www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 5 Issue 8 August 2016 Page No. 17643-17647

Analysis of Web base applications testing using mutant

Solanke Vikas* ***Prof. Shyam Gupta***
Department of Computer Engineering
solankevs@mmpolytechnic.com
Department of Computer Engineering
gohadshyam@rediffmail.com

***Abstract*— Mutation testing was initially proposed in the 1970s as intends to guarantee vigour in test case improvement. By making syntactically right substitutions inside the software under test (SUT) and rehashing the test execution stage against the adjusted code, an evaluation could be made of test quality contingent upon if the definitive test cases could locate the code adjustment. Mutation testing is normally used in small code programs, but for a small portion of large program or for a specific code it is used. This paper test a access control part of web based applications for mutant testing, till date testing of web based application is only up to application level only. Generally test cases are executed and stop the testing, but we can not check capability of test cases.**

Keywords:Mmutant, mutation testing, access control, web application, testing.

INTRODUCTION

Mutation Testing is a shortcoming based testing procedure which furnishes a testing rule called the "mutation ampleness score" or "mutation adequacy score". The mutation ampleness score could be utilized to measure the viability of a test set as far as its capability to catch faults .The general guideline underlying Mutation Testing work is that the shortcomings utilized by Mutation Testing speak to the errors that programmers regularly make.

One mutation operator to the program is called a mutant. If the test suite is able to detect the change (i.e. one of the tests fails), then the mutant is said to be killed.

For example, consider the following C++ code fragment:

```
if (a && b) {
   c = 1;
} else {
   c = 0;
}
```

The condition mutation operator would replace && with || and produce the following mutant:

```
if (a || b)
 {
   c = 1;
} else {
   c = 0;
}
```

Now, for the test to kill this mutant, the following three conditions should be met:

1. A test must reach the mutated statement.
2. Test input data should infect the program state by causing different program states for the mutant and the original program. For example, a test with a = 1and b = 0 would do this.
3. The incorrect program state (the value of 'c') must propagate to the program's output and be checked by the test.

These conditions are collectively called the RIP model.[3]

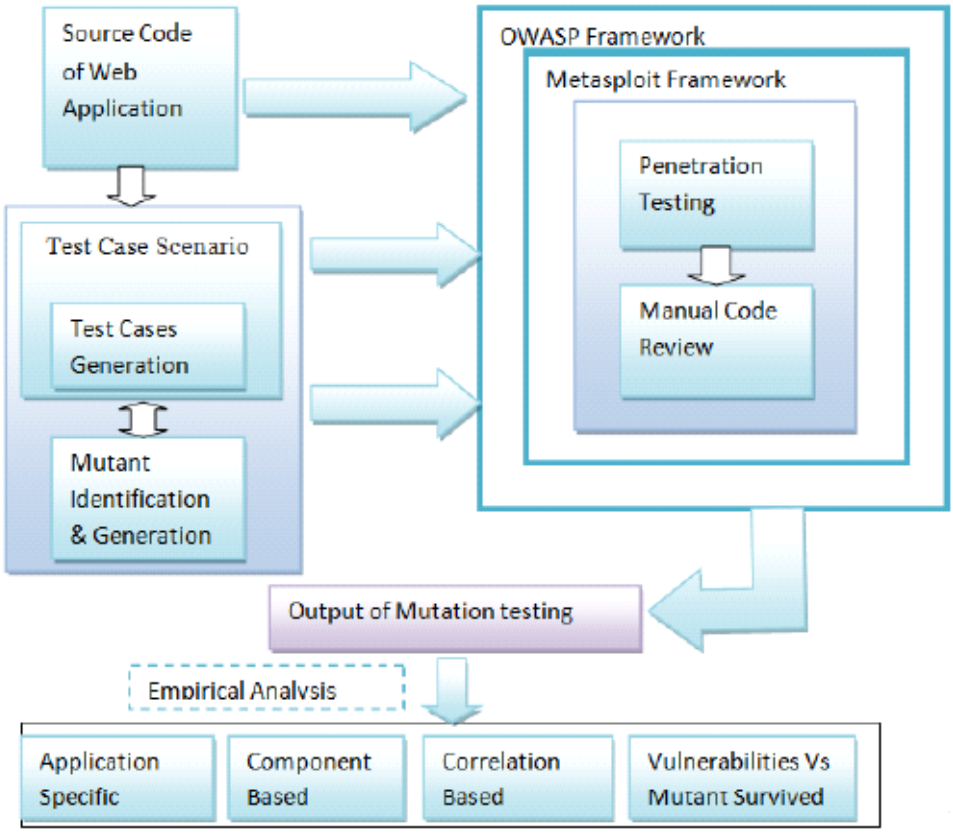
Weak mutation testing (or weak mutation coverage) requires that only the first and second conditions are satisfied. Strong mutation testing requires that all three conditions are satisfied. Strong mutation is more powerful, since it ensures that the test suite can really catch the problems. Weak mutation is closely related to code coverage methods. It requires much less computing power to ensure that the test suite satisfies weak mutation testing than strong mutation testing.

Mutation operators: Many mutation operators have been explored by researchers. Here are some examples of mutation operators for imperative languages:

Statement deletion
Statement duplication or insertion, e.g. goto fail;[15]
Replacement of boolean sub expressions with true and false
Replacement of some arithmetic operations with others, e.g. + with *, - with /
Replacement of some Boolean relations with others, e.g. > with >=, == and <=
Replacement of variables with others from the same scope (variable types must be compatible)

mutation score = number of mutants killed / total number of mutants
These mutation operators are also called traditional mutation operators. There are also mutation operators for object-oriented languages,[16] for concurrent constructions,[17] complex objects like containers,[18] etc. Operators for containers are called class-level mutation operators. For example the tool offers various class-level mutation operators such as Access Modifier Change, Type Cast Operator Insertion, and Type Cast Operator Deletion. Mutation operators have also been developed to perform security vulnerability testing of programs.

I. SYSTEM DESIGNING



Using the source code we write test cases considering various test case scenarios like code sanitation and covering all code components and all security functionalities. Then mutants are identified using the source code, mutants are generated manually again keeping the security functionalities and vulnerabilities in mind. These mutants are checked against the test cases written if mutants are killed then test cases are sufficient and if mutants are not killed then test cases are again written accordingly. We are using manual approach for mutant creation to avoid the problem of equivalent mutants which is present in case of automated mutant generation by tools. This whole process is iterative and mutation testing helps to write strong test cases. Test cases are written in Java. Now using these test cases we do penetrative testing of source code. In this part we are checking how well software's defenses against all types of vulnerabilities are.

STRUCTURAL MODELLING:
Structural modelling captures the static features of a system. They consist of the followings:

- Classes diagrams
- Objects diagrams
- Deployment diagrams
- Package diagrams
- Composite structure diagram
- Component diagram

Structural model represents the framework for the system and this framework is the place where all other components exist. So the class diagram, component diagram and deployment diagrams are the part of structural modelling. They all represent the elements and the mechanism to assemble them.
But the structural model never describes the dynamic behavior of the system. Class diagram is the most widely used structural diagram.

Behavioural Modelling:

Behavioural model describes the interaction in the system. It represents the interaction among the structural diagrams. Behavioural modelling shows the dynamic nature of the system. They consist of the following:

- Activity diagrams
- Interaction diagrams
- Use case diagrams

All the above show the dynamic sequence of flow in a system.

Architectural Modelling:

Architectural model represents the overall framework of the system. It contains both structural and behavioral elements of the system. Architectural model can be defined as the blue print of the entire system. Package diagram comes under architectural modeling.
UML is popular for its diagrammatic notations. We all know that UML is for visualizing, specifying, constructing and documenting the components of software and non software systems. Here the Visualization is the most important part which needs to be understood and remembered by heart.
UML notations are the most important elements in modeling. Efficient and appropriate use of notations is very important for making a complete and meaningful model. The model is useless unless its purpose is depicted properly.
So learning notations should be emphasized from the very beginning. Different notations are available for things and relationships. And the UML diagrams are made using the notations of things and relationships. Extensibility is another important feature which makes UML more powerful and flexible.
The chapter describes the UML Basic Notations in more details. This is just an extension to the UML buildling block section I have discussed in previous chapter.

Structural Things:

Graphical notations utilized in structural things are the most broadly used in UML. Those are considered because the nouns of UML models. Following are the listing of structural things.

- Classes
- object
- Interface
- Use case
- Component
- Collaboration
- Active Classes
- Nodes

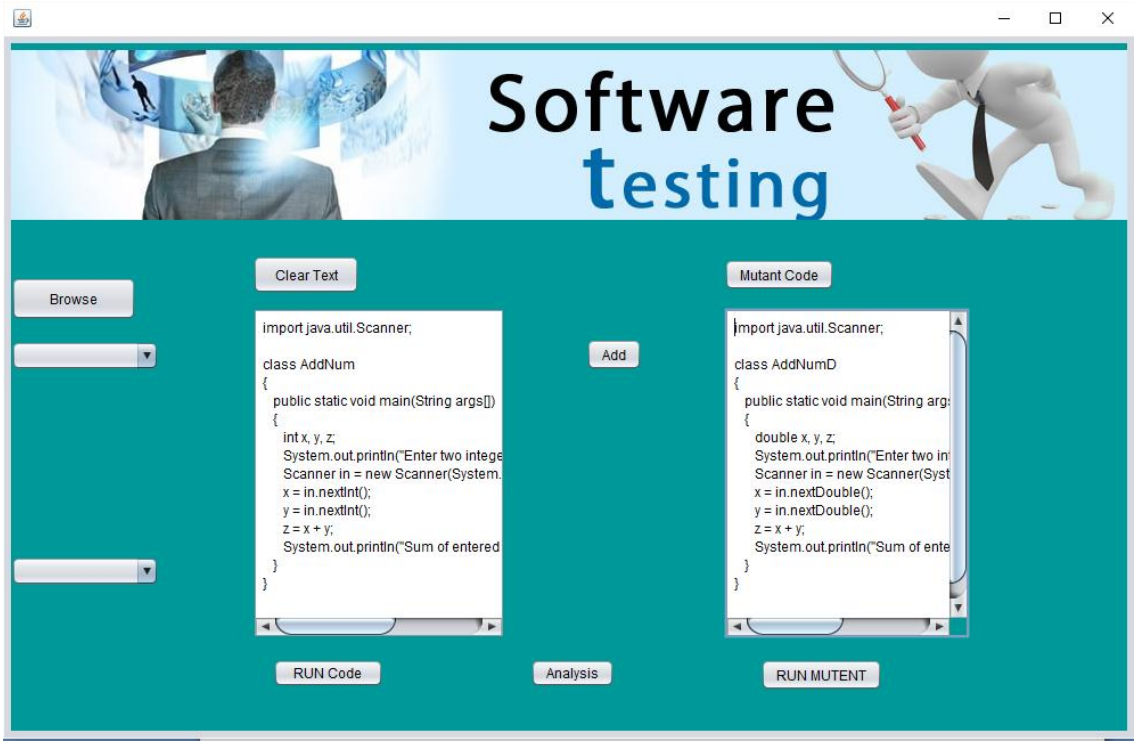
Operand replacement operators	**Expression Modification Operators**	**Statement modification Operators**
Replace the operand with another operand(x with y or y with x) or with the constant value.	Replace an operator or insertion of new operators in a program statement.	Programmatic statements are modified to create mutant programs.

Example-
 	
If(x>y) replace x and y values
 	
If(5>y) replace x by constant 5
	Example-
	
	If(x==y)
	
	We can replace == into >= and have mutant program as
	
	If(x>=y) and inserting ++ in the statement
	
	If(x==++y)
		Example-
		
		Delete the else part in an if-else statement
		
		Delete the entire if-else statement to check how program behaves
		
		Some of sample mutation operators:
		
		• GOTO label replacement
		• Return statement replacement
		• Statement deletion
		• Unary operator insertion(Like - and ++)
		• Logical connector replacement
		• Comparable array name replacement

DOI: 10.18535/ijecs/v5i8.38

		• Removing of else part in the if-else statement • Adding or replacement of operators • Statement replacement by changing the data • Data Modification for the variables • Modification of data types in the program

II. ACTUAL EXECUTION OF MUTANT



Mutant Operators : In java web based application we create our own operators where we consider following

- Information Hiding/Access control
- Inheritance
- Polymorphism
- Overloading
- Java specific Features
- Operators depend on common program mastics

Faults Class Mutation Operators State visibility anomaly	IOP IHD, IHI
State definition inconsistency (due to state variable hiding)	
State definition anomaly (due to overriding)	IOD
Indirect inconsistent state definition Anomalous construction behavior	IOD IOR, IPC, PNC
Incomplete construction Inconsistent type use	JID, JDC
Overloading methods misuse,	PID, PNC, PPD, PRV
Access modifier misuse	OAN OMD, OAO

static modifier misuse	AMC
Incorrect overloading methods implementation	JSC
super keyword misuse	OMR
this keyword misuse	
Faults from common programming mistakes	ISK
	JTD
	EOA, EOC, EAM, EMM

Operators used in Java Programs

III. CONCLUSION

In our approach we tried to empirically evaluate the process of mutation testing giving developer an idea for the future. One of the key security polishes that needs to be set up with specific end goal to relieve the expanding number of vulnerabilities in Web applications, is an organized security testing technique. The way of Web applications requires an iteration furthermore evolutionary methodology to advancement. Hence, the structured security testing approach requirements to have the capacity of being adjusts to such nature's domain, and it should be particular for Web applications. The most connected security testing approaches today are broad and are frequently excessively confused with their numerous exercises and stages. By applying such far reaching security testing strategies in the domain of Web applications, engineers have a tendency to disregard the testing procedure because the systems are recognized to be; excessively time intensive, failing to offer a critical result and unseemly to be connected on Web applications in light of the fact that they have a quite short opportunity to-market. This could be viewed as one of the variables to why security testing frequently is executed consistent with the infiltrate and-patch ideal model. In this postulation, the creator has demonstrated that by utilizing an organized security testing procedure particularly created for Web applications, expedites an altogether more powerful method for performing security tests on Web applications contrasted with existing specially appointed methods for performing security tests. The components that the creator used to measure the proficiency were: the measure of time used on the security testing process, the measure of vulnerabilities found throughout the security testing procedure and the capacity to moderate false-positives throughout the security testing procedure

REFERENCES

[1] J.H.Andrews, L.C. Briand and Y. Labiche,"Is Mutation an Approximate Tool for Testing Experiments?" Proc. IEEE Int'I Conf, Software Engg.pp. 402-411,2005.

[2] [2] H. Do and G.E Rothermel, "On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques," IEEE Trans. Software Eng., vol. 32, no. 9, pp. 733-752, Aug. 2006..

[3] [3] H. Agrawal, R.A. DeMilo, B.Hathaway,W. Hsu,E.W.Krauser, R.J. Martin, A.P. Mathur and E. Spafford, "Design of mutant Operator for C programming language", Technical report SERC-TR-41P,Purdue, West Lafayette,Ind,Mar.1989

[4] [4] J.J. Chilenski and S.P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing," Software Eng. J.,vol. 9, no. 5, pp. 193-200, 1994.C. J. Kaufman, Rocky Mountain Research Lab., Boulder, CO, private communication, May 1995.

[5] [5] D. Daniels, R. Myers, and A. Hilton, "White Box Software Development," Proc. 11th Safety-Critical Systems Symp., Feb. 2003.

[6] [6] R. Butler and G. Finelli, "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software," IEEE Trans. Software Eng., vol. 19, no. 1, pp. 3-12, Jan. 1993..

[7] [7] H. Do and G.E Rothermel, "On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques," IEEE Trans. Software Eng., vol. 32, no. 9, pp. 733-752, Aug. 2006..

[8] [8] M. Daran and P. The´venod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations," ACM SIGSOFT Software Eng. Notes, vol. 21, no. 3, pp. 158-177, May 1996.

[9] [9] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practical Programmer," Computer, vol. 11, no. 4, pp. 34-41, Apr. 1978..

[10] [10] J.J. Chilenski, "An Investigation of Three Forms of the Decision Coverage (MCDC) Criterion," Report DOT/FAA/AR-01/18, Office of Aviation Research, Washington, D.C.,Apr. 2001.G. R. Faulhaber, "Design of service systems with priority reservation," in Conf. Rec. 1995 IEEE Int. Conf. Communications, pp. 3–8.