation type was related to RIHA (replace innerHTML property with text node) and ADES (add escape function calls). For RIHA, it modifies text of HTML tags using the innerHTML property. This modification induces vulnerabilities to XSS attacks when injected text contains HTML tags. In one of the mutations, the innerHTML property was replaced by ’document.createTextNode(str)’. The outputs against the test were not equal and hence the mutation was KILLED [8].

On the other hand, ADES modifies arguments of write method of document (DOM object) which generates HTML contents with escape function call. A variable containing malicious script code could be intepreted and

executed while writing its content into the HTML. Injected HTML with escape function call is not intepreted by browsers.

5.3 Negate Conditionals Mutator

Negate Conditionals Mutator (NEGATE_CONDITIONALS_MUTATOR) mutates conditionals such as '==' to '!=' and vice versa. Mutations on access control using PRP and PPR simulates an incremental evolution of access control policy. Concerning security tests generated from the access control policy, they aim to validate the compliance of the security mechanism with its access control policy.

Next, the NEGC operator works by negating unit expressions (e.g., uid='aaa' to uid!='aaa') present in where conditions of SQL queries which includes SELECT, UPDATE, and DELETE type queries.

Lastly MQFT injects SQLIVs by setting the multi query flag value to true. Databases are connected through a URL string specifying the database name and location, which allows whether to allow executing multiple queries in a single connection. This makes it possible to perform piggybacked query attacks that append arbitrary SQL queries. The operator either adds the flag value by appending it to the connection URL or modifying the existing flag value to true.

6 Conclusion

In this paper, state-of-art of mutation testing approaches is presented with the use of open source tools available. Feasible tools which are PITClipse and PIT with Maven were being used to perform mutation testing on subjects selected. After that, discussion were carried out in detail on the mutation operators used, as well as security aspects that the mutation operators could be linked to. They are 3 mutation operators from PIT which are Return Values Mutator, Void Method Calls Mutator and Negate Conditionals Mutator. Each aspect is presented with respect to related mutation types. Several issues were encountered during the experiment phases including configuration issue of environments with tools used, compatibility of environment with versions of tools, lacking of proper documentation and so on. In the future, specific mutation operators could be used more precisely to detect faults in applications to ensure the quality of the application. Mutation testing covers up to the extent of faults in codes and it will be an essential testing process to ensure the least faults found in an application.

Acknowledgements. The authors benefitted from many discussions with various researchers and online communities in the field of Mutation Testing whom had been kind and provided helpful comments on this study. A sincere appreciation to them for their time and expertise as well as the efforts.

References

1. Amalfitano, D., Fasolino, A. R., Tramontana, P. and Robbins, B. Testing Android Mobile Applications: Challenges, Strategies, and Approaches. Advances in Computers (2013)
2. Smith, M. Making the Future Secure with Java. (2012)
3. Invincea. Protecting Java Applications. Technical report. Invincea (2014)
4. Deng, L., Mirzaei, N., Ammann, P. and Offutt, J. Towards Mutation Analysis of Android apps. Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on. IEEE (2015)
5. Shahriar, H. Mutation-based Testing of Buffer Overflows, SQL Injections, and Format String Bugs. (2008)
6. Schaefer, I. and Stamelos, I. Software Reuse for Dynamic Systems in the Cloud and Beyond: 14th International Conference on Software Reuse, ICSR 2015, Miami, FL, USA, January 4-6, 2015. Proceedings. vol. 8919. Springer (2014)
7. Avancini, A. and Ceccato, M. Security Testing of the Communication among Android Applications. Automation of Software Test (AST), 2013 8th International Workshop on. IEEE (2013)
8. Shahriar, H. and Zulkernine, M. Mutec: Mutation-based Testing of Cross Site Scripting. Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems. IEEE Computer Society. (2009)
9. Ennahbaoui, M. and Elhajji, S. Mutation Analysis for Security: 1309.6149. (2013)