2009 33rd Annual IEEE International Computer Software and Applications Conference

GUI-Based Testing of Boundary Overflow Vulnerability

T. Tuglular, C. A. Muftuoglu, O. Kaya
Department of Computer Engineering
Izmir Institute of Technology
Urla, Izmir, Turkey 35430
tugkantuglular/ardamuftuoglu/ozgurkaya@iyte.edu.tr

F. Belli, M. Linschulte
Department of Computer Science, Electrical Engineering and Mathematics
University of Paderborn
Germany
belli/linschulte@adt.upb.de

Abstract

Boundary overflows are caused by violation of constraints, mostly limiting the range of internal values of a program, and can be provoked by an intruder to gain control of or access to stored data. In order to countermeasure this well-known vulnerability issue, this paper focuses on input validation of graphical user interfaces (GUI). The approach proposed generates test cases for numerical inputs based on GUI specification through decision tables. If boundary overflow error(s) are detected, the source code will be analyzed to localize and correct the encountered error(s) automatically.

1. Introduction

Graphical user interfaces (GUIs) add up to half or more of the source code in software [2]. Unfortunately, GUIs may allow intruders to gain control over a system or access to its stored data by intentionally caused boundary overflows. A boundary overflow is an input error and occurs when values are entered that violate the range of values [9]. Such entries exceed the implicitly or explicitly specified but not implemented boundary values; thus this is a consequence of deficient control mechanism in the software concerning system constraints.

This paper focuses on input validation testing of GUIs. We propose a control mechanism, similar to the Design by Contract (DbC) concept, to check the boundary input values explicitly on the GUI and the source code; we assume that constraints might be missing or wrongly set. Our approach for input validation suggests to specify user interface requirements and to convert this specification into a formal model from which valid and invalid test cases can be generated [3].

The novelty of our approach stems from algorithms we introduce for the first time in this paper to detect and correct boundary overflow vulnerabilities. Apart from checking the error handling mechanism, these algorithms, implemented in Java, extend SUC by adding a simple exception handling mechanism if / where none exists. For validation of the approach an open source port scanner, developed in C++, has been tested in a local area network (LAN).

Next section summarizes related work before Section 3 summarizes the test generation algorithm. The core of the paper, Section 4, develops algorithms for detection and correction of boundary overflow vulnerabilities through analysis of the model and code of the SUC. Sections 5 and 6 include technical details of the approach and case study on a port scanner. Section 7 concludes the paper and outlines our research work planned.

2. Related Work

Our approach combines dynamic input validation with static analysis for evaluating given constraints and makes also use of DbC concept. Input validation checks the syntax and, partly, semantics of information provided by user via user interface (UI), mostly realized as a graphical UI (GUI) [6]. Because UI errors may lead to malfunctions of the entire system which, in turn, may lead to vulnerabilities for attacks [11], various specification- and implementation-based test techniques exist to validate UI [7]. Among them,

0730-3157/09 $25.00 © 2009 IEEE
DOI 10.1109/COMPSAC.2009.189

code coverage rate helps to detect possible input errors and, by using correction algorithms, the illegal inputs can be enforced to set them in defined boundaries. Event sequence graphs [3] can be used for analysis and validation of UI requirements prior to implementation and testing of the code [4]. An ESG is a simple albeit powerful formalism for capturing the behavior of a variety of interactive systems that include real-time, embedded systems, and graphical user interfaces [3].

Design by Contract (DbC) is an object-oriented design technique that was first introduced by Meyer in 1992 [10]. DbC focuses on the extension of source code, e.g., an operation, by pre- and post-conditions that can be evaluated during runtime (similar to a legal contract). Thus, implementing DbC could be used to avoid the boundary overflow vulnerability. On the basis of DbC, there exist some approaches that adopt the DbC-idea for testing. Zheng et al. [17] introduced an UML-based software component testing technique called Test by Contract.

Input: Decision table
Output: Test cases
n = number of rows of the decision table;
m = number of columns of the decision table;
$type$ = type of the relation;
for *column* 4 *TO* n **do**
  for *row* 0 *TO* m **do**
    $type$ = $DecisionTable[row][0]$;
    $var1$ = $DecisionTable[row][1]$;
    $operator$ = $DecisionTable[row][2]$;
    $var2$ = $DecisionTable[row][3]$;
    if $var1$ *is not in* $variablelist$ **then**
      Add $var1$ to the $variablelist$;
    end
    if $type$ = $boundarycondition$ **then**
      if $DecisionTable[row][column]$ = F **then**
        $operator$ = $complement(operator)$;
      end
      Add the $condition$ ($type$,$operator$,$var2$) to $var1.conditionlist$
    end
    else if $type$ = $variablerelation$ **then**
      if $DecisionTable[row][column]$ = T **then**
        Add the $condition$ ($type$,$operator$,$var2$) to $var1.conditionlist$
      end
    end
    Generate Test Case;
    Step 1: Generate test cases according to boundary conditions of the variables in variablelist;
    Step 2: Modify the test values by considering the relations between the variables;
    Clear the variablelist;
  end
end

Algorithm 1: Test Case Generation

3. Test Case Generation

While testing a system, a model of the system helps to predict and control its behavior. Modeling a system acquires the understanding of its abstraction, and in the case of testing GUIs, there is the need of a formal specification tool distinguishing between legal and illegal situations. Decision tables [1] are popular in information processing and are also used for testing, e.g., in cause and effect graphs [12]. A decision table logically links conditions ("if") with actions ("then") that are to be triggered, depending on combinations of conditions ("rules") [5]. Equivalence class testing supplemented with boundary value analysis [16], [8] is used to generate test cases. Equivalence class testing is strengthened by the cause-effect testing approach which uses decision tables to generate test cases where the input conditions represent the causes and actions represent the effects.

A test case generation algorithm is developed to generate test case values from a decision table by considering the boundary conditions (restrictions) and relations between the variables (dependencies). Algorithm 1 shows the test case generation algorithm.

For all the rules in the decision table, a specific test value of a variable is generated by regarding all its conditions. First four columns in the decision table are for holding the type of the relation, variable1, operator and variable2 (can be boundary or variable, due to the type of the relation) respectively.Starting from index 4, the columns hold the related rule (true or false) for that variable1, variable2 and operator combination. If one rule for the current row is false (represented as "F"), then the logical operator is complemented. The decision table that is used by the test case generation algorithm is shown in Table 1 (see Section 6).

4. Boundary Overflow Vulnerability Detection

A vulnerability is characterized as a "Boundary overflow" when the input being received by a system causes the system to exceed an assumed boundary resulting in a vulnerability [9]. Instead of inserting dynamic checks into the generated code as in [15], our approach uses static control condition insertion into the source code to avoid boundary overflow in advance.

We propose a detection algorithm to check the error handling mechanism of the SUC related to boundary over-

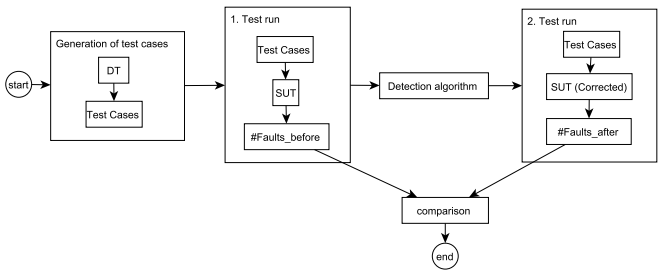

Figure 1. Summary of the approach

flow. The algorithm scans the source code statically, detects the points that may cause problems (possible violation of boundary values) and checks the error handling mechanism of the SUC against "out of boundary" values. The deficient parts of the error handling mechanism related to boundary overflow have to be identified first. Once detected, a mechanism is required to correct the deficiencies of the SUC. Our algorithm inserts an error handling mechanism with following features:

- insertion of necessary control conditions into the source code where control for the boundary overflow does not exist
- generation of related error messages when inputs are given that lead to boundary values
- exits the currently executed function.

The approach starts with generating test cases as defined in Section 3. After generating the test cases, the SUC is tested dynamically by entering these values to its user interface. The faults are obtained and extracted to a file. Applying our proposed detection and correction method, the new corrected version of the SUC is tested in the real environment again. The faults before and after applying our method are compared. The summary of the approach is shown in Figure 1.

Boundary overflow vulnerability detection algorithm consists of five steps. In step 1, the variable definitions with the specified types are obtained and the variables (as type, name, defined file, defined line, defined function) are entered into the hash table. In step 2, the variables in the hash table are matched with the conditions defined in decision table. Matched variable's boundary condition is set to true. Step 3 of the algorithm detects the points that may cause problems (input assignments from the user interface input) and sets the trace line of the variable as the current line.

Step 4 traces the variables from their trace lines and detects the lines where the variables are used first. If a variable is used in a control statement (like if, while or for), the expression(s) in the statement are parsed. After that, the conditions are compared with the defined conditions of the variable. If the parsed expression complies with the defined condition of the variable, condition check of the variable is set to true (i.e., if the defined condition is a $>$ 0, the expression should be a $<=$ 0 to catch the undesired input). Step 5 of the algorithm applies the correction mechanism. The error handling code is inserted after the trace line of the variable where the condition check for the variable does not exist. The boundary overflow detection algorithm (Algorithm 2) is represented below.

Input: Decision table
Output: Variable list
n = number of lines;
m = size of hashtable;
Step 1. Obtain the variable definitions with the specified types and add them to the variable list;
for *line* i = 1 *TO* n **do**
  if *line* i *contains a variable definition of specified types* **then**
    Add the $variable$ to $hashtable$ ($type$,$name$,$definedfile$,$definedline$,$definedfunction$);
  end
end
Step 2. Match the conditions to variables;
Match the variables with the conditions defined in decision table;
Step 3. Detect the points that may cause problems (GUI input lines);
for *line* i = 1 *TO* n **do**
  if *line* i *contains a GUI input* **then**
    if $assignedvariable$ *is of string type* **then**
      Trace the string variable and find the string to integer or double conversion line;
      Lookup the $assignedvariable$ from $hashtable$;
      if $variable.boundarycondition$ = *true* **then**
        $variable.usedfile$ = $currentfile$;
        $variable.traceline$ = $currentline$;
      end
    end
  end
end
Step 4. Trace the variables from the trace line numbers;
for *variable.traceline* i = 1 *TO* n **do**
  Find the first line that the variable is used;
  if *line* i *contains a control statement (if, while, for)* **then**
    Parse the expression(s);
    Lookup the variable(s) used in the expression from $hashtable$;
    if $variable.boundarycondition$ = *true AND* $expression$ = *complement(*$variable.condition$*)* **then**
      $variable.condition.check$ = true
    end
  end
  else if *line* i *contains a GUI input* **then**
    Go back to the statements in Step 3;
  end
end
Step 5. Apply correction mechanism;
for *all the variables in hash table* i = 1 *TO* m *DO* **do**
  if $variable.boundarycondition$ = *true AND* $variable.condition.check$ = *false* **then**
    Insert the related error handling code after the conversion line;
  end
end

Algorithm 2: Boundary Overflow Vulnerability Detection Algorithm

5. Implementation and Tool Support

For the implementation of our approach as introduced in Section 4, we developed a tool in Java. As a static analysis tool, it analyzes the source code of SUC, finds the deficient parts that may cause boundary overflow vulnerability and inserts related codes statically into the source code to compensate the deficiencies of the control mechanism related to boundary overflow. The tool runs on Microsoft Windows and works for software developed in C++.

The boundary overflow analysis tool takes two inputs: (1) the directory of the software to be analyzed and (2) decision table for the GUI. Our implementation requires a manual matching of the listed variables with the conditions in the decision table to identify the conditions of the variables. The tool outputs the variables that have the boundary con-

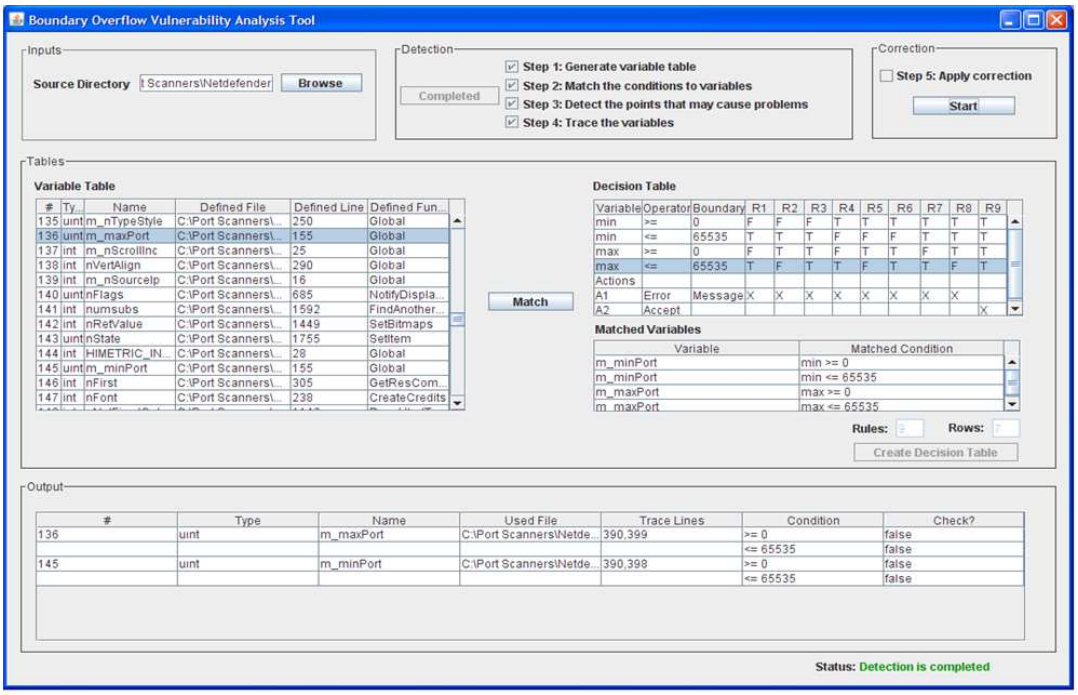

Figure 2. Boundary overflow vulnerability analysis tool graphical user interface screen

dition, displays the conditions of the variables as well as whether or not condition checks exist in the source code related to boundary overflow.

The correction mechanism is applied by informing the user about the insertion of the error handling code where the condition checks do not exist. Figure 2 shows GUI part of the tool that enables to input source directory of the software to be checked, shows the detection steps, provides the editable decision table, correction suggestions and displays the outputs.

6. Case Study

We evaluated our approach and the tool introduced in Section 5 on a port scanner. A port scan function scans a single port or a range of ports, i.e., ports between a given minimum and maximum, to check whether they are open or not. Test cases are generated for minimum port and maximum port from the decision table using Algorithm 1. Test of the port scan function is evaluated in a LAN and faults have been recorded. As a next step, our tool analyzed the source directory to detect and correct the vulnerabilities related to boundary overflow. Finally, the faults detected before and after applying the boundary overflow detection algorithm are compared.

We exemplify the case study on the basis of the port scanner part of open source firewall software, i.e., Netdefender Firewall (version 1.5) [13]. Its GUI is shown in

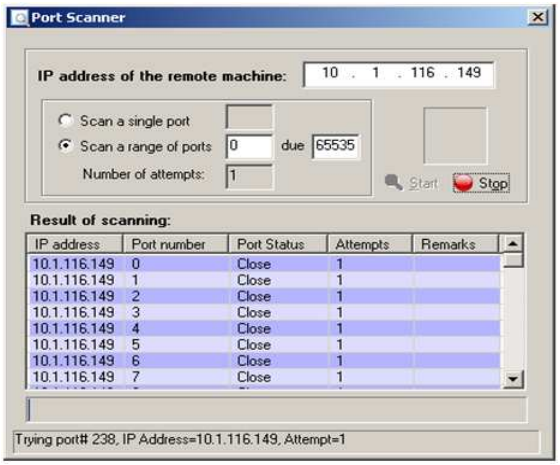

Figure 3. Netdefender Port Scanner

Figure 3. The user interface behavior of its port scan function is tested by modeling through a decision table and by applying our tool.

Table 1 structures the decision process by modeling possible actions for related conditions. The decision table is built to generate test data for the minimum and maximum port values of the port scanner according to the rules. ”Min” and ”max” are used many times in the table due to their relation with each other and the boundary values. Algorithm 1 is applied to generate test data according to the rules of the

Table 1. Decision Table for "Enter min and max ports"

Conditions	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15
min >= 0	F	F	F	F	F	T	T	T	T	T	T	T	T	T	T
min <= 65535	T	T	T	T	T	F	F	F	F	F	T	T	T	T	T
max >= 0	F	F	F	T	T	F	T	T	T	T	F	T	T	T	T
max <= 65535	T	T	T	F	T	T	F	F	F	T	T	F	T	T	T
min < max	F	F	T	T	T	T	F	F	T	F	F	T	F	F	T
min = max	F	T	F	F	F	F	F	T	F	F	F	F	F	T	F
min > max	T	F	F	F	F	F	T	F	F	T	T	F	T	F	F
Actions															
A1: Error1	X	X	X	X	X	X	X	X	X						
A2: Error2	X	X	X	X		X	X	X	X		X	X			
A3: Error3	X					X	X			X	X		X		
A4: Accept														X	X

Table 2. Test cases for min,max pairs

Rule	Value (min,max)
R1	(-1,-2)
R2	(-1,-1)
R3	(-2,-1)
R4	(-1,65536)
R5	(-1,0)
R6	(65536,-1)
R7	(65537,65536)
R8	(65536,65536)
R9	(65536,65537)
R10	(65536,0)
R11	(0,-1)
R12	(0,65536)
R13	(65535,0)
R14	(0,0)
R15	(0,65535)

decision table. For each rule, a test pair is generated based on equivalence class testing and boundary value approach. The constraints in the first part (rows 1-4) of the decision table indicate the boundary conditions. Meanwhile, the constraints in the second part (rows 6-8) indicate the relations of the variables with each other.

The algorithm creates a list for each constraint containing the conditions of the variables and generates the min and max test case pairs according to the fact that the boundaries for port values are defined as 0 for minimum and 65535 for maximum possible boundary. Table 2 presents test cases generated by using decision table given in Table 1. Figure 4 shows the generated test values as the output of the test data generation algorithm.

The port scanner is evaluated in a LAN and the generated test values are applied as inputs to the GUI of the port scanner. The user interface outputs are obtained and the network packet outputs are captured. The outputs are extracted to a spreadsheet document. Figure 4 shows a sample view of the spreadsheet document, which displays the test values as input pair, GUI and network packet outputs, state of the case (erroneous or not), error message and error type.

To sum up, the cases with out of boundary input pairs give rise to problems in the network environment. In certain cases (2, 3, 6, 8, 9, 11, 12), the corresponding error is a Type II error (false negative). In these cases, there are faulty input pairs that are out of boundary values but the program behaves as they are not faulty. This is critical, because the program does not abandon processing the related task, hence the resulting situation forces the program to work erroneously. For this reason, in some cases (2, 3, 6, 11), the client does not stop sending the TCP packets to the target computer, keeps on sending the packets from 1 to 65535 in an infinite loop and therefore generates a flood in LAN.

The test results of the SUC are presented above. It is observed that the original software does not have control mechanisms for the out of boundary input values that can cause boundary overflow. Since the SUC has no error handling mechanisms, our tool inserts control statements to fulfill the deficiencies of the software. After the insertion of control statements related to boundary constraints in the port scanner of Netdefender firewall, the software is evaluated in LAN again and the generated test cases are applied as inputs to the GUI of the port scanner. The outputs considerably differ from the ones in Figure 4. In erroneous cases (1-13), the software outputs the right error message and aborts sending the packets. Figure 5 displays the outputs of the test cases executed on the SUC modified by our correction algorithm. It is evident that our tool has successfully carried out detection and correction operations. The uncorrected fault in the test runs 1 and 2 is due to the fact that our tool does not yet check dependencies between variables.

Analysis of the evaluation results encourages the generalization that boundary overflow vulnerabilities are not considered and thus counter-measure actions are neglected during software development. Therefore, tools as we introduced in this paper might be useful to prevent likely failures or undesirable situations that may occur as a consequence of deficiency control mechanism in the software.

#	Input Pair	GUI Output	Network Packet	Erroneous	Error Message?	Error Type
1	(-1,-2)	No output	No packet	Yes	The maximum range cannot be less...	
2	(-1,-1)	(-1,0,1,2,3,...==)	65535,1,2,...65535...	Yes	No	Type II: False negative
3	(-2,-1)	(-2,-1,0,1,2,3,...==)	65534,65535,1,2,...65535...	Yes	No	Type II: False negative
4	(-1,65536)	No output	No packet	Yes	The maximum range cannot be less...	
5	(-1,0)	No output	No packet	Yes	The maximum range cannot be less...	
6	(65536,-1)	(65536...==)	1,2,...65535,1,2,....65535...	Yes	No	Type II: False negative
7	(65537,65536)	No output	No packet	Yes	The maximum range cannot be less...	
8	(65536,65536)	(65536)	No packet	Yes	No	Type II: False negative
9	(65536,65537)	(65536,65537)	1	Yes	No	Type II: False negative
10	(65536,0)	No output	No packet	Yes	The maximum range cannot be less...	
11	(0,-1)	(0...==)	1,2,...65535,1,2,....65535...	Yes	No	Type II: False negative
12	(0,65536)	(0...65536)	1,2,...65535	Yes	No	Type II: False negative
13	(65535,0)	No output	No packet	Yes	The maximum range cannot be less...	
14	(0,0)	(0)	No packet	No		
15	(0,65535)	(0...65535)	1,2,...65535	No		

Figure 4. Outputs of the test cases

#	Input Pair	GUI Output	Network Packet	Erroneous Case	Error Message?
1	(-1,-2)	No output	No packet	Yes	Input Validation Error
2	(-1,-1)	No output	No packet	Yes	Input Validation Error
3	(-2,-1)	No output	No packet	Yes	Input Validation Error
4	(-1,65536)	No output	No packet	Yes	Input Validation Error
5	(-1,0)	No output	No packet	Yes	Input Validation Error
6	(65536,-1)	No output	No packet	Yes	Input Validation Error
7	(65537,65536)	No output	No packet	Yes	Input Validation Error
8	(65536,65536)	No output	No packet	Yes	Input Validation Error
9	(65536,65537)	No output	No packet	Yes	Input Validation Error
10	(65536,0)	No output	No packet	Yes	Input Validation Error
11	(0,-1)	No output	No packet	Yes	Input Validation Error
12	(0,65536)	No output	No packet	Yes	Input Validation Error
13	(65535,0)	No output	No packet	Yes	The maximum range cannot be less than...
14	(0,0)	(0)	No packet	No	
15	(0,65535)	(1...65535)	1,2,...65535	No	

Figure 5. Outputs of the test cases after correction

7. Conclusion

In this paper, we have proposed a solution for the boundary overflow vulnerability problem. Decision tables are used for modeling GUI of SUC and generating test cases for input validation. An algorithm is introduced to validate the error handling mechanism of SUC related to boundary overflow and provides it with necessary exception handling mechanism where none exists. A port scanner has been tested for evaluation of our tool. Results of tests show that the approach is very effective for finding deficiencies in the error handling mechanism of SUC concerning boundary overflow problems. Moreover, our approach inserts appropriate checks into the source code of SUC to compensate those deficiencies of SUC. Our tool can be viewed as a novel extension of well-known tools such as pc-lint [14] for boundary overflow analysis. Our future plans include adding the capability to check the dependencies between the variables to our tool and extending our formal modeling with decision table augmented ESGs.

References

[1] I. 5806. Specification of single-hit decision tables. *Information processing*, 1984.

[2] P. Amman and J. Offutt. Introduction to software testing. 2008.

[3] F. Belli. Finite state testing and analysis of graphical user interfaces. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, Washington, DC, USA, 2001, pp. 34-43. IEEE.

[4] F. Belli, A. Hollmann, and N. Nissanke. Modeling, analysis and testing of safety issues - an event-based approach and case study. In *Proceedings of the 26th Int. Conf. Computer Safety, Reliability, and Security*. Springer, 2007, pp. 276-282.

[5] F. Belli and M. Linschulte. On negative tests of web applications. *Annals of Mathematics, Computing and Teleinformatics*, 1(5), 2007, pp. 44-56.

[6] J. H. Hayes and J. Offutt. Input validation analysis and testing. *Empirical Software Engineering*, 11(4), 2006, pp. 493-522.

[7] P. Jorgensen. Software testing: a craftman's approach. *CRC Press*, 2002, pp. 359.

[8] H. Liu and H. B. K. Tan. Covering code behavior on input validation in functional testing. *Information and Software Technology*, 51(2), 2009, pp. 546-553.

[9] P. Mell and M. C. Tracy. Procedures for handling security patches. *NIST Special Publication 800-40*, 2002.

[10] B. Meyer. Applying "design by contract". *Computer*, 25(10), 1992, pp. 40-51.

[11] MSDN. Design guidelines for secure web application. In *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod77.asp*, 2009.

[12] G. J. Myers. The art of software testing. *John Wiley and Sons*, 1979.

[13] Netdefender. Netdefender firewall version 1.5. In *http://www.codeplex.com/netdefender*, 2009.

[14] Pc-lint. In *http://www.gimpel.com/html/pcl.htm*, 2008.

[15] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, and T. Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Proceedings of the 20th ACSAC Conference*, 2004, pp. 82-90.

[16] T. Tuglular. Test case generation for firewall implementation testing using software testing techniques. In *Proceedings of the International Conference on Security of Inform. and Networks*, N. Cyprus, 2007, pp. 196-203.

[17] W. Zheng and G. Bundell. Test by contract for uml-based software component testing. In *Proceedings of the Int. Sym. on Comp. Sci. and its Appl.* IEEE, 2008, pp. 377-382.