Security mutation testing of the FileZilla FTP server

CONFERENCE PAPER · JANUARY 2011

DOI: 10.1145/1982185.1982493 · Source: DBLP

Daniel Woodraska
Dakota State University
3 PUBLICATIONS **18** CITATIONS
SEE PROFILE

Michael Sanford
Frostburg State University
5 PUBLICATIONS **19** CITATIONS
SEE PROFILE

Dianxiang Xu
Boise State University
107 PUBLICATIONS **1,039** CITATIONS
SEE PROFILE

Security Mutation Testing of the FileZilla FTP Server

Daniel Woodraska, Michael Sanford, Dianxiang Xu
National Center for the Protection of the Financial Infrastructure
College of Business and Information Systems
Dakota State University
Madison, SD 57042, USA

dcwoodraska@pluto.dsu.edu, {michael.sanford, dianxiang.xu}@dsu.edu

ABSTRACT

Security has become a priority for software development and many security testing techniques have been developed over the years. Benchmarks based on real-world systems, however, are in great demand for evaluating the vulnerability detection capability of these techniques. To develop such a benchmark, this paper presents an approach to security mutation analysis of FileZilla Server, a popular FTP server implementation as a case study. In the existing mutation testing research, mutants are created through syntactic changes. Such syntactic changes may not result in meaningful security vulnerabilities in security-intensive software. Our approach creates security mutants by considering the causes and consequences of vulnerabilities. The causes of vulnerabilities include design-level (e.g., incorrect policy enforcement) and implementation-level defects (such programming errors as buffer overflow and unsafe function calls). The consequences of vulnerabilities refer to various potential attacks, such as spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege (STRIDE). Using this approach, we have created 30 distinct mutants for FileZilla Server. They have been applied to the evaluation of two security testing methods that use attack trees and attack nets as threat models for test generation. The results show that, while these testing methods can kill most of the mutants, they have an important limitation – they cannot detect the vulnerabilities that are not captured by the threat models.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – *testing tools.*

General Terms

Reliability, Security, Verification

Keywords

Security testing, mutation analysis, software testing, FTP

1. INTRODUCTION

Testing for security is an important approach to security assurance of software. As security has become a priority for software development, many security testing techniques have been developed over the years [6]. Unfortunately, vulnerabilities in software products continue to be reported and exploited. Benchmarks based on real-world systems are in great demand for evaluating the vulnerability detection capability of security testing techniques.

Mutation has been used for evaluating software testing in the past several decades. The basic idea of mutation testing is to run test cases against mutants where faults are injected deliberately. These faults would represent the same faults a programmer might make. A mutant is said to be killed if one of the tests reports a failure. Usually, the percentage of mutants killed by the tests is an indicator of how effective the tests are [1]. So far, mutation testing research has focused on mutation creation through syntactic changes, such as replacing && (and) with || (or) and 0 with 1 [4]. However, syntactic changes may not result in meaningful vulnerabilities in security-intensive software.

This paper presents mutation analysis of FileZilla Server[1]. Our goal is to make FileZilla Server a benchmark for security testing techniques by systematic injection of security vulnerabilities. We chose FileZilla for several reasons. First, FTP is a widely used network-based file transfer protocol. Although FTP has been in use for decades, we found that various FTP implementations had security problems reported as late as 2010. Second, FileZilla is an open source project. The very nature of open source provides an excellent vehicle for which mutants can be created and tested. As an FTP server implementation, FileZilla Server is currently the sixth most downloaded program on SourceForge[2]. Third, FileZilla Server appears to be very stable and secure. Although various bugs have been reported on different FTP implementations, only a small number of them are about FileZilla Server.

Based on the analysis of system functionality and security requirements, our approach creates security mutants according to the causes and consequences of vulnerabilities. The causes of vulnerabilities include design-level defects (e.g., incorrect policy enforcement) and implementation-level programming errors (e.g., buffer overflow and unsafe function calls). Design-level vulnerabilities are a major source of security risks in software

[1] http:// filezilla-project.org/

[2] http://sourceforge.net/top/topalltime.php?type=downloads

[21]. For example, around 50% of the security flaws uncovered during the Microsoft's "security push" in 2002 were closely related to design-level problems [3]. The consequences of vulnerabilities refer to various potential attacks, such as spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege (STRIDE) [17]. We have created 30 mutants of FileZilla Server, which has also taken its bug history into consideration. Then we have used these mutants to evaluate two security testing methods that generate security tests from threat models represented by attack trees [6] and Petri nets [21], respectively. The results show that these testing methods have an important limitation – they cannot kill the mutants that are not captured by the threat models.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 describes the process used for creating security mutants. Section 4 discusses the mutants created for FileZilla Server. Section 5 presents the evaluation of two security testing techniques with the mutants. Section 6 concludes the paper.

2. RELATED WORK

There is a large body of literature on mutation testing [4]. The existing research focuses on mutant creation through syntactic changes. Nevertheless, there are two groups of work on mutation for security testing. One uses mutation for testing access control policies and the other for testing XSS and SQL injection attacks against web applications. In this paper, we focus on mutation analysis of general vulnerabilities for non-web applications.

For mutation testing of access control policies, there are two main systems used to write access control policies. One uses OrBAC (Organization-Based Access Control) or RBAC (Role-Based Access Control) as a modeling scheme to abstractly design a security policy without the implementation [5][10][12]. The other uses XACML (eXtensible Access Control Markup Language) to create the actual implementation of a security policy [7][8]. Some research has used both to implement a model [9]. The above work focuses on security testing of access control policy, rather than general security testing of software. Although access control is an important aspect of security (particularly authentication and authorization), security concerns go beyond access control, such as integrity and availability. In policy testing, test inputs are requests and test outputs are replies. The faults for the mutants are seeded in the policy. By passing the request through the policy, the replies will show whether the test should be killed by comparing them to the original [7].

The creation of mutants in policy testing can be automated by using mutation operators, such as type changing operators, parameter changing operators, hierarchy changing operators, and rule adding operators. Type changing operators modify rule types. Instead of a prohibition rule, the new rule would be a permission rule. Parameter changing operators change a rule's parameter, which could change a role into a different role. For instance, a role of administrator could be changed to a limited user to create the mutant. Hierarchy changing operators are used to replace a parameter by a parent or one of its descendants. Rule adding operators would add a rule to the existing policy to simulate cases of an additional requirement [10]. These operators represent the majority of what are used to create mutants for both XACML and OrBAC/RBAC. As such, the mutation analysis of a security policy turns out to be similar to functional mutation analysis in that it primarily relies on syntactic changes.

Shahriar and Zulkernine have developed the MUTEC and MUSIC tools for mutation testing of XSS and SQL injection attacks against web applications. MUTECT provides 5 and 6 mutation operators for modifying JavaScript code and PHP code, respectively [15]. MUSIC provides four operators for injecting faults into where conditions of SQL queries and five operators for injecting faults in database API method calls [16]. Tuya et al. [19] have developed the SQLMutation tool that provides four categories of mutation operators for SQL SELECT queries. They include SQL clause operators (e.g., replacing SELECT with SELECT DISTINCT), operator replacement mutation operators (e.g., AND is replaced by OR), NULL mutation operators (e.g., replacing NULL with NOT NULL), and identifier replacement (e.g., replacing one column name with other of similar types). Fonseca et al. [2] assess the effectiveness of web scanning tools by injecting XSS and SQL injection faults in web applications. Webgoat[3] is a deliberately insecure web application. It can be used to evaluate security techniques for web applications.

Testing for security is hard in that it is impossible to test a real-world program against all unintended behaviors or invalid inputs [18]. Even worse, security testing needs to test the "presence of an intelligent adversary bent on breaking the system"[11]. Recent research has demonstrated that threat models can provide a basis for security testing because they describe security threats from the standpoint of how the adversary would attack or exploit a system [6][20]. Marback et al. [6] have proposed an approach to security testing with threat models represented by attack trees. Attack trees [13] are a widely applied representation of threat models for secure software development. In an attack tree, an attack path is a sequence of primitive attack conditions and actions that represents a particular way to achieve the attack goal. Attack paths can be generated automatically from a given attack tree and then further transformed into security tests. Wang et al. [20] have proposed approach to security testing with threat models of security policies modeled by UML sequence diagrams. A set of threat traces is extracted from a given threat model. Each threat trace is an event sequence that should not occur during the system execution; otherwise it becomes an attack.

3. PROCESS OF CREATING MUTANTS

Unlike the existing mutation testing research that relies on syntactic changes, our approach is based on the program semantics, which requires an in-depth understanding about the system functionality and security requirements. To create mutants, we inject security vulnerabilities manually by commenting out/deleting code, modifying code, or writing new code. The process consists of four steps: understanding the system functionality and security requirements, understanding the program structure, identifying and creating mutants, and finally creating demonstration tests to check the mutants.

For the first step, we try to understand the system functionality and security requirements. This results in a list of functions or services, together with their security requirements. For FileZilla Server, the analysis starts with the various RFC (959, 1123, 3659, 4217) memorandums published by IETF (Internet Engineering

[3] http://code.google.com/p/webgoat/

Task Force). They provide complete specification of FTP services. The services include login, QUIT command, change directory, get current directory, change to passive mode, put a file on the server, get a file from the server, create a directory, delete a directory, delete a file, execute commands on the server, rename a file, append a file, remote administration of the server, and file logging. The first column of Table 1 shows the list of services. The security requirements for these services include confidentiality, integrity, availability, and non-repudiation.

Table 1: List of services and mutants

Service	Total	S	T	R	I	D	E
Login	11	4		1	1	3	3
QUIT	2					2	
Change directory	1					1	
Get current directory	2				2		
Change to passive mode	1					1	
Put file to server	1		1				1
Get file from server	1				1		1
Create directory	1		1				1
Delete a directory	3		2	1		1	2
Delete a file	1		1	1			1
Execute commands on server	1					1	
Rename a file	1		1				1
Append to a file	1		1				1
Remote Administration	2	1	1				2
File Logging	1			1		1	
Total	30	5	8	4	4	9	13

In the second step, we understand the structure of the FileZilla Server program. FileZilla Server version 0.9.34 used in this study has 94,017 lines of C++ code, 109 classes, 213 functions, with an average method complexity of 4.8. Method complexity is defined as the number of distinctive execution paths that can be made through a function or method. To understand the structure of the FileZilla Server program, we look through each file of source code and give it a brief description of what it controls and what header files it needs. Then, we map out and make a model of the source code to see which files affect what. Performing this process give us a better understanding of how changing one item might affect the others. This helps ensure that the mutant created would affect only what it is expected to affect. Also by reviewing the code several times, we can take notes on possible mutants and see the comments of the programmer for insight as to why certain parts of the server are coded in a certain way.

The third step is to identify and create mutants. To ensure that the mutants cover a broad range of security vulnerabilities, we consider the various causes (design-level and implementation-level defects) of vulnerabilities and the consequences of vulnerabilities (e.g., STRIDE attacks). This helps identify points of entry and seek out places of weak links in the code. Mutants are created by going through the list of FTP services and related code. Each mutant is made into a separate FileZilla Server program because the use of multiple vulnerabilities could cause unforeseen side effects upon each other. We also review the bug tracker for FileZilla to find the real vulnerabilities (not general bugs) that have occurred in earlier versions.

The final step is to develop demonstration test cases to verify the mutants. A demonstration test case for a mutant is a test that can reveal the vulnerability injected in the mutant. It shows that the mutant and the original program result in a different behavior. This will guarantee that the mutants have the security vulnerabilities as expected. Using an FTP client, we would log into the mutant FileZilla server program and test the mutant for the security vulnerability we have put into the mutant program. Both WinSCP and SmartFTP are used to test each mutant. Sometimes the way the FTP client is programmed would not allow us to take advantage of the security vulnerability during testing. For instance, in the mutant that causes a denial of service when an extremely large command is entered could not be done with WinSCP. This is because the WinSCP client would not allow more than 20 characters when trying to enter the command. For this mutant only SmartFTP could be used for the testing. To facilitate the process, we also use a record and replay tool called Test Automation FX to record and replay the demonstration tests. When creating and checking mutants, we also verify whether a mutant has the same behavior as an existing one. If multiple mutants have the same behavior, we only use one of them.

4. FILEZILLA SERVER MUTANTS

Following the process in Section 3, we have created 30 distinct mutants so far. As shown in Table 1, the mutants have covered the common FTP services. They have involved both design-level (13) and implementation-level (17) vulnerabilities. A design-level vulnerability is one that results from a design defect. For example, a mutant allowing a user to create directories without the necessary permissions has a design-level vulnerability. This is because no matter what the programmer does it is an unfixable problem using the current design. Implementation-level vulnerabilities are due to programming errors, e.g., use of insecure data structures and functions. For example, using a fixed array for storing FTP commands can suffer from buffer overflow when an extremely lengthy command is provided maliciously. This is an implementation issue because it could have been avoided within the existing design. Another example is a mutant that causes a memory leak to occur because of a commenting out of code that would have otherwise deleted the socket. Other implementation types of issues include unsanitized output, data taint, unsafe function calls, and conditional short circuits.

When creating the mutants, we have also placed an emphasis on the types of security attacks caused by the injected vulnerabilities. We use the well-known STRIDE classification system from Microsoft's threat modeling approach [17]. They provide a systematic coverage of security attacks that violates the security requirements, such as confidentiality, integrity, availability, and non-repudiation. Table 1 shows that there are 5, 8, 4, 4, 9, and 13 mutants that can lead to Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege, respectively. Note that a single vulnerability can be exploited by multiple types of attacks. In the following, we discuss the vulnerabilities related to these attacks.

The spoofing category is for attacks that are focused on using someone else's identity or posing as a system component to gain access to a system [17]. This can lead to other major security issues such as information disclosure and elevation of privileges. One mutant created to fit this category is a programming fault that lets users enter in false credentials into the login and even though these credentials are not in the server the FTP server will logon as the first user that is put into the server. An exception to the mutant is using a blank username. The server checks for a blank username and immediately denies the USER command. This is not a real limitation though because any other strings an attacker uses will be accepted. The mutant was created by modifying the code in the multiple source files as well as adding new code into these files. Depending on the rights of the first user created this could be high risk security vulnerability.

The tampering category is for threats that try to modify data within a system for malicious purposes [17]. Mutants created for this classification deal with allowing the deletion and creation of files and directories without the correct permissions to do so. One such mutant created for this category affects the permissions for deleting files. This of course can lead to modification of data for malicious purposes and is therefore classified as a tampering mutant. With no check being made on whether a user has deletion of file permissions, any user on the FTP server could delete files.

The repudiation category is for attacks where an attacker denies performing the attacks that were said to take place. Often these attacks are directed at logging files so that there is no proof of the attacker being on the system [17]. Our mutants in this category range from being able to delete log files to there being no log files ever created. One mutant focuses on the ability for the server to keep logs. It modifies the data structure holding the string of the pathname where the log is supposed to be created.

The information disclosure category is for attacks that reveal protected data that a user is not supposed to be able to access [17]. One such mutant affects the permissions for being able to see the subdirectories. The mutant is made by commenting out the lines of code that asked for a permissions check for viewing subdirectories when a listing of directory is called. Because there is no check being made on whether a user has these permissions to view the subdirectories, any user will be able to gain information about an entire subdirectory tree on the system. This mutant could allow attackers to gain information on the system that would have otherwise been unavailable to them.

The denial of service category is for attacks that occur when a legitimate user is prevented from using the normal functionality of a system [17]. Mutants in this category are varying and often very different from each other. A mutant created for this category uses execution of a 2100 character command to cause the FTP server service to stop making it unreachable to any legitimate user. The command can be larger than 2100, but that is the absolute minimum amount of characters to cause the denial of service. This mutant is created by changing the value of how many characters are allowed to be accepted by the server when a command is entered. Using the command line portion of the SmartFTP client, the user is allowed to enter in commands to type in the FTP commands themselves rather than use the GUI of the client to send the commands for them. By entering a command this way, we can send a very large string that the server could not handle causing an error that quits the FTP service running. The original version would have denied the attempt by shortening the command to 2000 characters and deny the command if it is not a valid command.

The elevation of privileges category focuses on when an attacker uses a vulnerability to get a higher trust level on a system then what they currently have [2]. Elevation of privileges is considered to be the most dangerous of attacks. The elevation of privilege mutants created usually deal with permissions being taken out allowing for a user without those permissions the same access as someone with them. One mutant created allows a user to put files on the server without the permissions set for them to do so. By commenting out the permission check when the STOR command is given, the mutant server does not check and the user will be allowed to put files up on the server.

5. EVALUATION

The mutants have been applied to two testing techniques, which generate security tests from threat models, represented by attack trees [6] and attack nets [21], respectively. Threat models describe how the adversary can perform security attacks. They result from threat modeling, which has become a viable practice for secure software development [17]. It is worth pointing out that the mutation analysis and the applications of the two testing techniques are performed independently by different researchers. The security tests created from each technique are used to exercise all mutants. The percentage of mutants killed by the tests is considered as an indicator of the vulnerability detection capability.

5.1 Security Testing with Attack Trees

The technique for security testing with attack trees [6] first builds threat models as attack trees, generates attack paths from attack trees, converts attack paths into security tests, and executes the security tests against the System Under Test (SUT). The root of an attack tree specifies the ultimate goal of an attack; child nodes represent the sub-goals of their parent node; and leaf nodes describes the primitive attack goals (conditions or actions). In an attack tree, an attack path is a sequence of primitive attack conditions and actions that represents a particular way to attack the SUT.

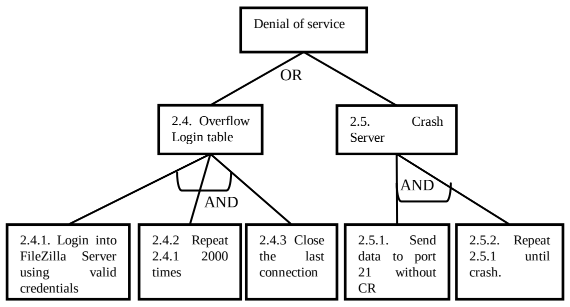
Denial of service
OR
2.4. Overflow Login table
2.5. Crash Server
AND
AND
2.4.1. Login into FileZilla Server using valid credentials
2.4.2 Repeat 2.4.1 2000 times
2.4.3 Close the last connection
2.5.1. Send data to port 21 without CR
2.5.2. Repeat 2.5.1 until crash.

Figure 1. An attack tree of denial of service attacks

Figure 1 shows a tree of denial of service threats that abuse the login service to implement the attacks. Sub-goal 2.4 tries to overflow the login table by continuing to create login connections until the system cannot create any more. Sub-goal 2.5 attempts to overload the input buffer on port 21. FTP requires a carriage return (CR) to be sent when a command is complete. In this case, data is sent to port 21 until the buffer overflows and the server crashes. Thus, the two attack paths in Figure 1, <2.4.1, 2.4.2, 2.4.3> and <2.5.1, 2.5.2>, represent two security tests.

To evaluate the above testing technique, we created an initial set of attack trees which were converted to 39 security tests. These security tests were then applied to an initial set of mutants. The initial set of security tests killed 57.14% of the initial set of mutants. The remaining mutants were not killed because the vulnerabilities were not captured by the attack trees. Nevertheless, we continued to build more attack trees which resulted in 12 more security tests to exercise the mutants. The new set of 51 security tests was able to kill 92.86 % of the initial set of mutants. As we continued to build more attack trees, create more security tests, and create more mutants (independently of the attack trees), we eventually we were able to kill 96.67 % of all created mutants. The remaining mutants fell into a classification called memory leaks. Due to the limitations of threat modeling, memory leak vulnerabilities were not able to be killed. An example of this is a mutant on the QUIT command. This mutant would not release the resources after the QUIT operation was executed. This vulnerability can lead to a denial of service attack. This type of vulnerabilities is difficult to capture by threat models.

5.2 Security Testing with Attack Nets

Xu and Nygard have developed a formal approach to threat-driven modeling of secure software with Predicate/Transition (PrT) nets [21]. In this approach, PrT nets are used as a unified formalism for modeling system functions, security threats, and security features. By modeling security threats, security features can be suggested and evaluated so as to achieve a secure software design. As high-level Petri nets, PrT nets can capture both control flows and data flows. They are more expressive than attack trees, e.g., in capturing complex attacks with partially ordered actions. Presence (or absence) of the security threats can be verified against the system functions before (or after) the security features are applied.

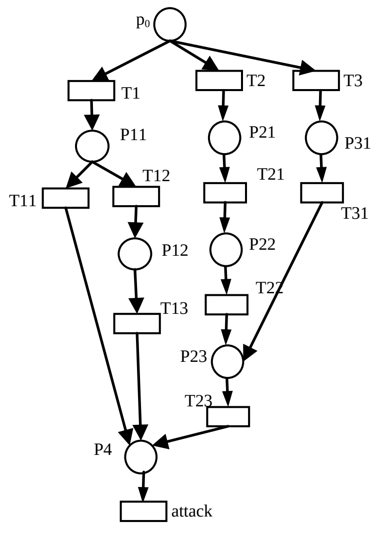

Figure 2. An attack net of elevation of privilege attacks

Here we refer to the PrT nets of threat models as attack nets. Figure 2 shows an attack net of four elevation of privilege attacks. The first two <T1, T11> and <T1, T12, T13> exploit buffer overflow on the USER and PASS commands respectively. The other two <T2, T21, T22, T23> and <T3, T31> make use of the Administration interface options: auto ban and lock server. The auto ban feature locks out the IP address after 10 failed login attempts for a minimum of one hour. The lock server option locks the server, preventing any new logins from succeeding.

Attack nets and attack trees represent threats in different ways. The process of security testing with attack nets include creating attack nets, generating security tests from the attack nets, and executing the tests against the SUT. The tool for attack net-based testing technique can generate security test code automatically. The test code can then be executed against the mutants. In the attack tree technique, however, only the attack paths can be generated automatically. The attack nets built for FileZilla Server yield a total of 60 unique security tests. These tests have killed 96.67% of the mutants. They did not kill the mutant with memory leak vulnerability because its effects were not captured by the attack nets.

The above experiments indicate that the two testing approaches have similar vulnerability detection capability – both depending on threat models. They cannot kill a mutant unless the threat model can capture the effect of the injected vulnerability in the mutant. The main differences are that attack nets are more expressive than attack trees and test generation from attack nets can be automated because attack nets are a formal method.

6. CONCLUSIONS

We have presented the mutation analysis of the FileZilla Server. The mutants with injected vulnerabilities have made it possible to compare two security testing methods that use threat models for test generation. The results show that they have similar fault-detection capabilities and cannot reveal the vulnerabilities whose effects are not captured by the threat models. Although this work focused on mutation analysis of the FileZillla Server, the mutation process can be applied to other systems.

So far, we have created 30 distinct mutants, covering all common FTP services and different vulnerabilities leading to various STRIDE attacks. The mutants and the demonstration tests are available to interested researchers. Although it is difficult, if not impossible, to create an exhaustive set of security mutants, the existing mutants are far from complete. We plan to continue to develop more mutants by introducing more programming errors that might lead to security vulnerabilities. We will also examine the existing vulnerability reports on other FTP server implementations and determine whether the vulnerabilities are applicable to FileZilla Server. This will make FileZilla Server a better benchmark for the research community to evaluate the vulnerability detection capability of security testing techniques.

In our current approach, the security mutants of FileZilla Server were created manually. An advantage is that they are more realistic than those created through syntactic changes. However, it is very time-consuming because it requires an in-depth understanding about the system functionality, security requirements, and source code. In a few extreme cases, developing one mutant took several days because the vulnerability to be injected involves complex interaction between several components and it is not just changes of the existing code, but also creation of new code. Future work will investigate how to automate the process of creating certain types of vulnerabilities. For example, vulnerabilities leading to denial of service attacks could be good candidate for automation, because some mutants of this kind allow for a program to stop working.

7. ACKNOWLEDGMENTS

This work was supported in part by NSF under grants CNS 0855106 and CNS 1004843.

8. REFERENCES

[1] Andrews, J. H., Briand, L. C. and Labiche, Y. Is mutation an appropriate tool for testing experiments? *Proc. of the 27th International Conference on Software Engineering (ICSE '05)*, 2005, pp. 402–411.

[2] Fonseca, J., Vieira, M., and Madeira, H. Testing and comparing Web vulnerability scanning tools for SQL injection and XSS attacks, *Proc. of the 13th Pacific Rim International Symposium on Dependable Computing*, Melbourne, Australia, December 2007, pp. 365-372.

[3] Hoglund, G. and McGraw, G. *Exploiting Software: How to Break Code*. Addison-Wesley, 2004.

[4] Jia, Y. and Harman, M. An analysis and survey of the development of mutation testing, *IEEE Transactions on Software Engineering*, June 2010.

[5] Le Traon, Y., Mouelhi, T. and Baudry ,B.: Testing security policies: Going beyond function testing. *Proc. of ISSRE'07*, pp. 93-102, 2007

[6] Marback, A., Do, H., He, K., Kondamarri, S., Xu, D. Security test generation using threat trees, *Proc. of AST'09*, Vancouver, Canada, May 2009.

[7] Martin, E. and Xie, T. A fault model and mutation testing of access control policies. *Proc. of the 16th international Conference on World Wide Web (WWW'07)*, pp. 667-676, May 2007.

[8] Martin, E. and Xie, T. Automated test generation for access control policies via change-impact analysis. *The 3rd International Workshop on Software Engineering for Secure Systems*, May 2007.

[9] Mouelhi, T., Fleurey, F., Baudry, B., and Le Traon, Y. A model-based framework for security policies specification, deployment and testing. *Proc. of the 11th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* 2008.

[10] Mouelhi, T., Le Traon, Y. and Baudry, B. Mutation analysis for security tests qualification, in *Mutation'07: Third Workshop on Mutation Analysis*, in conjunction with TAIC-Part. 2007.

[11] Potter, B., Allen, B. and Mcgraw, G. Software security testing. *IEEE Security & Privacy*, pp. 32-36, Sept. 2004.

[12] Pretschner, A., Le Traon, Y. and Mouelhi, T. Model-based tests for access control policies. *Proc. of the First International Conference on Software Testing Verification and Validation (ICST'08)*. Lillehamer, Norway, April 2008.

[13] Schneier, B. Attack trees. *Dr. Dobb's Journal of Software Tools* 24, 12, 1999, pp. 21-29

[14] Schuler, D. and Zeller, A. (Un-)Covering equivalent mutants, *Proc. of the Third International Conf. on Software Testing, Verification and Validation (ICST'2010)*, pp: 45-54.

[15] Shahriar, H. and Zulkernine, M. MUTEC: Mutation-based testing of cross site scripting, *Proc. of SESS'09*, pp. 47-53.

[16] Shahriar, H. and Zulkernine, M. MUSIC: Mutation-based SQL injection vulnerability checking. *Proc. of QSIC '08*. pp. 77-86. Oxford, Aug. 2008.

[17] Swiderski, F. and Snyder, W. *Threat Modeling*. Microsoft Press, 2004

[18] Thompson, H.H., Why security testing is hard? *IEEE Security & Privacy*, 2003. 1(4): pp. 83-86.

[19] Tuya, J., Suárez-Cabal, M., and Riva, C. Mutating database queries, *Information and Software Technology*, 49(4) 398-417, April 2007.

[20] Wang, L., Wong, W. and Xu, D. A threat model driven approach for security testing. *Proc. of SESS'07*, May 2007.

[21] Xu, D. and Nygard, K.E. Threat-driven modeling and verification of secure software using aspect-oriented Petri nets, *IEEE Trans. on Software Engineering*. Vol. 32, No. 4, pp. 265-278, April 2006.

BIOGRAPHY

Daniel Woodraska is currently a senior undergraduate student majoring in Computer Science at Dakota State University (DSU), USA. This paper was based on his research project conducted during the Summer 2010 REU (Research Experiences for Undergraduate) program in Information Assurance and Security sponsored by the National Science Foundation of the USA. Michael Sanford is a doctoral student of the Information Systems program at DSU. His research interests include information assurance and security. Dr. Dianxiang Xu is an associate professor and senior researcher of the National Center for the Protection of the Financial Infrastructure at DSU. His research interests include software security, software testing and verification, applied formal methods, and computer forensics. He received the B.S., M.S., and Ph.D. degrees in computer science from Nanjing University, China. Before joining DSU in May 2009, Dr. Xu was an assistant professor of computer science at North Dakota State University. He was a research assistant professor and engineer of computer science at Texas A&M University from 2000 to 2003, and research associate at Florida International University from 1999 to 2000. He is a senior member of the IEEE.