A Model-based Approach to the Security Testing of Network Protocol Implementations

William H. Allen, Chin Dou and Gerald A. Marin
Department of Computer Sciences
Florida Institute of Technology, Melbourne FL
wallen@cs.fit.edu cdou@fit.edu gmarin@cs.fit.edu

Abstract

Software is inherently buggy and those defects can lead to security breaches in applications. For more than a decade, buffer overflows have been the most common bugs found "in the wild" and they often lead to critical security issues. Several techniques have been developed to defend against these types of security flaws, all with different rates of success. In this paper, we present a systematic approach for the automated testing of network protocol server implementations. The technique is based on established black-box testing methods (such as finite-state model-based testing and fault-injection) enhanced by the generation of intelligent, semantic-aware test cases that provide a more complete coverage of the code space. We also demonstrate the use of a model-based testing tool that can reliably detect vulnerabilities in server applications.

1. Introduction

Network-aware software applications have a significant role in today's society. However, software security is in a critical state. The CERT Coordination Center [4] reported 5,990 known software vulnerabilities in 2005 alone, an increase of 63% from 2004. While significant efforts are being made to increase the overall security of software [10][15], clearly a large percentage of the applications released to the market are subsequently found to contain critical errors. These errors can be introduced at every stage of software development and many of them are not detected during the testing process. Greater attention must be given to both the testing and development phases in order to produce software that is reliable and secure.

The NIST National Vulnerability Database [12] reports that 66% of the software vulnerabilities detected in 2005 were caused by input validation errors (which includes unchecked buffer overflows and boundary conditions). Buffer overflow vulnerabilities arise from erroneous software implementation and can allow attackers, under certain conditions, to take complete control of vulnerable systems. Although the causes of buffer overflow vulnerabilities are well-known, developers have not yet been sufficiently educated to avoid all situations where they can occur and, currently, the best solution to this problem is thorough testing of the interfaces where vulnerabilities can be exploited. Assessing a buffer overflow vulnerability in a given piece of software involves two major steps: finding the actual error and determining whether or not it is exploitable. The complexity of modern software also means that automated testing procedures are necessary in order to provide a thorough test of the input state space.

The developers of new server implementations for well-known network application protocols (e.g. FTP, HTTP, SMTP) face many difficult issues, not the least of which is the need to test their code thoroughly before deployment. Once distributed, these new server implementations must provide stable, robust performance when communicating with client applications that were developed by other vendors. Such communication is typically based on protocols described in documents such as Internet RFCs [7].

In this paper, we present a systematic approach for the automated testing of network protocol server implementations that is based on established black-box testing techniques such as model-based testing and fault-injection, enhanced by the generation of intelligent, semantic-aware test cases that provide a more complete coverage of the code space. We refer to the approach presented here as Client/Server Behavioral Modeling (CSB Modeling). The protocol to be tested is modeled by creating a finite-state machine that describes valid communications between the client and the server. This model captures the semantics of the client/server interactions and, therefore, represents

the behaviors of a valid implementation of the protocol. We then use graph traversal algorithms to generate templates that we call *test patterns*. We also model the syntax of messages exchanged between client and server so that we can create "intelligent" test data. The test patterns are merged with the test data to generate a large set of executable test cases.

This paper is organized as follows: In Section 2, we briefly discuss common testing techniques and the importance of automation in the software testing process. In Section 3, we provide an overview of related work. Section 4 describes the design of our proposed approach. In Section 5, we demonstrate our automated testing technique by using it in a practical software testing example. In Section 6, we provide an objective evaluation of the proposed approach and demonstration tool.

2. Software Testing Techniques

Software testing offers an opportunity to detect implementation errors before software products are made available to the public. Software testing techniques can be broadly classified by the way in which the tester interacts with the application under test (AUT). White-box testing assumes that the tester has access to the application's source code and therefore has significant knowledge of the internals of the AUT. The most common white-box technique, static source code analysis, examines source code for constructs that could be the cause of errors. Grey-box testing is performed without access to the source code of the application being tested. To compensate for the lack of source code, the tester attempts to extract information about the application from the executable code by disassembling the binary files or even reverse engineering the application. The black-box testing approach is also conducted without any prior knowledge of the inner workings of the AUT, but focuses on triggering software failures by injecting test data through public interfaces. Done well, it tests the application in a realistic environment and allows the tester to explore the impact of combinations of conditions [17].

Once potentially vulnerable code is detected, one question arises. Is the vulnerability exploitable in a real-world situation? To be exploitable, user input must drive the application to execute the vulnerable code. Static source code analysis sometimes detects bugs that are not exploitable in real-world situations, but that determination requires time-consuming verification. However, when vulnerabilities are found by black-box testing, they are more likely to be exploitable since they were detected via a public interface.

Automation is crucial to thorough black-box software testing and can be employed in two different (although not mutually-exclusive) ways: automated test case generation and automated test execution. Our approach makes use of both techniques and reduces the number of test cases by generating only meaningful (i.e., intelligent) test cases rather than those based on random test data.

2.1 Fault Injection

There are many documented approaches for performing black-box testing; we will focus on one specific technique, fault injection [16], which is the practice of testing an application by providing it with a range of erroneous inputs while monitoring the response to those inputs. Successful fault injection depends on the effectiveness of the test cases created by the tester. By effectiveness, we refer to the test cases' ability to trigger a vulnerability. One challenge that fault injection faces is to bypass input sanitization [2][9]. Software error-handling is a layered process and input data are usually checked against several criteria before being accepted by the application. Useful test cases should pass through these preliminary checks so that the application can be tested more deeply. However, without direct knowledge of how "deep" within the application software errors might exist, the creation of test cases that can reach the deepest code sequences is a difficult task. Fortunately, one advantage for testers of network protocols is that the protocol specifications are well-known and the expected behavior of the AUT can be deduced from those specifications.

One technique used to performing fault injection on an application with the intent of finding buffer overflow vulnerabilities is referred as to *fuzzing* [11][13]. Network protocol fuzzing usually involves writing client code (using a scripting language, such as Perl or Python) which transmits fuzzed data to the server application. Fuzz data can be generated manually or programmatically and is generally created in one of two ways, by *data generation* or by *data mutation* [13]. Data generation produces fuzz data from specifications or descriptions of valid data. Data mutation starts with valid data and modifies it through bit-flipping or character substitution. Although these basic fuzzing techniques can directly uncover some buffer overflow vulnerabilities, the complex nature of most network protocols limits their usefulness as standalone tools.

Given the protocol specification, the most thorough testing approach would be to create test cases for the entire state space of the protocol. However, for all but the simplest of protocols, the number of states to be tested, multiplied by the range of possible valid and invalid input values, is too large for this approach to be feasible. Therefore a useful testing strategy must include a way to restrict the number of test cases without omitting those that are likely to trigger a vulnerability.

2.2 Model-based Testing

Model-based black-box testing uses behavioral models of the application under test to automate the generation of test cases. A finite state machine is used to model the application's behavior. Nodes represent the protocol's states and edges show the state transitions that can be triggered by user input. This technique has many advantages: it provides a formal way to describe the application's behavior, it provides a way to measure the extent of testing, and it allows testers to adapt rapidly to interface changes [3][5]. Yet, its most useful contribution is the ability to automate the generation of test cases. By applying a carefully selected graph traversal algorithm to the finite state model, testers can generate a sequence of state transitions that represent a valid interaction between client and server [6]. Each of these sequences can form a template, or test pattern, for test cases that will follow the set of state transitions that the sequence describes. When we combine a sequence of transitions with a set of input values that correctly trigger those transitions, we have a test case that can be used to replicate a valid client/server interaction. By replacing each of the valid input values with carefully crafted test data, we can determine whether the AUT will reject that erroneous input or if that input will exploit a vulnerability. Thus, repeated application of a graph traversal algorithm to a model will produce a set of test patterns that can be used to generate a large number of executable test cases automatically.

A useful automated test environment should provide both automated test generation and a test harness that controls the execution of test cases by launching (or resetting) the application under test, executing the test case scripts, capturing and comparing the outcome of each test case against an oracle and logging the results. An oracle is a mechanism for determining whether the AUT operated correctly by comparing the expected behavior with the measured results. For example, input buffer overflows can cause an application to fail because return values in the stack are overwritten. In general, a debugger attached to the AUT can easily determine if this has occurred.

2.3 Block-based Protocol Analysis

One technique that seeks to improve the efficiency of fault injection is block-based protocol analysis [2]. Recognizing that client/server communications involve the exchange of messages that are composed of a number of fields or variables, each with their own data type, testers identify a set of blocks within a message that delineate the values in those fields and label each of those blocks with the type of data that it contains. When test data is generated for a message, the data for each block within the message can be restricted to the range of values that are valid for its data type. For example, if a particular block contains a text string, the test data generated for that block will be limited to valid text strings and not arbitrary binary values. This restriction both increases the likelihood of avoiding input sanitization (which could reject non-text input) and reduces the number of test cases that will be generated.

3. Related Work

The first discussion of fuzzing as a software testing technique appeared in a study of the reliability of UNIX utilities when presented with invalid data [11]. The authors found several potential security vulnerabilities as well as cases where software failed because of buffer overflows.

The most widely-known testing tool that employs block-based protocol analysis is SPIKE [2], which has been used successfully to detect a number of vulnerabilities in network protocol implementations. While SPIKE does produce more "intelligent" test cases than tools which randomly generate test data, it does not make use of protocol models to generate behavior-based test cases.

The PROTOS project [8], uses attribute grammars to model selected parts of a protocol for focused testing. Although the PROTOS system is flexible and can be used to model a wide range of protocols, model development is rather complex and requires considerably more manual configuration and programming than the approach described in this paper.

An interesting approach to producing new implementations of network protocols from XML models of the protocols [1] could possibly be adapted to generate test patterns that can be used to detect vulnerabilities in the generated code.

4. A Model-based Approach to Network Protocol Testing

As discussed in Section 2, black-box testing of network protocol implementations with the goal of discovering buffer overflow vulnerabilities relies heavily on fault-injection. In order to perform effective fault injection, test data must be generated in such a way that it will escape error-checking mechanisms but still reach its intended target, either causing the application to fail or overwriting existing data with malicious code. Black-box testers generally have little information about the internal structure of the application under test. However, in the case of network protocols, developers must conform to published specifications and meet specific requirements for data formats and client/server interactions. Testers can gain significant insight into an application's behavior by carefully analyzing protocol specifications. This knowledge can be used to identify potential vulnerabilities in the application by determining when and where input data must be processed. Both the semantics and the syntax of messages exchanged between hosts are described in the protocol specifications and that information can be used to refine a model of their interactions and to restrict the generation of fuzz data to those variations that are most likely to exploit errors in the application's code.

4.1 Overview

In this section, we describe the Client/Server Behavioral Modeling (CSB Modeling) approach to testing network protocol implementations. This approach combines all of the testing techniques discussed in Section 2: model-based testing, fault-injection, fuzzing, block-based syntax analysis and test automation. The goal of this work is to perform exhaustive testing of network server implementations by employing automation at all possible stages of the process and by reaching every feasible area of the protocol's code space. While this paper describes testing for buffer overflows, this approach can be used to discover a wide range of vulnerabilities and errors.

4.2 Modeling Protocol Semantics and Syntax

Protocol modeling begins with a state machine that represents the protocol's behavior during communications between the client and the server. This state-based model provides an accurate representation of the protocol's semantics. A graph traversal algorithm is used to generate paths which serve as templates for test cases. These templates, which we call test patterns, provide a set of valid communication sequences which can then be used to produce test cases by fuzzing the input data carried in selected messages.

We use block-based protocol analysis (see Section 2.3) to model each message type specified by the protocol. Each message is broken down into blocks that represent the different data variables specified in the protocol. Blocks are then assigned a block type which is used to specify the ways in which data should be fuzzed. These block-based models describe the syntax of messages with sufficient detail to produce test data which is both likely to pass input error checking and to reveal unchecked buffer overflow vulnerabilities. The example below shows how a message is broken into blocks and how block-based analysis is used to create intelligent fuzz data.

Assume that the following message will be used to create fuzz data to test a web server:

```
GET http://www.ieee.org/ HTTP/1.0\r\n
```

The message is broken into blocks, where each block has a type:

Type:	HTTP_REQ	SP	URL
Block:	`GET`		`http://www.ieee.org/`

Type:	SP	HTTP_VER	CRLF
Block:		`HTTP/1.0`	`\r\n`

Fuzzing can be controlled on a variable-by-variable basis, creating only those modifications that are likely to pass input error checking. For example, any data in the HTTP_REQUEST field that did not match a valid HTTP request string would not be accepted by a server implementation with even the most rudimentary error checking. Likewise, fuzzing the HTTP version field is unlikely to be successful. However, it is unfeasible to check all possible values for the URL field, so it is a good candidate for fuzzing by either injecting invalid characters or appending long strings of valid characters.

The generation of test patterns from the protocol model and the merging of those test patterns with test data to create test cases are both automated, but several steps in the process still require manual input or editing. As yet, the protocol models must be constructed manually, although there has been some work on the automated creation of protocol models [1] which may prove useful. The human tester must specify the test data to be merged into the test patterns but is aided in this task by knowledge of the data types

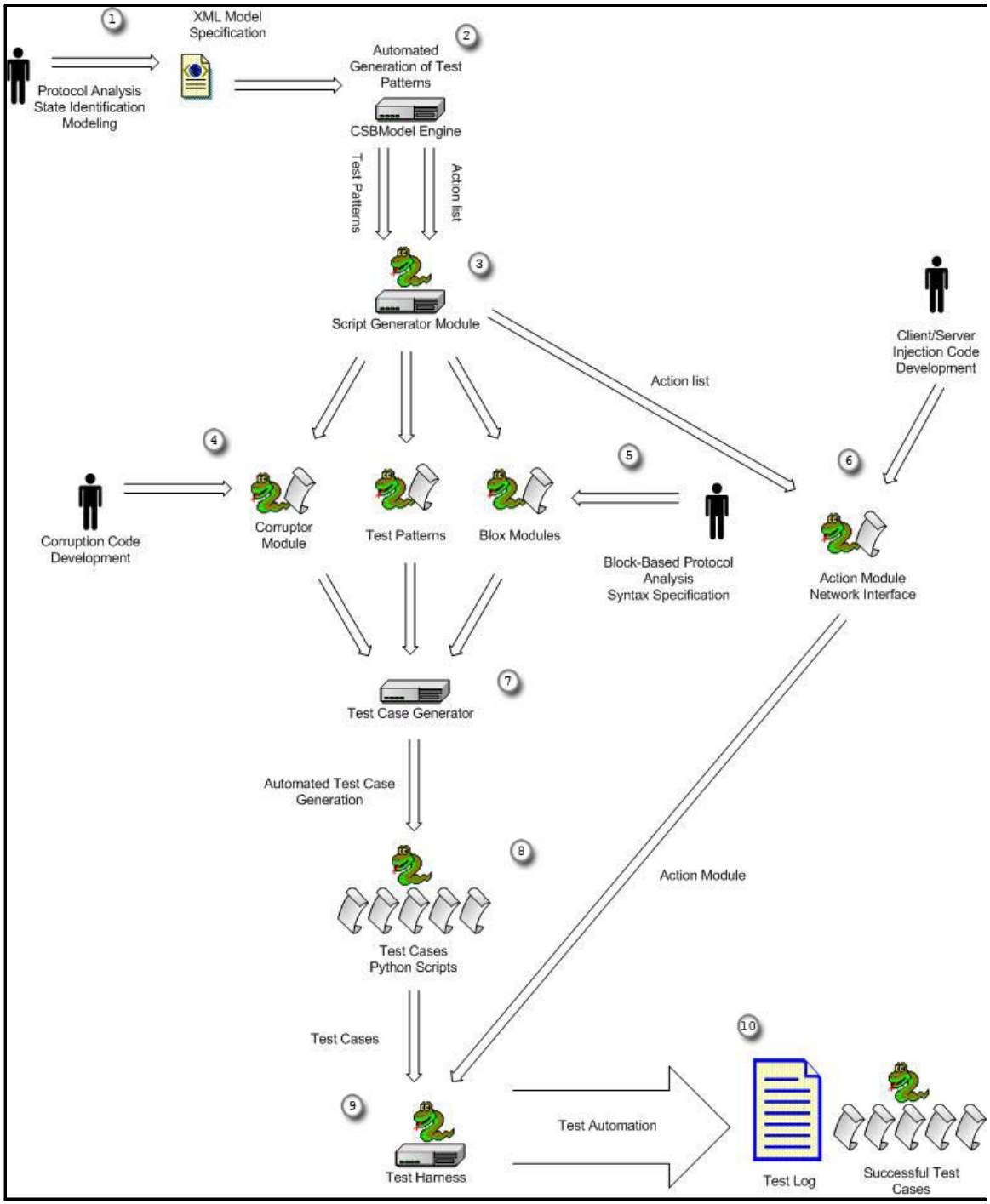

Figure 1. Overview of the Client/Server Behavioral Modeling System

expected in each block of data. The test patterns also contain metadata which places each block in its semantic context and that knowledge can guide the tester in crafting appropriate test data for each specific injection point. Once the test data is created, the test case generator will automatically merge the test data into the appropriate fields of the test patterns to create the set of executable test cases, which are implemented as Python scripts.

4.3 Architecture

Figure 1 provides an overview of the main components of the Client/Server Behavioral Modeling system. Each of steps in the diagram is explained further below. The "human" figures in the diagram represent steps that require manual input at this time, although we believe that at least some of those processes can be automated. The "python" figures represent Python scripts, some of which are generated by the system.

1. The Tester models the protocol semantics as a finite state automata and creates the *XML Model Specification* file.
2. The *CSBModel* Engine applies the appropriate graph traversal algorithm to produce a set of *test patterns* from the model. An *action list* is also issued from the *CSBModel* Engine that includes all client and server actions specified in the model.
3. From the information produced by the *CSBModel Engine*, the *Script Generator* creates a set of template scripts which the Tester will customize for the specific goals of the testing session. The *Corruptor* module scripts will describe how data corruption (fuzzing) should be produced. The *Blox* module templates are used for protocol syntax modeling, one for each *Client Action*. An *Action* module script specifies how network fault injection is handled.
4. The Tester edits the template *Corruptor* module script. Code is written to specify how each protocol block type should be fuzzed.
5. The Tester edits the *Blox* module scripts. For each *Client Action* specified by the model, a template *Blox* module has been created by the *Script Generator*. The Tester modifies the *Blox* objects according to the decisions made during syntax modeling.
6. The Tester edits the *Action* module template script which will be used to control network injection during execution of the test cases.
7. Based on the *Test Patterns* generated from the model and the edited *Blox* and *Corruptor* modules, the *Test Case Generator* can produce thousands of executable test cases.
8. The test cases and *Action* module are passed to the *Test Harness*.
9. The *Test Harness* imports the *Action* module written by the Tester and executes the test cases while monitoring the application under test and logging any successful test cases.
10. The output of this process is sent to a log file and includes a list of the successful test cases

5. Experimental Results

To demonstrate the effectiveness of the CSB Modeling approach, we tested five different FTP server implementations (see Table 1 below). The FTP protocol is described in detail in RFC 959 [14]. After modeling the FTP protocol's syntax and semantics, we used a simple shortest-path graph traversal algorithm to generate 39 test patterns, each representing a valid sequence of client/server interactions. From this set of templates, the system automatically generated more than 14,000 individual test cases, which were each applied to all five servers.

The five FTP server implementations listed in Table 1 were tested using the FTP model described above. Although each of the five servers had known vulnerabilities at the time of our experiments, we did not create specific test cases for those vulnerabilities. Instead, we performed blind testing on the servers by generating test cases only from the FTP model. Thus, each vulnerability discovered was the result of exhaustive testing. After testing, we did validate the results by comparing them with descriptions of the vulnerabilities known to exist for each server. The second column of Table 1 lists the vulnerability detected for each server.

Vulnerabilities were detected in the first three servers during the first round of testing. After a careful examination of the test cases, we found that the specifications for the generation of fuzz data were too restrictive. After refining those specifications, we re-ran the experiments on the last two servers and discovered the vulnerabilities shown in Table 1. Note that the test cases used for all five servers were based on the same state-based protocol model.

Table 1. FTP server implementations tested for vulnerabilities

FTP Server under test	Vulnerability associated with:	Detected?
Microsoft® IIS 3.0	the `NLST` command	Yes
Microsoft® IIS 4.0	the `STAT` command	Yes
Rhinosoft® Serv-U 4.0	the `MDTM` command	Yes
Ipswitch® WS_FTP 4.0	the `APPE` command	Yes
South River Titan 3.21	the `CWD` command	Yes

6. Summary and Conclusions

We have presented a novel and systematic approach for finding buffer overflow vulnerabilities in network protocol implementations. This approach models protocol syntax and semantics and creates "intelligent" test cases using block-based protocol analysis and data fuzzing.

A prototype tool called the CSB Modeling system was developed to demonstrate the approach and it successfully discovered buffer overflow vulnerabilities in five different FTP server implementations. An analysis of these results determined that the success of this approach relied on three important factors:

- the ability to model the protocol accurately.
- the ability to generate the proper set of test patterns.
- the ability to cover a wide range of vulnerabilities with the generated test data.

The modeling of the FTP protocol was simplified by the availability of specification documents such as RFC's. This approach may prove to be less effective if applied to proprietary protocols. Fortunately, the specifications for the most common application level network protocols, such as HTTP, SMTP and FTP, are publicly available through the Internet RFC documents [7].

History has shown that every new network protocol (no matter how well designed) has been accompanied by vulnerabilities in at least some of its implementations. New network protocols will continue to emerge and the security testing community needs a more systematic and efficient way to assess the security of those protocol implementations. Only by thoroughly inspecting these implementations can we avoid the security issues that have occurred in the past. Current tools and methods do not provide a reliable means to achieve that goal. The approach we presented in this paper attempts to fill gaps left by those techniques. We believe that further development and testing of the Client/Server Behavior Modeling system will produce a mature and useful tool for exhaustively testing new protocol implementations to improve their security.

Future work will include modeling more complex network protocols, automating segments of the modeling process and testing newer server applications with the goal of discovering previously unknown vulnerabilities.

7. References

[1] Abdullah, I. S. and D. A. Menasche, "Protocol Specification and Automatic Implementation Using XML and CBSE", Proceedings of the International Conference on Communications, Internet and Information Technology, 2003

[2] Aitel, Dave, "The Advantages of Block-Based Protocol Analysis for Security Testing", February 4th, 2004. `http://www.immunitysec.com`

[3] Beizer, Boris, "Black-box testing : techniques for functional testing of software and systems", Wiley, 1995.

[4] CERT Coordination Center, `www.cert.org`, 2006

[5] El-Far, Ibrahim K. and James A. Whittaker, "Model-based Software Testing", Encyclopedia of Software Engineering, editor: J.J. Marciniak, Wiley Publishing, 2001

[6] Gross, Jonathan, "Graph Theory and Its Applications", CRC Press, 1998

[7] Internet RFCs, `http://www.rfc-editor.org`, 2006

[8] Rauli Kaksonen. "A Functional Method for Assessing Protocol Implementation Security", Licentiate Thesis, University of Oulu, Finland, 2001

[9] Koziol, Jack, et al., "The Shellcoder's Handbook", Wiley, 2004

[10] Microsoft Security Developer Center, `http://msdn.microsoft.com/security/`, 2006

[11] Miller, Barton P., Louis Fredriksen and Bryan So, "An empirical study of the reliability of UNIX utilities", Communications of the ACM, Volume 33, Issue 12, 1990

[12] NIST National Vulnerability Database, `http://nvd.nist.gov`, 2006

[13] Peter Oehlert, "Violating Assumptions with Fuzzing", IEEE Security & Privacy, March/April 2005

[14] Postel, John, et al. "File Transfer Protocol". RFC 959, October 1985.

[15] Software Engineering Institute, "The DHS 'Build Security In' Secure Coding Initiative", `https://BuildSecurityIn.us-cert.gov`, 2006

[16] Voas, Jeffrey M., and Gary McGraw, "Software Fault Injection", Wiley, 1998

[17] Whittaker, James A., "How to Break Software", Pearson Education, 2003.